Flexible fixed-point function generation for FPGAs

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

Bogdan Pasca
Intel PSG
France

***Abstract*—Efficient fixed-point function implementation is critical in many FPGA application domains including convolutional neural networks, computer vision, and communication systems. In this work we focus on functions of the form x^p, with $p \in \{-1, -1/2, 1/2\}$ as part of a function generator targeting FPGAs. The generator implements architectures based on new but also existing algorithms. In this work we present three distinct methods implemented in this generator that outperform state-of-the-art implementations for certain configurations. Traditionally, fixed-point function implementation requires a normalization stage, compute and denormalization (reconstruction) of the result. The first proposed method implements the function holistically, thus saving the logic and latency required during the normalize and reconstruct stages. The second proposed method is based on a novel second order Taylor implementation. The third method is based on the cubic convergence of Halley's method, which is novel in this context. The proposed methods are compared and contrasted against state-of-the art implementations in the context of FPGA targets.**

I. Introduction

With increased capacities, compute-oriented embedded features such as DSP blocks and large on-chip memory blocks, FPGAs provide a great platform for accelerating applications. This family of devices are very efficient at implementing compute datapaths when using customized fixed-point arithmetic. Efficient implementations of fixed-point functions are critical for many application domains including convolutional networks [1], computer vision [2], MIMO communication-systems with matrix-decomposition [3], [4], scientific-computing at CERN [5], image-reconstruction in bioinformatics [6] and others. Since application requirements (data format, precision, target, frequency) are often very diverse, a high-level generator (Figure 1) is the best way of providing state-of-the art fixed-point function implementations.

In this paper we focus on the implementation of a family of fixed-point functions of the form x^p with $p \in \{-1, -1/2, 1/2\}$. We present three distinct methods implemented in the function generator, that outperform state-of-the-art implementations for certain configurations.

The first proposed method improves the latency and logic resources. It works in general for functions of the form 2^p, $p < 0$; in this work the reciprocal and reciprocal square root fit this profile. The savings of this method are thanks to the holistic implementation of the function, given the user specification, as opposed to a classical range-reduction/compute/reconstruct implementation. An example of such a specification is: unsigned reciprocal on 16 bits, with 8 integer and 8 fractional bits. This method is described in Section IV.

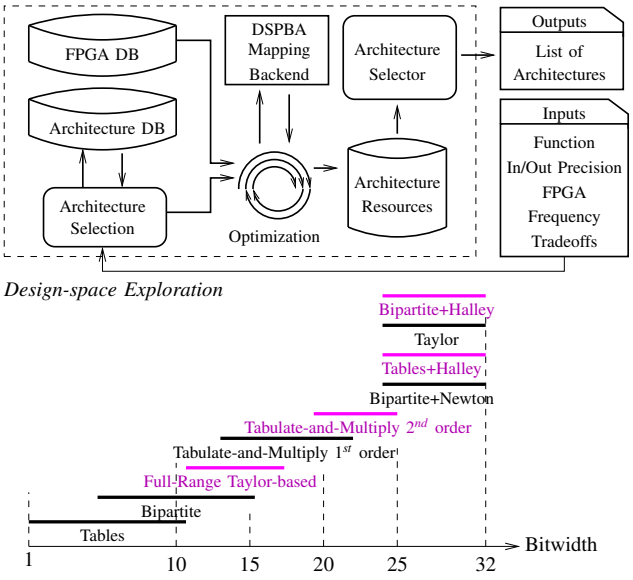


Fig. 1. High-level view of the fixed-point function generator and estimated input bit-width to implementation method mapping

The second proposed method is an extension to second order of the tabulate-and-multiply method from [7]. The method outperforms state-of-the-art minimax-approximation-based techniques by having the lowest memory requirement and lowest logic requirements while using a similar numbers of multipliers. The resource savings are thanks to the particular Taylor coefficients that allow for computations using bit-manipulations. The method is described in Section V-B.

The third proposed method is based on a novel use of Halley's cubic convergence algorithm. The method uses a particular form of the recurrence that allows computing functions such as the reciprocal and reciprocal-square-root efficiently in a hardware context. For wider bitwidths Halley's method is used together with the bipartite-table method, which returns the initial approximation. In this configuration the novel implementation clearly outperforms state-of-the art implementations. This method is described in Section VI-B.

The presented three methods, alongside other numerous classical implementation algorithms, presented on the bottom of Figure 1, are part of our fixed-point function generator. The generator is part of the DSP Builder Advanced (DSPBA) [8] backend, and allows for fully parametrized descriptions in terms of precision, FPGA target, and target frequency.

II. FPGA FEATURES

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 Intel FPGA 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 that share inputs. Logic resources are usually reported in terms of ALMs.

Contemporary FPGA devices also contain thousands of small flexible multipliers (packaged in DSP blocks) and memory blocks. In this paper we focus on the low-cost CycloneV [9] devices for which the DSP blocks can be configured in various modes: 2 18x19, 27x27 etc. Results either report multiplier usage in terms of 18x18 equivalents, or by stating the number of DSP blocks. Dedicated memory blocks in CycloneV devices have a capacity of 10Kb each (M10K). The relevant configurations for this works are: 2048x5bits, 1024x10bits, 512x20bits, 256x40bits.

III. BACKGROUND AND RELATED WORK

Methods initially designed for floating-point (FP) implementations require the input to the fixed-point kernel to be normalized to $[1,2)$. The steps for implementing a fixed-point function using these methods include a pre-scaling stage (get to $[1,2)$), followed by a computation on the restricted interval, and the reconstruction of the original function (post-scaling, function-specific). Arbitrary format fixed-point functions can be implemented in this way. Alternatively, if the function is in fixed-point we can use algorithms that would otherwise be insufficiently accurate for the fixed-point kernel of a floating-point core. The use of these algorithms depends on the function, the in/out data formats and the user's input ranges.

The classical range reduction for fixed-point kernels consists in performing the substitution $x = 2^e x_{\text{norm}}$ where x is the input, e may be considered its exponent and $x_{\text{norm}} \in [1,2)$ its normalized mantissa. With a mantissa in $[1,2)$ classical algorithms can be employed. For the fixed-point functions targeted by this work, the reconstruction phase involves:

$$y_{\text{inverse}} = 2^{-e}\frac{1}{x_{\text{norm}}}$$

$$y_{\text{recipSqrt}} = \begin{cases} 2^{-e/2}\frac{1}{\sqrt{x_{\text{norm}}}} & \text{if } e \text{ even} \\ 2^{(1-e)/2}1/\sqrt{2}\frac{1}{\sqrt{x_{\text{norm}}}} & \text{if } e \text{ odd} \end{cases}$$

$$y_{\text{sqrt}} = \begin{cases} 2^{e/2}\sqrt{x_{\text{norm}}} & \text{if } e \text{ even} \\ 2^{(e-1)/2}\sqrt{2}\sqrt{x_{\text{norm}}} & \text{if } e \text{ odd} \end{cases}$$

A number of strategies can be employed for implementing the fixed-point kernels with normalized inputs.

Table-based methods fit well contemporary FPGAs thanks to the various granularity look-up tables and block memories. Direct tabulation is straightforward and suitable for bitwidths up to 10-12 bits. Results can be stored correctly rounded and underflow/overflow logic is integrated. Above the 10-12-bit range, direct tabulation increases memory requirements exponentially. Therefore, several methods try to overcome this limitation: ATA method [10], iATA [11], the bipartite method [12], SBTM [13], STAM [14] and the generalized multipartite method [15], [16], [17].

With the increase of precision, table-based methods become impractical. *Polynomial approximation-based methods* scale better. The polynomial itself can be obtained by either using truncated *Taylor series* which approximates the function, or by using a *generic polynomial approximation* – for example a minimax polynomial. Some works building on the first approach are [7], [18] for smaller bitwidths (discussed in Section V-A) and [19] for larger bitwidths.

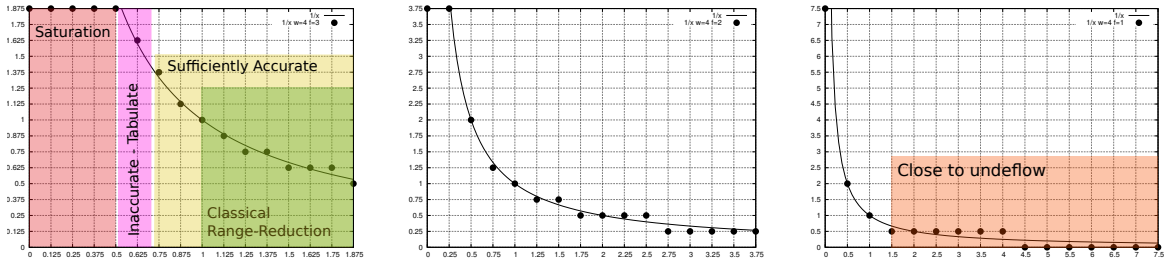
Iterative methods can also be used for higher precisions. The *Newton-Raphson method* (Section VI-A) doubles the accuracy of the initial approximation at each iteration. *Halley's method* triples the precision at each iteration. The generalized order n version (output precision after 1 iteration is n times higher than the input precision) is known as *Householder method* [20]. Iterative methods are covered in Section VI-B.

IV. HOLISTIC FULL-RANGE IMPLEMENTATION

One contribution of this article are architectures targeting bit-widths of up to 16-bits, that don't require range-reduction and reconstruction stages. The computational core of these architectures are based on 1st degree Taylor polynomials. The key observation is that the approximation polynomials are used on wider-than-typical input ranges. The input ranges where the approximation is insufficiently accurate are handled using tabulation-based techniques.

Figure 2 highlights three instances of the reciprocal function for the same input/output bit-width (4 bits), but for 3 different fixed-point formats: width (w) w=4, fraction (f) f=3, w=4 f=2 and w=4, f=1. The 3 formats correspond to different input/output ranges: [0,1.875], [0,3.75] and [0,7.5] respectively. The behavior of the function is also different: in Fig.2(a) a significant input range leads to saturation, while in Fig.2(c) many inputs come close to underflow.

Using a classical technique (range reduction and reconstruction), the computing kernel operates on a function with a normalized input $x \in [1,2)$. In case of the reciprocal the normalized function is depicted in green in Fig.2(a). As previously stated, many families of techniques exist for computing the function in green. A piecewise-polynomial-based architecture divides the input interval into a number of subintervals (2^k), and approximates the function on each subinterval using a polynomial (degree d). The parametrization (k,d) produces the required accuracy on the normalized input range, but can be sufficiently accurate on a wider range (example for the reciprocal is highlighted in yellow in Fig.2(a)). There will be ranges where the piecewise polynomial approximation is not sufficiently accurate (where the first derivative is large for instance). These regions will be handled by tabulation (example highlighted in pink in Fig.2(a)). Some functions will



(a) w=4 f=3 (b) w=4 f=2 (c) w=4 f=1

Fig. 2. Reciprocal function for 3 input/output formats

have regions where the output saturates to some value (example highlighted in red Fig.2(a)), or are close to underflowing (example highlighted in orange in Fig.2(c)).

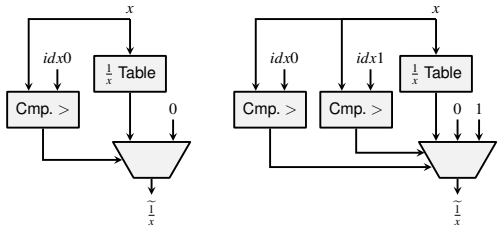
The proposed architectures use 1st degree piecewise-polynomial approximations using the minimax algorithm. Additionally, during generation intervals which violate the accuracy requirement are identified and hardware is built for these to be handled separately, either by simple or by composed tabulation (base + offset), in a similar way to [17]. There are a high number of variables which influence these decisions and hence these are implemented inside a function generator.

Typically, hardware will be built for detecting the saturation range, and for handling the inaccurate region (by tabulation). Since FPGA memory blocks are monolithic, it can prove more efficiently to sometime tabulate the entire saturate and inaccurate range.

The table output bit-width 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, similarly to what is done in [17]. Using a simple addition we 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.

For functions such as the reciprocal, the output of the architecture may underflow for a large number of inputs, depending on the input/output format. One example is the input/output format with 16-bits of precision (unsigned) and 4 bits of fraction. This is a case where a separate underflow architecture is more efficient to be used.

The underflow architecture will use tabulation for values that do not overflow or underflow. It has two implementation



(a) (b)

Fig. 3. Two architectures used for underflow handling

options: 1/ tabulation is used for all values that do not underflow (return 0) and 2/ tabulation is used for values that are different to 0, or to 1 ulp (this can easily be extended). 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 expensive. Figure 3 depicts the two architectures used for underflow-heavy formats. The results section presents the results of all these optimizations across an important number of input/output formats.

V. Tabulate-and-multiply

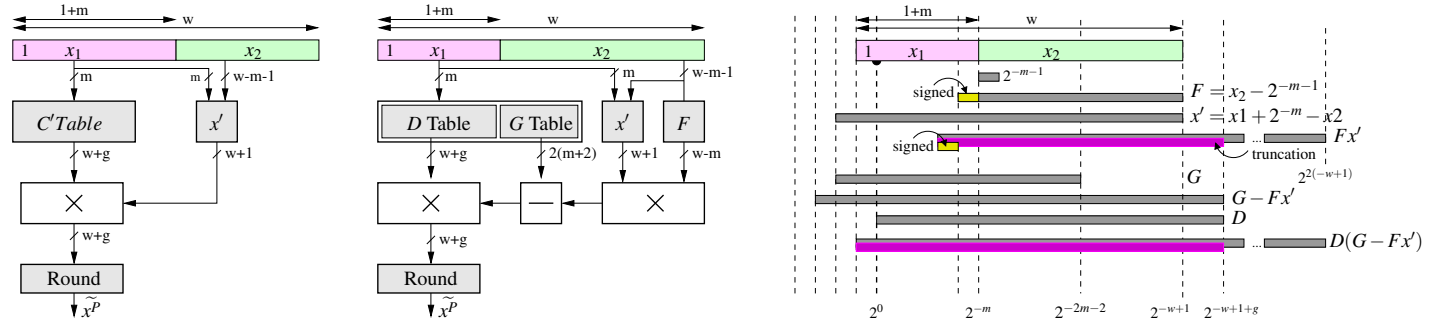
A. First Order

One of the shortcomings of the table-only methods is the exponential growth of the memory resources with the increase of the input precision. The method introduced in [7] is based on the Taylor series expansion; it reduces the memory requirements by using a linear polynomial approximation of x^P (for $P = \pm 2^k$, k integer).

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 approximated around the point $x = x_1 + 2^{-m-1}$, with the first two terms of the Taylor series. Factoring $(x_1 + 2^{-m-1})^{P-1}$ yields:

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

Keeping the notations of [7], we denote $C = (x_1 + 2^{-m-1})^{P-1}$ and $x' = x_1 + 2^{-m-1} + Px_2 - P2^{-m-1}$. Term C, can be obtained



(a) First Order Tabulate-and-Multiply Method Architecture

(b) Second Order Tabulate-and-Multiply Method Architecture (for the reciprocal function)

(c) Second Order Method: Operand Alignments (for the reciprocal function)

Fig. 4. First and Second-Order Table and Multiply Architectures

from a table addressed by x_1; x' can be obtained by using simple bit manipulations on x_2 and x_1. Thus, equation (1) is re-written so that it requires one multiplication:

$$x^P \approx Cx' \tag{2}$$

The error introduced by truncating the Taylor series is:

$$\varepsilon_m \approx P(P-1)(x_1+2^{-m-1})^{P-2}2^{-1}(x_2-2^{-m-1})^2 \tag{3}$$

Which means that $\varepsilon_m \approx 2^{-2m-3+\log|P(P-1)|}$. The method error ε_m can be slightly improved by replacing C in Eq. (2) by $C' = C + P(P-1)x_1^{P-3}2^{-2m-4}$ as shown in [7].

The architecture is depicted in Fig. 4(a). For our functions, x' consists mainly of inverting the bits of x_2 and concatenating them in a given order. The C' table requires $2^m(w+1)$ bits (w the input bitwidth and $m \approx w/2$). 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. The x' term requires $\approx w-m$ LUTs, and a w-bit addition for the final rounding.

B. Second Order

The use of only two terms of the Taylor series expansion in the first order tabulate-and-multiply method limits the accuracy which it can achieve. Take into consideration the bit-width range for which the first order tabulate-and-multiply method can be used, at the top of Figure 1. At the top of the range, for the larger precisions, the size of the C' table becomes a limiting factor. The *second order tabulate-and-multiply method* is an extension of the method in Section V-A, and tries to overcome this shortcoming.

The method makes use of the first three terms of the Taylor series. As in the case of the first order method, the input x is split into two parts x_1 (the top m MSBs) and x_2 (the remaining LSBs), with $1 \le x_1 < 2$ and $0 \le x_2 < 2^{-m}$. Therefore, an approximation around the point $x = x_1 + 2^{-m-1}$ results in:

$$\begin{aligned} x^P \approx & (x_1+2^{-m-1})^P + (x_2-2^{-m-1})(x_1+2^{-m-1})^{P-1}P \\ & + \frac{1}{2}(x_2-2^{-m-1})^2(x_1+2^{-m-1})^{P-2}P(P-1) \end{aligned} \tag{4}$$

If $(x_1+2^{-m-1})^{P-2}$ is factored out:

$$\begin{aligned} x^P \approx & (x_1+2^{-m-1})^{P-2}[(x_1+2^{-m-1})^2 \\ & + (x_2-2^{-m-1})(x_1+2^{-m-1})P + \frac{P(P-1)}{2}(x_2-2^{-m-1})^2] \end{aligned} \tag{5}$$

The terms of Equation 5 can be regrouped more conveniently:

$$\begin{aligned} x^P \approx & (x_1+2^{-m-1})^{P-2}[(x_1+2^{-m-1})^2 \\ & + P(x_2-2^{-m-1})(x_1+2^{-m-1} + \frac{P-1}{2}(x_2-2^{-m-1}))] \end{aligned} \tag{6}$$

We use the following notations:

$$\begin{aligned} D &= (x_1+2^{-m-1})^{P-2} \quad & x' &= x_1+2^{-m-1}+\frac{P-1}{2}(x_2-2^{-m-1}) \\ G &= (x_1+2^{-m-1})^2 \quad & F &= x_2-2^{-m-1} \end{aligned} \tag{7}$$

Therefore, Equation 6 can be evaluated as:

$$x^P \approx D[G+PFx'] \tag{8}$$

In terms of the implementation, the terms D and G can be read simultaneously from a table, indexed by x_1. The term F can be obtained through bit manipulations, by flipping the MSB of x_2 and storing the new sign of F, for later use. The term x' can also be obtained through bit manipulations. The actual operations depend on the value of P, but mainly consist of concatenations or negations, and possibly additions. A rectangular multiplication is additionally required, between F and x'. Both of the multiplications can be truncated to a smaller intermediary precision. An illustration of the architecture with data alignments for $1/x$ ($P=-1$) is presented in Fig.4(b) and Fig.4(c) respectively.

The method error ε_{method} inherent to the second order tabulate and multiply method is of the order of magnitude of the terms that were left out of the Taylor series expansion, and can be expressed as:

$$\varepsilon_{method} \approx P(P-1)(P-2)(x_1+2^{-m-1})^{P-3}\frac{1}{6}(x_2-2^{-m-1})^3 \tag{9}$$

Therefore, the method error can be bounded as $\overline{\varepsilon_{method}} \approx 2^{-3m-5+\log|P(P-1)(P-2)|}$ (with P taking the values -1, $\frac{1}{2}$ and $-\frac{1}{2}$ for the recip., sqrt and recip. sqrt functions, respectively). Compared to the first order method, which produces a result that is correct within approximately $2m$ bits, the second order method has an accuracy of approximately $3m$ bits.

The multiplications Fx' and the one between D and the term between square brackets can be truncated so as to save resources. A larger intermediary precision (g extra guard bits) is used in the datapath in order to absorb the rounding errors.

The error budget $\varepsilon_{total} < 2^{-w}$ (where w is the output precision) is classically divided as $\varepsilon_{total} = \varepsilon_{finalRound} + \varepsilon_{method} + \varepsilon_{round}$. The final rounding incurs an error bounded by 2^{-w-1}, which means that the method and the rounding errors must satisfy $\varepsilon_{method} + \varepsilon_{round} < 2^{-w-1}$. Finding a bound on ε_{method} determines the value of m, while finding a bound on ε_{round} gives the number of guard bits g.

Equation 8 can be rewritten so as to reflect the sources for the rounding errors: $x^P \approxeq \widetilde{D}[G - P\widetilde{Fx'}]$, where the tilded operations are approximations of the true mathematical functions. The multiplication by the factor P does not introduce an error, as the constant is a power of two for the three functions considered. The term G is also without error, as the square can be stored in the table correctly rounded. Introducing notations for the error terms, the equation can be further expressed as

$$x^P \approxeq (D + \varepsilon_{tab})[G - P(Fx' + \varepsilon_{mult})] + \varepsilon_{mult}$$

Expanding this expression and grouping the terms gives:

$$x^P \approxeq D[G - PFx'] - DP\varepsilon_{mult} + \varepsilon_{tab}[G - P(Fx' + \varepsilon_{mult})] + \varepsilon_{mult}$$

from which the rounding error can be identified as:

$$\varepsilon_{round} = -DP\varepsilon_{mult} + \varepsilon_{tab}[G - P(Fx' + \varepsilon_{mult})] + \varepsilon_{mult}$$

Taking into account the fact that the for the three functions studied the corresponding values of P are -1, $\frac{1}{2}$ and $-\frac{1}{2}$, and using the upper bounds for the terms D, G, F and x', a bound can be found for ε_{round}:

$$\varepsilon_{round} \leq \overline{\varepsilon_{round}} = \overline{\varepsilon_{mult}}(1 - P) + \overline{\varepsilon_{tab}}(1 - P\overline{\varepsilon_{mult}})$$

Considering that the same internal precision is required for the tabulations and for the truncated multiplications ($\overline{\varepsilon_{tab}} = \overline{\varepsilon_{mult}} = 2^{-w-g}$), then the number of extra guard bits must satisfy:

$$g > 2 + \log(2 - P - P2^{-w-g})$$

This remains, however, a pessimistic bound on g.

The methods presented in this section are practical for precisions of up to about 24 bits.

VI. Large Precision Methods

A. Newton-Raphson Method

Probably the best known iterative method is the Newton-Raphson scheme. This is a root-finding scheme with quadratic convergence. The recurrence relation is $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$ with the error entailed by this iteration approximated to: $\varepsilon_{n+1} \approx \varepsilon_n^2 f''(x_n)/2f'(x_n)$.

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:

$$x_{n+1} = x_n(2 - ax_n) \tag{10}$$

Given an initial approximation, the iteration given by Eq. (10) only requires additions and multiplications.

The total error is again divided as $\varepsilon_{total} = \varepsilon_{finalRnd} + \varepsilon_{rnd} + \varepsilon_m$, and $\varepsilon_{total} < 2^{-w}$. Eq. (10) thus becomes:

$$x_{n+1} = \widetilde{x_n}(2 - \widetilde{a\widetilde{x_n}}) \tag{11}$$

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. (11), it becomes:

$$x_{n+1} = (x_n + 2^{-m-1})(2 - a(x_n + 2^{-m-1}) + \varepsilon_{mul1}) + \varepsilon_{mul2}$$

which can be written as $x_{n+1} = x_n(2 - ax_n) + \varepsilon_{rnd}$ where ε_{rnd} represents the rounding errors:

$$\begin{aligned}\varepsilon_{rnd} &= 2^{-m-1}(2 - ax_n) - 2^{-m-1}a(x_n + 2^{-m-1}) \\ &\quad + \varepsilon_{mul1}(x_n + 2^{-m-1}) + \varepsilon_{mul2}\end{aligned} \tag{12}$$

The sum of the first two terms of Eq. (12) is of the order of 2^{-2m-2}. If ε_{mul1} and ε_{mul2} are of the form 2^{-w-g} (where g are guard bits), Eq. (12) shows that at least 3 guard bits are needed.

A good option for the initial approximation is the use of a bipartite table since for target precisions of up to 32 bits $\lceil \frac{w}{2} \rceil$ is suitable for use with the bipartite method. It is also possible to increase the number of iterations, therefore increasing the number of multiplications, but decreasing the resources needed to obtain the initial approximation.

B. Generalized Iterative Methods

There exist methods with faster than quadratic convergence. The second order Newton-Raphson method is also known as Halley's method. The recurrence equation is given by:

$$x_{n+1} = x_n - \frac{2f(x_n)f'(x_n)}{2(f'(x_n))^2 - f(x_n)f''(x_n)} \tag{13}$$

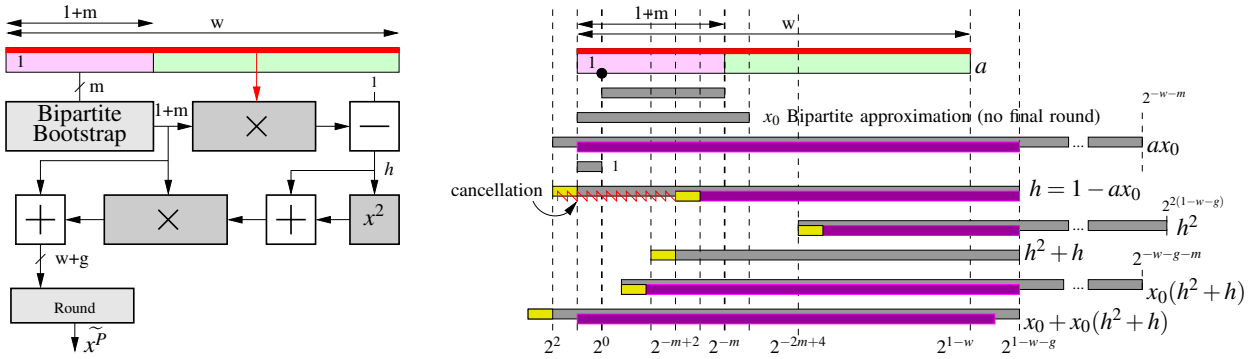
The error entailed by the method can be expressed by a recurrence that shows the cubic convergence of the method:

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

The Newton-Raphson and the Halley method are the first and the second in a class of iterative methods known as the Householder methods [20]. The $n+1$-st term in the class has the form:

$$x_{n+1} = x_n + (n+1)\left(\tfrac{1}{f(x_n)}\right)^{(n)} / \left(\tfrac{1}{f(x_n)}\right)^{(n+1)} \tag{14}$$

where the (n) denotes the n-th order derivative. One Householder iteration improves accuracy $n+1$ times.



(a) Computational datapath

(b) Operand fixed-point alignments

Fig. 5. Halley's method using bipartite bootstrapping for the reciprocal function

The Halley method (or the Householder method, more generally) replaces the tangent to the function plot (in the case of Newton-Raphson) 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, fit the plot better, giving a better approximation. However, as remarked in [21], 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 [22] and even further back presented in [23] 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. A root of $f(x)$ satisfies $f(x) = 0$; the value $x - x_n$ is expressed as a power series of $f(x_n)$:

$$(x - x_n) = af(x_n) + b(f(x_n))^2 + c(f(x_n))^3 + \dots \quad (15)$$

By replacing eq. (15) in the Taylor series expansion around x_n and using $f(x) = 0$ results in:

$$\begin{aligned} 0 &= f(x_n) + f'(x_n)(af(x_n) + b(f(x_n))^2 + c(f(x_n))^3 \dots) \\ &+ f''(x_n)/2!(af(x_n) + b(f(x_n))^2 + c(f(x_n))^3 \dots)^2 + \dots \end{aligned} \quad (16)$$

If eq. (16) 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:

$$a = -\frac{1}{f'(x_n)} \quad b = -\frac{f''(x_n)}{2(f'(x_n))^3} \quad c = -\frac{(f''(x_n))^2}{2(f'(x_n))^5} \quad (17)$$

Replacing the values of Eq. (17) in Eq. (15) results in (showing only the first three terms):

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} - \frac{(f(x_n))^2 f''(x_n)}{2(f'(x_n))^3} - \dots \quad (18)$$

Eq. (18) 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. (18) is chosen as $f(x) = \frac{1}{x} - a$:

$$x_{n+1} = x_n(1 + h_n(1 + h_n)) \qquad h_n = 1 - ax_n \quad (19)$$

In order to obtain the iteration for the reciprocal sqrt, $f(x)$ in eq. (18) is chosen as $f(x) = \frac{1}{x^2} - a$. This results in:

$$x_{n+1} = \frac{1}{8}x_n(8 + h_n(4 + 3h_n)) \qquad h_n = 1 - ax_n^2 \quad (20)$$

Both the iteration for the reciprocal and the one for reciprocal sqrt have cubic convergence. We show next the error analysis for the datapath implementing Eq. (19). The evaluation, on the other hand, is done as in Eq. (21) (where h_n keeps the meaning of eq. (19)):

$$x_{n+1} = x_n + x_n(h_n + h_n^2) \quad (21)$$

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 [19]). This means that fewer bits are needed for the squaring of h_n and for the multiplication $x_n(h_n + h_n^2)$, due to the terms being shifted to the right by $2m$ and m bits respectively.

The total error is divided in three parts, $\varepsilon_{total} = \varepsilon_{finalRnd} + \varepsilon_{rnd} + \varepsilon_m < 2^{-w}$. The final result is rounded, so $\varepsilon_{finalRnd} < 2^{-w-1}$. The rounding errors, ε_{rnd} are due to the initial approximation and to truncating the multiplications in Eq. (21). Making these errors explicit, Eq. (21) becomes (where m is the precision of the x_n approximation):

$$\begin{aligned} x_{n+1} &= (x_n + 2^{-m-1}) + (x_n + 2^{-m-1})(h_n - a2^{-m-1} - \varepsilon_{mul} \\ &+ (h_n - a2^{-m-1} - \varepsilon_{mul})^2) \end{aligned}$$

or $x_{n+1} = x_n + x_n(h_n + h_n^2) + \varepsilon_{rnd}$ where

$$\begin{aligned} \varepsilon_{rnd} &= 2^{-m-1}(1 + h_n + h_n^2) + \\ &(x_n + 2^{-m-1})(a2^{-m-1} + \varepsilon_{mul})(a2^{-m-1} + \varepsilon_{mul} - 1 - 2h_n) \end{aligned}$$

which can be re-written as:

$$\begin{aligned} \varepsilon_{rnd} &= 2^{-m-1}(1 + h_n + h_n^2) + \\ &+ (x_n + 2^{-m-1})[((-a)2^{-m-1} + a^2 2^{-2m-2} - a2^{-m}h_n) + \\ &+ \varepsilon_{mul}(1 - \varepsilon_{mul} - 2h_n + a2^{-m})] \end{aligned} \quad (22)$$

TABLE I
HOLISTIC IMPLEMENTATION SYNTHESIS RESULTS FOR $1/x$ AND $1/\sqrt{x}$ COMPARED TO CLASSICAL RANGE REDUCE/RECONSTRUCTION TECHNIQUE. TARGET DEVICE CYCLONE V, SPEEDGRADE -6.

f	(w, f)	ALM	DSPs	M10K	Latency	Frequency
$\frac{1}{x}$	16,15	49	1	3	7	310MHz
	16,14	82	1	8	9	310MHz
	16,13	85	1	10	9	310MHz
	16,12	79	1	11	9	310MHz
	16,11	69	1	8	7	310MHz
	16,10	72	1	8	7	310MHz
	16, 9	61	1	6	7	310MHz
	16,8	48	1	5	6	295MHz
	16,7	47	1	5	6	283MHz
	16,6	44	1	4	6	287MHz
	16,5	50	1	2	6	307MHz
	16,4	27	0	1	2	315MHz
	Generic Range Reduction					
	16 bit	148	0	5	11	310MHz
$\frac{1}{\sqrt{x}}$	16,15	49	1	3	7	310MHz
	16,14	73	1	9	9	310MHz
	16,13	79	1	11	9	310MHz
	16,12	70	1	8	7	310MHz
	16,11	69	1	8	7	310MHz
	16,10	48	1	6	6	294MHz
	16,9	46	1	5	6	285MHz
	16,8	45	1	4	6	289MHz
	16,7	44	1	4	6	302MHz
	16,6	52	1	2	6	294MHz
	16,5	49	1	2	6	310MHz
	16,4	39	1	2	6	310MHz
	Generic Range Reduction					
	16 bit	242	0	5	16	311MHz

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

The terms of Equation (22) that do not contain ε_{mul} will result in a term that is of the order of 2^{-3m-3}. Thus, if ε_{mul} 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.

VII. RESULTS

Table I shows the performance of the holistic implementation method introduced in Section IV for the reciprocal and reciprocal square root functions. We used various fixed-point formats to highlight the implementation results.

There are several architectural variations of this solution. A specialized underflow architecture will trigger when a significant output range underflows with respect to the output format. There are 2 variants of this architecture: for instance (16,4) triggers Fig. 3(a), while (16,5) triggers Fig. 3(b). Format (16,15) triggers the architecture with approximation sufficiently accurate on the compute range (except overflow). One architecture uses extra tables for inputs where the 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 that 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: regular or base+offset. 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 x; the range close to 0 will be handled separately as handling it in the table would increase the memory requirements.

The final row for each function (marked *Generic Range Reduction*) shows results for the bipartite method with the normalization, compute and reconstruct stages. This implementation is agnostic to the input/output format. Comparing the proposed holistic implementation to this generic one we observe that in general we always improve latency. Our implementations use one multiplier (half DSP) but generally consume fewer ALM resources. For the function $1/\sqrt{x}$ our implementation shows the best ALM savings compared to the generic implementation. This is due to the complexity of the reconstruction stage, that in addition to the classical barrel shifter, also requires a constant multiplier by $\sqrt{2}$ (see Section III). 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 otherwise.

Table II shows the performance for our proposed methods for kernels with $x \in [1,2)$. For $1/x$, our proposed 2^{nd} order *Tabulate and Multiply* method is compared against a state-of-the art FloPoCo minimax-based piecewise-polynomial approximation core [24]. For 24-bit precision, our method saves one memory block, consumes fewer ALMs and has a shorter latency while consuming one 18x18-bit multiplier more (there are two such multipliers per DSP block). We have also compared against [25] by extrapolating the minimax-based VLSI implementation to the FPGA target. Our proposed savings and tradeoffs are very similar to the first comparison. The results are very similar for the two other functions: $1/\sqrt{x}$ and $\sqrt{x}$. Here our proposed implementation offers a tradeoff by consuming fewer memory blocks, at the expense of 1 half-DSP multiplier. We have also compared resources against a 1^{st} order implementation, and have shown that our proposed 2^{nd} order implementation scales better for precisions larger than 20 bits in terms of memory requirements.

We have compared architectures based on the newly introduced Halley's method against FloPoCo's fixed-point implementation [24]. We have used an improved implementation that bootstraps the Halley iteration using a Bipartite approximation, as opposed to a table. This change has allowed saving 2 memory blocks at the expense of 17 ALMs. For $1/x$ our proposed implementation outperforms the FloPoCo core in all metrics. The savings are more than 40% ALMs, 3 memory blocks (75%) and a reduction of latency from 13 to 11. For the $1/\sqrt{x}$ the slightly more complicated recurrence (Eq. 20) makes the implementation a tradeoff, offering a saving in memory blocks at the expense of multipliers.

VIII. CONCLUSION

In this work we have proposed three methods for implementing fixed-point functions from the family x^p with $p \in \{-1,-1/2,1/2\}$. We have shown with our holistic implementation methodology that both logic resources and latency can be saved when accounting for the user-specific input/output formats when generating the implementation. We have extended to second order a method by Takagi [7] that makes

TABLE II
PERFORMANCE FOR SOME OF DESCRIBED ARCHITECTURES ON CYCLONEV C6 FOR $X \in [1,2)$. RESULTS OBTAINED USING QUARTUS 16.1. FREQUENCIES MARKED WITH * ARE RESTRICTED FMAX. MULTS COUNTED AS 18X18 EQUIVALENTS (2/DSP).

f	I/O (w,f)	Implementation	Resource utilization and Performance					Multipliers	Tables
			ALMs	MULTs	M10Ks	Lat.	Frequency		
$\frac{1}{x}$	19,18	Tab. Mult. 1^{st}	29	2	1	4	250MHz*	$U20_u \times U21_u$	512×21
	21,20	Tab. Mult. 1^{st}	30	2	3	4	250MHz*	$22_u \times 23_u$	1024×23
		Tab. Mult. 2^{nd}	48	4	1	5	250MHz*	$22_u \times S14_s, 23_u \times 24_u$	128×38
	23,22	Tab. Mult. 1^{st}	32	2	5	4	250MHz*	$24_u \times 25_u$	2048×25
		Tab. Mult. 2^{nd}	135	4	0	5	250MHz*	$24_u \times 16_s$, $25_u \times 26_s$	128×40
		Tab. Mult. 2^{nd}	51	4	1	5	250MHz*	$24_u \times 16_s$, $25_u \times 26_s$	128×40
	24,23	Tab. Mult. 1^{st}	33	2	13	4	250MHz*	$25_u \times 26_u$	4096×26
		Tab. Mult. 2^{nd}	60	4	1	5	250MHz*	$25_u \times 16_s$, $26_u \times 27_s$	$256 \times (40+3)$
		FloPoCo d=2	75	3	2	6	249MHz	$16_s \times 13_s$, $16_s \times 21_s$	256×61
		[25]		*3*	*2*	*5*		$15_u \times 15_u T$, $15_{uT} \times 7_u$, $15_u \times 15_u$	256×49
	32,31	Bipartite+Newton	106	4	5	9	246MHz	$32_u \times 18_u$, $18_u \times 35_u$	2048×18, 1024×6
		Table+Halley	92	6	3	10	250MHz*	$32_u \times 12_u$, $26_s \times 26_s$, $12_u \times 27_s$	2048×12
		Bipartite+Halley	109	6	1	11	250MHz*	$32_u \times 13_u$, $26_s \times 26_s$, $13_u \times 27_s$	256×13, 64×4
		FloPoCo d=3	191	6	4	13	196MHz	$19_s \times 15_s$, $19_s \times 22_s$, $24_s \times 31_s$	256×100
$\frac{1}{\sqrt{x}}$	24,23	Tab. Mult. 2^{nd}	87	4	1	9	250MHz*	$27_u \times 26_u$, $16_u \times 25_u$	$256 \times (40+3)$
		FloPoCo d=2	86	3	2	7	250MHz*	16×12, 16×20,	256×59
	32,31	Bipartite+Halley	169	9	1	19	247MHz	$13_u \times 13_u$, $32_u \times 25_u$, $24_s \times 24_s$, $24_s \times 13_u$	256×40 (Dual port)
		FloPoCo d=3	163	7	4	16	240MHz	18×14, 18×21, $24 \times 31s$	256×98
$\sqrt{x}$	24,23	Tab. Mult. 2^{nd}	74	4	1	8	250MHz*	$27_u \times 26_u$, $16_u \times 25_u$	$256 \times (40+3)$
		FloPoCo d=2	77	3	2	7	250MHz*	16×10, 16×21,	256×58

extensive use of logic and bit-fiddling techniques for reducing resources. The proposed technique has a smaller table footprint compared to state-of-the-art architectures based on minimax polynomials. The proposed higher order methods based on the cubic convergence of Halley's method, combined with efficient Bipartite bootstrapping have proved to clearly outperform state-of-the-art minimax-based implementations for $1/x$, and to provide a tradeoff for the $1/\sqrt{x}$. The proposed architectures often offer tradeoffs against other possible implementations. The best place for these types of methods is behind function generators, where the best possible candidates are selected, based on user-specification.

REFERENCES

[1] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "Cnp: An FPGA-based processor for convolutional networks," in *2009 International Conference on Field Programmable Logic and Applications*, Aug 2009, pp. 32–37.

[2] J. Schlessman, C.-Y. Chen, W. Wolf, B. Ozer, K. Fujino, and K. Itoh, "Hardware/software co-design of an FPGA-based embedded tracking system," in *2006 Conference on Computer Vision and Pattern Recognition Workshop (CVPRW'06)*, June 2006, pp. 123–123.

[3] D. Chen and M. Sima, "Fixed-point CORDIC-based QR decomposition by Givens rotations on FPGA," in *2011 International Conference on Reconfigurable Computing and FPGAs*, Nov 2011, pp. 327–332.

[4] M. Karkooti, J. R. Cavallaro, and C. Dick, "FPGA implementation of matrix inversion using QRD-RLS algorithm," in *Asilomar Conference on Signals, Systems, and Computers*, 2005.

[5] D. Perrelet, A. Villanueva, M. Sundal, Y. Brischetto, D. Oberson, and H. Damerau, "White-rabbit based revolution frequency program for the longitudinal beam control of the CERN PS," 2015.

[6] J. L. V. M. Stanislaus and T. Mohsenin, "Low-complexity FPGA implementation of compressive sensing reconstruction," in *Computing, Networking and Communications (ICNC), 2013 International Conference on*, Jan 2013, pp. 671–675.

[7] N. Takagi, "Generating a power of an operand by a table look-up and a multiplication," in *ARITH'13*, Jul 1997, pp. 126–131.

[8] A. J. Chung, K. Cobden, M. Jervis, M. Langhammer, and B. Pasca, "Tools and Techniques for Efficient High-Level System Design on FPGAs," *CoRR*, vol. abs/1408.4797, 2014.

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

[10] 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.

[11] 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.

[12] D. Das Sarma and D. Matula, "Faithful bipartite ROM reciprocal tables," in *ARITH'12*, Jul 1995, pp. 17–28.

[13] M. Schulte and J. Stine, "Symmetric bipartite tables for accurate function approximation," in *ARITH'13*, Jul 1997, pp. 175–183.

[14] J. Stine and M. Schulte, "The symmetric table addition method for accurate function approximation," *VLSI*, vol. 21, pp. 167–177, 1999.

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

[16] F. de Dinechin and A. Tisserand, "Some improvements on multipartite table methods," in *ARITH'15*, 2001, pp. 128–135.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

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

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

[24] F. de Dinechin, M. Joldes, and B. Pasca, "Automatic generation of polynomial-based hardware architectures for function evaluation," in *ASAP'21*, Rennes, Jul. 2010.

[25] J. A. Pineiro, J. D. Bruguera, and J. M. Muller, "Faithful powering computation using table look-up and a fused accumulation tree," in *Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001*, 2001, pp. 40–47.