On fixed-point hardware polynomials

Florent de Dinechin

On fixed-point hardware polynomials

Florent de Dinechin
Univ Lyon, INSA, CITI Lab
Florent.de-Dinechin@insa-lyon.fr

Abstract—Polynomial approximation is a general technique for the evaluation of numerical functions of one variable. This article addresses the automatic construction of fixed-point hardware polynomial evaluators. By systematically trying to balance the accuracy of all the steps that lead to an architecture, it simplifies and improves the previous body of work covering polynomial approximation, polynomial evaluation, and range reduction. This work is supported by an open-source implementation.

Index Terms—elementary function; hardware evaluator; polynomial approximation; computer arithmetic;

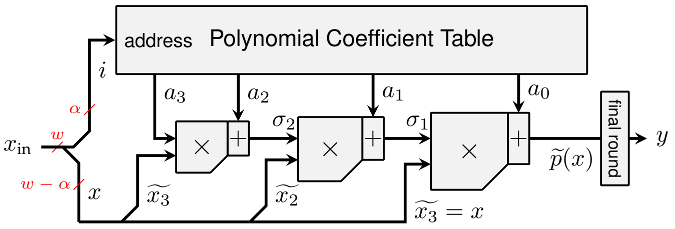

Fig. 2. A polynomial evaluator (uniform segmentation, Horner scheme)

1 Introduction

This article addresses the automatic construction of fixed-point polynomial evaluators for a real function f of one real variable x over a closed interval I. This is a very generally useful building block when designing hardware functions in fixed-point [1], [2], [3] or floating-point [4], [5], [6] for application-specific processors or reconfigurable computing [7], [8]. This article refines several previous works describing generic function approximators [9], [10], [11]: tools that input a function and parameters specifying its context, and produce an architecture, as illustrated by Figure 1.

The construction of a polynomial evaluator typically proceeds in three steps. The first is the computation if a polynomial that *approximates* the function f. The second is the generation of adders and multipliers that will *evaluate* this polynomial. These two steps are usually preceded by a *range-reduction* that transform the initial function into one more suited for polynomial approximation [12], [10], [6].

The main constribution of this article is to show that these three steps (Sections 2 to 4) are deeply linked by arithmetic considerations. Expliciting these links enables a very simple derivation of near-optimal values of most architectural parameters, for which previous works often resort to complex heuristics. Several minor improvements are introduced on the way.

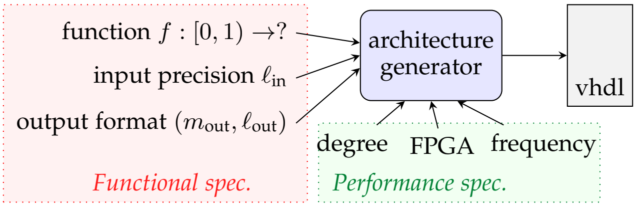
function $f : [0,1) \to ?$
input precision ℓ_{in}
output format $(m_{\text{out}}, \ell_{\text{out}})$
architecture generator
vhdl
degree
FPGA
frequency
Functional spec.
Performance spec.

Fig. 1. Interface to a generic fixed-point function approximator

This work is supported by an open-source implementation in the FloPoCo[1] generator [13]. Approximation and evaluation are implemented as the `BasicPolyApprox` and `FixHornerEvaluator` classes, while the `FixFunctionBySimplePoly` and `FixFunctionByPiecewisePoly` classes combine them to build architectures. The interested reader will find in their respective source code all the details not exposed here for clarity.

1.1 Unit intervals

We assume that f is continuously differentiable over a closed interval I up to a certain order. In this work, we will systematically normalize I to $[0,1]$ or $[-1,1]$. Any function g that has a different input interval may be transformed into a function f on $I=[0,1]$ or $I=[-1,1]$ by means of a suitable change of variable. Technically, in all practical cases, there exists an affine transformation g such that $g([-1,1]) = I$ and g entails little or no additional hardware cost. We then consider $f \circ g(x)$ on $[-1,1]$. Examples will be given in the sequel.

The reason for normalizing functions to unit intervals is that it greatly simplifies the arithmetic analysis needed to build an efficient architecture. This is indeed at the core of the contributions of this article. In other words, if a function has, in its original context, a different input range (for instance $[0, 2^{-k}]$ after range reduction in [2] or [4]), the present work claims that rewriting it as a function on $[0,1]$ or $[-1,1]$ is relevant for a fixed-point implementation.

[1] Version 3.1 or later, see http://flopoco.gforge.inria.fr/

1.2 Fixed-point formats

Following the VHDL standard, a fixed-point format is defined by two integers (m, ℓ) that respectively denote the position of the most significant and least significant bit (MSB and LSB) of the data. The respective values of the MSB and LSB are thus 2^m and 2^ℓ. The LSB position ℓ denotes the *precision* of the format. The MSB position denotes its *range*. Both m or ℓ can be negative if the format includes fractional bits. In signed arithmetic, m is the position of the sign bit.

For instance, on Figure 1, an input in $[0, 1]$ is represented in the $(-1, \ell_{\text{in}})$ format.

1.3 Error analysis for computing just right

One key idea of this work is that **architectures should compute, at each step, accurately enough, but no more**, if more accuracy costs hardware or latency. Formally, accuracy is achieved by bounding errors. An error, denoted ε, is always defined as the difference between two values, one being more accurate than the other. For instance, $\varepsilon_{\text{total}} = y - f(x)$ is the error of the computed output y with respect to the real value of the function $f(x)$. This error should not exceed the value of the LSB of the output: $\forall x \in I,\ |\varepsilon_{\text{total}}| < 2^{\ell_{\text{out}}}$. Indeed, a higher accuracy cannot be expressed on the output, while a lower accuracy would mean that some outputs bits hold no useful information. Therefore, the output precision ℓ_{out} also serves to specify the *accuracy* of the implementation (Figure 1). This accuracy objective is often termed *faithful rounding* in the literature.

The maximal absolute value of an error ε over the interval I is noted $\overline{\varepsilon}$, so the previous objective can also be stated as $\overline{\varepsilon}_{\text{total}} < 2^{\ell_{\text{out}}}$.

This total error is decomposed as follows. The function f is approximated by a polynomial p, with error $\varepsilon_{\text{approx}}(x) = p(x) - f(x)$. The evaluation of $p(x)$ involves rounding errors: the value actually computed by the hardware will be noted $\widetilde{p}(x)$. The sum of all rounding errors in the architecture is noted $\varepsilon_{\text{eval}}(x) = \widetilde{p}(x) - p(x)$. As $\widetilde{p}(x)$ must be evaluated to an internal precision slightly higher than ℓ_{out}, it finally needs to be rounded to the target format: $\varepsilon_{\text{final round}} = y - \widetilde{p}(x)$ is the corresponding error. Thus,

$$\begin{aligned} \varepsilon_{\text{total}} &= y - f(x) \\ &= y - \widetilde{p}(x) \ + \ \widetilde{p}(x) - p(x) \ + \ p(x) - f(x) \\ &= \varepsilon_{\text{final round}} + \varepsilon_{\text{eval}} + \varepsilon_{\text{approx}} \end{aligned} \tag{1}$$

Here, $\overline{\varepsilon}_{\text{final round}}$ is actually the dominant source of error. It is, at worst, one half of the weight of the output LSB: $\overline{\varepsilon}_{\text{final round}} = 2^{\ell_{\text{out}}-1}$. The constraint on the two other errors to ensure $\overline{\varepsilon}_{\text{total}} < 2^{\ell_{\text{out}}}$ is therefore $\varepsilon_{\text{approx}} + \varepsilon_{\text{eval}} < 2^{\ell_{\text{out}}-1}$.

To balance the errors, the polynomial approximation step is given the error budget $\overline{\varepsilon}^{\text{target}}_{\text{appprox}} = 2^{\ell_{\text{out}}-2}$. The approximation step computes the polynomial of minimum degree that allows to match this error budget. It also reports the actual $\overline{\varepsilon}_{\text{approx}}$. This defines the error budget for the evaluation as $\overline{\varepsilon}^{\text{target}}_{\text{eval}} = \overline{\varepsilon}_{\text{final round}} - \overline{\varepsilon}_{\text{approx}}$. This value is passed to the generator of polynomial hardware, whose task is to dimension the adders and multipliers to achieve it. This process is illustrated on Figure 3.

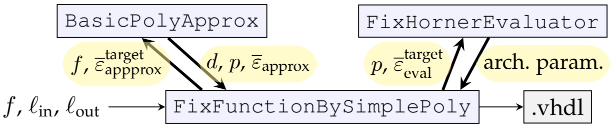
BasicPolyApprox
FixHornerEvaluator
f, $\overline{\varepsilon}^{\text{target}}_{\text{appprox}}$
d, p, $\overline{\varepsilon}_{\text{approx}}$
p, $\overline{\varepsilon}^{\text{target}}_{\text{eval}}$
arch. param.
f, ℓ_{in}, ℓ_{out}
FixFunctionBySimplePoly
.vhdl

Fig. 3. Execution flow of a basic polynomial approximator

1.4 Generic range reduction techniques

This general error analysis scheme may be adapted to most range reduction techniques. For example, the architecture of Figure 2 implements the uniform segmentation scheme [9], [11]: it splits the input interval into 2^α sub-intervals of identical size (Figure 5, left). An approximation polynomial can then be used on each interval. The coefficients are stored in a table addressed by the α MSBs of the input (noted i throughout this paper). The motivation for this is that the same accuracy can be obtained with polynomials of smaller degree on the smaller intervals. Actually, in such a scheme, the degree is an input to the generator. The generator computes the smallest α that entails that the target accuracy can be reached with this degree (see Figure 4). Thus, the degree is a way of controlling the trade-off between table cost and arithmetic cost.

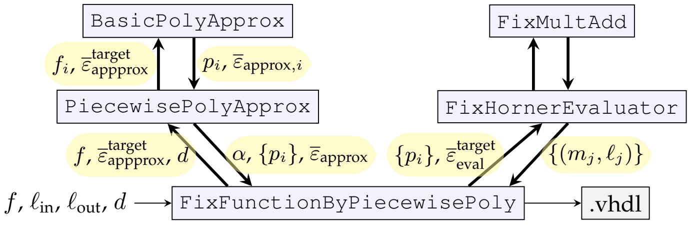
BasicPolyApprox
FixMultAdd
f_i, $\overline{\varepsilon}^{\text{target}}_{\text{appprox}}$
p_i, $\overline{\varepsilon}_{\text{approx},i}$
PiecewisePolyApprox
FixHornerEvaluator
f, $\overline{\varepsilon}^{\text{target}}_{\text{appprox}}$, d
α, $\{p_i\}$, $\overline{\varepsilon}_{\text{approx}}$
$\{p_i\}$, $\overline{\varepsilon}^{\text{target}}_{\text{eval}}$
$\{(m_j, \ell_j)\}$
f, ℓ_{in}, ℓ_{out}, d
FixFunctionByPiecewisePoly
.vhdl

Fig. 4. Execution flow of `FixFunctionByPiecewisePoly`

In such a scheme, the previous error analysis simply needs to consider the worst case over the sub-intervals i: the error approximation target must be reached by all the approximation polynomials p_i, each on its sub-interval; the evaluation error budget is computed out of the worst-case error over i; finally, the evaluation hardware needs to be shared among all the intervals, so its datapath must be dimensioned to fit all i. This simply consists in considering, for each fixed-point format, the max (over i) of the MSBs and the min of the LSBs.

Some functions have singularities on the interval endpoints (for instance the square root depicted on Figure 5, right). They may be best implemented by power-of-two segmentation [10]. The only difference with uniform seg-

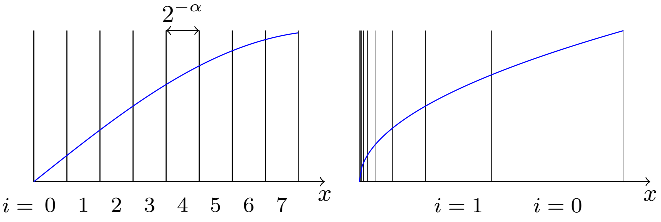

Fig. 5. Uniform (left) and power-of-two (right) segmentation

mentation is that the interval index is obtained by a leading-zero or leading-one counter, while the reduced argument is obtained by the corresponding shift on the input. These leading zero counters and shifters must be added to Figure 2.

Hierarchical segmentation is studied in [10]. The conclusion is that two levels always suffice in practice, with only the outer one needing power-of-two splitting.

This paper is demonstrated on the simpler uniform segmentation scheme, but the techniques presented here (in particular in Section 3) apply to all these other generic range reduction techniques, and even to function-specific range reductions [12], [2], [3], [4], [6].

2 Polynomial approximation

Textbooks [12] provide many methods for approximating a function with a polynomial. Taylor or Chebyshev are well-known analytical methods, but the best approximations are provided by Remez' algorithm, a numerical process that, under some conditions, converges to the minimax approximation: the polynomial p_{remez} of degree d that minimizes $||p-f||_\infty$. A problem is that the coefficients of p_{remez} are real numbers that must be rounded to finite-precision machine numbers. This introduces a new error that has to be taken into account [9], [10]. Besides, this gives a new polynomial $\widetilde{p}_{\text{remez}}$ which is in general no longer the best possible among the polynomials whose coefficients are machine numbers.

The state of the art is therefore the modified Remez algorithm [14] available as the `fpminimax` command of the Sollya tool [15]. It directly provides a minimax polynomial among the polynomials whose coefficients satisfy some user-specified constraints. These constraints specify the LSB weight of each coefficient.

To define these constraints, consider a theoretical architecture evaluating the developed form of a polynomial $a_0 + a_1x + a_2x^2 + \ldots + a_dx^d$ on $I = [0,1]$ or $I = [-1,1]$. It would first compute each monomial a_jx^j, then align them to compute the fixed-point sum. Such an alignment is depicted on Figure 6 for a polynomial evaluating $f(x) = e^x$ on $[0,1]$ to an accuracy of 10^{-6} (it is close to the Taylor series).

As $|x| \leq 1$ we also have $|x^j| \leq 1$, hence $|a_jx^j| \leq |a_j|$. Therefore the MSB of each monomial term a_jx^j is that of a_j.

Let us now discuss the LSBs to derive the constraints to input to `fpminimax`. There is no point in evaluating one of the monomials much more accurately than the others, as the sum will be no more accurate than its least accurate term. Besides, for x close to 1, the monomial a_jx^j cannot be more accurate than a_j itself. A near-optimal design decision is therefore to have each a_j accurate to the same LSB ℓ_a. The maximal value of ℓ_a may be determined out of the accuracy objective $\overline{\varepsilon}^{\text{target}}_{\text{appprox}}$. Intuitively, we need $2^{\ell_a} \leq \overline{\varepsilon}^{\text{target}}_{\text{appprox}}$ so the coefficients hold information that is at least as accurate as $\overline{\varepsilon}^{\text{target}}_{\text{appprox}}$. In practice, surprisingly, there usually exist polynomials satisfying $\overline{\varepsilon}_{\text{approx}} \leq \overline{\varepsilon}^{\text{target}}_{\text{appprox}}$ even when 2^{ℓ_a} is sligthly larger than $\overline{\varepsilon}^{\text{target}}_{\text{appprox}}$. This effect improves with the degree d, therefore a good heuristic is to start by calling `fpminimax` with $\ell_a = \lceil \log_2(\overline{\varepsilon}^{\text{target}}_{\text{appprox}} \times d) \rceil$. If `fpminimax` fails to obtain a polynomial that satisfies $\overline{\varepsilon}_{\text{approx}} < \overline{\varepsilon}^{\text{target}}_{\text{appprox}}$, then ℓ_a is decreased until it succeeds. This loop (formalized in Algorithm 1) rarely needs more than a few attempts and its execution time remains well below one second. It is implemented in FloPoCo in the `BasicPolyApprox` class. The computation of $\overline{\varepsilon}_{\text{approx}}$ is performed with Sollya `supnorm`(p, f, I) command [16].

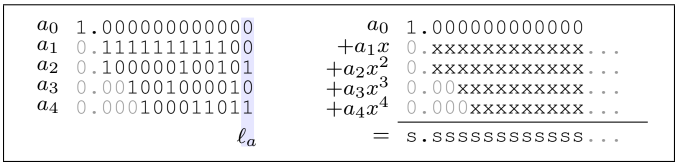

Fig. 6. If $x \in [-1,1]$, the alignment of the a_jx^j follows that of the a_j

Algorithm 1 Approximation for $I = [0,1]$ or $I = [-1,1]$

```
procedure FixedPointPolyApprox($f, d, I, \overline{\varepsilon}^{\text{target}}_{\text{appprox}}$)
    $\ell_a \leftarrow \lceil \log_2(\overline{\varepsilon}^{\text{target}}_{\text{appprox}} \times d) \rceil - 1$
    repeat
        $\ell_a \leftarrow \ell_a + 1$
        $p \leftarrow$ fpminimax($f, d, I, \{\ell_a\}$)
        $\overline{\varepsilon}_{\text{approx}} \leftarrow$ supnorm($p, f, I$)
    until $\overline{\varepsilon}_{\text{approx}} \leq \overline{\varepsilon}^{\text{target}}_{\text{appprox}}$
end procedure
```

This reasoning assumed a theoretical architecture evaluating the developed form, but it actually holds for any evaluation scheme. For instance, let us consider the classical Horner evaluation scheme, used in most previous work [9], [10], [11] because it minimizes the number of operations :

$$p(x) = a_0 + x \times (a_1 + x \times (a_2 + ... + x \times a_d)...) \,. \quad (2)$$

It can be expanded in the following recurrence:

$$\begin{cases} \sigma_d = a_d \\ \pi_j = x \times \sigma_{j+1} \quad \forall j \in \{0 \ldots d-1\} \\ \sigma_j = a_j + \pi_j \quad \forall j \in \{0 \ldots d-1\} \\ p(x) = \sigma_0 \end{cases} \quad (3)$$

The reader may observe that, for $|x| \leq 1$, having all the a_j accurate to the same LSB ℓ_a again entails a recurrence where no step is needlessly more accurate than the others: Algorithm 1 also works in this case. The same will be true of evaluation schemes intermediate between Horner and the developed form, such as Estrin's [12].

3 Argument range reduction

The analysis of the previous section is based on the fact that the function input range is normalized to [0,1] or [-1,1]. But the purpose of argument range reduction is, as the name suggest, to reduce this interval. For instance, the architecture of Figure 2 actually evaluates $f(x_{\text{in}})$ on small sub-intervals, e.g. $x_{\text{in}} \in [0, 2^{-\alpha}]$, enabling smaller-degree polynomial than on the full interval.

Let us define the change of variable $x_{\text{in}} = 2^{-\alpha}x$, such that $f(x_{\text{in}}) = f(2^{-\alpha}x) = g(x)$ for $x \in [0,1]$. Note that it translates both the MSB and the LSB of x_{in} by α positions. Applied to an approximation polynomial, this change of variables gives $p_f(x_{\text{in}}) = a_0 + a_1x_{\text{in}} + a_2{x_{\text{in}}}^2 \ldots = a_0 + 2^{-\alpha}a_1x + 2^{-2\alpha}a_2x^2 \ldots = p_g(x)$. In other words, **range**

reduction can also be viewed as a reduction of the polynomial coefficients, here by a factor $2^{j\alpha}$ for the coefficient of degree j. This is illustrated by Figure 7 on a few examples extracted from uniform piecewise approximations to exp and log. Having normalized the function to a unit interval then enables the use of Algorithm 1.

As a parenthesis, Figure 7 also gives a visual clue how the degree is determined by the target accuracy. Indeed, on Figures 7b/ and 7c/, a 4-th degree coefficient a_4 would be to the right of the picture, contributing no useful bits. However, this is only true if all the coefficients have the same sign, as Figure 7d/ illustrates. In the general case, the Sollya command guessdegree$(f, I, \overline{\varepsilon}_{\text{appprox}}^{\text{target}})$ provides a good estimate of the degree needed to reach a given accuracy.

In [9], [10], or [11], the change of variable used is $x_{\text{in}} = 2^{-\alpha}(i + x)$ for $x \in [0, 1)$ and $i \in \{0, 1 \dots 2^{\alpha} - 1\}$. The normalised i-th function is $f_i^u(x) = f(2^{-\alpha}(i + x))$. The reduced argument x is obtained as the LSBs of x_{in} in the case of uniform segmentation (Figure 2), as a shift of x_{in} in the case of power-of-two segmentation.

One novelty of the present work is to center x on its sub-interval: we introduce the change of variable $x_{\text{in}} = 2^{-\alpha}(i + \frac{1}{2} + \frac{x'}{2})$ for $x' \in [-1, 1)$. The normalised i-th function becomes $f_i^s(x') = f(2^{-\alpha}(i + \frac{1}{2} + \frac{x'}{2}))$. In general, this leads to smaller coefficients for the approximation polynomials, as illustrated by Figure 7c/ versus 7b/. The architectural overhead to obtain x' out of x is limited to one inverter (not shown on Figure 2 for simplicity) to complement the MSB of x. Its delay is largely hidden by the table access delay.

Another remark can be made about the sign of the coefficients: analytical properties of the function (monotinicity, convexity, and so on) are defined by the signs of the successive derivatives. Good approximation polynomials typically inherit these properties, which are then often reflected in the signs of the corresponding coefficients. For instance, on Figure 7, all the coefficients for the various exponentials are positive. As a consequence, for a piecewise approximation of an elementary function, it is typical (but not automatic due to numerical artifacts) that all the coefficients of a given degree have the same sign. In this case, the sign bit need not be stored, saving one bit per coefficient.

4 Polynomial evaluation

This section discusses the sizing of all variables on the Horner evaluator of Figure 2. As for evaluation, it can be straightforwardly extended to other schemes. .

Consider one Horner step of (3). As the a_j are known, it is possible to compute the ranges of the π_j and $\widetilde{\sigma}_j$, hence their MSBs, by straightforward interval evaluation of (3) on $[-1, 1]$. This is implemented in a procedure ComputeHornerMSBs. In the typical scenario where x is a reduced argument, a_j is larger in magnitude than π_j, therefore the MSB of σ_j will be that of a_j, sometimes plus one (overflow bit to the left). This is the case depicted on Figure 8. However it may also happen that a_j is smaller in magnitude than π_j.

Concerning the LSBs, we have two opportunities of reducing the size of intermediate computations. Firstly, the product π_j may be rounded or truncated at each step, entailing an error bounded by $\overline{\varepsilon}_{\pi}(j)$. Secondly, x may also be truncated to reduce the multiplier size (informally, to remove bits of x that the multiplication by σ_{j+1} will shift so far right that they don't affect the result – the formal formulation follows). This truncation of x was a contribution of [11].

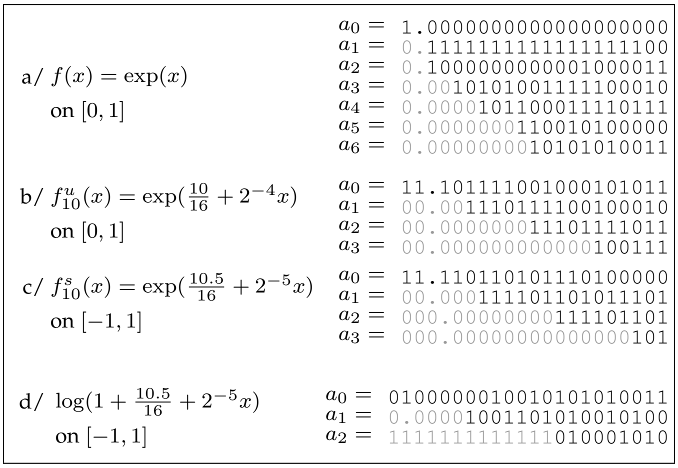
a/ $f(x) = \exp(x)$
on $[0, 1]$
$a_0 =$ 1.00000000000000000000
$a_1 =$ 0.11111111111111111100
$a_2 =$ 0.10000000000001000011
$a_3 =$ 0.00101010011111100010
$a_4 =$ 0.00001011000111110111
$a_5 =$ 0.00000001100101000000
$a_6 =$ 0.00000000010101010011
b/ $f_{10}^u(x) = \exp(\frac{10}{16} + 2^{-4}x)$
on $[0, 1]$
$a_0 =$ 11.101111001000101011
$a_1 =$ 00.001110111100100010
$a_2 =$ 00.000000011101111011
$a_3 =$ 00.000000000000100111
c/ $f_{10}^s(x) = \exp(\frac{10.5}{16} + 2^{-5}x)$
on $[-1, 1]$
$a_0 =$ 11.110110101110100000
$a_1 =$ 00.000111101101011101
$a_2 =$ 000.00000000111101101
$a_3 =$ 000.00000000000000101
d/ $\log(1 + \frac{10.5}{16} + 2^{-5}x)$
on $[-1, 1]$
$a_0 =$ 0100000010010101010011
$a_1 =$ 0.00001001101010010100
$a_2 =$ 1111111111111010001010

Fig. 7. Range reduction reduces the a_j. All the polynomials are accurate to 10^{-6}; f_{10}^u and f_{10}^s cover the same sub-interval of $\exp(x)$.

The recurrence that is actually computed is thus:

$$\begin{cases} \widetilde{\sigma}_d = a_d \\ \widetilde{\pi}_j = \widetilde{x}_j \times \widetilde{\sigma}_{j+1} + \varepsilon_{\pi}(j) & \forall j \in \{0 \dots d-1\} \\ \widetilde{\sigma}_j = a_j + \widetilde{\pi}_j & \forall j \in \{0 \dots d-1\} \\ \widetilde{p}(x) = \sigma_0 \end{cases}$$

Remark that the addition is exact, since it is a fixed-point addition.

The evaluation error of step j can now be defined as:

$$\begin{aligned} \varepsilon_{\text{eval}}(j) &= \widetilde{\sigma}_j - \sigma_j = (a_j + \widetilde{\pi}_j) - (a_j + \pi_j) = \widetilde{\pi}_j - \pi_j \\ &= \widetilde{\pi}_j - \widetilde{x}_j \widetilde{\sigma}_{j+1} \ + \ \widetilde{x}_j \widetilde{\sigma}_{j+1} - x_j \sigma_{j+1} \\ &= \varepsilon_{\pi}(j) + \widetilde{x}_j \widetilde{\sigma}_{j+1} - x_j \widetilde{\sigma}_{j+1} \ + \ x_j \widetilde{\sigma}_{j+1} - x_j \sigma_{j+1} \\ &= \varepsilon_{\pi}(j) \ + \ (\widetilde{x}_j - x_j) \widetilde{\sigma}_{j+1} \ + \ x_j \varepsilon_{\text{eval}}(j+1) \quad . \end{aligned} \tag{4}$$

Algorithm 2 computes all the MSBs and LSBs of the Horner datapath of Figure 2 according to (4). It is a loop again: it computes $\overline{\varepsilon}_{\text{eval}} = \overline{\varepsilon}_{\text{eval}}(0)$, and adds more LSB bits if this error exceeds the target. Again, the argument that no step needs to be more accurate than the others drives us to have a common LSB ℓ_{σ}. It will be the LSB of all the σ_j but also of the π_j (see Figure 8). As $\sigma_j = a_j + \widetilde{\pi}_j$, σ_j must be at least as accurate as a_j, otherwise bits of a_j would be useless. Therefore, line 3 of Algorithm 2 initializes ℓ_{σ} to ℓ_a.

Let us now address the truncation of x. If $\widetilde{x}_j$ has the LSB $\ell_{x,j}$, we have $|\widetilde{x}_j - x_j| < 2^{\ell_{x,j}}$, and the term $(\widetilde{x}_j - x_j)\widetilde{\sigma}_{j+1}$ can be bounded as $|\widetilde{\sigma}_{j+1}(\widetilde{x}_j - x_j)| \leq 2^{m_{\sigma,j+1}} \cdot 2^{\ell_{x,j}}$ where $m_{\sigma,j+1}$ is the MSB of $\widetilde{\sigma}_{j+1}$. Balancing this error term and $\varepsilon_{\pi}(j)$ yields $2^{m_{\sigma,j+1}} \cdot 2^{\ell_{x,j}} = 2^{\ell_{\sigma}}$. This defines $\ell_{x,j} =$

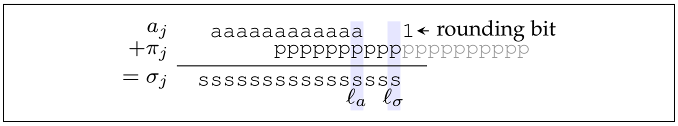

Fig. 8. Fixed-point alignment of one Horner step

$\ell_\sigma - m_{\sigma,j+1}$. In practice, it is a truncation only if $\ell_{x,j} < \ell_x$. Otherwise, x is not truncated and this first error term is zero. In the typical case where the MSB of a_j (hence σ_j) grows as j decreases (Figure 6), truncation of x happens in the earlier Horner steps.

A word now on the truncation of the product. If the hardware (for instance a fixed-size FPGA DSP block) computes the full product $\widetilde{x}_j\widetilde{\sigma}_{j+1}$, rounding it to $\widetilde{\pi}_j$ is almost for free: it can be achieved by extending the addition $a_j + \widetilde{\pi}_j$ one bit on the right to add a constant bit before truncation to ℓ_σ (Figure 8). In this case $\overline{\varepsilon}_\pi = 2^{\ell_\sigma - 1}$. Otherwise, a cheaper alternative is to use an ad-hoc `FixMultAdd` operator which combines a faithful truncated multplier and an adder in a single compression tree. In this case $\overline{\varepsilon}_\pi = 2^{\ell_\sigma}$.

Algorithm 2 Determining Horner evaluator parameters

```
procedure HORNERSIZING($\ell_x, \{m_{a,j}\}, \ell_a, \overline{\varepsilon}_{\text{eval}}^{\text{target}}$)
    $\{m_{\sigma,j}\}$ = COMPUTEHORNERMSBS()
    $\ell_\sigma \leftarrow \ell_a$
    repeat
        $\overline{\varepsilon}_{\text{eval}} = 0$
        for $i = d - 1$ down to $0$ do
            $\ell_{x,j} \leftarrow \max(\ell_x,\ \ell_\sigma - m_{\sigma,j+1})$
            if $\ell_{x,j} \neq \ell_x$ then                          ▷ if $x$ truncated
                $\overline{\varepsilon}_{\text{eval}} = \overline{\varepsilon}_{\text{eval}} + \overline{\varepsilon}_x$             ▷ add its error
            end if
            $\overline{\varepsilon}_{\text{eval}} = \overline{\varepsilon}_{\text{eval}} + \overline{\varepsilon}_\pi$                ▷ $\pi_j$ always truncated
        end for
        if $\overline{\varepsilon}_{\text{eval}} > \overline{\varepsilon}_{\text{eval}}^{\text{target}}$ then              ▷ not accurate enough:
            $\ell_\sigma \leftarrow \ell_\sigma - 1$                          ▷ need more LSB bits
        end if
    until $\overline{\varepsilon}_{\text{eval}} \leq \overline{\varepsilon}_{\text{eval}}^{\text{target}}$
end procedure
```

5 Results and conclusion

The techniques described in this article lead to consistently improved architectures with respect to those previously published. Table 1 shows typical improvements of coefficient sizes and multiplier sizes with respect to the previous state of the art: FloPoCo 2.3.1, which was used to write [11].

TABLE 1
The proposed method leads to smaller data paths.

Version	Table	Multipliers
$f(x) = 0.5\sqrt{1+x}$, $\ell_{\text{in}} = \ell_{\text{out}} = -23$, $d = 2$		
2.3.1	256×(27+19+10)	16×20, 16×10
Present	64×(26+17+9)	17×18, 10×10
$f(x) = 0.5\sqrt{1+x}$, $\ell_{\text{in}} = \ell_{\text{out}} = -52$, $d = 4$		
2.3.1	256×(56+46+36+28+19)	44×49, 35×37, 25×29, 25×19
Present	256×(56+44+33+23+14)	44×47, 36×36, 26×26, 15×15
$f(x) = \log(1+x)$, $\ell_{\text{in}} = \ell_{\text{out}} = -23$, $d = 2$		
2.3.1	256×(27+21+13)	16×22, 16×13
Present	128×(26+18+9)	16×20, 10×10
$f(x) = \log(1+x)$, $\ell_{\text{in}} = \ell_{\text{out}} = -52$, $d = 4$		
2.3.1	512×(56+49+40+32+23)	44×50, 44×41, 35×33, 25×23
Present	512×(56+46+35+24+14)	43×48, 37×37, 26×26, 15×15

All the architectures reported there were tested for last-bit accuracy thanks to the FloPoCo test framework [13].

However the main contribution of this work is a much simpler view of the problem, thanks to normalization to unit intervals and systematic balancing of errors. This lead to much cleaner and more robust open-source code.

This is a solid foundation on which to explore other range reductions [10], other evaluation schemes [12], [17], and FPGA-specific optimizations based on fixed-size memory and DSP blocks. Also, we have used the phrase "near optimal" in the introduction: it is usually still possible to shave one more bit here and there by relaxing the constraint that all the degrees share the same LSB. Is it worth the increased code complexity?

Acknowledgments to the Sollya team (N. Brisebarre, S. Chevillard, M. Joldes, Ch. Lauter), and to D. Thomas and J.M. Muller for discussions on this subject. This work was supported by the ANR INS MetaLibm project.

References

[1] R. Cheung, D.-U. Lee, W. Luk, and J. Villasenor, "Hardware generation of arbitrary random number distributions from uniform distributions via the inversion method," *IEEE Transactions on VLSI Systems*, vol. 8, no. 15, 2007.

[2] F. de Dinechin, M. Istoan, and G. Sergent, "Fixed-point trigonometric functions on FPGAs," *SIGARCH Computer Architecture News*, vol. 41, no. 5, pp. 83–88, 2013.

[3] F. de Dinechin and M. Istoan, "Hardware implementations of fixed-point Atan2," in *22nd Symposium of Computer Arithmetic*. IEEE, Jun. 2015.

[4] F. de Dinechin and B. Pasca, "Floating-point exponential functions for DSP-enabled FPGAs," in *Field Programmable Technologies*, Dec. 2010, pp. 110–117.

[5] F. de Dinechin, P. Echeverría, M. López-Vallejo, and B. Pasca, "Floating-point exponentiation units for reconfigurable computing," *Transactions on Reconfigurable Technology and Systems*, vol. 6, no. 1, 2013.

[6] D. B. Thomas, "A general-purpose method for faithfully rounded floating-point function approximation in FPGAs," in *22d Symposium on Computer Arithmetic*. IEEE, 2015.

[7] N. Kapre and A. DeHon, "Accelerating SPICE model-evaluation using FPGAs," *Field-Programmable Custom Computing Machines*, pp. 37–44, 2009.

[8] F. de Dinechin and B. Pasca, *High-Performance Computing using FPGAs*. Springer, 2013, ch. Reconfigurable Arithmetic for High Performance Computing, pp. 631–664.

[9] D. Lee, A. Gaffar, O. Mencer, and W. Luk, "Optimizing hardware function evaluation," *IEEE Transactions on Computers*, vol. 54, no. 12, pp. 1520–1531, 2005.

[10] D.-U. Lee, P. Cheung, W. Luk, and J. Villasenor, "Hierarchical segmentation schemes for function evaluation," *IEEE Transactions on VLSI Systems*, vol. 17, no. 1, 2009.

[11] F. de Dinechin, M. Joldes, and B. Pasca, "Automatic generation of polynomial-based hardware architectures for function evaluation," in *Application-specific Systems, Architectures and Processors*. IEEE, 2010.

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

[13] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, Jul. 2011.

[14] N. Brisebarre and S. Chevillard, "Efficient polynomial L^∞- approximations," in *18th Symposium on Computer Arithmetic*. IEEE, 2007, pp. 169–176.

[15] S. Chevillard, M. Joldeş, and C. Lauter, "Sollya: An environment for the development of numerical codes," in *Int. Conf. on Mathematical Software*, ser. LNCS, vol. 6327. Springer, 2010, pp. 28–31.

[16] S. Chevillard, M. Joldes, and C. Lauter, "Certified and fast computation of supremum norms of approximation errors," in *19th Symposium on Computer Arithmetic*. IEEE, 2009, pp. 169–176.

[17] J. Detrey and F. de Dinechin, "Table-based polynomials for fast hardware function evaluation," in *Application-specific Systems, Architectures and Processors*. IEEE, 2005, pp. 328–333.