A general-purpose method for faithfully rounded floating-point function approximation in FPGAs

David B. Thomas
Dept. of Electrical and Electronic Engineering
Imperial College London
dt10@imperial.com

***Abstract*—A barrier to wide-spread use of Field Programmable Gate Arrays (FPGAs) has been the complexity of programming, but recent advances in High-Level Synthesis (HLS) have made it possible for non-experts to easily create floating-point numerical accelerators from C-like code. However, HLS users are limited to the set of numerical primitives provided by HLS vendors and designers of floating-point IP cores, and cannot easily implement new fast or accurate numerical primitives. This paper presents a method for automatically creating high-performance pipelined floating-point function approximations, which can be integrated as IP cores into numerical accelerators, whether derived from HLS or traditional design methods. Both input and output are floating-point, but internally the function approximator uses fixed-point polynomial segments, guaranteeing a faithfully rounded output. A robust and automated non-uniform segmentation scheme is used to segment any twice-differentiable input function and produce platform-independent VHDL. The approach is demonstrated across ten functions, which are automatically generated then placed and routed in Xilinx devices. The method provides a 1.1x-3x improvement in area over composite numerical approximations, while providing similar performance and significantly better relative error.**

Keywords. Function Approximation, Floating-Point, Faithful Rounding, FPGA.

I. Introduction

FPGAs are now used to accelerate complex numerical algorithms, often using complex data-flow graphs of floating-point operations. Existing work on floating-point function approximations has looked for resource efficient methods of accurately evaluating common functions, such as the exponential [7] and logarithm [8], but each method is specific to a particular function. When FPGA designers are faced with a function which is not available as an optimised primitive, they are forced to use numerical approximations, which require significant area, have high latencies, and unknown error properties.

This paper presents a method for approximating any twice-differentiable function, including those with local minima and maxima. The resulting tool is completely automated, and produces faithfully rounded pipelined floating-point operators. This allows designers to produce hardware operators for functions which have not yet been hand optimised, and to replace composite uni-variate functions containing multiple rounding-errors with a single faithfully rounded operator. The main contributions of this paper are:

- A generic hardware architecture for pipelined floating-point function approximations, which uses non-linear segmentation and a fixed-point polynomial evaluator to provide faithful evaluation.
- A segmentation algorithm which reliably and automatically partitions the floating-point input domain of a function into segments which can be approximated with fixed-point polynomials of low degree.
- An evaluation of the architecture and segmentation algorithm over ten target functions, comparing post-place and route area and performance in Virtex-6 against reference implementations.

The C++ source code for the tool is freely available as part of the FloPoCo [1] framework.

II. Problem Statement

We will work with the extended set of real numbers $\overline{\mathbb{R}}$, which contains the positive and negative reals, as well as signed zeros, signed infinities, and NaN (not a number):

$$\begin{aligned}\overline{\mathbb{R}} = \{\mathrm{NaN}, -\infty\} \cup \{x : x \in \mathbb{R} \wedge x < 0\} \cup \{-0, +0\} \\ \cup \{x : x \in \mathbb{R} \wedge 0 < x\} \cup \{+\infty\}\end{aligned}$$

The extended reals are given a total order, which defines $\mathrm{NaN} < -\infty$ and $-0 < +0$.

A floating-point representation $F \subset \overline{\mathbb{R}}$ is defined using three parameters: $w_f(F)$, the number of explicit fractional bits; $w_e(F)$, the number of exponent bits; and $e_-(F)$, the minimum exponent. The number format F then consists of the special numbers $(\mathrm{NaN}, \pm 0, \pm\infty)$, plus regular numbers composed of sign, exponent, and significand:

$$\begin{aligned}F = \{\pm 0, \pm\infty, \mathrm{NaN}\} \cup \{s \times 2^e \times (1+f)\} \quad \text{where} \\ s \in \{-1, +1\} \\ e \in \mathbb{Z} \wedge \ e_-(F) \le e \le e_+(F) \\ 2^{w_f(F)} f \in \mathbb{Z} \wedge 0 \le f < 1\}\end{aligned}$$

Here the normalised significand will be considered as $(1+f) \in [1, 2)$ with f representing the explicit bits from $[0, 1)$. This definition follows the FloPoCo numbers format [1] which is often used in FPGAs, rather than IEEE; the main difference is the lack of denormal numbers.

We also separate a floating-point representation into binades, where each binade contains all consecutive numbers with the same exponent, with one binade for each of the special numbers, and binades associated with each distinct exponent:

$$\mathrm{bin}(x) = \begin{cases} x & \text{if } x \in \{\mathrm{NaN}, -\infty, -0, +0, +\infty\} \\ (\mathrm{sign}(x), \mathrm{expnt}(x)) & \text{otherwise} \end{cases}$$

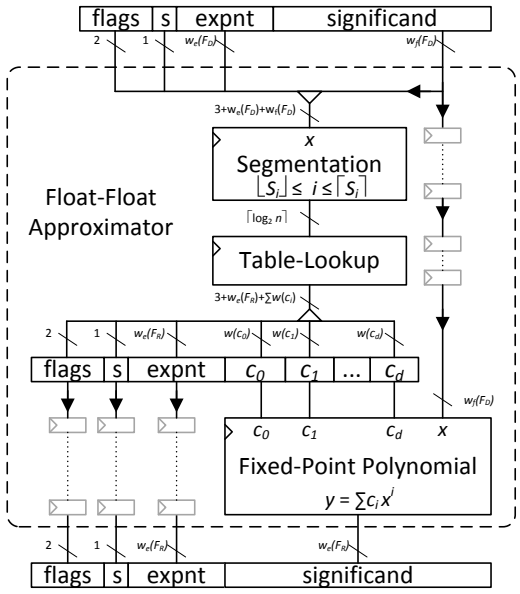

Fig. 1. Overview of the function approximation architecture.

There is a total order on the binades, following the order on the contained floating-point numbers.

The function approximation problem requires a target function $f_t : \overline{\mathbb{R}} \mapsto \overline{\mathbb{R}}$ defined by the user, and produces an approximation $f_a : F_D \mapsto F_R$ over some floating-point domain F_D and range F_R. Faithful rounding requires:

$$\forall x \in F_D : f_a(x) \in \{\mathrm{up}_{F_R}(f_t(x)), \mathrm{down}_{F_R}(f_t(x))\}$$

where $\mathrm{up}_F(x)$ returns x or the next representable number in F less than x, and $\mathrm{down}_F(x)$ returns x or the next representable number in F greater than x. At the boundaries of representable numbers, one or both of the numbers may be a special value, such as ∞ or Nan. The function f_t must be a total function over the input domain, and f_a will also be a total function. If the target function is only defined or needed over a subset of the domain, the user can maps other regions to NaN. The function must be twice-differentiable, and singularities are allowed at special numbers (zeros and infinities), as long as the function is well-behaved for regular inputs.

The first goal of this work is that the approximation should be automated and reliable, so a non-expert user should have confidence that an approximation will be produced in a reasonable amount of time. The second goal is that the approximation should be faithful and complete, over the entire input domain, so users should not need to understand or describe accuracy trade-offs.

III. Overall Approach

The approach suggested here is to partition the input domain $x \in F_D$ into n segments $S_1..S_n$ at compile time, then split the run-time calculation of $y = f_a(x)$ into three stages:

1) Use a segmentation function $s(.)$ to map the input $x \in F_D$ to an integer $i = s(x)$, where $i \in 1..n$ identifies which of n segments contains the input.
2) Evaluate a fixed-point degree-d polynomial $y_f = p(x_f, c_i)$, where $c_1..c_n$ each contain $d+1$ coefficients for segment i, mapping the input significand to the output significand.
3) Attach an exponent and sign or special number flag, producing $y = a_i(y_f)$.

The resulting hardware architecture is shown in Figure 1.

Note that Step 2 only depends on i and x_f, while Step 3 only depends on i and y_f, meaning that only the segment and the significand are needed after Step 1. To achieve this we need to choose the segments such that the only thing that varies within a given segment is the significand, and that the mapping $y_f = p(x_f)$ can be approximated with a polynomial of a chosen degree. The process for defining these segments will first be described, then the approximation architecture given a set of segments is shown.

IV. Segmentation Algorithm

We partition the input domain F_D into n segments $S_1..S_n$, where each segment is a contiguous subset of F_D and can be identified by its lower-bound $\lfloor S_i \rfloor$ and upper-bound $\lceil S_i \rceil$:

$$S_i = \{x : x \in F_D \wedge \lfloor S_i \rfloor \leq x \leq \lceil S_i \rceil\}$$

with the segments forming a dis-joint covering of the input domain. We will use the notation $\{a...b\}$ to represent a segment with $\lfloor \{a...b\} \rfloor = a$ and $\lceil \{a...b\} \rceil = b$.

Out approach uses uses the following steps, each of which ensures stricter properties on the segmentation.

1) Make all segments flat (contained within a single binade) in their domain.
2) Make all segments monotonically increasing, decreasing, or constant in the segment image.
3) Make all segments flat in their image.
4) Split segments until they can be approximated by a polynomial of degree at most d.
5) Split segments until they can be faithfully approximated by a fixed-point polynomial of degree at most d.

Each stage may result in one or more of the current segments being split, so the total segment count will never decrease. Certain target functions will cause more growth in different stages – for example, periodic and multi-modal functions such as $\sin(.)$ will experience growth in stage 2, many functions will grow in stage 4, while singularities such as $1/x$ may experience some growth in stage 5. The intent and execution of each stage will now be described.

A. Flatten segment domains

The starting point is the single segment $\mathrm{NaN}\,..+\infty$, which includes every binade in the domain. In this stage the single segment is replaced with one segment for each binade:

$$\begin{aligned}\mathbf{S}_1 = &\{\{\mathrm{NaN}\}, \{-\infty\}, \{-0\}, \{+0\}, \{+\infty\}\} \\ &\cup \left\{\left\{-2^e\left(2 - 2^{-w_f(F_D)}\right) ... - 2^e\right\} : e \in e_-(F_D)..e_+(F_D)\right\} \\ &\cup \left\{\left\{2^e ... 2^e\left(2 - 2^{-w_f(F_D)}\right)\right\} : e \in e_-(F_D)..e_+(F_D)\right\}\end{aligned}$$

Ensuring that each segment only covers one binade of the domain means that once an input has been assigned to a

segment, only the significand of the input is needed for further processing. In principle, this means that there will be $5 + 2^{w_e(F_D)+1}$ segments after this stage.

For practical purposes, this stage is also used to reflect explicit directions from the user about what input interval is interesting to them. For example, if a user specifies that only non-negative inputs will be used, the target function is modified to return NaN for all negative segments. These can all then be optimised into one large segment that returns NaN.

B. Ensure Monotonicity

Flattening the domain ensures the input doesn't cross binade boundaries, and if the output also doesn't cross binade boundaries then a significand-to-significand fixed-point polynomial can be used to approximate the segment function. However, as arbitrary (twice-differentiable) input functions are allowed, the segment output image must first be restricted to monotonically increasing or decreasing ranges. This will allow the next stage to reliably determine where the segment image intersects the range binade boundaries.

Segments covering special (NaN etc.) inputs will have constant outputs, so only segments covering regular inputs need to be considered. The algorithm which transforms the flat segments $\mathbf{S}_1$ to the monotonic segments $\mathbf{S}_2$ is shown in Algorithm 1. This assumes the existence of a function:

$$\mathrm{roots} : (\overline{\mathbb{R}} \mapsto \overline{\mathbb{R}}) \times \overline{\mathbb{R}} \times \overline{\mathbb{R}} \mapsto \mathcal{P}(\overline{\mathbb{R}})$$

which returns a set of the locations of all real roots within a given interval.

Algorithm 1 EnsureMonotonicity

1: $\mathbf{S}_2 \leftarrow \emptyset$
2: **for** $S \in \mathbf{S}_1$ **do**
3: **if** $\mathrm{special}(S)$ **then**
4: $\mathbf{S}_2 \leftarrow \mathbf{S}_2 \cup \{S\}$
5: **else**
6: $Z \leftarrow \mathrm{roots}(f'_t, \lfloor S \rfloor, \lceil S \rceil)$
7: **for** $z \in Z$ **do**
8: $z_d \leftarrow \mathrm{down}_{F_D}(z)$ {Round down to $z_d \in F_D$}
9: **if** $z_d \in S$ **then**
10: $\mathbf{S}_2 \leftarrow \mathbf{S}_2 \cup \{\{\lfloor S \rfloor ... z_d\}\}$
11: $S \leftarrow \{x : x \in S \wedge x > z_d\}$ {May be empty}
12: **end if**
13: **end for**
14: **if** $S \neq \emptyset$ **then** {No roots, or left-overs}
15: $\mathbf{S}_2 \leftarrow \mathbf{S}_2 \cup \{S\}$
16: **end if**
17: **end if**
18: **end for**

This approach is overly conservative, as it will split segments which contain a minima or maxima, but are still completely contained within one output binade. For example, if we choose $f_t(x) = 10\sin(x)/9$, it will correctly split the segment $\{0.5...1\}$ at $\pi/4$, as the image crosses between two binades in the range. However, with $f_t(x) = 9\sin(x)/10$ the output maximum at $\pi/4$ occurs entirely within one range binade, so splitting the segment is not strictly necessary.

C. Flatten Segment Images

The segments are now known to be contained within one binade, and the segment images are monotonic. This allows the segments to be safely split by Algorithm 2, which simply walks across each segment domain, and splits it at any points where the segment image crosses a binade boundary in the range.

Algorithm 2 FlattenImages

1: $\mathbf{S}_3 \leftarrow \emptyset$
2: **for** $S \in \mathbf{S}_2$ **do**
3: **while** $\mathrm{bin}(\lfloor S \rfloor) \neq \mathrm{bin}(\lceil S \rceil)$ **do**
4: $r \leftarrow \max\{x : x \in S \wedge \mathrm{bin}(\lfloor S \rfloor) = \mathrm{bin}(f_t(x))\}$
5: $\mathbf{S}_3 \leftarrow \mathbf{S}_3 \cup \{\{\lfloor S \rfloor ... r\}\}$
6: $S \leftarrow \{x : x \in S \wedge r < x\}$
7: **end while**
8: $\mathbf{S}_3 \leftarrow \mathbf{S}_3 \cup \{S\}$
9: **end for**

This algorithm doesn't depend on whether the segment image is increasing or decreasing, nor whether there are discontinuities in the initial segment images. Even if there are no discontinuities in f_t, the discreteness of F_D and F_R may mean that the output jumps by more than one output binade, so there is no requirement that adjacent segments have contiguous range binades. The maximum on line 4 can be implemented using bisection search over the segment input, taking $O(w_f(F_D))$ time, so is fast even for large significand widths.

D. Split to real-coefficient polynomials

The segments in $\mathbf{S}_3$ are now flat in both the domain and range, so all inputs for a segment are in the same domain binade, and all outputs are in the same range binade. This allows the problem to be recast from a floating-point approximation problem to a fixed-point approximation problem. Special segments in the domain map to constants, and any special segments in the range are special (constant) for the entire segment, so polynomial approximations are only needed for regular input and output binades.

Given a segment S, we can define a function f_S which directly maps an input significand to an output significand.

$$f_S : [0, 1) \mapsto [0, 1)$$
$$f_S(x) = s_r \times 2^{-e_r} \times f_t(s_d \times 2^{e_d} \times (1 + x)) - 1 \text{ where}$$
$$(s_d, e_d) = \mathrm{bin}(\lfloor S \rfloor) \quad (s_r, e_r) = \mathrm{bin}(f_t(\lfloor S \rfloor))$$

The transform function is not necessarily unique to a given segment, and will be shared by any segments with the same input and output binades.

Given the transformed function, the segments will be split as necessary until all segments can be faithfully approximated via a polynomial of degree d or less. This relies on a polynomial approximation function:

$$\mathrm{approx} : (\overline{\mathbb{R}} \mapsto \overline{\mathbb{R}}) \times \overline{\mathbb{R}} \times \overline{\mathbb{R}} \times \mathbb{N} \mapsto \overline{\mathbb{R}}^{\infty} \times \overline{\mathbb{R}}$$
$$\mathrm{approx}(f, a, b, d) \to (c, \epsilon)$$

The input consists of: a function f, approximation interval $[a, b]$, and maximum degree d; and returns a tuple of coefficients c, plus the maximum error ϵ. In an abuse of notation, the coefficients are an infinite set from $c_0, c_1, ..., c_\infty$, and the function ensures that:

$$\epsilon \geq \left|\left| f(x) - \sum_{i=0}^{\infty} c_i x^i \right|\right|_{\infty, x \in [a,b]} \quad \wedge \ \forall i > d : c_i = 0 \qquad (1)$$

Exactly what approximation is performed is left unspecified – a minimax approximation will guarantee the smallest ϵ, but for practical purposes it is sometimes useful to fall back on a more robust method which converges. The approximation algorithm (both theoretical and the implementation described later) only requires that the bound is correct, and is entirely content if $\mathrm{approx}(.)$ only returns linear approximations.

The $\mathrm{approx}(.)$ function is used by Algorithm 3 to adaptively split the segments until all segments can be faithfully rounded. Currently a recursive binary splitting approach is used, which is not guaranteed to provide an optimal split, but which is simple and reliable.

Algorithm 3 SplitPolynomials

```
1:  T ← S_3
2:  S_4 ← ∅
3:  for S ∈ T do
4:      T ← T/S
5:      (poly, err) ← approx(f_S, ⌊S⌋, ⌈S⌉, d)
6:      if err ≤ 2^{-w(F_R)-3} then
7:          S_4 ← S_4 ∪ {S}
8:      else
9:          m ← down_{F_D}((⌊S⌋ + ⌈S⌉)/2)
10:         T ← T ∪ {{⌊S⌋...m}}
11:         T ← T ∪ {{up_{F_D}(m)...⌈S⌉}}
12:     end if
13: end for
```

The temporary set $\mathbf{T}$ can be viewed as a list or queue, and in practise the splitting at line 9 is well handled using recursion. There can be at most $w_f(F_D)$ levels of recursion, so stack overflow is not a danger.

The splitting condition on line 6 is deliberately conservative, and requires a polynomial that is more than faithful. This is an empirical tradeoff between the time for (potentially sub-optimal) approximation over the reals, and the more expensive approximation over fixed-point numbers in the next stage. It results in more segments than are strictly necessary, but results in a better (faster) user experience.

E. Split to concrete polynomials

The final stage in the approximation algorithm is to ensure that each segment can be faithfully approximated by a polynomial with fixed-point coefficients. This requires a more complex function $\mathrm{fixapprox}$ which provides the same behaviour as approx, but takes an extra parameter l defining the least-significant-bit of each coefficient:

$$\mathrm{fixapprox} : (\overline{\mathbb{R}} \mapsto \overline{\mathbb{R}}) \times \overline{\mathbb{R}} \times \overline{\mathbb{R}} \times \mathbb{N} \times \mathbb{Z}^\infty \mapsto \overline{\mathbb{R}}^\infty \times \overline{\mathbb{R}}$$
$$\mathrm{fixapprox}(f, a, b, d, l) \to (c, \epsilon)$$

This ensures the same properties on ϵ and c as approx, but also ensures that coefficient c_i is represented as a fixed-point number with l_i fractional bits:

$$\forall i \geq 0 : 2^{l_i} c_i \in \mathbb{Z}$$

Choosing the LSBs for each coefficient requires some sort of heuristic, and this work appeals to that of [2]. They propose that for an input interval $[0, 1)$ split into k equal ranges, and a target output LSB p, the coefficients should have LSBs $l_i = 2^{-p+ik}$. In our case some segments will extend over the entire $[0, 1)$ range, so $k = 1$. Another empirical heuristic is used to encourage speed and reliability of approximation, so an extra bit is added, resulting in the approximation:

$$l_i = 2^{-w(F_R)+i-1} \qquad (2)$$

The final stage uses the segments $\mathbf{S}_4$ from the previous stage, and produces a final set of domain segments, but now augmented with the fixed-point polynomial and the error. This set of augmented segments contains all the information needed to build the concrete hardware architecture. The process is shown in Algorithm 4.

Algorithm 4 SplitConcrete

```
1:  l ← (2^{-w(F_R)-1}, 2^{-w(F_R)}, 2^{-w(F_R)+1}, ...)
2:  T ← S_4
3:  S_5 ← ∅
4:  for S ∈ T do {While segments left to process}
5:      T ← T/S
6:      i ← 0
7:      while S ≠ ∅ ∧ i ≤ d do {Find lowest degree poly}
8:          (poly, err) ← fixapprox(f_S, ⌊S⌋, ⌈S⌉, d, l)
9:          if err ≤ 2^{-w(F_R)-2} then
10:             S_5 ← S_5 ∪ {(S, poly, err)}
11:             S ← ∅
12:         else
13:             i ← i + 1
14:         end if
15:     end while
16:     if S ≠ ∅ then {Segment needs further splitting}
17:         m ← down_{F_D}((⌊S⌋ + ⌈S⌉)/2)
18:         T ← T ∪ {{⌊S⌋..m}}
19:         T ← T ∪ {{up_{F_D}(m)..⌈S⌉}}
20:     end if
21: end for
```

The while loop starting at line 7 is used to ensure that the lowest degree feasible polynomial is chosen, which has two purposes. First, it is used to decrease complexity for the $\mathrm{fixapprox}$ function, by not asking it to approximate very simple segments with high-degree polynomials. Highly linear segments are very common, for example in $\sin(x)$ close to zero, which can cause stability problems in minimax algorithms. Second, it avoids the situation where $\mathrm{fixapprox}$ *can* produce a high-degree approximation to a low-degree segment, but does so by producing exquisitely balanced polynomials. Such polynomials may result in very large high-order coefficients, drastically increasing the cost of the polynomial evaluator and coefficient tables.

The extra splitting step at line 16 is another step for reliability and usage. In principle the polynomials are already accurate

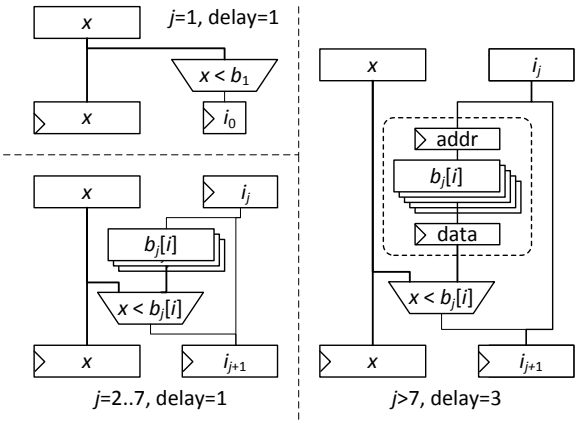

Fig. 2. Hardware implementation of the segmenter for increasing numbers of index bits, using LUTs for small j and block-RAMs for larger j.

enough, but in practise fixapprox may not be able to find a solution, either directly reporting an error, or taking excessively long. In such cases it is better to split the segment again, resulting in an extra segment and sub-optimal polynomials, but guaranteeing completion.

V. Hardware Architecture

The output of the segmentation algorithm is a set of tuples $\mathbf{S}_5 = \{S_1, S_2, ..., S_n\}$, where n depends on the approximation range, the input and output types, and the properties of the target function. Each segment is a tuple (S, p, ϵ), containing:

- S: the segment input domain, which is guaranteed not to overlap any other segment input domain.
- p: polynomial coefficients of (at most) degree d, rounded to the LSBs given in Equation 2;
- ϵ: the maximum absolute error between the polynomial output and the target function, expressed in terms of the output significand (i.e. $\epsilon \leq 2^{-w_f(F_R)}$ means it is within one ULP).

These segments can now be used to build the hardware architecture described earlier in Section III. There are only three components: segmenter; table-lookup; and polynomial evaluator. Table-lookup is straightforward, but design decisions still need to be made in the segmentation and polynomial evaluation stage.

A. Segmentation

Given an input $x \in F_D$, the segmenter needs to find the unique $i \in 1..n$ such that $\lfloor S_i \rfloor \leq i \leq \lceil S_i \rceil$. Because the segments form a complete partition of F_D, such a segment is guaranteed to exist, even if the segment just returns NaN. There is no special structure to the partition apart from the total ordering over the segments, so it takes $k = \lceil \log_2 n \rceil$ steps to locate the correct segment using binary search.

As the target is a pipelined hardware architecture, the binary search is unrolled into k stages, each of which recovers one bit of i, shown in Algorithm 5. At the start of each stage j there are $j-1$ known bits of i, which are used to select one

Algorithm 5 SearchSegments

$i_1 \leftarrow \text{lessp}(x, b_1[0])$
$i_2 \leftarrow 2i_1 + \text{lessp}(x, b_2[i_1])$
$i_3 \leftarrow 2i_2 + \text{lessp}(x, b_3[i_2])$
...
$i_k \leftarrow 2i_{k-1} + \text{lessp}(x, b_k[i_{k-1}])$

of 2^{j-1} boundary points from a table b_j. The boundary points are extracted from the segment boundaries as:

$$b_j[i] = \lfloor S_{2^{k-j}(2i+1)} \rfloor, \quad 1 \leq j \leq k, \quad 0 \leq i \leq 2^{j-1}$$

The total number of entries in the boundary tables grows linearly with the number of segments, while the number of boundary tables grows logarithmically.

In hardware each stage is implemented using a table look-up and comparison, with the area and delay cost increasing for later stages. The comparison function $\text{lessp}(.,.)$ in Algorithm 5 is derived from the ordering over the input domain F_D, and provides the new bit at each stage:

$$\text{lessp}(x, y) = \begin{cases} 1, & \text{if } x \leq y \\ 0, & \text{otherwise} \end{cases}$$

In a hardware this total order can be implemented using a case statement to order the two numbers according to the segment type, then an integer comparison of the exponent and fraction fields to resolve ordering within segment types.

Figure 2 shows the three types of stage used in the current implementation. For $j = 1$ the boundary constant can be combined with the comparator, requiring a single level of LUTs, and for $1 < j \leq 7$ it is feasible to store the boundaries in a LUT-based ROM in contemporary architectures, allowing a delay of one cycle per stage. For $j > 7$ the tables become large enough to that they must be stored in block RAM.

B. Polynomial Evaluation

The concrete segments describe n polynomials with fixed-point coefficients, all of which map from $[0, 1)$ with $w_f(F_D)$ fractional bits to $[0, 1)$ with $w_f(F_R)$ fractional bits. All polynomials share the same fractional precision for the coefficients, but both the range of the coefficients and the approximation error will vary for each segment.

The polynomial evaluator must guarantee faithful evaluation across all polynomial segments, while trying to minimimse pipeline depth and DSP+logic consumption. The approach used here follows a much simplified form of [2], which separates evaluation error into approximation, evaluation, and rounding error:

$$2^{-w_f(F_R)} \geq \epsilon_{approx} + \epsilon_{eval} + \epsilon_{round}$$

The maximum approximation error across all polynomials is:

$$\epsilon_{approx} = \max(\epsilon_1..\epsilon_n) \leq 2^{-w_f(F_R)-2} \tag{3}$$

The upper bound of $2^{-w_F(F_R)-2}$ on ϵ_{approx} is guaranteed by line 9 of Algorithm 4, while the final rounding error is $2^{-w_F(F_R)-1}$, which gives a bound on evaluation error of:

$$\epsilon_{eval} \leq 2^{-w_f(F_R)} - 2 \times 2^{-w_f(F_R)-2} = 2^{-w_f(F_R)-1}$$

Knowing the error budget and the range of the coefficients, intermediate rounding can be selected. A rather simple approach is chosen, which is to choose a number of guard-bits g, and to round each polynomial stage except the final to $-w_f F_R - i - g$ LSBs. So given the polynomial coefficients $c_0..c_d$, and an input $x \in [0, 1)$, the approximation is:

$$a_d = c_d \tag{4}$$

$$a_i = \text{round}\left(c_i + x \times a_{a+1}, 2^{-w_F(F_R)-i-g}\right) \tag{5}$$

$$a_0 = \text{round}\left(c_0 + x \times a_1, 2^{-w_F(F_R)}\right) \tag{6}$$

$$\text{where} \quad \text{round}\,(x, r) = r\,\lfloor x/r + 1/2 \rfloor \tag{7}$$

The guard bits are progressively increased from $g = 1$ up, and the first g giving faithful rounding according to the approach in Section II.D of [2] is chosen. All sources of rounding error are considered in this evaluation process, so if ϵ_{approx} can be accurately evaluated (using a tool such as Sollya), then the fixed-point segment will be faithfully rounded.

For segments which cover the first or last value within a binade there is a small chance that the faithfully rounded output will underflow or overflow, so we expect a rounded value $a_0 \in [0, 1)$, but the fixed-point evaluator returns $a_0 < 0$ or $a_0 \geq 1$. Due to the image flattening, we know that a correctly rounded output must exist within the target binade, so simply saturating the polynomial to the range $[0, 1)$ is sufficient to ensure correct rounding. These corner cases can automatically be tested by ensuring that the endpoints of domain segments are included in any test input sequences.

VI. Evaluation

The segmentation algorithm and hardware architecture have been implemented as a single combined C++ program within the FloPoCo [1] framework, which takes an approximation problem as input, and produces pipelined VHDL and a test-bench. The whole process is automated, and requires no manual assistance. The Sollya [3] library is used for expression parsing throughout, and is also used to provide $\text{approx}(.)$ using the `remez` and `uncertifiedInfNorm` functions, plus $\text{fixapprox}(.)$ using `fpminimax` and `infNorm`. MPFR [4] is used to provide all high-precision correctly rounded calculations, both indirectly via Sollya, and directly in many places.

To evaluate the performance of the function approximators, three types of functions are considered:

1) *Primitive* functions with existing hand-optimised implementations.
2) *Composite* functions which can be implemented in terms of existing hand-optimised primitives.
3) *Approximated* functions which cannot be expressed in terms of existing primitives, and must be approximated using polynomials or other techniques.

The reference library of primitive functions is taken to be those provided by FloPoCo, as a well-respected source of high-performance floating-point cores for FPGAs.

The list of functions tested is given in Table I, along with the function definitions, and an approximation interval. The restricted approximation intervals for certain operators are intended to reflect typical usage in a hardware application, based on the author's experience building such systems. For comparison purposes, the composite functions were implemented by chaining together existing FloPoCo operators, using the built-in critical path management to maintain performance.

TABLE I. Selected test functions.

Class	Name	Function	Interval
P	log	$\log(x)$	$[0, \infty]$
	exp	$\exp(x)$	$[-\infty, \infty]$
C	normpdf	$\exp(-x^2/2)/\sqrt{2\pi}$	$[-16, +16]$
	sigmoid	$1/(1+\exp(-x))$	$[-\infty, +\infty]$
	log1p	$\log(x+1)$	$[-1, +\infty]$
	expm1	$\exp(x)-1$	$[-\infty, +\infty]$
A	sin	$\sin(x)$	$[-\pi, \pi]$
	cos	$\cos(x)$	$[-\pi, \pi]$
	erf	$\text{erf}(x)$	$[-32, +32]$
	sintan	$\sin(\tan(x))$	$[-2\pi, 2\pi]$

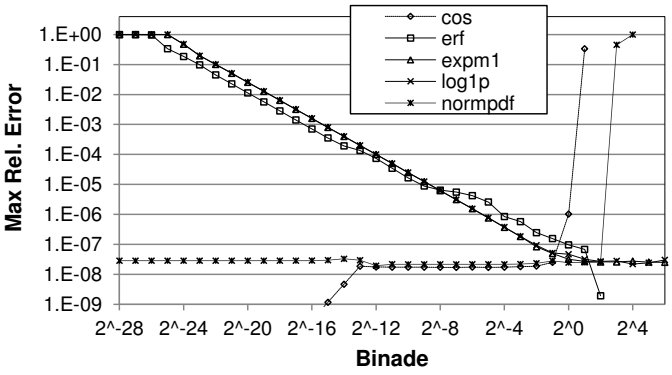

Fig. 3. Maximum relative error of composite references within positive binades for composite functions (generic approximations not shown, as they are all faithful).

For the approximate functions, a comparison point was provided by using existing published numerical approximations. This follows the process typically used by FPGA designers when faced with a function with no available hardware primitive – search the literature for an approximation which: a) contains no loops; and b) only relies on functions for which they do have hardware primitives. This paper uses approximations given in Abramowitz and Stegun [5], which is a well-known source of approximations.[1] The formulas and operation counts are given in Table II, which were implemented using FloPoCo, again using critical path management, and also taking advantage of constant multipliers and squarers where possible. To be clear, we do not claim the A&S approximations are the most accurate or most efficient, but they are typical of the solutions used by hardware engineers, who do not know about minimax tools or range-reduction methods.

One motivation for this work is that faithful rounding is important, as it ensures relative error bounds. To demonstrate how bad the relative error of the composite reference implementations can get, Figure 3 shows the maximum relative error of five of the functions within each positive input binade. As expected, the relative error of expm1 and log1p degrades

[1] The target audience of this paper knows that there are much better ways of generating approximations, as does the author. However, the target audience of the tool just want to get things done; in the absence of an off-the-shelf IP core, an A&S approximation is a solution, while a book or paper is not.

TABLE II. NUMERICAL APPROXIMATIONS USED FOR FUNCTIONS.

Function	A&S	$x \times C$	x^2	$x \times y$	$x + C$	$1/x$	$\exp(x)$
sin	4.3.97	1	1	5	5		
cos	4.3.99	1	1	4	5		
erf	7.1.26	2		5	7	1	1

dramatically as the input approaches zero, with erf following a similar degradation. In contrast, cos shows poor error for large inputs, due to naive range reduction for inputs in the range $[\pi/2, \pi]$, while normpdf loses accuracy due to rounding when squaring the input. The generic methods proposed here are not shown, as they are simply staight lines showing faithful rounding.

VHDL implementations were generated for each target, using the generic approximation method for $d = 3$ and $d = 4$, and the reference method (primitive, composite, or approximate). The target platform was set as Virtex-6 with a target of 400MHz for all generated functions. The generic implementations were then all tested to ensure they were all faithful. Exhaustive testing has been applied to all operators for half-precision $w_e(F_D) = w_e(F_R) = 5$, $w_f(F_D) = w_f(F_R) = 10$, with no errors observed. Single precision operators have been tested using: 1000 points evenly spaced within each input binade; 100000 points evenly spaced over the input interval; 100 points evenly spaced within each domain segment, including $\lfloor S \rfloor$ and $\lceil S \rceil$.

This testing is not intended to check whether the construction process is correct, it is only to check for coding or synthesis errors in the concrete tools. The process as given *should* be faithful by construction, as all floating-point calculations during construction are performed using either MPFR [4] with correct rounding, or Sollya [3], which uses interval arithmetic to control evaluation error. This includes all calculations related to segmentation, including root finding and function minimisation, and the bounding of the maximum approximation across all polynomials. This results in a conservative upper bound on the maximum polynomial error ϵ_{approx} in Equation 3, which is then taken into account while determining the number of guard bits needed in Equation 7 for faithful rounding.

A number of heuristics are introduced in the process desribed in this paper, such as the deliberate over-segmentation in Section IV-D and Section IV-E, but these are only aimed at achieving good compile-time performance and reliability - the result should always be faithful. As a tool for FPGA users, the emphasis is taking zero designer effort and a small amount of tool/compiler time, rather than an extremely optimal process which may take many hours and need manual tweaking. The worst-case would be failing with an error message that "remez failed to converge", or "function does not oscillate", as the user would not be able to take any meaningful action.

All implementations were synthesised, placed, and routed, using Xilinx ISE 14.3. The target device was a Virtex-6 xc6vcx130t-ff1156-2 part. Re-timing was enabled during synthesis, and gobal optimisation during mapping, but apart from that the defaults were used, and the same toolchain settings were used for both generic approximations and reference components.

Table III summarises the resulting single-precision implementations, both in terms of resources, and performance. Numbers in brackets measure the generic approximations relative to the reference, and bolded numbers indicate the best implementation for each category. For the first two primitive comparisons (log and exp) the generic approximations are clearly worse on every metric, but this is expected as the reference primitives take advantage of range-reductions, and have been heavily optimised. However, the approximation for $\log$ with $d = 3$ is within a factor of 3 of the primitive in all metrics except for DSPs.

For the next four comparisons (expm1, log1p, sigmoid, normpdf) the generic approximations are a little closer to the composite references, and for the sigmoid actually beat the composite in terms of LUT and FF resources due to the logic-based divider. However, the generic approximation is producing faithfully rounded results, while all of the composite functions have known regions of relative inaccuracy.

The last group of comparisons (erf, cos, sin, tansin) are the main target for this work: functions which have no direct composite implementation, and currently rely on an engineer to find some numerical approximation. Now the approximations are better on every metric except for block-RAMs – the numerical approximations rely heavily on polynomials, and so use little RAM. erf in particular shows substantial savings, requiring very few resources; apart from the latency, it requires resources similar to the hand-optimised implementation of log.

VII. RELATED WORK

Function approximation has been an active topic of research for many years, both for software and hardware. The central concepts of range-reduction, approximation, and reconstruction are well established [6], especially for floating-point inputs in both software and hardware. This section will focus on work specifically related to pipelined implementations of function approximation for use in FPGAs.

Most work has focussed on approximating known common floating-point functions, such as the exponential [7] and logarithm [8]. These use significant function-specific knowledge, both in the range-reduction and approximation stages, and are important building blocks both for manual FPGA design, and for HLS derived designs.

General purpose function approximations for hardware are less commonly used in FPGAs, and is mostly used in the fixed-point domain. One example is the approach used in [9], which uses a similar approach to that used here, using a non-uniform segmentation of the fixed-point input to select faithfully rounded polynomial segments. The segmentation is more restricted than the approach used here, allowing for a much more efficient mapping of input to segment, but restricting the ability to place polynomials exactly where needed.

Another general purpose fixed-point function evaluator is provided by FloPoCo [2], which uses a uniform segmentation of the input to make segment selection a simple table-lookup. This approach is unsuitable as a general-purpose function approximation, but is ideal for the relatively smooth functions found within a range-reduced approximation problem.

Altera have produced a large number of floating-point functions for their OpenCL HLS tool, covering all the functions

Function	Method	LUTs		FFs		DSPs		BRAMs		Cycles		MHz		Lat. (ns)	
exp	$d=3$	1372	(3.37)	1754	(4.34)	9	(9)	24	(24)	36	(3.6)	166	(0.63)	216	(5.68)
	$d=4$	1882	(4.62)	2456	(6.08)	15	(15)	15	(15)	42	(4.2)	188	(0.71)	224	(5.89)
	Prim	**407**	-	**404**	-	**1**	-	**1**	-	**10**	-	**264**	-	**38**	-
log	$d=3$	1376	(1.72)	1762	(2.95)	10	(2)	12	(12)	34	(2.62)	188	(1.22)	181	(2.15)
	$d=4$	1618	(2.03)	2068	(3.46)	13	(2.6)	7	(7)	40	(3.08)	200	(1.3)	200	(2.38)
	Prim	**798**	-	**597**	-	**5**	-	**1**	-	**13**	-	**154**	-	**84**	-
expm1	$d=3$	1304	(1.79)	1718	(2.23)	9	(9)	12	(12)	35	(1.94)	202	(0.78)	173	(2.47)
	$d=4$	1876	(2.57)	2419	(3.14)	15	(15)	7	(7)	41	(2.28)	134	(0.52)	306	(4.37)
	Comp	**730**	-	**770**	-	**1**	-	**1**	-	**18**	-	**259**	-	**70**	-
log1p	$d=3$	2203	(1.92)	3047	(3.14)	17	(3.4)	10	(10)	36	(1.89)	168	(1.1)	214	(1.73)
	$d=4$	4020	(3.5)	4974	(5.12)	28	(5.6)	15	(15)	48	(2.53)	**207**	(1.36)	232	(1.87)
	Comp	**1148**	-	**971**	-	**5**	-	**1**	-	**19**	-	153	-	**124**	-
sigmoid	$d=3$	**1259**	(0.81)	**1607**	(0.89)	9	(9)	5	(5)	**34**	(1)	206	(0.9)	165	(1.11)
	$d=4$	1559	(1.01)	1984	(1.09)	12	(12)	6	(6)	40	(1.18)	198	(0.87)	202	(1.36)
	Comp	1548	-	1814	-	**1**	-	**1**	-	**34**	-	**228**	-	149	-
normpdf	$d=3$	1498	(1.77)	1966	(3.12)	11	(2.75)	25	(25)	36	(2)	190	(0.9)	190	(2.21)
	$d=4$	1968	(2.33)	2513	(3.98)	15	(3.75)	16	(16)	42	(2.33)	178	(0.85)	236	(2.74)
	Comp	**846**	-	**631**	-	**4**	-	**1**	-	**18**	-	**210**	-	**86**	-
erf	$d=3$	**881**	(0.19)	**1064**	(0.26)	**6**	(0.55)	4	(4)	**31**	(0.44)	208	(1.1)	**149**	(0.4)
	$d=4$	1001	(0.22)	1289	(0.32)	8	(0.73)	5	(5)	37	(0.53)	**213**	(1.12)	174	(0.47)
	Approx	4627	-	4062	-	11	-	**1**	-	70	-	189	-	370	-
cos	$d=3$	**1220**	(0.47)	**1561**	(0.75)	**9**	(0.82)	5	(∞)	**32**	(0.68)	160	(0.87)	**200**	(0.78)
	$d=4$	1498	(0.57)	1889	(0.91)	12	(1.09)	5	(∞)	39	(0.83)	**188**	(1.03)	207	(0.81)
	Approx	2608	-	2087	-	11	-	**0**	-	47	-	184	-	256	-
sin	$d=3$	**1366**	(0.52)	**1690**	(0.79)	**10**	(0.77)	5	(∞)	**33**	(0.69)	174	(0.9)	**189**	(0.76)
	$d=4$	1545	(0.58)	1914	(0.89)	12	(0.92)	6	(∞)	39	(0.81)	**203**	(1.05)	192	(0.77)
	Approx	2648	-	2142	-	13	-	**0**	-	48	-	193	-	248	-
tansin	$d=3$	**855**	-	**1016**	-	**6**	-	**3**	-	**30**	-	178	-	**169**	-
	$d=4$	991	-	1266	-	8	-	4	-	36	-	**186**	-	193	-

TABLE III. POST-PLACE AND ROUTE RESULTS FOR SINGLE-PRECISION OPERATORS IN VIRTEX-6 .

required by the OpenCL spec. Their approach is to build a flexible set of tools and architectures, such as optimised floating-point polynomial evaluators [10]. This allows them to quickly construct robust floating-point approximations, but appears to be, at least partially, a manual process.

The closest approach to this work in terms of goals appears to be [11], which takes a similar approach to non-linear segmentation, followed by fixed-point approximation. They also target automated implementation, and use a similar approach to the binade approach used here to segment the input range. However, they use a segmentation scheme that is more structured, limiting the ability to place segments, and cannot handle multi-modal functions. There is also an emphasis on efficiency over accuracy, with relatively low accuracies for single-precision approximations.

VIII. CONCLUSION

This paper presents an algorithm and architecture for approximation of floating-point functions in FPGAs. The algorithm is designed to reliably produce a faithfully rounded approximation for any twice-differentiable function, while the architecture is designed to allow for high throughput pipelined implementations. A fully automated open-source implementation of the approach has been created, suitable for use by hardware designers with little knowledge of function approximation.

Experiments in Virtex-6 using ten target functions show that while the approximations cannot compete with hand-optimised primitives (as expected), for functions composed of multiple primitives it is able to provide guaranteed faithful outputs with a 2x-3x cost in resources. However, the real strength of the method is in approximating functions which cannot be decomposed into primitives, where it shows a 1.2x-3x improvement in resource usage over numerical approximations, while providing guaranteed accuracy.

REFERENCES

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

[2] 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 (ASAP), 2010 21st IEEE International Conference on*, July 2010, pp. 216–222.

[3] 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.

[4] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding," *ACM Trans. Math. Softw.*, vol. 33, no. 2, Jun. 2007. [Online]. Available: http://doi.acm.org/10.1145/1236463.1236468

[5] M. Abramowitz and I. A. Stegun, *Handbook of Mathematical Functions*. Dover Publications, 1965.

[6] J.-M. Muller, *Elementary Functions: Algorithms and Implementation*. Secaucus, NJ, USA: Birkhauser Boston, Inc., 1997.

[7] F. de Dinechin, P. Echeverría, M. López-Vallejo, and B. Pasca, "Floating-point exponentiation units for reconfigurable computing," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 6, no. 1, pp. 4:1–4:15, May 2013. [Online]. Available: http://doi.acm.org/10.1145/2457443.2457447

[8] N. Alachiotis and A. Stamatakis, "Efficient floating-point logarithm unit for fpgas," in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, April 2010, pp. 1–8.

[9] D.-U. Lee, R. Cheung, W. Luk, and J. Villasenor, "Hierarchical segmentation for hardware function evaluation," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 17, no. 1, pp. 103–116, Jan 2009.

[10] M. Langhammer and B. Pasca, "Efficient floating-point polynomial evaluation on fpgas," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, Sept 2013, pp. 1–6.

[11] "A synthesis method of general floating-point arithmetic units by aligned partition," in *Int. Cong. CSCC*, 2008, pp. 1177–1180.