Elementary Function Implementation with Optimized Sub Range Polynomial Evaluation

Martin Langhammer, Bogdan Pasca
Altera European Technology Centre
High Wycombe, UK

***Abstract*—Efficient elementary function implementations require primitives optimized for modern FPGAs. Fixed-point function generators are one such type of primitives. When built around piecewise polynomial approximations they make use of memory blocks and embedded multipliers, mapping well to contemporary FPGAs. Another type of primitive which can exploit the power series expansions of some elementary functions is floating-point polynomial evaluation. The high costs traditionally associated with floating-point arithmetic made this primitive unattractive for elementary function implementation on FPGAs. In this work we present a novel and efficient way of implementing floating-point polynomial evaluators on a restricted input range. We show on the atan(x) function in double precision that this very different technique reduces memory block count by up to 50% while only slightly increasing DSP count compared to the best implementation built around polynomial approximation fixed-point primitives.**

***Keywords*-elementary function; primitive; power series; polynomial evaluation; Horner; floating-point; FPGA**

I. Introduction

FPGAs have now sufficient capacity for implementing large computational datapaths using floating-point arithmetic. Programming FPGAs to perform these computations requires, among many others, the existence of mathematical libraries of optimized functions, which perform well on contemporary FPGAs.

Efficient elementary function implementations must be done on a function basis. The availability of efficient primitives for building these functions contributes to their quality and reduces development time. One such primitive is fixed-point function approximation based on lookup-tables and additions. Various implementations exist (see [4] for a review) but the method scalability is questionable beyond 16 bits. A better primitive which can scale up to ≈ 30 bits is the High Order Table-Based Method (HOTBM) [5]. The fixed-point function approximation is built using a piecewise polynomial approximation. The polynomial is evaluated by evaluating each monomial using a combination of table-lookups and low-precision power-and-multiply units and targets logic-based implementations.

A primitive which better targets the embedded multipliers and memories of recent FPGAs is the fixed-point function evaluator available in FloPoCo [3] and presented extensively in [6, Ch.7]. Unlike HOTBM, this primitive uses the Horner scheme to evaluate the polynomial and performs an automatic error analysis in order to reduce multiplier size. The primitive scales to any precision and makes good use of existing FPGA resources such as memory blocks and embedded multipliers. However, its efficiency has been questioned for large precisions in [7] where a combination of a 27-bit piecewise polynomial approximation and one Newton-Raphson iteration required fewer resources for a floating-point divider implementation.

In this paper we present an alternative primitive based on a novel technique in order to facilitate the implementation of some elementary functions based on their power series. The primitive implements the floating-point Horner datapath of a truncated power series which converges to the first coefficient as $x \to 0$. The novel technique removes the costly barrel shifters of the floating-point adder normalization stages, as well as the normalization and rounding stages between operators. The allowed input will be in an interval of the form $[0, 2^{-k}]$ where the typical values for k would be in the range $8-10$. The rest of the range $[2^{-k}, 1]$, if needed for computation is handled using identities. We show here that the inverse trigonometric functions such as atan(x) either directly fit this model or can be easily restructured to fit this model.

Our synthesis results on a recent StratixV device, for double precision, show that using this primitive, atan(x) requires 50% fewer memory blocks while only slightly increasing DSP and logic consumption compared to an implementation based on a fixed-point piecewise polynomial approximation primitive.

II. Elementary function example

The IEEE-754 standard on floating-point arithmetic (revised in 2008 [1]) uses a triplet (**s**ign, **e**xponent, **f**raction) to represent a floating-point number:

$$x = (-1)^s 2^e 1.f$$

Various combinations of the exponent and fraction widths (w_E and w_F) define the formats of the IEEE-754 standard (see Table I).

All floating-point elementary functions such as the trigonometric family (sin(x), tan(x), etc.), inverse trigonometric (asin(x), atan(x), etc.), logarithm and exponentials,

Table I
FLOATING-POINT FORMATS DEFINED BY THE IEEE-754 2008 STANDARD

w_E, w_F	Name IEEE-754 2008
5, 11	half precision (binary16)
8, 23	single precision (binary32)
11, 52	double precision (binary64)
15, 112	quadruple precision (binary128)

etc. all input and output data in one of these formats. While traditional microprocessors allow high performance by hardening in silicon basic operators for single and double precision, custom formats can better exploit the flexibility of FPGAs. Hence, in what follows although the running example uses double precision, the presented techniques can be easily used for any custom precisions.

Let us take as a running example $f : \mathbb{R} \to [-\frac{\pi}{2}, +\frac{\pi}{2}]$ with $f(x) = \text{atan}(x)$. The function f is an odd function $f(-x) = -f(x)$, therefore when implementing the function, we can just focus on the $x \geq 0$ case.

Next, there are two computational branches for implementing the function:

$$\text{atan}(x) = \begin{cases} \text{atan}(x) & \text{if } x \leq 1 \\ \frac{\pi}{2} - \text{atan}(\frac{1}{x}) & \text{otherwise} \end{cases}$$

The high-level architecture of the implementation is depicted in Figure 1. The architecture requires one single computation of atan(z), common to both branches. When $x \leq 1$ the absolute value of the input is forwarded directly to this unit. For the second branch, the input to the unit is fetched from the output of the reciprocal unit. Therefore, for both branches suffices to compute atan(x) for $0 \leq x \leq 1$.

The Taylor series for atan(x) is:

$$\text{atan}(x) = x - x^3/3 + x^5/5 - x^7/7 + \ldots \quad (1)$$

and allows to approximate atan(x) $\approx x$ for $x < 2^{-\lceil w_F/2 \rceil}$.

Next, there are two classical and efficient ways for computing the value of atan(x) for $x \in [2^{-\lceil w_F/2 \rceil}, 1]$. The first method is to cast the floating-point input into an approximately $1 + w_F + \lceil w_F/2 \rceil$-bit fixed-point value and use this to drive the input of a piecewise tuned Minimax polynomial approximation unit [6, Ch.7]. A normalization stage potentially shifting left the approximation result by at most $\lceil w_F/2 \rceil$ bits is then required. This method is efficient for lower precisions but becomes very inefficient for larger precisions, due to the high polynomial approximation degrees and the increasing multiplier widths. For instance, in double precision with an input fixed-point width of 79-bits, the required polynomial degree (depending on the number of subintervals) is either 8 or 9, using in the final multiplication stage a truncated multiplier of approximately 79×79 bits. Additionally, the number of memory blocks for storing the polynomial coefficients is also very high.

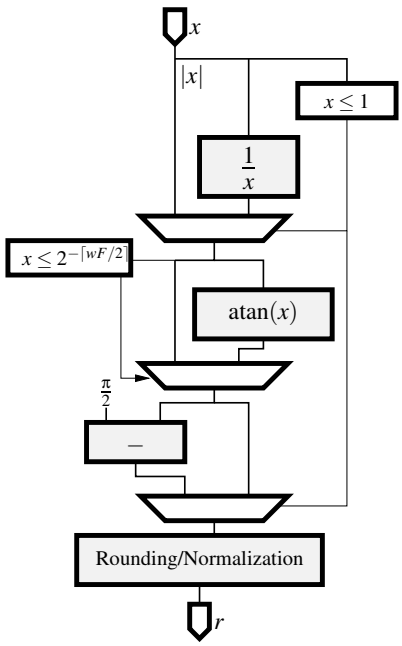

Figure 1. High-level architecture for the floating-point atan(x) function

An alternative way is to use the following rewrite:

$$\text{atan}(x) = x \left(\frac{\text{atan}(x)}{x} \right)$$

The second term of the multiplication is now in the interval (0.75,1) thus having a known MSB position. Therefore, this can be implemented using an approximation of roughly $1 + w_F$ bits. Using the same piecewise polynomial approximation and for a number of intervals equal to 256, the double-precision implementation would now require a degree 5 approximation, having an implementation cost significantly lower than the previous technique. We still require an additional multiplication of size $1 + w_F$, but the total cost still out-weights the previous cost for larger precisions. For precisions slightly smaller than single precision, a degree 2 piecewise polynomial approximation would suffice for the first technique (1+17-bit fraction, approximation size of 27-bits), but would require a degree 2 approximation + 1 multiplication for the second technique. Indeed, the size of the approximation tables and multipliers would be smaller in the second case, but due to the larger granularity of memories and embedded multipliers in FPGA devices this would only be marginally exploitable.

An alternative way for approximation atan(x), $0 \leq x \leq 1$ is to use the power series in Equation (1), in floating-point arithmetic in order to cope with the dynamic nature of the result when $x \to 0$. Traditionally this is considered very costly on FPGAs, where the cost of floating-point operators, especially adders, is very high. The main contribution of this paper is a novel technique for evaluating the polynomial associated to the truncated power series on a restricted input

range of the form $[0,2^{-k}]$ with values of k typically in 8..10. The rest of the range $[2^{-k},1]$ is handled using the following identity, where $a=b+c$ is the input argument.

$$\text{atan}(a) = \text{atan}(b) + \text{atan}\left(\frac{c}{1+ab}\right) \quad (2)$$

As shown in the results section, although the reconstruction phase is costly for this function, the presented technique allows savings of as much as 50% on the number of memory blocks while increasing DSP count by less than 10%.

III. Proposed technique

Let $P(x) = x - x^3/3 + x^5/5$ be the polynomial obtained by truncating the power series in Equation (1). The accuracy suffices for double precision when $x \in [0,2^{-9}]$. We evaluate this polynomial in floating-point for $x \in [0,2^{-9}]$ and for the interval $[2^{-9},1]$ we will use the identity (2). We restructure the computation $P(x) = x(1 - x^2/3 + x^4/5)$ and denote $y = x^2$. The right hand term to evaluate now becomes:

$$Q(y) = 1 - \frac{y}{3} + \frac{y^2}{5}.$$

When evaluating $Q(y)$, as $y < 1, y \to 0$, the $-y/3$ and $y^2/5$ monomial terms have increasingly lower contributions to the final result. Beyond a low value of y specific to each monomial, the monomial contribution in the final summation becomes lower than the accuracy of the format, and can safely be approximated to 0. As y decreases, when all monomials reach this threshold the result of the evaluation is 1. In rough exponent values, $y^2/5$ becomes vanishingly small for $e_y < -27$ whereas for $y/3$ the value is $e_y < -54$.

The relative alignment of the monomials in the final summation depends on the the *weight of the coefficients* and the *negative weight given by* y^i. The weight of the coefficient is fixed for a given polynomial, for instance $1/3$ has a weight of -2. The negative weight given by y^i shifts the coefficients right by a predefined amount. This value is dependent on the exponent of y and is depicted in Figure 2.

For $y < 1$ the evaluation of the polynomial can be performed in fixed-point, once the monomials of order greater than zero are aligned against a_0. The alignment of each monomial only depends on e_y and can be pushed in the alignment of the monomial coefficient. Therefore, the corresponding aligned coefficient may be obtained from a table indexed by the exponent of y.

The tabulated shifts for the coefficients when $y < 1$ have a particular pattern as presented in Figure III. Coefficient a_1 is shifted right in increments of one binary position as the exponent of y decreases. Coefficient a_2, which corresponds to the monomial $y^2 a_2$ is shifted right in increments of two positions as e_y decreases. For instance, when $e_y = -1$:

$$y^2 a_2 = (2^{e_y} 1.f_y)^2 a_2 = 2^{2e_y} 1.f_y{}^2 a_2 = 1.f_y{}^2 (2^{-2} a_2)$$

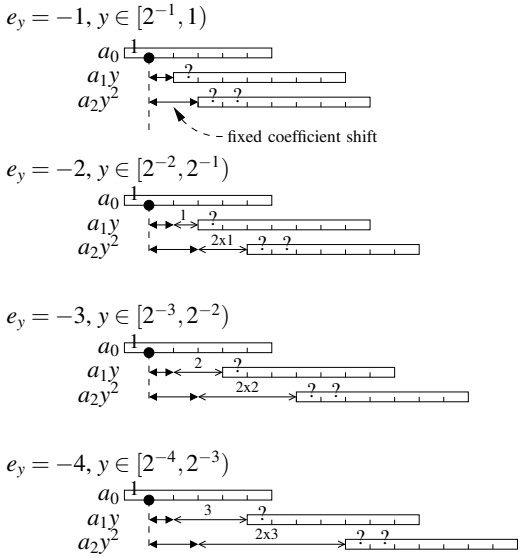

Figure 2. Typical monomial alignment for $Q(y)$ on a toy $w_F = 5$ format

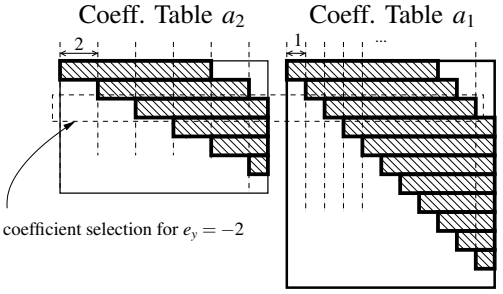

Figure 3. Tabulated coefficient shifts. The first line in each table corresponds to the signed and normalized coefficient. Subsequent lines correspond to weighted-down values of the coefficients corresponding to the shift amounts.

In general, the table corresponding to coefficient a_i will contain all instances of the coefficient shifted right in increments of i positions. As previously explained, for each monomial, as e_y becomes smaller, there exists a threshold value beyond which its contribution becomes smaller than the precision of a_0. This gives us a bound on the number of entries to be stored for each table. Exponent values lower than this threshold will address the final line of the table which contains zero.

IV. Optimizations

We implement this polynomial evaluation as a dynamical fixed-point solution as presented in Figure 4. The size of the shifted coefficient tables, and hence the size of the associated multipliers can be dimensioned to match the double precision accuracy while reducing resources.

In evaluating $Q(y)$ where $x \in [0,2^{-9}]$, we need to optimize the datapath for $y \in [0,2^{-18}]$. Using this information, the first monomial $a_1 y$ will be shifted right at least 18 positions

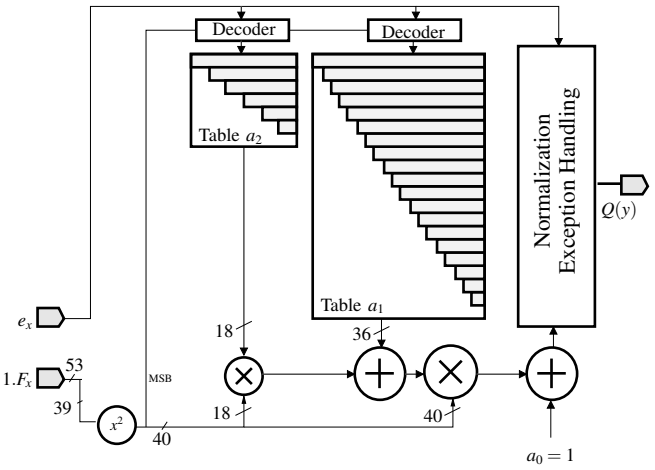

Figure 4. Polynomial evaluation scheme for double precision $atan(x)$, for $x < 1$. The size of the operators is proportional to their implementation size.

compared to $a_0 = 1$. Hence, there is no need to compute the monomial result on more bits than $54 - 18 + g = 36 + g$ bits, where g is the number of guard bits, typically 2 or 3. Similarly, a_2y^2 will be computed on roughly $18 = 54 - 2 \times 18$ bits.

Since for the two monomials of $Q(y)$ (a_2y^2 and a_1y) the maximum number of bits needed is $36 + g$, the squaring operation $y = x^2$ needs to be only accurate to $36 + g$ bits, and can be implemented using a truncated squarer. The final multiplier in $x \times Q(y)$, not depicted in Figure 4, can be implemented using a truncated multiplier.

V. Reconstruction

The computed value for $P(x)$ will be equal to the final atan(x) result if either x or $1/x$ (denoted by z) $\in [0, 2^{-9}]$. For $z \in [2^{-9}, 1]$ the input argument is cast to fixed-point (denoted by a) and split into an upper 9-bit chunk denoted by b and a lower 53-bit chunk denoted by c. Next, identity (2) is used to build the return value. The first term atan(b) is tabulated.

The argument of the second term is computed as follows: the product ab requires two DSPs by replacing it with xb (53×9) and then aligning the result at most 9 bits to the right before the next summation. The addition produces a result which is normalized and can input the inverse unit.

The inverse is computed using a degree-2 piecewise polynomial approximation followed by one Newton-Raphson iteration, as explained in [7], requiring 6 DSPs and 3 memory blocks. The final multiplication is computed using a truncated multiplier requiring 3 DSPs.

VI. Results

We give in Table II the results obtained for the double precision arctangent function on a StratixV FPGA. We compare against the atan(x) implementation in [2] which uses piecewise polynomial approximation, truncated multipliers and to our knowledge does the best set of optimizations for StratixV FPGAs. The proposed implementation has a slightly longer latency due to the longer argument computation for final result reconstruction. The number of DSP blocks and ALMs is slightly larger but we save approximately 50% memory blocks in this example.

Table II
ATAN SYNTHESIS RESULTS FOR DOUBLE PRECISION ON STRATIXV

Method	**Lat. & Freq.**	Resources
[2]	71 cycles @ 427 MHz	22 DSP 17 M20K 3458 ALM
Proposed	78 cycles @ 408 MHz	25 DSP 8 M20K 3773 ALM

VII. Conclusion

We have presented in this work a novel method for evaluating specific polynomials on restricted input ranges in floating-point. The presented techniques allow building a floating-point polynomial evaluation primitive which can successfully be used to implement floating-point elementary functions. For the double precision atan(x), despite the costly reconstruction the presented technique allows saving up to 50% of the memory blocks while increasing DSP and logic usage by less than 10%. For applications which require atan(x) on a small, restricted range $[0, 2^{-k}]$ in a given precision w_F, once a sufficiently accurate $P(x)$ has been determined the primitive can be directly used to generate hardware.

References

[1] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–58, 29 2008.

[2] DSP Builder – Advanced blockset with timing-driven Simulink synthesis, 2011. http://www.altera.com/products/software/products/dsp/adsp-builder.html.

[3] F. de Dinechin and B. Pasca. Designing custom arithmetic data paths with FloPoCo. *IEEE Design and Test*, 2011.

[4] F. de Dinechin and A. Tisserand. Multipartite table methods. *IEEE Transactions on Computers*, 54(3):319–330, 2005.

[5] J. Detrey and F. de Dinechin. Table-based polynomials for fast hardware function evaluation. In *Application-Specific Systems, Architectures, and Processors (ASAP'05)*, pages 328–333, Samos, Greece, July 2005. IEEE Computer Society.

[6] B. Pasca. *High-performance floating-point computing on reconfigurable circuits*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, Sept. 2011.

[7] B. Pasca. Correctly rounded floating-point division for DSP-enabled FPGAs. In *22th International Conference on Field Programmable Logic and Applications (FPL'12)*, Oslo, Norway, Aug. 2012. IEEE.