EFFICIENT FLOATING-POINT POLYNOMIAL EVALUATION ON FPGAS

Martin Langhammer, Bogdan Pasca

Altera European Technology Centre
High Wycombe, UK

ABSTRACT

Many applications require the evaluation of polynomials having floating-point coefficients - one example is rational polynomial approximation, often used to implement some special functions. The most resource efficient polynomial evaluation scheme (Horner) is costly to implement on FPGAs due to the high cost associated with floating-point arithmetic. Floating-point adders are particularly costly due to their alignment stages requiring large barrel shifters. In this work we present a novel FPGA-specific technique for evaluating polynomials using the Horner scheme. Our technique removes the majority of alignment shifters present in floating-point adders by building a fused evaluation operator. It pushes the possible alignment values of the monomials into tables containing multiple shifted coefficient instances which are selected using the exponent of the input argument. Compared to operator assembly this work reduces circuit latency by 30-50% and logic consumption by 40-60%. Our work can be easily extended to other polynomial evaluation methods.

1. INTRODUCTION

Many applications require the evaluation of polynomials having floating-point coefficients. Using a careful dynamical range analysis a fraction of these polynomial evaluators could be converted to smaller and more efficient fixed-point solution. This conversion requires specialized expertise, longer development times and is likely to be performed only for performance critical systems.

A larger ratio of floating-point polynomial evaluators must handle high dynamic data ranges. A solution built by simply assembling floating-point floating-point operators can usually outperform an equivalent fixed-point implementation having a very wide datapath.

Fused floating-point datapaths as presented in [1] can improve the general performance of floating-point datapaths on FPGAs. Unfortunately, the presented techniques can only marginally apply to Horner evaluation datapaths.

Operation-specific fusion, as presented in [2] shows how designing an specific implementation for $x^2 + y^2 + z^2$ can save up to 75% of the resources of a naive floating-point implementation. The work can be easily extended to larger sums-of-squares, and even norms, but is not clear how to extended it to polynomial evaluation.

We have recently presented in [3] a method for evaluating floating polynomials on restricted input range of the form $[0, 2^{-k}]$ where k is typically in the range 8..10. This has been used to implement the $\operatorname{atan}(x)$ function using its Taylor series. In this paper we extend the previously introduced technique to handle the entire input range. Our results prove that this method can reduce circuit latency by 30-50% and logic consumption by 40-60% for general polynomial coefficients.

Applications often require the evaluation of multiple polynomials using the same hardware. One such example is Acklam's algorithm [4] for approximating the inverse cumulative distribution function, where two disjoint rational polynomial approximations are required. We show in this paper how the presented technique can be extended to handle this requirement. We demonstrate in our results section that the impact on the implementation size is minimal.

2. BACKGROUND

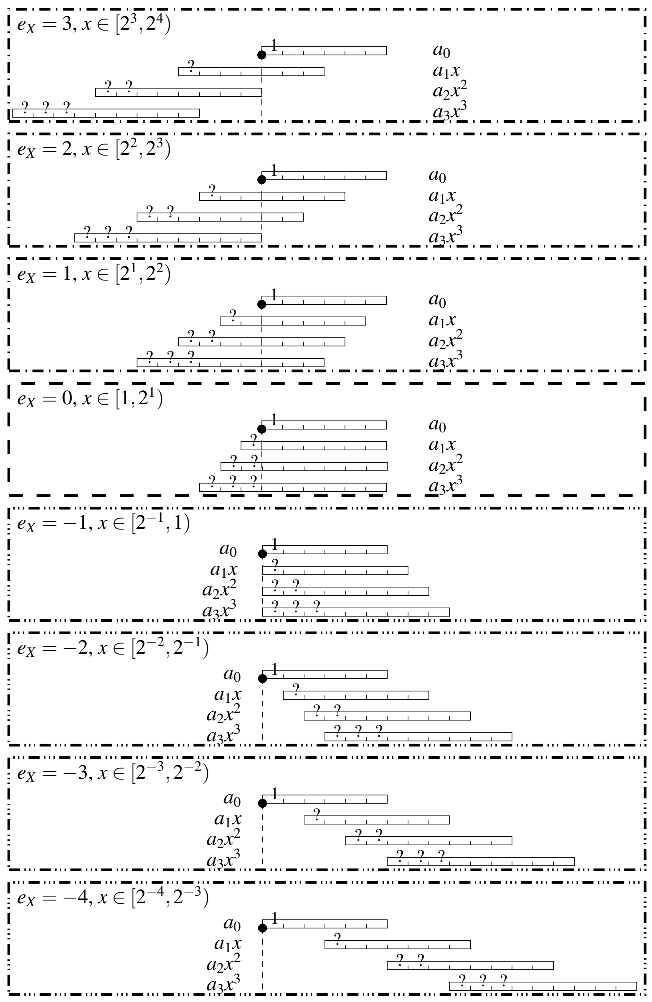
The IEEE-754 standard on floating-point arithmetic (revised in 2008 [5]) uses a triplet (**s**ign, **e**xponent, **f**raction) to represent a floating-point number:

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

The number of bits used to store the exponent and fraction (w_E and w_F) define the formats of the IEEE-754 standard. For instance, binary64 (known as double precision) has $w_E = 11$ and $w_F = 52$ and binary32 (single precision) has $w_E = 8$ and $w_F = 23$. Custom formats (pairs of w_E, w_F not specified in the standard) are used to bridge the precision gap between single and double precision. These formats can be used to exploit the flexibility of FPGAs and outperform microprocessors [6].

Let $P(x)$ be a degree d polynomial of one floating-point variable with floating point coefficients a_i.

$$P(x) = \sum_{i=0}^{d} a_i x^i$$



$e_X = 3, x \in [2^3, 2^4)$
$e_X = 2, x \in [2^2, 2^3)$
$e_X = 1, x \in [2^1, 2^2)$
$e_X = 0, x \in [1, 2^1)$
$e_X = -1, x \in [2^{-1}, 1)$
$e_X = -2, x \in [2^{-2}, 2^{-1})$
$e_X = -3, x \in [2^{-3}, 2^{-2})$
$e_X = -4, x \in [2^{-4}, 2^{-3})$
a_0
a_1x
a_2x^2
a_3x^3

Fig. 1. Typical monomial alignment for $x > 1$ (top) and $x < 1$ (bottom)

There are several well known methods for evaluating polynomials: Horner's method, Estrin's method and several brute-force methods [7]. Horner's method minimizes resources in an unrolled implementation while being the most numerically stable method.

Using Horner's method, the polynomial $P(x)$ will be evaluated in d steps, each consisting of a product π and a sum σ.

$$P(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + x(...x(a_d)...))))$$

3. PROPOSED ALGORITHM

Consider the previously defined polynomial P having constant fixed-point coefficients. For simplicity of the exposition assume that the coefficients are of the same magnitude, and $a_i \in [1/2, 1), i \in [0, d]$. We assume for now that $x \geq 0$. We will later show that the method works without these restrictions.

When evaluating $P(x)$, as $x < 1$ gets closer to zero, the higher order monomials (a_ix^i with i closer to d) also tend to zero, and have an increasingly lower contribution in the final result. In fact, beyond a certain threshold for x (x gets very close to zero) even the contribution of xa_1 has a weight lower than the precision of a_0. In this case the result of the evaluation is simply a_0. For a toy floating-point format with 5 fractional bits ($w_F = 5$) the monomial alignments for decreasing values of the exponent of x (e_X) are exemplified on the bottom of Figure 1. For $x > 1$, the higher order monomials tend to have an increasingly dominant contribution to the final evaluation result. This is exemplified on the top of Figure 1.

In both cases, the relative alignment of the monomials in the final summation has two components. First, the weight of the coefficients, which is equal to the weight of the monomials when $x = 1$ gives the initial fixed-point alignment of the monomial summation datapath. For instance, if $P(x) = 1 - 0.34x + 0.2x^2$ the weights of the monomials would be 0, -2 and -3. Secondly, the weight of x^i (which can either be positive when $x > 1$, or negative for $x < 1$) is associated with the dynamical contribution of the monomials in the summation. For the same polynomial, when $x = 0.5$ which is expressed in floating-point as $x = 2^{-1} \times 1.0$ the first monomial would be equal to $-0.34 \times 2^{-1} \times 1.0$ having a weight of $-2 - 1$. For the degree-2 monomial $0.2 \times (2^{-1} \times 1.0)^2$ which is equal to $0.2 \times 2^{-1 \times 2} \times 1.0^2$ has a weight of $-3 - 1 \times 2$.

Let the Horner evaluation datapath given below

$$P(x) = a_0 + x(a_1 + xa_2)$$

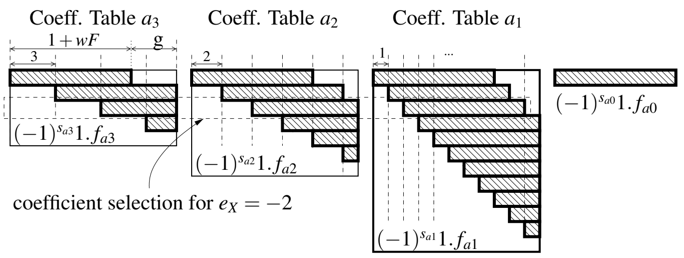
which expands each term into its floating-point components. For simplicity we assume that x and a_i are positive (sign is zero):

$$P(x) = 2^{e_{a0}}1.f_{a0} + 2^{e_x}1.f_x\left(2^{e_{a1}}1.f_{a1} + 2^{e_x}1.f_x 2^{e_{a2}}1.f_{a2}\right)$$

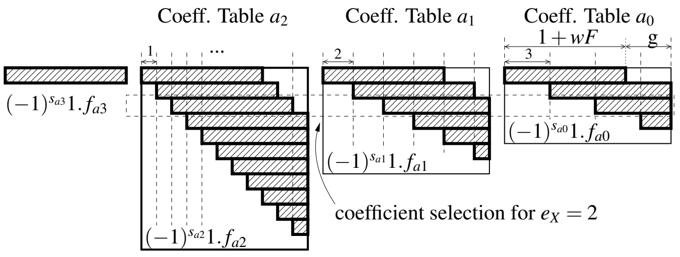
We use the following rewriting which regroups the terms such that the scaling produced by x^i is associated with the corresponding coefficient.

$$\begin{aligned}P(x) =& 2^{e_{a0}}1.f_{a0} + \\ & 1.f_x\Big(\underbrace{2^{e_{a1}+e_x}1.f_{a1}}_{2^{e_{a1}}1.f_{a1}\text{ scaled by }e_x} + 1.f_x \underbrace{2^{e_{a2}+2e_x}1.f_{a2}}_{2^{e_{a2}}1.f_{a2}\text{ scaled by }2e_x} \Big)\end{aligned}$$

For $x < 1$ 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_X and can be pushed in the



(a) Tabulated coefficient shifts for $x \leq 1$



(b) Tabulated coefficient shifts for $x \geq 1$

Fig. 2. 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.

alignment of the monomial coefficient. Therefore, the corresponding aligned coefficient may be obtained from a table indexed by the exponent of x.

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

$$x^2a_2 = (2^{e_x}1.f_X)^2a_2 = 2^{2e_x}1.f_X{}^2a_2 = 1.f_X{}^2(2^{-2}a_2)$$

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_X becomes smaller, there exists a threshold value ζ_i beyond which its contribution becomes smaller than the precision of a_0. This observation allows us to reduce the number of coefficient instances which need to be stored in the tables. Exponent values lower than ζ_i will saturate to this value and will address the last table entry which stores zero. Figure 2(a) presents the general layout of the coefficient tables for $x < 1$.

Let us consider again the same degree 2 polynomial $P(x)$, but now for the case $x > 1$. As previously presented and exemplified on the top of Figure 1, the dominant monomials in the final summation will be the high-degree ones. We use the following rewriting which aligns all the lower order monomials against a_d. The final result is scaled up by the weight of the a_dx^d monomial which is 2^{2e_x} for our example.

$$\begin{aligned} P(x) &= 2^{e_{a0}}1.f_{a0} + 1.f_x\left(2^{e_{a1}+e_x}1.f_{a1} + 1.f_x2^{e_{a2}+2e_x}1.f_{a2}\right)\\ &= 2^{2e_x}\cdot\\ &\Big(\underbrace{2^{e_{a0}-2e_x}1.f_{a0}}_{1.f_{a0}\text{ scaled by }2e_x} + 1.f_x\big(\underbrace{2^{e_{a1}-e_x}1.f_{a1}}_{1.f_{a1}\text{ scaled by }e_x} + 1.f_x2^{e_{a2}}1.f_{a2}\big)\Big) \end{aligned}$$

The relative shifts of the coefficients will be with respect to a_d and depend only on e_x. As previously, these relative shifts also have a specific pattern. Coefficient a_{d-1} will be shifted right in increments of one position, coefficient a_{d-2} in increments of two positions and more generally, coefficient a_{d-i} in increments of i positions. Figure 2(b) shows the general layout of the coefficient tables for $x > 1$ on the example degree 2 polynomial.

The fused shift coefficient tables which allow evaluation of an arbitrary x are presented in the top part of Figure 3.

4. IMPLEMENTATION

The high-level diagram of the implementation is depicted in Figure 3. Each coefficient is associated with a table which contains all the possible shift cases for this coefficient. The shifts consist of a union between required shift cases for $x \leq 1$ and those for $x > 1$ avoiding to store any duplicate shifts. For instance, in the case of a degree 2 polynomial, the shift table associated with a_1 will contain all the shifts with increment 1 for both $x \leq 1$ and $x > 1$. Consequently only one set of shifts will be stored. The fact that coefficient tables are fused is highlighted in Figure 3 where the shifts corresponding to both cases are crossed-hatched.

Each table will have a fixed-point output format. The weight of each table output is given by the exponent value of the corresponding coefficient. For instance, for $P(x) = 1 - 0.34x + 0.2x^2$ the exponent values are 0, -2 and -3. Since coefficients may have different signs, the tables will store the signed values of the coefficients.

The number of bits stored in each table will depend on w_F, the maximum relative shift of the the coefficient against both a_0 and a_d which we denote by τ_i and a number g of guard bits which depend on the polynomial degree such that:

$$w_i = 1 + w_F + \tau_i + g$$

The coefficient tables now contain the shifts corresponding to the exponent of x. We store x on $1 + 1 + wF$ bits in two's complement, having w_F fractional bits. All fixed-point formats are signed, such that we can use a couple $< w, f >$ to characterize each of these formats. There are some basic rules associated to fixed-point arithmetic. For

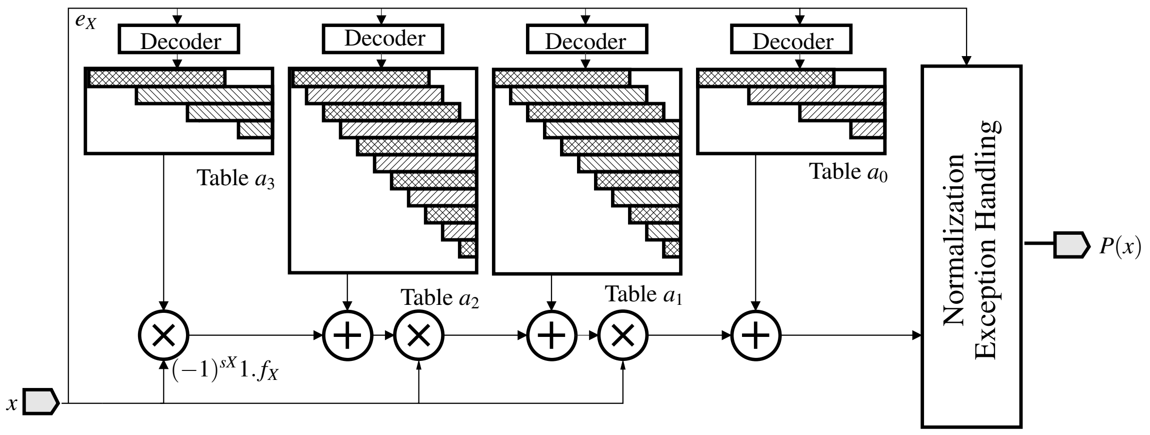


Fig. 3. Proposed polynomial evaluation architecture

multiplication, the format of the result is given by the following formula. The replication of the sign in the product of two signed inputs allows us to reduce its size by one bit.

$$a =< w_a, f_a >; b =< w_b, f_b >$$
$$a \times b =< w_a + w_b - 1, f_a + f_b >$$

The sum of two numbers in this format is given by the formula:

$$\begin{aligned}
&msb_a = w_a - f_a - 1; msb_b = w_b - f_b + 1;\\
&lsb_a = -f_a; lsb_b = -f_b;\\
&s = a + b;\\
&msb_s = \max(msb_a, msb_b) + 1; lsb_s = \min(lsb_a, lsb_b);\\
&w_s = \text{MSB}_s - \text{LSB}_s + 1; w_f = -\text{LSB}_s;
\end{aligned}$$

In order to construct the floating-point output value r^{FP}, the signed fixed-point result r_s^{FXP} having $< w_{r_s}, f_{r_s} >$ has to first be converted to unsigned $r_u^{FXP} =< w_{r_s} - 1, f_{r_s} >$ and then normalized. The number of leading zeros are of r_u^{FXP} is denoted by c.

Depending on the current computing branch, there are two cases for the exponent update which are presented in Equation (4). As expected, the exponent value for the $x > 1$ branch needs to be amplified by $d \times e_X$ which is the scaling factor of the datapath.

$$e_r = \begin{cases} w_{r_u} - f_{r_u} - c - 1 & \text{if } x \leq 1 \\ d \times e_X + w_{r_u} - f_{r_u} - c - 1 & \text{otherwise} \end{cases}$$

After this final shift and normalization only an $w_F + 1$ fraction is kept (not including the "hidden" 1). The *lsb* of this value is used for rounding which may update the exponent.

Finally, the exponent value is checked for overflow and underflow, and together with the input special cases (Zero, Inf, NaN) the final output value is set.

5. EVALUATING MULTIPLE POLYNOMIALS

Evaluating multiple polynomials can easily be done by fusing the shifted coefficient tables. Most of the time tables corresponding to one coefficient have different fixed-point formats across the set of polynomials. In this case, we need to first determine a wider format across the set of tables corresponding to one coefficient which allows all corresponding data to be represented without loss of information.

We simplify addressing the multiple coefficient banks by storing the same number of shifted coefficient instances for corresponding coefficients across all polynomials. This allows for a simple offset addressing scheme as an extension of the single polynomial addressing scheme.

Coefficient sets having very different magnitudes can significantly widen the table outputs and increase the evaluation datapath size. One simple solution is to use fixed prescaling factors for the polynomials in order to minimize the size of the tables. The results will need to be scaled back before constructing the final solution.

6. RESULTS

In this section we present two sets of experimental results. First, in Table 1 we compare the implementation size and latency between the proposed polynomial evaluator and one built out of standard floating-point components. The selected device for this test is a recent Altera Stratix-IV device [8], with a target frequency of 300MHz. The classical polynomial evaluator structures are generated using the Altera DSP Builder Advanced Blockset [9]. The polynomial is evaluated in single and double precision but our flexible architecture can handle any precision. For each format 4 polynomials are evaluated, with degrees ranging from 2 to 5. The polynomial used is $P(x) = 1.7 + 1.432x + 8.99x^2 + 0.07x^3 + 5.33x^4 + 3.99x^5$ for which higher degree monomials are truncated for the $d = 2$, $d = 3$ and $d = 4$ experiments.

For these parameters we can observe that for similar output frequencies the latency is reduced between 38% and 58%. As expected, the logic resources given by the ALM count(1 ALM = 2 look-up tables and 2 registers) is also reduced by 62% to 42%.

The proposed solution can occasionally use more embedded multipliers (DSPs blocks) due the wider internal multipliers (guard bits are required to in order to prevent overflows and increase accuracy). In some situations, the larger granularity of the multipliers (36-bit for efficient usage on StratixIV) can absorb this increase at no extra cost.

Of course, we know that the internal multipliers are wider than for the pure floating-point solution, and that their size increases in the datapath ($\pi_{i+1} < \pi_i$). For multipliers larger than the DSP sweet-spots (36-bit for StratixIV) we use truncated multipliers [10]. For instance, for single precision both degree 4 and 5 polynomials require multipliers with one input slightly wider than 36. Using truncated multipliers with small logic multipliers allows significantly reducing DSP count. When these soft multipliers become too big to implement efficiently in logic, moving to a DSP-based implementation will negatively impact the DSP count.

The shifted coefficient tables needed by our solutions can efficiently be implemented using MLAB (Memory LAB) resources and are included in the total ALM cost. The number of embedded memory blocks (M9K in Stratix-IV) in Table 1 are due to the wide and long delay lines needed to propagate x in the classical approach.

We have used our multi-polynomial evaluation solution in order to implement the inverse normal cumulative distribution function (ICDF) using Acklam's algorithm [4], depicted in Figure 4. Among other computations, this algorithm requires a rational polynomial approximation of degree 5 for the numerator and degree 4 for the denominator. The rational polynomial approximation takes two sets of coefficients: the first for when the probability p is close to 0 or 1, and the second for the general case. For the proposed solution we use the method described in Section 5, and merge both coefficient sets in the same tables; a unique evaluation datapath is built for the two coefficient sets. For the classical solution multiplexers selecting between the constants are used.

Table 2 presents the results for this experiment in both single and double precision. We present the performance and resources of the full system and separately for the multi-polynomial numerator and denominator of the rational polynomial approximation.

We can clearly observe that the same savings in terms of latency and logic can be obtained for multiple polynomials being evaluated using the same hardware.

As for multi-polynomial evaluation, for small number of polynomials the shifted coefficient tables can be stored in MLAB locations. Once transitioned to using memory blocks, a large number of shifted coefficient sets can be packed in these memories before more additional memory is required. For instance, moving from 5 to 8 polynomials should come at no cost once the initial 5 polynomial coefficients are using M9Ks.

Table 2. Synthesis results for the full circuit computing the normal ICDF usign Acklam's algorithm. Results are given for single and double precision, targeting a Statix-IV device with freq = 300MHz. The low part of the table presents the isolated numerator and denominator results.

Method	Lat.	Freq.	Resources
Full system			
Single Precision			
Proposed	89	283MHz	6055ALM, 71 18-bit Mul, 14M9K
Classical	123	278MHz	8791ALM, 71 18-bit Mul, 15M9K
Double Precision			
Proposed	187	247MHz	16881ALM, 360 18-bit Mul, 98M9K
Classical	222	241MHz	22413ALM, 360 18-bit Mul, 100M9K
Numerator + Denominator			
Single Precision			
Proposed	25	312MHz	1477ALM, 36 18-bit Mul
Classical	60	296MHz	4468ALM, 36 18-bit Mul
Double Precision			
Proposed	45	285MHz	5248ALM, 144 18-bit Mul
Classical	80	277MHz	11488ALM, 144 18-bit Mul, 4M9K

Fig. 4. Acklam's code ICDF [4]

```
a, b, c, d: coefficient vectors
Define break-points.
  p_low  <- 0.02425
  p_high <- 1 - p_low
Rational approximation for lower region.
if 0 < p < p_low
  q<- sqrt(-2*log(p))
  x<- (((((c(1)*q+c(2))*q+c(3))*q+c(4))*q+c(5))*q+c(6))/
      ((((d(1)*q+d(2))*q+d(3))*q+d(4))*q+1)
endif
Rational approximation for central region.
if p_low <= p <= p_high
  q<- p - 0.5
  r<- q*q
  x<- (((((a(1)*r+a(2))*r+a(3))*r+a(4))*r+a(5))*r+a(6))*q/
      (((((b(1)*r+b(2))*r+b(3))*r+b(4))*r+b(5))*r+1)
endif
Rational approximation for upper region.
if p_high < p < 1
  q<- sqrt(-2*log(1-p))
  x<- -(((((c(1)*q+c(2))*q+c(3))*q+c(4))*q+c(5))*q+c(6))/
       ((((d(1)*q+d(2))*q+d(3))*q+d(4))*q+1)
endif
```

Table 1. Synthesis results for various degree polynomials on a Stratix-IV with target FMax=300MHz, for single and double-precision

Method	Precision	Degree	Latency	Frequency	Resources
Proposed	Single	2	13	362MHz	372 ALMs, 8 18-bit Mul
Classical			24	290MHz	1036ALMs, 8 18-bit Mul
Proposed		3	17	324MHz	537 ALMs, 12 18-bit Mul
Classical			36	281MHz	1417ALMs, 12 18-bit Mul
Proposed		4	21	318MHz	563 ALMs, 16 18-bit Mul
Classical			48	259MHz	1951ALMs, 16 18-bit Mul
Proposed		5	25	292MHz	717 ALMs, 20 18-bit Mul
Classical			60	271MHz	2437ALMs, 20 18-bit Mul, 1 M9K
Proposed	Double	2	21	300MHz	1266ALMs, 32 18-bit Mul
Classical			32	266MHz	2265ALMs, 32 18-bit Mul
Proposed		3	29	296MHz	1613ALMs, 48 18-bit Mul
Classical			48	257MHz	3346ALMs, 48 18-bit Mul, 2 M9K
Proposed		4	37	289MHz	2279ALMs, 64 18-bit Mul
Classical			64	276MHz	4577ALMs, 64 18-bit Mul, 4M9K
Proposed		5	45	262MHz	2965ALMs, 80 18-bit Mul
Classical			80	274MHz	5720ALMs, 80 18-bit Mul, 6 M9K

7. CONCLUSION

Polynomial evaluation is costly mostly due to the number and the size of floating-point adders. Each floating-point adder contains alignment and normalization stages which are responsible for roughly 75% or the entire logic usage. In this work we have presented an efficient FPGA-specific alternative for evaluating floating-point polynomials. Our work accounts for the polynomial structure of the operation and transfers the input-dependent predictable monomial alignment shifts into tables. We are left with a dynamic fixed-point computation for which FPGAs are designed to perform well. A final and unique normalization stage is required in the general case in order to output the floating-point result. Compared operator assembly this work reduces circuit latency by 30-50% and logic consumption by 40-60% in the general case.

8. REFERENCES

[1] M. Langhammer, “Floating point datapath synthesis for FPGAs,” in *International Conference on Field Programmable Logic and Applications*, sept. 2008, pp. 355 –360.

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

[3] M. Langhammer and B. Pasca, “Elementary function implementation with optimized sub range polynomial evaluation,” in *FCCM 2013: 21St Annual IEEE Symposium On Field-Programmable Custom Computing Machines, Proceedings*, 2013.

[4] P. J. Acklam, “An algorithm for computing the inverse normal cumulative distribution function,” 2002, http://home.online.no/pjacklam.

[5] “IEEE Standard for Floating-Point Arithmetic,” *IEEE Std 754-2008*, pp. 1–58, 29 2008.

[6] F. de Dinechin, J. Detrey, I. Trestian, O. Creţ, and R. Tudoran, “When FPGAs are better at floating-point than microprocessors,” École Normale Supérieure de Lyon, Tech. Rep. ensl-00174627, 2007, http://prunel.ccsd.cnrs.fr/ensl-00174627.

[7] M. Wojko and H. ElGindy, “On determining polynomial evaluation structures for FPGA based custom computing machines,” in *in Proc. 4th Australasian Comput. Arch. Conf*, 1999, pp. 11–22.

[8] *StratixIV Device Handbook*, 2011, http://www.altera.com/literature/hb/stratix-iv/stx4_5v1.pdf.

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

[10] S. Banescu, F. de Dinechin, B. Pasca, and R. Tudoran, “Multipliers for floating-point double precision and beyond on FPGAs,” *SIGARCH Comput. Archit. News*, vol. 38, no. 4, pp. 73–79, Jan. 2011. [Online]. Available: http://doi.acm.org/10.1145/1926367.1926380