FPGA Implementation of Custom Floating-Point Logarithm and Division[⋆]

Nelson Campos[1][0000−0001−9709−2703], Slava Chesnokov[3], Eran Edirisinghe[1][0000−0002−7570−3670], and Alexis Lluis[2]

[1] Loughborough University, Loughborough, United Kingdom
{N.C.S.Campos, E.A.Edirisinghe}@lboro.ac.uk
[2] Imaging CV Ltd, United Kingdom
slava@chesnokov.org
[3] ARM Holdings PLC, Manchester, United Kingdom
alexis.lluis@arm.com

Abstract. The mathematical operations logarithm and division are widely used in many algorithms, including those used in digital image and signal processing algorithms and are performed by approximated computing through piece-wise polynomial functions. In this paper we present dedicated FPGA architectures for implementing the logarithm and division operations in floating-point arithmetic. The proposed hardware modules are customizable, with the mantissa and exponent fields of the floating-point representation defined as parameters that can be customized by the hardware designer. The design flow of the arithmetic blocks allows the generation of a set of architectures of custom-precision floating-point, which can result in compact hardware, when the numerical computations require less numerical range or precision. The paper also describes bit-width optimization using precision analysis and differential evolution (a genetic algorithm based method) is applied to reduce the power consumption and the resource usage in the FPGA minimizing the number of flip-flops, lookup tables and DSP blocks according to a desired accuracy chosen in the design, leading to significant resource savings compared to existing IP cores.

Keywords: Floating-Point · FPGA · VLSI · Divider · Logarithm · Polynomial approximation · Differential Evolution.

1 Introduction

Most of the modern embedded systems are powered by 32-bit CPUs capable of performing operations in both integer and single-precision floating-point operations [2]. Although the development of algorithms using fixed-point representation is less expensive in terms of computational power, the dynamic range and precision of the floating-point are typically the criteria used by designers when choosing between the two formats. Furthermore, the development of most

[⋆] Supported by CDT-EI at Loughborough University and ARM Holdings PLC.

of the algorithms are generally easier when using floating-point representation, as fixed-point format often requires additional numeric manipulations to reduce the quantization noise[10]. However, fixed-point processors are still used more extensively than the floating-point processors due to their reduced cost, size and power consumption.

Dedicated hardware architectures prototyped in FPGAs are usually implemented over their software-based models due to their advantages in flexibility and speed. However, the development of such architectures is generally a complex task which involves a tradeoff of resource usage of memory, circuit area and number of Digital Signal Processing (DSP) blocks. For instance, digit-recurrence methods to perform division operations require extensive use of hardware resources (lookup tables and flip-flops) and have high latency, whilst polynomial approximation techniques use excessive memory and DSP blocks [4].

One of the main objectives in the FPGA design flow is to produce an efficient design in terms of small circuit area and low latency, throughput and power consumption. A common optimization technique is to find the minimum number of bits that represents a signal within each design. This technique is often expensive in search space and an analytic range analysis is required to determine the bit-width of each signal [5].

In this paper we present dedicated FPGA architectures for implementing the logarithm and division operations in floating-point arithmetic. A numeric analysis is provided to find the optimum bit-width of each coefficient of the polynomial approximations used to compute the transcendental functions such as division and logarithm, but it is shown that the method can be expanded to implement any other function that can be approximated to a polynomial. These coefficients are also dependent on the width of the mantissa, once the modules are customizable, being compatible with the 32-bit IEEE-754 [1] standard format and also being expandable to fit for other numerical representations, such as the 16-bit half-precision floating-point, more suitable for embedded image processing applications.

To summarize, the main contributions of this paper are:

- propose floating-point hardware architectures for mathematical division and logarithm modules;
- parameterizable modules with the mantissa and exponent widths as inputs defined by the user;
- apply techniques of precision analysis and bit-width optimization using differential evolution in polynomial approximations to reduce the resource usage of the division and logarithm hardware modules.

The rest of this paper is organized as follows: Section 2 reviews previous published papers to provide a contextual support to this current work. Section 3 presents an overview of the proposed architectures of the floating-point hardware modules for division and logarithm. Section 4 discusses the results of precision analysis, synthesis and optimization of the proposed architecture and Section 5 gives conclusions and recommendations for future work.

2 Previous Work

Floating-point arithmetic have been widely implemented in FPGAs to speed up applications which require extended dynamic range and higher precision. Hardware architectures for the single precision natural logarithm and exponential, targeting FPGA architectures such as the Arria 10 and Stratix 10 FPGAs were presented in [4]. For both functions, the proposed implementations made efficient use of the available resources including: DSP blocks in fixed-point arithmetic mode, memory blocks, etc. Overall, the proposed cores offered high-performance, while generally reducing logic consumption, at the expense of DSP and M20K blocks. The work also presented a comparison of resource usage of the proposed modules with that at both industry and open-source competitors. Their analysis were only shown for single-precision. We extend the analysis of our results to both single and half-precision.

There is a number of commercial floating-point arithmetic hardware cores available in the market, which provide a high level of user specification. Xilinx LogiCORE [11] is a floating-point library, which enables the specification of floating-point functions such as adders and multipliers, logarithms and exponentiation. The exponent and mantissa fields can be customized by the user (with bit-width between 4 and 16 bits for the exponent and 4 to 64 bits for the mantissa). Another popular state-of-the-art in both industry and academia is FloPoCo (Floating-Point Cores) [3], which is an open-source C++ framework for generating custom data-path arithmetic cores in VHDL.

Section 4 compares the synthesis results of our proposed architectures with the synthesis of FloPoCo and LogiCORE for the mathematical operations of division and logarithm. Our experiments were performed in the Artix 7 FPGA using the software tool Vivado.

As mentioned previously, the implementation of transcendental functions are often performed through the use of polynomial approximations and one of the main bottlenecks in the design is to find the optimum bit-with to minimize resource usage in FPGAs. In [5] and [7] methodology for bit-width optimization were presented in the context of polynomial approximations. The methodology was based on Affine Arithmetic and Adaptive Simulated Annealing (ASA) aiming to reduce the fractional bits of the coefficients of the polynomial represented in fixed-point format. Although these works are relevant to us, their proposed techniques deal with fixed-point functions. In this paper we present a methodology to minimize the bit-widths of the coefficients of polynomials according to the accuracy desired, i.e. enabling variable range and precision. Further the optimisation technique used by the proposed approach uses Differential Evolution in order to reduce the resource usage of FPGAs when implementing floating-point cores of logarithms, division and exponentiation.

3 Proposed Architecture

In this section we describe the proposed floating-point architectures for mathematical operations of division and logarithm. The implementations are based on polynomial approximation functions that use read-only memory (ROM) to store the coefficients of the polynomials. We also present the novel method used for bit-width optimization, Differential Evolution, that is proposed to be used to minimize the FPGA resource usage within the hardware modules.

3.1 Division

Consider two numbers x and y represented in floating-point format. The number $z = \frac{x}{y}$ can be expressed according to the equation 1:

$$\frac{x}{y} = (-1)^{(s_x - s_y)} \times \frac{1.m_x}{1.m_y} \times 2^{(e_x - e_y) - bias} \tag{1}$$

The proposed architecture for the division is depicted in figure 1. As can be seen in equation 1, the determination of the sign of z s_z can be directly obtained with a xor of the two input signs s_x and s_y and the exponent is computed with just one subtraction.

The output mantissa $m_z = \frac{m_x}{m_y}$ can be expressed as a product $m_x \times m'_y$, where $m'_y = \frac{1}{m_y}$ and m'_y can be approximated by a degree-2 piecewise polynomial approximation with 4 segments [6]. Once $1.0 \leq 1.m < 2.0$, the reciprocal of $1.m$ can be approximated to $\frac{1}{m+1} = (c_0 \times m + c_1) \times m + c_2$, where the coefficients c_0, c_1 and c_2 are obtained through polynomial approximation and they are defined in the table 1b. Each coefficient is stored in a ROM memory that is indexed according to the range of the mantissa. To ensure normalised floating-point numbers, if the reciprocal of the mantissa is a number smaller than one, then the resulted mantissa is right-shifted by one bit, whereas the exponent is increased by one. The datapath uses multiplexers to check this normalisation.

3.2 Binary Logarithm

The computation of the binary logarithm $y = log_2(x)$ is given by the equation 2.

$$\log_2(x) = \begin{cases} log_2(1.m_x) + (e_x - bias), & s_x = 0, \\ NaN, & s_x = 1 \end{cases} \tag{2}$$

As the logarithm is defined only for positive numbers in the real domain, when the x is negative the result y should be not a number (NaN). One multiplexer is used to check the sign of the number comparing the exponent with the floating-point bias. When x is positive, the computation is the sum of the exponent $e_x + log_2(1.m_x)$. The architecture of the logarithm is illustrated in figure 1. A similar module for piecewise polynomial approximation is used in order to compute $log_2(1 + m_x)$, where the coefficients can be seen in table 1 and the result of $log_2(1 + m_x)$ is added to the exponent normalised[4] e_x and then converted from fixed-point to floating-point.

[4] The exponent e_x is left-shifted by M bits, where M is the mantissa width.

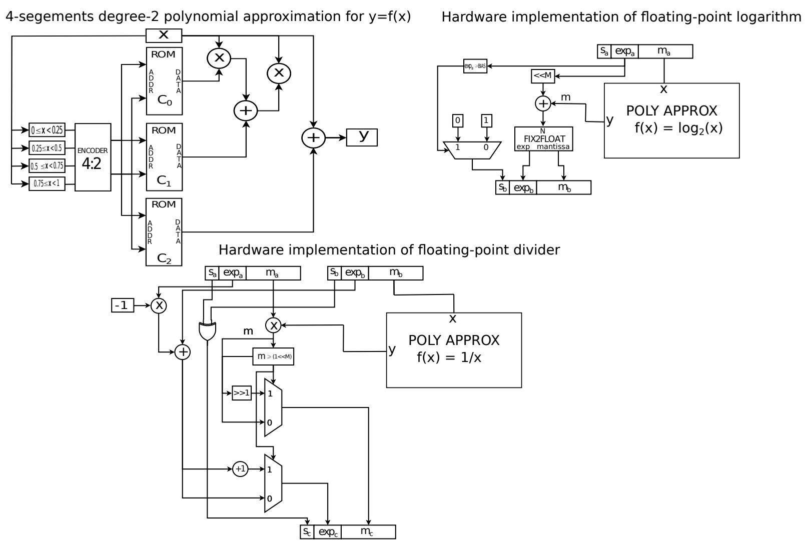
4-segements degree-2 polynomial approximation for y=f(x)
Hardware implementation of floating-point logarithm
Hardware implementation of floating-point divider
x
ROM
ADDR
DATA
C0
C1
C2
ENCODER 4:2
y
POLY APPROX f(x) = log2(x)
POLY APPROX f(x) = 1/x
FIX2FLOAT
<<M
m
-1
>>1
+1

Fig. 1: Floating-point divider and logarithm FPGA architectures

Table 1: Coefficients of the polynomial approximation for $log_2(1+m)$ and $\frac{1}{m+1}$, where $m \in [0,1)$

(a) Logarithm approximation

Range	c_0	c_1	c_2
$1.0 \leq 1.m < 1.25$	-0.573	1.4288	0.0003
$1.25 \leq 1.m < 1.5$	-0.3829	1.3381	0.0115
$1.5 \leq 1.m < 1.75$	-0.2739	1.2312	0.0379
$1.75 \leq 1.m < 2.0$	-0.2056	1.1299	0.0756

(b) Reciprocal approximation

Range	c_0	c_1	c_2
$1.0 \leq 1.m < 1.25$	0.7099	-0.9736	0.9995
$1.25 \leq 1.m < 1.5$	0.3874	-0.8222	0.9811
$1.5 \leq 1.m < 1.75$	0.2342	-0.6729	0.9444
$1.75 \leq 1.m < 2.0$	0.1523	-0.5517	0.8995

In order to illustrate the conversion from fixed to floating-point consider the following example. If $x = 6.75$, in floating-point format x is 1.6875×2^2 and $y = log_2(6.75) = 2 + log_2(1.6875) = 2.7549 = 1.37745 \times 2^1$. The value y can represented as a signed fixed-point number in the format $Q5.10$ as $y = 0000010.1100000101_2$. Since y can also be expressed as $y = 1.m \times 2^{exp}$, to convert this number to a half-precision floating-point, where the exponent and mantissa widths are 5 and 10 respectively, the exponent is found as the position of the Most Significant Bit (MSB) and the mantissa is composed by the 10 least significant bits (LSB) of y, where the MSB of the mantissa is at the position 0 and the LSB of y is discarded in the conversion[5].

[5] The MSB position of the fixed-point number is found using a combinational priority encoder.

3.3 Bit-width optimization

Consider the degree-1 polynomial $y = c_0 \times m + c_1$ and denote ϵ_y, ϵ_{c_0} and ϵ_{c_1} the errors at signals y, c_0 and c_1, respectively. The most straightforward way to quantize y is accomplished through truncation, which gives a maximum error of 1 *ulp* (unit in the last place), and faithful rounding, with maximum error of (1/2) *ulp*. As rounding (which gives one additional bit of accuracy) requires additional hardware in contrast to truncation, we chose the truncation as a quantization method for the signals in the precision analysis [7].

Adding the quantization effect to the degree-1 polynomial, equations 3 and 4 are obtained.

$$y + \epsilon_y = (c_0 + \epsilon_{c_0}) \times (m + \epsilon_m) + c_1 + \epsilon_{c_1} \tag{3}$$

$$\epsilon_y = m \times \epsilon_{c_0} + c_0 \times \epsilon_m + \epsilon_m \times \epsilon_{c_0} + \epsilon_{c_1} \tag{4}$$

Denoting ϵ_x the truncation error of the signal x, where $\epsilon_x = 2^{-FB_x}$ and FB_x is the number of fractional bits of x, the maximum output error $max(\epsilon_y)$ needs to be less than 1 *ulp*:

$$max(\epsilon_y) \leq 2^{-FB_y} \tag{5}$$

Since $max(m) = 1$, the maximum accuracy of y for a given mantissa with M-bits width is given as $max(acc_y) = \alpha \times 2^{-FB_{c_0}} + \beta + 2^{-FB_{c_1}}$, where $\alpha = 1 + 2^{-M}$ and $\beta = |c_0| \times 2^{-M}$.

As a design decision, suppose that y is accurate to 16 bits. Combining equations 4 and 5 gives the equation 6:

$$\alpha \times 2^{-FB_{c_0}} + \beta + 2^{-FB_{c_1}} < 2^{-16} \tag{6}$$

A straightforward solution to the expression 6 is to use Uniform Fractional Bit-widths (UFB), which results in a sub-optimal solution. Rather than using UFB, i.e., assigning the same number of bits for each signals, using Multiple Fractional Bit-width (MFB) leads to the optimal solution [7]. Lee *et. al* [7] propose to use Simulation Annealing (ASA) to optimize the bit-widths of c_0 and c_1 and the with the estimated bit-widths FB_{c_0} and FB_{c_1} compute the FB_{c_2}. In this work we apply differential evolution using *mystic* [8], a python library for non-linear and constrained global optimization, to minimize the area of the expression $y = c_0 \times m + c_1$, which is composed of one addition $(x_0 + x_1)$ and one multiplication $(x_0 \times x_1)$ modeled as a cost function $area_y(x_0, x_1) = max(\lceil FB_{x_0} \rceil, \lceil FB_{x_1} \rceil) + \lceil FB_{x_0} \rceil \times \lceil FB_{x_1} \rceil$ constrained to equation 6. After finding FB_{c_0} and FB_{c_1}, the determined values using *mystic* are substituted in the expression 7 to compute FB_{c_2}.

$$max(\epsilon_z) = \alpha \times \epsilon_y + max(|y|) \times \epsilon_m + \epsilon_{c_2} \tag{7}$$

4 Results and Discussion

The bit-widths for each coefficients of the polynomial approximations c_0, c_1 and c_2 were obtained using differential evolution[6] and the results are shown in table 2a. For both division and logarithm mathematical operations, the maximum accuracy is 10^{-7} when the mantissa is of precision 10-bit, which resulted in a reduction of 1-bit in c_1 and c_2 (when the coefficients has uniform fractional bit-width) in the division operation. When the desired accuracy is 2^{-16}, which guarantees the correctness up to the fourth decimal digit, the signals c_0, c_1 and c_2 decreased the bit-width from 23 bits to 19, 17 and 18 bits, respectively.

Table 2: Multiple Fractional Bit-width (MFB) coefficients found using differential evolution

(a) Divider

m	c_0	c_1	c_2	a_{cc}
10	10	9	9	10^{-7}
13	12	12	12	10^{-10}
16	15	16	15	10^{-13}
19	19	18	18	10^{-16}
23	17	18	19	10^{-16}

(b) Logarithm

m	c_0	c_1	c_2	a_{cc}
10	10	10	8	10^{-7}
13	13	13	11	10^{-10}
16	16	16	14	10^{-13}
19	19	19	17	10^{-16}
23	19	17	18	10^{-16}

The accuracy of the floating-point division and logarithm will be directly dependent on the accuracy of the polynomial approximations of $log_2(m+1)$ and $\frac{1}{(m+1)}$ over the interval $[0, 1)$. Better accuracy is achieved with higher degrees of the polynomial approximations, as well as with a large number of intervals with smaller segments[4]. The approximation of $log_2(m+1)$ and $\frac{1}{(m+1)}$ over $[0, 1)$ using degree-2 piecewise polynomial approximations is accurate to 12 bits with 1 *ulp* and 2 *ulp*, respectively[7].

The number of lookup tables , flip-flops, DSP blocks and Block RAM (BRAM) are of particular interest for this experiment. The LUT and flip-flops are the basic building blocks of an FPGA and are generally used for the implementation of combinational and sequential logic, respectively. The BRAM is an embedded memory on-chip which provides a relatively large amount of data and is often used when inferring memory. DSP blocks are arithmetic logic units embedded into the fabric of the FPGA and are usually inferred to perform multiplication operations.

[6] using the parameters maximum iterations to run without improvement **gtol= 2000** and population size **npop = 500**.

[7] According to [9] the Flopoco operator for division is correctly rounded and has accuracy of 0.5 ulp, whereas the LogiCORE documentation states that the division and logarithm has accuracy of 1 ulp.

Table 3: Resource Usage and Latency for Division and Logarithm on Artix 7 FPGA

Precision	**Function**	**Architecture**	**LUTs**	**FFs**	**DSPs**	**BRAM**	**Latency**
half-precision	**Div**	Proposed UFB	43	0	3	0.0	6 cycles
		Proposed MFB	41	0	3	0.0	6 cycles
		LogiCORE Div	253	431	0	0.0	variable
		FloPoCo Div	498	212	0	0.0	10 cycles
	Log	Proposed UFB	211	49	2	0.0	7 cycles
		Proposed MFB	210	49	2	0.0	7 cycles
		LogiCORE Log	274	469	2	0.0	variable
		FloPoCo Log	411	377	1	0.5	17 cycles
single-precision	**Div**	Proposed UFB	137	0	7	0.0	6 cycles
		Proposed MFB	127	0	6	0.0	6 cycles
		LogiCORE Div	809	1487	0	0.0	variable
		FloPoCo Div	1617	624	0	0.0	17 cycles
	Log	Proposed UFB	534	65	5	0.0	7 cycles
		Proposed MFB	555	65	4	0.0	7 cycles
		LogiCORE Log	721	1191	4	0.0	variable
		FloPoCo Log	681	842	3	2.0	23 cycles

Table 3 shows the FPGA resource usage for piecewise polynomial approximations for both $\frac{1}{m+1}$ and $log_2(1+m)$. The maximum latency of the Floating-Point for LogiCORE operators can be found on the Vivado IDE [11]. The throughput of the operations is one operation per clock cycles, which gives a performance of 100 million floating-point operations per second for a clock frequency of 100MHz.The modules were synthesized in the Artix 7 FPGA from Xilinx using the Vivado 2018.3. The polynomial approximation for division, when the mantissa width is 23 bits and the desired accuracy is 2^{-16}, the bit-width optimization showed a reduction of 10.3093% and 20% in the number of LUTs and DSP blocks, respectively. Similarly, for the same mantissa bit-width and accuracy, in the logarithm approximation the LUTs increased by 41.5584% and the DSP blocks reduced 20%. The trends observed for both logarithm and division approximations showed a smaller reduction in terms of LUTs and DSP blocks when the mantissa is 10 bits wide. The designs of the floating-point division and logarithm have a latency of 6 and 7 clock cycles, respectively, and throughput of one operation per clock cycle and RTL was written in SystemVerilog.

The logarithm and divider in floating-point half-precision (mantissa M=10-bit and exponent E=5-bit) and in single-precision (M=23-bit and E=8-bit) modules were also synthesized in Vivado. The synthesizer converts hardware description language into a gate-level netlist. The implementation stage is a step followed by the synthesis in the Vivado design flow. In this stage the netlist is placed and routed in the FPGA device. The target used is the Artix 7 FPGA at a clock frequency of 100MHz and the designs are described in SystemVerilog.

We have compared the proposed cores (both architectures with uniform fractional bit-width and multiple fractional bit-width obtained with the optimization using *mystic*[8]) with the available cores from the open-source framework FloPoCo[9] [3] and with the Xilinx LogiCORE floating-point cores for division and logarithm [11]. For the logarithm, our proposed core is a base 2 logarithm as opposed to the natural logarithm from FloPoCo and LogiCORE. However, our comparison is still fair, once the base conversion of logarithms can be performed with just one multiplication by constant[10]. Table 3 explores the FPGA resource usage for the logarithm and division cores. The optimization showed a reduction in the resource usage as compared to the same architectures without the bit-width optimization. To measure this reduction, we first synthesize the hardware modules (floating-point logarithm and divider cores) in Vivado using uniform bit-width for all the coefficients c_0, c_1 and c_2 in the polynomial approximation cores. Thereafter, for a given precision (single or half-precision) we use our optimization framework using *mystic* in two stages. Firstly, we find the minimum number of bits of c_0 and c_1 using equation 6. At this stage, we also estimate the error ϵ_y, defined in equations 3 to 5. During the second stage, we use the error ϵ_y found previously in equation 7 to compute c_2. Finally, with the new values of c_0, c_1 and c_2, we synthesize again the modules in Vivado and compare the synthesis results using both uniform and multiple fractional bit-widths.

Overall, the proposed cores showed less resource usage than FloPoCo and LogiCORE. It is important to note that unlike FloPoCo, our modules do not round the operations to the final result. In addition to that, the polynomial approximations are of order 2 with only 4-piecewise segments. Furthermore, for simplicity the proposed cores in this work does not have the exceptional cases to deal with subnormal numbers or infinity and not-a-numbers. These limitations were not an issue, as the accuracy and precision are sufficient for our applications in image processing and hardware compactness had higher priority during the design decisions. As can be observed from table 3, as opposed to FloPoCo, the proposed division cores consume fewer resources because it uses the embedded DSP blocks to perform the polynomial approximations, whilst FloPoCo implementation makes more use of LUTs and flip-flops. Furthermore, the FloPoCo logarithm also inferred block RAM memory.

We also compared the proposed cores with the IP-Cores from Xilinx LogiCORE, which resource usage was roughly half of the LUTs and double of the flip-flops consumed by FloPoCo for the divider module. For the logarithm, FloPoCo consumed less resources (LUTs, flip-flops and DSP blocks) than LogiCORE be-

[8] We chose this framework because of its simplicity to solve highly-constrained non-convex problems using Python.

[9] The Flopoco Div and Log (VHDL divider and logarithm modules) are obtained from the framework with the command **flopoco FPDiv we=8 wf=23** and **flopoco FPLog we=8 wf=23** for single-precision. For half-precision, the parameters are **we=5** and **wf=10**.

[10] If the logarithm base is known during synthesis time, than the coefficients of the polynomial approximation will be different than that for the approximation for $log_2(x)$. Otherwise, a change of base during runtime will require one floating-point multiplier, which increases the latency by one clock cycle and consumes one additional DSP block.

cause FloPoCo also uses BRAMs in contrast to our proposed modules and LogiCORE.

5 Conclusion

In this paper we have presented dedicated FPGA architectures for logarithm and division mathematical operations in floating-point arithmetic. For both modules, the mantissa and exponent are flexible parameters that can be defined by the user. The bit-width optimization of both methods is dependent of the desired accuracy defined in the design stage and is accomplished using differential evolution with the Python library *mystic*. For both modules, the bit-width optimization showed a reduction of LUTs, flip-flops and DSP blocks, leading to significant resource savings compared to existing IP cores.

References

1. Committee, I., et al.: 754–2008 ieee standard for floating-point arithmetic. IEEE Computer Society Std **2008** (2008)
2. Dang, D., Pack, D.J., Barrett, S.F.: Embedded systems design with the texas instruments msp432 32-bit processor. Synthesis Lectures on Digital Circuits and Systems **11**(3), 1–574 (2016)
3. De Dinechin, F., Pasca, B.: Designing custom arithmetic data paths with flopoco. IEEE Design & Test of Computers **28**(4), 18–27 (2011)
4. Langhammer, M., Pasca, B.: Single precision logarithm and exponential architectures for hard floating-point enabled fpgas. IEEE Transactions on Computers **66**(12), 2031–2043 (2017)
5. Lee, D.U., Gaffar, A.A., Cheung, R.C., Mencer, O., Luk, W., Constantinides, G.A.: Accuracy-guaranteed bit-width optimization. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **25**(10), 1990–2000 (2006)
6. Lee, D.U., Cheung, R., Luk, W., Villasenor, J.: Hardware implementation trade-offs of polynomial approximations and interpolations. IEEE Transactions on computers **57**(5), 686–701 (2008)
7. Lee, D.U., Villasenor, J.D.: A bit-width optimization methodology for polynomial-based function evaluation. IEEE Transactions on Computers **56**(4), 567–571 (2007)
8. McKerns, M., Strand, L., Sullivan, T., Fang, A., Aivazis, M.: Proceedings of the 10th python in science conference (2011)
9. Pasca, B.: Correctly rounded floating-point division for dsp-enabled fpgas. In: 22nd International Conference on Field Programmable Logic and Applications (FPL). pp. 249–254. IEEE (2012)
10. Renczes, B., Kollár, I.: Roundoff errors in the evaluation of the cost function in sine wave based adc testing. In: 20th IMEKO TC4 International Symposium and 18th International Workshop on ADC Modelling and Testing. pp. 15–17 (2014)
11. Xilinx: LogiCORE IP Floating-Point Operator v7.0. `https://www.xilinx.com/support/documentation/ip_documentation/floating_point/v7_0/pg060-floating-point.pdf`, [Online]