Efficient Single-Precision Floating-Point Division Using Harmonized Parabolic Synthesis

Süleyman Savas, Erik Hertz, Tomas Nordström, Zain Ul-Abdin
School of Information Technology, Halmstad University, Halmstad, Sweden

{suleyman.savas, erik.hertz, tomas.nordstrom, zain-ul-abdin}@hh.se

***Abstract*—This paper proposes a novel method for performing division on floating-point numbers represented in IEEE-754 single-precision (binary32) format. The method is based on an inverter, implemented as a combination of Parabolic Synthesis and second-degree interpolation, followed by a multiplier. It is implemented with and without pipeline stages individually and synthesized while targeting a Xilinx Ultrascale FPGA.**

The implementations show better resource usage and latency results when compared to other implementations based on different methods. In case of throughput, the proposed method outperforms most of the other works, however, some Altera FPGAs achieve higher clock rate due to the differences in the DSP slice multiplier design.

Due to the small size, low latency and high throughput, the presented floating-point division unit is suitable for high performance embedded systems and can be integrated into accelerators or be used as a stand-alone accelerator.

I. Introduction

Floating-point division is a crucial arithmetic operation required by a vast number of applications including signal processing. For instance, the advanced image creating sensors in synthetic aperture radar systems perform complex calculations on huge sets of data in real-time and these calculations include interpolation and correlation calculations, which consist of significant amount of floating-point operations [1], [2]. However, it is quite challenging to implement efficient floating-point arithmetics in hardware. Division itself is the most challenging basic operation to implement among the others such as addition, multiplication and subtraction. It requires larger area and usually achieves relatively lower performance.

In this paper, we implement a single-precision division method that performs division on floating-point numbers represented in IEEE-754 standard [3]. The implementations are synthesized and executed on a field-programmable gate array (FPGA) for validation and evaluation.

The proposed method consists of two main steps. The first step is the inversion of the divisor and the second step is the multiplication of inverted divisor and the dividend. In the inversion step Harmonized Parabolic Synthesis is used as the approximation method, which is a combination of Parabolic Synthesis and Second-Degree Interpolation [4], [5]. When compared to other methods, Parabolic Synthesis methodology converges faster and entails faster computation and smaller chip area, which in turn leads to lower power consumption. In the Harmonized Parabolic Synthesis methodology the Second-Degree Interpolation achieves the required accuracy by using intervals. The accuracy increases with the number of intervals and for single-precision floating-point inversion 64 intervals are sufficient.

978-1-5090-6762-6/17/$31.00

The division hardware that is implemented in this paper is a part of an accelerator that performs cubic interpolation on single-precision floating-point numbers. The accelerator is implemented in Chisel language [6] to be integrated to a RISC-V [7] core via Rocketchip [8] system on chip generator.

The accelerator and the division hardware will be used as basic blocks for building domain-specific heterogeneous manycore architectures. Based on the requirement of applications, these custom blocks or other similar blocks will be integrated to simple cores. Many of these cores together with many custom blocks will form efficient heterogeneous manycore architectures targeting specific application domains. If an application does not use all of the cores with custom blocks, they can be shut-down by utilizing dark-silicon concept [9].

The rest of this paper is structured as follows: In section II we provide background knowledge on single-precision floating-point representation and the floating-point division algorithm that we have used. Section III discusses related work. Section IV presents the approach that we have used for implementing our algorithm. The implementation on an FPGA board is explained in details in Section V. Section VI provides the results which are compared to related work in section VII. Section VIII finalizes the paper with conclusions and future work.

II. Background

In this section, we introduce the FPGA, floating-point number representation and the method used for performing division.

A. FPGA Features

The target platform in this paper is Virtex UltraScale XCVU095-2FFVA2104E FPGA from Xilinx. This FPGA is developed with 20 nm technology and consists of 537600 look-up tables (LUTs), 1075200 flip-flops (FFs), 1728 block RAMs and 768 DSP slices. Block rams are dual port with the size of 36 Kb and can be configured as dual 18 Kb blocks. The DSP slices include 27×18 two's complement multiplier.

B. Binary32 - Single-precision floating-point numbers

Even though there are many possible floating-point number formats, the leading format by far is the IEEE Standard for Floating-Point Arithmetic (IEEE-754) [3]. This standard defines the representation of floating-point numbers and the arithmetic operations. In this paper, we will focus on binary32 numbers, which are more commonly known as single-precision floating-point numbers.

The IEEE 754 standard specifies a binary32 (single-precision floating-point) number as having:

- Sign bit s: 1 bit
- Exponent width e: 8 bits
- Significand (mantissa) m: 24 bits (23 explicitly stored)

as illustrated in Figure 1.

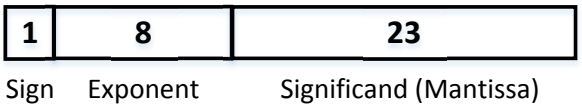

Fig. 1. Single-Precision (binary32) floating-point representation in IEEE-754 format

The sign bit determines the sign of the number. The exponent is an 8-bit signed integer and biased by $+127$. The true significand consists of 23 visible bits to the right of the decimal-point and 1 invisible leading bit to the left of decimal-point which is 1 unless the exponent is 0. The real value is calculated with the following formula:

$$(-1)^s \times (1+m) \times 2^{e-127} \tag{1}$$

The exponent ranges from -126 to 127 because -127 (all zeros) and 128 (all ones) are reserved and indicate special cases. This format has the range of $\pm 3.4 \cdot 10^{38}$.

C. Floating-point division

An overview of division algorithms can be found in [10]. According to the author's taxonomy, division algorithms can be divided into five classes: digit recurrence, functional iteration, very high radix, table look-up, and variable latency. These algorithms will differ in overall latency and area requirements. The algorithm we use, is a table look-up algorithm with an auxiliary function for decreasing the table size.

We implement the division $R = X/Y$ as an inversion of Y ($T = 1/Y$) followed by a multiplication of X ($R = X \cdot T$), as shown in Figure 2.

The floating-point multiplier [11] uses on-board DSP slices to multiply the significands and adder slices (carry logics) to add the exponents. The inverter utilizes the efficient Harmonized Parabolic Synthesis method [4], [5]. The Parabolic Synthesis method is founded on a multiplicative synthesis of factors, each of which is a second-order function. The more factors that are used, the higher is the accuracy of the approximation.

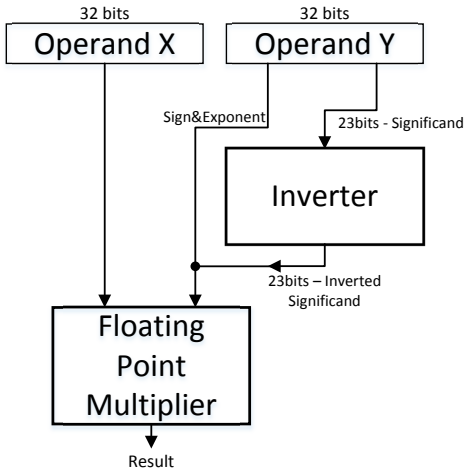

Fig. 2. Overview of the division method. X = dividend, Y = divisor

III. Related Works

There has been extensive research on implementing floating-point operations in hardware. The research has been focused on three aspects namely performance, area, and error characteristic. In the rest of this section, we first present approximation methods for calculating the inversion and then present the prior works about implementing division with single-precision.

There are several methodologies such as CORDIC [12], [13], Newton-Raphson [14], [15] and polynomial approximation [16], [17] for calculating the inverse of a number. However, these methods are additive, whereas the method used in this paper, Parabolic Synthesis, is multiplicative. This means that the approximation converges faster, which results in faster and smaller implementation.

With the introduction of the Parabolic Synthesis methodology, the following improvements were accomplished compared to CORDIC. First, due to a highly parallel architecture, a significant reduction of the propagation delay was achieved, which also leads to a significant reduction of the power consumption. Second, the Parabolic Synthesis methodology allows full control of the characteristics and distribution of the error, which opens an opportunity to use shorter word lengths and thereby gain area, speed and power.

Singh and Sasamal [18] implement single-precision division based on Newton-Raphson algorithm using subtractor and multiplier, which is designed using Vedic multiplication technique [19]. They use a Spartan 6 FPGA and require quite high amount of hardware resources when compared to the proposed method. Leeser and Wang [20] implement floating-point division with variable precision on a Xilinx Virtex-II FPGA. The division is based on look-up tables and taylor series expansion by Hung et al. [21], which uses a 12.5KB look-up table and two multiplications. Regardless of the FPGA, the memory requirement is more than the proposed implementations. It is difficult to compare the resource utilization as the underlying architecture is different between Virtex-II and Ultrascale series, however, they seem to use more

resources than the proposed implementations.

Prashanth et al. [22], Pasca [23], and Detrey and De Dinechin [24] implement different floating-point division algorithms on Altera FPGAs. Even though it is very difficult to do a comparison between these works and the proposed work in terms of the utilization and timing results, one can still see the clear difference in Table I.

Prashanth et al. [22] design a single-precision floating-point ALU including a non-restoring division block consisting of shifters, adders, and subtractors. In contrast to proposed implementations, they do not use DSPs or block RAMs and have quite high cycle count requirement. Pasca [23] presents both single-precision and double precision division architectures based on Newton-Raphson and piece-wise polynomial approximation methods. His work focuses on correct rounding, which comes with a cost of several extra operations whereas rounding has no cost in the proposed implementations. Additionally, memory requirement is more than the requirement of the proposed implementations. Detrey and De Dinechin [24] compute the mantissa as the quotient of the mantissas of the two operands by using a radix 4 SRT algorithm [25]. The exponent is the difference between the two exponents plus the bias. Their work does not utilize any block RAMs or DSPs, however, the requirements for the other resources are significantly higher than the method implemented in this paper.

The methodology, proposed in this paper, is based on an inversion block and a floating-point multiplication block. The novelty lies in the inversion block, which uses the Harmonized Parabolic Synthesis method. This block uses three look-up tables, 4 integer multipliers and 3 integer adders, all with different word lengths. The look-up tables consist of 64 words with 27, 17 and 12 bits of word length respectively. These are significantly smaller when compared to other table look-up methods for the same accuracy.

We are aware that a better comparison could be performed if the platforms were the same, however, we do not have access to the other methods and could not implement them on our target FPGA.

IV. Methodology

The proposed method consists of two main steps as presented in Figure 2. The first step is the inversion of the divisor and the second step is the multiplication of the inverted divisor and the dividend. Both divisor and dividend are in IEEE-754 single-precision floating-point (binary32) format.

In the first step, as seen in Figure 3, the sign bit, the exponent bits, and the significand bits of the divisor are extracted. The significand bits are forwarded to the inverter where the Harmonized Parabolic Synthesis (approximation) is performed. In this block, the coefficients of the synthesis method are stored in look-up tables.

In parallel to the inversion, sign of the exponent is inverted while taking the bias into account as follows:
removing the bias:

$$e' = e - 127 \tag{2}$$

inverting the sign and adding the bias:

$$e'' = 127 - e' \tag{3}$$

when equations 2 and 3 are combined:

$$e'' = 254 - e \tag{4}$$

The exponent is further decreased by 1 unless the result of the inverter is 0.

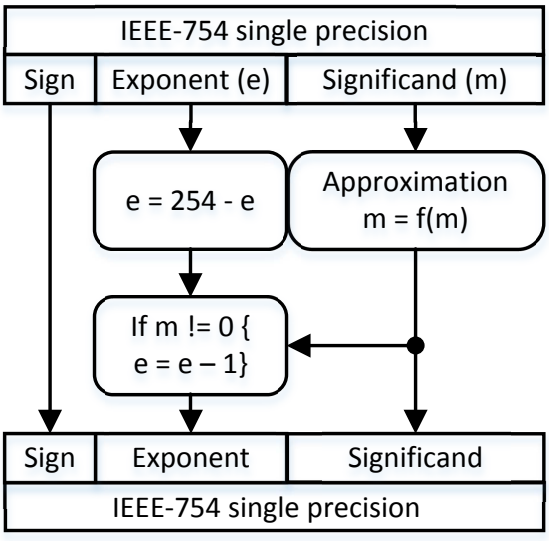

Fig. 3. Block diagram of the inversion algorithm

Sign bit, exponent with inverted sign, and the inverted significand are combined to form a floating-point number. This number is fed as the first input to a floating-point multiplier, which performs the second step of our method. The dividend is forwarded directly to the multiplier as the second input (shown in Figure 2).

In the multiplier, the significands are multiplied and adjusted, the exponents are added together and adjusted based on the resulting significand. The multiplication result of the significands is rounded with different rounding methods such as 'round toward zero', 'round to $+\infty$', and 'round to nearest (ties to even)'. 'Round to zero' provides the smallest max error and best error distribution when compared to other methods.

The inverter has been tested for every possible input. The max error is $1.18358967 \times 10^{-7}$ ($\approx 2^{-23.01}$) which is smaller than the machine epsilon (ϵ) (upper bound for the error) that is commonly defined as 2^{-23} (by ISO C, Matlab, etc) for the single-precision floating-point format.

The division result, which is the output of the floating-point multiplication block, is tested for $2^{23}(8,388,608)$ inputs. The rounding errors for the tested inputs are smaller than 1 ULP (unit in the last place) [26], that is used as a measure of accuracy in numeric calculations and calculated as follows:

$$ULP(x) = \epsilon \times 2^{e'} \tag{5}$$

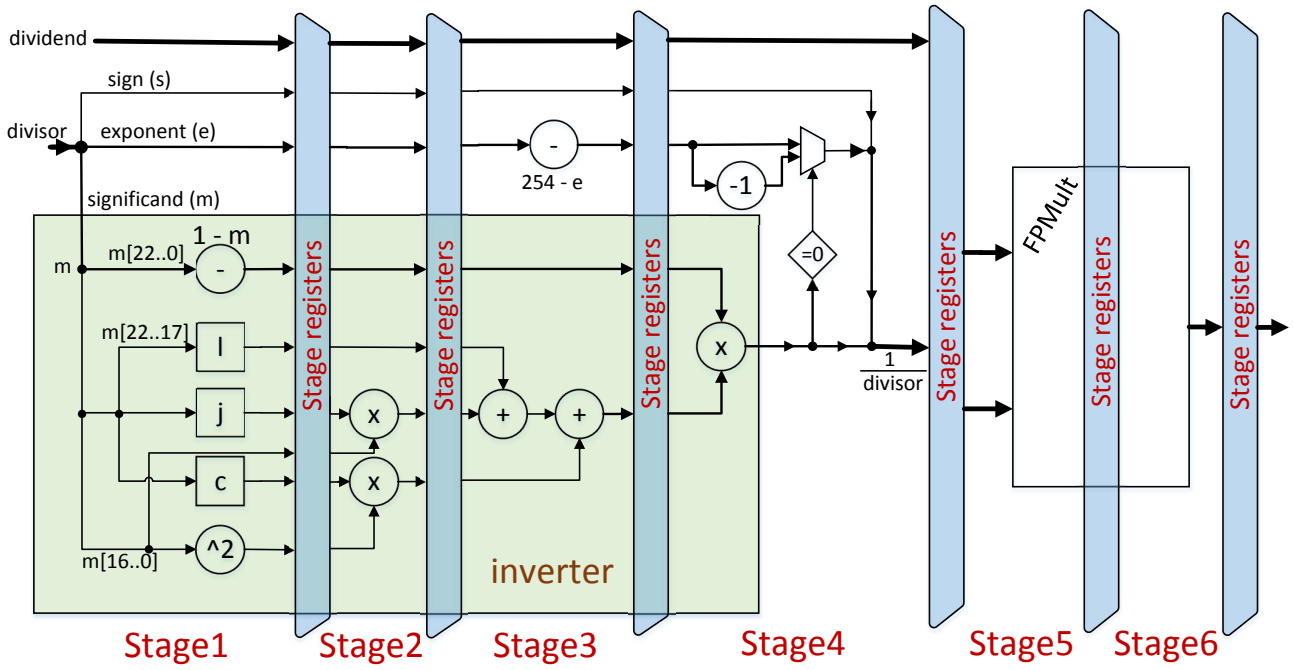

Fig. 4. Overview of the 6 stage pipelined division hardware with detailed view of the inversion component

where x is the result in the single-precision floating-point format, ϵ is machine epsilon and e' is the (unbiased) exponent of x.

V. Implementation Details

The division hardware is implemented in Chisel language and Verilog code is generated. In the generated code, the coefficients were produced via constant wires and multiplexers. The code is modified to reside the coefficients in a block ram. Synthesis is performed by Xilinx tools. The target platform is Xilinx Virtex UltraScale XCVU095-2FFVA2104E FPGA.

Two different implementations of the floating-point division hardware are presented in this paper. The first implementation consists of a single stage, whereas in the second implementation, the inverter is divided into 4 stages and the floating-point multiplier is divided into 2 stages to increase the clock frequency. Consequently the dividend, sign, and exponent bits of the divisor are delayed for four cycles before being fed to the floating-point multiplier. When the inverter and the multiplier are combined the total number of stages in the second implementation becomes 6.

The stages and other details of the hardware implementation of the first step are given in Figure 4. As shown in this figure, two 8-bit subtractors, a single 23-bit equation logic and a single multiplexer are used for computing the exponent of the inverted divisor. These components correspond to the flow of the exponent in Figure 3.

The significand bits of the divisor are connected to the input of the inverter. As shown in Figure 4, the inverter block consists of three look-up tables for storing the coefficients, four integer multipliers, two adders, and a single subtractor. Each look-up table stores 64 words, however, the length of the words change due to the changes in number of bits for storing the coefficients. The word lengths for l, j, and c coefficients are 27, 17, and 12, respectively. When synthesized, these look-up tables are stored in one block ram. The most significant 6 bits of the significand are used for addressing these look-up tables. The same address is fed to each table. Rest of the bits are fed to two multipliers one of which is actually a squarer.

Each integer multiplication unit in the inverter utilizes one DSP slice on the FPGA except the last unit, which requires two DSP slices due to having large inputs. The input and output sizes of the multipliers are $17 \times 12 \rightarrow 17$, $17 \times 17 \rightarrow 18$, $17 \times 17 \rightarrow 17$, and $23 \times 25 \rightarrow 24$. The subtractor and the two adders utilize 6, 6, and 7 carry logics, respectively.

The floating-point multiplier utilizes two DSP slices for multiplying the significands and an 8-bit adder to sum the exponents. Additionally, two 8-bit subtractors are used for adjusting the result exponent.

VI. Results

Two different implementation of the division hardware with different number of stages have been evaluated. The implementations have different results in terms of clock frequency and resource usage. These results are presented in the first two rows of Table I.

The first implementation computes the division in one stage and takes 15.2 ns. Thus, the clock frequency becomes 65 MHz. In the second implementation the inverter is divided into 4 stages whereas the floating point multiplier is divided into 2.

$Stage4$, that is shown in Figure 4, has the longest latency with 4.3 ns. The multiplier, in this stage, utilizes 2 DSP slices

TABLE I
COMPARISON OF DIFFERENT IMPLEMENTATIONS OF SINGLE-PRECISION FLOATING-POINT DIVISION

Published	LUT	FF	DSP	BRAM	Freq(MHz)	Period(ns)	Cycle	Latency(ns)	Platform
Proposed Imp 1	79	32	7	1	65	15.2	1	15.2	Xilinx Virtex Ultrascale
Proposed Imp 2	183	257	7	1	232	4.3	6	25.8	Xilinx Virtex Ultrascale
Singh2016[18]	10019	408	-	-	N/A	N/A	N/A	N/A	Xilinx Spartan 6 SP605
Prashanth2012[22]	762	201	-	-	94	11	50	550	Altera Stratix-III
Pasca2012[23]	426	408	4	2 M20K	400	2.5	15	37.5	Altera Stratix-V
Detrey2007[24]	1210	1308	-	-	233	4.2	16	67.2	Altera Stratix-V
Leeser2005[20]	335 slices x (2LUT, 2FF)		8	7	110	9	14	126	Xilinx Virtex-II

and causes a latency around 4 ns. The next stage consists of another multiplier utilizing 2 DSP slices for the multiplication of the mantissas and has a similar latency. Since it is not possible to divide these multiplications, the clock period is chosen as the latency of the $Stage4$ which is 4.3 ns and the stages are not divided further. As a result, the clock frequency is 232 MHz and the total latency is $4.3 \times 6 = 25.8$ ns.

The implementations use a pipelined approach and produce one result at each cycle. This means max throughput (number of floating point divisions per second) of each implementation is equal to its clock frequency.

Resource utilization of the whole implementation is given in Table I whereas Table II shows the utilization of each individual component for both implementations. The first implementation uses 32 registers to hold the result of the division. In the second implementation, the resource usage increases due to the stages. Some of the stage registers are moved out of the inverter block during optimization by Xilinx tools and they are included in the leaf cell registers in Table II. Most of these registers are used for delaying the dividend that needs to wait for the inverter result.

TABLE II
UTILIZATION RESULTS OF PROPOSED IMPLEMENTATIONS ON A XILINX VIRTEX ULTRASCALE FPGA

Implementation	Module	LUT	FF	DSP	BRAM
Imp 1 (1 stage)	Inverter	40	-	5	1
	FPMult	36	-	2	-
	Leaf cells	3	32	-	-
Imp 2 (6 stages)	Inverter	107	101	5	1
	FPMult	45	10	2	-
	Leaf cells	31	146	-	-
	Available	537600	1075200	768	1728

VII. DISCUSSION

Table I compares the utilization and timing results of our implementations with five prior works. The implementations can be compared based on two aspects; resource usage and timing. However, comparing implementations running on different FPGAs is very difficult, if not impossible. Hence we will mention only comparable results and try to give advantages and disadvantages of our implementations in comparison to the others.

The main advantage of our implementations is the low resource requirement and consequently small area usage. The implemented inverter design requires 3 look-up tables with 64 words in each, which sums up to 448 bytes and utilizes a single block RAM on the target FPGA. On the other hand, the method adopted by Leeser and Wang [20], which is another table look-up method, requires a look-up table with the size of 12.5 KB. Similarly, Pasca[23] uses look-up tables to store approximation results and targets an Altera FPGA. The implementation requires two M20K memory blocks, each of which consists of 12480 configurable bits. Another important resource is the DSP slices. Leeser and Wang [20] utilizes 8 DSP slices within a Virtex-II, which has 18×18 multipliers. The DSP utilization of the proposed implementations would remain as 7 with the same sized DSPs. The required number of slices is 4 for Pasca [23]. However, these DSP slices (on Altera FPGA) support 25×25 multiplication size, whereas the DSP slices on Virtex Ultrascale support 27×18. With larger sized DSP slices, the DSP requirement of the proposed implementations can be reduced to 5. Consequently, due to solving the bottlenecks in $Stage4$ and $Stage5$, the clock frequency can be increased significantly.

Within the implementations, which do not use look-up tables, the implementation of Prashanth [22] utilizes the least number of resources. However, it suffers from the long latency.

Comparing the timing results would hardly lead to any meaningful conclusion due to having different platforms for different implementations. Hence we present the timing results in Table I without going into any comparison to give a hint of the performance of our implementations on a high-end FPGA.

A brief study on increasing the accuracy shows that for every 4-bits accuracy, the number of intervals need to be doubled. The accuracy difference between double precision and single precision formats is 29 bits which would require doubling the number of intervals 8 times. This would increase the number of intervals from 64 to 16384 for each coefficient. The word length of the coefficients would increase as well however, it is difficult to estimate due to lack of a formula. The current memory requirement is 448 bytes and for the double precision division, the memory requirement would be more than 114688 bytes. In addition to the memory size, the component sizes such as the sizes of adders and multipliers would increase as well due to the increase in the word length of the inputs and the coefficients. However, the study shows

that memory requirement would dominate the circuit design when the accuracy would go beyond a threshold.

VIII. Conclusion

This paper presents a novel method for single-precision floating-point division based on an inverter, which utilizes a Harmonized Parabolic Synthesis method, and a floating-point multiplier. The method is used in two different implementations. These implementations perform the divisions on floating-point numbers represented in IEEE-754 binary32 format. In the current state, exceptions and denormalized numbers are not taken into account. However, the error performance of the inverter is validated by exhaustive testing. The maximum error of the division is found to be under 1 ULP (unit in the last place) after testing with random inputs and some corner cases.

The two implementations differ in the number of pipeline stages used and consequently provide different results in terms of latency, throughput and resource usage. Low resource requirement and high throughput make this floating-point division unit suitable for high performance embedded systems. It can be integrated into an existing floating-point unit or used as a standalone accelerator.

Future work includes handling the exceptions and denormalized numbers, completing the cubic interpolation accelerator, which adapts the 6 stage division design, and integrating it to a RISC-V core. Additionally, we plan to implement some other basic operations such as square root and inverse square root with Parabolic Synthesis method, make them IEEE-754 compliant and use them as basic building blocks while building manycore architectures.

References

[1] Z. Ul-Abdin, A. Ahlander, and B. Svensson, "Energy-efficient synthetic-aperture radar processing on a manycore architecture," in *2013 42nd International Conference on Parallel Processing*, Oct 2013, pp. 330–338.

[2] Z.Ul-Abdin, A. Ahlander, and B. Svensson, "Real-time radar signal processing on massively parallel processor arrays," in *2013 Asilomar Conference on Signals, Systems and Computers*, Nov 2013, pp. 1810–1814.

[3] IEEE Task P754, *IEEE 754-2008, Standard for Floating-Point Arithmetic*. New York, NY, USA: IEEE, Aug. 2008.

[4] E. Hertz, B. Svensson, and P. Nilsson, "Combining the parabolic synthesis methodology with second-degree interpolation," *Microprocessors and Microsystems*, vol. 42, pp. 142–155, 2016.

[5] E. Hertz, "Methodologies for approximation of unary functions and their implementation in hardware," Ph.D. dissertation, Halmstad University Press, 2016.

[6] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference*. ACM, 2012, pp. 1216–1225.

[7] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The risc-v instruction set manual, volume i: Base user-level isa," *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, 2011.

[8] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, "The rocket chip generator," 2016.

[9] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 3. ACM, 2011, pp. 365–376.

[10] S. F. Obermann and M. J. Flynn, "Division algorithms and implementations," *IEEE Transactions on computers*, vol. 46, no. 8, pp. 833–854, 1997.

[11] H. Z. Mao. (2017) Floating point chisel modules. [Online]. Available: https://github.com/zhemao/chisel-float

[12] J. E. Volder, "The cordic trigonometric computing technique," *IRE Transactions on electronic computers*, no. 3, pp. 330–334, 1959.

[13] R. Andraka, "A survey of cordic algorithms for fpga based computers," in *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*. ACM, 1998, pp. 191–200.

[14] I. Newton, "Methodus fluxionum et serierum infinitarum," 1664-1671.

[15] J. Raphson, "Analysis aequationum universalis," 1690.

[16] J.-M. Muller, *Elementary functions*. Springer, 2006.

[17] P. Behrooz, "Computer arithmetic: Algorithms and hardware designs," *Oxford University Press*, vol. 19, pp. 512 583–512 585, 2000.

[18] N. Singh and T. N. Sasamal, "Design and synthesis of single precision floating point division based on newton-raphson algorithm on fpga," in *MATEC Web of Conferences*, vol. 57. EDP Sciences, 2016.

[19] J. Swami, S. B. Krisna, and T. Maharaja, "Vedic mathematics or sixteen simple mathematical formulae from the veda, delhi (1965)," *Motilal Banarsidas, Varanasi, India*, 1986.

[20] M. Leeser and X. Wang, "Variable precision floating point division and square root," DTIC Document, Tech. Rep., 2005.

[21] P. Hung, H. Fahmy, O. Mencer, and M. J. Flynn, "Fast division algorithm with a small lookup table," in *Signals, Systems, and Computers, 1999. Conference Record of the Thirty-Third Asilomar Conference on*, vol. 2. IEEE, 1999, pp. 1465–1468.

[22] B. Prashanth, P. A. Kumar, and G. Sreenivasulu, "Design & implementation of floating point alu on a fpga processor," in *Computing, Electronics and Electrical Technologies (ICCEET), 2012 International Conference on*. IEEE, 2012, pp. 772–776.

[23] B. Pasca, "Correctly rounded floating-point division for dsp-enabled fpgas," in *22nd International Conference on Field Programmable Logic and Applications (FPL), 2012*. IEEE, 2012, pp. 249–254.

[24] J. Detrey and F. De Dinechin, "A tool for unbiased comparison between logarithmic and floating-point arithmetic," *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 49, no. 1, pp. 161–175, 2007.

[25] M. D. Ercegovac and T. Lang, *Division and square root: digit-recurrence algorithms and implementations*. Kluwer Academic Publishers, 1994.

[26] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys (CSUR)*, vol. 23, no. 1, pp. 5–48, 1991.