A Library of FSM-based Floating-Point Arithmetic Functions on FPGAs

Atiyehsadat Panahi, Keaten Stokke, David Andrews
Department of Computer Science and Computer Engineering
University of Arkansas
Fayetteville, USA
{apanahi, knstokke, dandrews}@uark.edu

***Abstract*—In this paper, we present a new low-latency method based on Finite State Machines (FSMs) for basic arithmetic as well as trigonometric functions in single-precision IEEE-754 standard format on a Virtex-7 Field-Programmable Gate Array (FPGA). Basic floating-point arithmetic methods are optimized to reduce latency for scalar operations where the cost of pipelining cannot be amortized over longer data streams. We provide additional trigonometric functions necessary for image correlation that are not found in other open-source libraries. Our trigonometric functions are optimized for use in system-on-chip FPGA architectures where Block Random Access Memories (BRAMs) are needed for buffering data and not available to hold large lookup tables. Results are presented comparing delay, area, operating frequency, and memory utilization of our new methods to previous works presented in the literature as well as the built-in Vivado IP. Comparisons show that the proposed approach offers key improvements in delay and area for basic arithmetic operations. Specific to trigonometric functions, our implementations offer a reduction in memory requirements and a higher operating clock frequency compared to existing works.**

***Index Terms*—arithmetic operations, finite state machine, floating-point, FPGA, single-precision, trigonometric functions**

I. Introduction

The best way to implement floating-point arithmetic operations within Field-Programmable Gate Arrays (FPGAs) remains an active area of research [4], [6], [7], [13], [15], [16]. This continued interest reflects the difficult challenge of delivering high-performance floating-point support within the resource and clocking constraints of an FPGA. Clocking frequencies within the reconfigurable fabric are still an order of magnitude slower than what can be achieved in modern Very-Large-Scale Integration (VLSI) designs. The majority of prior works overcome this issue for basic arithmetic operations by shortening and breaking up end to end critical paths into multi-cycle pipelines [1], [5], [9]. This allows the reconfigurable fabric to be clocked near peak levels to deliver the best throughput, but at the cost of increased Lookup Tables (LUTs), Flip-Flops (FFs) and routing resources.

Implementing more complex transcendental and trigonometric functions within an FPGA provides yet another level of complexity beyond those for basic arithmetic operations. Methods such as Taylor series expansion and COordinate Rotation Digital Computer (CORDIC [20]) have been used to provide trigonometric functions [12]–[16]. Such methods can be implemented with a relatively modest number of FFs and LUTs but incur long latencies associated with computing successive iterations to reach target levels of precision. Conversely, on-chip Block Random Access Memories (BRAMs) can be used as a table lookup to eliminate the latency of computing series expansions or coordinate transformations [2]–[4], [11], [12], [14]. However, this method can require substantial numbers of BRAM resources to hold large numbers of lookup values.

978-1-7281-1957-1/19/$31.00 ©2019 IEEE

Prior work in the area has resulted in community efforts to provide reusable open source libraries. The basic arithmetic operations of addition, subtraction, multiplication, and division are common. Works performed from specific application areas have also resulted in additional functionality such as reciprocal, square root, transcendental and common trigonometric functions. Additionally, vendors provide IP cores for floating-point operations for integration into a larger system design.

Reuse and/or modifications to existing libraries increase designer productivity and significantly shorten development times. Unfortunately, there still exist applications where such convenient reuse is not an option. This situation can occur when needed arithmetic functions are not available within the existing libraries, or when system performance requirements cannot be met with the available implementations. In such cases, new implementations are required.

Our application is the acceleration of select correlation functions within the Digital Image Correlation Engine (DICe) software [21]. Performing the correlations required a variety of trigonometric functions, none of which were available in the built-in Vivado IP. DICe also required arcsine and arccosine which were not available in the surveyed open source libraries. In this application, correlation is applied to multiple subsets of pixels across consecutive frames of large images. The data requirements dictated that all available BRAM resources were needed for buffering the multiple large frames. Timing requirements prevented the adoption of trigonometric functions implemented as coordinate rotations, and the unavailability of sufficient BRAM to hold lookup tables prevented the adoption of existing lookup and interpolation techniques. Finally, basic floating-point arithmetic operations needed to be applied concurrently on small subsets of pixel data. This represented a mismatch with how the vendor-supplied block IP was instantiated and called. Further analysis showed that

the fill and flush times for the pipelined implementations of existing library implementations could not be amortized with the relatively small numbers of pixel data processed during a correlation step.

Based on these requirements, we created a new FSM-based library of floating-point operations that included the missing functionality and provided basic arithmetic operations and trigonometric functions that are optimized for latency instead of throughput. It is worth mentioning that the work in this paper is not attempting to compete with any previous works, but rather add to the field by providing an option that satisfies different criteria.

The key contributions of this paper are:

- A single-precision floating-point IEEE-754 compliant library of low-latency functions for real-time applications where an improvement in the delay is preferable over throughput. This distinguishes our library from most other previous works that focus on improving throughput via pipelining.
- An implementation of trigonometric functions that, unlike most previous works that use BRAMs, are developed using Taylor series expansion. Included with this are the arcsine and arccosine trigonometric functions that have not been demonstrated by previous works.
- A detailed performance analysis is conducted between the proposed designs in this paper and state-of-the-art designs, including floating-point operations that are built-in the Vivado IP.

The remainder of the paper is organized as follows. First, previous works on floating-point operations on FPGAs are reviewed. Then, the proposed methods of this study are presented in detail. This is followed by an analysis and comparisons of delay, area, and performance with other floating-point libraries. Finally, we provide concluding remarks on the use of different methods of arithmetic operations for different applications in FPGAs.

II. Previous Works

This section provides a summary of previous studies on floating-point arithmetic operations on FPGAs. We start by discussing alternative formats that have been proposed to the IEEE-754 standard to reduce gate usage. This is followed by a discussion of representative implementation methods for basic arithmetic operations and methods for trigonometric functions.

The area constraints for floating-point arithmetic operations on FPGAs have led to the development of custom formats as alternatives to the standard IEEE-754 floating-point format [1]. In this work, the authors proposed a parameterized library for different floating-point formats. The proposed addition/subtraction function of the library includes four steps: (1) checking the input with the larger magnitude to ensure it's in the first operand position, otherwise the operands are swapped, (2) aligning the mantissa, (3) adding/subtracting the two mantissas, and (4) shifting the result for normalization [1]. Multiplication is simpler than addition and is performed with the following steps: (1) the multiplication of the mantissas, (2) the addition of the exponents, and (3) biasing the resulting exponent; steps (1) and (2) are conducted in parallel. The work in [1] was such a motivator for our paper that similar algorithms for addition/subtraction and multiplication are used, but with different implementation methods. The design in [1] is pipelined and parameterized to operate on any custom format whereas our proposed method is based on a sequential Finite-State Machine (FSM) design that operates on the standard IEEE-754 floating-point format.

The research reported in [2] proposed a method for division using table lookups and Taylor series expansion. The division function requires BRAMs for lookup tables to store the division of mantissas. The same authors proposed an improved version of their work in [3]. The proposed methods of their paper have been tested in a k-means clustering algorithm on a Virtex-II FPGA such as in [1], [2]. A modified version of the divider function has been reported in [4] which is implemented on a Virtex-6. The results of [4] are in double-precision format, therefore, this work is excluded from the comparison section of the current study.

The methods developed in [2]–[4] for division use BRAM table entries to store pre-computed results. The required BRAMs in [2], [3] for single-precision division is approximately 4% but increases to 80% for double-precision. In [4], the required memory for double-precision was improved to 6% of the Virtex-6 BRAM utilization. In contrast, BRAMs are not utilized in our proposed division function as it is FSM-based.

Similar to [1], [5] also used a standard algorithm for addition/subtraction and multiplication. These functions were implemented in a four-stage pipeline using shifting, normalizing, and rounding modules. Each module works at a different frequency which leads to a performance degradation to the system since the clock cannot run faster than the slowest pipeline stage. Therefore, only the lowest clock frequency is provided in the comparison section. The proposed arithmetic functions of [5] are used in matrix multiplication and the efficiency was compared with Pentium4 and G4 processors. The algorithm in their paper has the basic components of our method, but with different implementations. While the functions in [5] were implemented using a pipelined approach, the current study uses a sequential approach that is FSM-based.

The method proposed in [7] uses a CORDIC algorithm for implementing the multiplication and division functions on a Virtex-7 FPGA. No information regarding the delay was reported, even though this would be a critical factor in a CORDIC algorithm as it requires a large number of iterations to reach a reasonable precision.

Within floating-point multiplication, mantissa multiplication is implemented using the Vedic algorithm by utilizing smaller multipliers as the basic component as stated in [8]. In the Vedic algorithm, the individual bits of two operands are multiplied vertically and crosswise. The Vedic algorithm in this work was compared with other methods, such as Booth's algorithm, for mantissa multiplication in floating-point numbers.

The proposed floating-point functions of [9] include addi-

tion, multiplication, and division. The addition and multiplication algorithms of this study are similar to those proposed in [1], [5]. In the division algorithm, the mantissas are divided and rounded, then the exponents are subtracted. In this method, the mantissa division is separated from the rest of the design. This paper implemented the proposed methods for both single-precision and double-precision formats on different FPGA families. Using their proposed methods, they implemented the multiply-accumulate algorithm to test the functionality, and similar to [5], the results were compared to the Pentium4 in [9]. The same algorithms for addition and multiplication are utilized in our study but implemented as an FSM-based method instead of a pipelined method. For division, a different method is developed which will be discussed later in Section III.

The remainder of this section discusses different proposed methods for trigonometric functions. In [11], the sine and cosine functions are implemented using a table-based method. In this paper, the angle is reduced to a specific range close to zero and then the sine and cosine functions are computed using the HOTBM method for table-based Taylor series expansion.

The work in [12] recommended using both the CORDIC and Taylor series expansion methods to implement the sine and cosine functions. The CORDIC method was implemented using an FSM-unit, a micro rotation unit, and ROMs for storing the pre-computed angles. In order to implement the Taylor series expansion, the following units were utilized: a floating-point unit, an FSM-unit, and three ROMs for storing the division factors [12]. The results show that the Taylor series has a smaller area and a higher throughput, but with a lower accuracy level compared to the CORDIC implementation [12]. The difference between [12] and our study is that while they used BRAMs to store the pre-computed factors, all the factors in this study are computed using the proposed basic arithmetic floating-point functions.

It should be mentioned that more studies have been conducted on both basic arithmetic and trigonometric floating-point functions. These works are not included in the comparison section of this paper as they used different methods and formats compared to our proposed methods. For instance, a method for the complex division was proposed in [6]. In another study different configurations were proposed for each arithmetic core according to the number of supported exceptions; however, the results of this paper were reported in double-precision format [10]. Another division method was used in [**?**] which utilizes DSPs and BRAMs, but we cannot compare this to our proposed work because they use a different representation of floating-point numbers; for special cases, two leading bits are used.

Previous works proposed methods for trigonometric functions, such as [13] which compares the CORDIC and Taylor series algorithms, but not on FPGAs. A table-based Taylor series method was proposed in [14], but it has not been implemented on FPGAs either. In another study, the sine and cosine functions were implemented using a CORDIC algorithm on a Spartan-6 FPGA, but resource utilization and delay were not provided in the evaluation [15]. The work in [16] uses TCORDIC (combines low-latency CORDIC and Taylor series algorithms), but the results were reported for only double-precision. The method proposed in [17] uses small lookup tables and low-degree polynomial approximations. However, this work is not in standard floating-point format, therefore it is not compared with our proposed method that is presented in this paper.

III. Proposed Work

This section is devoted to the implementation details of our proposed arithmetic functions. The basic arithmetic algorithms of the addition, subtraction, multiplication, and division functions are first discussed. Then, the trigonometric functions of sine, cosine, arcsine, and arccosine are described. All functions are implemented as FSM-based implementations where the majority of states are processed in one clock cycle and transition at the positive edge of the clock. The basic floating-point algorithms [18] are utilized for the addition/subtraction and multiplication functions. Moreover, the division and trigonometric functions are implemented using a restoring method for mantissa division and Taylor series expansion, respectively.

Fig. 1 shows our proposed method for the addition function. The states in Fig. 1 - 3 are separated by dashed lines. As seen in Fig. 1, the exponents are compared in the first state to assign the mantissa with the larger exponent to the first operand and then the other mantissa is aligned by shifting it to the right. The larger exponent is also assigned to both input operands. In the second state, the actual operation is determined by the input operands sign and a simple fixed-point addition/subtraction on the mantissas is performed. This state also sets the final sign based on the sign of each input and its value. In order to normalize the final result, the mantissa is shifted to the left and the exponent is adjusted until the last extra bit of the mantissa is '1' to represent the implied one in the standard format.

The same procedure is applied for the subtractor function. At the beginning of the first state, it is required to negate the second operand and call the adder function with the negated value $(A - B = A + (-B))$. The first state of the subtractor includes changing the sign of the second operand and assigns the two operands to the adder function. Due to the similarities between the block diagrams of the adder and the subtractor, it is not represented in the figures.

The multiplication function follows the basic algorithm which is simpler than addition since the exponents are not required to be the same. Fig. 2 shows the block diagram of the multiplication algorithm developed in our study. First, a fixed-point addition is performed to add the two exponents together and then increment the result by one to take into account the implied ones of the two operands. The exponent is then biased by subtracting 127 (which is the bias for an exponent in the IEEE-754 single-precision format). In the second state, a fixed-point multiplication on the mantissas is carried out and only the least significant 23 bits of the result are assigned to the final mantissa. The last state of the multiplication function involves the normalization of the mantissa by shifting it to the left and

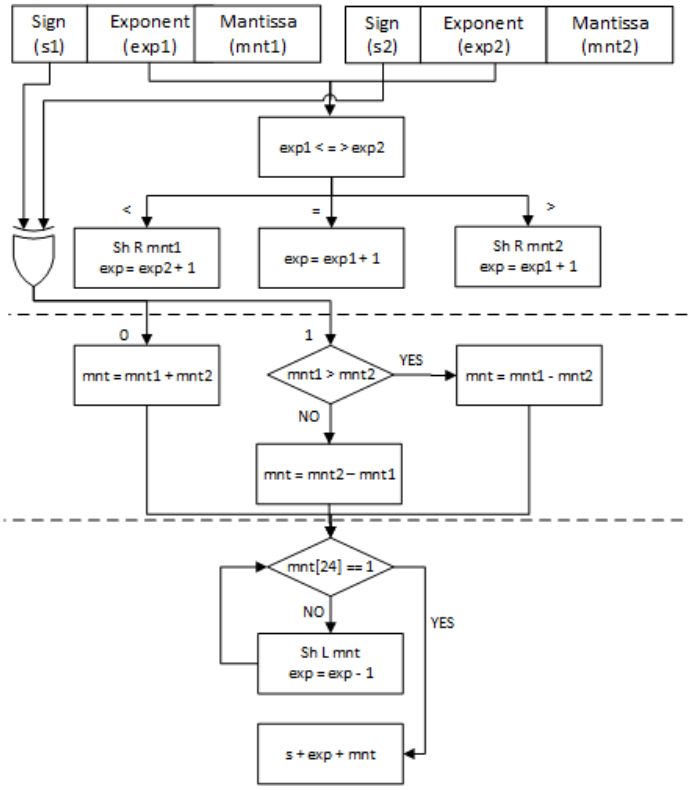

Fig. 1. Block diagram of the addition function.

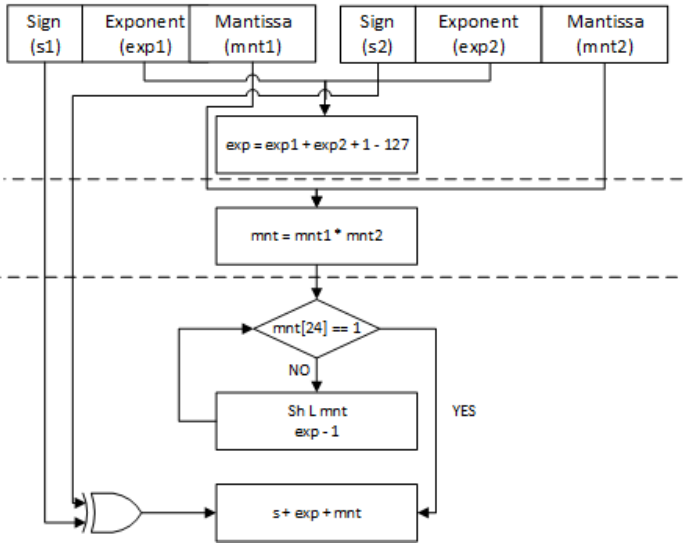

Fig. 2. Block diagram of the multiplication function.

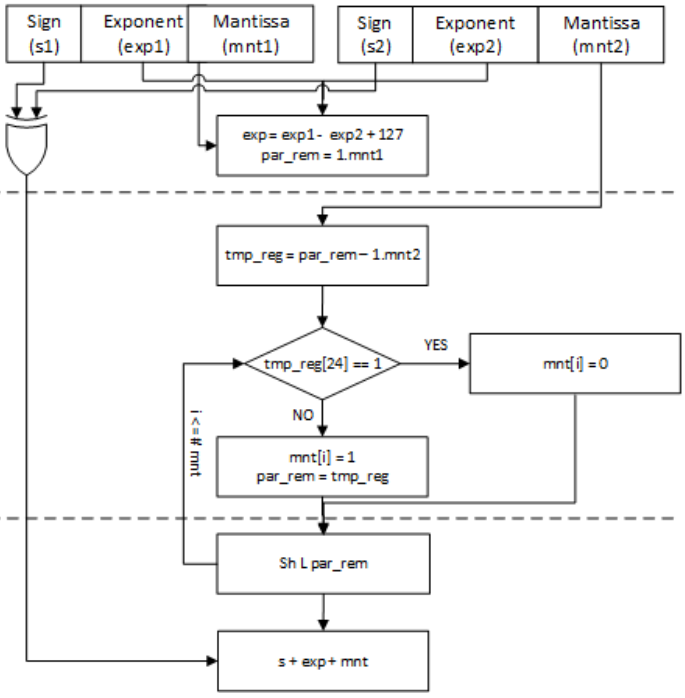

Fig. 3. Block diagram of the division function.

adjusting the exponent. Due to the number of operations in the normalization state, it is executed in two clock cycles.

For the division function, illustrated in Fig. 3, the operands are checked at the first state to handle exceptions such as division of a finite number by an infinite number, division of an infinite number by a finite number, or division by zero. If an exception is detected, the remaining states are skipped and an internal register is set. With our current design, no error flag output notifies the user if an exception has occurred. A single bit output could be added that would notify the user if an error or overflow occurred. If no exception occurs, based on the input operands, the exponents are subtracted and biased. A partial remainder is set to the first operand (dividend). The partial remainder is incremented by one (considering the implied one in the dividend's mantissa). The next steps consist of dividing the mantissa of the first operand by the second operand (divisor). In this regard, a restoring division algorithm for unsigned integers is utilized. To save the extra addition in the restoring state, a temporary register is declared which holds the value of the dividend prior to the subtraction of the divisor. Therefore, at the restoring state in Fig. 3, instead of adding the divisor, the value of the dividend before the subtraction of the divisor (temporary register) is used for the next iteration. The restoring state takes 12 clock cycles to finish execution. The last two states round and then check the result for either the overflow or underflow exceptions.

Our trigonometric functions are computed based on Taylor series expansion which has a fast convergence speed for small inputs. Our study utilizes the first three terms in the Taylor series expansion. This results in a minimum accuracy of 10^{-3} for $-1 < x < 1$ and 10^{-4} for $-0.5 < x < 0.5$ in arcsine and arccosine functions. The proposed basic arithmetic functions are used for calculating the Taylor series terms in the trigonometric functions. These functions, as well as the other basic arithmetic functions, are implemented in an FSM-based method in the proposed library. Equation (1) represents the Taylor series expansion of the $sin(x)$ function, in which n is the number of terms [19]. For example, in (1), if $sin(x)$ is

computed, the multiplier function is called to compute x^2; the result is then multiplied by x to have x^3. x^3 is also multiplied by x^2 (from the previous state) to compute x^5. These expressions are divided by the proper denominators to compute each term of the Taylor series. The same procedure is applied to the other trigonometric functions.

$$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - ... = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!} \quad (|x| < \infty) \tag{1}$$

IV. Results and Analysis

Our proposed methods are implemented using Vivado 2015.4 and Vivado 2018.3 on a Virtex-7 VC707 FPGA with a clock frequency of 200 MHz. Table I shows a comparative analysis of our work along with the results of previous works. The results are extracted from the information provided by the authors in their papers. It is important to note that we did not try to resynthesize the libraries from previous works on the Virtex-7. As mentioned in Section II, studies that either did not provide implementation results or did not work in the single-precision format, are not included in the comparison. Table I contains the listed columns: number of clock cycles to complete, operating frequency, delay, area, power, FPGA family, and method used. Blank entries within the table represent missing values from previous works.

To better illustrate the results of each study in terms of area and delay, as the most important factors, they are shown in Fig. 4. This figure excludes some previous works because they exceeded an acceptable range and it makes our figure more detailed. Delay is reported in nanoseconds for all works based on the data extracted from Table I. Because the number of LUTs and FFs in a slice varies for each FPGA, they were extracted out and compared individually in Fig. 4 to provide an even utilization comparison between FPGA resources.

Our proposed methods are also compared with the built-in Vivado IP for floating-point operations. For each basic arithmetic function in Vivado IP, it provides two optimization goals that can be set within the IP settings: performance and resources. The IP is synthesized on a Virtex-7 FPGA using both the performance and resources optimization goals which are, respectively, labeled as Vivado 1 and Vivado 2 in Table I and Fig. 4.

A. Basic Arithmetic Functions

Fig. 4(a) shows the delay for the basic arithmetic functions. From this figure, our proposed addition function is the fastest among all other approaches, including the built-in Vivado IP. To compare the area, the raw number of LUTs and FFs are reported in Fig. 4(b, c). The results indicate that the proposed addition function of our study reduces the number of LUTs and FFs compared to the previous works.

For the subtractor function, our proposed method improves the delay over Vivado 1 and Vivado 2. Additionally, the area for this function is lower than all previous works, but it is slightly higher than Vivado 2 when set to optimize resources. It should be mentioned that the normalizing and rounding was implemented as a separate module in [1], [3]. Because of this, the results of the addition/subtraction and multiplication functions differ from their reported numbers. Thus, we have added the results of the normalizing and rounding module to the listed functions to give a fair comparison as our proposed method and all others include this step within the algorithms.

Our proposed multiplication function is observed to be the fastest among all previous works, apart from Vivado 2. This leads us to believe that our approach is suitable for high-speed applications that require many multiplications. Delay in the pipelined methods [3], [5], [9] is high when compared to the other studies.

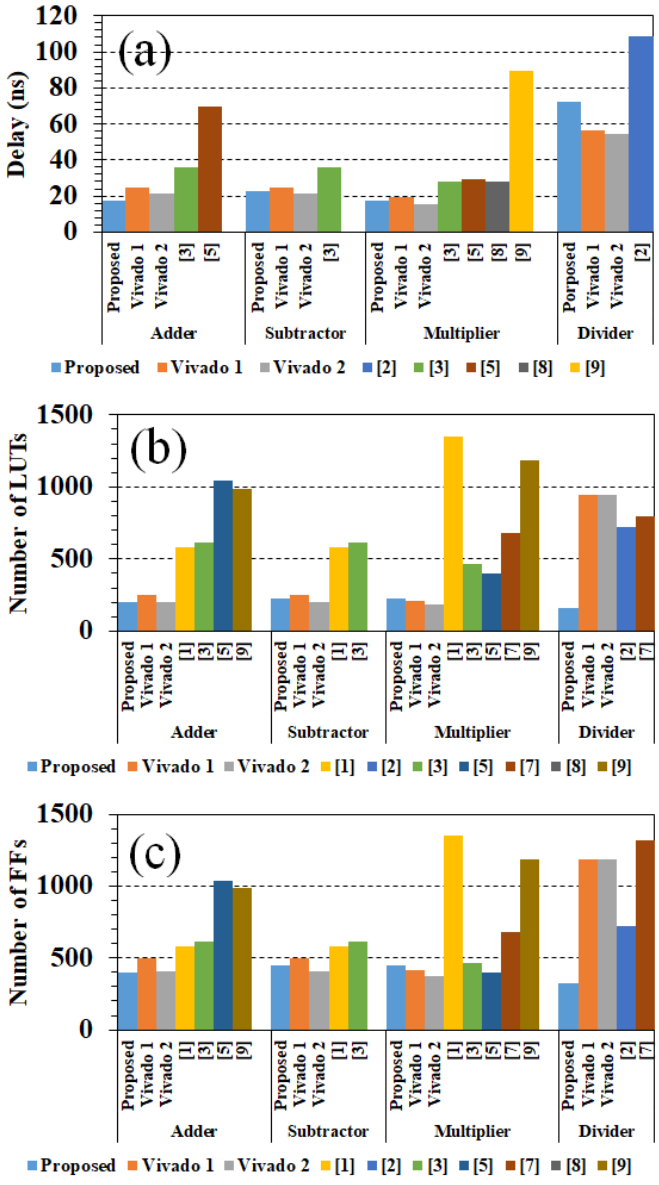

Fig. 4. Comparison of the proposed and the previous basic arithmetic functions. (a) Delay. (b) LUTs. (c) FFs.

TABLE I
COMPARISONS OF THE PROPOSED AND THE PREVIOUS BASIC ARITHMETIC AND TRIGONOMETRIC FUNCTIONS

Functions	Clock Cycles	Freq (MHz)	Delay (ns)	DSPs (%)	Slices (%)	Power (W)	FPGA	Method
Adder								
Proposed	**4**	**200**	**17.5**		**0.07**	**0.35**	**Virtex-7**	**FSM-Vivado (2015.4)**
Proposed	**4**	**200**	**17.5**		**0.01**	**0.35**	**Virtex-7**	**FSM-Vivado (2018.3)**
Vivado 1	14	540	24.97	0.07	0.08	0.25	Virtex-7	Performance (2015.4)
Vivado 2	12	540	21.27	0.07	0.07	0.25	Virtex-7	Resources (2015.4)
[1]	4+2*				5.68		Virtex-II	Pipeline
[3]	4+2*	167	35.98		0.91		Virtex-II	Pipeline
[5]	16	230	69.44		0.93		Virtex-II	Pipeline
[9]		195			1.12		Virtex-II	Pipeline
Subtractor								
Proposed	**4**	**200**	**17.5**		**0.07**	**0.36**	**Virtex-7**	**FSM-Vivado (2015.4)**
Proposed	**4**	**200**	**17.5**		**0.01**	**0.36**	**Virtex-7**	**FSM-Vivado (2018.3)**
Vivado 1	14	540	24.97	0.07	0.08	0.25	Virtex-7	Performance (2015.4)
Vivado 2	12	540	21.27	0.07	0.07	0.25	Virtex-7	Resources (2015.4)
[1]	4+2*				5.68		Virtex-II	Pipeline
[3]	4+2*	167	35.98		0.91		Virtex-II	Pipeline
Multiplier								
Proposed	**4**	**200**	**17.5**	**0.07**	**0.07**	**0.36**	**Virtex-7**	**FSM-Vivado (2015.4)**
Proposed	**4**	**200**	**17.5**	**0.07**	**0.07**	**0.36**	**Virtex-7**	**FSM-Vivado (2018.3)**
Vivado 1	11	540	19.42	0.07	0.07	0.35	Virtex-7	Performance (2015.4)
Vivado 2	9	540	15.72	0.07	0.06	0.37	Virtex-7	Resources (2015.4)
[1]	3+2*				13.16		Virtex-II	Pipeline
[3]	3+2*	178	28.08		0.68		Virtex-II	Pipeline
[5]	7	240	29.12		0.36		Virtex-II	Pipeline
[7]		516					Virtex-7	CORDIC
[8]			27.17				Virtex-7	Vedic
[9]	16	176	89.6		1.34		Virtex-II	Pipeline
Divider								
Proposed	**15**	**200**	**72.5**		**0.05**	**0.36**	**Virtex-7**	**FSM-Vivado (2015.4)**
Proposed	**15**	**200**	**72.5**		**0.06**	**0.37**	**Virtex-7**	**FSM-Vivado (2018.3)**
Vivado 1	31	540	56.4		0.31	0.27	Virtex-7	Performance (2015.4)
Vivado 2	29	520	54.7		0.3	0.27	Virtex-7	Resources (2015.4)
[2]	14	129	108.5		1.06		Virtex-II	BRAM-Taylor
[7]		568					Virtex-7	CORDIC
[9]	37	120	307.1		4.47		Virtex-II	Pipeline
sine								
Proposed	**62**	**200**	**307.5**	**0.07**	**0.53**	**0.39**	**Virtex-7**	**FSM-Taylor-Vivado (2015.4)**
Proposed	**62**	**200**	**307.5**	**0.07**	**0.52**	**0.39**	**Virtex-7**	**FSM-Taylor-Vivado (2018.3)**
[11]	40		109		64.84		Virtex-II	Table-based
[12]	46-82	86	534.52		1.52		Virtex-5	BRAM-CORDIC
[12]	11-27	95	115.72		0.87		Virtex-5	BRAM-Taylor
cosine								
Proposed	**56**	**200**	**277.5**	**0.07**	**0.48**	**0.39**	**Virtex-7**	**FSM-Taylor-Vivado (2015.4)**
Proposed	**56**	**200**	**277.5**	**0.07**	**0.51**	**0.38**	**Virtex-7**	**FSM-Taylor-Vivado (2018.3)**
[11]	40		109		64.84		Virtex-II	Table-based
[12]	46-82	86	534.52		1.52		Virtex-5	BRAM-CORDIC
[12]	11-27	95	115.72		0.87		Virtex-5	BRAM-Taylor
arcsine								
Proposed	**67**	**200**	**332.5**	**0.07**	**0.49**	**0.39**	**Virtex-7**	**FSM-Taylor-Vivado (2015.4)**
Proposed	**67**	**200**	**332.5**	**0.07**	**0.46**	**0.38**	**Virtex-7**	**FSM-Taylor-Vivado (2018.3)**
arccosine								
Proposed	**75**	**200**	**372.5**	**0.07**	**0.56**	**0.39**	**Virtex-7**	**FSM-Taylor-Vivado (2015.4)**
Proposed	**75**	**200**	**372.5**	**0.07**	**0.53**	**0.38**	**Virtex-7**	**FSM-Taylor-Vivado (2018.3)**

*The two extra clock cycles is for normalizing and rounding which is a separate module, a step that is already included in all other methods.

But, because these methods are pipelined, the throughput outperforms the sequential methods for applications with many consecutive multiplications. When examining the area in Fig. 4(b, c), our method has the second smallest area usage behind the Vivado IP. It is also important to note that based on Table I, our method utilizes the Digital Signal Processing (DSP) modules, which is considered to be an advantage over the previous works since using LUTs and FFs for the multiplication functions leads to inefficient use of the FPGAs resources.

According to Fig. 4(a), our proposed division function outperforms the previous works in terms of delay. However, the delay is not better than the built-in Vivado IP (in either optimization mode) due to the operating frequency. The proposed division function also benefits from a lower resource utilization than all previous works, including the built-in Vivado IP, as seen in Fig. 4(b, c).

B. Trigonometric Functions

These functions are compared with the two existing methods for single-precision sine and cosine functions in previous literature. To the best of our knowledge, no work has been proposed for arcsine and arccosine single-precision floating-point functions. For trigonometric functions, our proposed methods provide improvements over existing methods in terms of operating frequency as shown in Table I. It is observed that the proposed methods are faster than the first design of [12] and outperform [11] in terms of the number of LUTs and FFs. The reason why the proposed method in [12] utilizes fewer LUTs and FFs is that it uses BRAMs, which is another factor that should be considered in the comparison.

To compare memory utilization, the proposed work in [11] is table-based and [12] uses ROMs for storing the division factors ($\frac{1}{n!}$) for the Taylor series method and pre-computed angles in the CORDIC method. Higher accuracy requires larger BRAMs to store larger pre-computed values. This becomes a factor when higher accuracies are required.

BRAM requirements for saving the results for one trigonometric function (e.g. sine) in a 32-bit floating-point format, using a degree-0 table, and a precision of 10^{-5} for degrees ranging between $[0 - 2\pi]$, is 9.58 MB. This is not feasible on a Virtex-7. For cases where the degrees between $[0 - \frac{\pi}{4}]$ are stored, memory utilization would be 1.16 MB per function. In such cases, if the degree is not in the aforementioned range, the result would be computed based on the pre-stored degrees. This leads to a higher delay in computation time which is clearly not desirable. However, in our study, BRAMs are not utilized and the Taylor series components of each degree are computed using our proposed functions.

V. Conclusion

This paper explores the implementation of a single-precision floating-point library on FPGAs using an FSM-based method. In this regard, new methods were proposed for different floating-point functions including addition/subtraction, multiplication, division, and trigonometric functions. The basic algorithm was utilized for addition/subtraction, multiplication, and division functions and the trigonometric functions were implemented using Taylor series expansion. The presented methods of the current study were compared to the existing methods in terms of delay, area, operating frequency, and memory utilization. It was observed that for the basic arithmetic functions, the proposed methods led to better performance in terms of delay when compared to previous methods. However, for some functions, the Vivado IP showed better performance than the proposed methods. For the trigonometric functions, the proposed method is not the best among all the previous methods. This is because the proposed functions of previous studies used BRAMs to decrease the delay and area in terms of the number of LUTs and FFs. But in the current study, BRAMs are not utilized in any of the proposed functions to save them for the rest of the design, if needed.

From the comparisons, it can be concluded that for the pipelined methods, while the throughput is improved, the execution time of each individual instruction is increased due to the pipeline overhead. This indicates that the pipelined method should be used in applications where the increased throughput compensates for the overhead of the pipeline. Our method aims to reduce the latency of each individual instruction, rather than throughput. The built-in Vivado IP works at a high frequency which makes it an appropriate candidate for high-speed applications. However, it should be noted that for designs with other components such as BRAMs, Ethernet IPs, and custom IPs, this high operating frequency of the built-in floating-point Vivado IP may not be consistent with the other IPs in the block design because they may not tolerate such a high frequency. The advantage of the proposed methods over the Vivado IP is that they can easily be embedded locally into FSM-based designs and any custom IP to prevent back and forth data transfers. Our proposed methods are also a proper candidate for applications where latency is more critical than throughput. Additionally, none of our proposed methods utilize critical BRAM resources. This is important for applications, such as image processing, that use BRAMs for buffering streaming data.

Our basic arithmetic methods outperform the next best reported open-source libraries for different functions in terms of delay in a range of 33.17% to 51.36%. No estimates on power consumption were provided for the open-source libraries used in our comparisons. Compared to the built-in Vivado IP, our functions consistently have a higher power usage which would be a factor in low-power designs. For the trigonometric functions, our clock frequency outperforms all previous works by 110.52% or more which is critical for higher speed designs.

Acknowledgment

This work is funded by the Department of Energy's Kansas City National Security Campus, operated by Honeywell Federal Manufacturing & Technologies, LLC under contract number DE-NA0002839.

REFERENCES

[1] P. Belanović and M. Leeser, "A library of parameterized floating-point modules and their use," International Conference on Field Programmable Logic and Application, Springer, Berlin, Heidelberg, pp. 657-666, 2002.

[2] W. Xiaojun, Sh. Braganza, and M. Leeser, "Advanced components in the variable precision floating-point library," 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 249-258, 2006.

[3] W. Xiaojun and M. Leeser, "Vfloat: a variable precision fixed-and floating-point library for reconfigurable hardware," ACM Transactions on Reconfigurable Technology and Systems (TRETS), vol. 3, no. 3, p.16, 2010.

[4] F. Xin and M. Leeser, "Open-source variable-precision floating-point library for major commercial fpgas," ACM Transactions on Reconfigurable Technology and Systems (TRETS), vol. 9, no. 3, p. 20, 2016.

[5] G. Gokul, L. Zhuo, S. Choi, and V. Prasanna, "Analysis of high-performance floating-point arithmetic on FPGAs," IEEE 18th International Parallel and Distributed Processing Symposium, Proceedings, p. 149, 2004.

[6] H. Shaobing, L. Yu, F. Han, and Y. Luo, "A pipelined architecture for user-defined floating-point complex division on FPGA," IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE), pp. 1-4, 2017.

[7] L. B. Linlin Fang, Y. Xie, H. Chen, and L. Chen, "A unified reconfigurable floating-point arithmetic architecture based on CORDIC algorithm," IEEE International Conference on Field Programmable Technology (ICFPT), pp. 301-302, 2017.

[8] K. Ravi Kishore, L. Boppana, and S. S. Yenamachintala, "FPGA implementation of vedic floating-point multiplier," IEEE International Conference on Signal Processing, Informatics, Communication and Energy Systems (SPICES), pp. 1-4, 2015.

[9] U. Keith, "FPGAs vs. CPUs: trends in peak floating-point performance," Proceedings of the ACM/SIGDA 12th international symposium on Field programmable gate arrays, pp. 171-180, 2004.

[10] G. Gokul, R. Scrofano, and V. K. Prasanna, "A library of parameterizable floating-point cores for FPGAs and their application to scientific computing," Proceedings of the International Conference on Engineering Reconfigurable Systems and Algorithms, pp. 137-148. 2005.

[11] D. Jérémie and F. de Dinechin, "Floating-point trigonometric functions for FPGAs," IEEE International Conference on Field Programmable Logic and Applications, pp. 29-34, 2007.

[12] M. Daniel, D. F. Sánchez, C. H. Llanos, and M. Ayala-Rincón, "Tradeoff of FPGA design of floating-point transcendental functions," IEEE 17th IFIP international conference on very large scale integration (VLSI-SoC), pp. 239-242, 2009.

[13] A. Eski, D. Komici, and O. Zavalani, "Evaluation of CORDIC algorithm for the processing of sine and cosine functions," International Journal of Business and Management Invention (IJBMI), vol. 6, no. 3, pp. 50-54, 2017.

[14] J. Fredrik, "Efficient implementation of elementary functions in the medium-precision range," IEEE 22nd Symposium on Computer Arithmetic, pp. 83-89, 2015.

[15] M. A. Ait, and A. Addaim, "Optimized method for sine and cosine hardware implementation generator, using CORDIC algorithm," International Journal of Applied Engineering Research, vol. 13, no. 1, pp. 21-29, 2018.

[16] Zh. Baozhou, Y. Lei, Y. Peng, and T. He, "Low latency and low error floating-point sine/cosine function based TCORDIC algorithm," IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 64, no. 4, pp. 892-905, 2016.

[17] T. Arnaud, "Hardware operator for simultaneous sine and cosine evaluation," IEEE International Conference on Acoustics Speech and Signal Processing Proceedings, vol. 3, pp. III-III, 2006.

[18] J. L. Hennessy and D. A. Patterson, "Computer architecture: a quantitative approach," Elsevier, 2011.

[19] D. Kincaid, D. R. Kincaid, and E. W. Cheney, "Numerical analysis: mathematics of scientific computing," American Mathematical Society, vol. 2, 2009.

[20] J. E. Volder, "The CORDIC trigonometric computing technique," IRE Transactions on electronic computers, vol. 8, no. 3, pp. 330-334, 1959.

[21] D. Z. Turner, "Digital Image Correlation Engine (DICe) reference manual," Sandia Report, SAND2015-10606 O, 2015.

[22] F. De Dinechin, and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," IEEE Design & Test of Computers, vol. 28, no. 4, pp. 18-27, 2011.