Noname manuscript No.
(will be inserted by the editor)

Series Expansion based Efficient Architectures for Double Precision Floating Point Division

Manish Kumar Jaiswal, Ray C.C. Cheung,
M. Balakrishnan and Kolin Paul

Received: date / Accepted: date

Abstract Floating-point division is a complex operation among all floating-point arithmetic; it is also an area and performance dominating unit. This paper presents double-precision floating-point division architectures on FPGA platforms. The designs are area-optimized, running at higher clock-speed, with less latency, and are fully pipelined. Proposed architectures are based on the well-known Taylor-series expansion, using relatively smaller amount of hardware in-terms of memory (initial look-up table), multiplier blocks and slices. Two architectures have been presented with various trade-offs amongst area, memory and accuracy. Designs are based on the use of the partial block multipliers (PBM), in order to reduce hardware usage while minimizing the loss of accuracy. All the implementations have been targeted and optimized separately for different Xilinx FPGAs to exploit their specific resources efficiently. Compared to previously reported literature, the proposed architectures require less area, reduced latency, with the advantage of higher performance gain. The accuracy of the designs have been both theoretically analyzed and validated using random test cases.

Keywords Floating point arithmetic · Division · Partial Block Multiplication · Taylor Series Expansion · Karatsuba Method · Accuracy · Arithmetic · FPGA.

1 Introduction

Floating point arithmetic is a core function used in a large set of scientific and engineering applications [17, 23, 29]. Its large dynamic range and convenient scaling of the numbers in its range provides a convenient platform for designers to realize

Manish Kumar Jaiswal and Ray C.C. Cheung
Department of Electronic Engineering, City University of Hong Kong, Hong Kong
E-mail: manish.kj@my.cityu.edu.hk, r.cheung@cityu.edu.hk

M. Balakrishnan and Kolin Paul
Department of Computer Science Engineering, Indian Institute of Technology Delhi, India
E-mail: mbala@iitd.ernet.in, kolin@iitd.ernet.in

their algorithms. On the other hand, the complexity involved in implementing these arithmetic operations for floating point numbers in hardware is an issue. Among the basic floating point operations (add, subtract, multiply, divide), division is generally the most difficult (inefficient) to implement in hardware. Division is a fairly common operation in many scientific and signal processing applications.

The IEEE-754 standard [1,2] for floating point defines the format of the numbers, and also specifies various rounding modes that determine the accuracy of the result. For many signal processing, and graphics applications, it is acceptable to trade off some accuracy [20] (in the least significant bit positions) for faster and better optimized implementations. In the past few decades several works have been dedicated to performance improvement of floating point computations, both at algorithmic and architecture level [4, 11, 12, 15, 26, 27, 29, 30]. Many have also given prime attention to FPGA-based implementations [7, 10–12, 14, 25, 30, 33].

A set of related work has also focused on designing efficient division implementations. In general the implementation of division operation falls in three categories: digit recurrence, multiplicative-based, and approximation techniques [22]. Digit Recurrence (DR) is an iterative method with several variations. The most widely used digit recurrence method is SRT (Sweeney, Robertson, and Tocher) method. This method is well suited for smaller operands, specially up to single precision, because of less area requirement and circuit complexity. However, for large operands, this method needs higher latency and performance penalty, when compared to multiplicative-based or approximation techniques, though with less required area. Several researchers have focused their work using this method or its derivatives. Wang *et. al.* (SRT) [34], Thakkar *et. al.* (DR) [28], Hemmert *et. al.* (SRT) [13] are some of the works which use this method.

However, the multiplicative-based implementation is based on an initial approximation of the inverse of the divisor and iterative improvements of this initial approximation, and it is based on multipliers. The famous methods in this category are Newton-Raphson (NR) method [5, 21] and division by convergence (DC) algorithm also known as Goldschmidts (GS) division [9]. Several implementations based on this include works of Antelo *et. al.* (NR) [5], Venishetti *et. al.* (DC) [31], Govindu *et. al.* (NR) [10], Daniel *et. al.* (NR, GS) [7], and Pasca (NR) [24]. This method requires large amounts of logic (area) in terms of memory and multipliers, but is better in terms of latency and performance vis-a-vis digit recurrence method. The approximation method comes in to play when the desired level of accuracy is low, and generally falls in two categories: Direct Approximation (using look-up tables) and linear/polynomial approximation (using small look-up tables and/or partial product arrays) [16, 19, 33]. All these methods primarily vary in terms of area, speed, latency and/or accuracy, and mainly targeted the normalized implementation. In literature, most of the previous works require large look-up tables, along with wider multipliers, which affect the area and performance, with varying accuracies.

The proposed architecture in this work is based on the well-known Taylor-series expansion methodology. Based on the design metrics (discussed in section(2)), there is variation in the different hardware resources. So, two architectures have been proposed on same principle, to have an idea of trade-offs between various hardware resources like BRAM, multiplier blocks, slices. All the required intermediate multi-

pliers have been optimized for their accuracy requirement (at their respective stages), which results in smaller area, shorter delay, and accuracy up to the desired level (accuracy trade off). Multipliers based on the partial block multiplier (PBM) have been utilized, to save hardware with minimal accuracy effect. A detailed error analysis is presented to verify the accuracy of the designs. The design is currently aimed for normalized numbers, and all exceptional cases are detected at input and output. The comparison with the best state-of-the-art work in the literature shows that our proposed architectures are able to achieve better efficiency with a clear mechanism of area-accuracy trade-offs. We have used Xilinx ISE synthesis tool, ModelSim SE simulation tool, and Xilinx Virtex2-Pro, Virtex-4 and Virtex-5 FPGA platforms for evaluation of our proposed architectures and comparisons with other work.

This work builds on the work presented by Jaiswal *et. al.* [18]. Initially, the basic idea has been generalized and elaborated in much detail with the various possible hardware and accuracy trade-offs. This will help to further assess and opt for different architectural composition to achieve required accuracy with available hardware. We have done a design space exploration of the proposed approach. This manuscript explores two architectures. Both have been designed for two different latencies, to show the various hardware variations (slice, BRAMs, MULT18x18/DSP48). This design space exploration can further leads to some different architectures depending on the users / applications requirements on hardware usage and accuracy. Further, all the proposed designs have been targeted and optimized for different Xilinx FPGA platforms to exploit their specific resources and IPs. As a significant contribution, a detail theoretical and experimental error analysis has been presented for all the proposed architectures, to establish their potential. An extensive comparison with the several recent state-of-the-art division methodologies available in the literature has been presented and discussed comprehensively with different metrics including hardware utilization, performance and accuracy. Compared to the previous works reported in the literature, the proposed modules achieve higher performance with relatively lower latency and area reduction in terms of number of multiplier blocks as well as number of block memory reduces with less slices.

The main contributions of this paper can be summarized as follows:

1. Proposed an approach for double precision floating point division, with two architectures implemented on a range of FPGA platforms.
2. Error performance of both designs have been analyzed theoretically, as well as using large simulation.
3. Extensively compared with state-of-the-art methods published previously in literature.
4. Proposed architectures have improved area and speed numbers, with similar accuracy standard.

This paper is organized as follows. The next section 2 explains our design approach. Section 3 discusses the complete implementation with all required processing in floating point division operation. Section 4 discusses error analysis, the error cost of Partial Block Multipliers (PBMs) and total error. Section 5 has included the implementation results, while comparison with previously reported implementations along with discussion is included in Section 6. Finally, paper concludes in Section 7.

2 Design Approach

A double precision floating point number is represented as,

$$\overbrace{Sign-bit}^{1-bit}\overbrace{exponent}^{11-bits}\overbrace{mantissa}^{52-bits}$$

In order to explain the floating point division in detail, let X be the dividend and Y the divisor. To obtain the resultant quotient Q, the following operation is required.

$$Q = \frac{X}{Y} \tag{1}$$

The quotient Q is also a floating point number, whose

- Sign-bit is the `XOR` operation of the sign-bit of X and Y.
- Exponent is the difference of the exponent of X and Y with proper `biasing`.
- Mantissa is obtained by the division of the X-mantissa by the Y-mantissa.
- Finally, rounding and normalization of the mantissa division and adjustment of the output exponent are applied.

The sign and exponent manipulations are relatively trivial operations. The mantissa processing (division) is the most critical step in this arithmetic operation. It has a major impact on the required area and performance speed. The present method performs this mantissa processing as below.

Let x represent the mantissa of X, and y represent mantissa of Y. Let q be the division result, which can be computed as follows,

$$\begin{aligned} q &= \frac{x}{y} = x \times \frac{1}{y} \\ &= x \times \frac{1}{a_1 + a_2} = x \times (a_1 + a_2)^{-1} \end{aligned} \tag{2}$$

For this purpose, we have partitioned the denominator mantissa in two parts, m-bit a_1 and remaining as a_2. a_1 is used as an address input to a look-up table (memory) fetch some pre-computed value of a_1^{-1}.

Thus, we have

$$(a_1 + a_2)^{-1} = a_1^{-1} - a_1^{-2}.a_2 + a_1^{-3}.a_2^2 - a_1^{-4}.a_2^3 + \cdots \tag{3}$$

By inspecting the terms of the above equation 3, the content of each of the terms in the equation will look like:

$$
\begin{aligned}
a_1^{-1} &= X.\overbrace{XXXXXXXX}^{significant\ bits} \\
a_1^{-2}.a_2 &= 0.\overbrace{00\cdots00}^{m-zero\ bits}\ \overbrace{XX\cdots XX}^{significant\ bits} \\
a_1^{-3}.a_2^2 &= 0.\overbrace{00\cdots0\cdots00}^{2m-zero\ bits}\ \overbrace{XX\cdots XX}^{significant\ bits} \\
a_1^{-4}.a_2^3 &= 0.\overbrace{00\cdots0\cdots0\cdots00}^{3m-zero\ bits}\ \overbrace{XX\cdots XX}^{significant\ bits} \\
&\cdots and\ so\ on \\
&\text{where } m \text{ is the number of bits of } a_1.
\end{aligned} \tag{4}
$$

In the light of the above equation, the higher order terms contribution to the main result diminishes. Only the initial few terms significantly contribute to the final result (depending on the precision requirement). As a result, based on the precision choices, we can select suitable number of terms for calculating $(a_1+a_2)^{-1}$, (based on the value of m). For double precision accuracy requirement (2^{-53}), for a given m, the number of terms (N) can be decided by following in-equality,

$$
\begin{aligned}
|E_N| &= |a_1^{-(N+1)}.a_2^N(1-a_1^{-1}.a_2+a_1^{-2}.a_2^2-a_1^{-3}.a_2^3-\cdots)| \\
&= |\frac{a_1^{-(N+1)}.a_2^N}{1+a_1^{-1}.a_2}| \leq 2^{-53}
\end{aligned} \tag{5}
$$

where, E_N is composed by all the ignored terms. For maximum error, denominator of eq.(5) should be minimum and numerator should be maximum. So, with most pessimistic estimation, for minimum denominator, let $(1+a^{-1}.a2) \approx 1$, and for maximum numerator $a_1^{-1}=1$, and thus,

$$
|E_N|_{max} = |a_2^N| \leq 2^{-53} \tag{6}
$$

A variation on value of m and required number of terms is shown in Table 1. For a given accuracy requirement, as the value of m increases, the number of required terms decreases. On the contrary, the amount of memory address space for look-up table increases exponentially. The number of terms used for a given m directly decides the amount of logic needed for different multiplications, additions and subtractions. The more the number of terms are there, the more hardware it needs, and further more number of pipeline stages, would be needed to meet a given performance requirement. Thus, from the hardware implementation point of view, the value of m specifically determines the total hardware composition for a given accuracy requirement. So, based on the value of m we will have a trade-off between the required memory space for pre-computed look-up table, and other hardware resources plus latency of the design.

Table 1: Required numbers of terms (N) and Look-up Table Address Space for a given m, needed for Double Precision Accuracy

m	No. of terms (N)	Max Absolute Error		Look-up Table
7	9	a_{2max}^{9}	5.551 E-17	$2^6(64)$
9	7	a_{2max}^{7}	1.387 E-17	$2^8(256)$
11	6	a_{2max}^{6}	8.673 E-19	$2^{10}(1k)$
13	5	a_{2max}^{5}	8.673 E-19	$2^{12}(4k)$
15	4	a_{2max}^{4}	1.387 E-17	$2^{14}(16k)$
17	4	a_{2max}^{4}	5.421 E-20	$2^{16}(64k)$
19	3	a_{2max}^{3}	5.551 E-17	$2^{18}(256k)$

For double precision requirements, we consider two values of m namely, 9 and 13, and do a design space exploration. With $m = 9$, a small look-up table is required but with more number of terms. However, for $m = 13$, a bigger look-up table is required with relatively, less number of terms for computation. This gives us an idea of the variation of different logic (multipliers and memories) in both designs. However, more higher value of m leads to exponential increase in usage of memory, and more lower value leads to more computational logic (specially in terms of required multipliers). So, the $9 \leq m \leq 15$ would be more balance condition compare to other either side values. We have selected two of them for design space exploration.

2.1 Case-1: $m = 9$ *bits*

For $m = 9$ *bits* (including 1 hidden bit) of a_1, seven terms (up to $a_1^{-7}.a_2^6$) from series expansion have been taken for the purpose. We have further simplified the selected terms in such a way that helps use less hardware with low latency and good accuracy.

The simplification of all the selected terms are performed as below,

$$\begin{aligned} q &= x \times [a_1^{-1} - a_1^{-1}\{(a_1^{-1}.a_2 - a_1^{-2}.a_2^2) \\ &\quad \times (1 + a_1^{-2}.a_2^2 + a_1^{-4}.a_2^4)\}] \\ &= x.a_1^{-1} - x.a_1^{-1}\{(a_1^{-1}.a_2 - a_1^{-2}.a_2^2) \\ &\quad \times (1 + a_1^{-2}.a_2^2 + a_1^{-4}.a_2^4)\} \end{aligned} \tag{7}$$

Though we can simplify the above equations even further, it will affect the area, latency and accuracy of the final result. The accuracy is affected due to the fact that floating point operations are not completely associative, i.e. $u(v+w)$ may not be exactly equal to $(uv+uw)$. This is mainly due to the finite number of bits used to represent the numbers.

The error cost in this due to restricted number of terms (N) and m can be obtained from Table 1, which is $\leq 1.387\ E-17$, is with in double precision accuracy requirement.

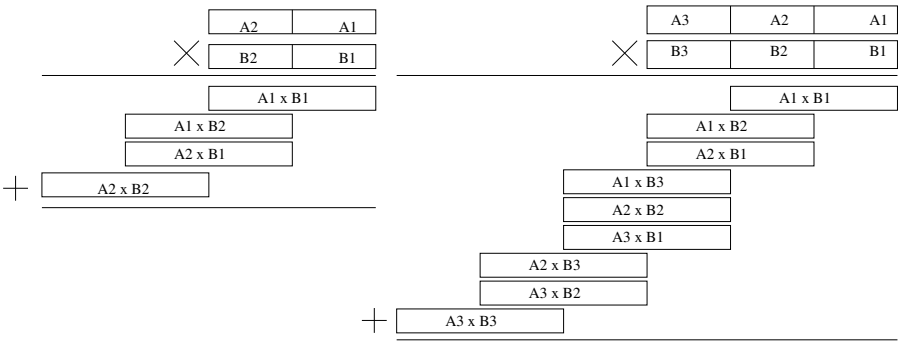

Fig. 1: Block multiplier for 2 and 3 blocks

2.2 Case-2: $m = 13$ *bits*

Likewise, for $m = 13$ *bits* (including 1 hidden bit) of a_1, five terms (up to $a_1^{-5}.a_2^4$) from series expansion have been selected. This again has been simplified as follows,

$$q = x.a_1^{-1} - x.a_1^{-1}\{(a_1^{-1}.a_2 - a_1^{-2}.a_2^2)(1 + a_1^{-2}.a_2^2)\} \tag{8}$$

Here, also the maximum error is within the acceptance limit of the required precision of double precision accuracy.

2.3 Partial Block Multiplication (PBM) Optimization

In order to implement the eq.(7,8) for mantissa division processing, we need a set of multipliers along with some adders and subtractors. The size of each of the operands in each multiplication is quite large ($\geq$ 51-bit), and we do need a large number of multiplier blocks in FPGA to implement all these multiplications. But, as we have seen in eq.(4), terms are associated with the leading zeros, and so, we can eliminate some of the multiplier blocks.

Another point of interest is that after all the processing, the desired output will need only 53-bit representation. In view of this, first, we will consider Fig. 1 for block multiplication of two operands. In multiplication, if we need only some of the most significant bits (MSBs) of the result to be accurate, we can discard some of the lower order multiplier blocks (depending on the precision requirements). For example, if we do multiplication of two 51-bit operands using three block partitioning, each of 17-bits, just by using 6 multiplier blocks (by ignoring top three multiplier blocks: $A1 \times B1$, $A1 \times B2$, and $A2 \times B1$ in Fig. 1), we can get a result that has 50-bit accuracy. Thus, it has a $\approx 33\%$ hardware saving with small loss of precision.

We have used the above discussed optimization approach (partial block multiplication, PBM) to perform all the multiplications. The details on these implementations are explained in the next sections, along with the required processing for floating point division.

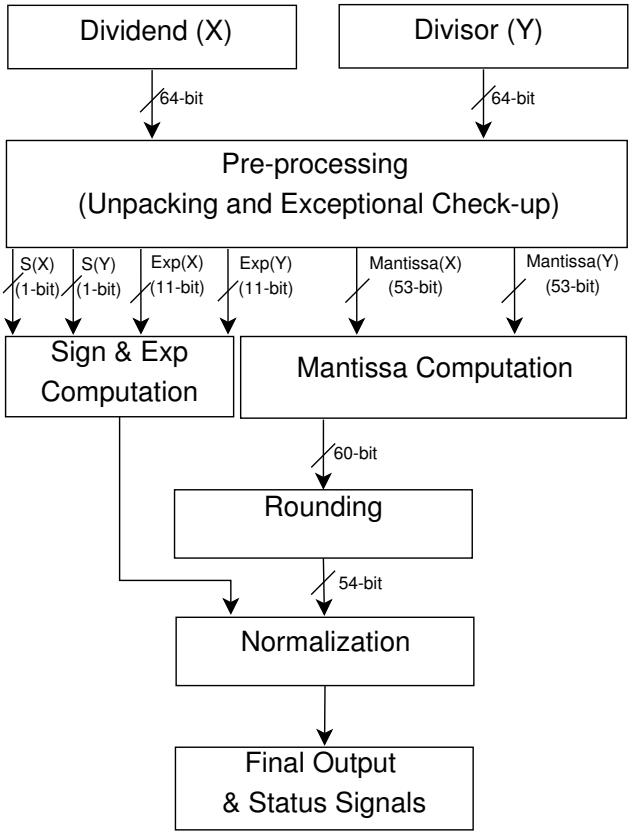

Fig. 2: Architecture of Floating Point Division

3 Design Implementation

In this section we discuss the implementation details of both FP division designs. The implementation work flow of design is shown in Fig. 2. A floating point arithmetic operation generally works separately on the sign, exponent and mantissa part and finally combines them after rounding and normalization to get the final result. Likewise, we have performed similar operations as follows.

The sign bit implementation of output quotient requires very simple logic, as it is only an `XOR` operation of the input operands sign bits.

$$Sign\_out = Sign\_in1 \oplus Sign\_in2 \tag{9}$$

The exponent computation of the output quotient is done in two phases. In the initial phase, a temporary exponent is computed by taking the difference of the dividend exponent and divisor exponent, with proper `BIAS` adjustments. In the case of double precision floating point numbers, the `BIAS` is equal to 1023, and generally computed

as $(2^{exp\_bit-1}-1)$.

$$\begin{aligned} Exp\_out\_tmp &= (Exp\_dividend - Bias) \\ &\quad - (Exp\_divisor - Bias) \\ &= Exp\_dividend - Exp\_divisor \end{aligned} \tag{10}$$

The next phase of the exponent computation occurs after the normalization of the mantissa. In this phase, the temporary exponent is adjusted based on the normalization, and finally `biased` to produce the final exponent result. The mantissa computation is the complex part of this routine. This computation is discussed ahead (in subsection 3.1, occurs in parallel with the sign & exponent computation. After, mantissa computation, the rounding of the mantissa has been performed using round-to-nearest method. Rounding first need to find out the correct rounding position and further requires a 54-bit adder along with some logic for round-bit computation using guard, round and sticky bit. Further, the normalization of mantissa (using right shift, if require), exponent update, and exceptional case status check, results in final outputs.

3.1 Mantissa Division Architectures

Here, we discuss the proposed architectures for the double precision mantissa division operation, for both the cases $m = 9$ and $m = 13$. In the mantissa division architectures shown in Fig. 3 and Fig. 4, for both cases, several stages use the multipliers. As discussed earlier, multipliers use partial block multiplication (PBM) in order to reduce the number of the multiplier blocks. However, it has a minor error overhead, which has been analyzed and discussed in section 4. The architectures of PBMs are discussed in section 3.2.

3.1.1 ***Case-1 :*** $m = 9$ *bits*

The architecture for $m = 9$ *bits* is based on the discussion in section 2.1. The underlying equation(7) to be implemented is reproduced here again for convenience.

$$\begin{aligned} q = x.a_1^{-1} - x.a_1^{-1}\{&(a_1^{-1}.a_2 - a_1^{-2}.a_2^2) \\ &\times (1 + a_1^{-2}.a_2^2 + a_1^{-4}.a_2^4)\} \end{aligned} \tag{11}$$

Here, for the case of $m = 9$ *bits*, a_1 has the form $1.XX$ (in hex) and a_2 will be like $0.00XXXXXXXXXX$ (in hex), where $XX..$ is significant.

The architecture for implementing eq(11) is shown in Fig. 3. In Fig. 3, mantissa is divided into two parts (8-bit a_1 and 44-bit a_2). a_1 has been used to access the pre-computed inverse of the $1.a_1$ (including the hidden bit of the mantissa). The word size of the pre-computed value of a_1 has been kept 53 bits and is stored in a block memory (BRAM) available on the FPGA as a hard IP core. The address space of this BRAM is $2^8 = 256$ words. Further processing involves several multiplications of intermediate terms, along with some addition and subtraction. The size of the multipliers has been

varied depending on the contribution of their result in the final result. Further, the size of the adders and subtractors are relatively longer, to save the precision, as loss in these is more than that of multiplications.

The architecture shown in Fig. 3 consists of 8-stages, each of which have been pipelined further for better performance. The pipeline depth of each stage is based on the type of multipliers, which is discussed later in more detail. Whereas, the pipeline depth of addition and subtraction in stages 4,5 and 8 has been kept 2. Each of the stages 2,3,4,6 and 7 consists of different multipliers.

Now, as soon as we receive the value of a_1, we can get the pre-computed value of a_1^{-1} from the BRAM. The next step involves the computation of $x.a_1^{-1}$ and $a_1^{-1}.a_2$. For the computation of $x.a_1^{-1}$ we have used a 53-bit PBM. The computation of $a_1^{-1}.a_2$ is done by a 51-bit PBM. Since, a_2 contains 8 bits of leading zero (LZ), the product $a_1^{-1}.a_2$ will be appended by 8'h00 LZ. Next processing step is the computation of $a_1^{-2}.a_2^2$. This is the square of the previous stage output, and it has been computed by using 51-bit square PBM. This is mainly the 51-bit multiplier, but due to the special nature of the inputs (same input), here we have saved more multiplier blocks. Further, the product $a_1^{-2}.a_2^2$ will be appended by 16'h0000 LZ for proper decimal point adjustment.

Further, stage-4 computes two terms, $a_1^{-4}.a_2^4$ and $a_1^{-1}.a_2 - a_1^{-2}.a_2^2$. Since the term $a_1^4.a_2^4$ contains 32 leading zeros, and very few parts of it actually contribute to the main result, we compute it using a 34-bit square full block multiplication scheme (section(3.2.4)). This multiplication uses 3-multiplier blocks. Term $a_1^{-1}.a2 - a_1^{-2}.a_2^2$ has been computed using a two stage 60-bit subtractor.

The next step (Stage-5) uses a two-stage 60-bit adder to compute $1 + a_1^{-2}.a_2^2 + a_1^{-4}.a_2^4$. The output of this adder has a special nature. It is in the form of $1. < 15'b0 > XXXX.....XXX$. To exploit the availability of this term, in the stage-6 multiplication, we have used a 51-bit m9-reduced PBM, which is able to further reduce some block multipliers.

In stage-7, we have computed the multiplication of xa_1^{-1} with the output of the last stage multiplier. Finally, this stage-7 output is subtracted from $x.a_1^{-1}$ to get the final mantissa division result. The complete mantissa processing needs 28 18x18 multiplier IP blocks and one BRAM.

After completing the mantissa computation, we round and normalize it to get it back in proper format and then adjust the exponent accordingly, to finalize the output result.

3.1.2 ***Case-2 :*** $m = 13\ bits$

Architecture for this is based on equation(8), repeated below for quick reference. In this case, a_1 is 13 bits (including 1-hidden bit) and is used to fetch the pre-computed look-up table value with 12-bit address space. The form of a_2 in this equation has the 12 leading zero's (LZ).

$$q = x.a_1^{-1} - x.a_1^{-1}\{(a_1^{-1}.a_2 - a_1^{-2}.a_2^2)(1 + a_1^{-2}.a_2^2)\} \tag{12}$$

The proposed architecture to implement eq(12) is shown in Fig. 4. As in the previous case, the computation flow is very straight forward. It requires a BRAM to

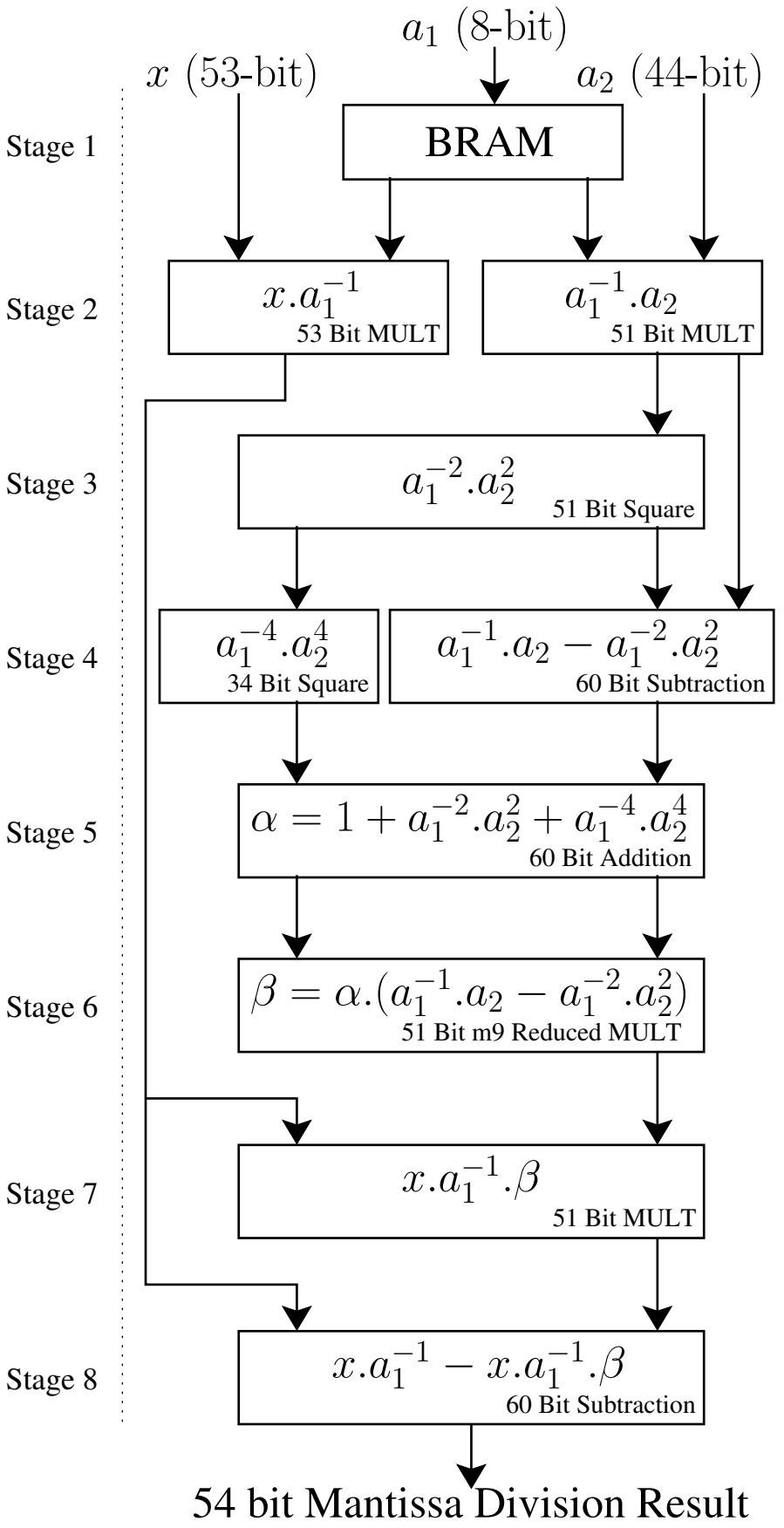

Fig. 3: Architecture of the Mantissa Division : Case-1 ($m = 9$ *bits*)

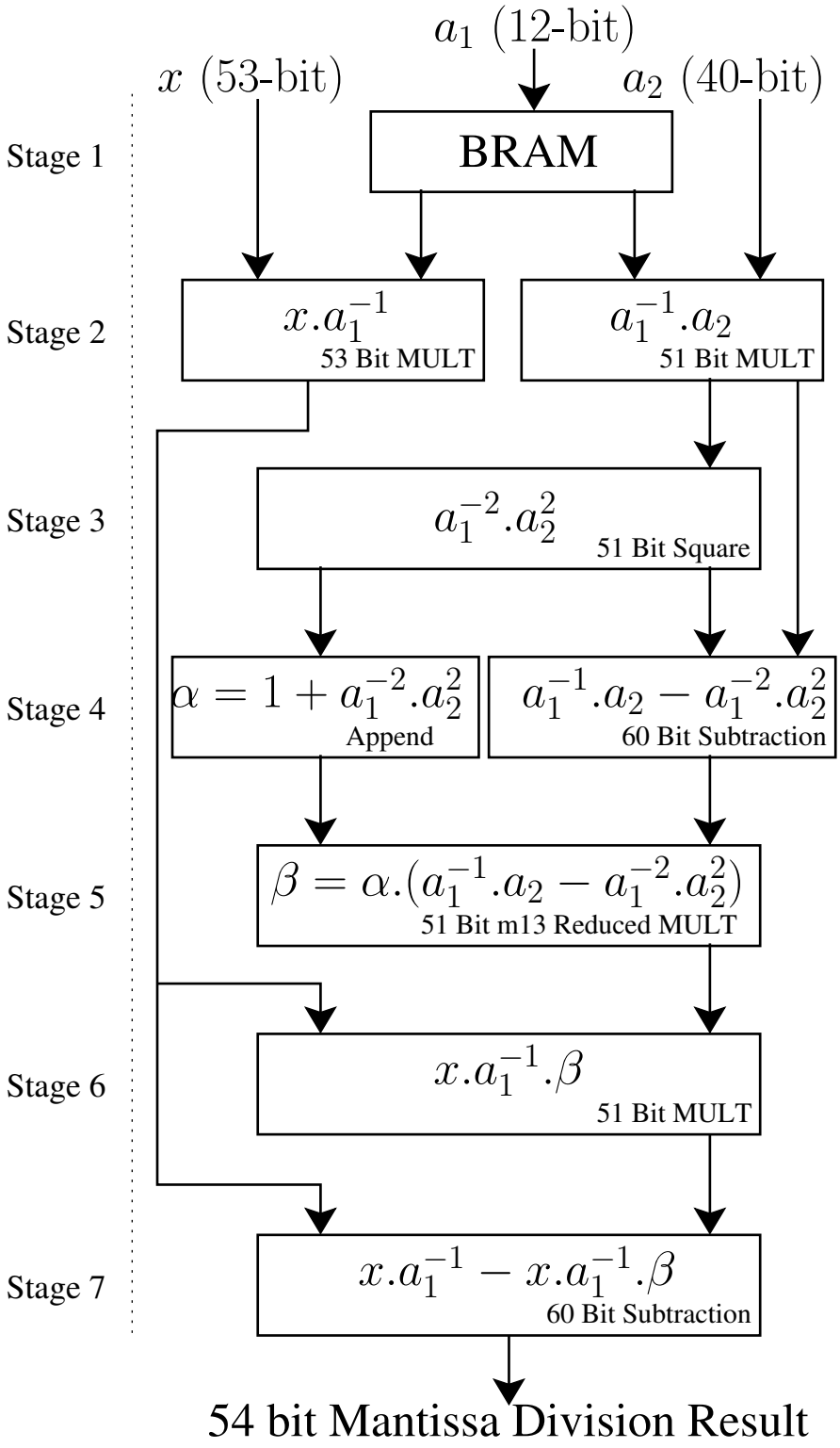

Fig. 4: Architecture of the Mantissa Division : Case-2 ($m = 13$ *bits*)

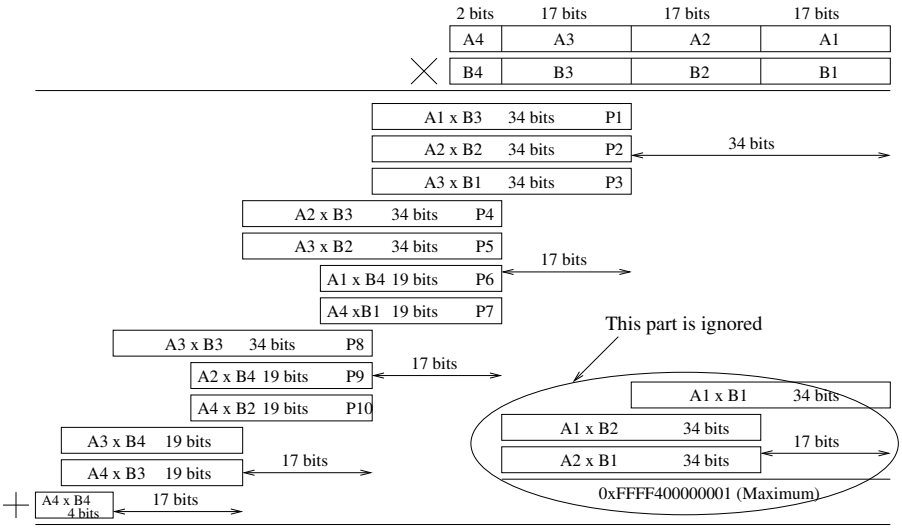

Fig. 5: 53-bit Partial Block Multiplication

fetch the pre-computed data of a_1^{-1} using the 12-bit address space of a_1. For other processing, it needs one 53-bit PBM (for stage-2), two 51-bit PBM (for stage 2 and 6), one 51-bit squarer PBM (stage-3), one 51-bit m13-reduced PBM (stage-5), and two subtractors (stage-4,7).

3.2 Multipliers Architecture

In this section, we discuss the computational flow and architecture of the different partial block multipliers (PBMs) used in the mantissa division architectures.

3.2.1 53-bit PBM

For the computation of $x.a_1^{-1}$ in each case, we have used a 53-bit partial block multiplier (Fig. 5). The 53-bit multiplier in its PBM format, as shown in Fig. 5, uses 6-multiplier blocks along with four 2x19 bit multipliers (implemented with logic slices) and a 2x2 multiplier (need four LUTs). The size of each block has been taken as 17 bits because of availability of 17x17 bit unsigned multiplier IP block on Xilinx FPGAs. The size of this multiplication is longer (53 bits) than the others, because the output of this term mainly contributes to the final result. Only 53 bits of the output result from this stage has been forwarded to the next stages.

The architectural flow of this multiplier is shown in Fig. 6. The shown architecture is targeted for using DSP48, but can be easily used with the simple MULT18x18 IP also. The design uses 6 DSP48/MULT18x18 blocks. When we use DSP48, we can save some logic for adders and registers available on DSP48. However, with MULT18x18 we need some extra logic. The design has a latency of 5.

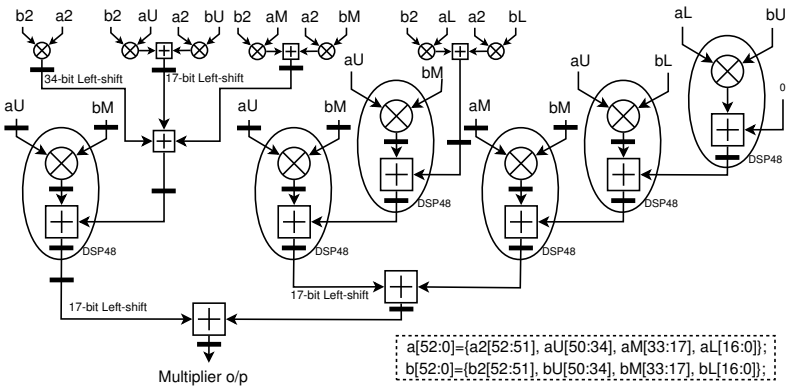

Fig. 6: Architecture of 53-bit PBM

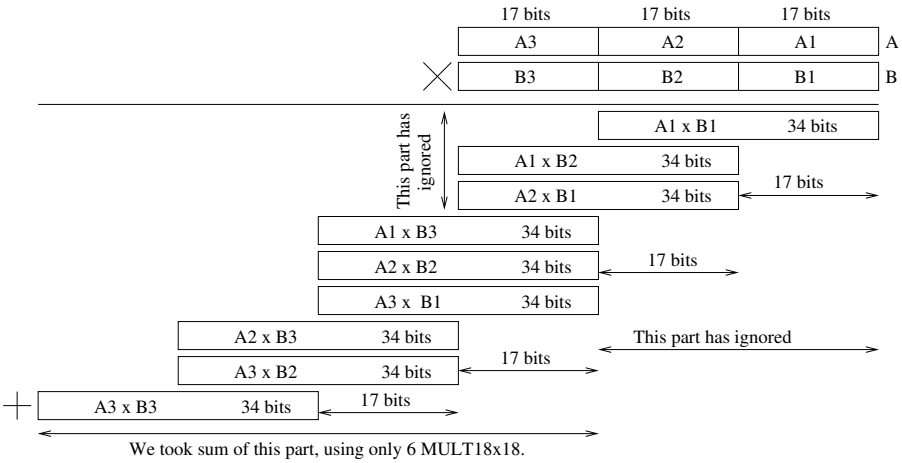

Fig. 7: 51-bit Partial Block Multiplication

3.2.2 51-bit Partial Block Multiplier

Similar to 53-bit PBM, in the 51-bit PBM we have left top three multiplier blocks with computation flow as shown in Fig. 7 and architecture as shown in Fig. 8. As, in previous cases, this also uses only 6-DSP48/MULT18x18 IP blocks, with a latency of 5 clock cycles.

3.2.3 51-bit Partial Block Square

The computation flow of the 51-bit partial block square is similar to the 51-bit PBM as in Fig. 7. However, we can reuse the block $A1 \times B3$ for $A3 \times B1$, and $A2 \times B3$ for $A3 \times B2$. Also, since the $A1 \times B3$ and $A3 \times B1$ are same, it's addition will be just a 1-bit shifting. It is likewise for $A2 \times B3$ and $A3 \times B2$. In this way we can save some

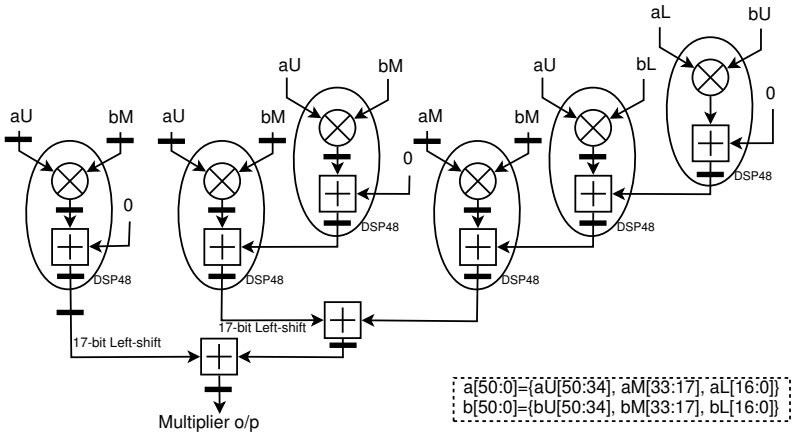

Fig. 8: Architecture of 51-bit PBM

two multiplier blocks and some logic. The architecture of this module is similar to 51-bit PBM (Fig. 8) with above discussed simple modifications. The latency of this design is 4 clock cycles.

3.2.4 34-bit Full Square Multipliers

The square full multiplier follows the conventional trend and use all multiplication blocks, except which are common. For 34-bit multiplier, by using two block method (as in Fig. 1), $A1 \times B2$ and $A2 \times B1$ are identical and one has been removed. Thus, it needs a total of 3-DSP48 blocks. The latency for 34-bit square is 3 clock cycles.

3.2.5 51-bit Reduced Partial Block Multiplier

In the mantissa division architectures, we have a kind of reduced multiplication due to its specific nature of inputs. The computational flow and architecture of these are similar to Fig. 7 and Fig. 8, with the simple modifications, discussed below, for both cases.

For case-1 design, the 51-bit reduced multiplication is used in stage-6. The input α (in Fig. 3) to this stage is of the form $1. < 15'b0 > XXX \ldots XX$. So, if we correlate α with A in Fig. 7, then $A3$ will be equal to $1. < 15'b0 > X$. The multiplication of any term with this quantity needs only one level of AND operation and only one addition. So, the multiplier blocks corresponding to the $A3 \times B1$, $A3 \times B2$ and $A3 \times B3$ have been replaced by simple logic, which further saved three more multiplier blocks. Thus, the architecture is similar to 51-bit PBM (Fig. 8) with these modifications. The latency of this design is also 5 clock cycles.

However for case-2, the reduced multiplication occurs at stage-5. In this case, on correlating α with A in Fig. 7, $A3$ comes out to be $1. < 16'b0 >$, then we do not need any multiplier block for multiplication with this term. Thus, here also we have saved 3 multiplier blocks as above.

3.3 Utilization of 25x18 DSP48 Architecture

The architecture of PBMs in previous subsection are based on 18x18 multiplier IPs available on the FPGAs. However, recent FPGAs have replaced them with 25x18 DSP48 multiplier blocks. To use them, the simplest strategy will be to directly use the previous mentioned architecture, as 25x18 is the super-set of 18x18 multiplier IPs. However, to achieve better optimization, we need to partition the operands differently. For 53-bit PBM, we can partition the first operands as $|24-bit|24-bit|5-bit|$ and second operands as $|17-bit|17-bit|17-bit|2-bit|$. In this case, we can ignore LSBs 5x2, 17x5 and 24x2 multipliers and can compute the multiplication using 5 numbers 24x17 multiplier and one 24x2 multiplier. The error cost in this case will be much less than previous case. Similarly for 51-bit PBM, operands partition can be done as $|24-bit|24-bit|3-bit|$ and $|17-bit|17-bit|17-bit|$, and this also needs only 5 multiplier IPs, compared to 6 using 18x18 IPs. For reduced 51-bit PBM case we will have similar benefit as in previous one, since one operand still would have similar partitioning format. However, for square 51-bit PBM, architecture using 18x18 multiplier IPs will be more area efficient compared to 25x18 IPs, because of same input operands. Thus, the proposed mantissa division architecture can save another 4 multiplier IP blocks using 25x18 DSP48 IPs vis-a-vis using 18x18 IPs, in both cases.

4 Error Analysis

There are three possible source of errors in the proposed architectures. First one is E_N, the error caused by the restricted number of terms used for computation, which decide the address space of the initial approximation by look-up table. The second cause is the number of bits used for initial approximation from look-up table. However, since we have used enough bits for it (53-bit), this error is beyond the double precision requirement, and has been ignored. Third error, E_{PBM}, is caused by partial block multiplication (PBM) used at different levels of mantissa computation.

In all of the used PBMs: 53-bit PBM, 51-bit PBM, 51-bit Square PBM, and 51-bit Reduced PBM, we have ignored the top three multipliers corresponding to the LSB side in Fig. 5 and Fig. 7. We can quantify the error by the sum of these three multiplier blocks outputs. The discarded multiplier blocks in these multiplications, as shown in Fig. 5 and Fig. 7, are $A_1 \times B_1$, $A_1 \times B_2$ and $A_2 \times B_1$. Then the maximum errors in these three blocks will be given by $A_1 = 0x1FFFF$, $A_2 = 0x1FFFF$, $B_1 = 0x1FFFF$, and $B_2 = 0x1FFFF$ (the maximum values of these components), which will be equal to

$$\begin{aligned} E_{PBM} &= (A_1 \times B_1) + \{(A_1 \times B_2) + (A_2 \times B_1), 17'h00000\} \\ &= 0xFFFF400000001 \end{aligned} \tag{13}$$

Since these multiplier modules have been used at several stages of mantissa division architectures, we analyze it on case by case basis.

Table 2: Error cost of PBMs at different stages of both designs

Stages	Computation	Operands Form	Max. Error
Error Cost of PBMs in Case 1 ($m = 9$ *bits*)			
Stage-2	$x.a_1^{-1}$	$x \rightarrow 1. < 52-bit\ Significant >$ $a_1^{-1} \rightarrow 0. < 53-bit\ Significant >$ $x.a_1^{-1} \rightarrow x. < Significant\ bits >$	$E_{M1-21} = E_{PBM} * 2^{-105}$ $= 1.11\ E-16 \leq 2^{-53}$
	$a_1^{-1}.a_2$	$a_1^{-1} \rightarrow 0. < 53-bit\ Significant >$ $a_2 \rightarrow 0. < 8'h00 >< 44-bit\ Significant >$ $a_1^{-1}.a_2 \rightarrow 0. < 8'h00 >< Significant\ bits >$	$E_{M1-22} = E_{PBM} * 2^{-(102+8)}$ $= 3.467\ E-18 \leq 2^{-58}$
Stage-3	$a_1^{-2}.a_2^2$	$a_1^{-1}.a_2 \rightarrow 0. < 8'h00 >< Significant\ bits >$ $a_1^{-2}.a_2^2 \rightarrow 0. < 16'h0000 >< Significant\ bits >$	$E_{M1-3} = E_{PBM} * 2^{-(102+16)}$ $= 1.35\ E-20 \leq 2^{-63}$
Stage-6	$\beta \approx \alpha.(a_1^{-1}.a_2 - a_1^{-2}.a_2^2)$	$\alpha \rightarrow 1. < 15'h0000 >< Significant\ bits >$ $a_1^{-1}.a_2 - a_1^{-2}.a_2^2 \rightarrow 0. < 8'h00 >< Significant\ bits >$ $\beta \rightarrow 0. < 8'h00 >< Significant\ bits >$	$E_{M1-6} = E_{PBM} * 2^{-(101+8)}$ $= 6.938\ E-18 \leq 2^{-57}$
Stage-7	$xa_1^{-1}.\beta$	$x.a_1^{-1} \rightarrow x. < Significant\ bits >$ $\beta \rightarrow 0. < 8'h00 >< Significant\ bits >$ $xa_1^{-1}.\beta \rightarrow 0. < 8'h00 >< Significant\ bits >$	$E_{M1-7} = E_{PBM} * 2^{-(101+8)}$ $= 6.938\ E-18 \leq 2^{-57}$
Error Cost of PBMs in Case 2 ($m = 13$ *bits*)			
Stage-2	$x.a_1^{-1}$	Similar to Case-1	$E_{M2-21} = E_{PBM} * 2^{-105}$ $= 1.11\ E-16 \leq 2^{-53}$
	$a_1^{-1}.a_2$	$a_1^{-1} \rightarrow 0. < 53-bit\ Significant >$ $a_2 \rightarrow 0. < 12'h00 >< 40-bit\ Significant >$ $a_1^{-1}.a_2 \rightarrow 0. < 12'h00 >< Significant\ bits >$	$E_{M2-22} = E_{PBM} * 2^{-(102+12)}$ $= 2.168\ E-19 \leq 2^{-62}$
Stage-3	$a_1^{-2}.a_2^2$	$a_1^{-1}.a_2 \rightarrow 0. < 12'h00 >< Significant\ bits >$ $a_1^{-2}.a_2^2 \rightarrow 0. < 24'h0000 >< Significant\ bits >$	$E_{M2-3} = E_{PBM} * 2^{-(102+24)}$ $= 5.29\ E-23 \leq 2^{-74}$
Stage-5	$\beta \approx \alpha.(a_1^{-1}.a_2 - a_1^{-2}.a_2^2)$	$\alpha \rightarrow 1. < 24'h0000 >< Significant\ bits >$ $a_1^{-1}.a_2 - a_1^{-2}.a_2^2 \rightarrow 0. < 12'h00 >< Significant\ bits >$ $\beta \rightarrow 0. < 12'h00 >< Significant\ bits >$	$E_{M2-5} = E_{PBM} * 2^{-(101+12)}$ $= 4.33\ E-19 \leq 2^{-61}$
Stage-6	$xa_1^{-1}.\beta$	$x.a_1^{-1} \rightarrow x. < Significant\ bits >$ $\beta \rightarrow 0. < 12'h00 >< Significant\ bits >$ $xa_1^{-1}.\beta \rightarrow 0. < 12'h00 >< Significant\ bits >$	$E_{M2-6} = E_{PBM} * 2^{-(101+12)}$ $= 4.33\ E-19 \leq 2^{-61}$

4.1 Error Cost of PBMs in Case:1 ($m = 9$ *bits*)

Here, we discuss the error produced by the PBMs at their respective stages, depending on their specific input operands. The details of computation involving PBMs at different stages, their input/output operands and maximum error at the corresponding stages are made available in Table [2].

In stage-2, a 53x53-bit PBM has been used for the computation of $x.a_1^{-1}$. The maximum error, E_{M1-21} (PBM error (E_M) of case (1) in stage 2, multiplier 1) , for this will be given by $E_{PBM} * 2^{-105} = 1.11\ E-16$, which is equivalent to 2^{-53}.

Similarly, in the computation of $a_1^{-1}.a_2$ in the second stage, the maximum error, E_{M1-22}, will be given by $E_{PBM} * 2^{-(102+8)} = 3.467\ E-18$, where 2^{-102} is due to the 51x51 bit multiplication and 2^{-8} is due to 8-bit leading Zeros after decimal point in a_2.

Likewise, the other stages PBMs error are tabulated in Table [2]. Stage-4 includes a 34-bit full multiplier (effectively, a 50-bit full multiplier, because of input operands nature $0. < 16'h0000 >< Significant\ bits >$), instead of PBM, and thus have no

inherent PBM error. The addition and subtraction operations at different stages are assumed to be error free.

We can see from above that none of the stages using PBMs produce error of more than 2^{-53} ($1.11\ E-16$).

4.2 Error Cost of PBMs in Case:2 ($m = 13\ bits$)

As in previous case, details of errors caused by PBMs at different stages of computation are shown in Table [2]. Here, in this also, all the PBM's maximum errors is less than 2^{-53}.

Thus, we can see that, although PBMs, individually, at the respective stages are less error prone, the propagation of these errors throughout the stages may cause some errors.

4.3 Total Error in Case:1 ($m = 9\ bits$)

To calculate the total error, we need to estimate the propagation of errors through all the stages of the architecture. In the stage-2 computation:

$$[x.a_1^{-1}]_{exact} = x.a_1^{-1} + E_{M1-21}$$
$$[a_1^{-1}.a_2]_{exact} = a_1^{-1}.a_2 + E_{M1-22}$$

After stage-3 computation:

$$\begin{aligned}[a_1^{-2}.a_2^2]_{exact} &= [a_1^{-1}.a_2]^2_{exact} + E_{M1-3} \\ &= a_1^{-2}.a_2^2 + 2.a_1^{-1}.a_2.E_{M1-22} + E^2_{M1-22} + E_{M1-3} \\ &\approx a_1^{-2}.a_2^2 + 2.a_1^{-1}.a_2.E_{M1-22} + E_{M1-3} \\ &\text{(by ignoring second order error term)}\end{aligned}$$

On stage-4 computation:

$$\begin{aligned}[a_1^{-4}.a_2^4]_{exact} &= [a_1^{-2}.a_2^2]^2_{exact} \\ &\approx a_1^{-4}.a_2^4 + 4.a_1^{-3}.a_2^3.E_{M1-22} + 2.a_1^{-2}.a_2^2.E_{M1-3}\end{aligned}$$

$$\begin{aligned}[a_1^{-1}.a_2 - a_1^{-2}.a_2^2]_{exact} &= (a_1^{-1}.a_2 - a_1^{-2}.a_2^2) + E_{M1-22} \\ &\quad - 2.a_1^{-1}.a_2.E_{M1-22} - E_{M1-3}\end{aligned}$$

On stage-5 computation:

$$[\alpha]_{exact} \approx 1 + a_1^{-2}.a_2^2 + a_1^{-4}.a_2^4 + 2.a_1^{-1}.a_2.E_{M1-22} + E_{M1-3}$$
(on ignoring higher order error terms)

After stage-6 computation:

$$\begin{aligned}[\beta]_{exact} &= [\alpha]_{exact}.[a_1^{-1}.a_2 - a_1^{-2}.a_2^2]_{exact} + E_{M1-6} \\ &\approx (1 + a_1^{-2}.a_2^2 + a_1^{-4}.a_2^4).(a_1^{-1}.a_2 - a_1^{-2}.a_2^2) + E_\beta \\ &\approx \beta + E_\beta\end{aligned}$$

where,

$$\begin{aligned}E_\beta &\approx (a_1^{-1}.a_2 - a_1^{-2}.a_2^2).(2.a_1^{-1}.a_2.E_{M1-22} + E_{M1-3}) \\ &+ (E_{M1-22} - 2.a_1^{-1}.a_2.E_{M1-22} - E_{M1-3}). \\ &(1 + a_1^{-2}.a_2^2 + a_1^{-4}.a_2^4) + E_{M1-6} \\ &\approx E_{M1-22} - E_{M1-3} + E_{M1-6} \\ &\leq 1.00\ E-17, \text{(on ignoring very small error terms} \\ &\text{and taking } (1 + a_1^{-2}.a_2^2 + a_1^{-4}.a_2^4) \approx 1)\end{aligned}$$

On stage-7 computation:

$$[x.a_1^{-1}.\beta]_{exact} \approx x.a_1^{-1}.\beta + E_{M1-21}.\beta + x.a_1^{-1}.E_\beta \tag{14}$$

And finally, after stage-8 computation:

$$\begin{aligned}[x.a_1^{-1} - x.a_1^{-1}.\beta]_{exact} &\approx (x.a_1^{-1} - x.a_1^{-1}.\beta) \\ &+ E_{M1-21} - E_{M1-21}.\beta - x.a_1^{-1}.E_\beta\end{aligned}$$

Thus, the mantissa computation error for case 1, E_{ME1} E_{T1}, will be

$$E_{ME1} = E_{M1-21} - E_{M1-21}.\beta - x.a_1^{-1}.E_\beta \tag{15}$$

and total error can given by the sum of mantissa error (E_{ME1}) and $E_{N=7}$ (eq.(5)) as follows,

$$E_{T1} = E_{M1-21} - E_{M1-21}.\beta - x.a_1^{-1}.E_\beta + a_2^7 \tag{16}$$

In eq.(16), β has format of $0.8'h00 < Significant\ bits >$, thus the error term $E_{M1-21}.\beta$ will have the order of 2^{-61}, is well beyond the double precision requirement. However, E_{M1-21} can contribute up to 2^{-53} (≈ 0.5 ulp), and $x.a_1^{-1}.E_\beta$ can also contribute up to 1 ulp. And, $E_{N=7}$ contribution is much below the 2^{-53}. Further, normalization and rounding can contribute another 0.5 ulp. Thus, the absolute error can range from 0.5 to 2 ulp and final mantissa output.

4.4 Total Error in Case:2 ($m = 13\ bits$)

Similar to first case, the total error in mantissa computation can be approximated as follows,

$$E_{T2} = E_{M2-21} - E_{M2-21}.\beta - x.a_1^{-1}.E_\beta + a_2^5 \tag{17}$$

Here, again, the maximum error will be dominated by E_{M2-21} and $x.a_1^{-1}.E_\beta$, and the final absolute error can range from 0.5 ulp to 2 ulp after rounding and normalization.

5 Implementation Results

In this section we present the complete implementation details of the proposed architectures for double precision floating point division. We have used Virtex2-Pro, Virtex-4 and Virtex-5 FPGA platforms for our implementations. The hardware implementation details are shown in Table [3]. All the results reported are based on the post-PAR analysis of the Xilinx tool.

All the proposed design architectures are fully pipelined, with a throughput of one clock cycle. The design with $m = 9$ $bits$, on Virtex-II pro FPGA, initially has been implemented for a latency of 29 with a frequency of 210 MHz. This can easily be pipelined even more for better performance metric. So, it has also been targeted with a latency of 36, which achieves a frequency of 275 MHz. This design has also been explored for higher-end FPGAs (Virtex-4 & Virtex-5) to take the benefit of their in-built IPs (DSP48) for area (and possible performance) improvement. It is clearly seen from the Table [3] that, on Virtex-4 and Virtex-5 FPGAs, the design uses less components compared to Virtex-II pro design (with latency 29), with much better frequency of operations. The frequency that can be realized are 285 MHz (on V4) & 315 MHz (on V5), with a latency of 31 clock cycles. Like in the Virtex-II pro case, the performance on higher-end FPGAs can be easily improved with further pipelining. Design with $m = 9$ $bits$ uses 28 Multiplier Block and 1-BRAM (RAM18k). Further, if we use 24x17 feature of DSP48 on Virtex-5, as discussed in 3.3, the number of DSP48 IPs can be reduced by 4 numbers on Virtex-5 FPGA.

Similarly, the implementation result of the architecture with $m = 13$ $bits$ are shown in Table [3]. This architecture has been implemented for a latency of 26 on all FPGA platforms and displays clear benefits of higher-end FPGAs. The design performance can be further improved with more pipelining. The trade-off between implementation results of the $m = 9$ $bits$ and $m = 13$ $bits$ is the requirement of hardware resources. The hardware requirement with $m = 13$ $bits$ in terms of slices and multiplier block is less, whereas it needs more memory to store the initial approximation. Performance, however, can be easily maintained with proper pipelining. Thus, depending on the nature of the platform in terms of available resources, user can go for any one of the designs, as the accuracy achieved is the similar.

The accuracy of a floating point arithmetic operation is a general metric to be considered. In theoretical error analysis, error found to be at maximum 2 ulp. In a similar context, the proposed designs has been validated over 5-million unique random test cases. In all cases, the average error was found to be less than $1.0\,E-16$, which is less than 0.5 ulp, whereas bound on maximum error is found to be 2 ulp. According to the literature this level of accuracy is suitable for a large set of applications [20].

6 Comparisons and Discussion

In this section we present comparison of our proposed designs with the best state-of-the-art designs available in the literature. The comparison is based on the various design metric (area in terms of slices, Multiplier IP cores, and BRAM, and frequency of operation) and accuracy of the designs. Comparison is mainly based

Table 3: Implementation details of Proposed FP Division Architectures

	Virtex-2Pro		Virtex-4	Virtex-5
For $m = 9 - bit$				
Latency	29	36	31	31
Slices	1661	2097	1287	567
LUTs	1702	1974	1148	1468
FFs	2661	3502	1595	1365
MULT18x18/DSP48	28	28	28	28
BRAM (18k)	1	1	1	1
Freq (MHz)	210	275	285	315
For $m = 13 - bit$				
Latency	26		26	26
Slices	1491		950	484
LUTs	1733		1173	1134
FFs	2339		1154	1117
MULT18x18/DSP48	25		25	25
BRAM (18k)	12		12	12
Freq (MHz)	217		245	290

around the Xilinx hardware resources. Even on this platform, various division architectures are available with different speed-area-latency-precision trade-offs. By using different instances we can obtain suitable trade-offs. Also, several previous designs have not reported the number of used multipliers and BRAMs. Also, some of them have shown their implementation fully combinational or with very small latency. For them we have tried to approximate the hardware resources in terms of BRAM and DSP48/MULT8x18. We have tried to include most of the available related work for a comprehensive comparison.

Table [4] contains the comparison of our proposed design with the best available literature work. Based on the different available method in literature we categories our comparison in different subsections.

6.1 Comparison with Digit Recurrence Method

Thakkar *et. al.* [28] have used digit recurrence method for the implementation of division architecture. It has shown the fully pipelined implementation with a latency of 60 clock cycles, with a very small speed of 102 MHz, with approximately 3000 Slices. It has used Virtex-IIpro FPGA platform for their implementation. The average accuracy loss was reported to be 0.5 ulp.

A SRT based implementation is shown in Hemmert *et. al.* [13] with a large latency of 62 clock cycles and 4100 slices on Viretx-4 platform, with a promising frequency. These type of algorithms are generally resource efficient, however, it needs much larger latency, in clock cycles, for computation. Further, Xilinx floating-point (v2.0 on Virtex-IIpro and v5.0 on Virtex-4 & Virtex-5) core also has a relatively low frequency with a large latency of 55 clock cycles.

Table 4: Comparison with other available designs

Method	Latency	DSP48/ MULT18x18	BRAM (18k)	Slices / LUTs,FFs	Freq (MHz)	Avg E (ulp)	Max E (ulp)
Virtex-IIPro							
[5,21](NR-2)	-	29	28	-	-	-	5
Wang03 [34] (SRT, S.P.)	47	-	2	3245	166.6	-	-
Thakkar06 [28](DR)	60	-	-	2920	102	0.5	-
Venishetti08 [31] (DC)	32	32	-	2653	216	-	-
Daga04 [6]	32	16+	24	4041	100	-	-
Govindu05 [10](NR)	68	N.A.	N.A.	4243	140	-	-
Govindu05 [10](NR)	60	N.A.	N.A.	3625	140	-	-
Govindu05 [10](NR)	58	N.A.	N.A.	3213	140	-	-
Wang10 [33] (10,29) (SE)	15	8	62	617	125	0.5	1
Xilinx [35]	55	-	-	3721	173	-	-
Proposed (m=9-bit)	**36**	**28**	**1**	**2097**	**275**	**0.5**	**2**
Proposed (m=13-bit)	**26**	**25**	**12**	**1491**	**217**	**0.5**	**2**
Virtex-4							
Hemmert07 [13] (SRT)	62	-	-	4100	250	-	-
Venishetti08 [31] (DC)	32	32	-	3448, 3672	256	-	-
Xilinx [35]	55	-	-	3721	223	-	-
Proposed (m=9-bit)	**31**	**28**	**1**	**1287**	**285**	**0.5**	**2**
Proposed (m=13-bit)	**26**	**25**	**12**	**950**	**245**	**0.5**	**2**
Virtex-5							
Daniel10 (GS) [7]	-	29	1	1256, 527	78	14	26
Daniel10 (NR) [7]	-	40	1	1114, 468	70	10	26
Xilinx [35]	55	-	-	3220, 5997	258	-	-
Proposed (m=9-bit)	**31**	**28**	**1**	**1468, 1365**	**315**	**0.5**	**2**
Proposed (m=13-bit)	**26**	**25**	**12**	**1134, 1117**	**290**	**0.5**	**2**

6.2 Comparison with Newton-Raphson (NR) Method

One of the most popular methods used for computing division is the Newton Raphson two-iterative (NR-2) procedure [5, 21]. For double-precision it requires one look-up table in 15-bit address space, two 15×30 multiplications, two 30×60 multiplication and one 53×53 multiplication (equivalently 28 BRAM and 29 MULT18x18). The error performance of NR method with two iterations is discussed in [21], with minimum error of 1.99×2^{-55} and maximum error of 1.28×2^{-49} (4 ulp), which is more than the proposed method. Govindu *et. al.* [10] has presented a Newton-Raphson (NR) Decomposition based floating-point division implementation for various latency as mentioned in Table [3]. The utilized BRAM and multiplier blocks has not been mentioned in the paper (the basic ingredients for NR method), however, it has used a large number of slices with relatively less frequency. As, discussed in other literature, this approach has got errors in precision (up to several ulps, based on number of iterations).

Pasca [24] have proposed a recent implementation of double precision division on a Altera Stratix V FPGA platform, a higher end FPGA platform. A combination of polynomial approximation and Newton-Raphson method has been used for implementation. An interesting error analysis has been presented to achieve faithful rounding result (1-ULP), however the error cost of inherent truncated multipliers have not been included, which will increase the total error. It is proposed for two latency, 18 and 25 clock cycles. With latency of 18 clock cycles (268 MHz), it reports 887 ALUTs, 823 REGs, 2 M20K block memory, and 9 (27x27) DSP IPs. And, with latency of 25 clock cycles (400 MHz), it needs 947 ALUTs, 1296 REGs, and same amount of block memories & DSP IPs. Further, it needs 4 extra 27x27 DSPs and some extra logic to achieve faithful rounding, which additionally requires extra clock cycles and probable speed & area overhead. The memory block requirement is equivalent to 4 number of 18k BRAMs on Xilinx FPGAs. ALUTs can be configured for up to 7-input functions and are more functionally strong than Xilinx LUTs, and thus requires lower in count for any logic. From multiplier IPs point of view, inherently, the method requires one 14x15, one 23x25 multiplier, one 28-bit squarer, two 56x53 truncated block multipliers, and one 54x54 full multiplier. All of these, in Xilinx 17x17 IPs equivalent, needs at least 35 IPs (one for 14x15, 4 for 23x25, 3 for 28-bit squarer, 9 for each 56x53 truncated block multiplier, and 9+some logic for 54x54 full multiplier), with some additional clock cycles. Thus, this design, with almost similar precision (after including truncated multipliers error), with similar performance and latency (can be managed on either side easily), needs 4 BRAM (18k), and 35 17x17 multiplier IPs. Thus, based on appropriate equivalent hardware analysis, area requirement of this design is more compared to our proposed design. Further, Stratix V is based on 28nm technology, and Virtex 5 is based on 65 nm technology, so direct performance comparison will not be fair, even though we are approaching almost similar performance.

6.3 Comparison with Digit Convergence (DC) Method

A low latency (32 clock cycles) pipelined implementation, using digit convergence method, has been reported in Venishetti *et. al.* [31] on Virtex-IIpro and Virtex-4 FPGAs. The reported hardware results were not explained clearly, with no indication of amount of BRAM, an explicit component for the method. Authors have mentioned to use 6-steps for generating mantissa division result. Each step used two multipliers. It has been mentioned that, the last step has used full 54x54 bit multipliers (which needs at least 2x9 = 18 MULT18x18). Other steps have not used full multiplication. So, it is not very clear that, how the paper achieves the total of 32 MULT18x18 in all 6-steps, given that a minimum of 18 is being used in the last step only. Also, the existence of error is mentioned in the paper, but it has not been quantified. Goldberg *et. al.* [8] have proposed division of double precision floating point number using Goldschmidts algorithm, implemented on a Altera STRATIX-II FPGA platform. The area reported is large in comparison to our proposed design (about 3500 ALMs, equivalent to about 4600 slices on a Virtex II [3], however it has less performance and throughput.

Daniel *et. al.* [7] have explored the division implementation on a Virtex-5 platform using two methods, Goldsmith (GS) and Newton-Raphson (NR) methods. The latency of the designs were not mentioned, however, with a five iteration of GS and NR, it has very low frequency of operation and a high error cost. The maximum error was reported to be 1.90E-08 ($\approx$ 26 ulp), which is slightly better that single precision accuracy requirements.

6.4 Comparison with Series Expansion (SE) Method

Hung *et. al.* [16] have presented a single precision floating point division architecture. However, later on [19] have proved that the Hung's method is not feasible for double precision computation, because of its huge memory requirement. Jeong *et. al.* [19] have presented an improved version of Hung's method for double precision implementation. Algorithm is based on first computing an initial quotient using the two terms of series expansion. Then compute a correction quotient using remainder (obtained using initial quotient), and then add both quotients. Their architecture has been reported for ASIC platform. It needs three 53x28 multipliers, one 58x58 multiplier and 16Kx28 look-up table memory. In FPGAs equivalent, one 53x28 multiplier needs 6 multiplier IPs, a 58x58 multiplier needs 16 multiplier IPs, and 16Kx28 look-up table needs 32 (18k) BRAMs. Thus, in total it requires 34 multiplier IP blocks and 32 BRAMs. The hardware requirement would be more than that of proposed architecture, however, the maximum error is within 1 ulp and average error is 0.5 ulp.

In Daga *et. al.* [6], the implementation is based on reciprocation followed by a multiplier. This is similar to using only one term of series expansion. Thus, with a look-up table with 2^{13} address space (equivalent to 24 BRAM (RAM18k)) with only one term, it will have a lot of precision loss (as per Table 1. The reported result has a latency of 32 clock cycles, with 4041 slices and 100 MHz clock speed. It also needs at least 16 multiplier IP blocks. Wang *et. al.* [32] have presented a library for single precision floating-point operations. The division implementation is based on Hung's [16] method. By extending it to double-precision, it requires $2^{27} \times 56 - bit$ storage in BRAM (impractical in available FPGA platforms) look-up table and 25 MULT18x18 IPs, which is indeed a huge hardware requirement for the design. Wang *et. al.* [33], based on Hung's [16] approach, has reported a custom precision floating-point division on a Virtex-IIPro FPGA for a 41-bit (10-bit exp and 29-bit mantissa) floating point format. The area complexity is quite large, with a requirement of 62 BRAMs with 125 MHz frequency, and is further reported to have precision loss. With an estimation for double precision (based on the proposed method), it needs more than 11 multiplier IPs and BRAM for $2^{27} \times 29$ table look-up, which is indeed impractical.

In summary, the comparison results show that the proposed module is able to give the best performance, with lower required latency and area. Proposed approach is also using lower number of DSP48/MULT18x18 and BRAM blocks. Accuracy of the proposed designs lags behind some methods, however, it is equivalent to the most of the earlier reported literature.

7 Conclusions

This paper has presented efficient architectures for the double precision floating point division on FPGA platform. The proposed designs are based on the Taylor series expansion method, with selective number of terms based on the accuracy requirements of the double precision. A trade-off between the required resources and selected terms has been shown. Along with this, based on the precision limit, the size of the multipliers have been determined in its partial block format, PBM, to reduce the amount of hardware. The proposed modules achieve higher performance and area reduction, mainly in terms of number of multiplier blocks, number of block memory with less slices, when compared to other previously reported modules in the literature. The proposed designs are fully pipelined with a throughput of one clock cycle, with relatively lower latency. The performance can be improved with further pipelining, and one example of such an instance has been shown on Virtex-II pro platform. On Virtex-4 and Virtex-5 it could be even more optimized. The designs have been explored on different FPGA platforms, to explore their inherent capability. And, implementation on Virtex-4 and Virtex-5 leads to much saving on slices compared to Virtex-IIpro implementation, which can be further improved as discussed in subsection 3.3. The accuracy metric has also been theoretically estimated and also tested over a large set of the random test cases. The average error found with such testing is only 0.5 ulp with a maximum bound of 2 ulp, which is reasonable for a large set of applications.

Acknowledgements This work was partly supported by Croucher startup allowance.

References

1. IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std 754-1985 (1985). DOI 10.1109/IEEESTD.1985.82928
2. IEEE Standard for Floating-Point Arithmetic. Tech. rep. (2008). DOI 10.1109/IEEESTD.2008.4610935
3. Altera: Stratix II vs. Virtex-4 Density Comparison. White Paper (2005). URL `http://www.altera.com/literature/wp/wpstxiixlnx.pdf`
4. Anderson, S.F., Earle, J.G., Goldschmidt, R.E., Powers, D.M.: The IBM system/360 model 91: floating-point execution unit. IBM J. Res. Dev. **11**, 34–53 (1967). DOI http://dx.doi.org/10.1147/rd.111.0034
5. Antelo, E., Lang, T., Montuschi, P., Nannarelli, A.: Low latency digit-recurrence reciprocal and square-root reciprocal algorithm and architecture. In: 17th IEEE Symposium on Computer Arithmetic, pp. 147–154 (2005). DOI 10.1109/ARITH.2005.29
6. Daga, V., Govindu, G., Prasanna, V., Gangadharpalli, S., Sridhar, V.: Floating-point based block lu decomposition on fpgas. In: Proceedings of the 2004 International Conference on Engineering Reconfigurable Systems and, pp. 276–279 (2004)
7. Daniel, M.M., Diego, F.S., Carlos, H.L., Mauricio, A.R.: Tradeoff of FPGA Design of a floating-point library for arithmeitic operators. J. Integrated Circuits and Systems **5**(1), 42 –52 (2010)
8. Goldberg, R., Even, G., Seidel, P.M.: An FPGA implementation of pipelined multiplicative division with IEEE Rounding. In: 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 2007, pp. 185–196 (2007). DOI 10.1109/FCCM.2007.59
9. Goldschmidt, R.E.: Application of division by convergence. Masters thesis, Massachusetts Institute of Technology (1964)
10. Govindu, G., Scrofano, R., Prasanna, V.K.: A library of parameterizable floating-point cores for fpgas and their application to scientific computing. In: Proc. Intl Conf. Engineering Reconfigurable Systems and Algorithms, pp. 137–148. IEEE Computer Society (2005)

11. Govindu, G., Zhuo, L., Choi, S., Prasanna, V.: Analysis of high-performance floating-point arithmetic on FPGAs. In: Proceedings of 18th International Parallel and Distributed Processing Symposium, pp. 149–156. IEEE (2004). DOI 10.1109/IPDPS.2004.1303135
12. Hemmert, K.S., Underwood, K.D.: Open Source High Performance Floating-Point Modules. In: 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM-06), pp. 349–350 (2006). DOI 10.1109/FCCM.2006.54
13. Hemmert, K.S., Underwood, K.D.: Floating-point divider design for FPGAs. IEEE Trans. Very Large Scale Integr. Syst. **15**(1), 115–118 (2007). DOI 10.1109/TVLSI.2007.891099
14. Hemmert, K.S., Underwood, K.D.: Fast, efficient floating-point adders and multipliers for fpgas. ACM Trans. Reconfigurable Technol. Syst. **3**(3), 11:1–11:30 (2010). DOI 10.1145/1839480.1839481. URL http://doi.acm.org/10.1145/1839480.1839481
15. Huang, M., Wang, L., El-Ghazawi, T.: Accelerating Double Precision Floating-point Hessenberg Reduction on FPGA and Multicore Architectures. In: Symposium on Application Accelerators in High Performance Computing (SAAHPC'10) (2010)
16. Hung, P., Fahmy, H., Mencer, O., Flynn, M.: 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, pp. 1465–1468 vol.2 (1999). DOI 10.1109/ACSSC.1999.831992
17. Jaiswal, M.K., Chandrachoodan, N.: FPGA-Based High-Performance and Scalable Block LU Decomposition Architecture. IEEE Transactions on Computers **61**, 60–72 (2012). DOI http://doi.ieeecomputersociety.org/10.1109/TC.2011.24
18. Jaiswal, M.K., Cheung, R.C.C.: High Performance Reconfigurable Architecture for Double Precision Floating Point Division. In: 8th International Symposium on Applied Reconfigurable Computing (ARC-2012), pp. 302–313. Springer LNCS, Hong Kong (2012)
19. Jeong, J.C., Park, W.C., Jeong, W., Han, T.D., Lee, M.K.: A cost-effective pipelined divider with a small lookup table. Computers, IEEE Transactions on **53**(4), 489–495 (2004). DOI 10.1109/TC.2004.1268407
20. Meredith, J.S., Alvarez, G., Maier, T.A., Schulthess, T.C., Vetter, J.S.: Accuracy and performance of graphics processors: A Quantum Monte Carlo application case study. Parallel Comput. **35**(3), 151–163 (2009). DOI http://dx.doi.org/10.1016/j.parco.2008.12.004
21. Montuschi, P., Ciminiera, L., Giustina, A.: Division unit with Newton-Raphson approximation and digit-by-digit refinement of the quotient. IEE Proceedings Computers and Digital Techniques **141**(6), 317–324 (1994)
22. Obermann, S.F., Flynn, M.J.: Division algorithms and implementations. Computers, IEEE Transactions on **46**(8), 833–854 (1997). DOI 10.1109/12.609274
23. Parizi, H., Niktash, A., Kamalizad, A., Bagherzadeh, N.: A Reconfigurable Architecture for Wireless Communication Systems. Third International Conference on Information Technology: New Generations **0**, 250–255 (2006). DOI http://doi.ieeecomputersociety.org/10.1109/ITNG.2006.16
24. Pasca, B.: Correctly rounded floating-point division for dsp-enabled fpgas. In: Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on, pp. 249 –254 (2012). DOI 10.1109/FPL.2012.6339189
25. Paschalakis, S., Lee, P.: Double Precision Floating-Point Arithmetic on FPGAs. In: 2nd IEEE International Conference on Field Programmable Techonology (FPT'03), pp. 352–358 (2003)
26. Strzodka, R., Göddeke, D.: Pipelined mixed precision algorithms on FPGAs for fast and accurate PDE solvers from low precision components. In: 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM '06., pp. 259–270 (2006). DOI 10.1109/FCCM.2006.57
27. Sweeney, D.W.: An analysis of floating-point addition. IBM Syst. J. **4**, 31–42 (1965). DOI http://dx.doi.org/10.1147/sj.41.0031
28. Thakkar, A.J., Ejnioui, A.: Pipelining of double precision floating point division and square root operations. In: Proceedings of the 44th annual Southeast regional conference, ACM-SE 44, pp. 488–493. ACM, New York, NY, USA (2006). DOI http://doi.acm.org/10.1145/1185448.1185555
29. Underwood, K.: FPGAs vs. CPUs: trends in peak floating-point performance. In: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays, FPGA '04, pp. 171–180. ACM, New York, NY, USA (2004). DOI http://doi.acm.org/10.1145/968280.968305
30. Venishetti, S.K., Akoglu, A.: A Highly Parallel FPGA based IEEE-754 Compliant Double-Precision Binary Floating-Point Multiplication Algorithm . In: International Conference on Field-Programmable Techonology (ICFPT 2007), pp. 145–152 (2007). DOI 10.1109/FPT.2007.4439243
31. Venishetti, S.K., Akoglu, A.: Highly Parallel FPGA Based IEEE-754 Compliant Double-Precision Floating-Point Division. In: ERSA'08, pp. 159–165 (2008)

32. Wang, X., Braganza, S., Leeser, M.: Advanced Components in the Variable Precision Floating-Point Library. In: 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM -06), pp. 249–258 (2006). DOI 10.1109/FCCM.2006.21
33. Wang, X., Leeser, M.: Vfloat: A variable precision fixed- and floating-point library for reconfigurable hardware. ACM Trans. Reconfigurable Technol. Syst. **3**(3), 16:1–16:34 (2010). DOI 10.1145/1839480.1839486. URL http://doi.acm.org/10.1145/1839480.1839486
34. Wang, X., Nelson, B.E.: Tradeoffs of designing floating-point division and square root on Virtex FPGAs. In: 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM-2003), pp. 195–203 (2003)
35. Xilinx: Xilinx Floating-Point IP Core. URL http://www.xilinx.com