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

An Unified Architecture for Single, Double, Double-Extended, and Quadruple Precision Division

Manish Kumar Jaiswal · **Hayden K.-H. So**

Received: date / Accepted: date

Abstract A hardware architecture for quadruple precision floating point division arithmetic with multi-precision support is presented. Division is an important yet far more complex arithmetic operation than addition and multiplication, which demands significant amount of hardware resources for a complete implementation. The proposed architecture also supports the processing of single, double and double-extended precision computations with varied latency. An iterative multiplicative-based architecture for multi-precision quadruple precision division is proposed with small size and promising performance. The proposed mantissa division architecture, the most complex sub-unit, employs a series expansion methodology of division. The architecture follows the standard state-of-the-art flow for floating point division arithmetic with normal as well as sub-normal processing. The proposed division architecture is synthesized using UMC 90nm ASIC standard cell library. It is also demonstrated using a Xilinx FPGA based implementation which is integrated with a wide integer multiplier for mantissa division further optimized for FPGA implementations facilitating the built-in DSP blocks efficiently. When compared to existing quadruple precision divider available in the literature, the proposed architecture has 25% equivalent area saving, 2x improvement in latency with improved speed on FPGA platform; and it has more than 50% area saving, 3x improvement in latency-throughput with better speed on ASIC platform.

Keywords Quadruple Precision Arithmetic · Division · ASIC · FPGA · Iterative Architecture · Multi-Precision Division · Digital Arithmetic

This work is partly supported by the Research Grants Council of Hong Kong (Project GRF 17245716), and the Croucher Foundation (Croucher Innovation Award 2013).

Manish Kumar Jaiswal
Department of EEE, The University of Hong Kong, Hong Kong. E-mail: manishkj@eee.hku.hk, manishkj25@gmail.com

Hayden K.-H. So
Department of EEE, The University of Hong Kong, Hong Kong. E-mail: hso@eee.hku.hk

1 Introduction

Floating point arithmetic (FPA) is an important component of scientific and engineering applications. The single precision (with 23-bit mantissa) arithmetic computation supports approximately 7 significant decimal digits, whereas, the double precision (with 52-bit mantissa) supports around 15 decimal digits. While the precision requirements for many applications are well within the reach of single and double precision computation, there remains a number of important applications which demand even higher precision computation [8,3]. Such high precision computation can be provided by quadruple precision (Q.P.) arithmetic which can computes on equivalently 30 decimal digits.

Several software libraries are available for the quadruple precision arithmetic. However, such software approach for the quadruple precision are slower up to a factor of 10 compared to the native double precision solution [3]. To effectively accelerate this class of high-precision, high-performance applications, it is therefore imperative to have efficient native support for QP arithmetic in the hardware accelerator.

In this view, this paper proposes a multi-precision division architecture that is capable of performing up to quadruple precision operation in hardware. The proposed multi-precision division architecture is practically suitable towards ASIC realization, however, also demonstrated for a FPGA implementation for better hardware analysis. FPGA realization is further optimized by the efficient implementation of the core 114x114 integer multiplier (it is the largest unit in the architecture responsible for approximately more than 80% area, and is used for mantissa division) to facilitate the built-in DSP48E blocks on Xilinx FPGA. The FPGA based 114x114 integer multiplier design can also be aimed towards floating point multiplier implementation due to its promising area benefit.

Very limited works on quadruple precision (high precision) arithmetic architectures are available in the literature. Researchers in [4,16,10] have aimed for the FPGA-based architectures of floating point multiplication arithmetic. Diniz *et al.* [9] has presented the design metrics for the FPGA-based quadruple precision division implementation, however, no architectural and implementation details are provided. Isseven *et al.* [14] has presented an ASIC based quadruple precision division architecture using digit-recurrence method, which requires a lot of area with poor throughput. Literature lacks for the architecture for the quadruple precision division arithmetic, which is a complex arithmetic than adder and multiplier. However, several literature have demonstrated the architectures for (up to) double precision division[23, 24,15,11] but their direct extension for higher precision (beyond double precision) are often impractical due to large area/look-up-table requirements. Our previous work [18] has presented an architecture for quadruple precision division using Taylor series expansion methodology, which latter studied for multi-precision division architectural support and presented briefly in [17].

The proposed multi-precision division architecture is an extension of our prior work [17], and uses a multiplicative based division methodology, the series expansion method, for the underlying mantissa division. The multiplicative based methods provide higher performance than conventional digit-recurrence method (eg. SRT method). Other multiplicative based methods (Newton Raphson, Goldschmidt [12]) can also be built using similar strategy, which is part of our future studies. Moreover, an iterative multi-precision division architecture is designed in order to effectively utilize the hardware resources.

The main contributions of present work can be summarized as follows:

- Proposed a multi-precision quadruple precision floating point division architecture which also supports the processing of single, double and double-extended precision computation.
- An unified mantissa division expression (10) is realized as a key contribution which facilitates the design of multi-precision mantissa division architecture. It is based on the series expansion methodology of division.
- Architecture is designed in an iterative fashion to minimize the area, and demonstrated using ASIC as well FPGA based implementation.
- Architecture is also studied with various pipeline strategies to study the area-speed-latency-throughput trade-offs.

The following sections of manuscript proceeds as follows. The section.2 discusses the floating point division algorithmic flow along with the proposed methodology of multi-precision mantissa division using Taylor series expansion method. Section.3 describes in detail the proposed architecture with all internal components designs. The implementation results of proposed architecture along with functional evaluation and details on accuracy analysis are presented in section.4. Section.5 includes a detailed discussion on related work and provides comparisons with them. Finally, this manuscript is concluded in section.6.

2 Floating Point Division Algorithmic Flow

The computational flow for floating point division arithmetic is shown in Algorithm 1. It basically comprised of pre-processing (includes data extractions, exceptional case detection and pre-normalization), core-processing (main arithmetic related processing) and post-processing (post-normalization, rounding, and exceptional case handling). This algorithm includes computational support for normal as well as subnormal processing. It also comprised of exceptional case handling (infinity, NaN) and their processing. The sign and exponent processing are very trivial. However, the mantissa division is the most critical part of the architecture in terms of area and performance.

2.1 Mantissa Division Methodology

Here, the underlying methodology for mantissa division is discussed. The proposed mantissa division architecture is based on the series expansion methodology of multiplicative division approach, which proceeds as follows.

Let us consider M_1 and M_2 be the normalized dividend and divisor mantissas, respectively. Then, the mantissa quotient q can be computed as follows,

$$q = \frac{M_1}{M_2} = \frac{M_1}{a_1 + a_2} = M_1 \times (a_1 + a_2)^{-1} \tag{1}$$

Algorithm 1 Floating Point Division Algorithm [1]

```
1: $(IN1\ (Dividend), IN2\ (Divisor))$ Input Operands;
2: **Data Extraction & Exceptional Check-up:**
   {S1(Sign1), E1(Exponent1), M1(Mantissa1)} $\leftarrow IN1$
   {S2, E2, M2} $\leftarrow IN2$
   Check for Infinity, Sub-Normal, Zero, Divide-By-Zero
3: **Process both Mantissa for Sub-Normal:**
   Leading-One-Detection of Mantissas ($\rightarrow L\_Shift1, L\_Shift2$)
   Dynamic Left Shifting of Mantissas
4: **Sign, Exponent & Right-Shift-Amount Computation:**
   $S \leftarrow S1 \oplus S2$
   $E \leftarrow (E1 - L\_Shift1) - (E2 - L\_Shift2) + BIAS$
   $R\_Shift\_Amount \leftarrow (E2 - L\_Shift2) - (E1 - L\_Shift1) - BIAS$
5: **Mantissa Computation:** $M \leftarrow M1/M2$
6: **Dynamic Right Shifting of Quotient Mantissa**
7: **Normalization & Rounding:**
   Determine Correct Rounding Position
   Compute ULP using Guard, Round & Sticky Bit
   Compute $M \leftarrow M + ULP$
   1-bit Right Shift Mantissa in Case of Mantissa Overflow
   Update Exponent
8: **Finalizing Output:**
   Determine STATUS signal & Check Exceptional Cases
   Determine Final Output
```

Where, a_1 (of size $W+1$ bits) and a_2 (all remaining bits) are two segments of divisor mantissa M_2 as below.

$$M_2 \rightarrow \overbrace{1.\underbrace{xxxxxxxx}_{W}}^{a_1}\overbrace{xx\ldots\ldots\ldots xxxxxxx}^{a_2}$$

Thus, using Taylor Series expansion of $(a_1+a_2)^{-1}$ in (1) leads to,

$$q = M_1 \times (a_1^{-1} - a_1^{-2}.a_2 + a_1^{-3}.a_2^2 - a_1^{-4}.a_2^3 + \cdots) \tag{2}$$

As (2) is an infinite series, so to obtain the quotient of a desired precision it requires to restrict the number of terms in series expansion, which may may incur some error E_N in computed quotient. For restriction up to N terms, this error can be quantify as follows,

$$\begin{aligned} |E_N| &= |M_1.a_1^{-(N+1)}.a_2^N(1 - a_1^{-1}.a_2 + a_1^{-2}.a_2^2 - a_1^{-3}.a_2^3 - \ldots)| \\ &= |\frac{M_1.a_1^{-(N+1)}.a_2^N}{1 + a_1^{-1}.a_2}| \end{aligned} \tag{3}$$

This error is computed as a geometric series which consists of all the ignored terms in (2). With most pessimistic error estimation (minimum denominator $(1 + a_1^{-1}.a_2) \approx 1$, and maximum numerator $M_1 \approx 2$, $a_1^{-1} \approx 1$), the maximum error would be,

$$|E_N| = |2a_2^N| \tag{4}$$

Table 1: Required numbers of terms (N) and look-up table address space for a given W, needed for quadruple precision accuracy

W	N	Max Absolute Error, $2a_{2\ max}^{N}$	Look-up Table Address Space
6	21	$2.350\,E-38$	2^{6}
8	17	$2.294\,E-41$	2^{8}
10	13	$1.469\,E-39$	2^{10}
12	11	$3.672\,E-40$	2^{12}

Table 2: Required numbers of terms (N) and look-up table address space for a given W, needed for double precision accuracy

W	N	Max Absolute Error, $2a_{2\ max}^{N}$	Look-up Table Address Space
6	9	$1.110\,E-16$	2^{6}
8	7	$2.774\,E-17$	2^{8}
10	6	$1.734\,E-18$	2^{10}
12	5	$1.734\,E-18$	2^{12}

Thus, the number of terms (N) for a given precision (P) requirement can be obtained by following inequality:

$$|E_N| = |2a_2^N| \leq 2^{-P} \tag{5}$$

Equation (2) can be solved using a pre-computed value of a_1^{-1}. In (5), the value of a_2 is determined by the size of a_1 (ie W). Thus, for a given quotient precision requirement, W determines the number of terms from the series expansion. Also, W determines the size of look-up-table (to store a_1^{-1}), and N determines the amount of other computations (adders, multipliers, subtractors).

For a given quotient precision (P), decrease in W increases the required number of terms N and vice-verse. A variation among the value of W and required number of terms (N), for the quadruple precision quotient requirement ($P = 113$), is shown in Table- 1.

Similarly, a variation among the value of W and N, for double precision quotient requirement ($P = 53$), is shown in Table- 2. Likewise, similar variation table can be sought for single precision and double-extended precision.

Table-1 and Table-2 shows the variation of "Required numbers of terms (N)" and "look-up table address space" for a given value of W, for quadruple precision and double precision accuracy, respectively. It can be seen from these tables that with the increase in value of W, the number of required terms reduces for the mantissa division computation. The number of required terms for a mantissa division computation directly determines the amount of hardware required for underlying basic arithmetic of addition, subtraction and multiplication in mantissa division computation.

It can also be envisioned from Table-1 and Table-2 that the increase in the value of W exponentially increases the amount of look-up table memory. Therefore, from the overall architectural point of view, the value of W plays a key role, which provides an initial idea of total required hardware in terms of memory look-up table and basic arithmetic building blocks, for a given accuracy.

Thus, for a given accuracy requirement, based on the value of W, a trade-off will appear among the required memory and other hardware resources. Also, in current scenario, as an unified architecture for QP, DEP, DP, and SP division is sought, this selection is majorly dominated by the QP requirement, which automatically accomplishes the lower precision requirements. As seen from the Table-1 and Table-2, the value of $W = 8 - bit$ provides a good balance between amount of memory and number of terms in order to build an unified architecture which also helps in constructing a nicely overlapped unified mantissa division equation, as discussed ahead.

For $W = 8\ bits$, it requires 17 terms (up to $a_1^{-17}.a_2^{16}$) to meet the requirement of quadruple precision computation. Similarly, for double-extended precision with $W = 8\ bits$, it requires 9 terms (up to $a_1^{-9}.a_2^{8}$). Likewise, with $W = 8\ bits$, it requires 7 terms (up to $a_1^{-7}.a_2^{6}$) for double precision, and 3 terms (up to $a_1^{-3}.a_2^{2}$) for single precision requirement.

The final quotients are expressed below for SP, DP, DEP, and QP quotient processing. These are simplified to obtained an overlapped representation with each other in order to model the multi-precision mantissa division functioning.

$$q_{SP} = M_1 a_1^{-1} - M_1 a_1^{-1}(a_1^{-1}a_2 - a_1^{-2}a_2^2) \tag{6}$$

$$q_{DP} = M_1 a_1^{-1} - M_1 a_1^{-1}(a_1^{-1}a_2 - a_1^{-2}a_2^2)(1 + a_1^{-2}a_2^2 + a_1^{-4}a_2^4) \tag{7}$$

$$q_{DEP} = M_1 a_1^{-1} - M_1 a_1^{-1}(a_1^{-1}a_2 - a_1^{-2}a_2^2)(1 + a_1^{-2}a_2^2 + a_1^{-4}a_2^4 + a_1^{-6}a_2^6) \tag{8}$$

$$q_{QP} = M_1 a_1^{-1} - M_1 a_1^{-1}(a_1^{-1}a_2 - a_1^{-2}a_2^2)(1 + a_1^{-2}a_2^2 + a_1^{-4}a_2^4 + a_1^{-6}a_2^6)(1 + a_1^{-8}a_2^8) \tag{9}$$

The above equations (6),(7),(8) & (9) are interestingly framed in such way, so that, the (9) acts as their super-set as shown in (10).

$$q = \underbrace{\underbrace{\overbrace{\overbrace{M_1 a_1^{-1} - M_1 a_1^{-1}(a_1^{-1}a_2 - a_1^{-2}a_2^2)}^{SP}(1 + a_1^{-2}a_2^2 + a_1^{-4}a_2^4}^{DP} + a_1^{-6}a_2^6)}_{DEP}(1 + a_1^{-8}a_2^8)}_{QP} \tag{10}$$

Thus, by solving only (10) up to the desired steps, the results for single, double, double-extended or quadruple precision can be obtained. Thus, this strategy is used for multi-precision mantissa division by the implementation of (10).

Thus, using series expansion method, the mantissa division required following steps:

- Partition divisor mantissa (M_2) in two parts, a_1 and a_2.
- Pre-compute the value of a_1^{-1} and store it in a look-up table.
- Determine the number of series expansion terms for the given precision.
- Using the value of M_1, a_1^{-1}, and a_2, compute the mantissa division quotient.

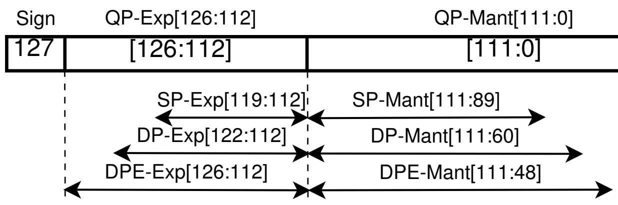

Fig. 1: Input/Output Register Format

3 Proposed Multi-Precision Division Architecture

The IEEE-754 [1] defines the standard for the floating point arithmetic. The standard format for the SP, DP, DEP and QP floating point numbers are shown in Table-3.

Table 3: Floating Point Format

	Word Size	Sign	Exponent	Mantissa
SP (32-Bit)	[31:0]	[31]	[30:23]	[22:0]
DP (64-Bit)	[63:0]	[63]	[62:52]	[52:0]
DEP (80-bit)	[79:0]	[79]	[78:64]	[63:0]
QP (128-Bit)	[127:0]	[127]	[126:112]	[111:0]

Table 4: Mode of Operation

Mode[1:0]		Processing
00	SP	Single Precision
01	DP	Double Precision
10	DEP	Double-Extended Precision
11	QP	Quadruple Precision

The proposed multi-precision division architecture works in four modes, each for SP, DP, DEP and QP processing mode, as shown in Table-4. For the purpose of multi-precision processing, the input/output operands for the unified floating point formats are assumed as shown in Fig. 1. Here, the sign-bit and decimal-point is fixed for all formats, and thus, respective exponents and mantissas are taken with-respect-to the decimal point. Remaining bits for a given format are treated as zero.

The proposed multi-precision division architecture flow is shown in Fig. 2. It primarily consists of three stages, which are discussed below in the related subsections.

3.1 Stage-1: Data-Extraction, Exceptional Handling and Pre-Normalization

The first stage of the architecture follows steps 2 and 3 of Algorithm 1. The data extraction, sub-normal and exceptional checks are shown in Fig. 3. As discussed above and shown in Fig. 1, due to common decimal point in input operands across all the

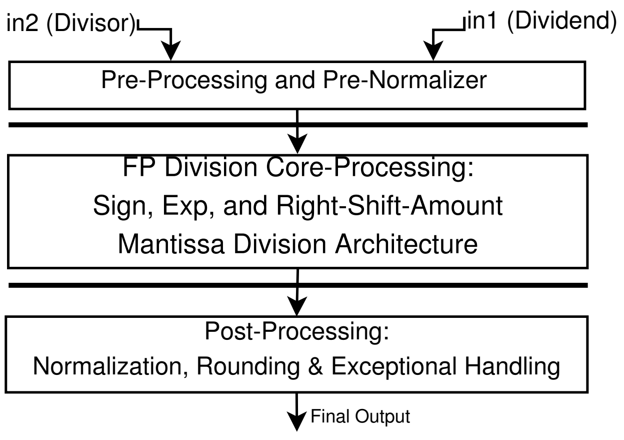
in2 (Divisor)
in1 (Dividend)
Pre-Processing and Pre-Normalizer
FP Division Core-Processing:
Sign, Exp, and Right-Shift-Amount
Mantissa Division Architecture
Post-Processing:
Normalization, Rounding & Exceptional Handling
Final Output

Fig. 2: Division Architecture Flow

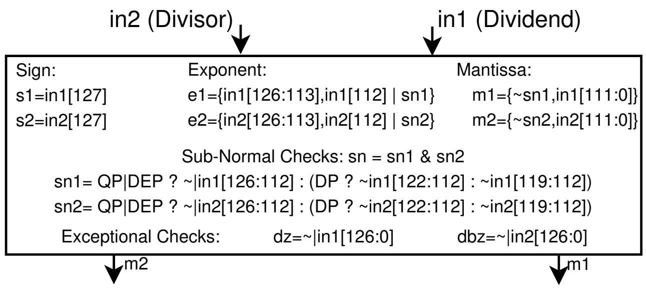
in2 (Divisor)
in1 (Dividend)
Sign:
s1=in1[127]
s2=in2[127]
Exponent:
e1={in1[126:113],in1[112] | sn1}
e2={in2[126:113],in2[112] | sn2}
Mantissa:
m1={~sn1,in1[111:0]}
m2={~sn2,in2[111:0]}
Sub-Normal Checks: sn = sn1 & sn2
sn1= QP|DEP ? ~|in1[126:112] : (DP ? ~in1[122:112] : ~in1[119:112])
sn2= QP|DEP ? ~|in2[126:112] : (DP ? ~in2[122:112] : ~in2[119:112])
Exceptional Checks:
dz=~|in1[126:0]
dbz=~|in2[126:0]
m2
m1

Fig. 3: Multi-precision data extraction, subnormal and exceptional handling

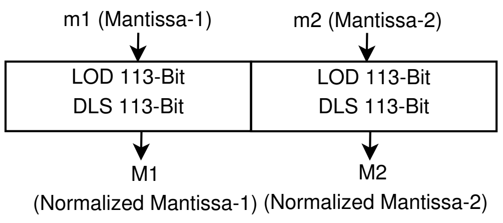

Fig. 4: Mantissas Pre-Normalization

processing mode, the data extraction for quadruple precision acts as a super-set for single precision, double precision, and double-extended precision data extraction. Further, as shown in Fig. 3, an unified subnormal check is constructed for multi-precision division. Similar to the unified sub-normal checks for multi-precision environment, the checks for infinity and NaN are also done. This unit also performs checks for `divide-by-zero` (dbz) and `zero`(dz). Since, the decimal point position is same for all modes (as shown in Fig. 1), unified/same signal for sign, exponent and mantissa works for all mode.

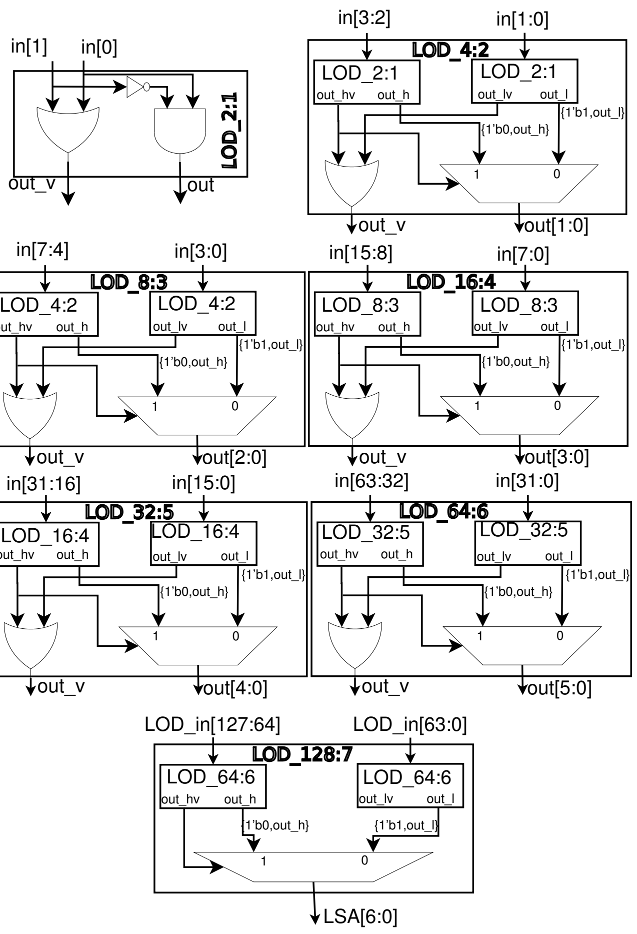

Fig. 5: Leading-One-Detector Architecture

The pre-normalizer consists of a 113-bit leading-one-detector (LOD) and 113-bit dynamic left shifter (DLS) for each mantissa (as shown in Fig. 4, which performs pre-normalization of sub-normal operands, and bring the sub-normal mantissa (if any) into the normalized format. It first computes the left-shift-amount (LSA) using a 113-bit LOD and then shifting the mantissa by 113-bit dynamic left shifter. The corresponding left-shifting-amount (LSA) is further adjusted in exponent computation. The LOD is designed in a hierarchical fashion using basic building block of 2:1 LOD which consists of an AND, an OR, and a NOT gate, and is shown in Fig 5.

The mantissas and their associated left shift amount are then fed into the dynamic left shifter. The dynamic left shifter is designed using a 7-stage barrel shifter unit, which can shift the input to a maximum of 128-bit towards left, and it requires seven 113-bit 2:1 MUXs. This architecture of DLS is not shown due to its simplicity.

Above combination of Leading-One-Detector (LOD) and Dynamic-Left-Shifter (DLS) work for all modes (SP, DP, DEP, or QP) due to the common decimal point position of mantissa operands in all these cases (Fig 1), thus facilitate the hardware saving for small precision computations. This strategy works for the most part of processing in the later stages of architecture.

After pre-normalization of mantissa, the next unit in this stage of division architecture, the 8-bit (a_1) MSB part (after decimal point) of normalized divisor mantissas (M2) is used to fetch the pre-computed initial approximation of its inverse, as discussed in the section(2). A look-up table of size $2^8 \times 113$ is used. Same look-up-table is used for all modes of operation.

In this stage, except for few variations, in sub-normal and exceptional checks, it mostly contains the computational unit needed for the QP processing, and the same units process for other modes as well. This stage also includes the part of mantissa division unit, the pre-fetching of initial approximation of divisor mantissa inverse from look-up table.

3.2 Stage-2: The Core Division Processing Architecture

This stage corresponds to the steps 4 and 5 of Algorithm 1, which processes core computations related to floating point division, as shown in Fig. 6. An unified "BIAS" signal is used for multi-precision processing, however, the processing flow of sign, exponent and right-shift-amount (RSA) proceed in a trivial way. The unified "BIAS" is represented below, where QP/DEP/DP are mode signals.

$$BIAS[14:0] = \{\{4\{QP|DEP\}\}, \{3\{QP|DEP|DP\}\}, 7'b7F\}$$

The related exponent computation is the difference of dividend (in1) exponent and divisor (in2) exponent, with proper BIASing and the adjustment of mantissa's left-shift-amount (LSA) :

$$BIAS + (Exp\_in1 - LSA\_in1) - (Exp\_in2 - LSA\_in2)$$

In case above computation leads to a negative value, then the resultant mantissa division result required to right-shift by right-shift-amount (RSA), and it will results into a subnormal output.

$$RSA = (Exp\_in2 - LSA\_in2) - BIASing - (Exp\_in1 - LSA\_in1)$$

3.2.1 Mantissa Division Unit

This is the most critical component of the floating point division processing. It builds around the implementation of (10), and is shown in Fig. 7. The size of unified look-up table to store a_1^{-1} is taken as $2^8 \times 113$, which is used for the purpose of QP, DEP, DP, as well as SP mantissa computation. To compute all the terms of (10) a full multiplier of size 114x114-bit is used iteratively with the help of a FSM design.

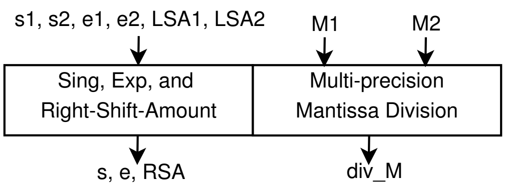
s1, s2, e1, e2, LSA1, LSA2
M1
M2
Sing, Exp, and Right-Shift-Amount
Multi-precision Mantissa Division
s, e, RSA
div_M

Fig. 6: Core-Processing Computations

The initial inverse approximation of divisor mantissa accessed from a look-up-table (in first-stage), and the remaining processing is performed using a Finite State Machine (FSM). FSM is build around a 114x114 bit integer multiplier, in an iterative fashion. Based on the mode of operation, the FSM decides the effective inputs for the multiplier in each state.

The 114x114 integer multiplier is the major component of the division architecture, which significantly affects the total area and performance and hence designed separately for ASIC and FPGA platforms, based on their specific features. Multipliers are designed with various pipeline stages, to see their effects on overall latency-area-throughput-period trade-offs.

Due to in-built features of ASIC and FPGA platform, to achieve a desired performance ASIC implementation needs smaller multiplier pipelining, whereas, FPGA based implementation would requires more pipelining. Thus, ASIC implementations are shown with two pipelined version of multiplier (1-stage, and 2-stage multipliers), and FPGA implementations are shown with 3 pipelined version of multiplier (1-stage, 3-stage, and 6-stage multipliers).

3.2.2 Mantissa Division Finite State Machine

The aim of the FSM design is the realization of (10) in an iterative manner over the 114x114 multiplier. The (10) is listed here as (11) for an easy reference, where, M_1 is the normalized dividend mantissa; and M_2 is the normalized divisor mantissa. As discussed in sec.2, M_2 is partitioned into a_1 (first 8-bit right to the decimal point) and a_2 (all remaining bits right to the a_1) as shown in (12). Also, various combinations of terms in (11) are listed in (13) for the ease of readability in later description.

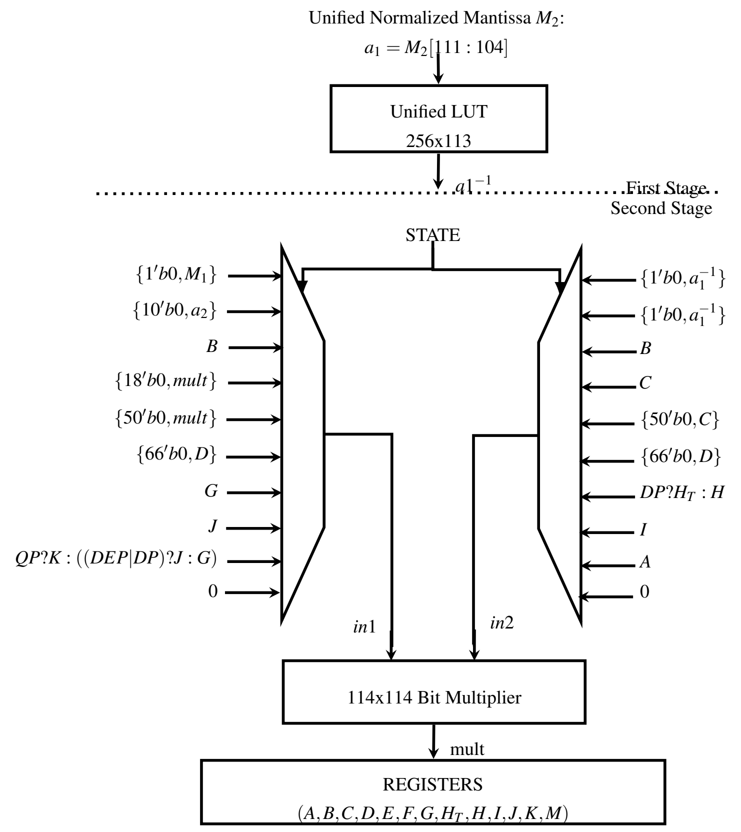

Fig. 7: Mantissa Division Architecture for Unified QP, DEP, DP and SP processing (Various terms as Registers are defined in (13))

$$q = \underbrace{\underbrace{\overbrace{\overbrace{M_1 a_1^{-1} - M_1 a_1^{-1}(a_1^{-1}a_2 - a_1^{-2}a_2^2)}^{SP}(1 + a_1^{-2}a_2^2 + a_1^{-4}a_2^4}^{DP} + a_1^{-6}a_2^6)}_{DEP}(1 + a_1^{-8}a_2^8)}_{QP} \tag{11}$$

$$M_2 \rightarrow 1.\underbrace{\overbrace{xxxxxxxx}^{a_1}}_{8-bit}\underbrace{\overbrace{xxxxxxx\ldots\ldots\ldots\ldots\ldots\ldots.xxxxxxx}^{a_2}}_{QP:104-bit,DEP:56-bit,DP:44-bit,SP:15-bit} \tag{12}$$

$$\begin{gathered}
A = M_1 a_1^{-1}, \quad B = a_1^{-1}a_2, \quad C = (B^2 =)a_1^{-2}a_2^2, \quad D = (B^4 = C^2 =)a_1^{-4}a_2^4 \\
E = (B^6 = C^3 =)a_1^{-6}a_2^6, \quad F = (B^8 = C^4 =)a_1^{-8}a_2^8, \quad G = (B - C =)a_1^{-1}a_2 - a_1^{-2}a_2^2 \\
H_T = (1 + C + D =)1 + a_1^{-2}a_2^2 + a_1^{-4}a_2^4, \quad H = (H_T + E =)1 + a_1^{-2}a_2^2 + a_1^{-4}a_2^4 + a_1^{-6}a_2^6 \\
I = (1 + F =)1 + a_1^{-8}a_2^8, \quad J = GH, \quad K = JI, \quad L = AK, \quad M = A - L
\end{gathered} \tag{13}$$

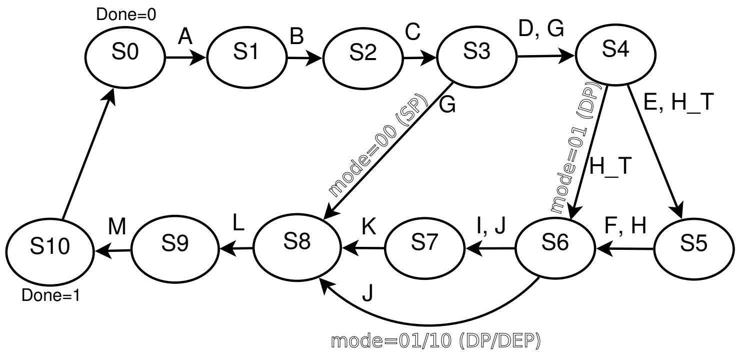

Fig. 8: Mantissa Division FSM With 1-Stage Multiplier (Various terms as Registers are defined in (13))

$$\begin{aligned}
S0:\ & in1 = \{1'b0, M_1\} && in2 = \{1'b0, a_1^{-1}\} \\
S1:\ & in1 = \{10'b0, a_2\} && in2 = \{1'b0, a_1^{-1}\} \\
& A[127:0] = mult[225:98] && \\
S2:\ & in1 = in2 = B = mult[216:103] && \\
S3:\ & in1 = in2 = \{18'b0, mult[227:132]\} && C = mult \\
& G = B - \{8'b0, C[227:122]\} && \\
S4:\ & in1 = \{50'b0, mult[227:164]\} && in2 = \{50'b0, C[227:164]\} \\
& H_T = \{1'b1, 16'b0, C[227:130\} && D = mult[191:0] \\
& \qquad + \{33'b0, D[191:110\} && \\
S5:\ & in1 = in2 = \{66'b0, D[191:144]\} && E = mult[127:0] \\
& H \leftarrow H_T + \{49'b0, E[127:62])\} && \\
S6:\ & in1 = G,\ \ in2 = DP\ ?\ H_T\ :\ H && F = mult[95:0] \\
& I = \{1'b1, 64'b0, F[95:47]\} && \\
S7:\ & in1 = J = mult[227:114] && in2 = I \\
S8:\ & in1 = QP\ ?\ (K \leftarrow mult[227:114]) && in2 = A[127:14] \\
& \quad :\ ((DEP|DP)\ ?\ J\ :\ G) && \\
S9:\ & L = (QP|DEP)\ ?\ \{6'b0, mult[227:106]\} && in1 = in2 = 0 \\
& \quad : (DP\ ?\ \{7'b0, mult[227:107]\} && \\
& \quad : \{8'b0, mult[227:108]\}) && \\
S10:\ & in1 = in2 = 0 && M = A - L
\end{aligned} \tag{14}$$

FSM with 1-stage multiplier is consists of 11 states (S0 to S10) as shown in Fig. 8. In each state of FSM, inputs (*in*1 and *in*2) for the 114x114 multiplier is determined, and its output is assigned to the designated terms, as shown in (14). The selection of

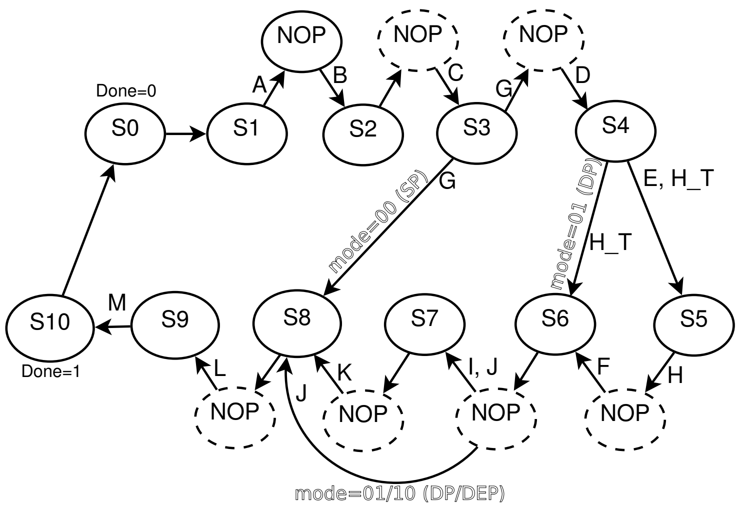

Fig. 9: Mantissa Division FSM With 2-Stage Multiplier (Various terms as Registers are defined in (13))

bits for a term is based on the position of the decimal point. Generally, the multiplications are done in full 114x114 and addition/substraction are performed in 128-bit (it helps in preserving the precision loss for QP processing).

In Fig. 8, FSM with 1-stage multiplier, for QP-mode it passes through all the states (S0 to S10). Whereas, for DEP mode it skips the state S7, which performs the multiplication of $(1+a_1^{-8}.a_2^8)$ with remaining expression. In DP-mode, it does not requires the processing of state S5 and S7; and for SP-mode, the states S4-to-S7 are not required. Some mode specific assignments can also be seen in the states S6, S8, and S9 using mode control signals (QP, DEP, DP, and SP). Thus, this FSM requires 11 cycles, 10 cycles, 9 cycles and 7 cycles respectively for QP-mode, DEP-mode, DP-mode and SP-mode processing, with 1-stage multiplier.

With single stage multiplier, due to its large size, the performance of the architecture would be slow. To improve the performance, pipelining of the multiplier is required. This requires suitable modification in the mantissa division FSM. It can be easily achieved by inserting the NOP (No-Operation state) between two states, where the output of multiplier from previous state acts as the input of multiplier in next state. For the case with 2-stage multiplier, it shown in Fig. 9, where one NOP is inserted for each of the above cases. It requires 7 extra cycles for QP-mode, 6 extra cycles for DEP-mode, 5 extra cycles for DP-mode and 3 extra cycles for SP mode. And, thus it requires 18 cycles (11+7*NOP) for QP, 16 cycles (10+6*NOP) for DEP, 14 cycles (9+5*NOP) for DP, and 10 cycles (7+3*NOP) for SP mode.

Similarly, it is shown with 3-stage multiplier in Fig. 10, where two NOP needs to be inserted at the respective places. Here, it requires 25 cycles for QP-mode, 22 cycles for DEP-mode, 19 cycles for DP-mode and 13 cycles for SP mode processing. Similarly, it is be formed with a 6-stage multiplier by inserting 5-NOP states at the similar instances.

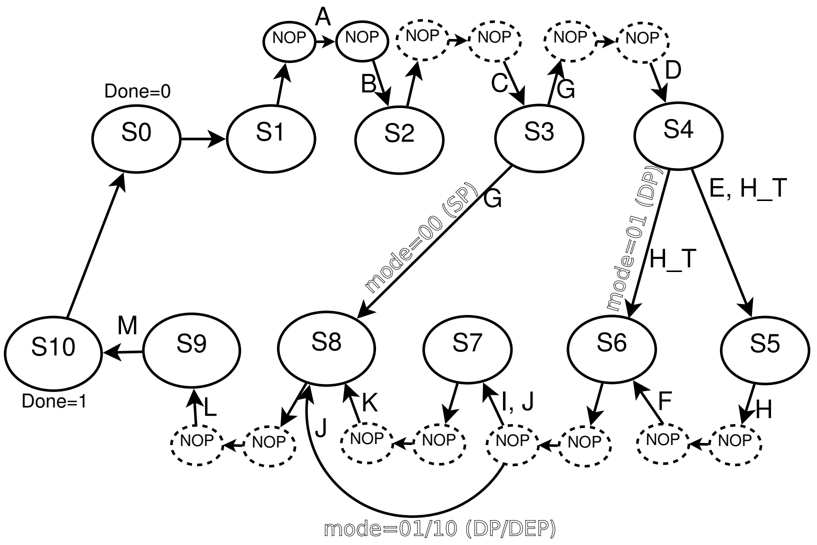

Fig. 10: Mantissa Division FSM With 3-Stage Multiplier (Various terms as Registers are defined in (13))

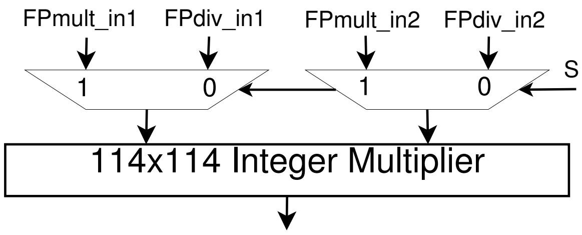

Fig. 11: Possible use of NOP states for Floating Point Multiplication ($S = !DIV \mid (DIV \,\&\, NOP)$)

The contemporary methods of FP division (using multiplicative methodologies) usage the integer multiplier in-built in FPU (Floating Point Unit), which is also used for the floating point multiplication. In above discussion of FSM with multi-stage multiplier, the NOP state can be used for scheduling a FP multiplication, as shown in Fig 11, thus fully utilizing the integer multiplier.

3.2.3 Multiplier Architecture for ASIC

The ASIC based 114x114-bit multiplier (as shown in Fig. 12) is designed using radix-4 modified booth encoding method, while the partial products are compressed using 10-levels DADDA-tree [5]. The Kogge-Stone [21] adder is used as the final carry propagate adder. Two pipelined version is explored for the ASIC implementation of proposed multi-precision division architecture: with 1-stage and 2-stage multipliers. For two-stage multiplier, a layer of pipeline registers are inserted after 8th level of DADDA-tree.

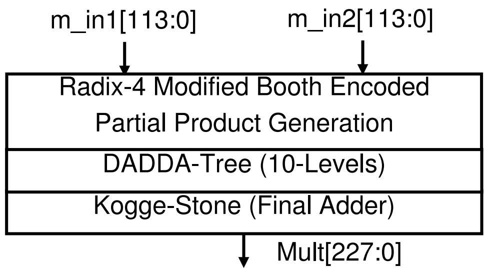
m_in1[113:0]
m_in2[113:0]
Radix-4 Modified Booth Encoded Partial Product Generation
DADDA-Tree (10-Levels)
Kogge-Stone (Final Adder)
Mult[227:0]

Fig. 12: 114x114 Integer Multiplier on ASIC

Though the proposed multi-precision division architecture is more suitable for ASIC based realization, however, to demonstrate its hardware suitability and to observe better comparison with state-of-the-art, it is also targeted for a FPGA based implementation.

3.2.4 Multiplier Architecture for FPGA

The FPGA based multiplier is designed around DSP48E IP available on Xilinx devices. A 114x144 integer multiplier architecture is discussed with 5 levels of pipeline registers (R0-R5) to build a 6-stage multiplier (with registered inputs). The 114x114 bit integer multiplier architecture is constructed with 2-partition and 3-partition Karatsuba method [20]. The 2-partition Karatsuba method helps in reducing the multiplier blocks from 4 to 3, whereas, the 3-partitioning method reduces it to 6 from 9 multiplier blocks, albeit with some extra adders/substractors requirement. As shown in Fig. 13, initially, the 114-bit multiplier operands are partitioned into three sets of 38-bit each. This needs three 38x38 multiplier blocks and three 39x39 multiplier blocks. Both of these are accomplished using a 39x39 multiplier architecture.

Further, the 39x39 multiplier, as shown in Fig. 14, uses 2-partitioning Karatsuba method. This can be accomplish by one 19x19, one 20x20 and one 21x21 multiplier blocks. The architecture for 21x21 multiplier block (as shown in Fig 15, which also made the basis for 19x19 and 20x20 multipliers, is efficiently build around the DSP48E IP of Xilinx FPGA (available on higher end FPGA series). A single DSP48E is used with some extra logic to implement these smaller multiplier blocks.

To demonstrate the various trade-offs among area-latency-speed-throughput, 3 pipelined version of multiplier is used (1-stage, 3-stage and 6-stage). With 1-stage multiplier all registers (R0-R5) are absent, and all registers are present for 6-stage. And, for 3-stage only R0 and R3 are present.

The key benefit of proposed 114x114 multiplier is the requirement of only 18 DSP48E blocks (3 DSP48E for each 39x39/38x38). On the contrary, it would requires 49 DSP48E blocks using traditional method. Our proposal out-performs some recent state-of-art on it. The Karatsuba partitioning technique for FPGA based multiplier is also explored and discussed in [7], however, for up to 54x54 multiplier and the similar strategy requires more DSP48 blocks for 114x114 multiplier.

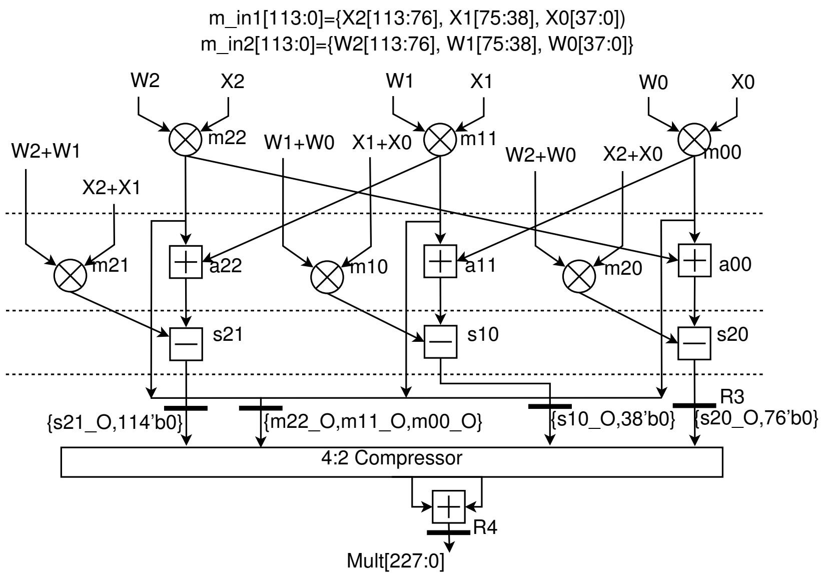

Fig. 13: 114x114 Integer Multiplier on FPGA (Using 3 Partition Karatsuba)

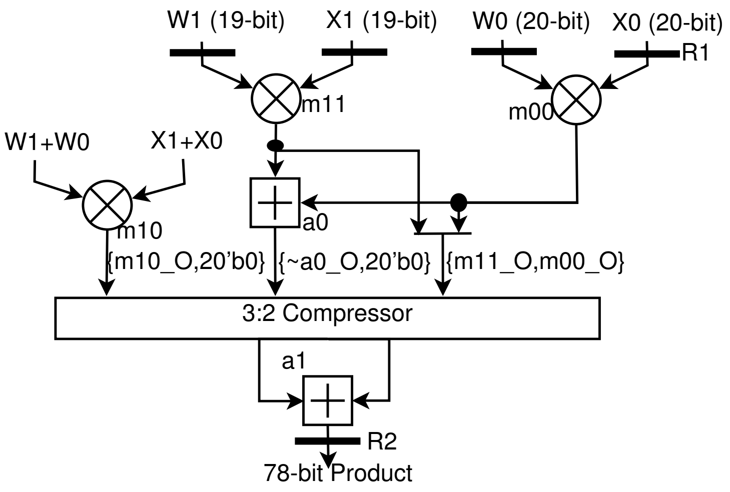

Fig. 14: 39x39 Integer Multiplier on FPGA (Using 2 Partition Karatsuba)

The 113x113 multiplier presented by Banescu *et al.* [4], on a Virtex-5 device, requires 34 DSP48E (also 2070 LUTs, 2062 FFs) with a 13 cycles latency and 407 MHz speed. Similarly, the 113x113 multiplier in [16] usage 24 DSP48E (along with 3030 LUTs and 3698 FFs) with a 14 cycles latency and 310 MHz speed. Whereas, the proposed 114x144 multiplier with 6-stage pipelining ask for only 18 DSP48E (as well as 3319 LUTs and 2185 FFs) with 172 MHz speed, on a Virtex-5 device (similar device used here as in [4, 16]). Thus, our proposal on 114x114 integer multiplier

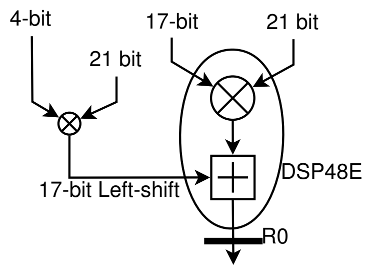

Fig. 15: 21x21 Integer Multiplier Using a DSP48E on FPGA

architecture helps in improving the utilization of inbuilt DSP48E IPs (with a balance LUTs and FFs requirement), thus facilitating more parallelization.

3.3 Stage-3: Post-Processing: Normalization, Rounding and Exceptional Handling

The third stage of the FP division architecture corresponds to the computations of steps 6,7 and 8 of the Algorithm 1. In this stage, for the case of exponent underflow (if dividend exponent is smaller than the divisor exponent), mantissa division quotient is first process for the dynamic right shifting (DRS). Which is followed by the rounding of the quotient mantissa, and then it undergoes normalization and exceptional case processing. The dynamic right shifter is designed using 7-stage barrel right shifter, which requires 7 113-bit 2:1 MUXs.

The rounding component is aimed to produce the faithful rounding results. It comprised of two steps, first the unit-at-last-place (ULP) computation (needs few gates logic) and then ULP addition with quotient mantissa (requires a 114 bit adder). ULP computation is based on the rounding position bit, Guard-bit, Round-bit and Sticky-bit. Based on the mode of the operation, the rounding position is determined by the MSB of the quotient mantissa.

The rounded quotient mantissa is further normalized (in-case of any mantissa overflow due to rounding ULP-addition). It requires 1-bit right shift for overflow. And correspondingly, due to the ULP-addition overflow, the exponents are incremented by one. Further to this, each exponents are updated for exceptional cases (either of infinity, subnormal or underflow cases).

4 Implementation Results

The proposed multi-precision quadruple precision division architecture is implemented using UMC90nm ASIC standard cell library and using Xilinx Virtex-7 FPGA device. The ASIC implementation results are shown in Table-5 and FPGA implementation results are shown in Table-6. ASIC implementation details also includes the technological independent parameters like, gate-count for area and Fan-Out-of-4 (FO4) delay metric for period.

Table 5: ASIC Implementation Details

Multiplier-Pipeline	MP-1	MP-2
Latency (cycles) (SP/DP/DEP/QP)	9/11/12/13	12/16/18/20
Throughput (cycles) (QP/DEP/DP/SP)	8/10/11/12	11/15/17/19
Area (μm)	522801	547373
Gate Count[1]	174267	182458
Period (ns)	2.23	1.22
Period (FO4)[2]	49.56	27.11

[1]Using minimum inverter size [2]1 FO4 (ns) $\approx$ (Tech. in μm) / 2

Table 6: FPGA Implementation Details

Multiplier-Pipeline	MP-1	MP-3	MP-6
Latency (cycles) (SP/DP/DEP/QP)	9/11/12/13	15/21/24/27	26/38/44/50
Throughput (cycles) (SP/DP/DEP/QP)	8/10/11/12	14/20/23/26	24/36/42/48
LUTs	7440	7997	7710
FFs	2584	2825	4383
DSP48E	18		
Freq (MHz)	89.08	160.5	226.54

MP: with Multiplier Pipeline Stage of 1, 3 & 6.

ASIC division implementations are done for two cases (with 1-stage and 2-stage multipliers), while FPGA implementations are carried out for three cases (using 1-stage, 3-stage and 6-stage multipliers). In the division architecture with 6-stage multiplier on FPGA, the first-stage and third-stage of division architecture are also pipelined by one extra level to meet the critical path of the multiplier. However, with the 1,2 and 3-stage multipliers (on ASIC as well as FPGA), they are in single stage, as the critical path is dominated by the multiplier delay.

Therefore, in QP-mode, the latency of the division architecture with 1-stage multiplier becomes 13 cycles (1 cycle first-stage, 11-cycles FSM, and 1 cycle third-stage) with throughput of 12 cycles (next input can be applied after every FSM processing, ie. after every first-stage and FSM processing). Similarly, with 1-stage multiplier, DEP mode will have latency of 12 cycles and throughput of 11 cycles, DP mode will have latency of 11 cycles and throughput of 11 cycles, and for SP mode latency is 9 cycles with 8 cycles throughput.

Likewise, with 6-stage multiplier division architecture, the latency for QP-mode is 50 cycles (2 cycles first-stage, 46 cycles FSM and 2 cycles third-stage) with throughput of 48 cycles. Similarly, the latency and throughput for other cases are mentioned in the implementation tables.

The inclusion of multi-stage multiplier in the mantissa division clearly improves the speed of the architecture, however, it decreases the throughput, with minor changes in requires area.

Here, it is very interesting to see that the computational cycle requirements for QP processing needs only few extra cycles compared to the smaller precision computations (SP, DP and DEP), which will adds to the performance benefits for QP computation. The available software solutions for QP arithmetic are slower up to a factor of 10 compared to double precision hardware solutions.

4.1 Functional Verification

The functional verification of the proposed multi-precision quadruple division architecture is carried out using 5-millions random test cases for each of the normal-normal, normal-subnormal, subnormal-normal and subnormal-subnormal operands combination, along with the other exceptional case verification, for each of the precision format. The results of proposed architecture is analyzed against the correctly rounded floating point division IP (intellectual property) from Synopsys, for each precision format. It produces a maximum of 1-ULP (unit at last place) precision loss. The method used for the mantissa division in proposed architecture is able to produce faithful rounded result.

4.2 Accuracy Analysis

In proposed mantissa computations, there are three possible source of accuracy loss. First is E_N, the error caused by the restricted number of terms used from series expansion for computation. This is discussed earlier and shown in Table-**??**, and thus, the number of terms (for each precision computation) are selected such as it falls beyond the required precision. Second error (E_{approx}) is caused by the restricted number of bits used for initial approximation (a_1^{-1}). Since 113 bits are used for it, this will introduce a 0.5 ULP (unit at last place) accuracy for quadruple requirement ($P = 113\ bits$). The third source of error is caused by the restricted width of multiplier of size 114x114. All these errors propagated through till the last stage of computation.

In related mantissa computation equations (10) and (13), computation of term $A = M_1.a_1^{-1}$ may have a 0.5 ULP loss (for QP) due to E_{approx}. Since a_2 has the form of 0.0000_0000_$xxxx \dots xxxx$, all other multiplication terms will leads to the accuracy loss well beyond the required precision ($P = 113\ bits$), as shown below in (15).

$$\begin{aligned}
a_2 &\leftarrow & 0. < 8'h0 > \_xxxx\ldots xxxx \\
B = a_1^{-1}a_2 &\leftarrow & 0. < 8'h0 > \_xxxx\ldots xxxx \\
C = B^2 &\leftarrow & 0. < 16'h0 > \_xxxx\ldots xxxx \\
D = C^2 &\leftarrow & 0. < 32'h0 > \_xxxx\ldots xxxx \\
E = C^3 &\leftarrow & 0. < 48'h0 > \_xxxx\ldots xxxx \\
f = C^4 &\leftarrow & 0. < 64'h0 > \_xxxx\ldots xxxx \\
G = B - C &\leftarrow & 0. < 8'h0 > \_xxxx\ldots xxxx \\
H_T = 1 + C + D &\leftarrow & 1. < 16'h0 > \_xxxx\ldots xxxx \\
H = 1 + C + D + E &\leftarrow & 1. < 16'h0 > \_xxxx\ldots xxxx \\
I = 1 + F &\leftarrow & 1. < 64'h0 > \_xxxx\ldots xxxx \\
J = G.H &\leftarrow & 0. < 8'h0 > \_xxxx\ldots xxxx \\
K = J.I &\leftarrow & 0. < 8'h0 > \_xxxx\ldots xxxx \\
L = A.K &\leftarrow & 0. < 8'h0 > \_xxxx\ldots xxxx \qquad (15)
\end{aligned}$$

Also, all the above multiplication is performed on main significant data only (skipping zeros of MSBs) to improve on the required accuracy. The last multiplication, $L = A.K$ may introduces another 0.5 ULP error due to error in A. Thus, the final subtraction ($M = A - L$ in (13)) will produce an error of 1 ULP. The above discussion is made in reference to QP processing, similarly, as QP processing super-sets the other lower precision processing those also leads to a 1 ULP accuracy losses. Thus, proposed architecture support the faithfully rounded results. Faithful rounding result is suitable for a large set of applications. However, correctly rounded result can be obtained by processing one more full multiplication [23,22].

5 Related Work and Comparison

To the best of author's knowledge, literature does not contains any multi-precision quadruple precision division architecture, which can support SP, DP, and DEP, either on FPGA or ASIC platform. However, there are few single mode quadruple precision implementation available. The comparison is made against them and discussed below.

Isseven *et al.* [14] has presented an ASIC based architecture for single-mode and dual-mode quadruple precision division architecture. Their design metrics for single-mode divider is presented in Table-7. Their single-mode divider requires 428783 gates with 33.68 FO4 period with a latency and throughput of 59 clock cycles. It is based on the Radix-4 SRT method of digit recurrence division methodology. This architecture requires a large area, latency and throughput.

Compared to [14], the proposed multi-precision division architecture shows more than 50% improvement in required area, 3x improvement in latency-throughput, with improved speed. Performance of proposed architecture can easily be improved further by another one or two pipelining of inherent 114x114 multiplier, while still

Table 7: Comparison of Division Architecture

ASIC Implementation Comparison		
	[14]	Proposed with MP-2
Latency	59	20
Throughput	59	19
Gate Count	428783	182458
Period (FO4)	33.68	27.11

FPGA Implementation Comparison		
	[9]	Proposed with MP-6
Latency	118	50
Throughput	118	48
LUTs	26811	7710
FFs	13809	4383
DSP48	-	18
Freq (MHz)	50	226

maintaining better latency-throughput numbers relatively (as can be seen from the latency-throughput numbers of FPGA based implementations with more pipelined multipliers). Here, the comparison is made in technological independent parameters, like gate-counts and period in FO4 delay (Fan-Out-Of-4 delay).

A single-mode quadruple precision division architecture is presented by Diniz *et al.* [9] on a FPGA device. It does not provide any architectural and implementation details, and without any idea of underlying mantissa division methodology. However, from the reported latency and throughput numbers it appears to have used Radix-2 digit-recurrence methodology of division. The implementation result shows a large resource utilization (26811 LUTs, 13809 FFs), with a 118 clock cycles latency and throughput, at 50 MHz clock speed. With a consideration of equivalent LUTs for a DSP48E (with "(a[23:0]*b[16:0])+c[47:0]" operation), it requires 644 LUTs. Thus, the proposed architecture requires approximately 19K equivalent LUTs, which provides more than 25% improvements in area, along with improved latency and speed, in comparison to [9].

Also, as discussed in sub-section(3.2.4), the proposed FPGA based 114x114 integer multiplier utilizes in-built DSP48E efficiently and out-performs the available state-of-the-art in terms of required DSP48 block as well as total equivalent area/LUTs requirements.

Due to limited literature on quadruple precision division architecture, a discussion based on the methods used in some recent double precision (DP) architecture is provided here. A SRT-based digit-recurrence double precision division architecture is proposed by Hemmert *et al.* [13]. Using SRT Radix-2 implementation it requires 4100 slices with 67 cycles latency at 250 MHz speed on Xilinx Virtex-4 device, and thus, for quadruple precision, this method would need a large latency and area. However, performance of digit-recurrence method lacks behind multiplicative methods.

Daniel et al. [6] have proposed the double precision division architecture on Virtex-5 FPGA using two methods, Goldschmidt (GS) and Newton-Raphson (NR) methods. With GS method it requires 1 BRAM, 29 DSP48, 1250 LUTs at 78 MHz speed, with maximum error of 26 ULP and average error of 14 ULP. While with NR method, it uses 1 BRAM, 40 DSP48, 1100 LUTs at 70 MHz speed, with max error of 26 ULP and average error of 10 ULP. Due to large amount of precision loss Daniel et al. [6] is not suitable for quadruple precision implementation.

A 41-bit floating point (10-bit exp and 29-bit mantissa) division architecture presented by Wang *et al.* [25] requires a large area with huge memory space (62 BRAMs). It has also reported for precision loss. This division method needs an address space of half size of operands for initial look-up table, ie 2^{27} for DP and 2^{57} for QP, which makes it impractical for the current purpose.

A combination digit-recurrence approximation and Newton-Raphson (NR) iteration for double precision division is presented by Antelo *et al.* [2]. It requires an address space of 15-bit, (ie 2^{15}, approximately 28 18k BRAMs), thus memory requirement is large. It also need an equivalent of 29 MULT18x18 IPs. Further, a combination of polynomial approximation and NR method is presented by Pasca *et al.* [23]. It is implemented on an Altera Stratix V device. In Xilinx equivalent, it requires 4 block RAM and 35 multiplier IPs, and along with 1000 ALUTS (ALUTs on Altera devices provides more functioning than Xilinx LUTs).

Fang *et al.* [11] has presented a double precision division architecture which is based on an initial approximation with Goldschmidt method. A possible extension of this for quadruple division would require a memory with huge address space of 2^{29}, and some multipliers. Jeong et al. [19] have presented a double precision division implementation. Algorithm is based on first computing an initial quotient using the two terms of series expansion, then computing a correction quotient using remainder (obtained using initial quotient), and then adding both quotients. Their architecture needs three 53x28 multipliers, one 58x58 multiplier, and $2^{14} \times 28$ look-up-table memory. In FPGAs equivalent, it requires 34 multiplier IP blocks and 32 BRAMs. The maximum error is within 1 ULP and average error is 0.5 ULP.

In summary, above methods requires larger look-up-table size even for double precision division architecture (and sometime impractical also) and thus, their algorithmic and architectural extension for quadruple precision division implementation may leads to much larger memory requirements. Further, these methods also requires a though-rough investigation for quadruple precision implementation, which is the goal of our future study on it.

6 Conclusions

This paper presented an iterative multi-precision quadruple precision division architecture for the hardware accelerators, which can also performs single precision, double precision and double-extended precision computations. The architecture is based on the series expansion methodology of multiplicative-method of mantissa division, which is implemented on ASIC platform and tuned for the FPGA implementation as well by efficient utilization of in-built DSP48E IPs. The proposed architecture provides the full computational support for the normal as well subnormal operands, and it also handles the processing of various exceptional cases. The proposed division architecture is explored with various stage multiplier unit, to see the trade-off among area, speed, latency and throughput. The proposed multi-precision division is demonstrated using UMC90nm ASIC and Xilinx Virtex-7 FPGA implementation. Compared to the available literature, the proposed architecture out-performs them in terms of area, speed, latency and throughput. The performance number (latency and throughput) for QP processing is promising in relation to small precision processing of SP, DP and DEP, which is a desired outcome when compared against the available software solution for QP division. Our future work in this area will focus on a deeper investigation of other division methodology for QP processing.

References

1. IEEE standard for floating-point arithmetic. IEEE Std 754-2008 pp. 1–70 (2008). DOI 10.1109/IEEESTD.2008.4610935
2. 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
3. Bailey, D.H., Barrio, R., Borwein, J.M.: High-precision computation: Mathematical physics and dynamics. Applied Mathematics and Computation **218**(20), 10,106–10,121 (2012). DOI 10.1016/j.amc.2012.03.087
4. Banescu, S., de Dinechin, F., Pasca, B., Tudoran, R.: Multipliers for floating-point double precision and beyond on FPGAs. SIGARCH Comput. Archit. News **38**, 73–79 (2011). DOI http://doi.acm.org/10.1145/1926367.1926380
5. Dadda, L.: Some schemes for parallel multipliers. Alta Frequenza **34**, 349–356 (1965)
6. 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)
7. de Dinechin, F.: Large multipliers with fewer DSP blocks. In: International Conference on Field Programmable Logic and Applications, pp. 250–255 (2009). DOI 10.1109/FPL.2009.5272296
8. de Dinechin, F., Villard, G.: High precision numerical accuracy in physics research. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment **559**(1), 207–210 (2006). DOI 10.1016/j.nima.2005.11.140
9. Diniz, P., Govindu, G.: Design of a field-programmable dual-precision floating-point arithmetic unit. In: Field Programmable Logic and Applications, 2006. FPL '06. International Conference on, pp. 1–4 (2006). DOI 10.1109/FPL.2006.311302
10. Dou, Y., Lei, Y., Wu, G., Guo, S., Zhou, J., Shen, L.: FPGA accelerating double/quad-double high precision floating-point applications for ExaScale computing. In: ICS '10: Proceedings of the 24th ACM International Conference on Supercomputing, pp. 325–336. ACM, New York, NY, USA (2010). DOI 10.1145/1810085.1810129
11. Fang, X., Leeser, M.: Vendor Agnostic, High Performance, Double Precision Floating Point Division for FPGAs. In: The 17th IEEE High Performance Extreme Computing (HPEC). Waltham, MA (2013)
12. Goldschmidt, R.E.: Application of division by convergence. Master's thesis, Massachusetts Institute of Technology (1964)
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. Isseven, A., Akkaş, A.: A dual-mode quadruple precision floating-point divider. In: Signals, Systems and Computers, 2006. ACSSC '06. Fortieth Asilomar Conference on, pp. 1697–1701 (2006). DOI 10.1109/ACSSC.2006.355050
15. Jaiswal, M.K., Cheung, R., Balakrishnan, M., Paul, K.: Series expansion based efficient architectures for double precision floating point division. Circuits, Systems, and Signal Processing **33**(11), 3499–3526 (2014). DOI 10.1007/s00034-014-9811-8. URL `http://dx.doi.org/10.1007/s00034-014-9811-8`
16. Jaiswal, M.K., Cheung, R.C.C.: Area-Efficient Architectures for Large Integer and Quadruple Precision Floating Point Multipliers. In: The 20th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 25–28. IEEE Computer Society, Los Alamitos, CA, USA (2012). DOI http://doi.ieeecomputersociety.org/10.1109/FCCM.2012.14
17. Jaiswal, M.K., So, H.K.H.: Architecture for Quadruple Precision Floating Point Division with Multi-Precision Support. In: 2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP), pp. 239–240 (2016). DOI 10.1109/ASAP.2016.7760807

18. Jaiswal, M.K., So, H.K.H.: Taylor series based architecture for quadruple precision floating point division. In: 2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), pp. 518–523 (2016). DOI 10.1109/ISVLSI.2016.10
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. Karatsuba, A., Ofman, Y.: Multiplication of Many-Digital Numbers by Automatic Computers. In: Proceedings of the USSR Academy of Sciences, vol. 145, pp. 293–294 (1962)
21. Kogge, P.M., Stone, H.S.: A parallel algorithm for the efficient solution of a general class of recurrence equations. Computers, IEEE Transactions on **C-22**(8), 786–793 (1973). DOI 10.1109/TC.1973.5009159
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. 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
24. 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)
25. 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`