Taylor Series Based Architecture for Quadruple Precision Floating Point Division

Manish Kumar Jaiswal[1], and Hayden K.-H So[2]
Dept. of EEE, The University of Hong Kong, Hong Kong
Email: [1]manishkj@eee.hku.hk, [2]hso@eee.hku.hk

Abstract—**This paper presents an area efficient architecture for quadruple precision division arithmetic on the FPGA platform. Many application demands for the higher precision computation (like quadruple precision) than the single and double precision. Division is an important arithmetic, but requires a huge amount of hardware resources with increasing precision, for a complete hardware implementation. So, this paper presents an iterative architecture for quadruple precision division arithmetic with small area requirement and promising speed. The implementation follows the standard processing steps for the floating point division arithmetic, including processing of sub-normal operands and exceptional case handling. The most dominating part of the architecture, the mantissa division, is based on the series expansion methodology of division, and designed in an iterative fashion to minimize the hardware requirement. This unit requires a 114x114 bit integer multiplier, and thus, a FPGA based area-efficient integer multiplier is also proposed with better design metrics than prior art on it. These proposed architectures are implemented on the Xilinx FPGA platform. The proposed quadruple precision division architecture shows a small hardware usage with promising speed.**

Keywords-**Floating Point Arithmetic, Division, FPGA, Iterative Architecture, FSM, Taylor Series Expansion Division, Digital Arithmetic.**

I. Introduction

Floating point arithmetic is an integral part of scientific and engineering applications. The single precision (SP) floating point arithmetic supports roughly 7 significant decimal digits, whereas, the double precision (DP) supports roughly for 15 decimal digits. The precision requirements for a large set of applications are well with the reach of single precision and double precision computation. However, a significant number of important applications need a higher precision computation [1], [2], that can be suffice by the quadruple precision (QP) arithmetic which provides roughly 30 decimal digits precision.

The software solution for the quadruple precision are much slower [2]. So, in the absence of direct hardware support for quadruple precision arithmetic on contemporary processing systems, another better alternative would be the use of reconfigurable platform. FPGA (Field Programmable Gate Arrays) based reconfigurable platforms are well explored during last couple of decades. A large set of applications from various domains, ranging from scientific and numerical processing, image and signal processing, cryptography, communications, bio-medical application etc are well experimented on the FPGA devices.

Literature contains a limited work on the FPGA based quadruple precision (high precision) arithmetic architectures [3], [4], [5], which aimed for floating point multiplication arithmetic. The work by Diniz *et al.* [6] has presented the design metrics for the quadruple precision division, but without any architectural and implementation details. Literature lacks for the efficient architecture for the FPGA based quadruple precision division arithmetic, which is a complex arithmetic than adder and multiplier. Although, many researchers have demonstrated the architectures for (up to) double precision division[7], [8], [9], [10] but their direct extension for higher precision (beyond double precision) are often impractical due to large area/look-up-table requirements.

In this view, this work is aimed towards an architecture for quadruple precision division arithmetic on FPGA platform, moreover, similar strategy can be easily built for any higher precision division implementation. This paper used a multiplicative based division methodology, the series expansion method [9], 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 [11]) can also be built using similar strategy. Moreover, since the complete hardware requirement for the quadruple precision arithmetic is large, and the division is not so frequent than addition/multiplication arithmetic, an iterative architecture is proposed for the quadruple precision floating division arithmetic on FPGA device.

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

- Proposed a quadruple precision floating point division architecture on a reconfigurable FPGA platform. It includes the computational support for normal as well as sub-normal operands, along with all the exceptional case handling. Architecture is fully pipelined, and designed in iterative fashion for area-efficiency.
- An efficient mantissa division architecture is designed using series expansion methodology of division. It comprised of a finite state machine (FSM) which schedules the data over a large integer multiplier iteratively to compute the mantissa division.
- For mantissa division architecture, an area efficient 114x114 bit integer multiplier is designed which

is specifically aimed for FPGA platform, and outperforms prior literature work on it.

II. Background

The quadruple precision is a 128-bit format of IEEE floating point standard [12]. The standard format for the quadruple precision floating point number is as below.

$$\overbrace{Sign-bit}^{1-bit}\ \overbrace{exponent}^{15-bit}\ \overbrace{mantissa}^{112-bit}$$

The computational flow for floating point division arithmetic is shown in Algo 1. This algorithm provides computational support for both, the normal and sub-normal processing. It also comprised of exceptional case handling (infinity, NaN) and their processing.

Algorithm 1 Floating Point Division Flow [12]

```
($IN1$ ($Dividend$),$IN2$ ($Divisor$)) Input Operands;
**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
**Process both Mantissa for Sub-Normal:**
    Leading One Detection of both Mantissa ($\rightarrow L\_Shift1, L\_Shift2$)
    Dynamic Left Shifting of both Mantissa
**Sign, Exponent & Right-Shift-Amount Computation:**
    $S \leftarrow S1 \oplus S2$
    $E \leftarrow (E1 - L\_Shift1) - (E2 - L\_Shift2) + BIAS$
    $R\_Shift\_Amount \leftarrow (E2 - L\_Shift2) - (E1 - L\_Shift1) - BIAS$
**Mantissa Computation:** $M \leftarrow M1/M2$
**Dynamic Right Shifting of Quotient Mantissa**
**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
**Finalizing Output:**
    Determine STATUS signal & Check Exceptional Cases
    Determine Final Output
```

A. Mantissa Division Methodology

The most critical part of the above algorithm is the mantissa division implementation. Here, the underlying methodology for this computation is discussed. It is based on the series expansion method of division, as follows.

Let m_1 be the normalized dividend mantissa, m_2 be the normalized divisor mantissa and q will be the mantissa quotient, which can be computed as follows,

$$q = \frac{m_1}{m_2} = \frac{m_1}{a_1 + a_2} = m_1 \times (a_1 + a_2)^{-1} \tag{1}$$

Here, the divisor mantissa m_2 is partitioned in to two part as a_1 (W-bit) and a_2 (all remaining bits) as below.

$$m_2 \rightarrow \overbrace{\underbrace{1.xxxxxxxx}_{W}}^{a_1}\overbrace{xx.........xxxxxxx}^{a_2}$$

By using Taylor Series expansion,

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

In above equation, the pre-computed value of a_1^{-1} can be used (from pre-stored look-up table) to perform the entire computation, by using integer multipliers, adders and subtractor; which are easily realizable in hardware implementation. The pre-computed value of a_1^{-1} acts as an initial approximation for m_2^{-1}, which further improved with remaining computation. Here, the size W (bit width) of a_1 (here, the hidden '1' bit place in a_1 is not counted, as it remains as constant value in the normalized format) determines the size of memory (for look-up table to store a_1^{-1}) and the number of terms from the series expansion, to perform the computation for a given precision. The number of terms (N) (with a given W) for the quadruple precision (precision requirement 2^{-113}) can be determined by following inequality:

$$\begin{aligned}|E_N| &= |a_1^{(N+1)}.a_2^N(1 - a_1^{-1}.a_2 + a_1^{-2}.a_2^2 - a_1^{-3}.a_2^3 - \dots)| \\ &= |\frac{a_1^{(N+1)}.a_2^N}{1 + a_1^{-1}.a_2}| \leq 2^{-113}\end{aligned} \tag{3}$$

where, E_N is error caused by all the ignored terms in eq.2. For maximum error, numerator of eq.3 should be maximum with the minimum value for numerator. So, for most pessimistic estimation (for minimum denominator, let $(1 + a_1^{-1}.a_2) \approx 1$, and for maximum numerator let $a_1^{-1} = 1$),

$$|E_N| = |a_2^N| \leq 2^{-113} \tag{4}$$

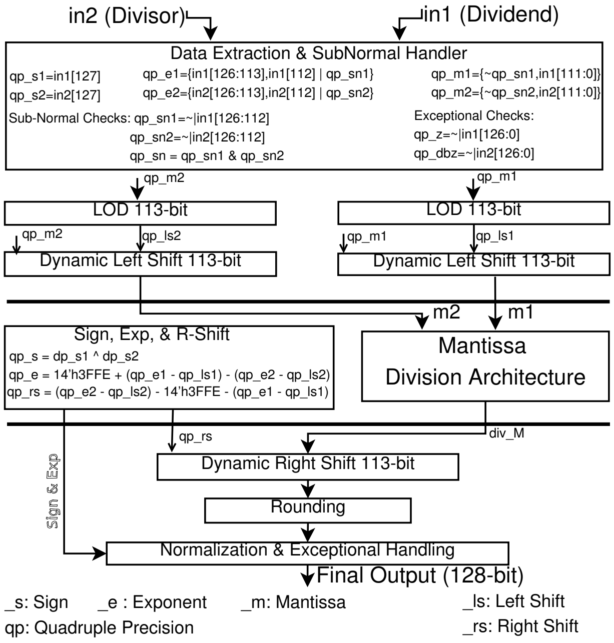
Thus, it can seen that for a given precision requirement, increase in W would reduces the required terms N and vice-versa. Here, the value of W determines the size of memory (to store the pre-computed a_1^{-1}), and N determines the amount of other hardwares (multipliers, adders, subtractors). For, a good balance, bit width of $W = 8$ for a_1 is selected, which requires seventeen terms (up to $a_1^{-17}.a_2^{16}$) to meet the requirement of quadruple precision computation. The final quotient is simplified and expressed as below, in order to optimize the hardware resource and latency of the computation.

$$\begin{aligned}q = m_1.a_1^{-1} - m_1.a_1^{-1}\{(a_1^{-1}.a_2 - a_1^{-2}.a_2^2) \\ \times (1 + a_1^{-2}.a_2^2 + a_1^{-4}.a_2^4 + a_1^{-6}.a_2^6) \times (1 + a_1^{-8}.a_2^8\}\end{aligned} \tag{5}$$

The size of look-up table to store a_1^{-1} is taken as $2^8 \times 113$, which provides sufficient precision for remaining computations. Also, a full multiplier of size 114x114-bit is used iteratively for computing all the terms, which helps in preserving the accuracy of all the terms[7], [13].

III. Proposed Quadruple Precision Division Architecture

The proposed division architecture is shown in Fig. 1. Here, it composed of three stages. The details of each stage architecture is discussed below in following subsections one-by-one.



in2 (Divisor)
in1 (Dividend)
Data Extraction & SubNormal Handler
qp_s1=in1[127]
qp_s2=in2[127]
qp_e1={in1[126:113],in1[112] | qp_sn1}
qp_e2={in2[126:113],in2[112] | qp_sn2}
qp_m1={~qp_sn1,in1[111:0]}
qp_m2={~qp_sn2,in2[111:0]}
Sub-Normal Checks: qp_sn1=~|in1[126:112]
qp_sn2=~|in2[126:112]
qp_sn = qp_sn1 & qp_sn2
Exceptional Checks:
qp_z=~|in1[126:0]
qp_dbz=~|in2[126:0]
qp_m2
qp_m1
LOD 113-bit
LOD 113-bit
qp_m2
qp_ls2
qp_m1
qp_ls1
Dynamic Left Shift 113-bit
Dynamic Left Shift 113-bit
m2
m1
Sign, Exp, & R-Shift
qp_s = dp_s1 ^ dp_s2
qp_e = 14'h3FFE + (qp_e1 - qp_ls1) - (qp_e2 - qp_ls2)
qp_rs = (qp_e2 - qp_ls2) - 14'h3FFE - (qp_e1 - qp_ls1)
Mantissa Division Architecture
qp_rs
div_M
Sign & Exp
Dynamic Right Shift 113-bit
Rounding
Normalization & Exceptional Handling
Final Output (128-bit)
_s: Sign
_e : Exponent
_m: Mantissa
_ls: Left Shift
qp: Quadruple Precision
_rs: Right Shift

Figure 1: Division Architecture

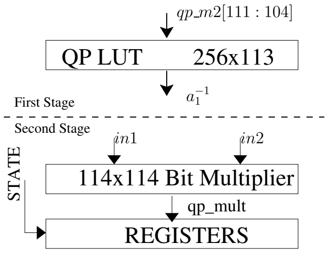


Figure 2: QP Mantissa Division Architecture

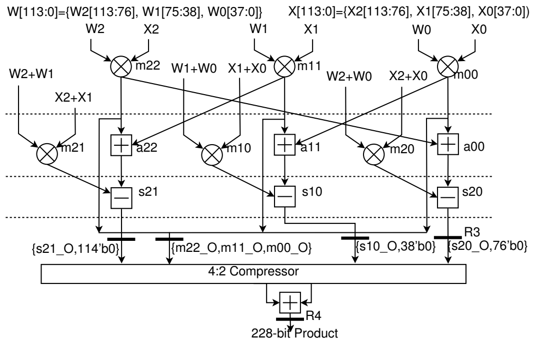


Figure 3: 114x114 Integer Multiplier Architecture

A. Stage-1: Pre-processing and Sub-normal Processing

This stage consists of steps 2 and 3 of Algo 1, and it also includes the part of mantissa division unit, the pre-fetching of initial approximation of divisor mantissa inverse from look-up table. Here, trivial processing is used for data-extraction and exceptional cases, as shown in Fig. 1. The sub-normal processing done by a 113-bit leading-one-detector (LOD) and 113-bit dynamic left shifter (DLS). They bring the sub-normal mantissa (if any) into the normalized format, first by computing the amount of left-shift using LOD and then shifting the mantissa by DLS. The corresponding left shifting amount is further adjusted in exponent part. 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. The dynamic left shifter is designed using a 7-stage barrel shifter unit, and it requires seven 113-bit 2:1 MUXs.

After pre-normalization, the 8-bit (a_1) MSB part (after decimal point) of normalized divisor mantissas (m_2) is used to fetch the pre-computed initial approximation of its inverse from a $2^8 \times 113$ look-up table. The look-up table is implemented as distributed RAM using LUTs on the FPGA device, however, can also be implemented using a block RAM (BRAM) on Xilinx FPGA.

B. Stage-2: The Core Division Processing Architecture

This stage performs all the core operation of division architecture which all corresponds to the steps 4 and 5 of Algo 1. The sign, exponent and right shift amount computation is shown in Fig. 1, and done in trivial way.

1) Mantissa Division Unit: The mantissa computation is the most complex part of the floating point division arithmetic. This architecture includes the implementation of eq.(5), and is shown in Fig. 2. It includes a look-up table to pre-store the initial inverse approximation of divisor mantissa (in first-stage), and the remaining computation is built around a 114x114 bit integer multiplier, in an iterative fashion. A finite state machine (FSM) is designed which decides the effective inputs for the multiplier in each state, the details for which are discussed after the description for the architecture of 114x114 bit multiplier.

2) Multiplier Architecture: A 114x114 bit integer multiplier is designed using combination of 3-partition and 2-partition Karatsuba method [14].First, it is partitioned in to three sets of 38-bit, which requires three multiplier blocks of 38x38 and three of size 39x39. For this, 39x39 multiplier is designed, and also used as 38x38 multiplier. The 39x39 multiplier is designed using two partition method, which need three blocks of multipliers (one 19x19, one 20x20 and one 21x21). A 21x21 multiplier is designed using a DSP48E block on Xilinx FPGA (available on Virtex-5 onward series) and some logics. These architectures are shown in Fig. 3 and Fig. 4 for 114x114 and 39x39 multiplier, respectively.

Three pipelined version of the 114x114 multiplier is used for the purpose of mantissa division; 1-stage, 3-stage and 6-stage multiplier. This is to discuss the trade-off among latency-throughput-performance-area. The multiplier architectures are presented with 5 level pipeline registers (R0-R5), which forms a 6-stage multiplier (with registered inputs). For

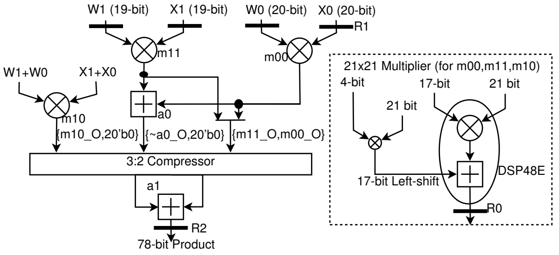


Figure 4: 39x39 Integer Multiplier Architecture

3-stage multiplier, only R0 and R3 are present, and R1, R2, and R4 are absent, whereas for 1-stage, none of pipeline registers are present.

The interesting point of the presented 114x114 multiplier is that, it requires only 18 DSP48E blocks (3 for each 39x39), whereas, using traditional method it need 49 DSP48E blocks.

3) Mantissa Division Finite State Machine: An iterative mantissa division is designed to have an area efficient architecture. The architecture is based on the hardware realization of eq.(5), which is listed below for an easy reference.

$$\begin{aligned} m_1.a_1^{-1} - m_1.a_1^{-1}\{(a_1^{-1}.a_2 - a_1^{-2}.a_2^2) \\ \times (1 + a_1^{-2}.a_2^2 + a_1^{-4}.a_2^4 + a_1^{-6}.a_2^6) \times (1 + a_1^{-8}.a_2^8\} \end{aligned}$$

Here, m_1 is the normalized dividend mantissa; and m_2 is the normalized divisor mantissa. Here, m_2 is further partitioned into a_1 (first 8-bit right to the decimal point) and a_2 (all remaining bits right to the a_1).

$$m_2 \rightarrow 1.\overbrace{\underbrace{xxxxxxxx}_{8-bit}}^{a_1}\overbrace{\underbrace{xxxxxxx\ldots\ldots\ldots xxxxxxx}_{QP:\ 104-bit}}^{a_2}$$

Here, for the ease of understanding and better readability of the later description, various combinations of terms in above equation are listed as follows:

$$\begin{array}{ll} A = m_1.a_1^{-1} & B = a_1^{-1}.a_2 \\ C = B^2 = a_1^{-2}.a_2^2 & D = B^4 = C^2 = a_1^{-4}.a_2^4 \\ E = B^6 = C^3 = a_1^{-6}.a_2^6 & F = B^8 = C^4 = a_1^{-8}.a_2^8 \\ G = B - C = a_1^{-1}.a_2 - a_1^{-2}.a_2^2 & H_T = 1 + C + D \\ H = H_T + F & \quad = 1 + a_1^{-2}.a_2^2 + a_1^{-4}.a_2^4 \\ \quad = 1 + a_1^{-2}.a_2^2 + a_1^{-4}.a_2^4 + a_1^{-6}.a_2^6 & I = 1 + F = 1 + a_1^{-8}.a_2^8 \\ J = G.H & K = J.I \\ L = A.K & M = A - L \end{array} \quad (6)$$

The implementation is achieved by designing a FSM. First, we will discuss the FSM with 1-stage multiplier, which consists of 11 states(S0 to S10), and later explore with multi-stage multiplier. 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, which proceeds as follows:

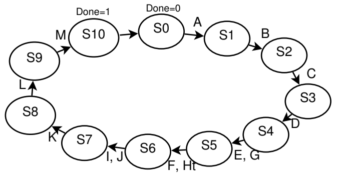


Figure 5: Mantissa Division FSM (With 1-Stage Multiplier)

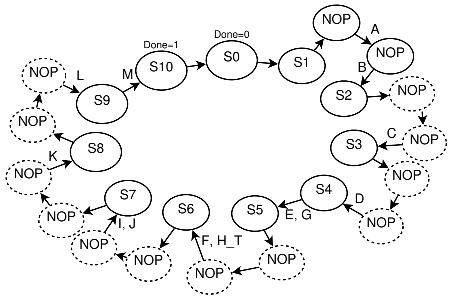


Figure 6: Mantissa Division FSM (With 3-Stage Multiplier)

$$\begin{array}{rll} S0: & in1 = \{1'b0, qp\_m_1\} & in2 = \{1'b0, qp\_m_2\_a_1\_i\} \\ S1: & in1 = \{10'b0, qp\_m_2\_a_2\} & in2 = \{1'b0, qp\_m_2\_a_1\_i\} \\ & A[113:0] = qp\_mult[225:109] & \\ S2: & in1 = in2 = B = qp\_mult[216:103] & \\ S3: & in1 = in2 = \{18'b0, qp\_mult[227:132] & C = qp\_mult \\ S4: & in1 = \{50'b0, qp\_mult[227:132]\} & in2 = \{50'b0, C[227:164] \\ & D = qp\_mult[191:0] & G = B - \{8'b0, C[227:122]\} \\ S5: & in1 = in2 = \{66'b0, D[191:144]\} & E = qp\_mult[127:0] \\ & H_T = \{1'b1, 16'b0, C[227:130\} & \\ & \quad + \{33'b0, D[191:110\} & \\ S6: & in1 = G & in2 = H = H_T + \{49'b0, E[127:62]\} \\ & F = qp\_mult[95:0] & I = \{1'b1, 64'b0, F[95:47]\} \\ S7: & in1 = J = qp\_mult[227:114] & in2 = I \\ S8: & in1 = K = qp\_mult[227:114] & in2 = A \\ S9: & in1 = in2 = 0 & L = \{6'b0, qp\_mult227, 117]\} \\ S10: & in1 = in2 = 0 & M = A - L \end{array} \quad (7)$$

The finite state machine (FSM) (with 1-stage multiplier) is shown in Fig. 5. The selection of bits for a term is based on the position of the decimal point. Generally, the multiplications are done in 114-bit (sufficient for it's precision requirement) and add/sub are performed in 128-bit (to preserve precision). The mantissa division FSM requires 11 cycles for processing, with 1-stage multiplier.

Performance can improve by pipelining the multiplier, which asks for 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 of FSM with 3-stage multiplier, it is shown in Fig. 6, where it requires to insert two NOP states for each of the above discussed case. It is presented over the '1-stage multiplier FSM' for better understanding of where to perform the insertion of the NOP states. Thus, it requires 25 states with three stage multiplier. Similarly, it is formed with a 6-stage multiplier by inserting 5-NOP states at the similar instances, and it requires a total of 47 states, (similarly, it can be designed with any multiple stage multiplier).

C. Stage-3:Normalization, Rounding and Post-processing

The third stage of the FP division architecture corresponds to the computations of steps 6,7 and 8 of the Algo 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 (needs 7, 113-bit, 2:1 MUXs). Which is followed by the rounding `round-to-nearest` of the quotient mantissa, and then it undergoes normalization and exceptional case processing, and final updates of exponent and mantissa.

IV. Implementation Results & Comparisons

The proposed quadruple precision division architecture is implemented and placed & routed on various Xilinx FPGA Devices (Virtex-5, Virtex-6 and Virtex-7) are used for the implementation. The implementation details are shown in Table-I. In the case of division architecture with 6-stage multiplier, the first-stage and third-stage of division architecture (in Fig. 1) are also pipelined by one extra level to meet the critical path of the multiplier. However, with the 3-stage multiplier, they are in single stage. Thus, the latency of the QP 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 (after every first-stage and FSM processing). Similarly, with 3-stage multiplier, the latency become 27 cycles and throughput becomes 26 cycles. Likewise, with 6-stage multiplier division architecture, the latency is 50 cycles (2 cycle first-stage, 46 cycles FSM and 2 cycles third-stage) and throughput is 48 cycles. 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 LUTs counts, and additional FFs for pipeline registers. It is also obvious to see the improvements in speed with the use of higher grade FPGA devices, from Virtex-5 to Virtex-7.

The proposed method is able to produce faithful rounded result. Faithful rounding result is suitable for a large set of applications. However, correctly rounded result can be obtained by processing one more full multiplication [7], [13], which adds few more cycles count in FSM. The functional verification of the proposed architecture is also 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, which produces a maximum of 1-ulp (unit at last place) precision loss.

Table I: Implementation Details

Multiplier-Pipeline (MP)		MP-1	MP-3	MP-6
Latency (cycle)		13	27	50
Throughput (cycle)		12	26	48
LUTs	Virtex-5	7198	7168	7333
	Virtex-6	7232	7370	7228
	Virtex-7	7220	7437	7229
FFs	Virtex-5	1734	2287	4355
	Virtex-6	2298	2677	4405
	Virtex-7	2002	2683	4397
DSP48E		18		
Freq (MHz)	Virtex-5	64.2	128.9	172.4
	Virtex-6	86.2	169.3	224.7
	Virtex-7	92.7	183	236

A. Comparisons and Related Discussion

Diniz *et al.* [6] has shown the results for a quadruple precision division implementation, but, without any architectural and mantissa division methodology details. Their design usages a large amount of hardware resources (26811 LUTs, 13809 FFs), with latency of 118 cycles at 50 MHz speed.

To the best of author's knowledge, there is no other quadruple precision division architecture on FPGA is available in the literature. So, a discussion based on the methods used in some double precision (DP) implementation is provided here. A SRT-based digit-recurrence double precision division architecture is proposed by Hemmert *et al.* [15] with 62 cycles latency and 4100 slices, and thus, for quadruple precision, this method would need a large latency and area. However, performance of digit-recurrence method lacks behind multiplicative methods.

Antelo *et al.* [16] has proposed a combination digit-recurrence approximation and Newton-Raphson (NR) iteration, and its implementation for double precision requires an address space of 15-bit (*approx* 28 18k BRAMs), and an equivalent of 29 MULT18x18 IPs. Pasca *et al.* [7] has proposed a combination of polynomial approximation and NR method on an Altera Stratix V device, which usages roughly 1000 ALUTs (ALUTs on Stratix is computationally richer that Xilinx LUTs) and an Xilinx equivalent of 4 BRAMs and 35 multiplier IPs. Wang *et al.* [17] has reported a 41-bit (10-bit exp and 29-bit mantissa) floating point format division architecture. It requires a large area with 62 BRAMs, and reported to have precision loss. This method requires a huge look-up table with address space of half size of operands, ie 2^{27} for DP and 2^{57} for QP. Another DP division implementation presented by Fang *et al.* [10] is based on an initial approximation with Goldschmidt method. An extension of this method for QP division needs a look-up table with address space of 2^{29}, and some multipliers. Thus, in view of huge look-up table requirements, above methods appears impractical for quadruple precision implementation.

The proposed architecture of 114x114 bit integer multiplier is also compared here with some recent proposals.

Banescu *et al.* [3] has designed a 113x113 multiplier using tiling method which requires 2070 LUTs, 2062 FFs, and 34 DSP48E with a latency of 13 cycles and 407 MHz speed, on Virtex-5 FPGA. Similarly, in [4] a 113x113 bit multiplier architecture requires 2373 slices and 24 DSP48E with a latency of 14 cycles and 310 MHz speed. However, the proposed 6-stage 114x114 integer multiplier architecture requires 3319 LUTs, 2185 FFs, and 18 DSP48E, with a latency of 6 cycles and 172 MHz speed, on a Virtex-5 device. On taking account of LUTs equivalent of a DSP48E (for "(a[23:0]*b[16:0])+c[47:0]" operation, in a scope of current usage), a simple synthesis results into 644 LUTs. Thus, the proposed 114x114 bit multiplier improves on overall equivalent area requirements, provides better DSP48E utilization, while speed can be improved by more pipelines.

V. Conclusions

This paper presented an iterative architecture for the quadruple precision floating point division arithmetic on reconfigurable FPGA platform. This architecture provides computational support for the normal & subnormal operands, with processing of various exceptional cases, and produces faithful rounded results. The mantissa division architecture is designed using the series expansion methodology of division operation, which is a multiplicative based division method. For this, an area efficient architecture for 114x114 integer multiplier is also proposed, which requires small number of DSP48E blocks compared to the previous literature works. The division architecture is explored with various stage multiplier unit, to see the trade-off among area, speed, latency and throughput. The proposed division architecture will leads to the exploration of applications with high precision requirement, on the reconfigurable FPGA platforms.

VI. Acknowledgments

This work is party supported by the "The University of Hong Kong" grant (Project Code. 201409176200), the "Research Grants Council" of Hong Kong (Project ECS 720012E), and the "Croucher Innovation Award" 2013.

References

[1] F. de Dinechin and G. Villard, "High precision numerical accuracy in physics research," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 559, no. 1, pp. 207–210, 2006.

[2] D. H. Bailey, R. Barrio, and J. M. Borwein, "High-precision computation: Mathematical physics and dynamics," *Applied Mathematics and Computation*, vol. 218, no. 20, pp. 10 106–10 121, 2012.

[3] S. Banescu, F. de Dinechin, B. Pasca, and R. Tudoran, "Multipliers for floating-point double precision and beyond on FPGAs," *SIGARCH Comput. Archit. News*, vol. 38, pp. 73–79, Jan 2011.

[4] M. K. Jaiswal and R. C. C. Cheung, "Area-Efficient Architectures for Large Integer and Quadruple Precision Floating Point Multipliers," in *The 20th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Los Alamitos, CA, USA: IEEE Computer Society, 2012, pp. 25–28.

[5] Y. Dou, Y. Lei, G. Wu, S. Guo, J. Zhou, and L. Shen, "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*. New York, NY, USA: ACM, 2010, pp. 325–336.

[6] P. Diniz and G. Govindu, "Design of a field-programmable dual-precision floating-point arithmetic unit," in *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, Aug 2006, pp. 1–4.

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

[8] X. Wang and M. Leeser, "Vfloat: A variable precision fixed- and floating-point library for reconfigurable hardware," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, no. 3, pp. 16:1–16:34, Sep. 2010.

[9] M. Jaiswal, R. Cheung, M. Balakrishnan, and K. Paul, "Series expansion based efficient architectures for double precision floating point division," *Circuits, Systems, and Signal Processing*, vol. 33, no. 11, pp. 3499–3526, 2014.

[10] X. Fang and M. Leeser, "Vendor Agnostic, High Performance, Double Precision Floating Point Division for FPGAs," in *The 17th IEEE High Performance Extreme Computing (HPEC)*, Waltham, MA, 2013.

[11] R. E. Goldschmidt, "Application of division by convergence," *Master's thesis, Massachusetts Institute of Technology*, June 1964.

[12] "IEEE standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, Aug 2008.

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

[14] A. Karatsuba and Y. Ofman, "Multiplication of Many-Digital Numbers by Automatic Computers," in *Proceedings of the USSR Academy of Sciences*, vol. 145, 1962, pp. 293–294.

[15] K. S. Hemmert and K. D. Underwood, "Floating-point divider design for FPGAs," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 15, no. 1, pp. 115–118, 2007.

[16] E. Antelo, T. Lang, P. Montuschi, and A. Nannarelli, "Low latency digit-recurrence reciprocal and square-root reciprocal algorithm and architecture," in *17th IEEE Symposium on Computer Arithmetic*, Jun. 2005, pp. 147–154.

[17] X. Wang and M. Leeser, "Vfloat: A variable precision fixed- and floating-point library for reconfigurable hardware," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, no. 3, pp. 16:1–16:34, Sep. 2010.