Implementation of Floating Point Division Using Brent-kung Adder for Area Optimization

Ms. Shaik Sheeba Kowsar, M.Tech (DSCE), JNTUACEA, Ananthapuramu
Mr. K.Srinivasa Rao, Lecturer, JNTUACEA, Ananthapuramu

Abstract:

Floating point division is a core arithmetic widely used in scientific and engineering applications. This paper proposes architecture for double precision floating point division. This architecture is designed for dual-mode functionality, which can either compute on a pair of double precision operands or on two pairs of single precision operands in parallel. The architecture is based on the series expansion multiplicative methodology of mantissa computation. For this, a novel dual mode Radix-4 Modified Booth multiplier is designed, which is used for division in the architecture of dual-mode mantissa computation. Other key components of floating point division flow (such as leading-one-detection, left/right dynamic shifters, rounding, etc.) are also re-designed for the dual-mode operation. The proposed dual-mode architectures Multiplication stage is done with approximate adder circuits which can give better results in terms of Area Compared to standalone Dual mode double precision division architecture. In comparison to prior art on this, the proposed architecture out-performs them in terms of required area, time period. The proposed dual-mode architecture is synthesized Virtex7 FPGA family In Xilinx ISE.

Keywords: Floating Point Division, Single precision, Double Precision, Booth Multiplier, Brant kung adder, Approximate adder.

I. INTRODUCTION

Floating point arithmetic (FPA) architectures underwent significant advancement by scientific research in the past several decades. FPA is a basic ingredient of a large set of scientific and engineering domain applications. To boost the application performances, the FPA architectures developed from scalar to vector architectures in various processing platforms. Arrays of single precision and double precision computing units are being used for floating point vector processing. The current research work is aimed towards the idea of unified vector-processing units. That is, instead of having separate vector arrays of single precision and double precision, it can have an array of configurable floating point arithmetic blocks where each of these configurable blocks can process either a double precision or two parallel single precision computations. This configurable block array arrangement can lead to significant area improvement, while providing the required performance. Our research work is focused on the architecture design of configurable floating point arithmetic blocks. This paper is focused on the design of configurable dual-mode double precision division arithmetic unit. Floating point (FP) division is a core computation required in a multitude of applications.

FP division is a complex arithmetic operation which requires larger area with poor performance than the basic arithmetic operations (adder, subtractor and multiplier). In view of large area requirement of division arithmetic per unit of computation, this work is aimed for multi-precision dual-mode architecture for this computation. The proposed architecture can be configured either for a double precision or two parallel (dual) single precision division computations, and thus named as DPdSP division architecture. The proposed architecture is based on the series expansion methodology of division algorithm [1]–[5].

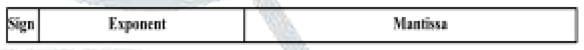
A novel dual-mode Radix-4 Modified Booth multiplier is designed for the purpose of mantissa division, which has minimal area and performance overhead over single-mode multiplier. It is based on the Radix-4 Modified Booth Algorithm. The proposed DPdSP division architectures are designed for normal as well as sub-normal computational support, which also include the exceptional case handling and processing. It is able to produce faithful rounded results with round-to-nearest rounding, both in double and single precision. All the major building blocks (like mantissa division, leading-one-detection, dynamic right/left shifting, rounding) are designed and optimized for the efficient dual-mode processing.

Rest of the paper is organized as follows; In Section II the floating point arithmetic concepts are discussed. In section III the DPDSP architecture is presented. In section IV We Presented the approximate adder based. And in section IV We have Results and Discussion. Finally, This paper concludes in section V.

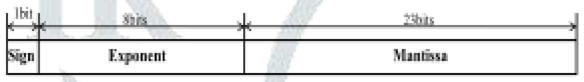
II. FLOATING POINT ARITHMETIC

The term floating point implicates that there is no fixed number of digits before and after the decimal point; i.e. the decimal point can float. Floating-point representations are slower and less accurate than fixed-point representations, but can handle a larger range of numbers. Because mathematics with floating-point numbers requires a great deal of computing power, many microprocessors come with a chip, called a floating point unit (FPU), specialized for performing floating-point arithmetic.

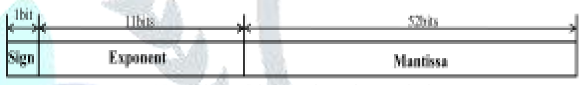
Floating point encodings and functionality are defined in the IEEE 754 Standard last revised in 2008. Floating-point representation has a complex encoding scheme with three basic components: mantissa, exponent and sign. Usage of binary numeration and powers of 2 resulted in floating point numbers being represented as single precision (32-bit) and double precision (64-bit) floating-point numbers.



(a) IEEE Floating point Standard



(b) Single Precision Floating point Standard



(c)Double Precision Floating point Standard

This range of real numbers, when fixed point representation is used, is not sufficient in many practical problems. Therefore, another representation called normalized floating point representation is used for real numbers in computers. In this representation, 32 bits are divided into two parts: a part called the mantissa with its sign and the other called the exponent with its sign. The mantissa represents fractions with a non-zero leading bit and the exponent the power of 2 by which the mantissa is multiplied. This method increases the range of numbers that may be represented using 32 bits. In this method, a binary floating point number is represented by

$$(sign) \times mantissa \times 2^{\pm\, exponent}$$

where the sign is one bit, the mantissa is a binary fraction with a non-zero leading bit, and the exponent is a binary integer.

And also, encodings to represent infinity and not-a-number (NaN) data are reserved. The IEEE 754 Standard describes floating point encodings in full. Given that the fraction field uses a limited number of bits, not all real numbers can be represented exactly. For example the mathematical value of the fraction 2/3 represented in binary is 0.10101010... This has an infinite number of bits after the binary point. The value 2/3 must be rounded first in order to be represented as a floating point number with limited precision.

FLOATING POINT DIVISION

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:

$$Q = {}^{X}/_{Y} \qquad (1)$$

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

$$q = {}^{x}/_{y} = x \times {}^{1}/_{y}$$

$$= x \times \frac{1}{a1 + a2} = x \times (a1 + a2)^{-1} \qquad (2)$$

For this purpose, we have partitioned the denominator mantissa into two parts, *m*-bit $a1$ and remaining as $a2$. $a1$ is used as an address input to a look-up table (memory) fetch some pre-computed value of ${a_1}^{-1}$.

III. DPDSP Division Architecture

The Existing architecture is shown in Fig.2. It is composed of three pipelined stages. The details of each stage of architecture are discussed below in following subsections one-by-one. Two 64-bit operands, one dividend (*in*1) and another divisor (*in*2) are the primary inputs along with the mode-control signal *dp_sp* (double precision or dual single precision).

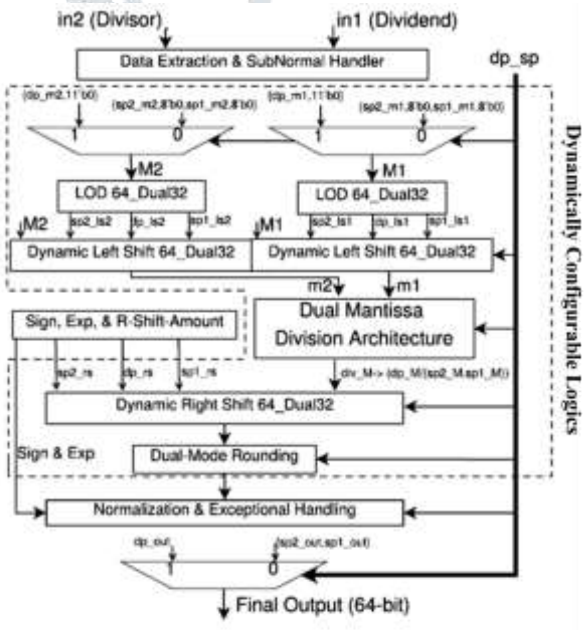

in2 (Divisor)
in1 (Dividend)
Data Extraction & SubNormal Handler
dp_sp
(dp_m2,11'b0)
(sp2_m2,8'b0,sp1_m2,8'b0)
(dp_m1,11'b0)
(sp2_m1,8'b0,sp1_m1,8'b0)
1
0
1
0
M2
M1
LOD 64_Dual32
LOD 64_Dual32
M2
sp2_ls2
dp_ls2
sp1_ls2
M1
sp2_ls1
dp_ls1
sp1_ls1
Dynamic Left Shift 64_Dual32
Dynamic Left Shift 64_Dual32
m2
m1
Sign, Exp, & R-Shift-Amount
Dual Mantissa Division Architecture
sp2_rs
dp_rs
sp1_rs
div_M-> (dp_M|(sp2_M,sp1_M))
Dynamic Right Shift 64_Dual32
Sign & Exp
Dual-Mode Rounding
Normalization & Exceptional Handling
dp_out
(sp2_out,sp1_out)
1
0
Final Output (64-bit)
Dynamically Configurable Logics

Fig.2 DPdSP Division Architecture

Both of the input operands either contains DP operands (as entire 64-bit pair) or two parallel SP operands (as two sets of 32-bit pair), as shown in Figure 3

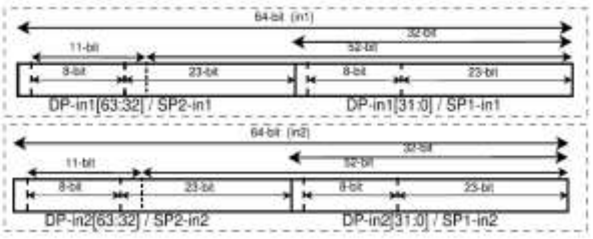
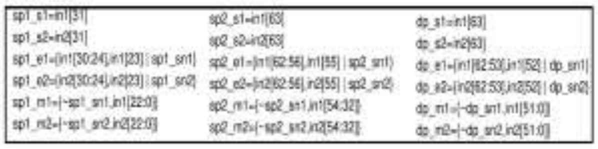


Fig.3 DPdSP Input Output Format

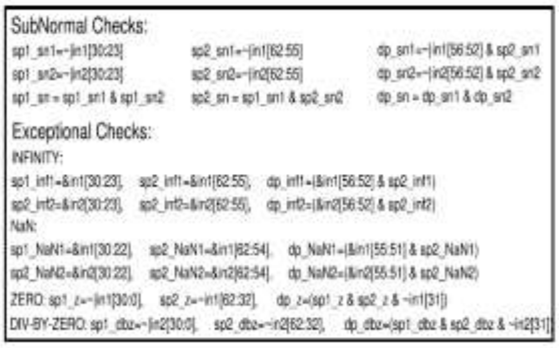
First stage comprised of the basic processing for data-extraction, exceptional case handling, and sub normal processing. It also includes the part of mantissa division unit, the pre-fetching of initial approximation of divisor mantissa inverse from look-up table.

The data extraction computation is shown in Fig. 4. These take the primary operands and extract the signs, exponents, mantissas for double precision and single precision is given below



```
sp1_s1=in1[31]                        sp2_s1=in1[63]                        dp_s1=in1[63]
sp1_s2=in2[31]                        sp2_s2=in2[63]                        dp_s2=in2[63]
sp1_e1={in1[30:24],in1[23] | sp1_sn1}  sp2_e1={in1[62:56],in1[55] | sp2_sn1}  dp_e1={in1[62:53],in1[52] | dp_sn1}
sp1_e2={in2[30:24],in2[23] | sp1_sn2}  sp2_e2={in2[62:56],in2[55] | sp2_sn2}  dp_e2={in2[62:53],in2[52] | dp_sn2}
sp1_m1={~sp1_sn1,in1[22:0]}           sp2_m1={~sp2_sn1,in1[54:32]}          dp_m1={~dp_sn1,in1[51:0]}
sp1_m2={~sp1_sn2,in2[22:0]}           sp2_m2={~sp2_sn2,in2[54:32]}          dp_m2={~dp_sn2,in2[51:0]}
```

Fig.4 DPdSP Division: Data Extraction



```
SubNormal Checks:
sp1_sn1=~|in1[30:23]          sp2_sn1=~|in1[62:55]          dp_sn1=~|in1[56:52] & sp2_sn1
sp1_sn2=~|in2[30:23]          sp2_sn2=~|in2[62:55]          dp_sn2=~|in2[56:52] & sp2_sn2
sp1_sn = sp1_sn1 & sp1_sn2    sp2_sn = sp1_sn1 & sp2_sn2    dp_sn = dp_sn1 & dp_sn2
Exceptional Checks:
INFINITY:
sp1_inf1=&in1[30:23],  sp2_inf1=&in1[62:55],  dp_inf1=(&in1[56:52] & sp2_inf1)
sp2_inf2=&in2[30:23],  sp2_inf2=&in2[62:55],  dp_inf2=(&in2[56:52] & sp2_inf2)
NaN:
sp1_NaN1=&in1[30:22],  sp2_NaN1=&in1[62:54],  dp_NaN1=(&in1[55:51] & sp2_NaN1)
sp2_NaN2=&in2[30:22],  sp2_NaN2=&in2[62:54],  dp_NaN2=(&in2[55:51] & sp2_NaN2)
ZERO: sp1_z=~|in1[30:0],  sp2_z=~in1[62:32],  dp_z=(sp1_z & sp2_z & ~in1[31])
DIV-BY-ZERO: sp1_dbz=~|in2[30:0],  sp2_dbz=~in2[62:32],  dp_dbz=(sp1_dbz & sp2_dbz & ~in2[31])
```

Fig.5 DPdSP Division: Sub-Normal and Exceptional Handling

The next two units, the leading-one-detector (LOD) and dynamic left shifter, in this stage perform sub-normal processing. They bring the sub-normal mantissa (if any) into the normalized format. First it computes the amount of left-shift using dual-mode LOD and then shifts the mantissa with the dual-mode dynamic left shifter. The corresponding left shifting amount is further adjusted in the exponent. The architecture for dual-mode LOD is shown in Fig 6.

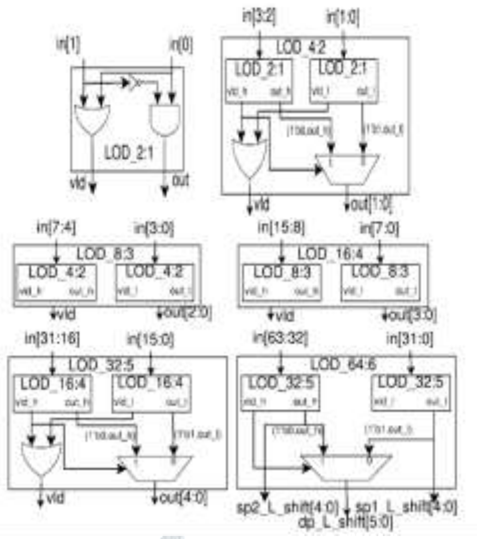


Fig.6 DPdSP Division: Dual-Mode LOD.

The unified mantissas (M1 and M2) are then fed in to the dual-mode dynamic left shifter along with the corresponding left-shifting-amount from LOD units In case of DP mode processing, the SPs left-shift amounts are set to zero, otherwise, the DP left shift amounts are set to zero. The architecture for dual mode dynamic left shifter is shown in Fig. 7

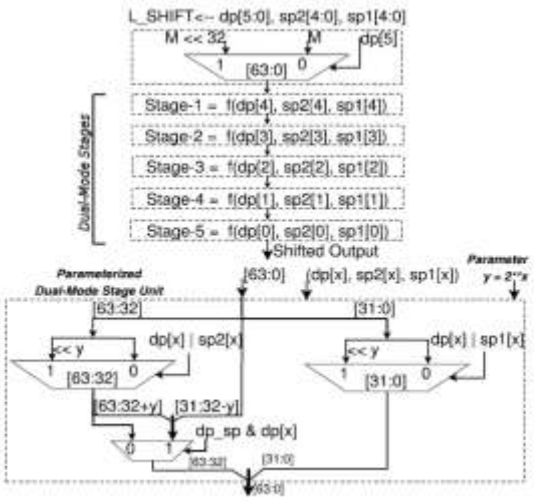

L_SHIFT<-- dp[5:0], sp2[4:0], sp1[4:0]
M << 32
M
dp[5]
[63:0]
Dual-Mode Stages
Stage-1 = f(dp[4], sp2[4], sp1[4])
Stage-2 = f(dp[3], sp2[3], sp1[3])
Stage-3 = f(dp[2], sp2[2], sp1[2])
Stage-4 = f(dp[1], sp2[1], sp1[1])
Stage-5 = f(dp[0], sp2[0], sp1[0])
Shifted Output
Parameterized Dual-Mode Stage Unit
[63:0]
(dp[x], sp2[x], sp1[x])
Parameter y = 2^x
[63:32]
[31:0]
<< y
dp[x] | sp2[x]
<< y
dp[x] | sp1[x]
[63:32]
[31:0]
[63:32+y]
[31:32-y]
dp_sp & dp[x]
[63:32]
[31:0]
[63:0]

Fig. 7 DPdSP Division: Dual-Mode Dynamic Left Shifter

After left shifting, mantissas appear into normalized form $m1$ and $m2$, as shown in Fig. 2. In the next unit in this stage of division architecture, the second stage architecture performs core operation of division architecture. It computes on the core sign, exponent and mantissa processing of FP division arithmetic and the computation related to right shift amount, The computations related to the sign, exponent and right shift amount processing are shown in Fig. 8.

The sign computation is a simple XOR operation among both input operands sign bits. The related exponent computation is the difference of dividend ($in1$) exponent and divisor ($in2$) exponent, with proper BIAS'ing and the adjustment of mantissa left shift amount (LSA):

$$BI\ AS + (Exp\_in1 - LSA\_in1)\ -\ (Exp_{in2} - LSA\_in2)$$

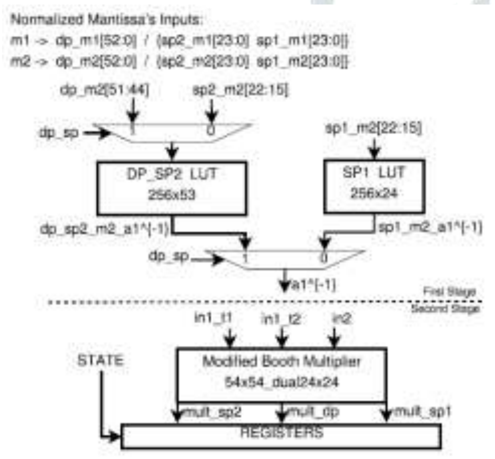


Fig. 8 DPdSP Dual Mode Mantissa Division Architecture

The resultant mantissa division result requires to be shifted right by right-shift-amount (RSA). This wills results into a subnormal output

$$RSA\ =\ (Exp\_in2 - LSA\_in2) - (Exp\_in1 - LSA\_in1) - BI\ AS$$

The mantissa computation is the most complex part of the floating point division arithmetic.

Dual-Mode Radix-4 Modified Booth Multiplier Architecture:

Here, it is a 54-bit integer multiplier (for DP processing), which is also designed to process two parallel sets of 24-bit unsigned operands (for two SPs processing) multiplication. In fact it can process two parallel sets of 26-bit unsigned operand multiplication, using entire 14 partial products of PP1 for first set and entire PP2 for second set operand, without corrupting each other. However, here it is presented for current requirement only

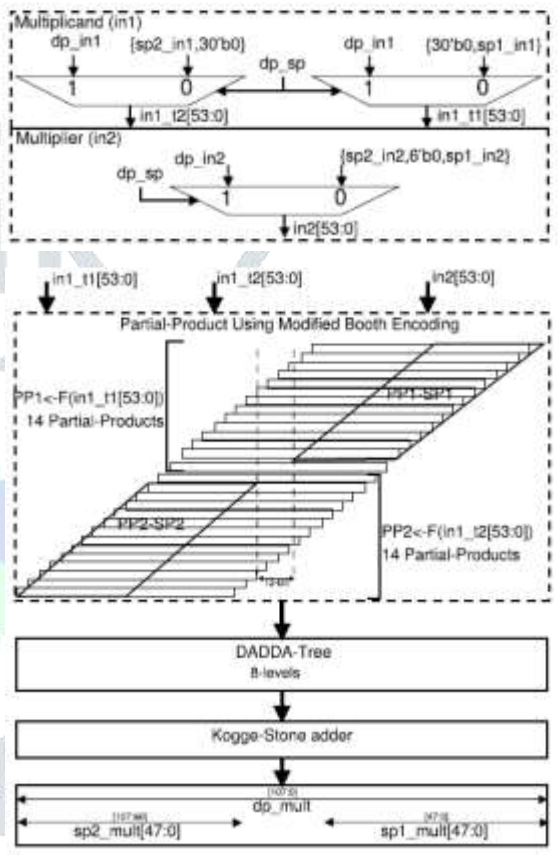


Fig. 9 Dual-Mode Modified Booth Multiplier Architecture

The presented dual-mode Booth architecture has three input operands (two multiplicands and a multiplier). A set of two inputs ($in1\_t1$ and $in1\_t2$) forms the multiplicand operands. Partial products PP1 are the result of $in1\_t1$ and $in2$, and PP2 is derived from $in1\_t2$ and $in2$. In DP mode of operation all the partial products of PP1 and PP2 collectively produce the multiplication result. Whereas, in dual-SP mode, first 13 partial products of PP1 (i.e. PP1-SP1, with all MSBs beyond

25th-bit are zero) results for SP-1 and, the last 13 partial products of PP2

A DADDA-tree of 8 levels is designed to compress all the partial products into two operands, which are further added using a parallel-prefix Kogge-Stone final adder. The final product contains either DP or dual-SP results as shown in Fig. 9. Thus, the few key modification in the Modified Booth multiplication flow leads to a novel dual-mode functioning.

The third stage of the FP division architecture corresponds to the computations in this stage, mantissa division quotient is first process for the dynamic right shifting. This is followed by the dual-mode rounding of the quotient mantissa, and then it undergoes normalization and exceptional case processing. The architecture for dual-mode dynamic right shifter is shown in Fig. 11. The right-shift-amount (Fig. 8) and mantissa quotient acts as primary inputs to the dual-mode dynamic right shifter

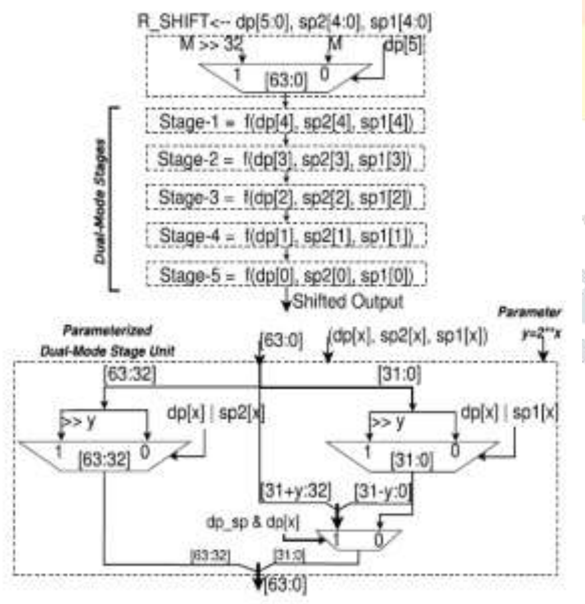

R_SHIFT<-- dp[5:0], sp2[4:0], sp1[4:0]
M >> 32
M
dp[5]
1
[63:0]
0
Stage-1 = f(dp[4], sp2[4], sp1[4])
Stage-2 = f(dp[3], sp2[3], sp1[3])
Stage-3 = f(dp[2], sp2[2], sp1[2])
Stage-4 = f(dp[1], sp2[1], sp1[1])
Stage-5 = f(dp[0], sp2[0], sp1[0])
Dual-Mode Stages
Shifted Output
Parameterized Dual-Mode Stage Unit
Parameter
[63:0]
(dp[x], sp2[x], sp1[x])
[63:32]
[31:0]
>> y
dp[x] | sp2[x]
dp[x] | sp1[x]
[63:32]
[31:0]
[31+y:32]
[31-y:0]
dp_sp & dp[x]
[63:0]

Fig.11 DPdSP Dual-Mode Dynamic Right Shifter

Architecture is designed with faithful rounding, using round-to-nearest method. It is comprised of two steps, first the unit-at-last-place (ULP) computation addition with quotient mantissa.

Existing Dual mode double precision Floating point Division Scheme has more area consumption so that have more output delay. To reduce the total area of this existing multiplier we propose a new approximate adder where it is used at the radix-4 booth multiplication stage. A Dadda tree Partial product reduction stage with approximate adder circuits will give better results but gives approximate results.

IV. PROPOSED DUAL MODE FLOATING POINT DIVISION

It can be dynamically configured for double precision with dual single precision (DPdSP) floating point division arithmetic. The mantissa division is based on the series expansion methodology of division arithmetic. All the components are designed for efficient dual mode processing. Novel dual-mode Radix-4 Modified Booth multiplier architecture is proposed with minimal overhead, for the purpose of dual-mode mantissa processing.

The proposed architecture outperforms the prior art on this in terms of required area, period Based on the current proposed DPdSP division architecture, similar architectures for dual-mode division can be formed using other multiplicative based methods. Multiplication stage is the important stage is the important stage If we enhance the performance of this multiplication stage we can get better results for overall. In the proposed Division Scheme we use new Full adder and half adder designs at Dadda tree reduction stage. By using this Gates We can reduce Area but gives some approximate results

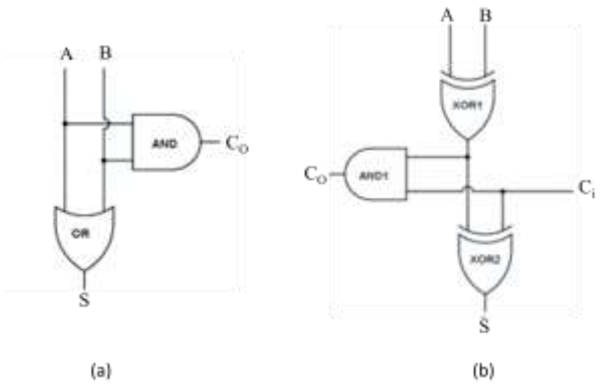


Fig.12 New proposed Approximate Adder designs

(a) Half adder, (b) Full adder

After left shifting, mantissas appear into normalized form *m*1 and *m*2, as shown in Fig. 2. In the next unit in this stage of division architecture, the 8-bit (after decimal point position) MSB party (*a*1) of normalized divisor mantissas (*m*2) is used to fetch the pre-computed initial approximation of their inverse. After calculating the inverse of Mantissa m2 then we multiply it with mantissa m1 which uses radix-4 booth multiplier which is given in below figure

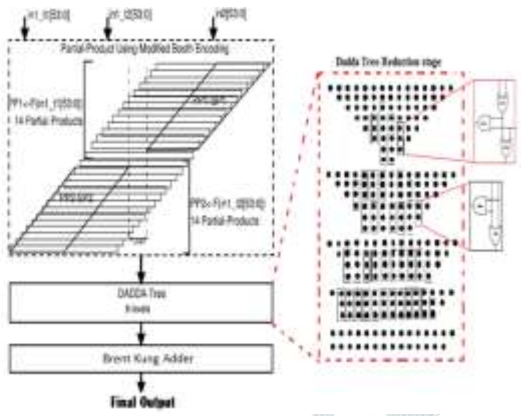
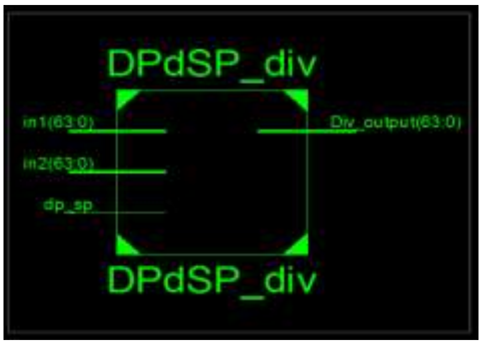
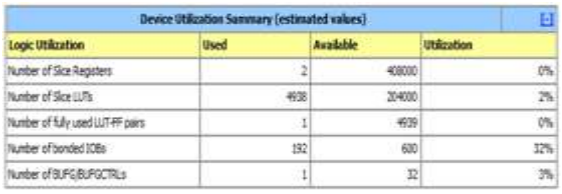


Fig.13 Proposed multiplication stage with new HA's and FA's

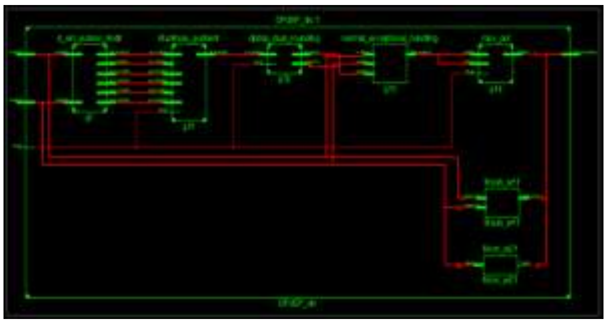
V. RESULTS AND DISCUSSION

The proposed dual-mode architectures Multiplication stage is done with approximate adder circuits which can give better results in terms of Area Compared to standalone Dual mode double precision division architecture. In comparison to prior art on this, the proposed architecture out-performs them in terms of required area, time period. The proposed dual-mode architecture is synthesized Virtex 7 FPGA family In Xilinx ISE.





Device Utilization Summary (estimated values)			
Logic Utilization	Used	Available	Utilization
Number of Slice Registers	2	408000	0%
Number of Slice LUTs	4938	204000	2%
Number of fully used LUT-FF pairs	1	4939	0%
Number of bonded IOBs	192	600	32%
Number of BUFG/BUFGCTRLs	1	32	3%



VI. CONCLUSION

In This paper we implemented that the dual mode (1doubel/2singel) precision floating point division using Radix-4 modified booth multiplier. In radix-4 modified booth multiplier we use the Dadda tree reduction with approximate adders and Brant kung adder for improvement in area and speed of operation. By using this adder the area is reduced to 22% and delay is reduced to 2% as compared with existing method.

REFERENCES

[1] Manish Kumar Jaiswal and Hayden K. H. So, "Area-Efficient Architecture for Dual-Mode Double Precision Floating Point Division", *IEEE transactions on circuits and systems–i: regular papers, vol. 64, no. 2, february 2017.*

[2] S. F. Oberman and M. Flynn, "Division algorithms and implementations," *IEEE Trans. Comput.*, vol. 46, no. 8, pp. 833–854, Aug. 1997.

[3] M. K. Jaiswal and R. C. C. Cheung, "High performance reconfigurable architecture for double precision floating point division," in *Proc. 8th Int. Symp. Appl. Reconfigurable Comput. (ARC)*, Hong Kong, China, Mar. 2012, pp. 302–313.

[4] 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.

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

[6] B. Pasca, "Correctly rounded floating-point division for DSP-enabled FPGAs," in *Proc. 22nd Int. Conf. Field Program. Logic Appl. (FPL)*, Aug. 2012, pp. 249–254

[7] Akka¸s, "Dual-mode quadruple precision floating-point adder," in *Proc. Euromicro Symp. Digit. Syst. Design*, 2006, pp. 211–220.

[8] M. Jaiswal, R. Cheung, M. Balakrishnan, and K. Paul, "Configurable architecture for double/two-parallel single precision floating point division," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, Jul. 2014, pp. 332–337.

[9] IEEE Standard for Floating-Point Arithmetic, IEEE Standard 754-2008, Aug. 2008, pp.1-70