ijiTECH International Journal of Innovative Technologies

WWW.IJITECH.ORG

ISSN 2321-8665
Vol.03,Issue.11,
December-2015,
Pages:2107-2112

An Efficient Implementation of Floating Point Multiplier using Verilog

G. SRUTHI[1], M. RAJENDRA PRASAD[2]
[1]PG Scholar, Dept of ECE, VidyaJyothi Institute of Technology, JNTU Hyderabad, Telangana, India,
Email: gottesruthi@gmail.com.
[2]Assoc Prof, Dept of ECE, VidyaJyothi Institute of Technology, JNTU Hyderabad, Telangana, India,
Email: rajendraprasad@vjit.ac.in.

Abstract: To represent very large or small values, large range is required as the integer representation is no longer appropriate. These values can be represented using the IEEE-754 standard based floating point representation. Floating point multiplication is a most widely used operation in DSP/Math processors, robots, air traffic controller, digital computers. Because of its vast areas of application, the main emphasis is on the implementing it effectively such that it uses less combinational delay with high Speed This project implements high speed implementation of a floating point arithmetic unit which can perform multiplication function on 32-bit operands. Multiplication is one of the common arithmetic operations; this floating point multiplication handles various conditions like overflow, underflow, normalization, rounding. In this project we use IEEE rounding method for perform the rounding of the resulted number. This project reviews the implementation of an IEEE 754 single precision floating point multiplier the multiplier implementation handles the overflow and underflow cases. Pre-normalization unit and post normalization units are also discussed along with exceptional handling. All the functions are built by feasible efficient algorithms with several changes incorporated that can improve overall latency, and if pipelined then higher throughput. The algorithms are modeled in Verilog HDL, the RTL code for multiplier is synthesized using Xilinx and the multiplier is simulated using Model Sim.

Keywords: Normalization Unit, Higher Throughput, Verilog HDL, Adder, Multiplier.

I. INTRODUCTION

The demand for floating point arithmetic operations in most of the business, financial and web based mostly applications is increasing day by day. So it becomes essential to seek out an option to feed binary numbers directly as input for these applications. This helps in saving time and is way easier. within the current situation, this is often unattainable, because, within the adder/subtractor, inputs ought to lean in IEEE 754 format [1]. The binary inputs can't be given in and of itself, however it must be reborn to the sign, exponent and mantissa form, regarding that, are going to be delineate very well later. Hence during this project we've enforced a binary to floating point convertor for single exactness bits which can solve this issue to anextend. The convertor is predicated on IEEE single exactness format and this is often thirty two bits wide. Numerous modules square measure written victimisation Verilog Hardware Description Language [6] and is simulated with the assistance of Xilinx. They're then synthesized victimisation Xilinx Integrated Software Environment (ISE) design suite. The work has been disbursed at Centre for Development and Advanced Computing (C-DAC) wherever a 32- bit floating point adder/subtractor module [3] in line with IEEE - 754 format is already enforced and is presently in use for several specific applications. The projected convertor may be further into the already existing adder/subtractor to urge the total practicality of the design[9]. the basic distinction between fastened and floating purpose digital signal processors (DSPs) is their various numeric illustration of knowledge. Whereas fastened purpose hardware performs strictly number arithmetic, floating purpose DSPs support number or real arithmetic, the latter normalized within the style of scientific notation. A 32-bit, binary floating purpose DSP [5], supporting industry-standard, single exactness operations, provides bigger accuracy and bigger exactness than fastened purpose devices thanks to its wider word breadth, operation and actual internal representations of knowledge. Fastened purpose devices had to implement real arithmetic indirectly through package routines that add recursive directions [2] and development time, whereas with floating purpose format, real arithmetic may well be coded directly into hardware operations. So, this thesis emphasizes on utilizing the capabilities of floating purpose format. The binary input given can vary from 0-256 bits,that is that the most input vary which will be provided which can satisfy the exponent zero in the thirty two bit IEEE 754 single precision format.

II. BINARY TO FLOATING POINT CONVERSION

Converting a base ten complex quantity into associate degree IEEE 754 binary32 format [4] exploitation the subsequent outline:

- Consider a true range with associate degree number and a fraction half like twelve.375.
- Convert and normalize the number half into binary.

Copyright @ 2015 IJIT. All rights reserved.

- Convert the fraction half exploitation the subsequent methodology shown below.
- Add 2 results and modify them to provide a correct final conversion.

Conversion of the aliquot half is finished as shown below: contemplate zero.375, the aliquot a part of twelve.375. To convert it into a binary fraction, multiply the fraction by two, take the number half and re-multiply new fraction by two till a fraction of zero is found or till the preciseness limit is reached that is twenty three fraction digits for IEEE 754 binary32 format.

0.375 x two = zero.750 = zero + zero.750 => b-2 = zero, the number half represents the binary fraction digit. Next step is to re-multiply zero.750 by two to proceed.

0.750 x two = one.500 = one + zero.500 => b-2 = one

0.500 x two = one.000 = one + zero.000 => b-2 = one, fraction = zero.000, terminate.

We see that (0.375)10 will be precisely described in binary as (0.011)2. Not all decimal fractions will be described in a very finite digit binary fraction. As an example decimal zero.1 cannot be described in binary precisely. Therefore it's solely approximated.

Therefore (12.375)10 = (12)10 + (0.375)10 = (1100)2 + (0.011)2 = (1100.011)2

- Conjointly in IEEE 754 binary32 format at real values got to be described in normalized form
- Hence it becomes one.100011 x twenty three from this
- The exponent is three (and within the biased kind it's so 127+3=130 = (1000 0010)2).
- The fraction is 100011 (looking to the proper of the binary point)

The ensuing thirty two bit IEEE 754 binary32 format illustration of twelve.375 as: 0-10000010-100011000000 00000000000 = 41460000H.

III. SINGLE PRECISION FLOATING POINT MULTIPLIER

Floating point numbers are one possible way of representing real numbers in binary format; the IEEE 754 [1] standard presents two different floating point formats, Binary interchange format and Decimal interchange format. Multiplying floating point numbers is a critical requirement for DSP applications involving large dynamic range. This paper focuses only on single precision normalized binary interchange format. Fig.1 shows the IEEE 754 single precision binary format representation; it consists of a one bit sign (S), an eight bit exponent (E), and a twenty three bit fraction (M or Mantissa). An extra bit is added to the fraction to form what is called the significand1. If the exponent is greater than 0 and smaller than 255, and there is 1 in the MSB of the significand then the number is said to be a normalized number;In this case the real number is represented by (1)

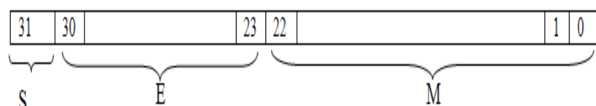

Figure1. IEEE single precision floating point format

$$Z = (-1^{S}) * 2^{(E - Bias)} * (1.M) \quad (1)$$

Where $M = m_{22}2^{-1} + m_{21}2^{-2} + m_{20}2^{-3}+\ldots+ m_{1}2^{-22}+ m_{0}\, 2^{-23}$;

Bias = 127.

Multiplying two numbers in floating point format is done by 1- adding the exponent of the two numbers then subtracting the bias from their result, 2- multiplying the significand of the two numbers, and 3- calculating the sign by XORing the sign of the two numbers. In order to represent the multiplication result as a normalized number there should be 1 in the MSB of the result (leading one). Floating-point implementation on FPGAs has been the interest of many researchers. In [2], an IEEE 754 single precision pipelined floating point multiplier was implemented on multiple FPGAs (4 Actel A1280).

A. Floating Point Multiplication Algorithm

As stated in the introduction, normalized floating point numbers have the form of $Z = (-1^{S}) * 2^{(E - Bias)} * (1.M)$. Tomultiply two floating point numbers the following is done:

1. Multiplying the significand; i.e. $(1.M_1*1.M_2)$
2. Placing the decimal point in the result
3. Adding the exponents; i.e. ($E_1 + E_2 -$ *Bias*)
4. Obtaining the sign; i.e. $s_1$xor s_2
5. Normalizing the result; i.e. obtaining 1 at the MSB ofthe results' significand
6. Rounding the result to fit in the available bits
7. Checking for underflow/overflow occurrence

Consider a floating point representation similar to the IEEE754 single precision floating point format, but with a reducednumber of mantissa bits (only 4) while still retaining the hidden

'1' bit for normalized numbers:

A = 0 10000100 0100 = 40, B = 1 10000001 1110 = -7.5

To multiply A and B

1. Multiply significand: 1.0100

$\times$ 1.1110

____

00000

10100

10100

10100

10100

____

1001011000

2. Place the decimal point: 10.01011000

3. Add exponents: 10000100

+ 10000001

____

100000101

The exponent representing the two numbers is already shifted/biased by the bias value (127) and is not the true exponent; i.e. $E_A = E_{A\text{-true}} + \text{bias}$ and $E_B = E_{B\text{-true}} + \text{bias}$

And

$E_A + E_B = E_{A\text{-true}} + E_{B\text{-true}} + 2\ \text{bias}$

So we should subtract the bias from the resultant exponent otherwise the bias will be added twice.

100000101

- 01111111

____

10000110

4. Obtain the sign bit and put the result together:

1 10000110 10.01011000

5. Normalize the result so that there is a 1 just before the radix point (decimal point). Moving the radix point one place to the left increments the exponent by 1; moving one place to the right decrements the exponent by 1.

1 10000110 10.01011000 (before normalizing)

1 10000111 1.001011000 (normalized)

The result is (without the hidden bit):

1 10000111 00101100

6. The mantissa bits are more than 4 bits (mantissa available bits); rounding is needed. If we applied the truncation rounding mode then the stored value is:

1 10000111 0010.

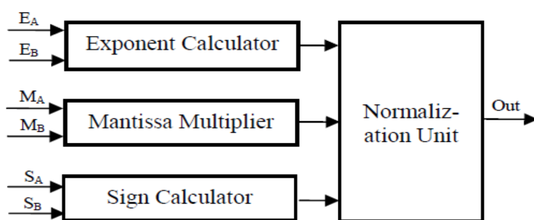

Figure2. Floating point multiplier block diagram

In this paper we present a floating point multiplier in which normalizing is implemented. Figure2.shows the multiplier structure; Exponents addition, Significand multiplication, and Result's sign calculation are independent and are done in parallel. The significand multiplication is done on two 8 bit numbers and results in a 16 bit product, which we will call the intermediate product (IP). The IP is represented as (15 down to 0) and the decimal point is located between bits 15 and 14 in the IP. The following sections detail each block of the floating point multiplier.

IV.DADDA MULTIPLIER (MANTISSA MULTIPLICATION)

Dadda proposed a sequence of matrix heights that are predetermined to give the minimum number of reduction stages. To reduce the N by N partial product matrix, dada multiplier develops a sequence of matrix heights that are found by working back from the final two-row matrix. In order to realize the minimum number of reduction stages, the height of each intermediate matrix is limited to the least integer that is no more than 1.5 times the height of its successor. The process of reduction for a dadda multiplier [4] is developed using the following recursive algorithm.

1. Let d1=2 and dj+1 = [1.5*dj], where dj is the matrix height for the jth stage from the end. Find the smallest j such that at least one column of the original partial product matrix has more than dj bits.
2. In the jth stage from the end, employ (3, 2) and (2, 2) counter to obtain a reduced matrix with no more than dj bits in any column.
3. Let j = j-1 and repeat step 2 until a matrix with only two rows is generated.

This method of reduction, because it attempts to compress each column, is called a column compression technique. Another advantage of utilizing Dadda multipliers is that it utilizes the minimum number of (3, 2) counters. For Daddamultipliers there are N2 bits in the original partial product matrix and 4.N-3 bits in the final two row matrix. Since each (3, 2) counter takes three inputs and produces two outputs, the number of bits in the matrix is reduced by one with each applied (3, 2) counter therefore}, the total number of (3,2) counters is #(3, 2) = N2 – 4.N+3 the length of the carry propagation adder is CPA length = 2.N–2. The 8 by 8 multiplier takes 4 reduction stages, with matrix height 6, 4, 3 and 2. The reduction uses 35 (3, 2) counters, 7 (2, 2) counters, reduction uses 35 (3, 2) counters, 7 (2, 2) counters, and a 14-bit carry propagate adder. The total delay for the generation of the final product is the sum of one AND gate delay, one (3, 2) counter delay for each of the four reduction stages, and the delay through the final 14-bit carry propagate adder arrive later, which effectively reduces the worst case delay of carry propagate adder. The decimal point is between bits 45 and 46 in the significand IR. Critical path is used to determine the time taken by the Dadda multiplier. The critical path starts at the AND gate of the first partial products passes through the full adder of the each stage, then passes through all the vector merging adders. The stages are less in this multiplier compared to the carry save multiplier and therefore it has high speed

A. DADDA Algorithm

The performance of Mantissa calculation Unit determines overall performance of theFloating Point Multiplier. The picture shown in figure 3 elucidates the DaddaAlgorithm. This algorithm goes on as follows:

1. Multiply (logical AND) each bit of one of the inputs, by each bit of the otheryielding partial product matrix.
2. Reduce the partial product matrix to two vectors using series of full and halfadders.
3. This reduction can be done with the help of height of each level.
4. The height of the preceding level is the height of the succeeding level * (1.5).
5. Till the height is less than the number of bits in the operands we are operatingon.
6. Make the vectors into two numbers, and add them with a conventional multi bitadder.

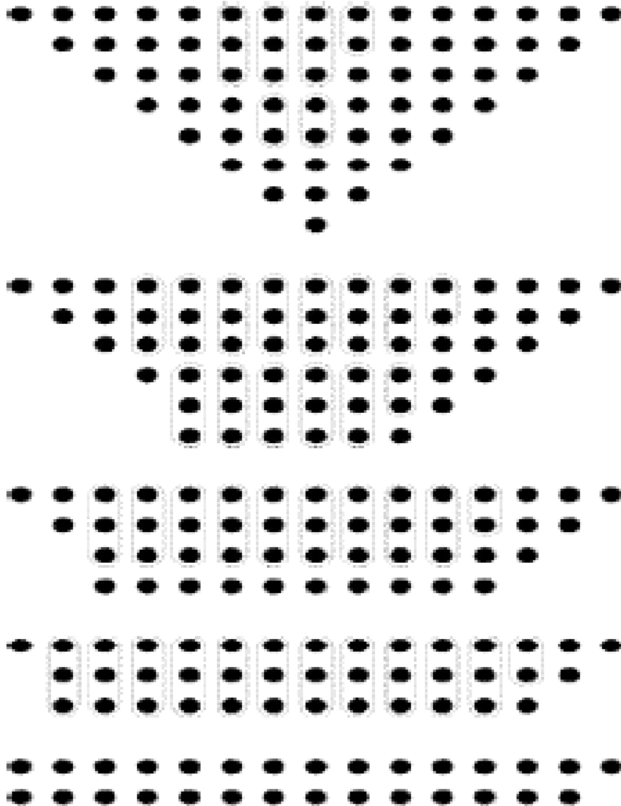

Fig.3. Implementation of Dot Diagram for Dadda Multiplier.

B. Normalizer

The result of the significant multiplication (intermediate product) must be normalized to have a leading '1' just to theleft of the decimal point (i.e. in the bit 46 in the intermediate product). Since the inputs are normalized numbers then the intermediate product has the leading one at bit 46 or 47

1. If the leading one is at bit 46 (i.e. to the left of the decimal point) then the intermediate product is already a normalized number and no shift is needed.
2. If the leading one is at bit 47 then the intermediate product is shifted to the right and the Exponent is incremented by 1.

The shift operation is done using combinational shift logic made by multiplexers. Fig 4 shows a simplified logic of aNormalizer that has an 8 bit intermediate product input and a 6bit intermediate exponent input.

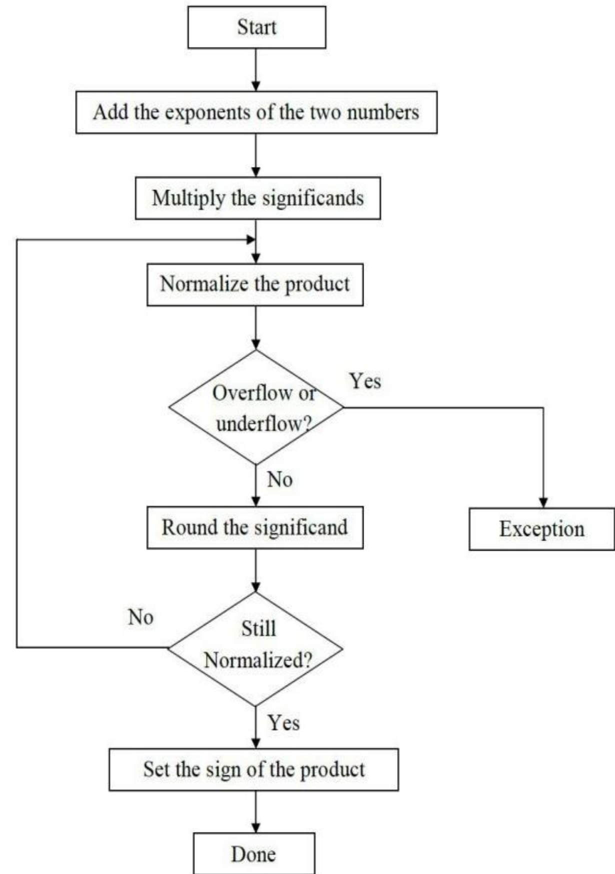

Fig.4.Flow Graph of Floating Point Multiplier

V. UNDERFLOW/OVERFLOW DETECTION

Overflow/underflow means that the result's exponent is too large/small to be represented in the exponent field. The exponent of the result must be 11 bits in size, and must be between 1 and 2048 otherwise the value is not a normalized one. An overflow may occur while adding the two exponents or during normalization. Overflow due to exponent addition may be compensated during subtraction of the bias; resulting in a normal output value (normal operation). An underflow may occur while subtracting the bias to form the intermediate exponent. If the intermediate exponent < 0 then it's an underflow that can never be compensated; if the intermediate exponent = 0 then it's an underflow that may be compensated during normalization by adding 1 to it. When an overflow occurs an overflow flag signal goes high and the result turns to ±Infinity (sign determined according to the sign of the floating point multiplier inputs). When an underflow occurs an underflow flag signal goes high and the result turns to ±Zero (sign determined according to the sign of the floating point multiplier inputs). Denormalized numbers are signaled to Zero with the appropriate sign calculated from the inputs and an underflow flag is raised. Assume that E1 and E2 are the exponents of the two numbers A and B respectively. The result's exponent is calculated by below formula.

$$Eresult = E1 + E2 - 1023$$

E1 and E2 can have the values from 1 to 2047; resulting in E result having values from -1021 (2-1023) to 381 (1508-1023);

but for normalized numbers, Eresult can only have the values from 1 to 254.

VI. PIPELINING THE MULTIPLIER

In order to enhance the performance of the multiplier, three pipelining stages are used to divide the critical path thus increasing the maximum operating frequency of the multiplier. The pipelining stages are imbedded at the following locations:

1. In the middle of the significand multiplier, and in the middle of the exponent adder (before the bias subtraction).
2. After the significand multiplier, and after the exponent adder.
3. At the floating point multiplier outputs (sign, exponent and mantissa bits).

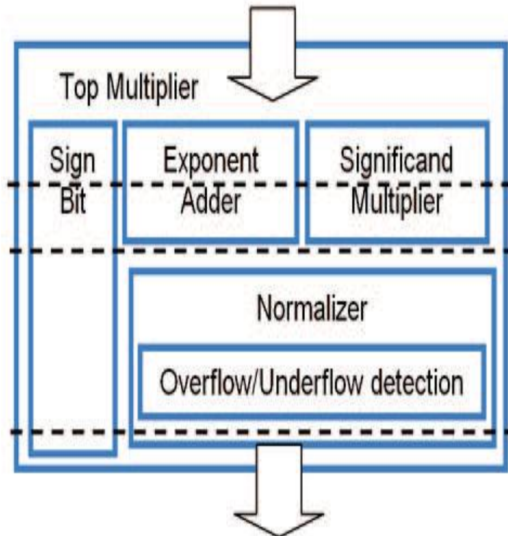

Fig.5. Pipelining stages as dotted lines.

Three pipelining stages mean that there is latency in the output by three clocks. The synthesis tool "retiming" option was used so that the synthesizer uses its optimization logic to better place the pipelining registers across the critical path.

VII. SIMULATION RESULTS

In this chapter all the simulation results which are done using Xilinx simulator are shown and also synthesis results.

A. Simulation Results

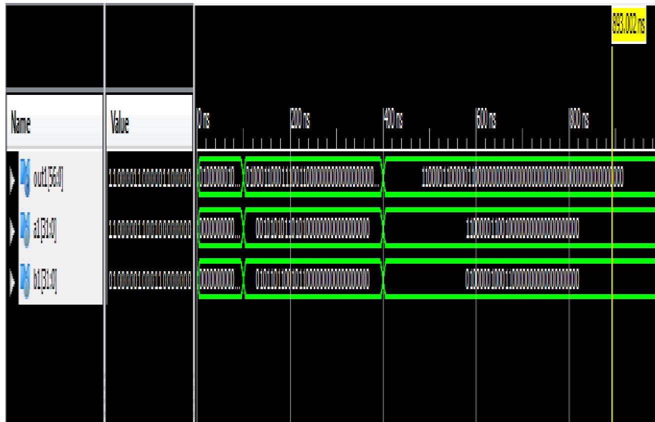

Fig.6. Simulation result for Single precision Floating point multiplier.

B. Synthesis Result

1.Technological View

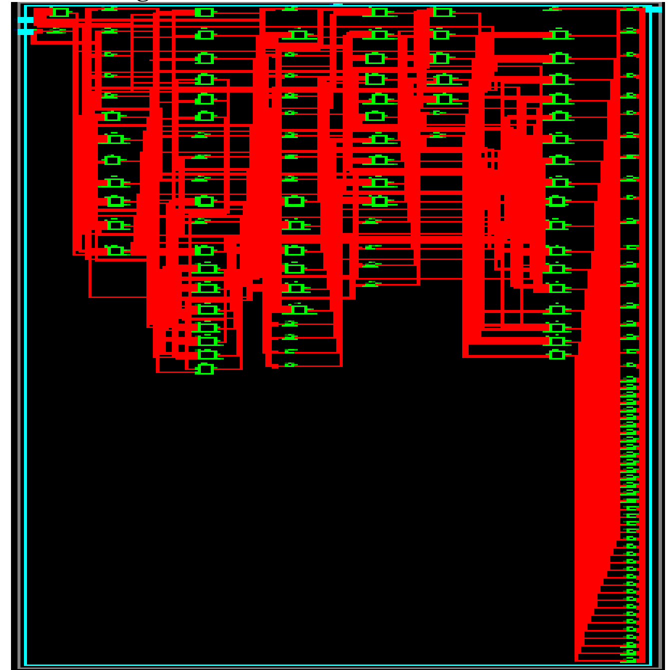

Fig.6. Technological view of the multiplier

3.RTL diagram

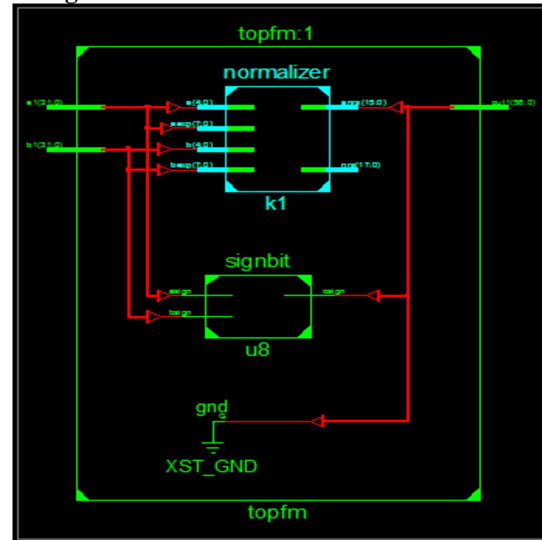

Fig.8. Single precision floating point RTL schematic

VIII. CONCLUSION

This paper describes an implementation of a floating point multiplier using Dadda Multiplier that supports the IEEE 754-2008 binary interchange format. To improve speed multiplication of mantissa is done using Dadda multiplier replacing Carry Save Multiplier. The design achieves high speed with maximum frequency of 526 MHz compared to existing floating point multipliers.

IX.REFERENCES

[1] Mohamed Al-Ashrfy, Ashraf Salem and WagdyAnis "An Efficient implementation of Floating Point Multiplier" IEEE Transaction on VLSI 978-1-4577-0069-9/11@2011 IEEE,Mentor Graphics.

[2] B. Fagin and C. Renard, “Field Programmable Gate Arrays and Floating Point Arithmetic,” IEEE Transactions on VLSI, vol. 2, no. 3, pp. 365-367, 1994.

[3] N. Shirazi, A. Walters, and P. Athanas, “Quantitative Analysis of Floating Point Arithmetion FPGA Based Custom Computing Machines,” Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM’95), pp.155-162, 1995.

[4] L. Louca, T. A. Cook, and W. H. Johnson, “Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs,” Proceedings of 83 the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM’96), pp. 107-116, 1996.

[5] Jaenicke and W. Luk, "Parameterized Floating-Point Arithmetic on FPGAs", Proc. of IEEE ICASSP, 2001, vol. 2, pp.897-900.

[6] Whytney J. Townsend, Earl E. Swartz, “A Comparison of Dadda and Wallace multiplier delays”. Computer Engineering Research Center, the University of Texas.

[7] B. Lee and N. Burgess, “Parameterisable Floating-point Operations on FPGA,” Conference Record of the Thirty-Sixth Asilomar Conference on Signals, Systems, and Computers, 2002.

[8] Xilinx13.4, Synthesis and Simulation Design Guide”, UG626 (v13.4)
January 19, 2012.

[9] “DesignChecker User Guide”, HDL Designer Series 2010.2a, Mentor Graphics, 2010.

[10] “PrecisionR Synthesis User’s Manual”, Precision RTL plus 2010a update 2, Mentor Graphics, 2010.

[11] Patterson, D. & Hennessy, J. (2005), Computer Organization and Design: The Hardware/software Interface , Morgan Kaufmann.

[12] John G. Proakis and Dimitris G. Manolakis (1996), “Digital Signal Processing: Principles,.Algorithms and Applications”, Third Edition.

[13] L. Song, K.K. Parhi, “Efficient Finite Field Serial/ Parallel Multiplication”, Proc. of International Conf. on Application Specific Systems, Architectures and Processors, pp. 72-82, Chicago, USA, 1996.

[14] P. E. Madrid, B. Millar, and E. E. Swartzlander, “Modified Booth algorithm for high radix fixed- point multiplication,” IEEE Trans. VLSI Syst., vol. 1, no. 2, pp. 164-167, June 1993.

[15] A Booth, “A signed binary multiplication technique,” Q. J. Me& Appl. March., vol. 4, pp. 236-240, 19.51.

[16] Nhon T. Quach, Member, IEEE, Naofumi Takagi, Senior Member, IEEE, and Michael J. Flynn, Fellow, IEEE” Systematic IEEE Rounding Method for High-Speed Floating-Point Multipliers” IEEE transactions on very large scale integration (vlsi) systems, vol. 12, no. 5, may 2004.

[17] Report on Efficient Floating Point 32-bit single Precision Multipliers Design using VHDL by Dr. Raj Singh, Group Leader, VLSI Group, CEERI, Pilani.

[18] Xianyang Jianga, Peng Xiaoa, Meikang Qiub, Gaofeng Wanga” Performance effects of pipeline architecture on an FPGA-based binary32 bit floating point multiplier “Microprocessors and Microsystems xxx (2013) xxx–xxx.

[19] Mamu Bin Ibne Reaz, MEEE, Md. Shabiul Islam, MEEE, Mohd. S. Sulaiman, MEEE Faculty of Engineering, Multimedia University, 63 100 Cybejaya, Selangor, Malaysia “Pipeline Floating Point ALU Design using VHDL “ICSE2002 Proc. 2002, Penang, Malaysia.

[20] Al-Ashrafy M., Salem A. and Anis W., ―An Efficient Implementation of Floating Point Multiplier‖, 2011. [2] Eldon A.J., Robertson C., A Floating Point Format For Signal Processing‖, pp. 717-720, 1982.

[21] Brisebarre N., Muller J.M., ―Correctly Rounded Multiplication by Arbitrary Precision Constants‖, Symposium on Computer Arithmetic, pp. 1-8, 2005.

[22] Enriquez A.B., and JONES K.R., ―Design of a MultiMode Pipelined Multiplier for Floating-point Applications‖,pp. 77-81, 1991.

[23] Amaricai A., Vladutiu M., Udrescu M., Prodan L. and Boncalo O., ―Floating Point Multiplication Rounding Schemes for Interval Arithmetic‖, pp. 19-24, 2008.

[24] Louca L., Cook T.A. and Johnson W.H., ―Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs‖, pp. 107-116, 1996.

[25] Awan M.A., Siddiqui M.R., ―Resolving IEEE FloatingPoint Error using Precision-Based Rounding Algorithm‖, pp. 329-333, 2005.

[26] Fagin B., Renard C., ―Field Programmable Gate Arrays and Floating Point Arithmetic‖, pp. 365-367, Vol. 2, No. 3, 1994.

[27]Nhon T. Quach, Member, IEEE, Naofumi Takagi, Senior Member, IEEE, and Michael J. Flynn, Fellow, IEEE” Systematic IEEE Rounding Method for High-Speed Floating-Point Multipliers” IEEE transactions on very large scale integration (vlsi) systems, vol. 12, no. 5, may 2004.

[28] Report on Efficient Floating Point 32-bit single Precision Multipliers Design using VHDL by Dr. Raj Singh, Group Leader, VLSI Group, CEERI, Pilani.

[29] Loucas Louca, Todd A. Cook, William H. Johnson, “Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs”, IEEE, Sept. 1996

[30] Pardeep Sharma, Ajay Pal Singh,“Implementation of Floating Point Multiplier on Reconfigurable Hardware and Study its Effect on 4 input LUT‘s”, International Journal of Advanced Research in Computer Science and Software Engineering, Volume 2, Issue 7, July 2012, pp 244-248.