# Simulation of IEEE 754 Standard Double Precision Multiplier using Booth Techniques

**[1]Puneet Paruthi, [2]Tanvi Kumar, [3]Himanshu Singh**
[1,2]Dept. of ECE, BMSCE, Mukhsar, [3]Asstt. Professor,ISTK,Kalawad.

## Abstract

**Multiplication is an important fundamental function in arithmetic operations. It can be performed with the help of different multipliers using different techniques. The objective of good multiplier is to provide a physically compact high speed and low power consumption. To save significant power consumption of multiplier design, it is a good direction to reduce number of operations thereby reducing a dynamic power which is a major part of total power dissipation. The main objective of this Dissertation is to design "Simulation of IEEE 754 standard double precision multiplier" using VHDL.**

**Keywords** - IEEE floating point arithmetic; Rounding; Floating point multiplier

## I. INTRODUCTION

Every computer has a floating point processor or a dedicated accelerator that fulfils the requirements of precision using detailed floating point arithmetic. The main applications of floating points today are in the field of medical imaging, biometrics, motion capture and audio applications. Since multiplication dominates the execution time of most DSP algorithms, so there is a need of high speed multiplier with more accuracy. Reducing the time delay and power consumption are very essential requirements for many applications. Floating Point Numbers: The term floating point is derived from the fact that there is no fixed number of digits before and after the decimal point, that is, the decimal point can float. There are also representations in which the number of digits before and after the decimal point is set, called fixed-point representations.

Floating Point Numbers are numbers that can contain a fractional part. E.g. following numbers are the floating point numbers: 3.0, -111.5, ½, 3E-5 etc. IEEE Standard for Binary Floating Point Arithmetic. The Institute of Electrical and Electronics Engineers (IEEE) sponsored a standard format for 32-bit and larger floating point numbers, known as IEEE 754 standard.

This paper presents a new floating-point multiplier which can perform a double-precision floating-point multiplication or simultaneous single precision floating-point multiplications. Since in single precision floating-point multiplication results are generated in parallel, the multiplier's performance is almost doubled compared to a conventional floating point multiplier.

### A. Floating Point Arithmetic

The IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754) is the most widely used standard for floating-point computation, and is followed by many CPU and FPU implementations. The standard defines formats for representing floating-point number (including ±zero and denormals) and special values (infinities and NaNs) together with a set of floating-point operations that operate on these values. It also specifies four rounding modes and five exceptions. IEEE 754 specifies four formats for representing floating-point values: single-precision (32-bit), double-precision (64-bit), single-extended precision (≥ 43-bit, not commonly used) and double-extended precision (≥ 79-bit, usually implemented with 80 bits). Many languages specify that IEEE formats and arithmetic be implemented, although sometimes it is optional. For example, the C programming language, which pre-dated IEEE 754, now allows but does not require IEEE arithmetic (the C float typically is used for IEEE single-precision and double uses IEEE double-precision).

### B. Double Precision Floating Point Numbers

Thus, a total of 64 bits is needed for double-precision number representation. To achieve a bias equal to $2^{n-1} - 1$ is added to the actual exponent in order to obtain the stored exponent. This equal 1023 for an 11-bit exponent of the double precision format. The addition of bias allows the use of an exponent in the range from −1023 to +1024, corresponding to a range of 0.2047 for double precision number. The double precision format offers a range from $2^{-1023}$ to $2^{+1023}$, which is equivalent to $10^{-308}$ to $10^{+308}$.
Sign: 1-bit wide and used to denote the sign of the number i.e. 0 indicate positive number and 1 represent negative number.
Exponent: 11-bit wide and signed exponent in excess- 1023 representation. Mantissa: 52-bit wide and fractional component

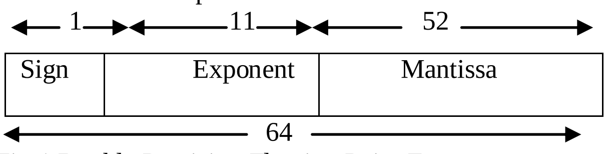


Fig.1 Double Precision Floating Point Format

### C. Floating-Point Multiplication

Multiplication of two floating point normalized numbers is performed by multiplying the fractional components, adding the exponents, and an exclusive or operation of the sign fields of both of the operands The most complicated part is performing the integer-like multiplication on the fraction fields . Essentially the multiplication is done in two steps, partial product generation and partial product addition. For double precision operands (53-bit fraction fields), a total of 53 53-bit partial products are generated.

The general form of the representation of floating point is:

$$(-1)\ S * M * 2E$$

Where

S represents the sign bit, M represents the mantissa and E represents the exponent.

Given two FP numbers n1 and n2, the product of both, denoted as n, can be expressed as:

$n = n1 \times n2$

$= (-1)\ S1 \cdot p1 \cdot 2^{E1} \times (-1)\ S2 \cdot p2 \cdot 2^{E2}$

$= (-1)^{S1+S2} \cdot (p1 \cdot p2) \cdot 2^{E1+E2}$

In order to perform floating-point multiplication, a simple algorithm is realized:

- Add the exponents and subtract 1023.
- Multiply the mantissas and determine the sign of the result.
- Normalize the resulting value, if necessary.

### D. Model Sim Overview

ModelSim is a verification and simulation tool for VHDL, Verilog, SystemVerilog, SystemC, and mixed-language designs. ModelSim VHDL implements the VHDL language as defined by IEEE Standards 1076-1987, 1076-1993, and 1076-2002. ModelSim also supports the 1164-1993 Standard Multivalue Logic System for VHDL Interoperability, and the 1076.2-1996 Standard VHDL Mathematical Packages standards. Any design developed with ModelSim will be compatible with any other VHDL system that is compliant with the 1076 specifications

## II. LITERATURE SURVEY

A few research work have been conducted to explain the concept of Floating Point Numbers. D. Goldberg [11] explained the concept of Floating Point Numbers used to describe very small to very large numbers with a varying level of precision. They are comprised of three fields, a sign, a fraction and an exponent field. B. Parhami [8] proposed IEEE-754 standard defining several floating point number formats and the size of the fields that comprise them. This Standard defines several rounding schemes, which include round to zero, round to infinity, round to negative infinity, and round to nearest. Michael L. Overton [7] performed the multiplication of two floating point normalized numbers by multiplying the fractional components, adding the exponents, and an Exclusive OR operation of the sign fields of both of the operands. Cho, J. Hong et al. and N. Besli et al.[5][6] multiplied double precision operands (53-bit fraction fields),in which a total of 53 53-bit partial products are generated . To speed up this process, the two obvious solutions are to generate fewer partial products and to sum them faster. Sumit Vaidya et al.[1] compared the different multipliers on the basis of power, speed, delay and area to get the efficient multiplier. It can be concluded that array Multiplier requires more power consumption and gives optimum number of components required, but delay for this multiplier is larger than Wallace Tree Multiplier. Hasan Krad et al.[4] presented a performance analysis of two different multipliers for unsigned data, one uses a carry-look-ahead adder and the second one uses a ripple adder. In this author said that the multiplier with a carry-look-ahead adder has shown a better performance over the multiplier with a ripple adder in terms of gate delays. In other words, the multiplier with the carrylook-ahead adder has approximately twice the speed of the multiplier with the ripple adder, under the worst case. Soojin Kim et al.[2] described the pipeline architecture of high-speed modified Booth multipliers. The proposed multiplier circuits are based on the modified Booth algorithm and the pipeline technique which are the most widely used to accelerate the multiplication speed. In order to implement the optimally pipelined multipliers, many kinds of experiments have been conducted. P. Assady [3] presented a new high-speed algorithm. As multipler has to do three important steps, which include partial product generation, partial product reduction, and final addition step. In partial product generation step, a new Booth algorithm has been presented. In partial product reduction step, a new tree structure has been designed and in final addition step, a new hybrid adder using 4-bit blocks has been proposed.

## III.METHODOLOGY

### A. Methods for multiplication

There are number of techniques that can be used to perform multiplication. In general, the choice is based upon factors such as latency, throughput, area, and design complexity.

a) Array Multiplier b) Booth Multiplier

Booth's multiplication algorithm is a multiplication algorithm that multiplies two signed binary numbers in two's complement notation. The algorithm was invented by Andrew Donald Booth..

#### 1) Booth Multiplier

Conventional array multipliers, like the Braun multiplier and Baugh Woolley multiplier achieve comparatively good performance but they require large area of silicon, unlike the add-shift algorithms, which require less hardware and exhibit poorer performance. The Booth multiplier makes use of Booth encoding algorithm in order to reduce the number of partial products by considering two bits of the multiplier at a time, thereby achieving a speed advantage over other multiplier architectures. This algorithm is valid for both signed and unsigned numbers. It accepts the number in 2's complement form.

### 2) Array Multiplier

Array multiplier is an efficient layout of a combinational multiplier. Multiplication of two binary number can be obtained with one micro-operation by using a combinational circuit that forms the product bit all at once thus making it a fast way of multiplying two numbers since only delay is the time for the signals to propagate through the gates that forms the multiplication array. In array multiplier, consider two binary numbers A and B, of m and n bits. There are mn summands that are produced in parallel by a set of mn AND gates. n x n multiplier requires n (n-2) full adders, n half-adders and n2 AND gates. Also, in array multiplier worst case delay would be (2n+1) td.

## B. Booth Multiplication(Proposed work)

Booth's algorithm involves repeatedly adding one of two predetermined values *A* and *S* to a product *P*, then performing a rightward arithmetic shift on *P*. Let m and r be the multiplicand and multiplier, respectively; and let *x* and *y* represent the number of bits in m and r.

Determine the values of *A* and *S*, and the initial value of *P*. All of these numbers should have a length equal to (*x* + *y* + 1). Fill the most significant (leftmost) bits with the value of m. Fill the remaining (*y* + 1) bits with zeros.

1) Fill the most significant bits with the value of (−m) in two's complement notation. Fill the remaining (*y* + 1) bits with zeros.
2) P: Fill the most significant *x* bits with zeros. To the right of this, append the value of r. Fill the least significant (rightmost) bit with a zero.
3) Determine the two least significant (rightmost) bits of *P*.
    a. If they are 01, find the value of *P* + *A*. Ignore any overflow.
    b. If they are 10, find the value of *P* + *S*. Ignore any overflow.
    c. If they are 00, do nothing. Use *P* directly in the next step.
    d. If they are 11, do nothing. Use *P* directly in the next step.
4) Arithmetically shift the value obtained in the 2nd step by a single place to the right. Let *P* now equal this new value.
5) Repeat steps 2 and 3 until they have been done *y* times.
6) Drop the least significant (rightmost) bit from *P*. This is the product of m and r.

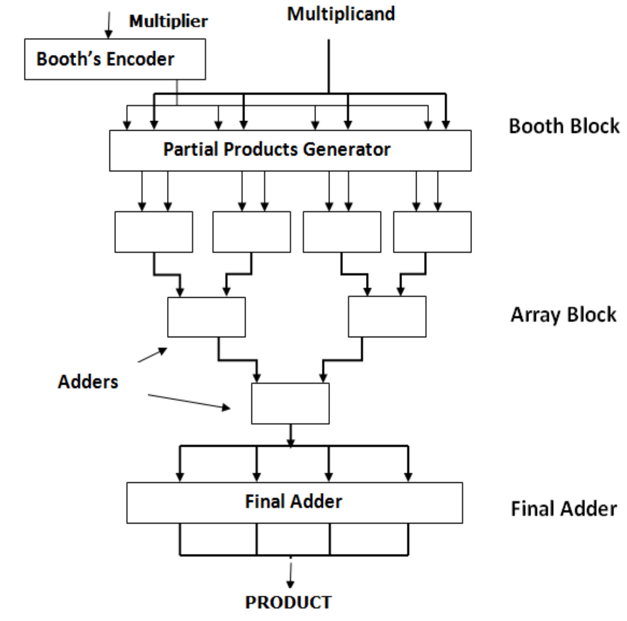


Figure 2: Block Diagram of the Multiplier

## C. Double Precision Booth Multiplication

Let's suppose a multiplication of 2 floating-point numbers A and B, where A=-18.0 and B=9.5

Binary representation of the operands:

A = -10010.0

B = +1001.1

Normalized representation of the operands:

A = -1.001x2$^4$

B = +1.0011x2$^3$

IEEE representation of the operands:

A = 110000000011001000000000000000000000000000 0000000000000000000000

B = 010000000010001100000000000000000000000000 0000000000000000000000

Multiplication of the mantissas:

•we must extract the mantissas, adding an1 as most significant bit, for normalization

A= 100100000000000000000000000000000000000000 00000000000

B= 100110000000000000000000000000000000000000 00000000000

the 106-bit result of the multiplication is: 0x558000000000

only the most significant bits are useful: after normalization (elimination of the most significant 1), we get the 52-bit mantissa of the result. This normalization can lead to a correction of the result's exponent

In our case, we get:

01
**0101011000000000000000000000000000000000000000000000**
000000000000000000000000000000000000000000000000000000
Addition of the exponents:
exponent of the result is equal to the sum of the operands exponents. A 1
can be added if needed by the normalization of the mantissas multiplication (this is not the case in our example)

As the exponent fields (Ea and Eb) are biased, the bias must be removed in order to do the addition. And then, we must to add again the bias, to get the value to be entered into the exponent field of the result (Er):

Er = (Ea-1023) + (Eb-1023) + 1023
= Ea + Eb – 1023
In our example, we have:
Ea 10000000011
Eb 10000000010
-1023 10000000001
Er 10000000110
what is actually 7, the exponent of the result
Calculation of the sign of the result:
The sign of the result (Sr) is given by the exclusive-or of the operands signs
(Sa and Sb):
Sr = Sa ⊕ Sb
In our example, we get:
Sr = 1 ⊕ 0 = 1
i.e. a negative sign
Composition of the result: the setting of the 3 intermediate results (sign, exponent and
mantissa) gives us the final result of our multiplication:
**1 10000000110**
**0101011000000000000000000000000000000000000000000000**
AxB = -18.0x9.5 = -1.0101011 x $2^{1030-1023}$ = -10101011.0 = $-171.0_{10}$

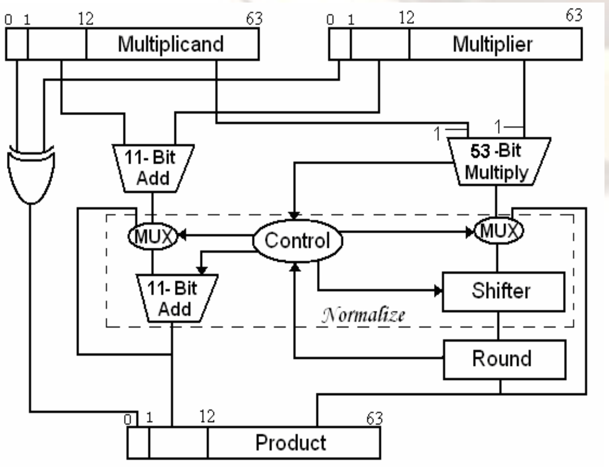


Figure 3: Schematic Diagram of Double Precision Multiplier

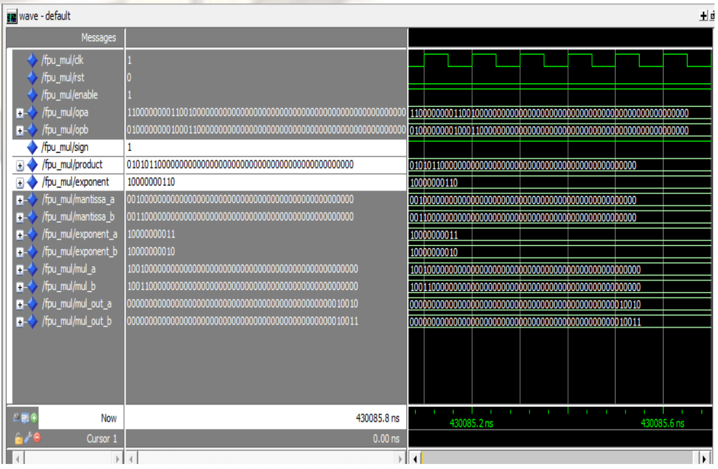
## IV. SYNTHESIS RESULTS

This design has been implemented, simulated on ModelSim and synthesized for VHDL. The HDL code uses VHDL 2001 constructs that provide certain benefits over the VHDL 95 standard in terms of scalability and code reusability. Simulation based verification is one of the methods for functional verification of a design. In this method, test inputs are provided using standard test benches. The test bench forms the top module that instantiates other modules. Simulation based verification ensures that the design is functionally correct when tested with a given set of inputs. Though it is not fully complete, by picking a random set of inputs as well as corner cases, simulation based verification can still yield reasonably good results.

| Design | Number Of Slices | Adders/Subtractors | 4 input LUTs | Bonded IOBs | Power |
|---|---|---|---|---|---|
| Array Multiplier | 97 | 20 | 169 | 16 | 134mW |
| Booth Multiplier | 29 | 7 | 55 | 16 | 124mW |

Table 1. The comparison of Multipliers

The following snapshots are taken from ModelSim after the timing simulation of the floating point multiplier core.
Consider the inputs to the floating point multiplier are:
1) A = -18.0 = -1.001x$2^4$
A = 1 10000011
0010000000000000000000000000000000000000000000000000 = 0x40F00000
B = 9.5 = 1.0011x$2^3$
B = 0 10000010
0011000000000000000000000000000000000000000000000000 = 0x41780000
AxB = -171.0 = -1.0101011 x$2^7$
AxB = 1 10000110
0101011000000000000000000000000000000000000000000000 = 0x42E88000



```
wave - default
Messages
/fpu_mul/clk            1
/fpu_mul/rst            0
/fpu_mul/enable         1
/fpu_mul/opa            1100000000110010000000000000000000000000000000000000000000000000
/fpu_mul/opb            0100000000100011000000000000000000000000000000000000000000000000
/fpu_mul/sign           1
/fpu_mul/product        0101011000000000000000000000000000000000000000000000
/fpu_mul/exponent       10000000110
/fpu_mul/mantissa_a     0010000000000000000000000000000000000000000000000000
/fpu_mul/mantissa_b     0011000000000000000000000000000000000000000000000000
/fpu_mul/exponent_a     10000000011
/fpu_mul/exponent_b     10000000010
/fpu_mul/mul_a          1001000000000000000000000000000000000000000000000000
/fpu_mul/mul_b          1001100000000000000000000000000000000000000000000000
/fpu_mul/mul_out_a      0000000000000000000000000000000000000000000000010010
/fpu_mul/mul_out_b      0000000000000000000000000000000000000000000000010011
Now                     430085.8 ns
Cursor 1                0.00 ns
430085.2 ns   430085.6 ns
```

Figure 4: Simulation of Double Precision Multiplier

2) A = 134.0625 = $1.00001100001 \times 2^7$

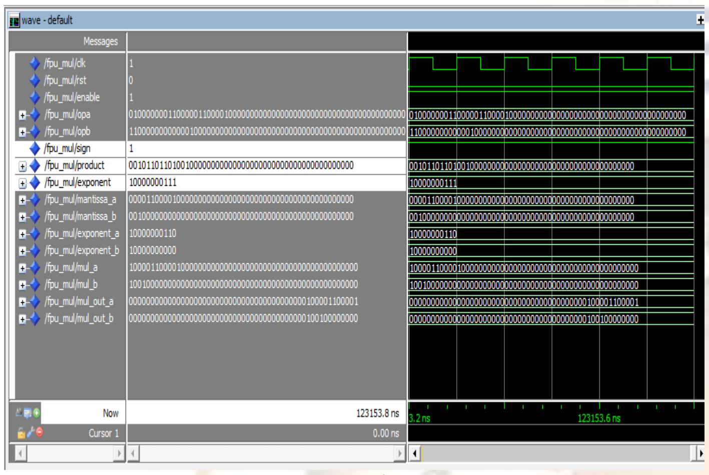
A = 0100000001100000110000100000000000000000000000000000000000000000

B = -2.25 = $-1.001 \times 2^1$

B = 1100000000000001000000000000000000000000000000000000000000000000

AxB = -301.640625 = $-1.00101101101001 \times 2^8$

AxB = 1 10000000111 0010110110100100000000000000000000000000000000000000



```
wave - default
Messages
/fpu_mul/clk          1
/fpu_mul/rst          0
/fpu_mul/enable       1
/fpu_mul/opa          0100000001100000110000100000000000000000000000000000000000000000
/fpu_mul/opb          1100000000000001000000000000000000000000000000000000000000000000
/fpu_mul/sign         1
/fpu_mul/product      0010110110100100000000000000000000000000000000000000
/fpu_mul/exponent     10000000111
/fpu_mul/mantissa_a   0000110000100000000000000000000000000000000000000000
/fpu_mul/mantissa_b   0010000000000000000000000000000000000000000000000000
/fpu_mul/exponent_a   10000000110
/fpu_mul/exponent_b   10000000000
/fpu_mul/mul_a        10000110000100000000000000000000000000000000000000000
/fpu_mul/mul_b        10010000000000000000000000000000000000000000000000000
/fpu_mul/mul_out_a    00000000000000000000000000000000000000000100001100001
/fpu_mul/mul_out_b    00000000000000000000000000000000000000000100100000000
Now                   123153.8 ns
Cursor 1              0.00 ns
```

Figure 5: Simulation of Double Precision Multiplier

## V. CONCLUSION

In this study a floating point multiplier design which is capable of executing a double precision floating-point multiplication using booth algorithm. One of the important aspects of the presented design method is that it can be applicable to all kinds of floating-point multipliers. The presented design is compared with a ordinary floating point array multiplier via synthesis. The synthesis results showed that proposed design is more efficient than conventional multiplier and critical path increment is only one or two gate delay. Since modern floating-point multiplier designs have significantly larger area than the standard floating-point multiplier, the percentage of the extra hardware will be less for those units. The methods presented in this study will be used on the design of floating-point multiplier-adder circuits. Also, the future work will enhance the proposed designs to support all IEEE754 rounding modes.

## REFERENCES

[1] Sumit Vaidya and Deepak Dandekar, " Delay-Power Performance comparison of multipliers in VLSI circuit design",International Journal of Computer Networks & Communications (IJCNC), Vol.2, No.4, pg 47-55, July 2010.

[2] Soojin Kim and Kyeongsoon Cho, "Design of High-speed Modified Booth Multipliers Operating at GHz Ranges", World Academy of Science, Engineering and Technology 61, 2010.

[3] P. Assady, "A New Multiplication Algorithm Using High-Speed Counters",European Journal of Scientific Research ISSN 1450-216X, Vol.26 No.3 ,pp.362-368, 2009.

[4] Hasan Krad and Aws Yousif Al-Taie, **"**Performance Analysis of a 32-Bit Multiplier with a Carry-Look-Ahead Adder and a 32-bit Multiplier with a Ripple Adder using VHDL", Journal of Computer Science 4 , ISSN 1549-3636,pp. 305-308, 2008.

[5] Cho, J. Hong, and G Choi, "54x54-bit Radix-4 Multiplier based on Modified Booth Algorithm," 13th ACM Symp.VLSI, pp 233-236, Apr. 2003.

[6] N. Besli, R. G. DeshMukh, "A 54*54-bit Multiplier with a new Redundant Booth's Encoding," IEEE Conf. Electrical and Computer Engineering, vol. 2, pp 597-602, 12-15 May 2002.

[7] Michael L. Overton, "Numerical Computing with IEEE Floating Point Arithmetic," Published by Society for Industrial and Applied Mathematics, 2001.

[8] B. Parhami, "Computer Arithmetic: Algorithms and Hardware Designs", Oxford University Press, 2000.

[9] N. Itoh, Y. Naemura, H. Makino, Y. Nakase, "A Compact 54*54-bit With Improved Wallace-Tree Structure," Dig. Technical Papers of Symp. VLSI Circuits, pp 15-16, Jun. 1999.

[10] R. K. Yu, G. B. Zyner, "167MHz Radix-4 Floating-Point Multiplier," IEEE Symp. on Computer Arithmetic, pp 149-154, Jul. 1995.

[11] D. Goldberg, "What every computer scientist should know about floating-point arithmetic" ,ACM Computing Surveys vol. 23-1 , pp. 5-48 ,1991.

[12] A. Goldovsky and et al., "Design and Implementation of 16 by 16 Low-Power Two's Complement Multiplier. In *Design and Implementation of Adder/Subtractor and Multiplication Units for Floating-Point Arithmetic IEEE International Symposium on Circuits and Systems*, **5**, pp 345–348, 2000.

[13] P. Seidel, L. McFearin, and D. Matula. "Binary Multiplication Radix-32 and Radix-256", *In 15th Symp. On Computer Arithmetic*, pp 23–32, 2001.

[14] U. Kulisch, "*Advanced Arithmetic for the Digital Computers*", Springer- Verlag, Vienna, 2002.

[15] M. Schulte, E. Swartzlander, "Hardware Design and ArithmeticAlgorithms for a

Variable-Precision, Interval Arithmetic Coprocessor", *Proc. IEEE 12th Symposium on Computer Arithmetic (ARITH-12)*, pp 222-230, 1995.

[16] R. V. K. Pillai, D. A1 - IShalili and A. J. A1 - Khalili, "A Low Power Approach to Floating Point Adder Design", *in Proceedings of the I997 International Conference on Computer Design*, pp. 178-185.

[17] Kari Kalliojarvi and Yrjo Neovo, "Distribution of Roundoff Noise in Binary Floating - Point Addition", *Proceedings of ISCAS92*, pp. 1796-1799.

[18] Wakerly, John F., "Digital Design – Principles and Practices", Tata McGraw Hill Series.

[19] Flores, Ivan, The *Logic of Computer Arithmetic*, Prentice Hall, Inc. Englewood Cliff (N.J.).

[20] G. Todorov, BASIC Design, Implementation and Analysis of a Scalable High-radix Montgomery Multiplier," Master Thesis, Oregon State University, USA, December 2000.

[21] L. A. Tawalbeh, "Radix-4 ASIC Design of a Scalable Montgomery Modular Multiplier using Encoding Techniques," M.S. Thesis, Oregon State University, USA, October 2002.

[22] W. Gallagher and E. Swartzlander, "High Radix Booth Multipliers Using Reduced Area Adder Trees", *In Twenty- Eighth Asilomar Conference on Signals, Systemsand Computers*, **1**, PP 545–549, 1994.

[23] B. Cherkauer and E. Friedman, "A Hybrid Radix-4/Radix- 8 Low Power, High Speed Multiplier Architecture for Wide Bit Widths", *In IEEE Intl. Symp. onCircuits and Systems*, **4**, pp 53–56, 1996.

[24] Israel Koren, Computer Arithmetic Algorithms, Prentice Hall, Inc. Englewood Cliff (N.J.) 1993.

[25] Nabeel Shirazi, A1 Walters, and Peter Athanas. "Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines". *In IEEE Symposium on FPGAs for Custom Computing Machines*, pages 155-162, April 1995.

[26] Y.Wang, Y. Jiang, and E. Sha, "On Area-Efficient Low Power Array Multipliers", In *The 8th IEEE International Conference on Electronics, Circuits and Systems*, pp 1429– 1432, 2001.

[27] Computer Architecture: A Quantitative Approach, D.A. Patterson and J.L. Hennessy, Morgan Kaufman, San Mateo, C.A. 1996, Appendix A.

[28] K. Yano and et al. A 3.8-ns CMOS 16 _ 16-b Multiplier Using Complementary Pass-Transistor Logic. *Journal of Solid-State Circuits*, 25:388–395, 1990.

[29] I. Khater, A. Bellaouar, and M. Elmasry, "Circuit Techniques for CMOS Low-Power High-Performance Multipliers", *IEEE Journal of Solid-State Circuits*, **31**:1535–1546, 1996.

[30] G.Bewick, "*Fast Multiplication: Algorithms and Implementation*", PhD. Thesis, Stanford University, 1992