An Efficient Implementation of High Speed Modified Booth Encoder for Floating Point Signed & Unsigned Numbers

P.V.Krishna Mohan Gupta[1] Ch.S.V.Maruthi Rao[2], G.R. Padmini[3],
[1]M.Tech (DSCE), Sreyas Institutite of Engineering & Technology, Hyderabad,
[2]Associate Professor, Sreyas Institutite of Engineering & Technology, Hyderabad,
[3]Associate Professor, Vasavi College of Engineering, Hyderabad,

***Abstract*---Multiplication is an important fundamental function in arithmetic operations. It can be performed with the help of different multipliers using different techniques. In this paper we focus on an efficient implementation of an IEEE 754 single precision floating point multiplier with signed and unsigned numbers. The multiplier implementation in floating point multiplication is done by Modified Booth Encoding (MBE) multiplier to reduce the partial products by half. The multiplier takes care of overflow and underflow cases. Rounding is not implemented to give more precision when using the multiplier in a Multiply and Accumulate (MAC) unit. By using MBE multiplier we increases the speed of multiplication, reduces the power dissipation and cost of a system. The proposed multiplier will be designed and verified using Modelsim with Verilog HDL. Xilinx is used for synthesis.**

***Keywords*-floating point multiplication, Array Multiplier, MBE, Partial Products, overflow and underflow**

I. INTRODUCTION

Today the main applications of floating point numbers are in the field of medical imaging, biometrics, motion capture and audio applications. Since multiplication dominates the execution time of most DSP algorithms, 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.

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 [1].

This paper presents a new floating point multiplier which operates on single-precision floating point. The conventional floating point multipliers use Array multiplier for Mantissa multiplication [2]. In this paper we presents a Modified Booth Encoder (MBE) multiplier [3][4] for Significand or Mantissa multiplication, which reduces the partial products by half, to achieve speed of operation and reduces the time delay[6].Normal MBE multiplier having Carry Save Adder(CSA) for partial product addition[5], here in this floating point multiplier we proposed Ripple Carry Adder(RCA) to reduce the complexity of the circuit and complicated more number of bit addition. This paper is organized as follows: In section I (B) mainly concentrated on Single Precision floating point numbers, section II focuses on floating point algorithm, section III concentrate on different multiplication methods[7][8], section IV looking for hardware of floating point multiplier(Block diagram), section V contains the exponent result on overflow or underflow and final sectional explains synthesis results and conclusions of this project

A. *Floating Point Arithmetic*

The IEEE 754 [1] standard is the most widely used standard for floating point computation, and is followed by many CPU implementation. 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. IEEE 754[1] 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).

B. *Single Precision Floating point Numbers*

Thus, a total of 32 bits is needed for single-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 127 for an 8-bit exponent of the single precision format. The addition of bias allows the use of an exponent in the range from -126 to +127. The single precision format offers a range from 2^{-126} to2^{+127}. 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 significand[1]. 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)

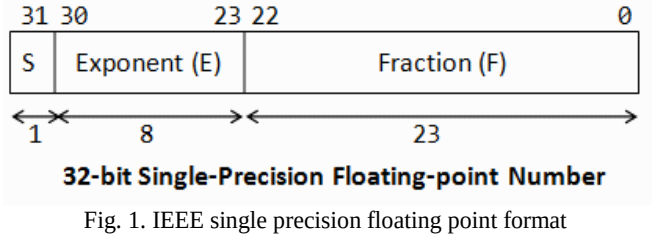

Fig. 1. IEEE single precision floating point format

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

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.

[1]Significand is the mantissa with an extra MSB bit.

C. *Floating Point Multiplication*

Multiplication of two numbers in floating point format is done by following steps: 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), detailed explanation is given in next section.

II. FLOATING POINT MULTIPLICATION ALGORITHM

Floating point multiplication algorithm is shown in fig (2) in the below flowchart.

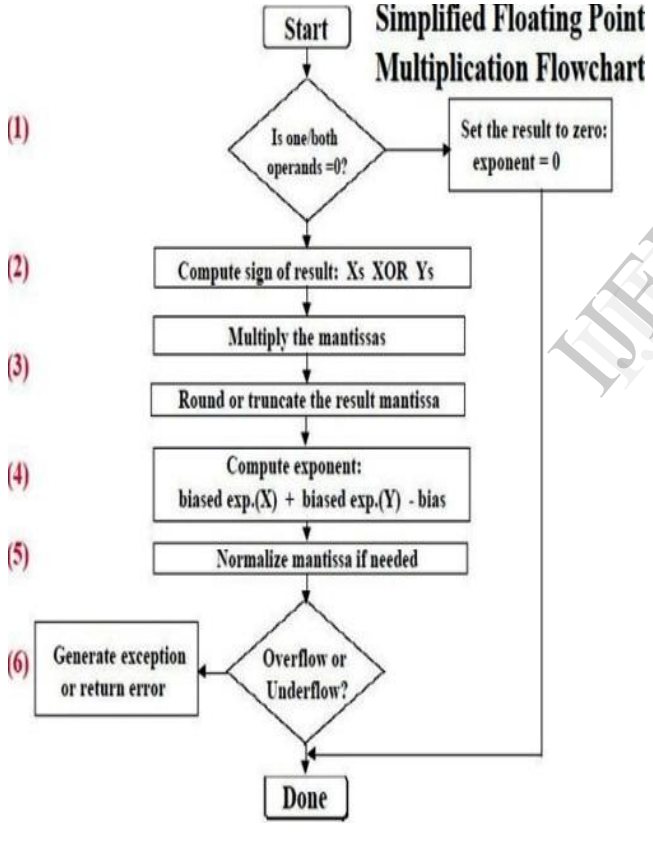

Fig 2. IEEE single precision floating point format

As stated in the introduction, normalized floating point numbers have the form of $Z= (-1^{S}) * 2^{(E - Bias)} * (1.M)$. To multiply 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 of the 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 IEEE 754 single precision floating point format, but with a reduced number 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
  × 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
```

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

```
1.10000110  10.01011000
```

2. 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.

   a. `10000110 10.01011000` (before normalizing)
   b. `10000111 1.001011000` (normalized)

The result is (without the hidden bit):

```
1 10000111 00101100
```

3. 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

In this paper we present a floating point multiplier in which rounding support isn't implemented. Rounding support can be added as a separate unit that can be accessed by the multiplier or by a floating point adder, thus accommodating for more precision if the multiplier is connected directly to an adder in a MAC unit. Fig. 3 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 24 bit numbers and results in a 48 bit product, which we will call the intermediate product (IP). The IP is represented as (47 downto 0) and the decimal point is located between bits 46 and 45 in the IP.

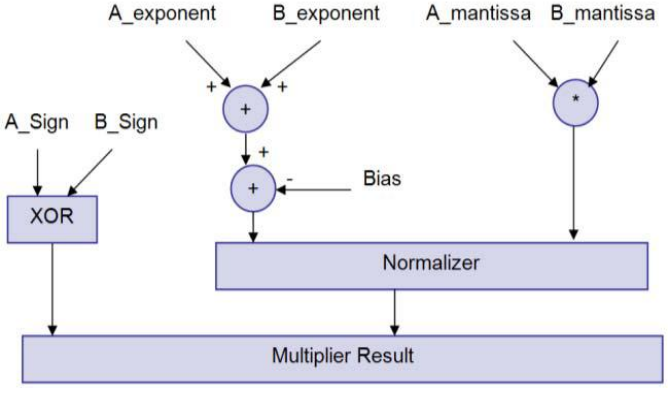

Fig. 3 Floating point multiplier block diagram

III. 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

a) 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 tow 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 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 [4].

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, there by 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. The Modified Booth encoder Floating point multiplier architecture is identified with the following blocks as shown in fig (3). The following sections detail each block of the floating point multiplier.

IV. Hardware Of Floating Point Multiplier

The hardware of floating point multiplier is mainly divided into a) sign bit calculation, b) exponent addition, c) significand multiplication and d) normalize unit.

A. Sign bit calculation

Multiplying two numbers results in a negative sign number if one of the multiplied numbers is of a negative value. By the aid of a truth table we find that this can be obtained by XORing the sign of two inputs.

B. Unsigned Adder (for exponent addition)

This unsigned adder is responsible for adding the exponent of the first input to the exponent of the second input and subtracting the Bias (127) from the addition result (i.e. A_exponent + B_exponent - Bias). The result of this stage is called the intermediate exponent. The add operation is done on 8 bits, and there is no need for a quick result because most of the calculation time is spent in the significand multiplication process (multiplying 24 bits by 24 bits); thus we need a moderate exponent adder and a fast significand multiplier.

An 8-bit ripple carry adder is used to add the two input exponents. As shown in Fig. 3 a ripple carry adder is a chain of cascaded full adders and one half adder; each full adder has three inputs (A, B, C_i) and two outputs (S, C_o). The carry out (C_o) of each adder is fed to the next full adder (i.e. each carry bit "ripples" to the next full adder).

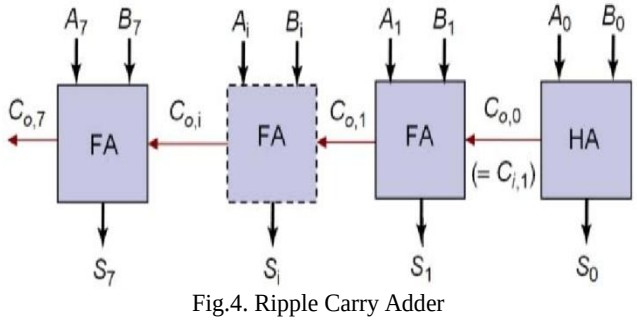

Fig.4. Ripple Carry Adder

The addition process produces an 8 bit sum (S_7 to S_0) and

a carry bit ($C_{o,7}$). These bits are concatenated to form a 9 bit addition result (S_8 to S_0) from which the Bias is subtracted. The Bias is subtracted using an array of ripple borrow subtractors.

Fig.5 shows the Bias subtractor which is a chain of 7 one subtractors (OS) followed by 2 zero subtractors (ZS); the borrow output of each subtractor is fed to the next subtractor. If an underflow occurs then $E_{result} < 0$ and the number is out of the IEEE 754 single precision normalized numbers range; in this case the output is signaled to 0 and an underflow flag is asserted.

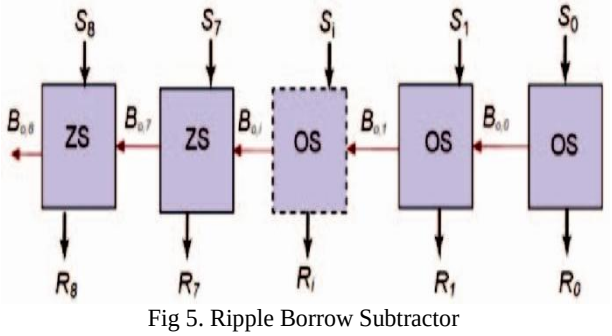

Fig 5. Ripple Borrow Subtractor

C. Unsigned Multiplier (for significand multiplication)

This unit is responsible for multiplying the unsigned significand and placing the decimal point in the multiplication product. The result of significand multiplication will be called the intermediate product (IP). The unsigned significand multiplication is done on 24 bit. Multiplier performance should be taken into consideration so as not to affect the whole multiplier's performance. The modified-Booth algorithm is extensively used for high-speed multiplier circuits. The multiplier here we are design and implement multiplier for signed and unsigned numbers using MBE technique. Table 1 shows the truth table of MBE scheme. From table 1 the MBE logic diagram is implemented as shown in Fig.5. Using the MBE logic and considering other conditions the Boolean expression for one bit partial product generator is given by the equation 2.

TABLE 1: Truth Table of MBE Scheme.

bi+1	b_i	*bi-1*	value	X1_a	X2_a	Z	Neg
0	0	0	0	1	0	1	0
0	0	1	1	0	1	1	0
0	1	0	1	0	1	0	0
0	1	1	2	1	0	0	0
1	0	0	-2	1	0	0	1
1	0	1	-1	0	1	0	1
1	1	0	-1	0	1	1	1
1	1	1	0	1	0	1	0

Equation 3 is implemented as shown in Fig.5. The SUMBE multiplier does not separately consider the encoder and the decoder logic, but instead implemented as a single unit called partial product generator as shown Fig 6.

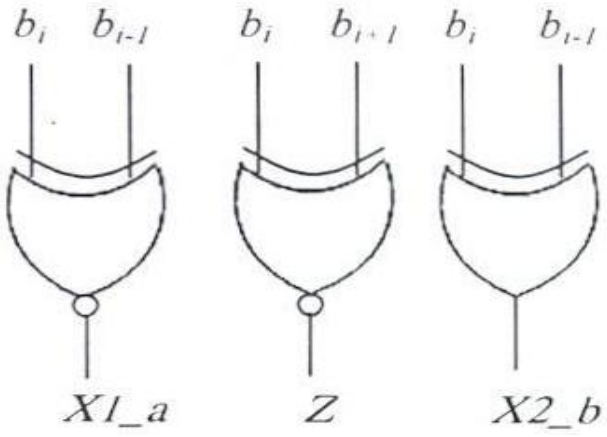

Fig. 6. Logic diagram of MBE.

TABLE 2: Shows the SUMBE multiplier operation.

Sign-unsign	Type of operation
0	Unsigned multiplication
1	Signed multiplication

Fig. shows the partial products generated by partial product generator circuit which is shown in Fig. 5. There are 5-partial products with sign extension and negate bit N_i. All the 5-partial products are generated in parallel.

$a_7\ a_6\ a_5\ a_4\ a_3\ a_2\ a_1\ a_0$

$b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$

$p_{08}p_{08}\ p_{08}\ p_{07}\ p_{06}\ p_{05}\ p_{04}\ p_{03}\ p_{02}\ p_{01}\ p_{00}$ X1

$1\ p_{18}p_{17}\ p_{16}\ p_{15}\ p_{14}\ p_{13}\ p_{12}\ p_{11}\ p_{10}\ N_0$ X2

$1\ p_{28}p_{27}\ p_{26}p_{25}\ p_{24}p_{23}\ p_{22}p_{21}\ p_{20}\ N_1$ X3

$1\ p_{38}\ p_{37}p_{36}p_{35}\ p_{34}\ p_{33}\ p_{32}\ p_{31}\ p_{30}\ N_2$ X4

$P_{47}p_{46}\ p_{45}\ p_{44}\ p_{43}\ p_{42}\ p_{41}p_{40}$ N3 X5

$p_{15}\ p_{14}\ p_{13}\ p_{12}\ p_{11}\ p_{10}\ p_9\ p_8\ p_7\ p_6\ p_5\ p_4\ p_3\ p_2\ p_1\ p_0$

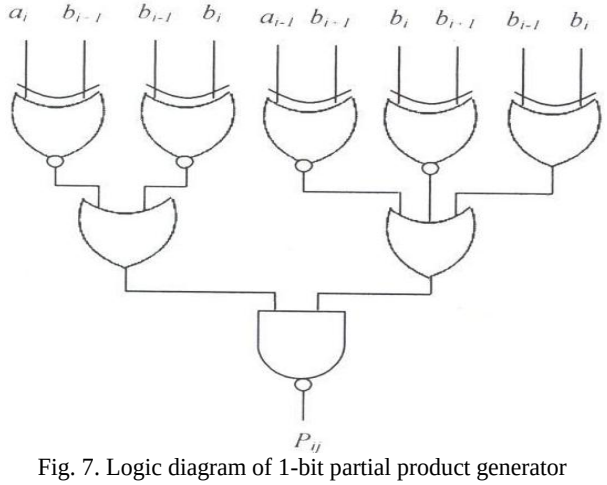

Fig. 7. Logic diagram of 1-bit partial product generator

D. Normalizer

The result of the significand multiplication (intermediate product) must be normalized to have a leading '1' just to the left 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. 8 shows a simplified logic of a Normalizer that has an 8 bit intermediate product input and a 6 bit intermediate exponent input.

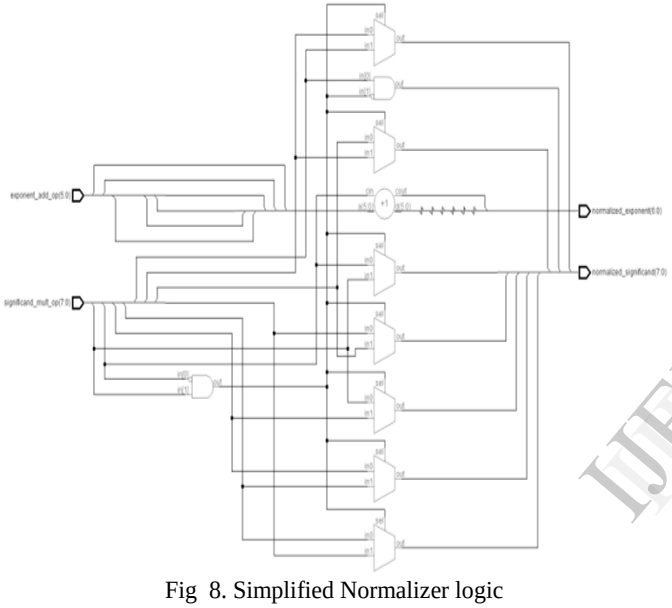

Fig 8. Simplified Normalizer logic

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 8 bits in size, and must be between 1 and 254 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 E_1 and E_2 are the exponents of the two numbers A and B respectively; the result's exponent is calculated by (6)

$$E_{result} = E_1 + E_2 - 127 \quad (2)$$

E_1 and E_2 can have the values from 1 to 254; resulting in E_{result} having values from -125 (2-127) to 381 (508-127); but for normalized numbers, E_{result} can only have the values from 1 to 254. Table III summarizes the E_{result} different values and the effect of normalization on it.

TABLE III. Normalization Effect On Result's Exponent And Overflow/Underflow Detection

Eresult	Category	Comments
$-125 \leq E_{result} < 0$	Underflow	Can't be compensated during normalization
Eresult = 0	Zero	May turn to normalized number during normalization (by adding 1 to it)
$1 < E_{result} < 254$	Normalized number	May result in overflow during normalization
255 ≤ Eresult	Overflow	Can't be compensated

VI. SYNTHESIS RESULTS

This design has been implemented using Modelsim and synthesized for Verilog HDL. The Verilog code is used for coding. Simulation based verification is one of the methods for functional verification of a design. 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.

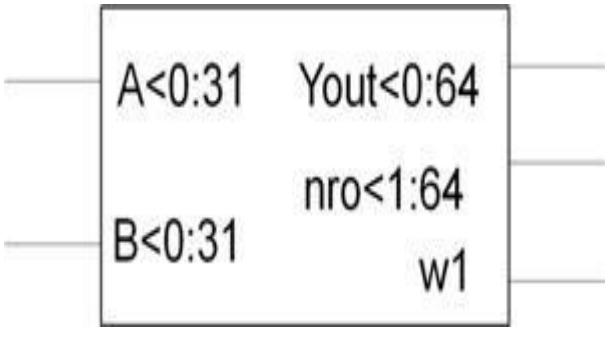

Fig 9. Floating point Multiplier RTL top module

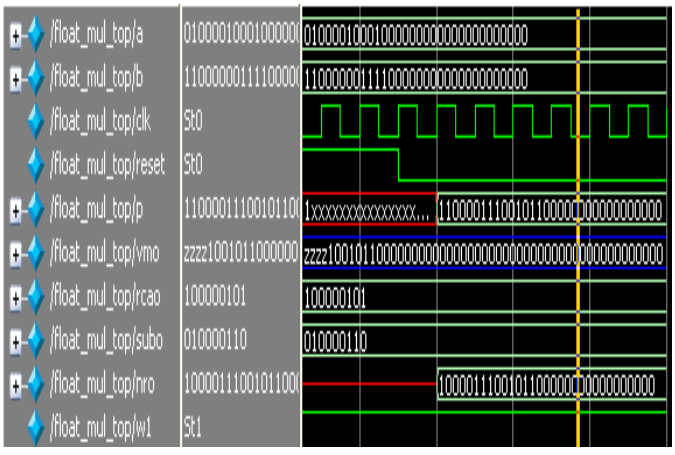

Fig 10. Floating point Multiplier output

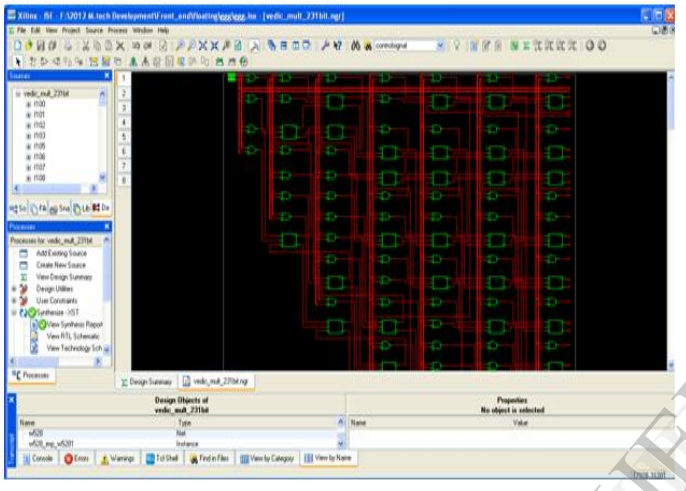

Fig 11. Floating point Multiplier synthesis results using Xilinx

Design	Number of slices	Adders/ Subtractors	4 input LUTs	Bonded IOBs	Delay
Array Multiplier	630	32	1147	96	96.08ns
Booth Multiplier	1582	26	2781	97	36.97ns

TABLE IV: Comparison of Multipliers.

From the above table, it is clearly found that the delay associated with MBE multiplier is almost 60% less than the delay of Array Multiplier.

VII. CONCLUSIONS AND FUTURE WORK

This paper presents an implementation of a floating point multiplier that supports the IEEE 754 binary interchange format; one of the important aspects of the presented design method is that it can be applicable to all kinds of floating-point multipliers. The present design is compared with an ordinary floating point array multiplier and modified Booth encoder multiplier via synthesis. It shows that Booth's floating point multiplier is faster than the array multiplier, by seeing the delay value we can known this factor and power dissipation is also less compare to array multiplier. The future work shows it can be implemented for double precision floating point multiplier also.

VIII. REFERENCES

[1] IEEE 754-2008, IEEE Standard for Floating-Point Arithmetic, 2008
[2] Mohamed Al-Ashrafy, Ashraf Salem, Wagdy Anis"An Efficient Implementation Floating Point Multiplier", IEEE 2011
[3] Ravindra P Rajput & M.N.Shanmukha Swamy "High speed Modified Booth multiplier for signed and unsigned numbers"IEEE 2012, 649-653
[4] A. D. Booth, "A signed binary multiplication technique," *Quart. J.Math.*, vol. IV, pp. 236–240, 1952.
[5] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Trans. Electron Comput.*, vol. EC-13, no. 1, pp. 14–17, Feb. 1964.
[6] Puneet Paruthi, Tanvi Kumar,Himanshu Singh, "Simulation of IEEE754 standard Double precision Multiplier using Booth Techniques" IJERAVol.2, Issue 5, Septmenber2012.
[7] A. R. Cooper, "Parallel architecture modified Booth multiplier," *Proc. Inst. Electrnoic. Eng. G*, vol. 135, pp. 125–128, 1988.
[8] F. Elguibaly, "A fast parallel multiplier–accumulator using the modified Booth algorithm," *IEEE Trans. Circuits Syst.*, vol. 27, no. 9, pp.902–908, Sep. 2000.
[9] J. Fadavi-Ardekani, "MxN Booth encoded multiplier generator using Optimized Wallace trees," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 1,
[10] Cho, J.Hong, and G Choi, "54x54-bit Radix-4 multiplier based on Modified Booth Algorithm",13th ACM Symp.VLSI,pp233-236,Apr.2003.
[11] 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.