International Journal of Latest Engineering Research and Applications (IJLERA) ISSN: 2455-7137

Volume – 02, Issue – 11, November – 2017, PP – 01-07

Design of high speed single precision floating point multiplier

Rajitha Kudipudi $_{\text{B.Tech;}}$ [1], G.A.Arun Kumar $_{\text{M.Tech;}}$ [2]

[1](Electronics & Communication Engineering, Aditya College of Engineering & Technology/ JNTUK Kakinada, India)

[2](Electronics & Communication Engineering, Aditya College of Engineering & Technology/ JNTUK Kakinada, India)

Abstract: A fast and energy-efficient floating point unit is always needed in electronics industry especially in DSP, image processing and as arithmetic unit in microprocessors. Many numerically intensive applications require rapid execution of arithmetic operations. Arithmetic circuits form an important class of circuits in digital systems. With the remarkable progress in the very large scale integration (VLSI) circuit technology, many complex circuits, unthinkable yesterday have become easily realizable today. Algorithms that seemed impossible to implement now have attractive implementation possibilities for the future. This means that not only the conventional computer arithmetic methods, but also the unconventional ones are worth investigation in new designs.

We present the design of an IEEE 754 single precision floating point multiplier. Verilog HDL is used to implement this design. Floating point numbers are one possible way of representing real numbers in binary format; the IEEE 754 standard presents two different floating point formats, Binary interchange format and Decimal interchange format. Multiplying floating point numbers is a critical requirement for applications involving large dynamic range. In this project we focus on single precision normalized binary interchange format. The multiplier design handles the overflow and underflow cases. Rounding is to give more precision when using the multiplier in a Multiply and Accumulate (MAC) unit.

Keywords: MAC: Multiply and Accumulate.

I. INTRODUCTION

Many applications require numbers that aren't integers. There are a number of ways that non-integers can be represented. Adding two such numbers can be done with an integer add, whereas multiplication requires some extra shifting. There are various way to represent the number systems. However, only one non-integer representation has gained widespread use, and that is floating point. In computing, floating point is a method of representing an approximation of a real_number in a way that can support a trade-off between range and precision. A number is, in general, represented approximately to a fixed number of significand digits (the significand) and scaled using an exponent; the base for the scaling is normally two, ten, or sixteen. A number that can be represented exactly is of the following form:

$$\text{significand} \times \text{base}^{\text{exponent}}.$$

For example:

$$1.2345 = \underbrace{12345}_{\text{significand}} \times \underbrace{10}_{\text{base}}{}^{\overbrace{-4}^{\text{exponent}}}$$

The term floating point refers to the fact that a number's radix point (decimal point, or, more commonly in computers, binary point) can "float"; that is, it can be placed anywhere relative to the significand digits of the number. This position is indicated as the exponent component, and thus the floating-point representation can be thought of as a kind of scientific notation.

II. FLOATING POINT MULTIPLICATION ALGORITHM:

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

1. Multiplying the significand; i.e. (1.M1*1.M2)
2. Placing the decimal point in the result
3. Adding the exponents; i.e. (E1 + E2 – *Bias*)
4. Obtaing the sign; i.e. s1 xor s2
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 '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. EA = EA-true + bias and EB = EB-true + bias

And

EA + EB = EA-true + EB-true + 2 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.

Fig. Below 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. The following sections detail each block of the floating point multiplier.

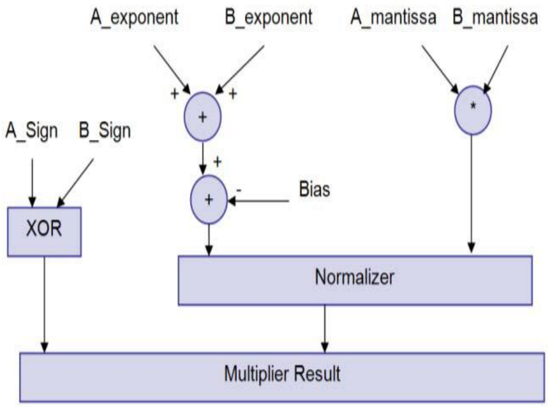


Figure2.1 Floating point multiplier block diagram

III. INDENTATIONS AND EQUATIONS

3.1 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 XOR ing the sign of two inputs.

3.1.2. 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, Ci) and two outputs (S, Co). The carry out (Co) of each adder is fed to the next full adder (i.e each carry bit "ripples" to the next full adder).

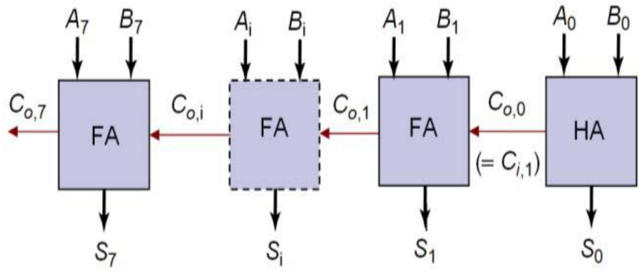


Figure 3.1. Ripple Carry Adder

The addition process produces an 8 bit sum (S7 to S0) and a carry bit (Co,7). These bits are concatenated to form a 9 bit addition result (S8 to S0) from which the Bias is subtracted. The Bias is subtracted using an array of ripple borrow subtractors. A normal subtractor has three inputs (minuend (S), subtrahend (T), Borrow in (Bi)) and two outputs (Difference (R), Borrow out (Bo)). The subtractor logic can be optimized if one of its inputs is a constant value which is our case, where the Bias is constant (127|10 = 001111111|2).

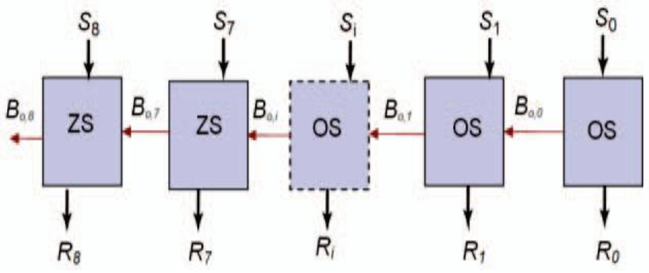


Figure 3.2 . Ripple Borrow Subtractor

3.1.3. 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. A 24x24 bit carry save multiplier architecture is used as it has a moderate speed with a simple architecture. In the carry save multiplier, the carry bits are passed diagonally downwards (i.e. the carry bit is propagated to the next stage). Partial products are made by ANDing the inputs together and passing them to the appropriate adder.

Carry save multiplier has three main stages:

1. The first stage is an array of half adders.
2. The middle stages are arrays of full adders. The number of middle stages is equal to the significand size minus two.
3. The last stage is an array of ripple carry adders. This stage is called the vector merging stage. The number of adders (Half adders and Full adders) in each stage is equal to the significand size minus one. For example, a 4x4 carry save multiplier is shown in Fig. 7 and it has the following stages:

- The first stage consists of three half adders.
- Two middle stages; each consists of three full adders.
- The vector merging stage consists of one half adder and two full adders.

The decimal point is between bits 45 and 46 in the significand multiplier result. The multiplication time taken by the carry save multiplier is determined by its critical path. The critical path starts at the AND gate of the first partial products (i.e. a1b0 and a0b1), passes through the carry logic of the first half adder and the carry logic of the first full adder of the middle stages, then passes through all the vector merging adders. The critical path is marked in bold in Fig

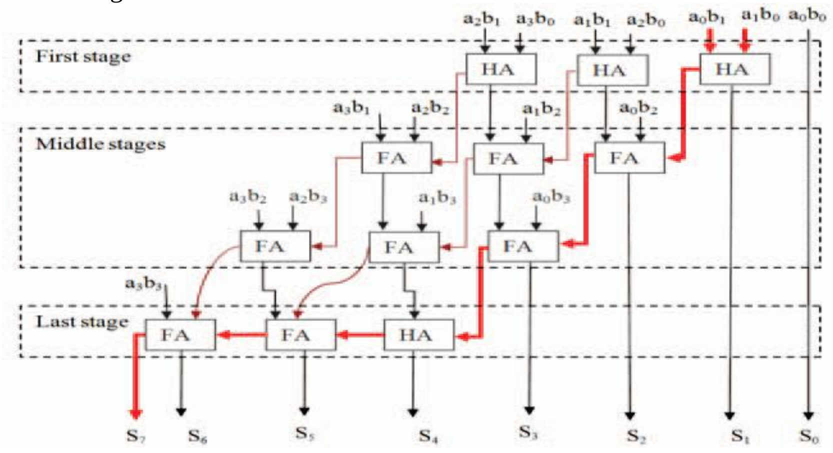

a_2b_1 a_3b_0 a_1b_1 a_2b_0 a_0b_1 a_1b_0 a_0b_0
First stage
HA
HA
HA
a_3b_1 a_2b_2 a_1b_2 a_0b_2
Middle stages
FA
FA
FA
a_3b_2 a_2b_3 a_1b_3 a_0b_3
FA
FA
FA
a_3b_3
Last stage
FA
FA
HA
S_7 S_6 S_5 S_4 S_3 S_2 S_1 S_0

Figure 3.3 4x4 bit Carry Save multiplier

In Fig.3.3:

1. Partial product: aibj = ai and bj
2. HA: half adder
3. FA: full adder

3.1.4. 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.

3.2. 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 E1 and E2 are the exponents of the two numbers A and B respectively; the result's exponent is calculated by

Eresult = E1 + E2 - 127

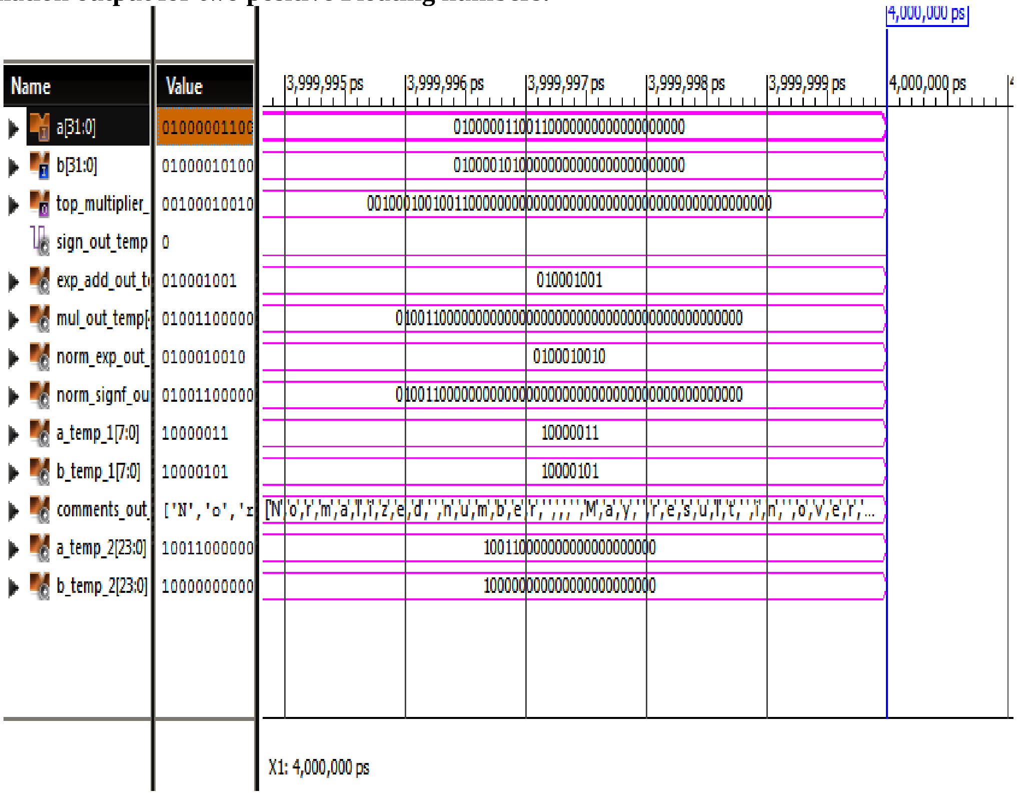
E1 and E2 can have the values from 1 to 254; resulting in Eresult having values from -125 (2-127) to 381 (508-127); but for normalized numbers, Eresult can only have the values from 1 to 254. Table III summarizes the Eresult different values and the effect of normalization on it.

E_{result}	Category	Comments
$-125 \le E_{result} < 0$	Underflow	Can't be compensated during normalization
$E_{result} = 0$	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
$255 \le E_{result}$	Overflow	Can't be compensated

Table 3.2.1 . Normalization Effect On Result's Exponent And Overflow/Underflow Detection

IV. FIGURES AND TABLES

4.1 Simulation output for two positive Floating numbers:



```
Name               Value          |3,999,995 ps |3,999,996 ps |3,999,997 ps |3,999,998 ps |3,999,999 ps |4,000,000 ps
a[31:0]            01000001100    01000001100110000000000000000000
b[31:0]            01000010100    01000010100000000000000000000000
top_multiplier_    00100010010    0010001001001100000000000000000000000000000000000000000000000000
sign_out_temp      0
exp_add_out_t      010001001      010001001
mul_out_temp[      01001100000    010011000000000000000000000000000000000000000000
norm_exp_out_      0100010010     0100010010
norm_signf_ou      01001100000    010011000000000000000000000000000000000000000000
a_temp_1[7:0]      10000011       10000011
b_temp_1[7:0]      10000101       10000101
comments_out_      ['N','o','r    ['N','o','r','m','a','l','i','z','e','d',' ','n','u','m','b','e','r',',',' ','M','a','y',' ','r','e','s','u','l','t',' ','i','n',' ','o','v','e','r',...
a_temp_2[23:0]     10011000000    100110000000000000000000
b_temp_2[23:0]     10000000000    100000000000000000000000

X1: 4,000,000 ps
```

Figure 4.1. simulation output for two positive Floating numbers

The simulation results for corresponding inputs are shown in Fig. The simulation is done using Modelsim 6.3f and for synthesis purpose Xilinx 14.3 software is used.
Considering the random floating point numbers,
Inputs: a = 19.2;
b = 66.6;
Output: result = 1278.72;
Here we have taken two positive 32-bit Floating point numbers.
A = 0 10000011 0011 = 19.2
B = 0 10000101 0000 = 66.6
The above figure shows the different result corresponding the two floating point multiplication regarding
Sign_out_temp: It represent the result of sign bit that is $0 \oplus 0 = 0$
Exp_add_out_temp: It represent the result of exponent bit that is

10000011+10000101 = 100001000

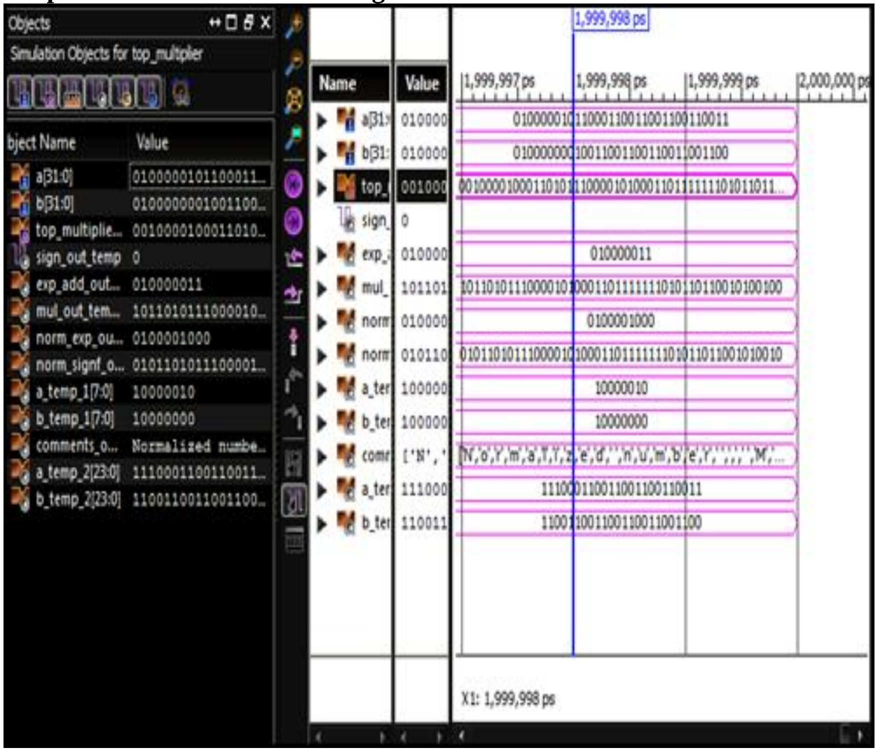
The exponent representing the two numbers is already shifted/biased by the bias value (127) and is not the true exponent; i.e. EA = EA-true + bias and EB = EB-true + bias And EA+ EB = EA-true + EB-true + 2 bias So we should subtract the bias from the resultant exponent otherwise the bias will be added twice.

$$\begin{array}{r} 100001000 \\ \underline{-\ 01111111} \\ 10001001 \end{array}$$

Mul_out_temp: It represent the result of Mantissa bit 1.0011 X 1.0000 = 100110000

Top_Multiplier_out_temp: It represents the overall the result of two floating point multiplier. That is a combination of Sign bit, Exponent and Mantissa result.
0 10001001 00110000000000000000000

4.2 simulation output for two Different Floating numbers:


Objects
Simulation Objects for top_multiplier

```
bject Name          Value
a[31:0]             0100000101100011...
b[31:0]             0100000001001100...
top_multiplie...    0010000100011010...
sign_out_temp       0
exp_add_out...      010000011
mul_out_tem...      1011010111000010...
norm_exp_ou...      0100001000
norm_signf_o...     0101101011100001...
a_temp_1[7:0]       10000010
b_temp_1[7:0]       10000000
comments_o...       Normalized numbe...
a_temp_2[23:0]      1110001100110011...
b_temp_2[23:0]      1100110011001100...
```

```
Name     Value    |1,999,997 ps   |1,999,998 ps   |1,999,999 ps   |2,000,000 ps
a[31:    010000   01000001011000110011001100110011
b[31:    010000   01000000010011001100110011001100
top_     001000   0010000100011010110000101000110111111101011011...
sign_    0
exp_     010000   010000011
mul_     101101   1011010111000010100011011111110101101100101001100
norm     010000   0100001000
norm     010110   0101101011100001010001101111111010110110010100 10
a_ter    100000   10000010
b_ter    100000   10000000
comr     ['N','   [N,o,r,m,a,l,i,z,e,d,' ',n,u,m,b,e,r,' ',',',' ',M,'...
a_ter    111000   111000110011001100110011
b_ter    110011   110011001100110011001100
```

X1: 1,999,998 ps

Figure 4.2 simulation output for two Different Floating numbers

The simulation results for corresponding inputs are shown in Fig. The simulation is done using Modelsim 6.3f and for synthesis purpose Xilinx 14.3 software is used.
Considering the different floating point numbers,

Inputs: a = -19.2;
b = 66.6;
Output: result = -1278.72;

Here we have taken two different 32-bit Floating point numbers.
A = 1 10000011 0011 = -19.2
B = 0 10000101 0000 = 66.6
The above figure shows the different result corresponding the two floating point multiplication regarding
Sign_out_temp: It represent the result of sign bit that is $1 \oplus 0 = 1$
Exp_add_out_temp: It represent the result of exponent bit that is

10000011+10000101 = 100001000

The exponent representing the two numbers is already shifted/biased by the bias value (127) and is not the true exponent; i.e. EA = EA-true + bias and EB = EB-true + bias And EA+ EB = EA-true + EB-true + 2 bias So we should subtract the bias from the resultant exponent otherwise the bias will be added twice.

$$\begin{array}{r} 100001000 \\ \underline{-\ 01111111} \\ 10001001 \end{array}$$

Mul_out_temp: It represent the result of Mantissa bit 1.0011 X 1.0000 = 100110000

Top_Multiplier_out_temp: It represents the overall the result of two floating point multiplier. That is a combination of Sign bit, Exponent and Mantissa result.

1 10001001 00110000000000000000000

V. CONCLUSION

Single Precision Floating Point Multiplier unit has been designed to fast adder and fast multipliers. IEEE 754 standard based floating point representation has been used. This is achived using Verilog HDL. Many numerically intensive applications require rapid execution of arithmetic operations. Arithmetic circuits form an important class of circuits in digital systems. With the remarkable progress in the very large scale integration (VLSI) circuit technology, many complex circuits, unthinkable yesterday have become easily realizable today. Algorithms that seemed impossible to implement now have attractive implementation possibilities for the future.

Acknowledgements

I would like to gratefully acknowledge our Head of the department of Electronics and Communication Engineering **Mr. R. V. V. KRISHNA M. Tech., (Ph.D)** has been abundantly helpful and has assisted me in numerous ways. I would like to express my sincere thanks to my Project guide **Mr. G. A. ARUNKUMAR M. Tech.** for his valuable guidance, best suggestions and constant encouragement right to the inception to the end of this report which is also enabled to complete the Project successfully in time.

REFERENCES

[1]. IEEE 754-2008, IEEE Standard for Floating-Point Arithmetic, 2008.
[2]. “Review on Floating Point Multiplier Using Vedic Mathematics” by Sneha Khobragade1 Mayur Dhait2 Value (2013): 6.14 | Impact Factor (2013): 4.438
[3]. “Review on floating point multiplier using ancient techniques” by Sushma S. Mahakalkar1 , Dr.S.L.Haridas2 e-ISSN: 2278-1676, p-ISSN: 2320-3331 PP 01-04 (ICAET-2014)
[4]. 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.
[5]. N. Shirazi, A. Walters, and P. Athanas, “Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines,” Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM’95), pp.155–162, 1995.
[6]. 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.