ISSN: 2278 – 1323
International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)
Volume 1, Issue 7, September 2012

Design and Implementation of Floating Point Multiplier for Better Timing Performance

B.Sreenivasa Ganesh[1],J.E.N.Abhilash[2], G. Rajesh Kumar[3]
SwarnandhraCollege of Engineering& Technology[1,2], Vishnu Institute of Technology[3]

Abstract*— IEEE Standard 754 floating point is the most common representation today for real numbers on computers. This paper gives a brief overview of IEEE floating point and its representation. This paper describes a single precision floating point multiplier for better timing performance. The main object of this paper is to reduce the power consumption and to increase the speed of execution by implementing certain algorithm for multiplying two floating point numbers. In order to design this VHDL is the hardware description language is used and targeted on a Xilinx Spartan-3 FPGA. The implementation's tradeoffs are area speed and accuracy. This multiplier also handles overflow and underflow cases. For high accuracy of the results normalization is also applied. By the use of pipelining process this multiplier is very good at speed and accuracy compared with the previous multipliers. This pipelined architecture is described in VHDL and is implemented on Xilinx Spartan 3 FPGA. Timing performance is measured with Xilinx Timing Analyzer. Timing performance is compared with standard multipliers.

***Index Terms*— Floating point,multiplication,VHDL,Spartan-3 FPGA, Pipelined architecture, Timing Analyzer, IEEE Standard 754**

I.INTRODUCTION

Because of the complexity of the algorithms, floating point operations are very hard to implement on FPGA. The computations for floating point operations involve large dynamic range, but the resources required for this operations is high compared with the integer operations. We have unsigned/signed multiplier for multiplication of binary numbers, for multiplication of floating point numbers floating point multiplier is used. There is a separate algorithm for this multiplication.

Unsigned Multiplier: A parallel-input parallel-output circuit (also known as array multiplier) that performs the operations depicted is shown in below figure.The circuit is combinationalbecause its output depends only on its currentinputs.As expected, it employs an array of AND gates plus full-adder units. Indeed,

$p_0 = a_0b_0$, $p_1 = a_0b_1 + a_1b_0$, $p_2 = a_0b_2 + a_1b_1 + a_2b_0 + carry(p_1)$, etc. This circuit operates only with positive (unsigned) inputs, also producing a positive (unsigned) output.

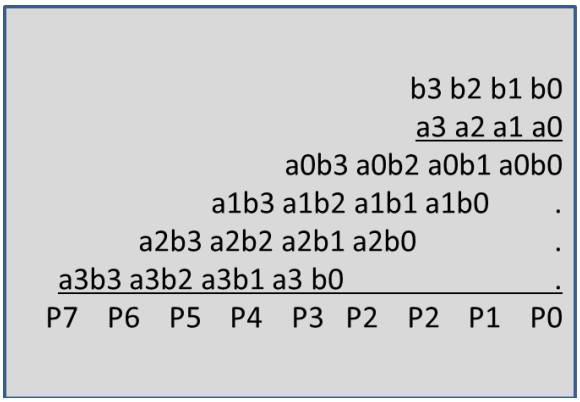

Fig 1 Binary Multiplication

This unsigned multiplier is useful in the multiplication of mantissa part of floating point numbers.

Floating point multiplier: This multiplier is mainly used to multiply two floating point numbers. Separate algorithm is essential for multiplication of these numbers. Here multiplication operation is simple than addition this is especially true if we are using a 32-bit format.

II. DESCRIPTION

Fixed point Format

A value of a fixed-point data type is essentially an integer that is scaled by a specific factor determined by the type. For example, the value 1.23 can be represented as 1230 in a fixed-point data type with scaling factor of 1/1000, and the value 1230000 can be represented as 1230 with a scaling factor of 1000. Unlike floating-point data types, the scaling factor is the same for all values of the same type, and does not change during the entire computation.

The scaling factor is usually a power of 10 (for human convenience) or a power of 2 (for

computational efficiency). However, other scaling factors may be used occasionally, e.g. a time value in hours may be represented as a fixed-point type with a scale factor of 1/3600 to obtain values with one-second accuracy.

The maximum value of a fixed-point type is simply the largest value that can be represented in the underlying integer type, multiplied by the scaling factor; and similarly for the minimum value. For example, consider a fixed-point type represented as a binary integer with *b* bits in two's complement format, with a scaling factor of $1/2^f$ (that is, the last *f* bits are fraction bits): the minimum representable value is -2^{b-1} the maximum value is $2^{b-1}-1/2^f$.

Floating point format

One of the way to represent real numbers in binary is the floating point formats. There are two different formats for the IEEE 754 standard . Binary interchange format and Decimal interchange format. In the multiplication of floating point numbers involves a large dynamic range which is useful in DSP applications. This paper concentrates only on single precision normalized binary interchange format. The below figure shows the IEEE 754 single precision binary format representation; consisting of a one bit sign (S), an eight bit exponent (E), and a twenty three bit fraction (M or Mantissa).

The term floating point refers to the fact that the radix point (decimal point, or, more commonly in computers, binary point) can "float"; that is, it can be placed anywhere relative to the significant digits of the number. This position is indicated separately in the internal representation, and floating-point representation can thus be thought of as a computer realization of scientific notation. Over the years, a variety of floating-point representations have been used in computers. However, since the 1990s, the most commonly encountered representation is that defined by the IEEE 754 Standard.

The advantage of floating-point representation over fixed-point and integer representation is that it can support a much wider range of values. For example, a fixed-point representation that has seven decimal digits with two decimal places can represent the numbers 12345.67, 123.45, 1.23 and so on, whereas a floating-point representation (such as the IEEE 754 decimal32 format) with seven decimal digits could in addition represent 1.234567, 123456.7, 0.00001234567, 1234567000000000, and so on. The floating-point format needs slightly more storage (to encode the position of the radix point), so when stored in the same space, floating-point numbers achieve their greater range at the expense of precision.

The speed of floating-point operations, commonly referred to in performance measurements as FLOPS, is an important machine characteristic, especially in software that performs large-scale mathematical calculations.

For ease of presentation and understanding, decimal radix with 7 digit precision will be used in the examples, as in the IEEE 754 decimal32 format. The fundamental principles are the same in any radix or precision, except that normalization is optional (it does not affect the numerical value of the result). Here, s denotes the significand and e denotes the exponent.

Addition and subtraction

A simple method to add floating-point numbers is to first represent them with the same exponent. In the example below, the second number is shifted right by three digits, and we then proceed with the usual addition method:

123456.7 = 1.234567 × 10^5
101.7654 = 1.017654 × 10^2 = 0.001017654 × 10^5

Hence:
123456.7 + 101.7654 = (1.234567 × 10^5) + (1.017654 × 10^2)
= (1.234567 × 10^5) + (0.001017654 × 10^5)
= (1.234567 + 0.001017654) × 10^5
= 1.235584654 × 10^5

To multiply, the significands are multiplied while the exponents are added, and the result is rounded and normalized.

e=3; s=4.734612
× e=5; s=5.417242
---
e=8; s=25.648538980104 (true product)
e=8; s=25.64854 (after rounding)
e=9; s=2.564854 (after normalization)

III. ALGORITHM FOR FLOATING POINT MULTIPLICATION

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.M1*1.M2)
2. Placing the decimal point in the result
3. Adding the exponents; i.e. (E1 + E2 – Bias)
4. Obtaining 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 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 alreadyshifted/biased by the bias value (127) and is not the trueexponent; 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 exponentotherwise 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.

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. 2 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.

IV. VHDL Implementation

To multiply floating-point numbers, the mantissas are first multiplied together with an unsigned integer multiplier. Then, the exponents are added, and the excess value (exponent_offset) $2^{(n-1)}$ is subtracted from the result. The sign of the output (s_out) is the XNOR of the signs of the inputs (SAand SB). After multiplication has taken place, the ostnormalizer normalizes the result, if necessary, by adjusting the mantissa and exponent of the result to ensure that the MSB of the mantissa is 1.

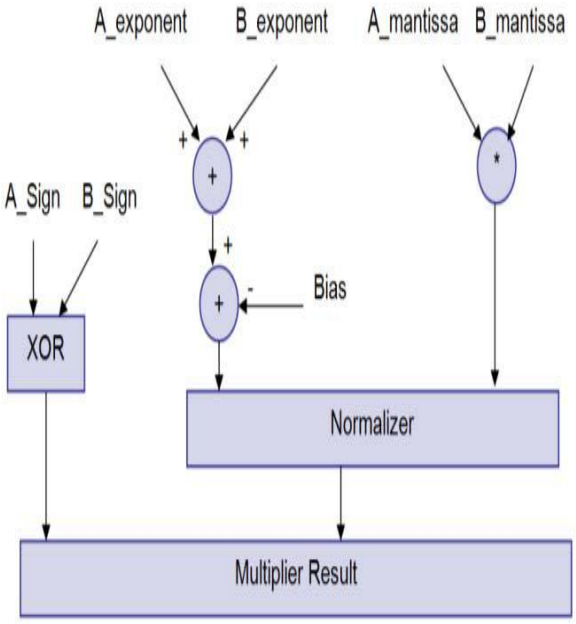

Fig 2 Floating Point Multiplier

Using the fp_mult function in a design produces a double precision output (i.e., the mantissa in the result has twice the number of bits of either input mantissa). Therefore, the result does not lose precision and does not require rounding.

The mantissa must be post-normalized whenever floating-point numbers are multiplied. As a double precision output is created, the implied denominator of the mantissa fraction is squared in value, from 8 to 16 bits.

The denominator of the mantissa fraction is 65536 (= 2^{16}) in double precision format, and 256 (= 2^8) in single precision. To retain as many significant digits as possible for consequential floating point operations, the result must be normalized.

To normalize the mantissa, the mantissa is shifted left (i.e., the mantissa is multiplied by powers of 2). For each bit shifted left, the exponent must be reduced by 1. The following example shows a number before normalization and its normalized equivalent:

Unnormalized number: 0.0001011001110001 X 2^{50}
Normalized equivalent: 0.1011001110001000 X 2^{47}

Table I shows an example of floating-point multiplication. The subscript "d" indicates that the number is a decimal number.

Table I Floating Point Multiplication

Operation	Decimal Equivalent (Exponent in Excess 0)	Binary (Exponent in Excess 64)
Multiplication	$(39 \times 2^{10d}) \times (203 \times 2^{6d})$	$(00100111.0 \times 2^{74}) \times (11001011.0 \times 2^{70})$
Normalization	$(0.609375 \times 2^{16d}) \times (0.79296875 \times 2^{14d})$	$(0.10011100 \times 2^{80}) \times (0.11001011 \times 2^{78})$
Result	7917×2^{16d}	$0.0001111011101101 \times 2^{80}$
Normalize	63336×2^{13d}	$0.1111011101101000 \times 2^{77}$
Decimal result	518,848,512	–

Parallel Adder:

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).

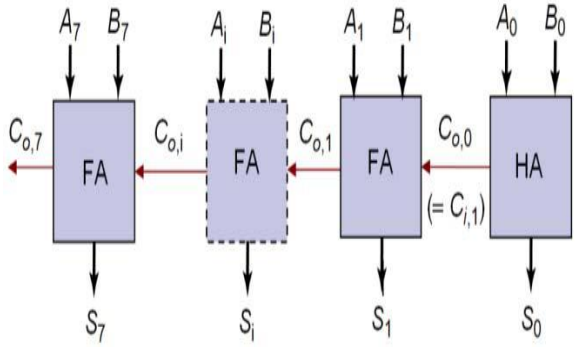

Fig 3 Ripple Carry Adder

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.

Sign Bit Generator:

Multiplying two numbers results in a negative sign number iff 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.

Normalizer:

More formally, the internal representation of a floating point number can be characterized in terms of the following parameters: The sign is either -1 or 1. The base or radix for exponentiation, an integer greater than 1. This is a constant for a particular representation. The exponent to which the base is raised. The upper and lower bounds of the exponent value are constants for a particular representation.

Sometimes, in the actual bits representing the floating point number, the exponent is biased by adding a constant to it, to make it always be represented as an unsigned quantity. This is only important if you have some reason to pick apart the bit fields making up the floating point number by hand, which is something for which the GNU library provides no support. So this is ignored in the discussion that follows. The mantissa or significand is an unsigned integer which is a part of each floating point number. The precision of the mantissa. If the base of the representation is b, then the precision is the number of base-b digits in the mantissa. This is a constant for a particular representation.

Many floating point representations have an implicit hidden bit in the mantissa. This is a bit which is present virtually in the mantissa, but not stored in memory because its value is always 1 in a normalized number. The precision figure (see above) includes any hidden bits.

Again, the GNU library provides no facilities for dealing with such low-level aspects of the representation. Floating point number is used to enhance the range of representation

The mantissa of a floating point number represents an implicit fraction whose denominator is the base raised to the power of the precision. Since the largest representable mantissa is one less than this denominator, the value of the fraction is always strictly less than 1. The mathematical value of a floating point number is then the product of this fraction, the sign, and the base raised to the exponent.

We say that the floating point number is normalized if the fraction is at least 1/b, where b is the base. In other words, the mantissa would be too large to fit if it were multiplied by the base. Non-normalized numbers are sometimes called denormal; they contain less precision than the representation normally can hold.

If the number is not normalized, then you can subtract 1 from the exponent while multiplying the mantissa by the base, and get another floating point number with the same value. Normalization consists of doing this repeatedly until the number is normalized. Two distinct normalized floating point numbers cannot be equal in value.

(There is an exception to this rule: if the mantissa is zero, it is considered normalized. Another exception happens on certain machines where the exponent is as small as the representation can hold. Then it is impossible to subtract 1 from the exponent, so a number may be normalized even if its fraction is less than 1/b.)

Underflow/Overflow:

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

$$E_{result} = E_1 + E_2 - 127$$

E1 and E2 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 I summarizes the E_{result} different values and the effect of normalization on it.

V. Experimental Results

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).

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.

Unsigned Multiplier for Mantessa:

A binary multiplier is an electronic circuit used in digital electronics, such as a computer, to multiply two binary numbers. It is built using binary adders.

A variety of computer arithmetic techniques can be used to implement a digital multiplier. Most techniques involve computing a set of partial products, and then summing the partial products together. This process is similar to the method taught to primary schoolchildren for conducting long multiplication on base-10 integers, but has been modified here for application to a base-2 (binary) numeral system.

The method taught in school for multiplying decimal numbers is based on calculating partial products, shifting them to the left and then adding them together. The most difficult part is to obtain the partial products, as that involves multiplying a long number by one digit (from 0 to 9):

```
    123
x 456
  =====
738  (this is 123 x 6)
  615   (this is 123 x 5, shifted one position to the left)
 + 492   (this is 123 x 4, shifted two positions to the left)
  =====
  56088
```

A binary computer does exactly the same, but with binary numbers. In binary encoding each long number is multiplied by one digit (either 0 or 1), and that is much easier than in decimal, as the product by 0 or 1 is just 0 or the same number. Therefore, the multiplication of two binary numbers comes down to calculating partial products (which are 0 or the first number), shifting them left, and then adding them together (a binary addition, of course):

```
     1011   (this is 11 in binary)
x 1110   (this is 14 in binary)
   ======
    0000   (this is 1011 x 0)
    1011    (this is 1011 x 1, shifted one position to the left)
   1011    (this is 1011 x 1, shifted two positions to the left)
 + 1011     (this is 1011 x 1, shifted three positions to the left)
 =========
  10011010   (this is 154 in binary)
```

This is much simpler than in the decimal system, as there is no table of multiplication to remember: just shifts and adds.

This method is mathematically correct and has the advantage that a small CPU may perform the multiplication by using the shift and add features of its arithmetic logic unit rather than a specialized circuit. The method is slow, however, as it involves many intermediate additions. These additions take a lot of time. Faster multipliers may be engineered in order to do fewer additions; a modern processor can multiply two 64-bit numbers with 16 additions (rather than 64), and can do several steps in parallel.

The second problem is that the basic school method handles the sign with a separate rule ("+ with + yields +", "+ with - yields -", etc.). Modern computers embed the sign of the number in the number itself, usually in the two's complement representation. That forces the multiplication process to be adapted to handle two's complement numbers, and that complicates the process a bit more. Similarly, processors that use ones' complement, sign-and-magnitude, IEEE-754 or other binary representations require specific adjustments to the multiplication process.

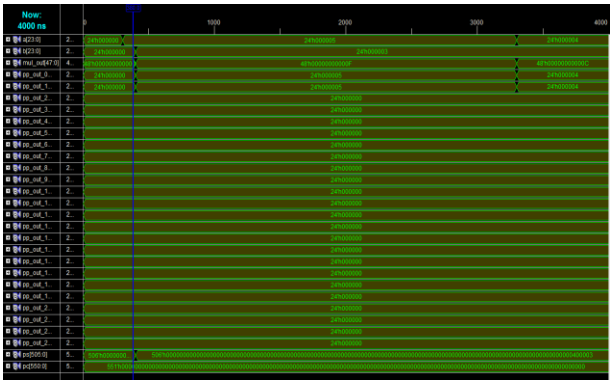

Fig 4 Simulation Results Showing Multiplication Process

The exponential addition involves adding two 8 bit digits producing a 9 bit result. The exponential addition was carried out by ripple carry adder as discussed before.The significand multiplier will take two 24 bit numbers and will produce 48 bit result. The significand multiplier module was designed using carry save multiplier architecture as discussed before.a and b are inputs which are forced with 32 bits each with respect to the given number the multiplied output result will be on the top multiplier result which is a 56 bit result. The remaining all are intermediate results such as sign bit, exponential addition o/p, Normalizer o/p are presented in.Figure 5.

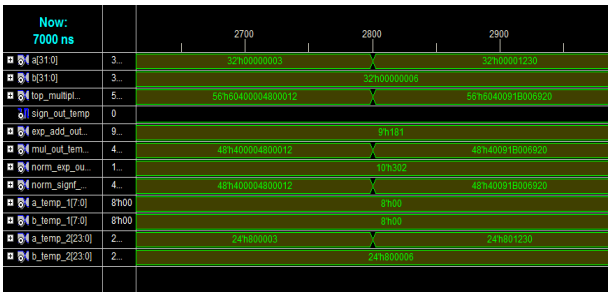

Fig 5: Simulation results for pipelined architecture

VI. Conclusion

This paper presents an implementation of a floating point multiplier with pipelined architecture. The multiplier just presents the

significandmultiplication result as is (48 bits); this gives better precision if the whole 48 bits are utilized in another unit; i.e. a floating point adder to form a MAC unit. The design has three pipelining stages. After simulation this design is implemented on a target device Xilinx Spartan 3 FPGA. Timing performance is observed with the tool Xilinx Timing Analyzer. Comparative results states that the proposed pipelined architecture is of high speed design. Proposed pipelined approach is 0.54ns faster compared with conventional approach. This advantage is well utilized in high speed DSP applications.

REFERENCES

[1] 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.

[2] John L. Hennessy and David **A.** Patterson. *Computer Architecture A Quantitative Approach, Second Edition.* Morgan Kaufmann, 1996.

[3] Richard I. Hartley and Keshab K. Parhi. Digit-Serial Computation. Kluwer Academic Publishers, Boston, MA, 1995

[4] A. El Gama1 et al., "An architecture for electrically configurable gate arrays," IEEE J. Solid State Circ., vol. 24, pp. 394398, 1989.

[5] C. Renard, "FPGA implementation of an IEEE Standard floating-point unit," Tech. Rep., Thayer School of Engineering, Dartmouth College, NH USA.

[6] Shirazi, N., Walters, A. and Athanas, P. "Quantitative analysis of floating-point arithmetic on FPGA based custom computing machines", Proc. IEEE symp. on FPGAs for Custom Comput. Machines, 1995, pp. 155-162.

[7] Styles, H. and Luk, W. "Customising graphics applications: techniques and programming interface", Proc. IEEE Symp. on Field-Programmable Custom Computing Machines, 2000, pp. 77-87.

[8] Ligon 111, W.B. et al. "A re-evaluation of the practicality of floating-point operations on FPGAs", Proc. IEEE Synzp. On FPGAs for Custom Comput. Machines, 1998, pp. 206-215.

[9] Louca, L. et al. "Implementation of IEEE single precision floating-point addition and multiplication on FPGAs", Proc. IEEE Symp. on FPGAs for Custom Comput. Machines, 1996

[10] Luk, W. and McKeever, S. "Pebble: a language for parametrised and reconfigurable hardware design", Field-Programmable Logic and Applications, LNCS 1482, Springer, 1998, pp. 9-18.

[11] Bruce Dawson, *Comparing floating point numbers*

[12] William Stallings, *Computer Organization and Architecture*, pp. 222-234 Macmillan Publishing Company, ISBN 0-02-415480-6

B. Sreenivasa Ganesh, persuingM.Tech in Swarnandhra College of Engineering and Technology, Narsapur, India. He has two years of teaching experience. His research interests are VLSI, Embedded Systems and Radar Engineering.

J. E. N. Abhilash, working as Associate Professor in Swarnandhra College of Engineering and Technology, Narsapur, India. He has seven years of teaching experience. He has guided many undergraduate and post graduate students. His research interests are VLSI, Embedded Systems and Imaga Processing.

G. Rajesh Kumar: Asst. Professor in Vishnu institute of technology, India. Has five years of teaching experience. Major working areas are Digital electronics, Embedded Systems and VLSI. Presented research papers in five international conferences and three national conferences.Published one research paper in an International Journal.Research Scholar in JNTUK.