DESIGN AND IMPLEMENTATION OF HIGH SPEED IEEE-754 DOUBLE-PRECISION FLOATING POINT MULTIPLIER

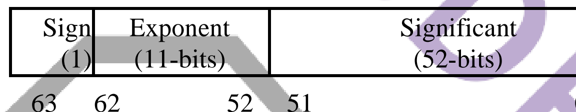
[1]DOWLAPALLI GUNASRI, [2]P.S.N.BHASKAR, [3]UNDRAKONDA JYOTHSNA

[1]M. Tech Student, [2,3]Assistant Professor
ECE Department,
Sankethika Vidya Parishad Engineering College, Visakhapatnam, A.P, India

Abstract: **In this paper we present an efficient implementation of an IEEE 754 double precision floating point multiplier using VHDL. Several enhancements are introduced to the design; the parallel proposed multiplier is implemented. As decimal multiplication is important in many commercial applications like financial analysis, banking, tax calculation, currency conversion, insurance, and accounting etc. this multiplier design adding several features including exponent generation, shifting of the intermediate product, rounding, and exception detection and handling. The core of the decimal multiplication algorithm is an iterative scheme of partial product accumulation employing decimal ripple carry addition to reduce the critical path delay. Area and delay estimates are provided for a verified VHDL register transfer level model of the multiplier. This new implementation is different in the sense that it accepts as a user parameter the operated size of the unit about to be synthesized and creates the requested unit. This feature makes our implementation a very convenient tool for rapid application prototyping. We also highlight the strengths and limitations of this approach in the creation of custom floating point units.**

1. INTRODUCTION:

A standard notation enables easy exchange of data between machines and simplifies hardware algorithms. IEEE 754 standard defines how floating point numbers are represented.



In IEEE 754 standards, the floating-point numbers is represented by three field as first field is a sign bit „S" (0 is positive and 1 is negative) this representation is called sign and magnitude. The second field is exponent „E" (signed, very large numbers have large positive exponents and very small close-to-zero numbers have negative exponents also More bits in exponent field increases range of values, the third field 'F' is the Fraction field (fraction after binary point and More bits in fraction field improves the precision of Floating point numbers.

Value of a floating-point number is
$(-1)^{Sign} \times (1.Significant) \times 2^{(exponent\ bias)}$.
where bias is equal to 1023.

2. FLOATING POINT MULTIPLICATION ALGORITHM:

Multiplying two numbers in floating point format is done by
1- Adding the exponent of the two numbers then subtracting the bias from their result.
2- Multiplying the significand of the two numbers.
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). As stated in the floating point format, normalized floating point numbers have the form of as
Value = (-1S) * 2 (E - Bias) * (1.M).

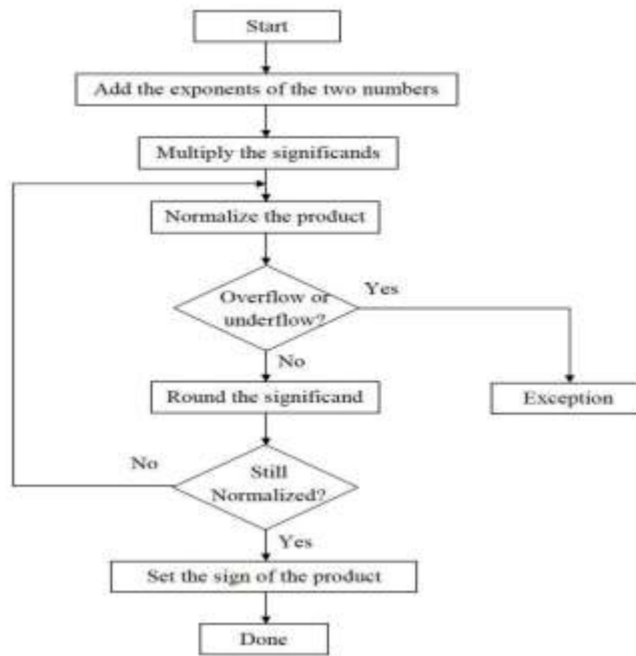


Fig 1: FP Multiplier Operational Flow Chart

To multiply two floating point numbers the following is done:

a. Multiply the significand (1.M1*1.M2).
b. Placing the decimal point in the result
c. Add the exponents; i.e. (E1+E2– Bias).
d. Obtaining the sign; i.e. S1 xor S2.
e. Normalizing the result; i.e. obtaining 1 at the MSB of the results significand.
f. Rounding the result to fit in the available bits.
g. Checking for underflow/overflow occurrence

Consider a floating point representation similar to the IEEE 754 double 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 100000000000 1010 = 16.33
B = 0 10000000011 0111 = 27.44
To multiply A and B

1. Multiply significand: 1.1010 X 1.0111

```
      11010
     11010
    11010
   00000
    11010
 ____________
      100010110
```

2. Place the decimal point: 1.00010110

3. Add exponents: 100000000000+10000000011=
1000000000011

The exponent representing the two numbers is already shifted/biased by the bias value (1023) 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.

```
1000000000011
- 011111111111
 1000000000100
```

4. Obtain the sign bit and put the result together: 1000000000100
1.00010110

5. There is a 1 at the radix point (decimal point). So no need to normalize the resultant
0 1000000000100 1.00010110
The result is (without the hidden bit):
0 1000000000100 00010110

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:
0 1000000000100 0001

We present a floating point multiplier in which rounding technique is 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. The multiplier structure; Exponents addition, Significand multiplication, and Result‘s sign calculation are independent shown in figure 2. The significand multiplication is done on two 52 bit numbers and results in a 104 bit product, which we will call the intermediate product (IP). The IP is represented as (103 downto 0). The following sections detail each block of the floating point multiplier.

3. HARDWARE OF FLOATING POINT MULTIPLIER

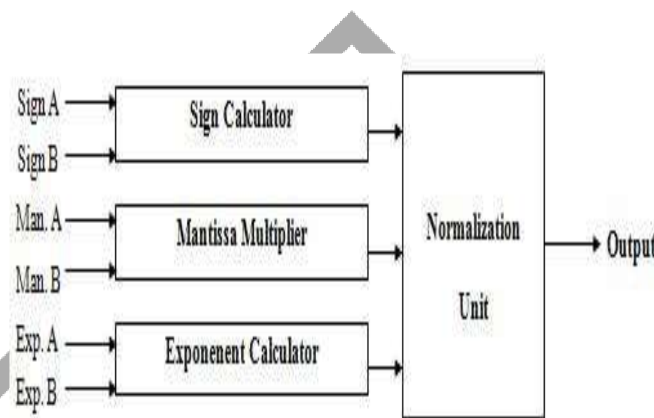


Fig 2: Floating Point Multiplier Block Diagram

A. Sign calculation:
The main component of Sign calculator is XOR gate. If any one of the numbers is negative then result will be negative. The result will be positive if two numbers are having same sign

B. Exponent Adder

This sub-block adds the exponents of the two floating point numbers and the Bias (1023) is subtracted from the result to get true result i.e. EA + EB – bias. To perform addition of two 11-bit exponents, an 11-bit ripple carry adder (RCA) is used. The Bias is subtracted using an array of ripple borrow subtractors.

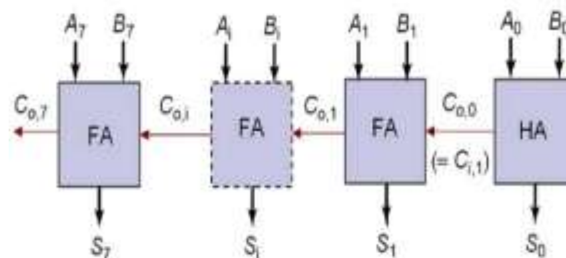


Fig 3. Ripple Carry Adder.

C. Multiplier for Unsigned Data :

This unit is used to multiply the two unsigned significand numbers and it places the decimal point in the multiplied product. The result of this significand multiplication will be called the intermediate product (IP). Multiplication is to be carried out so as not to affect the whole multiplier’s performance. In proposed parallel multiplier the carry bits are passed diagonally downwards. Partial products are generated by AND the inputs of two numbers and passing them to the appropriate adder.

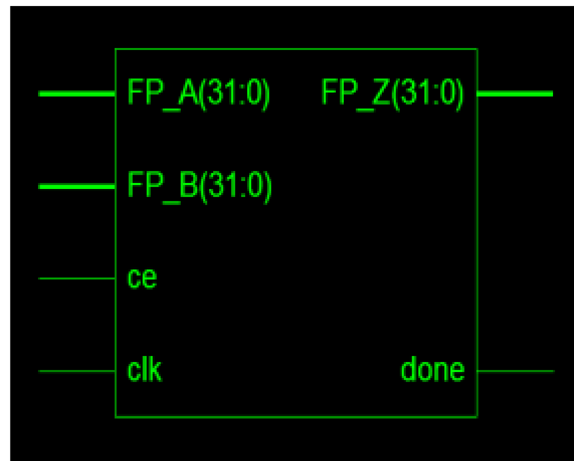


Fig 4. Schematic representation of Multiplier

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. The shift operation is done using combinational shift logic made by multiplexers.

4. UNDERFLOW / OVERFLOW DETECTION

Overflow/underflow means that the result's exponent is too large/small to be represented in the exponent field. The exponent of the result must be 11 bits in size, and must be between 1 and 1024 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

$$E_{result} = E1 + E2 - 1023.$$

E1 and E2 can have the values from 1 to 1022; resulting in E_{result} having values from -1021 (2-1023) to 762 (1785-1023); but for normalized numbers, E_{result} can only have the values from 1 to 1023. Table I summarizes the E_{result} different values and the effect of normalization on it.

ERESULT	Specification	Observation
-1023 = ERESULT < 0	Underflow	Cannot be compensated during normalize
ERESULT = 0	Zero	May turn to normalized number during normalization
1 < ERESULT < 1023	Normalised Number	May result in overflow during normalization
1023 = ERESULT	Overflow	Cannot be compensated

Table I: Normalization Effect On Result's Exponent And Overflow/Underflow Detection

5. SIMULATION RESULTS

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 = 16.33;
b = 27.44;
Output: result = 448.0952;

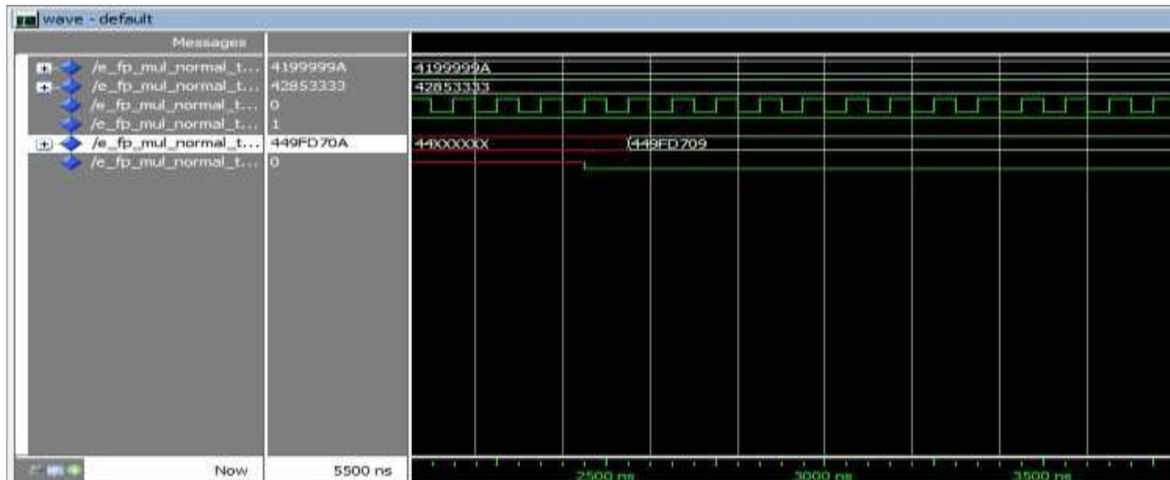


Fig. 5. Floating point Multiplier Simulation

A. Observation

	Existing Technique	Proposed technique
Time taken for execution	2500ns	2200 ns

6. CONCLUSION

The new design of multiplier has efficient in operation as well as has used less components as need of Subtractor circuit for the exponent calculation also reduced and the delay time for the multiplication is 14 nanosecond which is also less than the previously designed multipliers. So we say that the proposed multiplier has efficient in operation and cost. This paper presents an implementation of a floating point multiplier that supports the IEEE 754 -2008 binary interchange format; the multiplier doesn't implement rounding and just presents the significand multiplication result as is (104 bits); this gives better precision if the whole 104 bits are utilized in another unit.

In future, Double Precision floating point multiplier has been implemented by using proposed parallel multiplier, which consume low power and took 2500ns to execute. With unsigned multiplication there is no need to take the sign of the number into consideration. However in signed multiplication the same process cannot be applied because the signed number is in a 2's compliment form which would yield an incorrect result if multiplied in a similar fashion to unsigned multiplication. Therefore such algorithm is required which can be applicable for both numbers. Booth multiplier is such a multiplier which is used for signed number. Booth algorithm provides a procedure for multiplying binary integers in signed-2's complement representation. Therefore floating point multiplier can also be implemented by using Booth Algorithm.

REFERENCES:

[1] Shahzad Khan. "Implementation and Simulation of Ieee 754 Single- Precision floating point multiplier", Bhopal, 2014.
[2] Pierre, L. "VHDL description and formal verification of systolic multipliers. In CHDL", N. Holland, 1993.
[3] A. Kaldewaij, "Programming: The Derivation of Algorithms", Prentice-Hall, 1990.
[4] John G. Proakis and Dimitris G. Manolakis (1996), "Digital Signal Processing: Principles,. Algorithms and Applications", Third Edition
[5] A. D. Booth, "A signed binary multiplication technique". Quart. J. of Mech. Appl. Math, 4(2), 1951.
[6] Volnei A. Pedroni: "Circuit Design and Simulation with VHDL", Second edition, PHI.
[7] BROWN, Stephen D., "Fundamentals of Digital Logic with VHDL design", Boston: McGraw-Hill, 2000.
[8] Peter J. Ashenden, "The Designer"s Guide to VHDL", Morgan Kaufmann Publishers, 95 Inc., 1996.
[9] V. Kantabulra, "Designing optimum one-level carry-skip adders," IEEE Transactions on Computers, vol. 42, no. 6, pp. 759-764, June 1993.
[10] Beebe, H.F.Nelson, "Floating Point Arithmetic, Computation in Modern Science & Technology", December, 2007
[11] N. Weste, D. Harris, "CMOS VLSI Design", Third Edition, Addison Wesley, 2004