SINGLE PRECISION FLOATING POINT MULTIPLIER USING SHIFT AND ADD ALGORITHM

Ms. Pallavi Ramteke*
Communication Engg., RTMNU

Dr. N. N. Mhala
Electronics Engg. RTMNU

Prof. P. R. Lakhe
Electronics & Comm. Engg. RTMNU

Abstract— ***Floating-point numbers are widely adopted in many applications due to their dynamic representation capabilities. Basically floating point numbers are one possible way of representing real numbers in binary format. Floating-point representation is able to retain its resolution and accuracy compared to fixed-point representations. Multiplying floating point numbers is also a critical requirement for DSP applications involving large dynamic range. The IEEE has produced a standard to define floating point representation and arithmetic which is known as IEEE 754 standards and which is the most common representation today for real numbers on computer. The IEEE 754 standard presents two different floating point formats, Binary interchange format and Decimal interchange format. This paper presents a single precision floating point multiplier based on shift and add algorithm that supports the IEEE 754 binary interchange format..***

Keywords— ***floating point multiplier, Shift and Add Multiplier, Modelsim 6.3f simulator, Xilinx9.1 Synthesizer***

I. INTRODUCTION

Floating Point (FP) multiplication is widely used in large set of scientific and signal processing computation. Multiplication is one of the common arithmetic operations in these computations. Also the need of high speed multiplier is increasing as the need of high speed processors are increasing. Higher throughput arithmetic operations are important to achieve the desired performance in many real time signal and image processing applications. One of the key arithmetic operations in such applications is multiplication and the development of fast multiplier circuit has been a subject of interest over decades. Also reducing the time delay and power consumption are very essential requirements for many applications.

Floating point numbers are one possible way of representing real numbers in binary format. IEEE 754 basically specifies two formats for representing floating point values. They are single precision and double precision floating point format. This paper presents a single precision floating point format. 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 the significand. 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 the equation (1). Significand is the mantissa with an extra MSB bit.

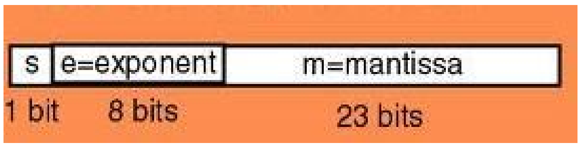

Figure 1. IEEE single precision floating point format

$$Z = (-1^{S}) * 2^{(E - Bias)} * (1.M)$$
$$Bias = 127$$
$$\therefore \text{Value} = (-1^{\text{Sign bit}}) * 2^{(\text{Exponent} -127)} * (1.\text{Mantissa})$$

II. FLOATING POINT MULTIPLICATION ALGORITHM

The normalized floating point numbers have the form of $Z = (-1S) * 2^{(E - Bias)} * (1.M)$. The following algorithm is used to multiply two floating point numbers.

1. Multiplying the significand; i.e. (1.M1*1.M2) (By using Shift and Add algorithm)
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.

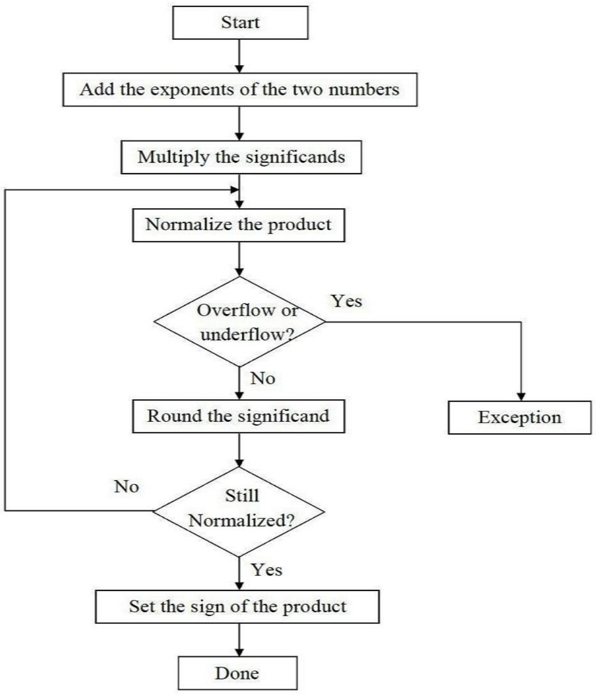

Fig. 2. Floating Point Multiplier Flow Chart

III. MAIN BLOCKS OF FLOATING POINT MULTIPLIER

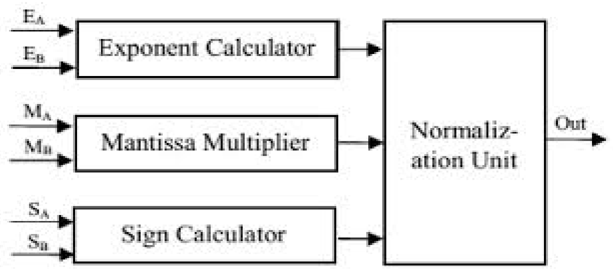

Fig. 3. Floating point multiplier block diagram

A. *Sign calculator*

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 (127) is subtracted from the result to get true result i.e. EA + EB – bias. To perform addition of two 8-bit exponents, an 8-bit ripple carry adder (RCA) is used. The Bias is subtracted using an array of ripple borrow subtractors.

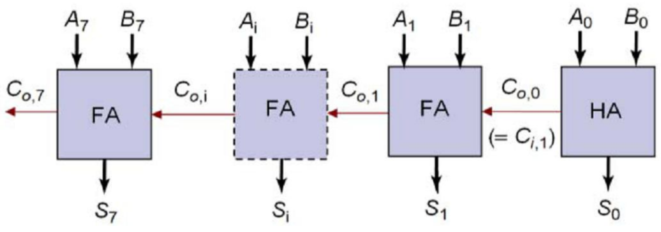

Fig. 4. Ripple Carry Adder

C. *Unsigned Multiplier (for significand multiplication)*

i) *Shift and Add Multiplier*

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 shift and add 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.

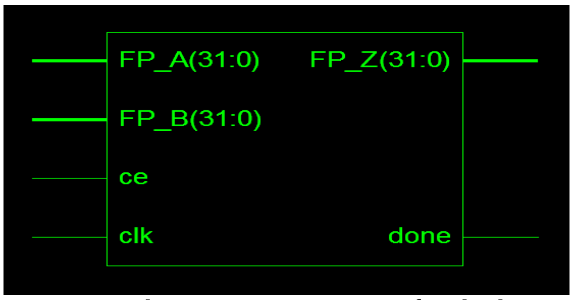

Fig. 5. Schematic representation of Multiplier

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.

IV. Underflow/Overflow Detection

Underflow/Overflow means that the result's exponent is too small/large to be represented in the exponent field. 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.

TABLE I. NORMALIZATION EFFECT ON
RESULT'S EXPONENT AND OVERFLOW/UNDERFLOW DETECTION

E result	**Category**	**Comments**
-125 ≤ 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	May result in overflow during normalization
255 ≤ E result	Overflow	Can't be compensated

V. Simulation Result

The simulation results for corresponding inputs are shown in Fig. The simulation is done using Modelsim 6.3f and for synthesis purpose Xilinx 9.1 software is used.
Considering the random floating point numbers,
Inputs: a = 19.2;
b = 66.6;

Output: result = 1278..72;

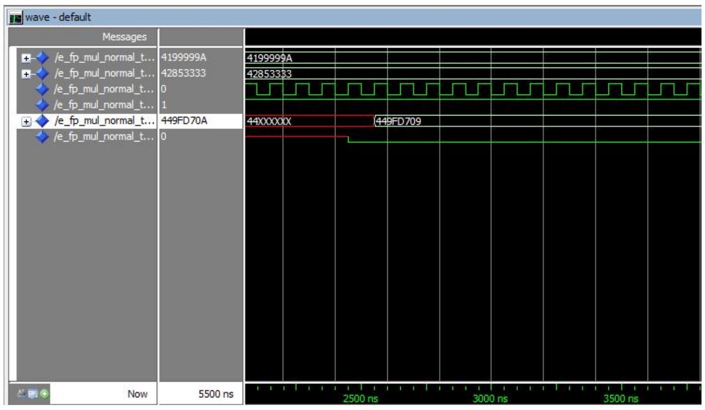

Fig. 6. Floating point Multiplier Simulation

A. *Observation*

Time taken for execution	2500 ns

VI. Conclusion

This paper describes an implementation of a floating point multiplier using Shift and Add Algorithm that supports the IEEE 754 binary interchange format; the multiplier is more precise because it doesn't implement rounding and just presents the significand multiplication. The multiplication time is reduced by using Shift and Add Algorithm. The design has been simulated on Modelsim 6.3f and synthesizes on a Xilinx 9.1 and achieved better speed.

VII. Future Work

Single Precision floating point multiplier has been implemented by using Shift and Add 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.

Acknowledgment

I would like to thank my guide Dr. N. N. Mhala and co-guide Prof. P. R. Lakhe for their support and guidance.

References

[1] IEEE 754-2008, IEEE Standard for Floating-Point Arithmetic, 2008.
[2] Mohamed Al-Ashrfy, Ashraf Salem and Wagdy Anis "An Efficient implementation of Floating Point Multiplier" IEEE Transaction on VLSI
[3] 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.
[4] 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.
[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.S. M. Metev and V. P. Veiko, *Laser Assisted Microtechnology*, 2nd ed., R. M. Osgood, Jr., Ed. Berlin, Germany: Springer-Verlag, 1998.
[6] A. Jaenicke and W. Luk, "Parameterized Floating-Point Arithmetic on FPGAs", Proc. of IEEE ICASSP, 2001, vol. 2, pp.897-900.
[7] B. Lee and N. Burgess, "Parameterisable Floating-point Operations on FPGA," Conference Record of the Thirty-Sixth Asilomar Conference on Signals, Systems, and Computers, 2002.
[8] "DesignChecker User Guide", HDL Designer Series 2010.2a, Mentor Graphics, 2010.
[9] "Precision® Synthesis User's Manual", Precision RTL plus 2010a update 2, Mentor Graphics, 2010.