Implementation of Floating Point Multiplier

Unnati Mehta, Prerna Gupta, Pooja Sengar, Hitanshi Shishodia, Priyanka Yadav
Department of Electronics and Communication Engineering College
18th km stone, Nh-24-ABES Engineering College,
Ghaziabad, Uttar Pradesh, India

Abstract: In VERILOG design it is possible to perform normal multiplication, addition, subtraction but it is difficult to perform floating point multiplication. So in this we implementing a new algorithm for performing the floating point multiplication. Floating point number can represent a very large or a very small. It could also represent very large negative number and very small negative number as well as zero. Floating point number is typically expressed in the scientific notation, with a fraction (F), and exponent (E) of a certain radix(r). Modern computers adopt IEEE 754 standard for representing floating point numbers. Floating point number consists of two fixed point components, whose range depends exclusively on the number of bits or digits in their representation. Whereas components linearly depend on their range, the floating point range linearly depends on the significant range and exponentially on the range exponent component, which attaches outstandingly wider range to the number. In this paper we perform -32-bit and 64-bit floating-point multiplication. Floating point multiplication is important in many commercial applications including financial analysis, banking, tax calculation, currency conversion, insurance, and accounting.

Keywords*: Floating point number, Exponent, Mantissa, Normalization, rounding.*

INTRODUCTION:

IEEE 754 floating point standard is the most common representation today for real numbers on computers. The IEEE (Institute Of Electrical And Electronics Engineers) has produced a standard to define floating –point representation and arithmetic. Although there are other representation used for floating point numbers. The standard brought out by the IEEE come to be known as IEEE 754.It is interesting to note that the string of significant digits is technically termed the mantissa of the number, while the scale factor isappropriately called the exponent of the number.

LITERATURE REVIEW

Various researches have been done to increase the performance on getting best and fast multiplication result on two floating point numbers. Some of which are listed below-

Addanki Puma Ramesh, A. V. N. Tilak, A.M.Prasad [1] the double precision floating point multiplier supports the LEEE-754 binary interchange format. The design achieved the increased operating frequency. The implemented design is verified with single precision floating point multiplier and Xilinx core, it provides high speed and supports double precision, which gives more accuracy compared to single precession. This design handles the overflow, underflow, and truncation rounding mode resp.

Itagi Mahi P and S. S. Kerur [2] ALU is one of the important components within a computer processor. It performs arithmetic functions like addition, subtraction, multiplication, division etc along with logical functions. Pipelining allows execution of multiple instructions simultaneously. Pipelined ALU gives better performance which will evaluated in terms of number of clock cycles required in performing each arithmetic operation. Floating point representation is based on IEEE standard 754. In this paper a pipelined Floating point Arithmetic unit has been designed to perform five arithmetic operations, addition, subtraction, multiplication, division and square root, on floating point numbers. IEEE 754 standard based floating point representation has been used. The unit has been coded in VHDL. The same arithmetic operations have also been simulated in Xilinx IP Core Generator.

Remadevi R [3] Multiplying floating point numbers is a critical requirement for DSP applications involving large dynamic range. This paper presents design and simulation of a floating point multiplier that supports the IEEE 754-2008 binary interchange format, the proposed multiplier does not implement rounding and presents the significant multiplication result. It focuses only on single precision normalized binary interchange format. It handles the overflow and underflow cases. Rounding is not implemented to give more precision when using the multiplier in a Multiply and Accumulate (MAC) unit. Rakesh Babu, R. Saikiran and Sivanantham S [4] A method for fast floating point multiplication and the coding is done for 32-bit single precision floating point multiplication using Verilog and synthesized. A floating point multiplier is designed for the calculation of binary numbers represented in single precision IEEE format. In this implementation exceptions like infinity, zero, overflow are considered. In this implementation rounding methods like round to zero, round to positive infinity, round to negative infinity, round to even are considered. To analyse the working of our designed multiplier we designed a MAC unit and is tested. These results are compared with the previous work done by various authors.

FLOATING POINT MULTIPLICATION ALGORITHM:

As stated in the introduction, normalized floating point numbers. 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

IEEE 754 Floating Point Formats:

IEEE 754 specifies four formats for representing floating-point values:

1. Single precision (32-bit)
2. Double precision (64-bit)
3. Single-extended precision (≥43-bits, not commonly used)
4. Double-extended precision (≥79-bit, usually implemented with 80 bits)

A. Single Precision floating point Numbers:

The Single-precision number is 32-bit wide. The single-precision number has three main fields that are sign, exponent, and mantissa .The 24-bit mantissa can approximately represents a 7-digit decimal number, while an 8-bit exponent to an implied base of 2 provides a scale factor with a reasonable range. Thus a total of 32-bit is needed for single-precision number representation. To achieve this, a bias equal to 2^{n-1}-1 is added to the actual exponent in order to obtain the stored exponent. This equals 127 for an eight-bit exponent of the single127 for an eight-bit exponent of the single precision format. The addition of bias allows the use of an exponent in the range from -127 to +128, corresponding to a range of 0-255 for single precision. The single-precision format offers the range from $2^{-1\,27}$ to 2^{+127}. Which equivalent to 10^{-38} to 10^{+38}.

Steps for conversion:

Let us represent the decimal number (-0.03125)10 in IEEE floating-point format.

STEP1: Convert the number into binary form (0.03125) 10 = (0.00001)2

STEP2: Convert (0.00001)2 into floating point representation. 0.00001× 2+0 = 0.00001

STEP3: Normalized the value 0.00001 000001×2-5 = 1× 2-5

STEP4: Biased exponent =127-5

=122 =1111010

1	01111010	10...............0

Fig: example of single precision

EXAMPLE:

Let A=85.125 and B=45.125

IEEE Representation of operands

A =01000010101010100100000000000000

B =01000010001101001000000000000000

To multiply A and B

1. Multiply significand

1.01010100100000000000000

*1.01101001000000000000000

2. Add exponents

10000101 + 10000100= E

The exponent representing the two numbers is already shifted/biased by the bias value (127) and is not the true exponent; i.e. $E_A = E_{A\text{-true}} +$ bias and $E_B = E_{B\text{-true}} +$ biasand$E_A + E_B = E_{A\text{-true}} + E_{B\text{-true}} + 2$ bias .So we should subtract the bias from the resultant exponent otherwise the bias will be added twice.

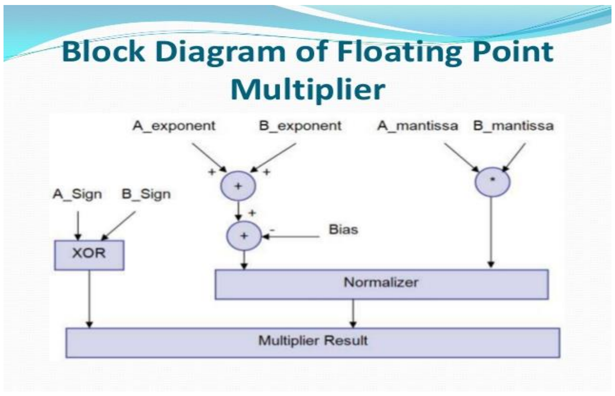
E-01111111

3.Xoring the sign bit and put the result together:

Xor bits:0 Exponent:10001010 mantissa:11.11000000001010001000000000000000000000000000000

4.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.

The result is (without the hidden bit):01000101011100000000101000100000 0



IMPLEMENTATION

Simulation flow in ModelSim:

Creating the working library: In ModelSim , all the designs are compiled into a library. We start a new simulation in ModelSim by creating a working library called work. Work is the library name used by the compiler as the default destination for compiled design units.

Compile the design: Before the simulate a design, we must first create and compile the source code into that library.

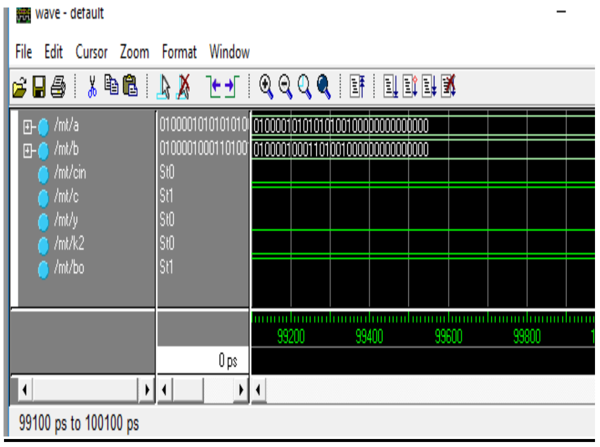
Loading the design into simulator: Load the test design module into the simulator. Double click test design in the Main window Workspace to load the design. It can also load the design by selecting Simulate > Start Simulation in the menu bar. This opens the Start Simulation dialog.

Running the simulation: Go to simulate > start simulation > run > run all. Time taking for simulation is 950ps.

Debugging the results: If we don't get the results we expect, then we can use ModelSim's robust debuggung environment to track down the cause of the problem.

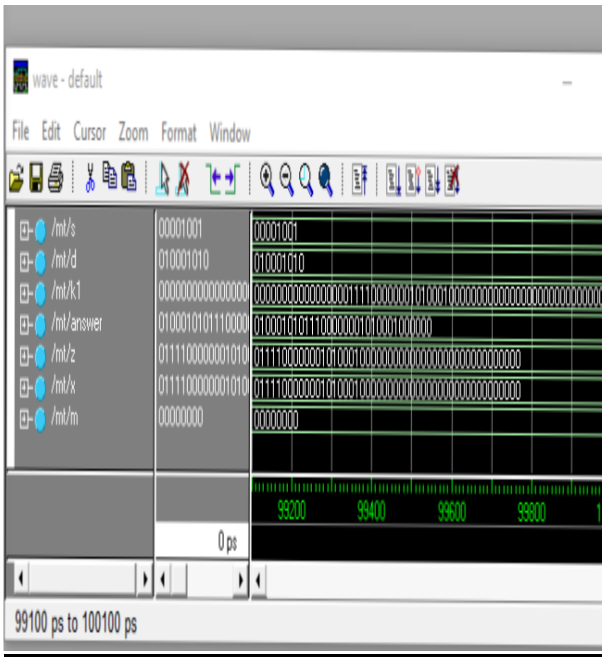
CONCLUSION:

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 (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 and after implementation on a Xilinx Virtex5 FPGA it achieves 301 MFLOPS



```
/mt/a     01000010101010100  01000010101010010000000000000000
/mt/b     01000010001101001  01000010001101001000000000000000
/mt/cin   St0
/mt/c     St1
/mt/y     St0
/mt/k2    St0
/mt/bo    St1
```

99100 ps to 100100 ps



```
/mt/s       00001001          00001001
/mt/d       010001010         010001010
/mt/k1      000000000000000   000000000000000011110000000101000100000000000000000000000000
/mt/answer  010001010111000   01000101011100000000101000100000
/mt/z       011110000000101   01111000000010100010000000000000000000000000
/mt/x       011110000000101   01111000000010100010000000000000000000000000
/mt/m       00000000          00000000
```

99100 ps to 100100 ps

REFERENCES:

[1] IEEE754-2008,IEEE Standardfor Floating-PointArithmetic,2008.

[2] B.Faginand C.Renard,“Field ProgrammableGateArraysand FloatingPointArithmetic,”IEEETransactionsonVLSI,vol.2,no.3 .

[3] N.Shirazi,A.Walters,andP.Athanas,“QuantitativeAnalysis of Floating Point Arithmeticon FPGA Based CustomComputing Machines,”Proceedingsofthe IEEESymposiumon FPGAsforCustom Computing Machines (FCCM’95),pp.155–162,1995.

[4] L.Louca,T.A.Cook,andW.H.Johnson,“Implementation of IEEE Single PrecisionFloating PointAdditionand Multiplicationon FPGAs,” Proceedingsof83theIEEESymposiumonFPGAsforCustom Computing Machines(FCCM’96),pp.107–116,1996.