International Journal of Engineering Research and Development
e-ISSN: 2278-067X, p-ISSN: 2278-800X, www.ijerd.com
Volume 10, Issue 3 (March 2014), PP.73-78

Design of Floating Point Multiplier Using Vhdl

P.Gayatri[1], P.Krishna Kumari[2], V.Vamsi Krishna[3], T.S.Trivedi[4], V.Nancharaiah[5]
[1,2,3,4,5]*(Department of Electronics & Communication Engineering, Lendi Institute of Engineering and Technology/JNTUK, India)*

Abstract:- In VHDL design 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, Overflow.

I. 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 is appropriately called the exponent of the number. The general form of the representation is the following $(-1)^S * M* 2^E$. Where S represents the sign bit, M represents the mantissa and E represents the exponent. When it comes to their precision and width in bits, the standard defines two groups: base and extended format. The basic format is further divided into Single –Precision format with 32-bits wide, and double-precision format with 64-bits wide. The three basic components are the sign, exponent, and mantissa.

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 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}.

Sign: 1-bit wide and used to denote t he sign of the number i.e. 0 indicate positive number, 1represent negative number.
Exponent: 8-bit wide and signed exponent in excess -127 representations.
Mantissa: 28-bit wide and fractional component**.**

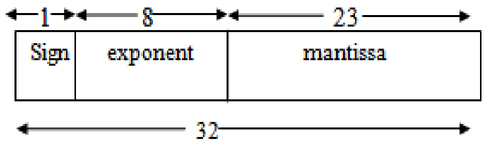
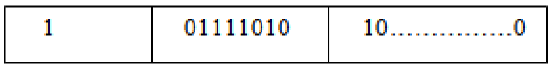


Fig1: Single-precision floating point representation

Number Representation using Single Precision Format:
Let us try and 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.
$0.00001\times 2^{+0} = 0.00001$
STEP3: Normalized the value 0.00001
$000001\times 2^{-5} = 1\times 2^{-5}$
STEP4: Biased exponent =127-5
=122 =1111010



1	01111010	10…………..0

Fig2: example of single precision

B. Double Precision floating point Numbers:

The double precision number is 64-bit wide. The double-precision number has three main fields that are sign, exponent, and mantissa. The 52-bit mantissa, while an 11-bit exponent to an implied base of 2 provides a scale factor with reasonable range. Thus a total of 64 bits 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 is equal 1023 for an 11-bit exponent of the double-precision format. The addition of bias allows the use of an exponent in the range from -1023 to +1024, corresponding to a range of 0-2047 for double precision. . The double-precision format offers the range from $2^{-10\ 23}$ to 2^{+1023}. Which equivalent to 10^{-308} to 10^{+308}.
Sign: 1-bit wide and used to denote t he sign of the number i.e., 0 indicate positive number, 1 represent negative number.
Exponent: 11-bit wide and signed exponent in excess -1023representations.
Mantissa: 52-bit wide and fractional component**.**

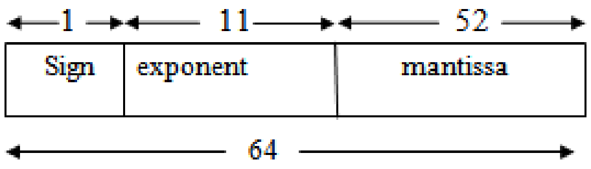
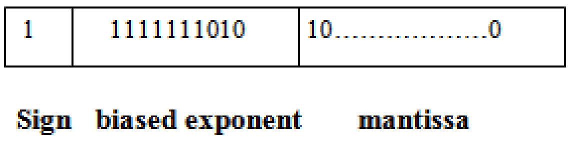


Fig3: Double-precision floating point representation

Number Representation using Double Precision Format:
Let us try and represent the decimal number $(-0.03125)_{10}$
STEP1: Convert the number into binary form
$(0.03125)_{10} = (0.00001)_2$
STEP2: Convert $(0.00001)_2$ into floating point representation.
$0.00001\times 2^{+0} = 0.00001$
STEP3: Normalized the value 0.00001
$000001\times 2^{-5} = 1\times 2^{-5}$
STEP4: Biased exponent =1023-5
=1018
= 1111111010



1	1111111010	10……………..0
Sign	**biased exponent**	**mantissa**

Fig4: example of double precision

II. DESIGN

Floating point multiplier block diagram:

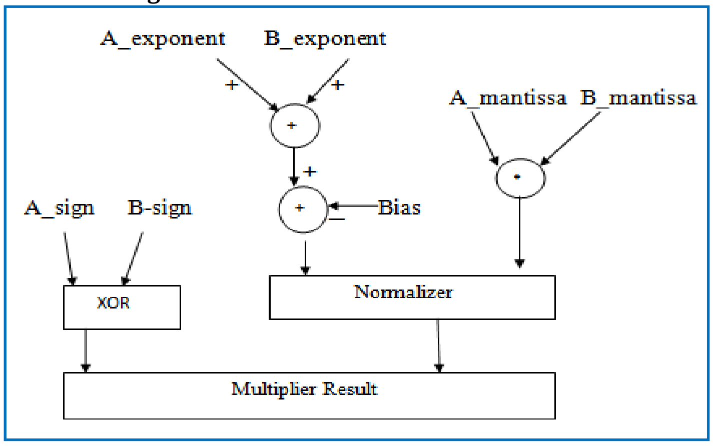


Fig5: Floating point multiplier

The figure5 shows the multiplier structure; Exponents addition, Significand multiplication, and Result's sign calculation are independent and are done in parallel.
The significand and 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 down to 0) and the decimal point is located between bits 46 and 45 in the IP.

Floating point multiplication:
The simplest floating point operation is multiplication, so we discuss it first. A binary floating point number x is represented as a significand and an exponent.

$$X = s * 2^e$$

The formula

$$(s1 * 2^{e1}) \cdot (s2 * 2^{e\,2}) = (s1 \cdot s2) * 2^{e1+e\,2}$$

Shows that a floating-point multiply algorithm has several parts. T he first part multiplies the significands using ordinary integer multiplication. Because floating point numbers are stored in sign magnitude form , the multiplier need only deal with unsigned numbers. The second part rounds the result. If the significands are unsigned p-bit number, then the product can have as many as 2p bits and must be rounded to a p-bit number. The third part compues the new exponent. Because exponents are stored with bias, this involves subtracting the bias from the sum of biased exponent.

Example:

Let's suppose a multiplication of 2 floating-point numbers A and B, where A= −18 ·0 and B=9·5

- Binary representation of the operands:

A = −10010 · 0
B = + 1001.1

- Normalized representation of the operands:

  $A = -1 \cdot 001 \times 2^4$
  $B = +1 \cdot 0011 \times 2^3$

- IEEE representation of operands:

  A = 1 10000011 00100000000000000000000
B = 0 10000010 0011000000000000000000

- Multiplication of the mantissa:
- We must extract the mantissa, adding an 1 as most significant bit, for normalization
  100100000000000000000000
  10011000000000000000000
- The 48-bit result of the multiplication is:
  0×558000000000
- Only the most significant bits are useful: after normalization , we get the 23-bit mantissa of the result. This normalization can lead to correction of the result's exponent
- In our case , we get:

01	01010110000000000000000	0000000000000000000000

Fig6: Normalization

- Addition of the exponents:
- Exponent of the result is equal to the sum of the operands exponents. A 1 can be added if needed by the normalization of the mantissa multiplication
- As the exponent fields (E_a and E_b) are biased, the bias must be removed in order to do the addition. And then, we must to add aginto the exponent the bias, to get the value to be entered into the exponent field of the result (E_r):

$$E_r = (E_a - 127) + (E_b - 127) + 127$$
$$= E_a + E_b - 127$$

- In our example, we have:

$$\begin{array}{l} E_a = 10000011 \\ E_b = 10000010 \\ -127 \\ \hline E_r\ 10000110 \end{array}$$

What is actually 7, the exponent of the result

- Calculation of the sign of the result:
- T sign of the result (S_r) is given by the exclusive-or of the operands signs (Sa and Sb) :

Sr = Sa xor Sb·

- in our example ,we get

Sr =1 xor 0 =1 i.e a negative sign

- Composite of the result:

The setting of the 3 intermediate result (sign, mantissa and exponent) gives us the final result of our multiplication:

1	10000110	01010110000000000000000

Fig7:Multiplication result

$A \times B = -18 \cdot 0 \times 9 \cdot 5 = -1 \cdot 0101011 \times 2^{134\ -127} = -10101011 \cdot 0 = -171 \cdot 0_{10}$

Normalization:

The normalization step requries:

The detection of the position of the leading 1 uses LOD (Leading-One-Detector)

A shift performed by the shifter :

- no shift
- Right shift of one position , or
- Left shift of up to m positions

Rounding:

- Round to nearest
- Round toward zero
- Round toward plus infinity
- Round toward minus infinity

Zero: When one of the operands has value 0 and the other is not $\mp$ infinity;

- Zero result set

Over flow: exponent too large;

Detected after exponent update;

Over flow set; result value is $\mp$ infinity

Underflow: resulting exponent too small;

Underflow flag set; exponent set to E=0

Significand shifted right to represent a denormal

III. IMPLIMENTATION

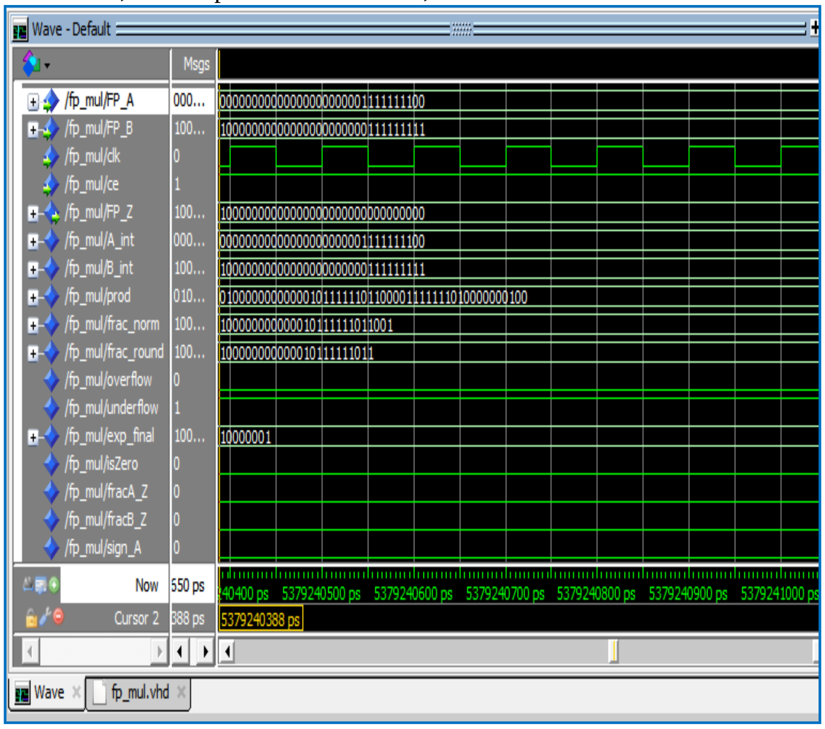
Simulation flow in Model sim:

- **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

IV. RESULTS AND ANALYSIS:

The design has been implemented and simulated by using ModelSim.
Consider inputs to the floating point multiplier are:
A = 00111111110000000000000000000000
B = 11111111100000000000000000000001
The output of the multiplier should be
0100000000000010111111011000011111111010000000100
Flag outputs of this multiplier are
Overflow = 0; underflow = 1; final exponent = 10000001; zero = 0



	Msgs	
/fp_mul/FP_A	000...	00000000000000000000001111111100
/fp_mul/FP_B	100...	10000000000000000000000111111111
/fp_mul/clk	0	
/fp_mul/ce	1	
/fp_mul/FP_Z	100...	10000000000000000000000000000000
/fp_mul/A_int	000...	00000000000000000000001111111100
/fp_mul/B_int	100...	10000000000000000000000111111111
/fp_mul/prod	010...	0100000000000010111111011000011111111010000000100
/fp_mul/frac_norm	100...	100000000000010111111011001
/fp_mul/frac_round	100...	100000000000010111111011
/fp_mul/overflow	0	
/fp_mul/underflow	1	
/fp_mul/exp_final	100...	10000001
/fp_mul/isZero	0	
/fp_mul/fracA_Z	0	
/fp_mul/fracB_Z	0	
/fp_mul/sign_A	0	
Now	550 ps	40400 ps 5379240500 ps 5379240600 ps 5379240700 ps 5379240800 ps 5379240900 ps 5379241000 ps
Cursor 2	388 ps	5379240388 ps

Fig8: Input and output waveform

V. CONCLUSION

The floating point multiplier is design for both 32-bit and 64-bit by varying the input variables.

REFERENCES

[1]. IEEE standard for binary-floating point arithmetic, ANSI/IEEE Std 754-1985, The Institute of Electrical and Engineers Inc..New York. August 1985.
[2]. David Goldberg: What Every Computer Scientist Should Know About Floating-point Arithmetic, 1991.
[3]. I.Koren, Computer Arithmetic Algorithms, Second Edition, prentice Hall, 2002.
[4]. An ANSI/IEEE Standard for Radix-Independent Floating-point arthmetic, technical Committee on microprocessor of IEEE computer society, October, 1987.
[5]. Steve Hollasc , IEEE Standard 754 Floating Point Numbers ,February 2005.
[6]. BROWN, Stephen D. Fundamentals of Digital Logic with VHDL designs. Boston: McGraw-Hill , 2000.
[7]. John L Hennesy & David A. Patterson "Computer Architecture A Quantitative Approach" Second edition: A Harcourt Publisher International Company
[8]. J. Bhasker , A VHDL Primer ,Third Edition, Pearson, 1999.

[9]. M. Ercgovac and T. Lang, Digital Arithmetic, Morgan Kaufmann Publishers, 2004.
[10]. John. P. Hatyes , “Computer Architecture and Organization”, McGraw Hill , 1998.
[11]. Peter J. Ashenden , The Designer’s Guide to VHDL, Morgan Kaufmann Publishers , 95 Inc., 1996.
[12]. Prof. W. Kahan, Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-point Arithmetic .
[13]. Wikipedia, the free encyclopedia, IEEE 754-1985.
[14]. Behrooz Parhami, Computer Arithmetic, Algorithms and Hardware Design Oxford University Press . 2000.
[15]. IEEE Floating Point Representation of Real Number, Fundamentals of Computer Science.
[16]. M . J. Flym and S . F. Oberman, Advanced Computer Arithmetic Design, Jhon Wiley and Sons, 2001.
[17]. N. Weste, D. Harris, CMOS VLSI Design, Third Edition, Addison Wesely, 2004.
[18]. Beebe, H . F. Nelson, Floating Point Arithmetic, Computation in Modern Science & Technology, December, 2007.
[19]. P . Karlstrom , A. Ehliar , High Performance Low Latrncy Floating Piont Multiplier , November 2006