© 2015 IJSRSET | Volume 1 | Issue 1 | Print ISSN : 2395-1990 | Online ISSN : 2394-4099
Themed Section: Engineering and Technology

Design of Single Precision Floating Point Multiplication Algorithm with Vector Support

T.Govinda Rao[*1], D.Arun Kumar[2]
[*1]Research Scholar, Department of ECE, GMRIT, RAJAM, AP, INDIA
[2]Department of ECE, GMRIT, RAJAM, AP, INDIA

ABSTRACT

This paper presents floating point multiplier capable of supporting wide range of application domains like scientific computing and multimedia applications. The floating point units consume less power and small part of total area. Graphic Processor Units (GPUS) are specially tuned for performing a set of operations on large sets of data. This paper work presents the design of a single precision floating point multiplication algorithm with vector support. The single precision floating point multiplier is having a path delay of 72ns and also having the operating frequency of 13.58MHz.Finally this implementation is done in Verilog HDL using Xilinx ISE-14.2.

Keywords: floating point multiplier, GPUS, operating frequency, HDL

I. INTRODUCTION

Floating Point numbers represented in IEEE 754 format is used in most of the DSP Processors. Floating point arithmetic is useful in applications where a large dynamic range is required or in rapid prototyping applications where the required number range has not been thoroughly investigated. A Floating point multiplier is the most common element in most digital applications such as digital filters, digital signal processors, data processors and control units.

There are two types of number formats present.

1. Fixed point representation
2. Floating point representation.

These refer to the format used to store and manipulate numbers within the devices. Fixed point DSPs usually represent each number with a minimum of 16 bits. In comparison, floating point DSPs use a minimum of 32 bits to store each value. This results in many more bit patterns than for fixed point. All floating point DSPs can also handle fixed point numbers, a necessary to implement counters, loops, and signals coming from the ADC and going to the DAC.

In general purpose fixed point arithmetic is much faster than floating point arithmetic. However, with DSPs the speed is about the same, a result of the hardware being highly optimized for math operations. The internal hardware of floating point DSP is much complicated than for a fixed device. Floating point has better precision and a higher dynamic range than fixed point. In addition, floating point programs often have a shorter development cycle, since the programmer doesn't generally need to worry about issues such as overflow, underflow and round-off error.

Noise in signals is usually represented by its standard deviation. For here, the important fact is that the standard deviation of this quantization noise is about one-third of the gap size. This means that the signal-to-noise ratio for storing a floating point number is about 30 million to one, while for a fixed point number it is only about ten-thousand to one. In other words, floating point has roughly 30,000 times less quantization noise than fixed point.

The important idea is that the fixed point programmer must understand dozens of ways to carry out the very basic task of multiplication. In contrast, the floating point programmer can spend is time concentrating on the algorithm the cost of the DSP is insignificant, but the performance is critical. In spite of the larger number of fixed point DSPs being used, the floating point market is the fastest growing segment. Verilog programming has been used to implement Floating Point Multiplier.
Tool used for programming →XILINX ISE SUITE 14.2 Version.

II. METHODS AND MATERIAL

IEEE754 FLOATINGPOINT REPRESENTATION

Basic Representation

IEEE floating point numbers have three basic components: the sign, the exponent, and the mantissa. The mantissa is composed of the fraction and an implicit leading digit. The exponent base 2 is implicit and need not be stored.

Single Precision:

31	30		24	23		2	1	0

Figure IEEE-754 Single Precision Bit Format

$$Z = (-1)^{S} \times 2^{(E-\mathrm{Bias})} \times (1 \cdot M)$$

Where
S = Sign Bit
E = Exponent
M = Mantissa

- The sign bit is as simple as it gets. 0 denotes a positive number; 1 denotes a negative number. Flipping the value of this bit flips the sign of the number.
- The exponent field needs to represent both positive and negative exponents. To do this, a bias is added to the actual exponent in order to get the stored exponent.
- For IEEE single-precision floats, this value is 127. Thus, an exponent of zero means that 127 is stored in the exponent field. A stored value of 200 indicates an exponent of (200-127), or 73. For reasons discussed later, exponents of -127 (all 0s) and +128 (all 1s) are reserved for special numbers.
- Significand is the mantissa with an extra MSB bit i.e.,1 which represents the precision bits of the number. It is composed of an implicit leading bit and the fraction bits.
- $M = m_{22}2^{-1} + m_{21}2^{-2} + m_{20}2 + \cdots\cdots\cdots + m_{1}2^{-22} + m_{0}2^{-23}$

Floating Point Multiplication Algorithm

Normalized floating point numbers have the form of

$Z = (-1^{S}) * 2^{(E - Bias)} * (1.M)$.

Steps for Floating Point Multiplication

To multiply two floating point numbers the following is done:

1. Multiplying the significand; i.e. $(1 \cdot M_1 \times 1 \cdot M_2)$
2. Placing the decimal point in the result
3. Adding the exponents; i.e. $E_1 + E_2 - \mathrm{Bias}$
4. Obtaining the sign; i.e. s_1EXOR s_2
5. Normalizing the result; i.e. obtaining 1 at the MSB of the results significant
6. Rounding the result to fit in the available bits
7. Checking for underflow/overflow occurrence

Multiplication using Two Numbers

Consider a floating point representation similar to the IEEE 754 single precision floating point format, but with a reduced number of mantissa bits. Here only 4 bits are considered for mantissa instead of 23 bits for easy understanding.

Let the two numbers be:

A = 0 00001001 1110 = 60

B = 1 10000001 1010 = -6.5

The significant of the above numbers can be obtained by retaining the hidden bit 1 of the two mantissa.

To multiply A and B

1. Multiply significant:

```
        1.1110
       x1.1010
      --------
        00000
        11110
        00000
    11110
    00000
   ----------
   110001100
```

2. Place the decimal point: 11.00001100

3. Add exponents:

```
     10000100
   + 10000001
   ----------
    100000101
```

The exponent representing the two numbers is already shifted/biased by the bias value (127) and is not the true exponent; i.e. $E_A = E_{A\text{-true}} + \text{bias}$ and $E_B = E_{B\text{-true}} + \text{bias}$

And

$$E_A + E_B = E_{A-\text{true}} + E_{B-\text{true}} + 2 * \text{Bias} \qquad (3.1)$$

So we should subtract the bias from the resultant exponent otherwise the bias will be added twice.

```
    100000101
  -   1111111
  -----------
     10000110
```

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

   1 10000110 11.0001100

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 11.00001100 (before normalizing)

   1 10000111 1.100001100 (normalized)

The result is (without the hidden bit):

1 10000111 100001100

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 1000

Structure of the Multiplier

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.

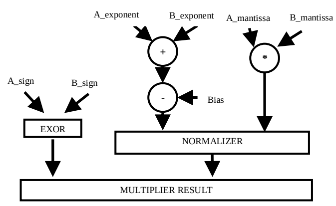

Figure 1. Floating Point Multiplier Block Diagram

The floating point multiplier structure contains the:

1. Exponents addition
2. Significant multiplication
3. Result's sign calculation

These functions are independent and are done in parallel. The significant multiplication is done on two mantissa bit numbers, which we will call the intermediate product (IP). The IP is represented as (47 down to 0) for single precision (127 down to 0) for double precision. The following sections detail each block of the floating point multiplier.

Hardware of Floating Point Multiplier

Unsigned 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. 4.2 a ripple carry adder is a chain of cascaded full adders and one half adder; each full adder has three inputs (A, B, C_i) and two outputs (S, C_o). The carry out (C_o) of each adder is fed to the next full adder (i.e. each carry bit "ripples" to the next full adder).

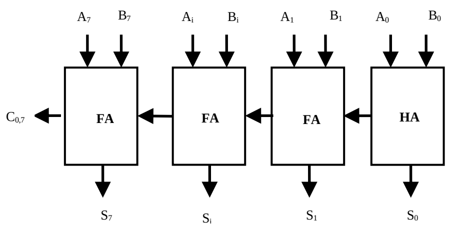

Figure 2 : Ripple Carry Adder

The addition process produces an 8 bit sum (S_7 to S_0) and a carry bit ($C_{o,7}$). These bits are concatenated to form a 9 bit addition result (S_8 to S_0) from which the Bias is subtracted.

Table 1: Port list Ripple Carry Adder

S.No.	Port	Direction	Size	Description
1.	E_1	Input	8	Bits 23-30 for the first input
2.	E_2	Input	8	Bits 23-30 for the second input
3.	S_0	Output	8	Result of the addition
4.	Ca	Output	1	Carry due to addition

SUBTRACTOR

The exponent of the IEEE 754 forat consists of the sum of original exponent and the bias value. While adding two exponent values, bias is added two times. So bias is subtracted from the result of the adder. The Bias is subtracted using an array of ripple borrow subtractors.

A normal subtractor has three inputs (minuend (S), subtrahend (T), Borrow in (B_i)) and two outputs (Difference (R), Borrow out (B_o)). The subtractor logic can be optimized if one of its inputs is a constant value which is our case, where the Bias is constant .In single precision the Bias subtractor which is a chain of 7 one subtractors (OS) followed by 2 zero subtractors (ZS); the borrow output of each subtractor is fed to the next subtractor. If an underflow occurs then $E_{result} < 0$ and the number is out of the IEEE 754 single precision normalized numbers range; in this case the output is signaled to 0 and an underflow flag is asserted.

Table 2: Port list of Ripple Borrow

S.No.	Port	Direction	Size	Description
1.	T	Input	8	Bias value
2.	S	Input	8	Result of ripple carry adder
3.	C_0	Input	1	Carry of ripple carry adder
4.	B_0	Output	1	Borrow output of subtractor
5.	R	Output	8	Output of the subtractor

Subtractor

Unsigned Multiplier

This unit is responsible for multiplying the unsigned significand and placing the decimal point in the multiplication product. The result of significand multiplication will be called the intermediate product (IP). The unsigned significand multiplication is done on 24 bit. Multiplier performance should be taken into consideration so as not to affect the whole multiplier's performance. A 24x24 bit carry save multiplier architecture is used and for the double precision 52 x52 bit carry save multiplier is used as it has a moderate speed with a simple architecture. In the carry save multiplier, the carry bits are passed diagonally downwards (i.e. the carry bit is propagated to the next stage).

Carry save multiplier has three main stages:

1. The first stage is an array of half adders.
2. The middle stages are arrays of full adders. The number of middle stages is equal to the significand size minus two.
3. The last stage is an array of ripple carry adders. This stage is called the vector merging stage.

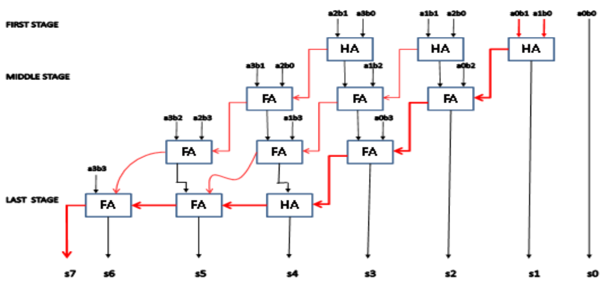

Figure 3 : Unsigned Multiplier

The number of adders (Half adders and Full adders) in each stage is equal to the significand size minus one. For example, a 4x4 carry save multiplier has the following stages

- The first stage consists of three half adders.
- Two middle stages; each consists of three full adders.
- The vector merging stage consists of one half adder and two full adders.

The decimal point is between bits 45 and 46 in the significand multiplier result. The multiplication time taken by the carry save multiplier is determined by its critical path. The critical path starts at the AND gate of the first partial products (i.e. a_1b_0 and a_0b_1), passes through the carry logic of the first half adder and the carry logic of the first full adder of the middle stages, then passes through all the vector merging adders. The critical path is marked

Table 3 : Port list of Unsigned Multiplier

S.No.	Port	Direction	Size	Description
1.	A_1	Input	24	Bit 0-22 of first input
2.	A_2	Input	24	Bit 0-22 of second input
3.	S	Output	47	Output of multiplier

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 (i.e. in the bit 46 in the intermediate product). Since the inputs are normalized numbers then the intermediate product has the leading one at bit 46 or 47

1. If the leading one is at bit 46 (i.e. to the left of the decimal point) then the intermediate product is already a normalized number and no shift is needed.
2. If the leading one is at bit 47 then the intermediate product is shifted to the right and the exponent is incremented by 1.

The shift operation is done using combinational shift logic made by multiplexers. Fig. 8 shows a simplified logic of a Normalizer that has an 8 bit intermediate product input and a 6 bit intermediate exponent input

Table 4 : Port list of Normalizer

S.No.	Port	Direction	Size	Description
1.	S_i	Input	24	Result of multiplier
2.	Ei	Input	8	Result of the subtractor
3	S_0	Output	23	Significand result due to normalization
4	E_0	Output	8	Final output of the exponent

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 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

$$E_{result} = E1 + E2 - 127$$

Table 5 : Underflow and Overflow Conditions

E_{result}	Category	Comments
$-125 \leq E_{result} < 0$	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 number	May result in overflow during Normalization
$255 \leq E_{result}$	Overflow	Can't be compensated

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 4.9 summarizes the E_{result} different values and the effect of normalization on it.

III. RESULTS AND DISCUSSION

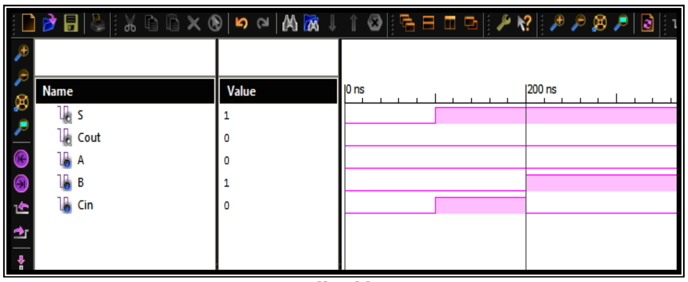

Figure 4 : : Full Adder Output

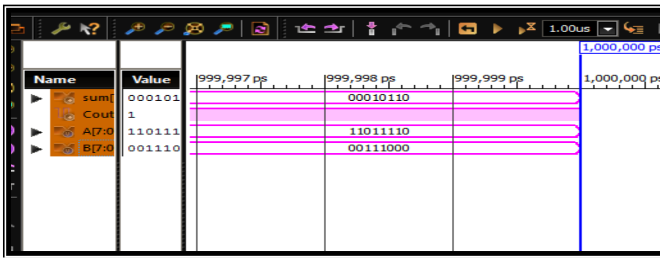

Figure 5 : Ripple Carry Adder Output

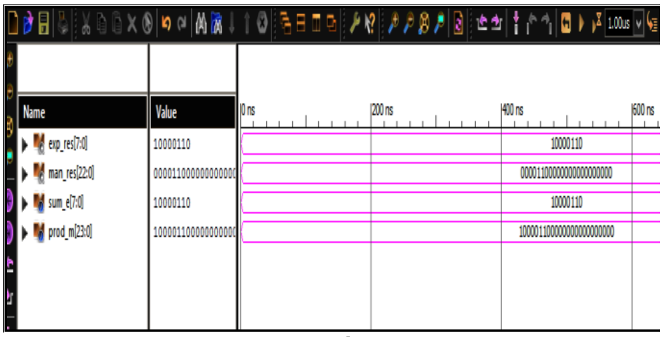

Figure 5 : Normalizer Output

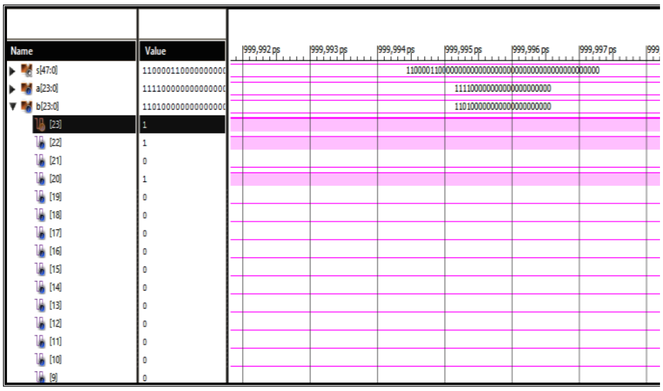

Figure 6 : Unsigned Multiplier Output

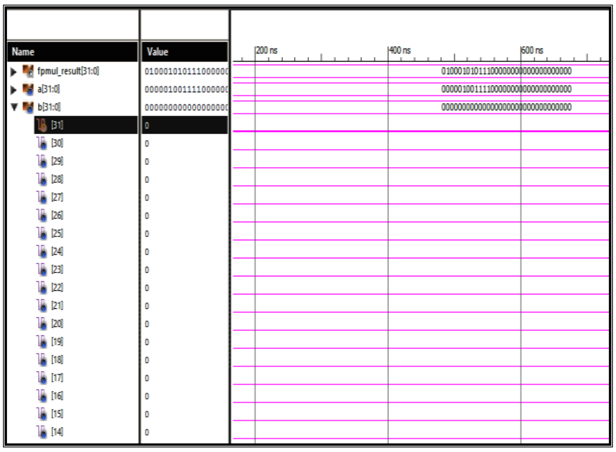

Figure 7 : Single Precision Floating point Multiplier Output (60×-6.5)

DESIGN SUMMARY

fp_mul Project Status (03/15/2013 - 18:33:34)			
Project File:	FP_multiplier.xise	**Parser Errors:**	X 3 Errors
Module Name:	fp_mul	**Implementation State:**	Synthesized
Target Device:	xc3s100e-4tq144	**• Errors:**	No Errors
Product Version:	ISE 14.2	**• Warnings:**	6 Warnings (0 new)
Design Goal:	Balanced	**• Routing Results:**	
Design Strategy:	Xilinx Default (unlocked)	**• Timing Constraints:**	
Environment:	System Settings	**• Final Timing Score:**	

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	649	960	67%
Number of 4 input LUTs	1128	1920	58%
Number of bonded IOBs	96	108	88%

Detailed Reports					[-]
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**
Synthesis Report	Current	Tue Apr 1 14:15:01 2014	0	6 Warnings (0 new)	0
Translation Report					
Map Report					
Place and Route Report					
Power Report					

IV. CONCLUSION AND FUTURE WORK

This paper presents an implementation of a floating point multiplier that supports the IEEE 754-2008 binary interchange format. A methodology for estimating the power and speed has been developed. This Pipelined vectorized floating point multiplier supporting FP16, FP32, FP64 input data and reduces the area, power, latency and increases throughput. Precision can be implemented by taking the 128 bit input operands.

Register Transfer Logic has developed for Double precision Floating Point Multiplier further simulation results can be implemented. The performance of the Floating point multiplier can be increased by taking the 256 bit input bus instead of the 128 bit bus. The throughput and area optimization can be improved by using more general significand multipliers and exponent adders. Two 53 bit multipliers and two 24 bit multipliers are used to compute the significands of all supported Floating point formats.

V. REFERENCES

[1] Alok Baluni,Farhad Merchant ,s.k.nandy,s.Balakrishnan,"a Fully Pipelined Modular Multiple Precision Floating Point Multiplier With Vector Support" 2011 international symposium on Electronic system design(ISED)PP.45-50.

[2] IEEE,IEEE Standard for Binary Floating-Point Arithmetic.IEEE.1985.

[3] K.Manopolous,D.Reises,V.A.Chouiaras "An Efficient Multiple Precision Floating Point Multiplier"2011 IEEE.PP 153-156.

[4] E.Stenersen,"Vectorized 256-bit input fp16/fp32/fp64 floating point multiplier " Norwegain University of Science and Technology,2007.

[5] I. Koren, Computer Arithmetic Algorithms. Natick, MA, USA: A. K.Peters, Ltd., 2001.

[6] S. T. Oskuli, Design of Low-Power Reduction-Trees in Parallel multipliers PhD thesis, Norwegian University of Science and Technology,2008.

[7] G. Even and P.-M. Seidel, "A comparison of three rounding algorithms for IEEE floating-point multiplication," IEEE Trans. Comput., vol. 49, no. 7, pp. 638–650,

[8] N. T. Quach , N. Takagi, and M. J. Flynn, "Systematic IEEE rounding method for high Speed floating-point multipliers," IEEE Trans. Very Large Scale Integr. Syst., vol. 12, no. 5, pp. 511–521, 2004.

[9] L. Wanhammar, DSP Integrated Circuits. Academic Press, 1999.

T. GOVINDA RAO pursing PhD in Pondicherry Central University. He is an Assistant Professor in the Department of Electronics and Communication engineering, GMRIT, Rajam. His current research interests include the areas of Wireless Sensor Networks, very large scale integration (VLSI) testing and fault-tolerant computing, video coding techniques, and Architectures design.

D.ARUN KUMAR received M.Tech from Andhra University, Visakhapatnam. He is an Assistant Professor in the Department of Electronics and Communication engineering, GMRIT, Rajam. His current research interests