An Efficient Implementation of Floating Point Multiplier

Mohamed Al-Ashrafy
Mentor Graphics
Cairo, Egypt
Mohamed_Samy@Mentor.com

Ashraf Salem
Mentor Graphics
Cairo, Egypt
Ashraf_Salem@Mentor.com

Wagdy Anis
Communications and Electronics Engineering
Ain Shams University
Cairo, Egypt
Wagdy4451@yahoo.com

Abstract—**In this paper we describe an efficient implementation of an IEEE 754 single precision floating point multiplier targeted for Xilinx Virtex-5 FPGA. VHDL is used to implement a technology-independent pipelined design. The multiplier implementation 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. With latency of three clock cycles the design achieves 301 MFLOPs. The multiplier was verified against Xilinx floating point multiplier core.**

Keywords-floating point; multiplication; FPGA; CAD design flow

I. INTRODUCTION

Floating point numbers are one possible way of representing real numbers in binary format; the IEEE 754 [1] standard presents two different floating point formats, Binary interchange format and Decimal interchange format. Multiplying floating point numbers is a critical requirement for DSP applications involving large dynamic range. This paper focuses only on single precision normalized binary interchange format. Fig. 1 shows the IEEE 754 single precision binary format representation; 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 what is called the significand[1]. 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 (1)

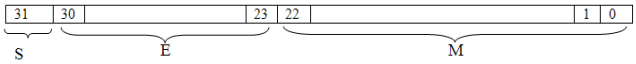

Figure 1. IEEE single precision floating point format

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

Where $M = m_{22}2^{-1} + m_{21}2^{-2} + m_{20}2^{-3} + \ldots + m_1 2^{-22} + m_0 2^{-23}$;

$\underline{Bias} = 127$.

[1] Significand is the mantissa with an extra MSB bit.

This research has been supported by Mentor Graphics.

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, and 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).

Floating-point implementation on FPGAs has been the interest of many researchers. In [2], an IEEE 754 single precision pipelined floating point multiplier was implemented on multiple FPGAs (4 Actel A1280). In [3], a custom 16/18 bit three stage pipelined floating point multiplier that doesn't support rounding modes was implemented. In [4], a single precision floating point multiplier that doesn't support rounding modes was implemented using a digit-serial multiplier: using the Altera FLEX 8000 it achieved 2.3 MFlops. In [5], a parameterizable floating point multiplier was implemented using the software-like language Handel-C, using the Xilinx XCV1000 FPGA; a five stages pipelined multiplier achieved 28MFlops. In [6], a latency optimized floating point unit using the primitives of Xilinx Virtex II FPGA was implemented with a latency of 4 clock cycles. The multiplier reached a maximum clock frequency of 100 MHz.

II. FLOATING POINT MULTIPLICATION ALGORITHM

As stated in the introduction, normalized floating point numbers have the form of $Z= (-1^{S}) * 2^{(E - \underline{Bias})} * (1.M)$. 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. $(E_1 + E_2 - \underline{Bias})$
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

Consider a floating point representation similar to the IEEE 754 single precision floating point format, but with a reduced

978-1-4577-0069-9/11/$26.00 ©2011 IEEE

number of mantissa bits (only 4) while still retaining the hidden '1' bit for normalized numbers:

A = 0 10000100 0100 = 40, B = 1 10000001 1110 = -7.5

To multiply A and B

1. Multiply significand:

```
   1.0100
 × 1.1110
   00000
  10100
 10100
10100
10100
1001011000
```

2. Place the decimal point: `10.01011000`

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

$$E_A + E_B = E_{A\text{-true}} + E_{B\text{-true}} + 2\ \text{bias}$$

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

```
 100000101
-  01111111
  10000110
```

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

   `1 10000110 10.01011000`

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 10.01011000` (before normalizing)

   `1 10000111 1.001011000` (normalized)

   The result is (without the hidden bit):

   `1 10000111 00101100`

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 0010`.

In this paper we present a floating point multiplier in which rounding support isn't 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 if the multiplier is connected directly to an adder in a MAC unit. Fig. 2 shows the multiplier structure; Exponents addition, Significand multiplication, and Result's sign calculation are independent and are done in parallel. The significand 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 downto 0) and the decimal point is located between bits 46 and 45 in the IP. The following sections detail each block of the floating point multiplier.

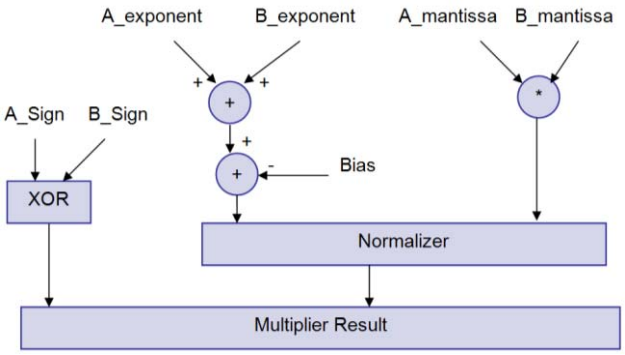

Figure 2. Floating point multiplier block diagram

III. HARDWARE OF FLOATING POINT MULTIPLIER

A. *Sign bit calculation*

Multiplying two numbers results in a negative sign number iff one of the multiplied numbers is of a negative value. By the aid of a truth table we find that this can be obtained by XORing the sign of two inputs.

B. *Unsigned Adder (for exponent addition)*

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. 3 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).

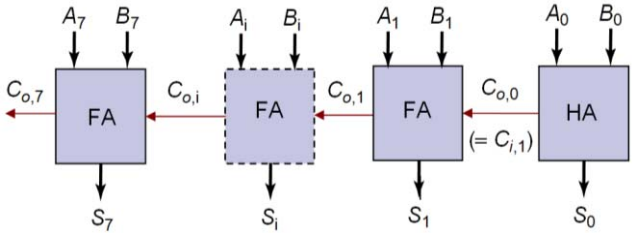

Figure 3. Ripple Carry Adder

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. 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 ($127|_{10} = 001111111|_2$). Table I shows the truth table for a 1-bit subtractor with the input T equal to 1 which we will call "one subtractor (OS)"

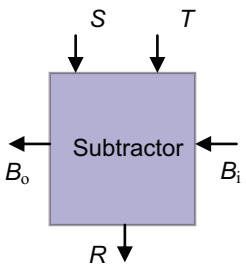

TABLE I. 1-BIT SUBTRACTOR WITH THE INPUT T = 1

S	T	B_i	Difference(R)	B_o
0	1	0	1	1
1	1	0	0	0
0	1	1	0	1
1	1	1	1	1

The Boolean equations (2) and (3) represent this subtractor:

$$Difference\ (R) = \overline{S \oplus B_i} \quad (2)$$

$$Borrow_{out}(B_o) = \overline{S} + B_i \quad (3)$$

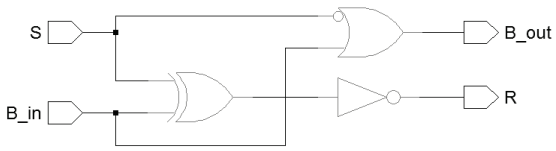

Figure 4. 1-bit subtractor with the input T = 1

Table II shows the truth table for a 1-bit subtractor with the input T equal to 0 which we will call "zero subtractor (ZS)"

TABLE II. 1-BIT SUBTRACTOR WITH THE INPUT T = 0

S	T	B_i	Difference(R)	B_o
0	0	0	0	0
1	0	0	1	0
0	0	1	1	1
1	0	1	0	0

The Boolean equations (4) and (5) represent this subtractor:

$$Difference\ (R) = S \oplus B_i \quad (4)$$

$$Borrow_{out}(B_o) = \overline{S} \cdot B_i \quad (5)$$

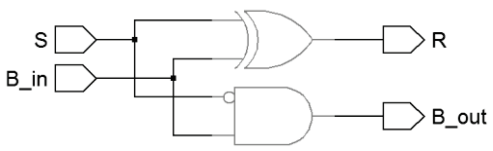

Figure 5. 1-bit subtractor with the input T = 0

Fig. 6 shows 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.

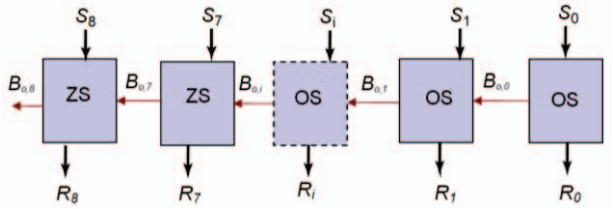

Figure 6. Ripple Borrow Subtractor

C. Unsigned Multiplier (for significand multiplication)

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 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). Partial products are made by ANDing the inputs together and passing them to the appropriate adder.

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.

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 is shown in Fig. 7 and it has the following stages:

1- The first stage consists of three half adders.
2- Two middle stages; each consists of three full adders.
3- 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 in bold in Fig. 7

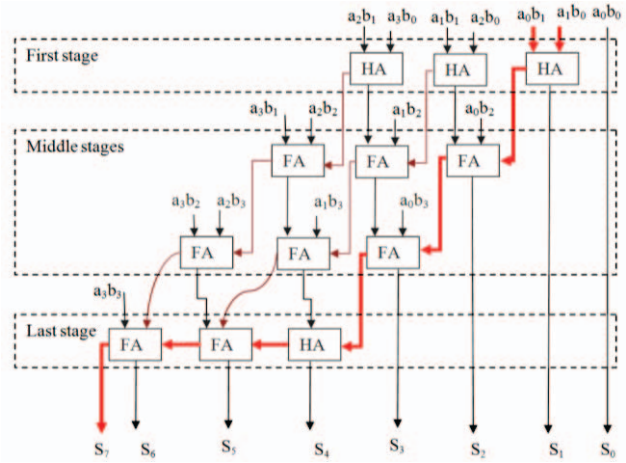

Figure 7. 4x4 bit Carry Save multiplier

In Fig. 7:

1- Partial product: $a_i b_j = a_i$ and b_j
2- HA: half adder
3- FA: full adder

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 (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.

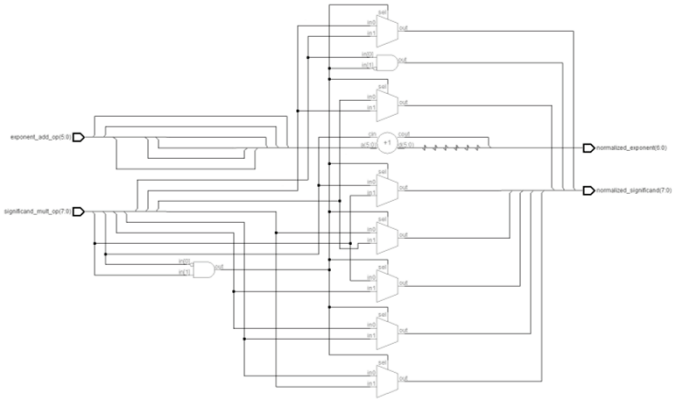

Figure 8. Simplified Normalizer logic

IV. 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 E_1 and E_2 are the exponents of the two numbers A and B respectively; the result's exponent is calculated by (6)

$$E_{result} = E_1 + E_2 - 127 \qquad (6)$$

E_1 and E_2 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 III summarizes the E_{result} different values and the effect of normalization on it.

TABLE III. Normalization Effect on Result's Exponent and Overflow/Underflow Detection

E_{result}	Category	Comments
$-125 \le E_{result} < 0$	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
$255 \le E_{result}$	Overflow	Can't be compensated

V. Pipelining the Multiplier

In order to enhance the performance of the multiplier, three pipelining stages are used to divide the critical path thus increasing the maximum operating frequency of the multiplier. The pipelining stages are imbedded at the following locations:

1. In the middle of the significand multiplier, and in the middle of the exponent adder (before the bias subtraction).
2. After the significand multiplier, and after the exponent adder.
3. At the floating point multiplier outputs (sign, exponent and mantissa bits).

Fig. 9 shows the pipelining stages as dotted lines.

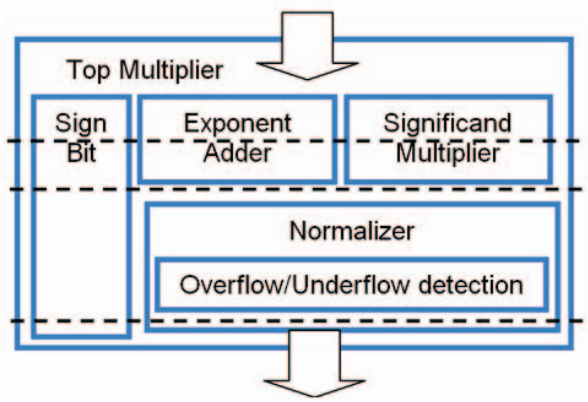

Figure 9. Floating point multiplier with pipelined stages

Three pipelining stages mean that there is latency in the output by three clocks. The synthesis tool "retiming" option was used so that the synthesizer uses its optimization logic to better place the pipelining registers across the critical path.

VI. Implementation and Testing

The whole multiplier (top unit) was tested against the Xilinx floating point multiplier core generated by Xilinx coregen. Xilinx core was customized to have two flags to indicate overflow and underflow, and to have a maximum latency of three cycles. Xilinx core implements the "round to nearest" rounding mode.

A testbench is used to generate the stimulus and applies it to the implemented floating point multiplier and to the Xilinx core then compares the results. The floating point multiplier code was also checked using DesignChecker [7]. DesignChecker is a linting tool which helps in filtering design issues like gated clocks, unused/undriven logic, and combinational loops. The design was synthesized using Precision synthesis tool [8] targeting Xilinx Virtex-5 5VFX200TFF1738 with a timing constraint of 300MHz. Post synthesis and place and route simulations were made to ensure the design functionality after synthesis and place and route. Table IV shows the resources and frequency of the implemented floating point multiplier and Xilinx core.

TABLE IV. Area and Frequency Comparison between the Implemented Floating Point Multiplier and Xilinx Core

	Our Floating Point Multiplier	**Xilinx Core**
Function Generators	1263	765
CLB Slices	604	266
DFF	293	241
Max Frequency	301.114 MHz	221.484 MHz

The area of Xilinx core is less than the implemented floating point multiplier because the latter doesn't truncate/round the 48 bits result of the mantissa multiplier which is reflected in the amount of function generators and registers used to perform operations on the extra bits; also the speed of Xilinx core is affected by the fact that it implements the round to nearest rounding mode.

VII. Conclusions and Future Work

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.

Acknowledgment

Authors would like to thank Randa Hashem for her invaluable support and contribution.

References

[1] IEEE 754-2008, IEEE Standard for Floating-Point Arithmetic, 2008.

[2] 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.

[3] 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.

[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] A. Jaenicke and W. Luk, "Parameterized Floating-Point Arithmetic on FPGAs", Proc. of IEEE ICASSP, 2001, vol. 2, pp. 897-900.

[6] 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

[7] "DesignChecker User Guide", HDL Designer Series 2010.2a, Mentor Graphics, 2010

[8] "Precision® Synthesis User's Manual", Precision RTL plus 2010a update 2, Mentor Graphics, 2010.

[9] Patterson, D. & Hennessy, J. (2005), Computer Organization and Design: The Hardware/software Interface , Morgan Kaufmann .

[10] John G. Proakis and Dimitris G. Manolakis (1996), "Digital Signal Processing: Principles,. Algorithms and Applications", Third Edition.