FPGA Design of Pipelined 32-bit Floating Point Multiplier

[1]Shaifali, [2]Sakshi
[1,2]Thapar University, Patiala, India

Abstract
An architecture for a fast 32-bit floating point multiplier compliant with the single precision IEEE 754-2008 standard has been proposed in this paper. Verilog is used to implement a technology-independent pipelined design. Floating Point Multiplier is synthesized and targeted for Xilinx Spartan-3E FPGA.

Keywords-Floating point, IEEE-754 Standard, Multiplication, Synthesis, Verilog, Modified Booth's Algorithm, Pipeline

I. INTRODUCTION

Multipliers are key components of many high performance systems such as FIR filters, microprocessors, digital signal processors, etc. Multiplication based operations such as multiply and accumulate(MAC) and inner product are among some of the frequently used computation- intensive arithmetic functions currently implemented in many digital signal processing (DSP) applications such as convolution, fast fourier transform(FFT), filtering and in microprocessors in its arithmetic and logic unit. Since multiplication dominates the execution time of most DSP algorithms, so there is a need of high speed multiplier.

Floating point is a way to represent numbers and do arithmetic in computing machines, ranging from simple calculators to computers. The term floating point is derived from the fact that there is no fixed number of digits before and after the decimal point; that is, the decimal point can float. In general, floating-point representations are slower than fixed-point representations, but they can handle a larger range of numbers.

Floating point multiplication is much like integer multiplication. Because floating-point numbers are stored in sign-magnitude form, the multiplier needs only to deal with unsigned integer numbers and normalization. Integer multiplication using modified booth's algorithm and carry save adder is one way to increase the speed. Because modified Booth algorithm reduces the number of partial products to be generated and is known as the fastest multiplication algorithm and many researches on the multiplier architectures including array, parallel and pipelined multipliers have been pursued which shows that pipelining is the most widely used technique to reduce the propagation delays of digital circuits.

In this paper an architecture for a fast floating point multiplier compliant with the single precision IEEE 754 standard has been proposed. To attain a generic design, Verilog hardware description language was used for design entry of the entire multiplier unit as it presents a tremendous productivity improvement for circuit designers and descriptions of large circuits can be written in a relatively compact and concise form[1].

II. IEEE-754 FLOATING POINT REPRESENTATION

Over the years, several different floating-point representations have been used in computers; however, for the last ten years the most commonly encountered representation is that defined by the IEEE Standard for Floating-Point Arithmetic (IEEE 754) [3]. It is a technical standard established by the Institute of Electrical and Electronics Engineers (IEEE) and the most widely used standard for floating-point computation, followed by many hardware and software implementations. Single precision representation occupies 32 bits: a sign bit, 8 bits for exponent and 23 for the mantissa.

The most significant bit starts from the left. The three basic components are the sign, exponent, and mantissa. The storage layout for single-precision is show in figure 1:

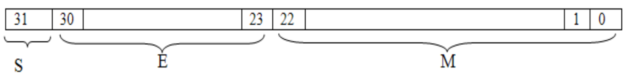


Figure 1: Single Precision Format for Floating Point Numbers[2]

The number represented by the single-precision format is:
Value = $(-1)^s * 2^{E-127} * 1.M$(normalized) when E>0
Where M = $m_{22}\,2^{-1} + m_{21}\,2^{-2} + m_{20}\,2^{-3} + \ldots + m_1\,2^{-22} + m_0\,2^{-23}$;
S= sign (0 is positive, 1 is negative)
E =biased exponent; $E_{max} = 255$; $E_{min} = 0$. E=255 and E=0 are used to represent special values.
e =unbiased exponent; e = E-127(bias)

A bias of 127 is added to the actual exponent to make negative exponents possible without using a sign bit. So for example if the value 100 is stored in the exponent placeholder, the exponent is actually -27(100 – 127). In a normalized mantissa, the digit 1 always appears to the left of the decimal point. In fact, the leading 1 is omitted from the mantissa in the IEEE storage format because it is redundant.

Table 1 shows that this format uses 8 bits for exponent with a bias of 127. Twenty-three bits are used as significant (mantissa) with one hidden bit, which will always be concatenated as 1 while being operated.

Table 1: Features of the ANSI/IEEE Standard Floating-Point Representation[4]

Feature	Single
Word length, bits	32
Significant bits	23+1(hidden)
Exponent Bits	8
Exponent Bias	127
Zero	(± 0) e + bias = 0, f = 0
Denormal	e + bias = 0, f ≠0
Infinity	e + bias = 255, f = 0
Not-a-Number (NAN)	e + bias = 255, f ≠ 0
Overflow	e+bias >= 255
Underflow	-125 <=e+bias< 0

III. Floating Point Multiplication Algorithm

Normalized floating point numbers have the form of $Z= (-1^S) * 2^{(E - Bias)} * (1.M)$. To multiply two floating point numbers the following is done as shown in figure 2[2]:

a) Multiplying the significand; i.e. (1.M1*1.M2).
b) Placing the decimal point in the result.
c) Adding the exponents; i.e. (E1 + E2 – *Bias*).
d) Obtaining the sign; i.e. s1 xor s2.
e) Normalizing the result; i.e. obtaining 1 at the MSB of the results' significand.
f) Checking for underflow/overflow occurrence.

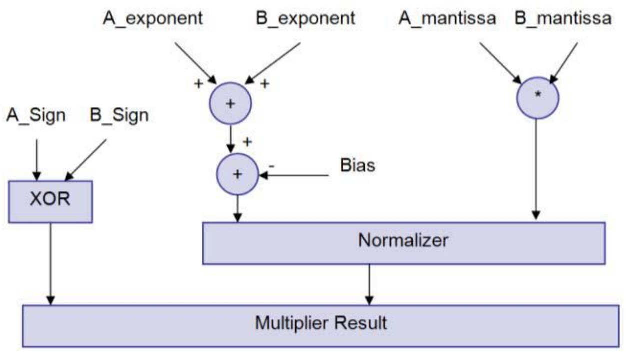


Figure 2: Floating Point Multiplier block diagram[2]

IV. MULTIPLICATION

This is the most important stage, product of the mantissa bits is calculated. The multiplication of mantissa bits is performed in the following stages.

A. Generation of Partial Products

The Booth multiplier makes use of Booth encoding algorithm in order to reduce the number of partial products by considering certain number of bits of the multiplier at a time, thereby achieving a speed advantage over other multiplier architectures. This algorithm is valid for both signed and unsigned numbers. It can handle signed binary multiplication by using 2's complement representation.

For generating the partial products Radix-8 Modified Booth's Algorithm is used. Since the multiplier and multiplicand comprises of 24 bits, this algorithm will generate 8 partial products.

The shortcoming of Radix 2 Booth algorithm is that it becomes inefficient when there are isolated 1's. For example, 001010101(decimal 85) gets reduced to 01-11-11-11-1(decimal 85), requiring eight instead of four operations.001010101(0) recoded as 011111111, requiring 8 instead of 4 operations.

Radix-8 Modified Booth's Algorithm

Recoding extended to 3 bits at a time - overlapping groups of 4 bits each. Radix-8 recoding applies the same algorithm as radix-4, but now we take quartets of bits instead of triplets. Consequently, a multiplier based on this radix-8 scheme generates fewer partial products than a radix-4 multiplier, but the computation of each partial product is more complex[7]. In particular, a partial product corresponding to an encoding x=$\underline{+}$3 requires the computation of 3x, and therefore a full addition. Each quartet is codified as a signed-digit using the table 2:

Table 2: Recoding in Booth Radix-8 Algorithm [7]

Quartet value	Signed-digit value	Quartet value	Signed-digit value
0000	0	1000	-4
0001	+1	1001	-3
0010	+1	1010	-3
0011	+2	1011	-2
0100	+2	1100	-2
0101	+3	1101	-1
0110	+3	1110	-1
0111	+4	1111	0

Synthesis Results on FPGA

Table 3 shows the synthesis report on Xilinx for generation of partial products.

Table 3: Synthesis Report of Partial Product Generation

	Spartan 3E xc3s500E	Virtex 4 xc4vlx15
No. of Slices	2003/4656 (43%)	1913/6144 (31%)
No. of LUTs	3653/9312 (39%)	3794/12288 (30%)
Minimum Period	10.344ns	6.537ns
Maximum Frequency	96.676MHz	152.986MHz

B. Partial Product Reduction.

8 Partial products are generated using Radix-8 Modified Booth's Algorithm. They are reduced using 4:2 compressors.

Carry Save Adder

A Carry-Save Adder is just a set of one-bit full adders, without any carry-chaining. The most important application of a carry-save adder is to calculate the partial products in integer multiplication. 4:2 compressors are used as carry save adders. The 4:2 compressor structure actually compresses five partial products bits into three. The architecture is connected in such a way that four of the inputs are coming from the same bit position of the weight j while one bit is fed from the neighboring position j-1(known as carry-in). The outputs of 4:2 compressor consists of one bit in the position j and two bits in the position j+1.

A 4:2 compressor can also be built using 3:2 compressors. It consists of two 3:2 compressors (full adders) in series and involves a critical path of 4 XOR delays as shown in Figure 3[8]. The output Cout, being independent of the input Cin accelerates the carry save summation of the partial products. 4:2 compressor is made from 2 full adders. The final carry is saved and hence is called carry save adder. The delay of 4:2 compressor is equal that of 4 xor gates.

Initially two 4:2 compressors are used to reduce each 4 partial products pair to generate the pair of sum and carry. Then these final 4 partial products generated from above two 4:2 compressors are further reduced to generate final sum and carry. The final sum and carry are added in next Carry Propagate adder.

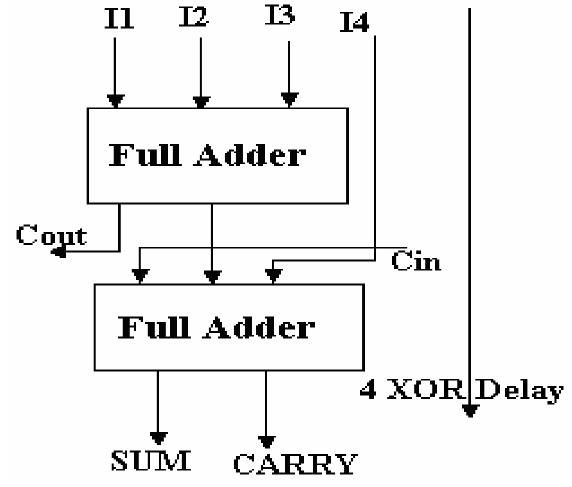


Figure 3: 4:2 Compressor Design using Full Adders

Synthesis Results on FPGA

Table 4 shows the synthesis report on Xilinx for Partial Product Addition.

Table 4: Synthesis Report of Partial Product Addition

	Spartan 3E xc3s500E	Virtex 4 xc4vlx15
No. of Slices	501/4656 (10%)	502/6144 (8%)
No. of LUTs	706/ 9312 (7%)	702/12288 (5%)
Minimum Period	5.184ns	3.075ns
Maximum Frequency	192.905MHz	325.256MHz

C. Final stage Carry Propagate Adder

Further the partial products generated through carry save adders are further reduced by using Ripple Carry Adder.

Ripple Carry Adder

Ripple Carry Adder is used to obtain the final sum and the output carry by adding the partial products from the carry save adders. It creates a logical circuit using multiple full adders to add N-bit numbers. Each full adder inputs a C_{in}, which is the C_{out} of the previous adder. This kind of adder is a ripple carry adder, since each carry bit "ripples" to the next full adder. The 48-bit sum and carry outputs obtained from the partial product accumulator are added in the final stage adder to give the product of the mantissas.

As shown in Figure 4 a ripple carry adder is a chain of cascaded full adders and one half adder; each full adder has three inputs (A, B, Ci) and two outputs (S, Co). The carry out (Co) of each adder is fed to the next full adder (i.e each carry bit "ripples" to the next full adder).

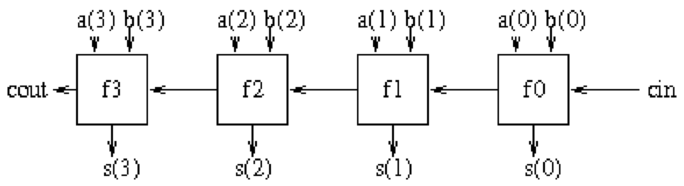


Figure 4: Ripple Carry Adder[2]

Synthesis Results on FPGA

Table 5 shows the synthesis report on Xilinx for Final 48-bit Stage Carry Propagate Adder.

Table 5: Synthesis Report of 48-bit Ripple Carry Adder

	Spartan 3E xc3s500E	Virtex 4 xc4vlx15
No. of Slices	150/4656 (3%)	146/6144 (2%)
No. of LUTs	246/9312 (2%)	250/12288 (2%)
Minimum Period	8.235ns	5.206ns
Maximum Frequency	121.438MHz	192.095MHz

V. NORMALIZATION

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

a) 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.
b) If the leading one is at bit 47 then the intermediate product is shifted to the right and the exponent is incremented by 1[2].

VI. PIPELINING

A pipeline is a set of data processing elements connected in series, so that the output of one element is the input of the next one. It is divided into segments and each segment can execute its operation concurrently with the other segments. When a segment completes an operation, it passes the result to the next segment in the pipeline and fetches the next operation from the preceding segment. The final results of each instruction emerge at the end of the pipeline in rapid succession.

The pipeline technique is widely used to improve the performance of digital circuits. As the number of pipeline stages is increased, the path delays of each stage are decreased and the overall performance of the circuit is improved.

a) 5-Stage Pipelining

In order to enhance the performance of the multiplier, five pipelining stages are used to divide the critical path thus increasing the maximum operating frequency of the multiplier. Five pipelining stages mean that there is latency in the output by five clocks[8]. The pipelining stages are embedded at the following locations:

i. After the Pre-processing of the Multiplicand and Multiplier.
ii. After the Exponent Adder and Generation of 8 Partial Products.
iii. After subtracting the Bias and Compressing the partial Products to 4.
iv. After Compressing the Partial Products to 2.
v. After Normalization and Final Carry Propagate Adder.

Table 6 shows the synthesis result. Comparing with 3-stage pipelined multiplier, the frequency is increased as the pipeline stages are increased.

Synthesis Results on FPGA

Table 6: Synthesis Report of 5-Stage Floating Point Multiplier

	Spartan 3E xc3s500E	Virtex 4 xc4vlx15
No. of Slices	2045/4656 (43%)	2067/6144 (33%)
No. of LUTs	3893/9312 (41%)	3900/12288 (31%)
Minimum Period	12.154ns	6.532ns
Maximum Frequency	82.279MHz	153.081MHz

Figure 5 shows the various pipeline stages in the Multiplier.

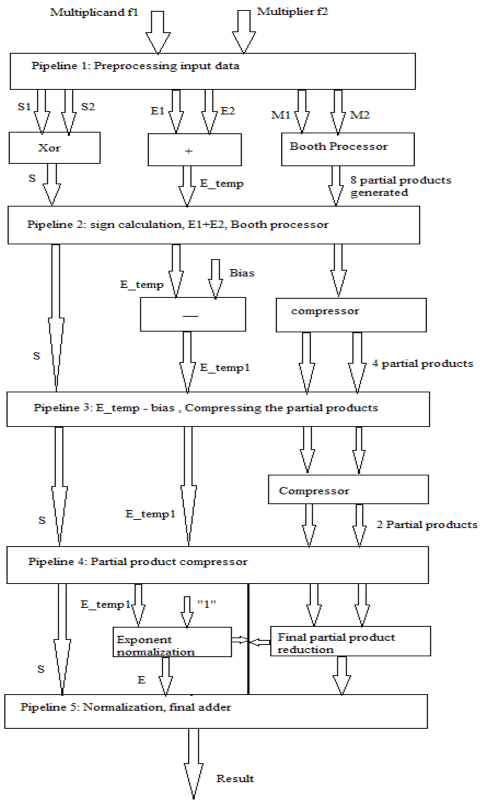
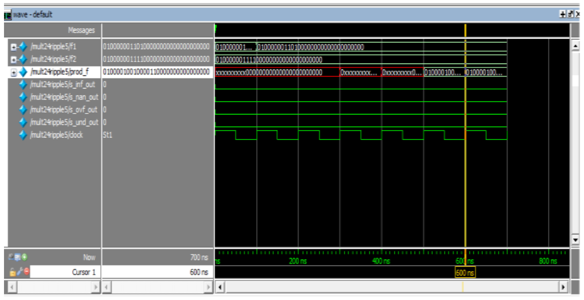

Multiplicand f1
Multiplier f2
Pipeline 1: Preprocessing input data
S1
S2
E1
E2
M1
M2
Xor
+
Booth Processor
S
E_temp
8 partial products generated
Pipeline 2: sign calculation, E1+E2, Booth processor
E_temp
Bias
—
compressor
S
E_temp1
4 partial products
Pipeline 3: E_temp - bias , Compressing the partial products
Compressor
S
E_temp1
2 Partial products
Pipeline 4: Partial product compressor
E_temp1
"1"
Exponent normalization
Final partial product reduction
S
E
Pipeline 5: Normalization, final adder
Result

Figure 5: Floating Point Multiplier with 5 Pipeline Stages

VII. SIMULATION RESULTS OF 5-STAGE FLOATING POINT MULTIPLIER

Figure 6 shows the functional simulation of the multiplier on Modelsim 6.3f. It takes two 32-bit floating point numbers and gives their resultant product of 32-bits. After 5 Clock cycles result is obtained in every clock cycle.



```
wave - default
Messages
/mult24ripple5/f1        01000000110100000000000000000000
/mult24ripple5/f2        01000000111100000000000000000000
/mult24ripple5/prod_f    01000010010000110000000000000000
/mult24ripple5/s_inf_out 0
/mult24ripple5/s_nan_out 0
/mult24ripple5/s_ovf_out 0
/mult24ripple5/s_und_out 0
/mult24ripple5/clock     St1
Now       700 ns
Cursor 1  600 ns
```

Figure 6: Functional Simulation of Multiplier

VIII. CONCLUSION

A new hardware implementation of a high speed floating point multiplier based on the IEEE-754 single precision format is developed based on pipeline technique. The modules are written in Verilog HDL to optimize implementation on FPGA. The multiplier doesn't implement rounding and just presents the significand multiplication result as is (48 bits). The design has five pipelining stages and after implementation on a Xilinx Spartan 3E FPGA it achieves 82.279MHz.

REFERENCES

[1]. Anna Jain, Baisakhy Dash, Ajit Kumar Panda, Member, IEEE, Muchharla Suresh, Member, IEEE, "FPGA design of a fast 32-bit floating point multiplier unit," IEEE 15-16 March 2012.

[2]. Al-Ashrafy, M. Salem, A. Anis, W., Mentor Graphics, Cairo, Egypt, "An efficient implementation of floating point multiplier," IEEE 24-26 April 2011.

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

[4]. Dhiraj Sangwan& Mahesh K. Yadav,"Design and Implementation of Adder/Subtractor and Multiplication Units for Floating-Point Arithmetic, International Journal of Electronics Engineering," 2(1), 2010, pp. 197-203.

[5]. Soojin Kim and Kyeongsoon Cho, "Design of High-speed Modified Booth Multipliers Operating at GHz Ranges," World Academy of Science, Engineering and Technology 61 2010.

[6]. Young-Ho Seo, *Member*, *IEEE*, and Dong-Wook Kim, *Member, IEEE, "*A New VLSI Architecture of Parallel Multiplier–Accumulator Based on Radix-2 Modified Booth Algorithm," IEEE transactions on very large scale integration (vlsi) systems, vol. 18, no. 2, February 2010.

[7]. Behrooz Parhami, Computer Arithmetic, Algorithms and Hardware Design second edition.

[8]. Gong Renxi, Zhang Hainan, Meng Xiaobi,Gong Wenying,"Hardware Implementation of a High Speed Floating Point Multiplier Based on FPGA,"2009 4th International Conference on Computer Science & Education.

[9]. Rashmi Sharma, Anshuman Singh, Performance Analysis and Optimization of Low power SRAM, International Journal of Computational Engineering and Management IJCEM, Vol. 16 Issue 3, May 2013.

[10]. 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.