Spartan 3E Synthesizable FPGA Based Floating-Point Arithmetic Unit

Yedukondala Rao Veeranki[#1], R. Nakkeeran[*2]
[#]Department of Electronics Engineering, Pondicherry University
Puducherry, India.
[*]Department of Electronics Engineering, Pondicherry University
Puducherry, India

Abstract— **Floating point operations are hard to implement on Field Programmable Gate Arrays (FPGA) because of the complexity of algorithms is more. Then again, many scientific applications require floating point arithmetic because of high accuracy in their calculations. Therefore, an attempt is made to explore FPGA implementations in Institute of Electrical and Electronics Engineers (IEEE) -754 standard floating-point numbers. Many algorithms depend on floating point arithmetic because floating point representation supports huge range. In this paper an efficient implementation of an IEEE 754 single precision floating point arithmetic unit is designed in Xilinx SPARTAN 3E FPGA. VHDL environment is performed for floating point arithmetic unit design using pipelining, which provides high performance. Pipelining is used to execute multiple instructions simultaneously. In top-down design approach, four arithmetic modules, addition/ subtraction, multiplication and division are combined to form a floating point arithmetic unit. Synthesis and simulation results are obtained** ***by using Xilinx13.1i platform.***

Keywords— ALU - Arithmetic Logic Unit; Top-Down design; floating point; FPGA; Pipelined Architecture.

I. INTRODUCTION

By using Field Programmable Gate Arrays (FPGAs) the designers can build any logic device in hardware quickly and easily. The programmability and flexibility of FPGAs make them ideal for prototyping, quick time-to-market applications, one-off implementations, and customized hardware. They are especially valuable in applications when a custom circuit is required, but the production volume does not justify the costs and time of fabricating them on application-specific integrated circuits (ASICs). Advances in process technology have led to dramatic increase in FPGAs densities and speeds. FPGAs are now becoming more suitable for supporting designs with dense computations and high operating frequencies. Consequently, FPGAs are becoming more suitable for supporting high speed floating point arithmetic units. Floating point units are widely used in digital applications such as digital signal processing, digital image processing and multimedia. In conventional floating point units, the most frequently used floating point operations are multiplication and addition/subtraction counting for more than 94% of all floating point instructions. Hence the employment of highly performing divider, multiplier and adder/subtractor modules is of high importance.

Floating-point addition is the most complex operation in a floating-point arithmetic and consists of many variable latency- and area dependent sub-operations. In floating-point addition implementations, latency is the primary performance bottleneck. Much work has been done to improve the overall latency of floating-point adders. Various algorithms and design approaches have been developed by the VLSI community in the last two decades. For the most part, digital design companies around the globe have focused on FPGA design instead of ASICs because of their effective time to market, adaptability, and, most importantly, low cost. The floating-point unit is one of the most important custom applications needed in most hardware designs, as it adds accuracy, robustness to quantization errors, and ease of use. There are many commercial products for floating-point addition that can be used in custom designs in FPGAs but cannot be modified for specific design qualities like throughput, latency, and area. Much work has also been done to design custom floating-point adders in FPGAs. Most of this work aims to increase the throughput by means of deep pipelining.

II. DESIGN OF FLOATING POINT ADDER/SUBTRACTOR

The algorithms for addition/subtraction require more complex operations due to the need for operator alignment. Three floating point add/subtract algorithms are briefly introduced in this section: standard, leading-one predictor (LOP), and 2-path. The implementation of these steps defines floating point arithmetic unit latency and area. To illustrate comparisons, we consider the block diagrams of the floating point adder/subtractor shown in Figures 1, 2, 3. Standard floating point addition requires five steps:

1. Exponent difference
2. Pre-shift for mantissa alignment
3. Mantissa addition/subtraction
4. Post-shift for result normalization
5. Rounding

The area-efficient standard floating point adder is shown in Figure 1. The exponents of the two input operands,

ExponentA and ExponentB are fed to the exponent comparator. In the pre-shifter, a new mantissa is created by right shifting the mantissa corresponding to the smaller exponent by the difference of the exponents so that the resulting two mantissas are aligned and can be added. Right shifting is nothing but dividing by power of 2. If the mantissa adder generates a carry output the resulting mantissa is shifted one bit to the right and the exponent is increased by one. The normalizer transforms the mantissa and exponent into normalized format. Result of subtraction may require a massive left shift during normalization. It first uses a Leading-One-Detector (LOD) circuit to locate the position of the most significant one in the mantissa.

Based on the position of the leading one, the resulting mantissa is left-shifted by an amount subsequently deducted from the exponent. In the normalization process if the adder result is too large then it shifts to right (divide by2) and if the adder result is too small then it shifts to right (multiply by 2).Precission is lost when some bits are shifted to right of the right most bit or are thrown. To obtain the accuracy the shifted out bits are also used as G(I), R(round),S(sticky).If G=R=1 then add 1 to the LSB of result. If G=R=0 then no change in result. If G=1 & R=0 then look at S. If S=1 add 1 to LSB and if S=0 round the nearest even i.e. add 1 to LSB if LSB=1.

A. Standard Floating Point Add/Subtract Algorithm

For the standard algorithm, the exponent comparator is implemented with a subtractor and a multiplexer. The comparator requires about $2 \times n$ LUTs, where n is the exponent bit-width. The size of the pre-shifter is about $m \times log(m)$ LUTs, where m is the bit-width of the mantissa. The size of the mantissa adder depends on the adder architecture and sign mode. If a ripple-carry adder is used for an unsigned mantissa, about m LUTs are required. The 752normalizer LOD is nearly the same size as the mantissa adder. The shifter is equal in size to the pre-shifter and the subtractor (SUB) is about the same size as the exponent comparator. Overall, the size of the 752normalizer is about the sum of the sizes of the other three components.

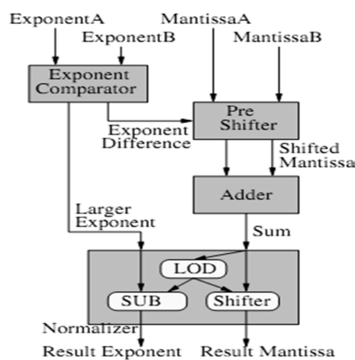

Fig. 1 LOD Algorithm

B. LOP Algorithm

Figure 2 shows a block diagram of a Leading-One-Predictor (LOP) floating point adder. LOP is a technique in which no. of preceding 1's or 0's in the result can be predicted directly from the input operands to within an error of 1-bit, in parallel with addition/subtraction step. The error comes from possible carry-in. It has mainly two purposes first, it detects the bit pattern and second it is used for sticky bit computation. This adder implementation requires more area than a standard adder, but exhibits reduced latency. The primary difference between the adders is the replacement of the leading-one detector (LOD) circuit with a leading-one predictor (LOP) circuit. Since the LOP circuit can be executed in parallel with mantissa addition, overall latency can be reduced.

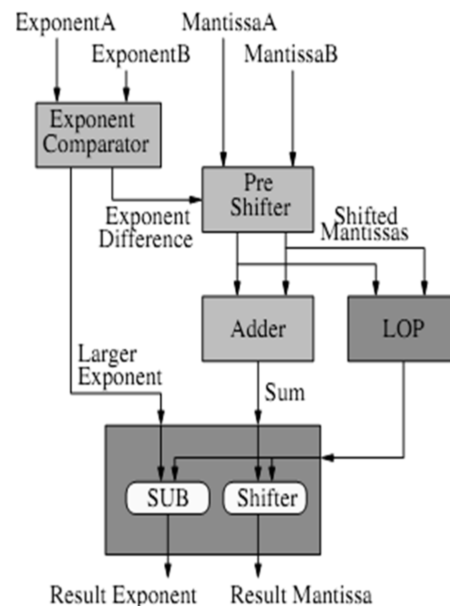

Fig.2 LOP Algorithm

C. FAR and CLOSE Path Algorithm

The 2-path adder, shown in Figure 3, has two parallel data paths. This implementation exhibits the smallest latency of the three adders, due to the elimination of a shifter from the critical path, at the cost of additional mapping area. When the exponents of the two values are

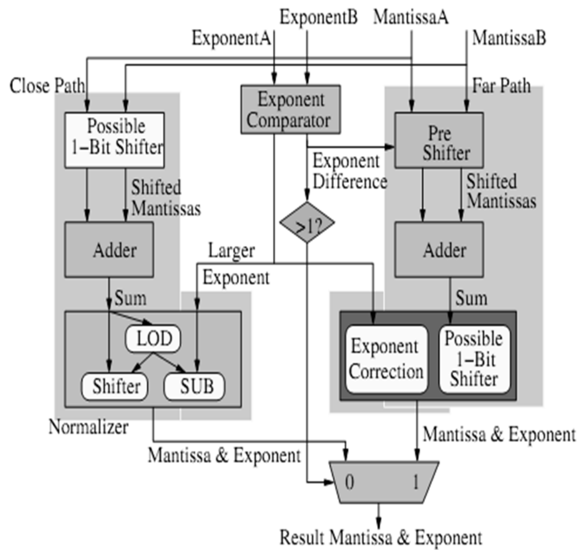

Fig. 3 Two-path Algorithm

larger than 1, the far path, on the right in Figure 3, is taken. Otherwise, the close path on the left is taken. After alignment, one of the mantissas is reduced and shifted by at most one bit. This close path implementation eliminates the preshifter.

III. FLOATING POINT MULTIPLICATION ALGORITHM

Normalized floating point numbers have the form of

Z= (-1S) * 2 (E - *Bias*) * (1.M).

To multiply two floating point numbers the following is done:

1. Multiplying the significand; i.e. (1.M1*1.M2)
2. Placing the decimal point in the result
3. Adding the exponents; i.e. (E1 + E2 – *Bias*)
4. Obtaining the sign; i.e. s1 xor s2
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

Fig.4 shows the data path for a floating-point Multiplication. Only the main parts of the data path are shown for clarity. The prealignment and normalization stages require large shifters. The prealignment stage requires a right shifter that is twice the number of mantissa bits (i.e., 48 bits for single-precision, 106 bits for double-precision) because the bits shifted out have to be maintained to generate the guard, round and sticky bits needed for rounding. The shifter only needs to shift right by up to 24 places for single-precision or 53 places for double-precision.

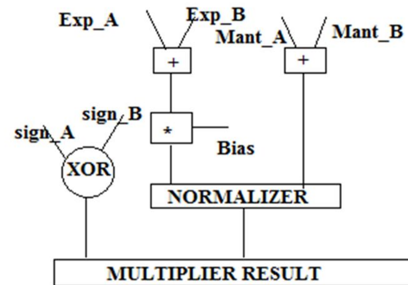

Fig.4 Floating Point Multiplication Algorithm

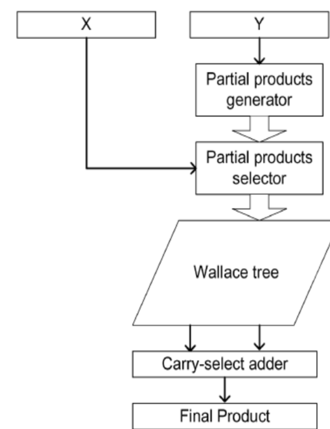

Fig.5 Booth Wallace Multiplier

The normalization stage requires a left shifter equal to the number of mantissa bits plus 1 (to shift in the guard bit), i.e., 25-bits for single-precision and 54-bits for double-precision. If the rounding of the mantissa results in an overflow, the mantissa is shifted right by one and the exponent is incremented. a very wide multiplier—53 53-bit unsigned multiplier for double-precision and 24 24-bit for single-precision. Therefore, an efficient multiplier must be employed.

In this work, we use a Radix-4 modified booth encoded (MBE) Wallace multiplier as shown in Fig. 5, which was based on the designs in. Radix-4 recoding halves the number of partial products, thus reducing the number of levels required in the Wallace tree, which improves performance and reduces area.

IV. FLOATING POINT DIVIDER ALGORITHM

To divide two floating point numbers the following is done:

1. Divide the significands; i.e. (1.M1÷1.M2)
2. Placing the decimal point in the result
3. Subtracting the exponents; i.e. (E1 + E2 – *Bias*)

4. Obtaining the sign; i.e. s1 xor s2

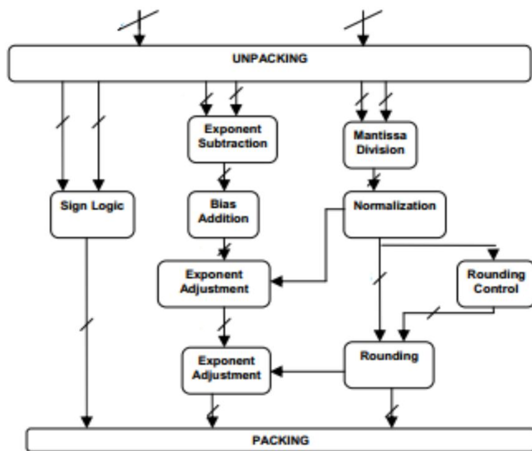

Fig. 6 Floating Point Arithmetic Unit Design

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.

V. ALU DESIGN

The ALU design using VHDL the Specifications for a 16-bit floating-point ALU design are:

i. Input A and B and output result are 32-bit binary floating point.

ii. Operands A and B operate as follows

A (operation) B=results

Operation can be addition (+), subtraction (-), Multiplication (*), division (/)

iii. 'Selection' a 2-bit input signal that selects ALU operation and operate as shown in table1.

iv. Status- a 4-bit output signal work as flag a microprocessor.

v. Clock pulse is only provided to the module which is selected using demux.

vi. Concurrent processes are used to allow processes to run in parallel hence pipelining is achieved by this execution.

Table I. Status Signals

S_1 and S_0	operation
00	add
01	sub
10	Multiply
11	divide

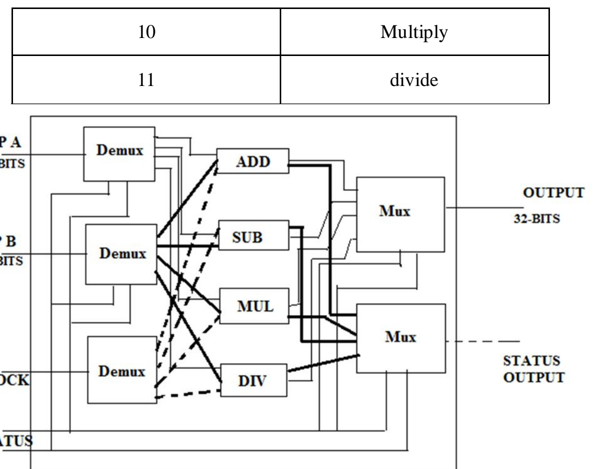

Fig. 7 Pipelining

VI. SIMULATION RESULTS

A. *Adder/subtractor*

The opa and opb are the two inputs (32-bits) of floating point adder and add (32- bits) is the output of floating point adder. The overflow bit will be high if range exceeds the maximum value and underflow bit will be high if range is smaller than minimum value

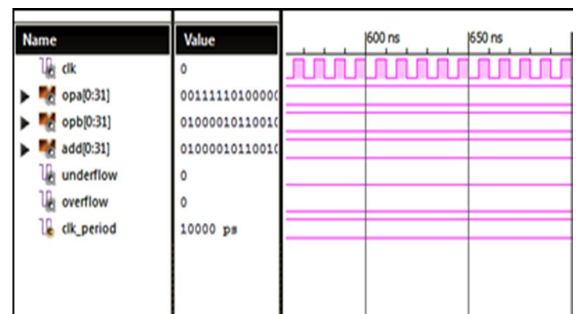

Fig. 8 Behavioral Simulation of Floating Point Adder/Subtractor.

B. *Multiplier*

The fp_a and fp_b are the two inputs (32-bits) of floating point multiplier and fp_z (32-bits) is the output of floating point multiplier.

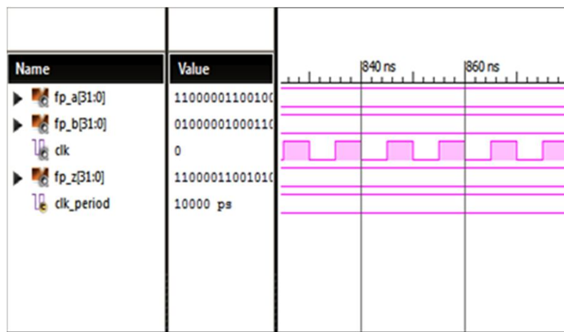

Fig.9 Behavioral simulation of floating-point multiplier.

C. Divider

The ‘a’ and ‘b’ are the two inputs (32-bits) of floating point divider and result (32-bits) is the output of floating point divider. The ‘go’ signal should be high and reset signal should be ‘0’.The output bit overflow will be high if range of the number is exceeds the maximum value.

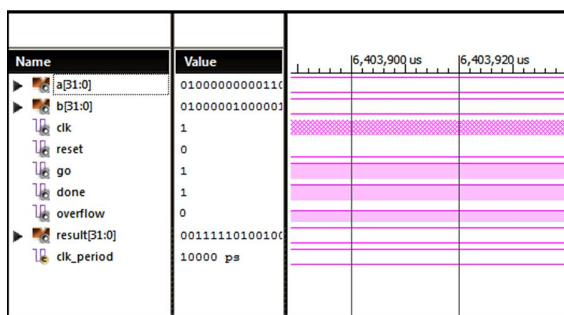

Fig.10 Behavioral Simulation of Floating Point Divider.

Table II. Floating point adder Analysis

module		Clock period(ns)	Area(slices)	Logic levels
FP ADDER	***LOD***	33.159	694	44
	LOP	28.358	731	31
	Two-path	22.313	1020	29
FP multiplier		10.402	272	34
FP divider		7.058	185	27

VII. CONCLUSION

This paper presents single precession floating point arithmetic unit. Four operations are implemented and simulated: addition, subtraction, multiplication and division. FP addition is implemented using LOD, LOP and two-path algorithms. The tradeoff between area and delay is observed in the floating point arithmetic unit by replacing different types of floating point adder algorithms in place of addition.

VIII. FUTURE SCOPE

The future scopes of this project are to implement the proposed floating point arithmetic unit using Field-Programmable Gate Arrays (FPGAs).

REFERENCES

[1] Yee Jern Chong and sri Parameswaram, “Configurable Multimode Embedded units floating-point for FPGAs”, *IEEE Transactions on VLSI systems*, pp. 2033-2044, Vol.19, No.11, November 2011.

[2] IEEE Standard Board and ANSI, “IEEE Standard for Binary Floating-Point Arithmetic”, IEEE Std 754.

[3] J. D. Bruguera and T. Lang, “Leading-One Prediction with Concurrent Position Correction”, *IEEE Transactions on Computers*, pp. 1083–1097, Vol. 48, No.10.

[4] Xilinx, http://www.xlinix/com.

[5] Taek-Jun Kwon, Jeff Sondeen, Jeff Draper, “Design Trade-Offs in Floating-Point Unit, Implementation for Embedded and Processing-In-Memory Systems”, *USC Information Sciences Institute*, *4676 Admiralty Way Marina del Rey*, CA 90292 U.S.A.

[6] Jinwoo Suh, Dong-In Kang, and Stephen P. Crago, “Efficient Algorithms for Fixed-Point Arithmetic Operations in an Embedded PIM”, *University of Southern California/Information Sciences Institute,* 2005.

[7] Yu-Ting Pai and Yu-Kumg Chen, “The Fastest Carry Lookahead Adder”, *Tutorial Report, Department of Electronic Engineering, Huafan University.*

[8] David Narh Amanor, “Efficient Hardware Architectures for Modular Multiplication”, *Communication and Media Engineering, University of Applied Sciences Offenburg, Germany, 2005.*

[9] G. Govindu, L. Zhuo, S. Choi, and V. Prasanna, “Analysis of High-Performance Floating-Point Arithmetic on FPGAs,” *International Parallel and Distributed.Processing Symp.*, pp. 149b, April 2004.

[10] Nabeel Shirazi,Al walters,and peter Athanas, “Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines”, *IEEE Symposisum on FPGAs for Custom Computing Machines.*

[11] V. G. Oklobdzija, “An Algorithmic and Novel Design of a Leading Zero Detector Circuit: Comparison with Logic Synthesis”, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* pp. 124-128, Vol. 2, No. 1.