Design and Implementation of IEEE-754 Decimal Floating Point Adder, Subtractor and Multiplier

S. Murali, B. Srinivas

Abstract— ***This paper describes the development of a Decimal Floating Point adder/subtractor multiplier and division for ALU in verilog with the help of ModelSim and will be synthesized by using Xilinx tools. These are available in single cycle and pipeline architectures and fully synthesizable with performance comparable to other available high speed implementations. The design is described as graphical schematics and code. This representation is very valuable as allows for easy navigation over all the components of the units, which allows for a faster understanding of their inter relationships and the different aspects of a Floating Point operation. The presented DFP adder/subtractor supports operations on the decimal 64 format and our extension is decimal floating point multiplier. The fixed-point design is extended to support floating-point multiplication by adding several components including exponent generation, rounding, shifting, and exception handling. And DFP multiplier is compared with the booth multiplier technique.***

Keywords— ***DFP, Booth multiplier, IEEE 754-1985 standard, Floating point multiplication.***

I. Introduction

Decimal arithmetic plays a key role in many commercial and financial applications, which process decimal values and perform decimal rounding. However, current software implementations are prohibitively slow prompting hardware manufacturers such as IBM to add decimal floating point (DFP)arithmetic support to their microprocessors. Furthermore, the IEEE has developed the newly IEEE 754-2008standard for Floating-Point Arithmetic adding the decimal representation to the IEEE754-1985 standard. Floating-point operations have been acknowledged to be useful for many real-time graphic and multimedia applications since it provides a wide dynamic range of representable real numbers. Most modern processors perform floating-point operation according to the IEEE 754-1985 Standard for binary floating-point arithmetic which has been recently revised to include specifications for decimal floating-point arithmetic (IEEE 754R) . IEEE 754-1985 floating-point multiplication is typically performed by first generating the partial products of two n-bit (n=24 or 53) normalized significand in parallel, followed by reducing these partial products with an addition tree into the sum (s) and carry (c) terms. Next, a carry-propagate adder is used to sum up these two 2n-bit terms to produce a 2n-bit product P, which is then normalized and rounded to produce the final rounded n-bit product. The exponent of the product is the sum of the exponents with proper bias adjustment and increment if the prenormalized product requires a 1-bit normalization shift.

Manuscript Received on October 2014.
S. Murali, Department of Electronics and Communication, MVGR College of Engineering, Vizianagaram, India.
B. Srinivas, Dept. of Electronics and Communication, MVGR College of Engineering, Vizianagaram, India.

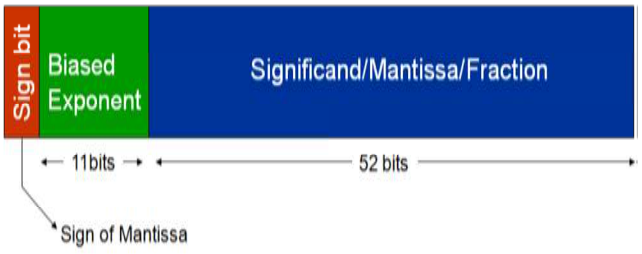


Figure 1. IEEE-754 double precision floating point format

2. Implementation of Double Precision Floating Point Adder/Subtractor

The black box view and block diagram of double precision floating point adder/subtractor is shown in Figures 2 and 3 respectively. The input operands are separated into their sign, mantissa and exponent components. This module has inputs opa and opb of 64-bit width and clk, enable, rst are of 1-bit width. One of the operands is applied at opa and other operand at opb. Larger operand goes into 'mantissa_large' and 'exponent_large', similarly the smaller operand goes into 'mantissa_small' and 'exponent_small'. To determine which operand is larger, compare only the exponents of the two operands, so in fact, if the exponents are equal, the smaller operand might populate the mantissa_large and exponent_large registers. This is not an issue because the reason the operands are compared is to find the operand with the larger exponent, so that the mantissa of the operand with the smaller exponent can be right shifted before performing the addition. If the exponents are equal, the mantissas are added without shifting. The inter-connection of sub-modules of double precession floating point adder/subtractor is shown in Figure 4. Subtraction is similar to addition in that you need to calculate the difference in the exponents between the two operands, and then shift the mantissa of the smaller exponent to the right before subtracting.

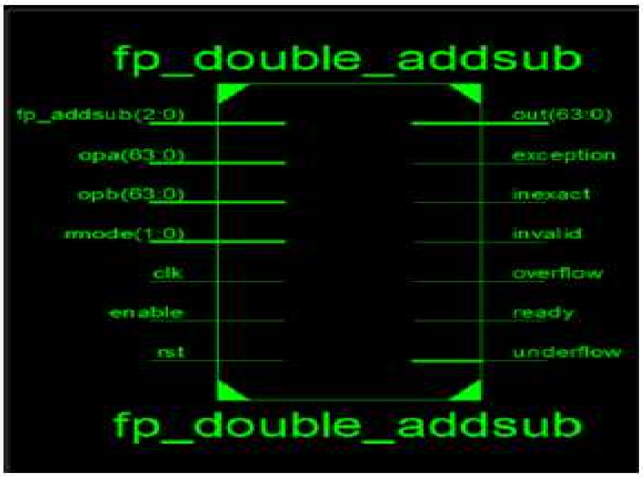


Figure 2. Black box view of double precision floating point adder/subtractor

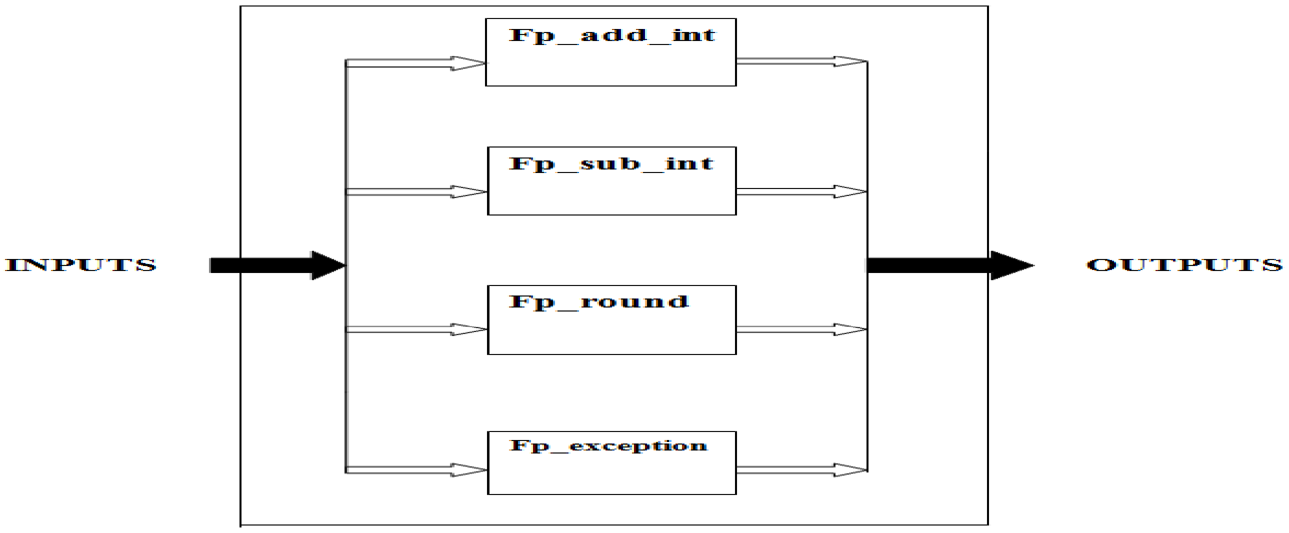


Figure 3. Block diagram of double precision floating point adder/subtractor

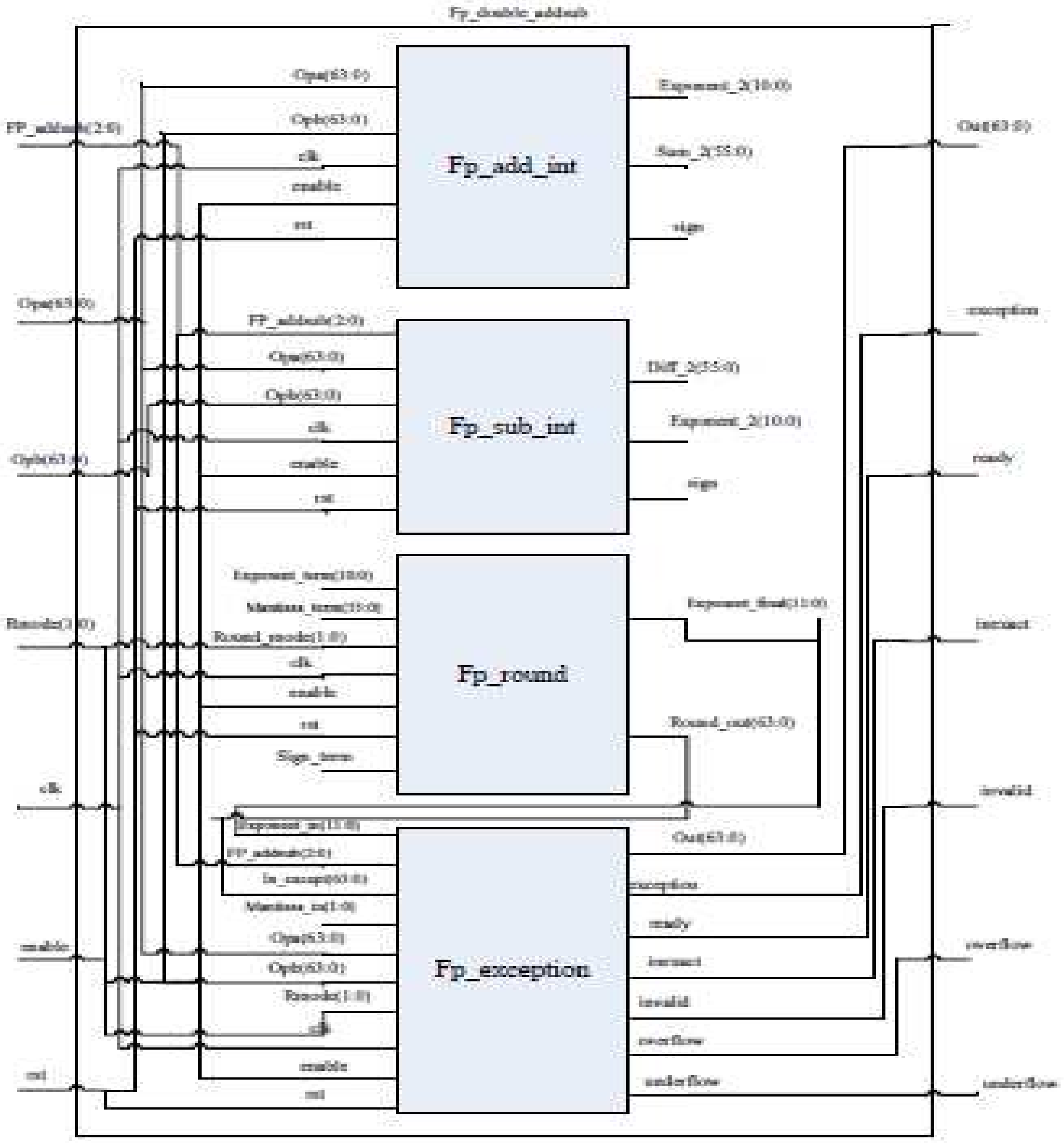


Figure 4. Inter-connection of sub-modules of double precision floating point adder/subtractor

2.1 Algorithm

1. Compare exponent and mantissa of both numbers. Find the large exponent and mantissa and small exponent and mantissa.
2. If both exponents are equal then put leading 1 and 0 in front of each mantissa for overflow, *i.e.*, 52+2 bits.
3. If overflow occurs the exponent must be increased by one. 54th bit will be shifted out of mantissa and this will be saved for rounding purpose.
4. The leftmost 1 in the result becomes leading 1 of mantissa and next 52 bits are actual mantissa.
5. If there is overflow 1 in result, the exponent of larger operand is increased by one. Then add both mantissas.
6. If one exponent is larger than other then subtract smaller exponent from larger exponent, and difference is saved.
7. Now shift smaller mantissa by that subtracted exponent difference.
8. If exponents are equal, *i.e.*, larger exponent is equal to smaller exponent after shifting then put leading 1 in both mantissas and perform addition for mantissas.
9. For subtraction if both exponents are equal then put implied 1 in front of mantissa and perform subtraction.
10. Store that result in “diff” register. Count the number of zeros in “diff” before the left most 1.
11. Reduce the large operand exponent by number of zeros in front of least 1.
12. The left most 1 will be leading 1 in front of mantissa.
13. Finally rounding and normalization is done.

3. Implementation of Double Precision Floating Point Multiplier

3.1 Floating Point Multiplication Algorithm

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
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).The following steps are necessary to multiply two floating point numbers.
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

3.2 Implementation

In this paper we implemented a double precision floating point multiplier with exceptions and rounding. Figure 5 shows the multiplier structure that includes exponents addition, significand multiplication, and sign calculation. Figure 6 shows the multiplier, exceptions and rounding that are independent and are done in parallel.

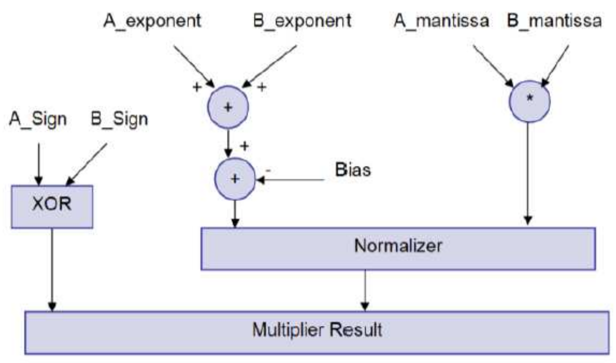


Figure 5. Multiplier Structure

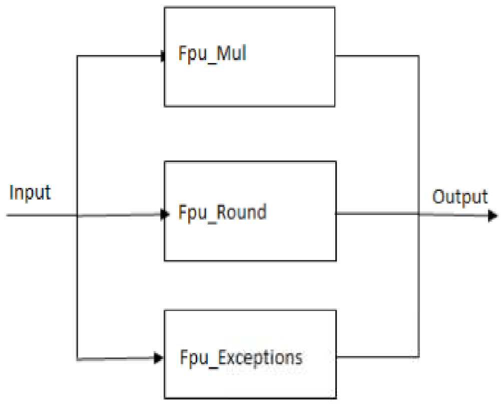


Figure 6. Multiplier structure with rounding and exceptions

3.2.1 Multiplier

The black box view of the double precision floating point multiplier is shown in figure 8.The Multiplier receives two 64-bit floating point numbers. First these numbers are unpacked by separating the numbers into sign, exponent, and mantissa bits. The sign logic is a simple XOR. The exponents of the two numbers are added and then subtracted with a bias number i.e., 1023. Mantissa multiplier block performs multiplication operation. After this the output of mantissa division is normalized, i.e., if the MSB of the result obtained is not 1, then it is left shifted to make the MSB 1. If changes are made by shifting then corresponding changes has to be made in exponent also.

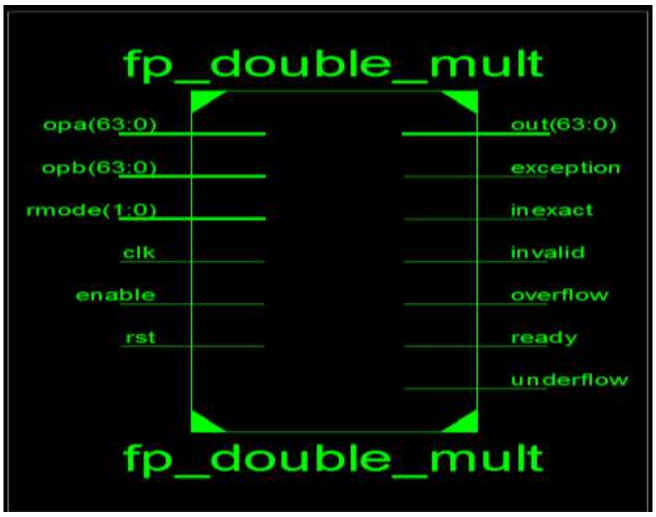
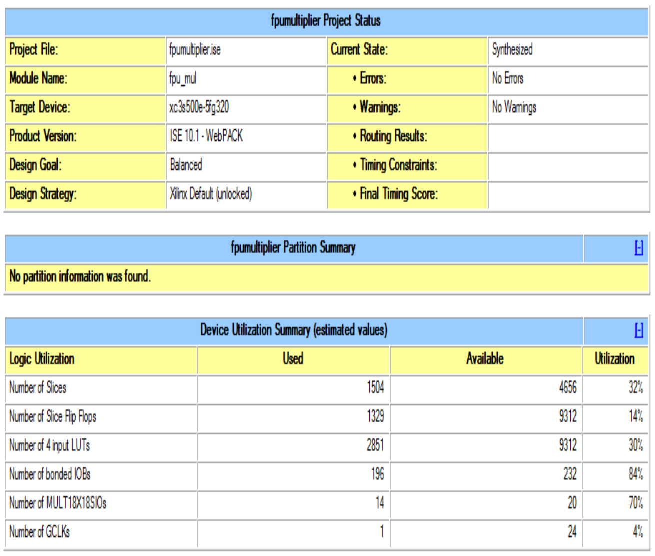


Figure 7. Black box view of floating point double precision multiplier



fpumultiplier Project Status			
Project File:	fpumultiplier.ise	Current State:	Synthesized
Module Name:	fpu_mul	• Errors:	No Errors
Target Device:	xc3s500e-5fg320	• Warnings:	No Warnings
Product Version:	ISE 10.1 - WebPACK	• Routing Results:	
Design Goal:	Balanced	• Timing Constraints:	
Design Strategy:	Xilinx Default (unlocked)	• Final Timing Score:	

fpumultiplier Partition Summary
No partition information was found.

Device Utilization Summary (estimated values)			
Logic Utilization	Used	Available	Utilization
Number of Slices	1504	4656	32%
Number of Slice Flip Flops	1329	9312	14%
Number of 4 input LUTs	2851	9312	30%
Number of bonded IOBs	196	232	84%
Number of MULT18X18SIOs	14	20	70%
Number of GCLKs	1	24	4%

Table design summary of DFP multiplier

The multiplication operation is performed in the module (fpu_mul). The mantissa of operand A and the leading '1' (for normalized numbers) are stored in the 53-bit register (mul_a). The mantissa of operand B and the leading '1' (for normalized numbers) are stored in the 53-bit register (mul_b). Multiplying all 53 bits of mul_a by 53 bits of mul_b would result in a 106-bit product. 53 bit by 53 bit multipliers are not available in the most popular Xilinx and Altera FPGAs, so the multiply would be broken down into smaller multiplies and the results would be added together to give the final 106-bit product. The module (fpu_mul) breaks up the multiply into smaller 24-bit by 17-bit multiplies. The Xilinx Virtex-6 device contains DSP48E1 slices with 25 by 18 two's complement multipliers, which can perform a 24-bit by 17-bit unsigned multiply.

Bit combination	Rounding Mode
00	round_nearest_even
01	round_to_zero
10	round_up
11	round_down

Table Rounding modes selected for various bit combinations of rmode

4. Rounding and Exceptions

The IEEE standard specifies four rounding modes round to nearest, round to zero, round to positive infinity, and round to negative infinity. Table shows the rounding modes selected for various bit combinations of rmode. Based on the rounding changes to the mantissa corresponding changes has to be made in the exponent part also. In the exceptions module, all of the special cases are checked for, and if they are found, the appropriate output is created, and the individual output signals of underflow, overflow, inexact, exception, and invalid will be asserted if the conditions for each case exist.

II. BOOTH MULTIPLIER

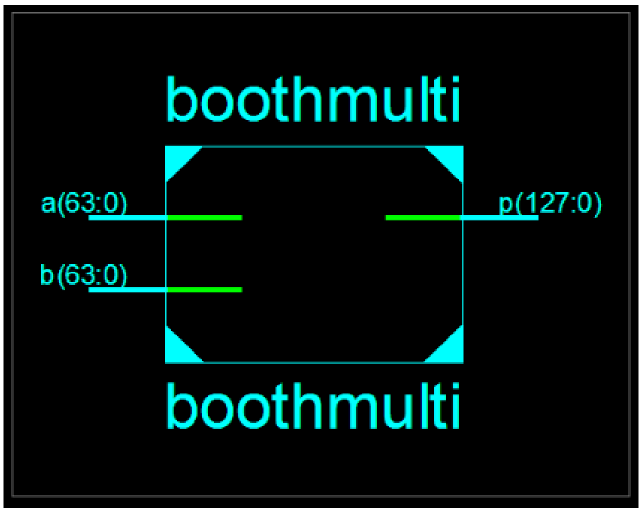


Figure 8 Booth Multiplier

A. *Schematic block*

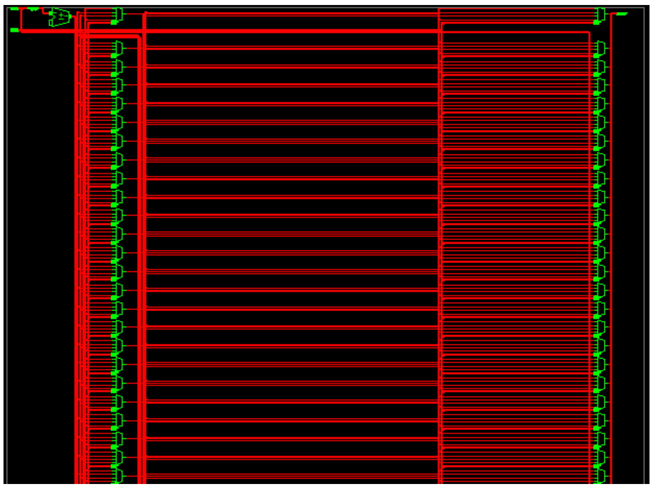
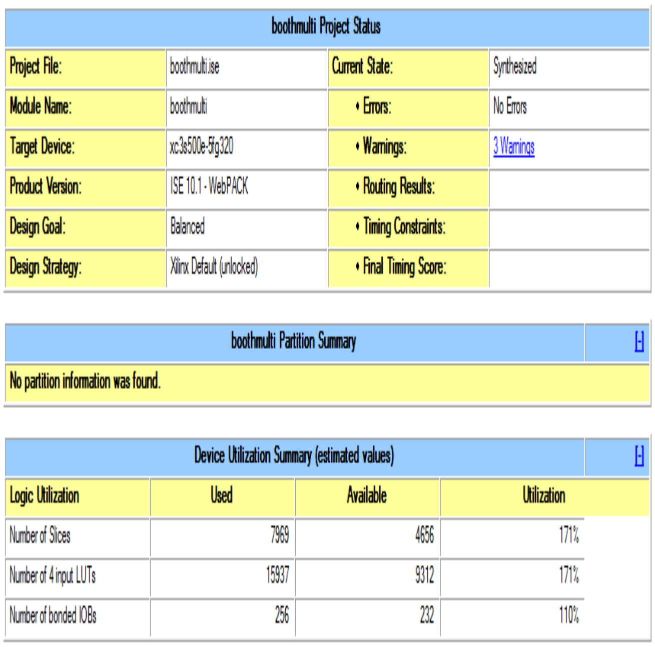


Figure 9: Schematic block booth multiplier

B. *Synthesis Report*

Table1: Design summary



boothmulti Project Status			
Project File:	boothmulti.ise	Current State:	Synthesized
Module Name:	boothmulti	• Errors:	No Errors
Target Device:	xc3s500e-5fg320	• Warnings:	3 Warnings
Product Version:	ISE 10.1 - WebPACK	• Routing Results:	
Design Goal:	Balanced	• Timing Constraints:	
Design Strategy:	Xilinx Default (unlocked)	• Final Timing Score:	

boothmulti Partition Summary
No partition information was found.

Device Utilization Summary (estimated values)			
Logic Utilization	Used	Available	Utilization
Number of Slices	7969	4656	171%
Number of 4 input LUTs	15937	9312	171%
Number of bonded IOBs	256	232	110%

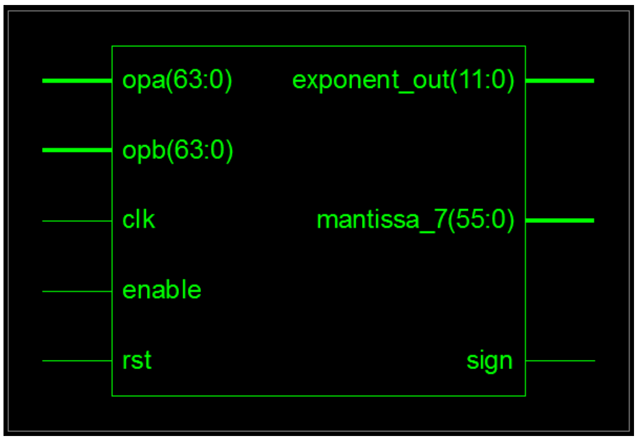
Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication Pvt. Ltd.

Table design summary of booth multiplier

Comparison between booth multiplier and decimal floating point multiplier

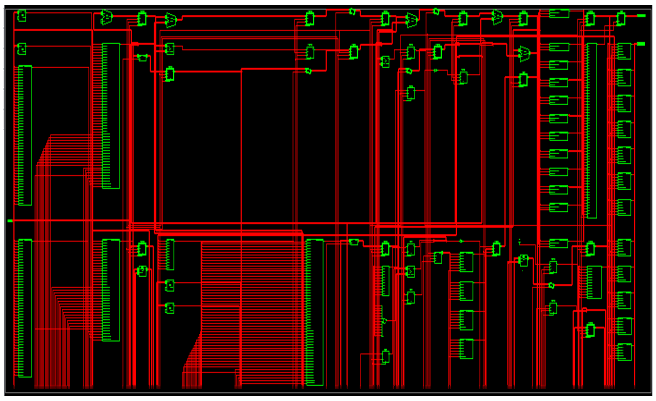
Parameter	Booth multiplier	DFP Multiplier
Delay	327.997ns	4.283ns
No. of 4 input LUT's	15937	2851
No. of slices	7969	1504
Levels of logic	376	7
No.o bonded IOB's	256	196

III. DFP DIVISION

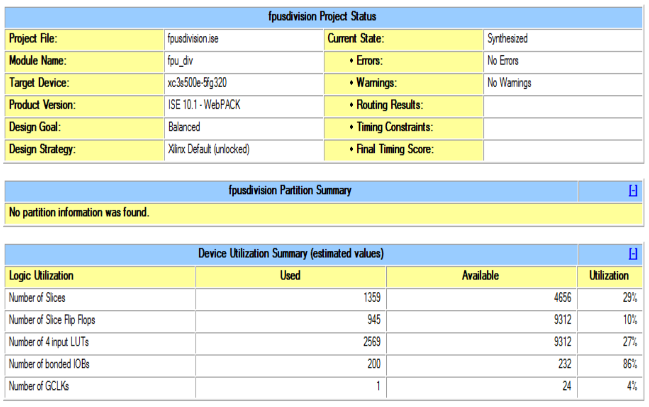


.

IV. INTERNAL SCHMATIC BLOCK DIAGRAM



V. DESIGNSUMMARY

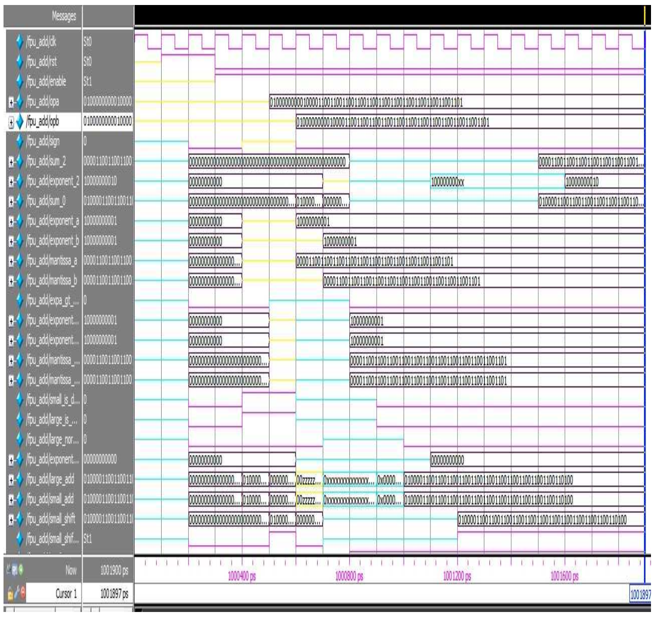


fpusdivision Project Status			
Project File:	fpusdivision.ise	Current State:	Synthesized
Module Name:	fpu_div	• Errors:	No Errors
Target Device:	xc3s500e-5fg320	• Warnings:	No Warnings
Product Version:	ISE 10.1 - WebPACK	• Routing Results:	
Design Goal:	Balanced	• Timing Constraints:	
Design Strategy:	Xilinx Default (unlocked)	• Final Timing Score:	

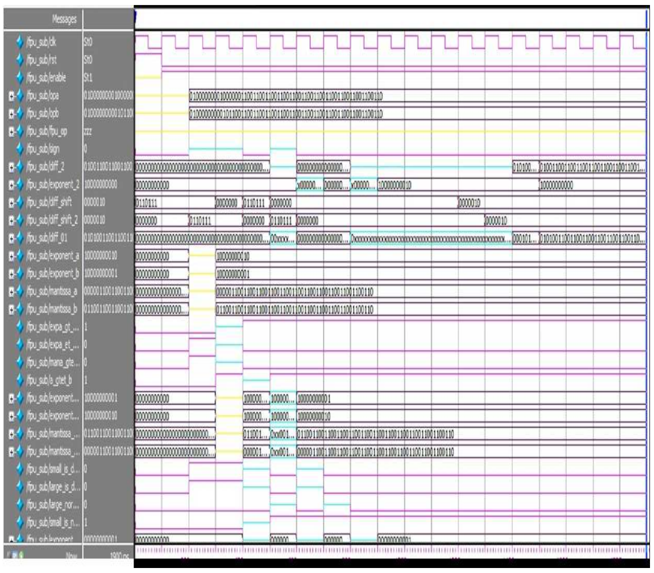
fpusdivision Partition Summary
No partition information was found.

Device Utilization Summary (estimated values)			
Logic Utilization	Used	Available	Utilization
Number of Slices	1359	4656	29%
Number of Slice Flip Flops	945	9312	10%
Number of 4 input LUTs	2569	9312	27%
Number of bonded IOBs	200	232	86%
Number of GCLKs	1	24	4%

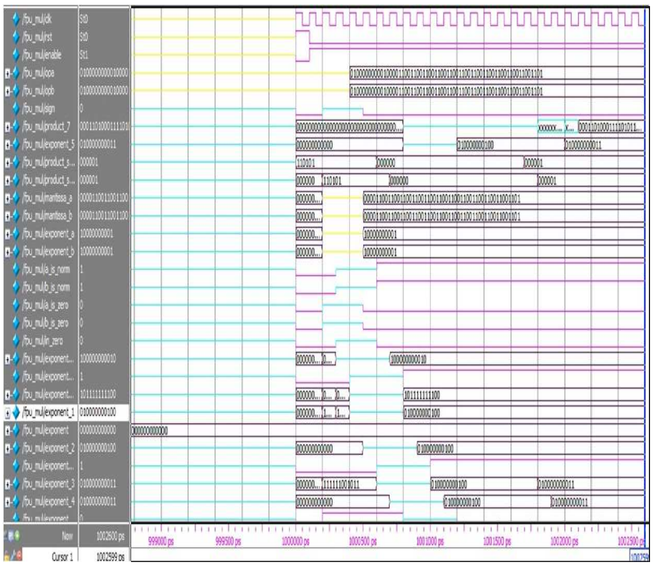
VI. RESULTS



VII. DFP ADDER OUTPUT

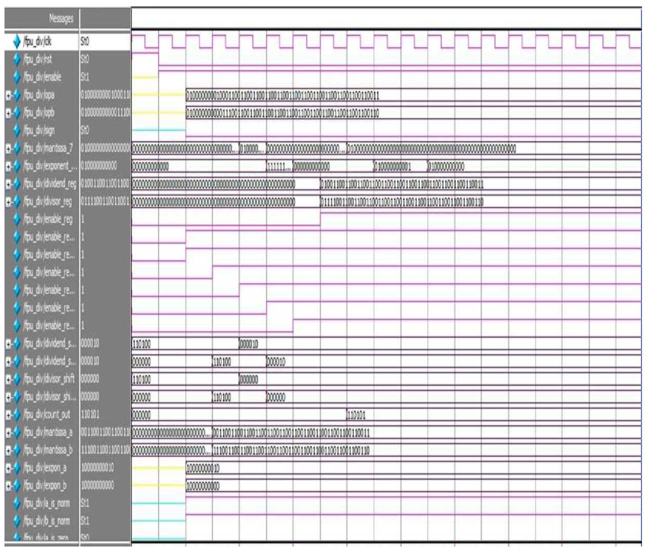


VIII.DFP SUBTRACTOR OUTPUT



Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication Pvt. Ltd.

IX. DFP MULTIPLIER OUTPUT



X. DFP DIVISION OUTPUT

XI. CONCLUSION

In these paper DFP adder/subtractor/division are implemented and DFP multiplier is compared with booth multiplier by using Xilinx software.In booth multiplier 327.997ns delay is obtained, but in floating point multiplier the delay has reduced to 4.283ns. So from the obtained results, it is clear that the floating point multiplier can perform better performance than the existed booth multiplier.

REFERENCES

[1] Improved Architectures for a Fused Floating-Point Add-Subtract Unit Jongwook Sohn, *StudentMember, IEEE, and Earl E. Swartzlander, Jr., Life Fellow,* IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—I: REGULAR PAPERS, VOL. 59, NO. 10, OCTOBER 2012.

[2] *IEEE Standard for Floating-Point Arithmetic, ANSI/IEEE Standard* 754-2008, New York: IEEE, Inc., Aug. 29, 2008.

[3] E. Hokenek, R. K. Montoye, and P. W. Cook, “Second-generation RISC floating point with multiply-add fused,” *IEEE J. Solid-State Circuits*, vol. 25, no. 5, pp. 1207–1213, Oct. 1990.

[4] T. Lang and J. D. Bruguera, “Floating-point fused multiply-add with reduced latency,” *IEEE Trans. Comput., vol. 53, no. 8, pp. 988–1003,* Aug. 2004.

[5] IEEE Standard for Floating-Point Arithmetic, pp. 1 58, 2008, iEEE Std 754-2008.

[6] M. F. Cowlishaw, Decimal floating-point: algorism for computers, in *Proc. 16th IEEE Symp. Computer Arithmetic, 2003, pp. 104 111.*