Design and Simulation of Pipelined Double Precision Floating Point Adder/Subtractor and Multiplier Using Verilog

Onkar Singh (1) Kanika Sharma (2)

Dept. ECE, Arni University, HP (1) Dept. ECE, NITTTR Chandigarh (2)

Abstract

A floating-point unit (FPU) is a part of a computer system specially designed to carry out operations on floating point numbers. This paper presents FPGA implementation of a single unit named Adder/Subtractor which is able to perform both double precision floating point addition and subtraction and a double precision floating point multiplier. Both the design is based on pipelining so the overall throughput is increased. Both units are implemented using Verilog and the code is dumped into vertex-5 FPGA.

1. Introduction

An arithmetic unit (AU) is the part of a computer processing unit that carries out arithmetic operations on the operands in computer instruction words. Generally arithmetic unit (AU) performs arithmetic operations like addition, subtraction, multiplication and division. Some processors contain more than one AU for example, one for fixed-point operations and another for floating-point operations. To represent very large or small values, large range is required as the integer representation is no longer appropriate. These values can be represented using the IEEE-754 standard based floating point representation. Typical operations are addition, subtraction, multiplication and division. In most modern general purpose computer architectures, one or more FPUs are integrated with the CPU; however many embedded processors, especially older designs, do not have hardware support for floating-point operations. Almost every language has a floating-point data type; computers from PC's to supercomputers have floating-point accelerators; most compilers will be called upon to compile floating-point algorithms from time to time; and virtually every operating system must respond to floating-point exceptions such as overflow.

In the proposed design both adder/subtractor and multiplier units are designed by using pipelining so the throughput of operation can be increased.

Basically in designing floating point units there are three stages to do for completing the tasks and these stages are:

Pre-normalize: The operands are transformed into formats that makes them easy and efficient to handle internally.

***Arithmetic core*:** The basic arithmetic operation are done here for example addition, subtraction or multiplication

***Post-normalize*:** The result will be normalized if possible and then transformed into the format specified by the IEEE standard.

The pipelining concept is used in between these three stages. When certain inputs are come into pre-normalize stage after pre-normalizing these inputs are transferred into arithmetic core and this time the first unit named pre-normalize unit is free to serve for next inputs so the throughput of overall design can be improved.

Inputs	**Pipeline Stages**		
1st pair of inputs	Pre-normalize	Arithmetic core	Post-normalize
2nd pair of inputs		Pre-normalize	Arithmetic core
3rd pair of inputs			Pre-normalize

Figure 1.1: Pipelining operation

The IEEE 754 is a floating point standard established by IEEE in 1985. It contains two representations for

floating-point numbers, the IEEE single precision format and the IEEE double precision format.

1.1 IEEE Single Precision Format: The IEEE single precision format uses 32 bits for representing a floating point number, divided into three subfields, as illustrated in figure 1.2

S	Exponent	Fraction
1 bit	8 bits	23 bits

Figure 1.2: IEEE single precision floating point format

1.2 IEEE Double Precision Format: The IEEE double precision format uses 64 bits for representing a floating point number, as illustrated in figure 1.3

S	Exponent	Fraction
1 bit	11 bits	52 bits

Figure 1.3: IEEE double precision floating-point format

2. Floating Point Adder/Subtractor

The block diagram of the proposed adder/subtractor unit is shown in figure 2.1 the unit supports double precision floating point addition and subtraction. In this design pipelining concept is used so the throughput of the design is increased.

Two floating point numbers are added as shown.

$$(F_1 * 2^{E1}) + (F_2 * 2^{E2}) = F * 2^{E}$$

Two floating point numbers are subtracted as shown.

$$(F_1 * 2^{E1}) - (F_2 * 2^{E2}) = F * 2^{E}$$

In order to add/Subtract two fractions, the associated exponents must be equal. Thus, if the exponents E_1 and E_2 are different, we must unnormalize one of the fractions and adjust the exponents accordingly. The smaller number is the one that should adjusted so that if significant digits are lost, the effect is not significant

2.1 The unit has following inputs:

1. Two 64-bit operands (opa, opb)
2. Four rounding mode

***00=Round to nearest even*:** This is the standard default rounding. The value is rounded up or down to the nearest infinitely precise result. If the value is exactly halfway between two infinitely precise results, then it should be rounded up to the nearest infinitely precise even.

01=Round-to-Zero: Basically in this mode the number will not be rounded. The excess bits will simply get truncated, e.g. 3.47 will be truncated to 3.5

10=Round-Up: In this mode the number will be rounded up towards +∞, e.g. 5.2 will be rounded to 6, while -4.2 to -4

11=Round-Down: The opposite of round-up, the number will be rounded up towards -∞, e.g. 5.2 will be rounded to 5, while -4.2 to -5

3. Clock (Global)
4. Enable (set high to start operation)
5. Fpu_op (0=add, 1=sbtract)
6. Restart (global)

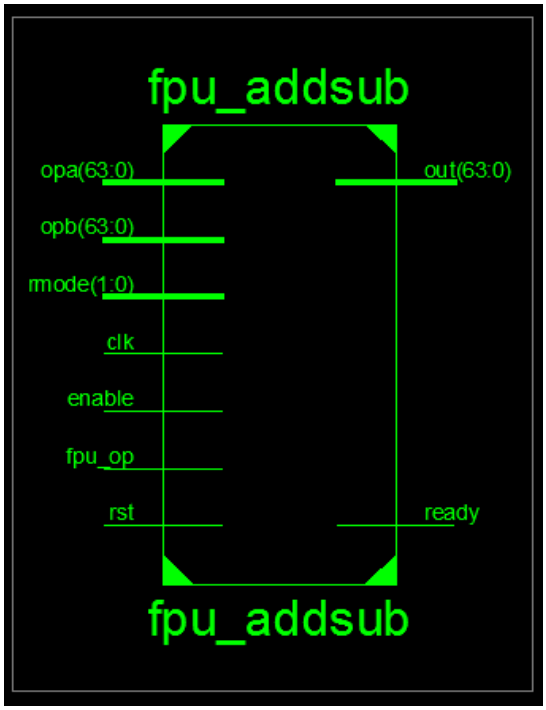

Figure 2.1: Double precision floating point adder/subtractor

2.2 The unit has following outputs:

1. 64-bit output (63:0)
2. Ready (goes high when output is available)

2.3 Steps required to carry out floating point addition/Subtraction are as follows

1. Compare exponents. If the exponents are not equal, shift the fraction with the smaller exponent right and add 1 to its exponent; repeat until the exponents are equal.
2. Add/Subtact the fractions.
3. If the result is 0, set the exponents to the appropriate representation for 0 and exit.
4. If fraction overflow occurs, shift right and add 1 to the exponent to correct the overflow.
5. If the fraction is unnormalized, shift left and subtracts 1 from the exponent until the fraction is normalized.
6. Check for exponent overflow. Set overflow indicator, if necessary
7. Round to the appropriate number of bits.

3. Floating Point Multiplier

In this section, the design of multiplier for floating point numbers is proposed.
Given two floating point numbers, the product is

$$(F1 * 2^{E1}) * (F2 * 2^{E2}) = (F1 * F2) * 2^{(E1+E2)} = F * 2^{E}$$

3.1 The unit has following inputs:

1. Two 64-bit operands (opa, opb)
2. Four rounding mode (00=Round to nearest even, 01=Round to zero, 10=Round up, 11=Round down)
3. Clock (Global)
4. Enable (set high to start operation)
5. Restart (global)

3.2 The unit has following outputs:

1. 64-bit output (63:0)
2. Ready (goes high when output is available)

3.3 Double Precision Floating Point Multiplication Operation:

There are two operand named operand A and operand B to be multiplied. 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. Depending on the synthesis tool used, this might be synthesized in different ways that would not take efficient advantage of the multiplier resources in the target device. 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. Instead of relying on the synthesis tool to break down the multiply, which might result in a slow and inefficient layout of FPGA resources, the module (fpu_mul) breaks up the multiply into smaller 24-bit by 17-bit multiplies. The Xilinx Virtex5 device contains DSP48E slices with 25 by 18 twos complement multipliers, which can perform a 24-bit by 17-bit unsigned multiply. The breakdown of the 53-bit by 53-bit floating point multiply into smaller components

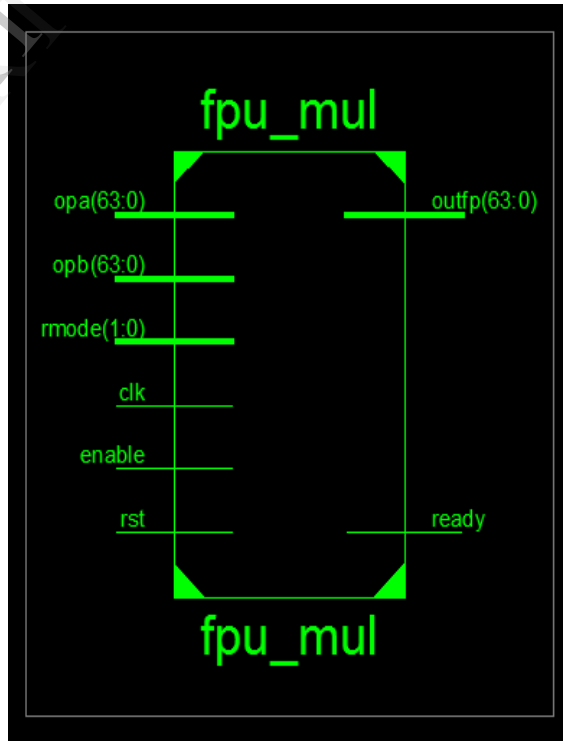

Figure 3.1: Double precision floating point multiplier

The multiply is broken up as follows:
product_a = mul_a[23:0] * mul_b[16:0]
product_b = mul_a[23:0] * mul_b[33:17]
product_c = mul_a[23:0] * mul_b[50:34]
product_d = mul_a[23:0] * mul_b[52:51]

product_e = mul_a[40:24] * mul_b[16:0]
product_f = mul_a[40:24] * mul_b[33:17]
product_g = mul_a[40:24] * mul_b[52:34]
product_h = mul_a[52:41] * mul_b[16:0]
product_i = mul_a[52:41] * mul_b[33:17]
product_j = mul_a[52:41] * mul_b[52:34]

The mantissa output from the (fpu_mul) module is in 56-bit register (product_7). The MSB is a leading '0' to allow for a potential overflow in the rounding module. The first bit '0' is followed by the leading '1' for normalized numbers, or '0' for denormalized numbers. Then the 52 bits of the mantissa The products (a-j) are added together, with the appropriate offsets based on which part of the mul_a and mul_b arrays they are multiplying. The summation of the products is accomplished by adding one product result to the previous product result instead of adding all 10 products (a-j) together in one summation. The final 106-bit product is stored in register (product). The output will be left-shifted if there is not a '1' in the MSB of product. The exponent fields of operands A and B are added together and then the value (1022) is subtracted from the sum of A and B. If the resultant exponent is less than 0, than the (product) register needs to be right shifted by the amount. The final exponent of the output operand will be 0 in this case, and the result will be a denormalized number.

4. Synthesis Report

These are the final results which are obtained in the synthesis report when we are going to synthesis Verilog code of floating point adder/subtractor and multiplier on Virtex 5. Table 1 shows the device utilization summary for adder/subtractor and Table 2 shows device utilization summary for multiplier. The parameters such as number of slices registers, number of slice flip flop, GCLKs etc are outline in the synthesis report are as follows.

Table 1 Device utilization summary for adder/subtractor

Slice Logic Utilization	Used	Available	Utilization
Number of Slice Registers	2,462	19,200	12%
Number used as Flip Flops	2,462		
Number of Slice LUTs	2,557	19,200	13%
Number used as logic	2,358	19,200	12%
Number used as Memory	198	5,120	3%
Number used as Shift Register	198		
Number used as exclusive route-thru	1		
Number of route-thrus	106		
Number of occupied Slices	944	4,800	19%
Number of LUT Flip Flop pairs used	3,341		
Number of fully used LUT-FF pairs	1,678	3,341	50%
Number of bonded IOBs	199	220	90%
Number of BUFG/BUFGCTRLs	2	32	6%
Number used as BUFGs	2		
Average Fanout of Non-Clock Nets	4.18		

Table 2 Device utilization summary for Multiplier

Slice Logic Utilization	Used	Available	Utilization
Number of Slice Registers	1,655	19,200	8%
Number used as Flip Flops	1,654		
Number of Slice LUTs	1,100	19,200	5%
Number used as logic	954	19,200	4%
Number used as Memory	145	5,120	2%
Number used as Shift Register	145		
Number used as exclusive route-thru	1		
Number of route-thrus	126		
Number of occupied Slices	519	4,800	10%
Number of LUT Flip Flop pairs used	1,780		
Number of fully used LUT-FF pairs	975	1,780	54%
Number of bonded IOBs	198	220	90%
Number of BUFG/BUFGCTRLs	1	32	3%
Number used as BUFGs	1		
Number of DSP48Es	9	32	28%
Average Fanout of Non-Clock Nets	2.92		

5. Simulation Result

The simulation results of double precision floating point adder/subtractor (Addition, Subtraction) are shown in figures 5.1 and 5.2 respectively and the simulation result for double precision floating point multiplication is shown in figure 5.3

In the waveform ready is 1 that means the output is available at the output. Clock is 1 that means clock is applied to the code. Reset is 0 that define the out is not zero if the value of reset is 1 that means out is zero. The value of enable is 1 that means particular operation is started. fpu_op is 0 for addition and 1 for subtraction. Opa1 and Opa1 defines the operand one and operand two respectively and out define the final result. The r_mode signal defines the various rounding modes.

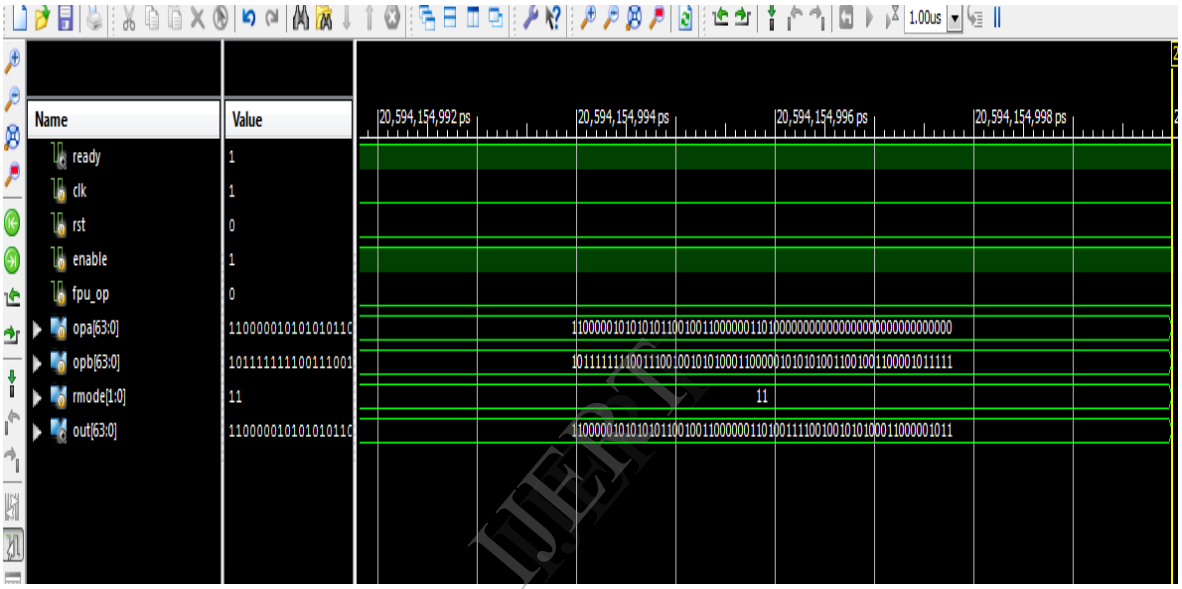

Figure 5.1: Simulation waveform of double precision floating point addition

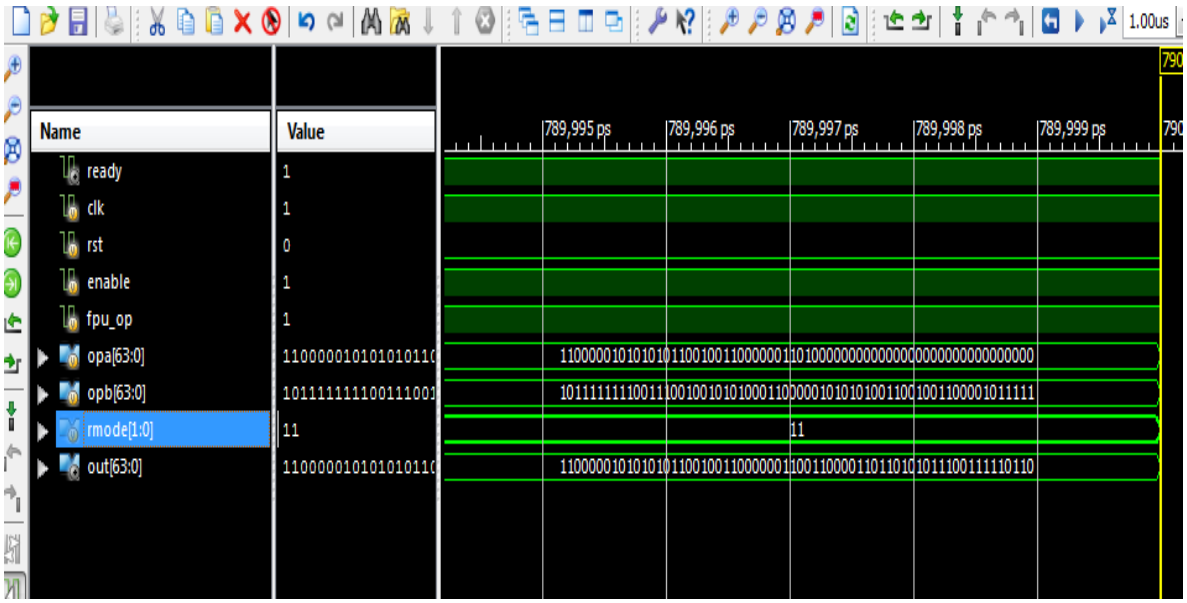

Figure 5.2: Simulation waveform of double precision floating point subtraction

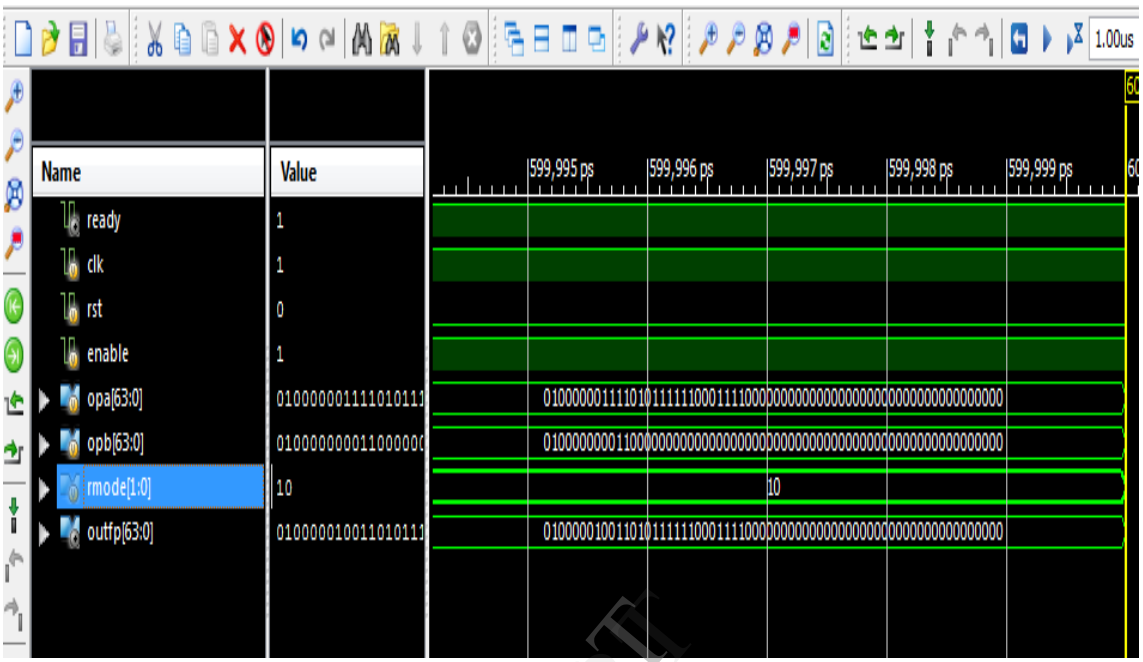

Figure 5.3: Simulation waveform of double precision floating point multiplication

6. Conclusion and Future Work

This paper presents the implementation of double precision floating point adder/subtractor and multiplier. The whole design was captured in Verilog Hardware description language (HDL), tested in simulation using Model Tech‟s modelsim, placed and routed on a Vertex 5 FPGA from Xilinx. The proposed VLSI design of the Double Precision adder/subtractor increases the precision over the Single Precision arithmetic unit and also throughput. For future work the whole design can be implemented on vertex-6 FPGA and also other mathematical units such as divider can be designed.

7. References

[1] Deepa Saini , Bijender M'dia "Floating Point Unit Implementation on FPGA" International Journal Of Computational Engineering Research(IJCER), Vol. 2 , pp.972-976, Issue No.3, May-June 2012

[2] Karan Ghmber, Sharmelle Thangjam "Performance Analysis of Floating Point Adder using VHDL on Reconfigurable Hardware" International Journal of Computer Application, Vol.46-No 9, May 2012

[3] Addanki Purna Ramesh, Pradeep "FPGA based Implementation of Double Precision Floating Point Adder/subtractor using Verilog" International Journal of Emerging Technology and Advanced Engineering, pp.13-142Volume 2, Issue 7, July 2012

[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), pp. 197-203, 2010

[5] Sateesh Reddy, Vinit T Kanojia "Unified Reconfigurable Floating-Point Pipelined Architecture" International Journal of Advanced Engineering Sciences and Technologies, Vol No. 7, Issue No. 2, pp. 271 – 275, 2011

[6] Rathindra Nath Giri, M.K.Pandit "Pipelined Floating-Point Arithmetic Unit (FPU) for Advanced Computing Systems using FPGA" International Journal of Engineering and Advanced Technology (IJEAT), Volume-1, Issue-4, pp. 168-174, April 2012

[7] Shrivastava Purnima, Tiwari Mukesh, Singh Jaikaran and Rathore Sanjay "VHDL Environment for Floating point

Arithmetic Logic Unit - ALU Design and Simulation" Research Journal of Engineering Sciences, Vol. 1(2), pp.1-6, August -2012

[8] Jongwook Sohn, Earl E. Swartzlander "Improved Architectures for a Fused Floating Point Add-Subtract Unit" IEEE Transactions on Circuits and Systems—I: regular papers, Vol. 59, No. 10, pp. 2285-2291, October 2012

[9] Per Karlstrom, Andreas Ehliar, Dake Liu "High Performance, Low Latency FPGA based Floating Point Adder and Multiplier Units in a Virtex 4", 24th Norchip Conference, pp. 31 – 34, Nov. 2006.

[10] Liangwei Ge, Song Chen, yuichi Nakamura "A Systhesis Method of General Floating Point Arithmetic Units by Aligned Partition", 23rd International Conference on Circuitd/Systems, Computers and Communications, pp. 1177-1180, 2008