IOSR Journal of Electronics and Communication Engineering (IOSR-JECE)
e-ISSN: 2278-2834,p- ISSN: 2278-8735.Volume 10, Issue 1, Ver. 1 (Jan - Feb. 2015), PP 49-54
www.iosrjournals.org

Design and Implementation of High Speed Area Efficient Double Precision Floating Point Arithmetic Unit

Onkar Singh[1], Kanika Sharma[2]
[1]*Arni University, Kathgarh, Indora, HP, India.*
[2]*NITTTR, Chandigarh. India.*

Abstract: *A floating-point arithmetic unit designed to carry out operations on floating point numbers. Floating point numbers can support a much wider range of values than fixed point representation. Floating Point units are mainly used in high speed objects recognition system, high performance computer systems, embedded systems, mobile applications. Latch based design is implemented in the proposed work so the longer combinational paths can be compensated by shorter path delays in the subsequent logic gates. That is why the performance has increased in the design. All four individual units addition, subtraction, multiplication and division are designed using Verilog verified by using Questa Sim and implemented on vertex-5 FPGA*
Keywords: *Floating Point, IEEE, FPGA, Vertex-5, Double Precision, Verilog, Arithmetic Unit*

I. Introduction

A floating point arithmetic unit designed to carry out operations on floating point numbers. Floating point arithmetic unit is widely used in high speed objects recognition system, high performance computer systems, embedded systems, mobile applications and signal processing applications. Floating point representation can support a much wider range of values than fixed point representation. To represent very large or small values, wide 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. The design has been implemented on latches so the longer combinational paths can be compensated by shorter path delays in the subsequent logic gates. That is why the performance has increased in the design.

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. The first field is the sign bit for the fraction part. The next field consists of 8 bits which are used for exponent the third field consists of the remaining 23 bits and is used for the fractional part.

Sign	Exponent	Fraction
1 bit	8 bits	23 bits

Fig 1: IEEE format for single precision

IEEE Double Precision Format: The IEEE double precision format uses 64 bits for representing a floating point number, as illustrated in figure 2. The first bit is the sign bit for the fraction part. The next 11 bits are used for the exponent, and the remaining 52 bits are used for the fractional part.

Sign	Exponent	Fraction
1 bit	11 bits	52 bits

Fig 2: IEEE format for double precision

II. Implementation Of Double Precision Floating Point Arithmetic Unit

The block diagram of the proposed floating point arithmetic unit is given in figure 3. The unit supports four arithmetic operations: Add, Subtract, Multiply and Divide. All arithmetic operations have been carried out in four separate modules one for addition, one for subtraction, one for multiplication and one for division as shown in figure 3. In this unit one can select operation to be performed on the 64-bit operands by a 3-bit op-code and the same op-code selects the output from that particular module and connects it to the final output of the unit. Particular exception signal will be high whenever that type of exception will occur.

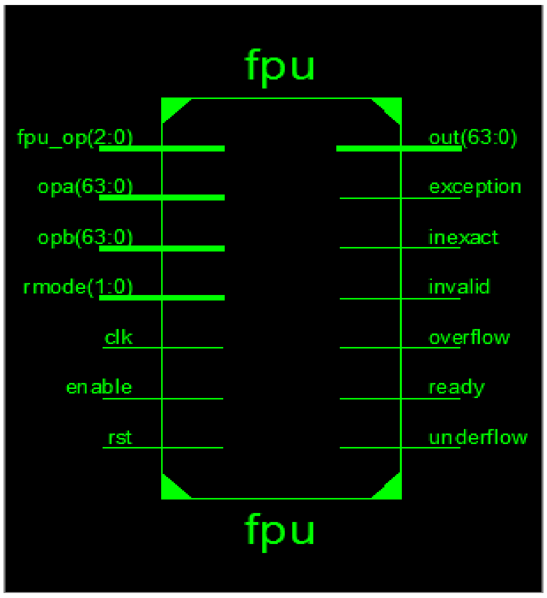

Fig 4: RTL view of double precision floating point arithmetic unit

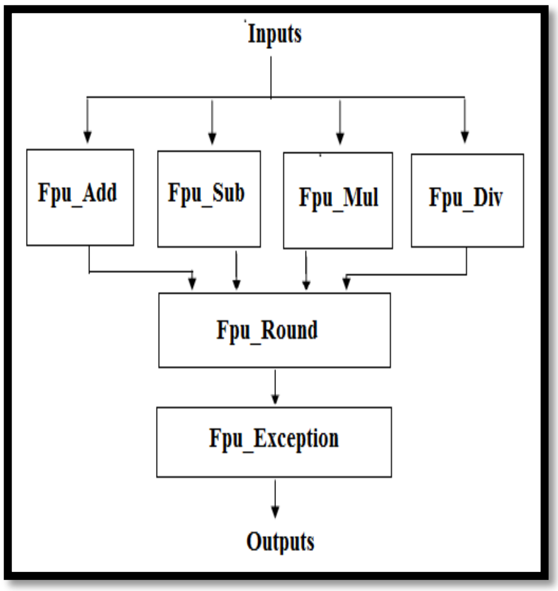

Fig 3: Block diagram of double precision floating point arithmetic unit

The floating point arithmetic unit consists of following blocks
2.1) Fpu_Add- Floating Point adder
2.2) Fpu_Sub- Floating Point Subtractor
2.3)Fpu_Mul-Floating Point Multiplier
2.4) Fpu_Div- Floating Point Division
2.5)Fpu_Round-Floating Point Rounding Unit
2.6) Fpu_Exception- Floating Point Exception Unit

2.1) Fpu_Add- Floating Point adder- Two floating point numbers are added as shown below.

$$(f_1 \times 2^{e1}) + (f_2 \times 2^{e2}) = F \times 2^E$$

In order to add two fractions, the associated exponents must be equal. Thus, if the two exponents are different, we must un normalize 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. In this module two 64-bit numbers are added and after going through rounding and exception part final added result will come to the output.

2.2) Fpu_Sub- Floating Point Subtractor- Two floating point numbers are subtracted as shown below.

$$(f_1 \times 2^{e1}) - (f_2 \times 2^{e2}) = F \times 2^E$$

In order to subtract two fractions, the associated exponents must be equal. Thus, if the two exponents are different, we must un normalize 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. In this module two 64-bit numbers are subtracted and after going through rounding and exception part final subtracted result will come to the output.

2.3) Fpu_Mul- Floating Point Multiplier-Two floating point numbers are multiplied as shown below.

$$(f1 \times 2^{e1}) \times (f2 * 2^{e2}) = (f1 \times f2) \times 2^{(e1+e2)} = F \times 2^E$$

In this module two 64-bit numbers are multiplied using sub multipliers and after going through rounding and exception part final multiplied result will come to the output. The mantissa of operand A are stored in the 53-bit register (mul_a). The mantissa of operand B 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 wide product and a 53 by 53 bit multiplier is not available in the most popular Xilinx 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 Virtex5 device contains

DSP48E slices with 25 by 18 twos complement multipliers, which can perform a 24 by 17-bit multiply. The multiply is broken up as follows:

Multiplier_1 = mul_a[23:0] x mul_b[16:0]
Multiplier_2 = mul_a[23:0] x mul_b[33:17]
Multiplier_3 = mul_a[23:0] x mul_b[50:34]
Multiplier_4= mul_a[23:0] x mul_b[52:51]
Multiplier_5 = mul_a[40:24] x mul_b[16:0]
Multiplier_6 = mul_a[40:24] x mul_b[33:17]
Multiplier_7= mul_a[40:24] x mul_b[52:34]
Multiplier_8 = mul_a[52:41] x mul_b[16:0]
Multiplier_9 = mul_a[52:41] x mul_b[33:17]
Multiplier_10 = mul_a[52:41] x mul_b[52:34]

The multiplier (1-10) 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 multiplier (1-10) together in one summation. The final 106-bit product is stored in register (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.

2.4) Fpu_Div- Floating Point Division - Two floating point numbers are divided as shown below.

$$(f1 \times 2^{e1}) / (f2 \times 2^{e2}) = (f1 / f2) \times 2^{(e1-e2)} = F \times 2^{E}$$

In this module two 64-bit numbers are divided and after going through rounding and exception part final divided result will come to the output. The leading ‘1’ (if normalized) and mantissa of operand A is the dividend, and the leading ‘1’ (if normalized) and mantissa of operand B is the divisor. In division one bit of the quotient calculated each clock cycle based on a comparison between the dividend and divisor register. If the dividend is greater than the divisor, the quotient bit is ‘1’, and then the divisor is subtracted from the dividend, and the resulting difference is shifted one bit to the left, and after shifting it becomes the dividend for the next clock cycle. And in another case if the dividend is less than the divisor, the dividend is shifted one bit to the left, and then this shifted value becomes the dividend for the next clock cycle. Repeat the steps by the number of bits time. The number in the dividend place gives remainder value and quotient place gives quotient value

2.5) Fpu_Round-Floating Point Rounding Unit - Rounding module is used to modifies a number and fit it in the destination’s format. The various rounding modes are written below.
2.5.1) Round to nearest even**:** This is the standard default rounding. The value is rounded down or up to the nearest infinitely precise result.
2.5.2) Round-to-Zero: Basically in this mode the number will not be rounded. The excess bits will simply get truncated, e.g. 5.48 will be truncated to 5.5
2.5.3) Round-Up: In this mode the number will be rounded up towards +∞, e.g. 6.4 will be rounded to 7, while -5.4 to -5
2.5.4) Round-Down: The opposite of round-up, the number will be rounded up towards -∞, e.g. 6.4 will be rounded to 6, while -5.4 to -6

2.6) Fpu_Exception- Floating Point Exception Unit- Exception occurs when an operation on some particular operands has no outputs suitable for a reasonable application.
The five possible exceptions are:
2.6.1) Invalid**:** Operation are like square root of a negative number, returning of NaN by default, etc., output of which does not exist.
2.6.2) Division by zero: It is an operation on a finite operand which gives an exact infinite result for e.g., 1/0 or log (0) that returns positive or negative infinity by default.
2.6.3) Overflow**:** It occurs when an operation results a very large number that can’t be represented correctly i.e. which returns ±infinity by default (for round-to-nearest mode).
2.6.4) Underflow: It occurs when an operation results very small i.e. outside the normal range and inexact by default.
2.6.5) Inexact: It occurs whenever the result of an arithmetic operation is not exact due to the restricted exponent or precision range.

III. Synthesis, Timing And Simulation Result

3.1) Synthesis Result

DEVICE UTILIZATION SUMMARY			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	4762	28800	16%
Number of Slice LUTs	6525	28800	22%
Number of fully used LUT-FF pairs	3013	7600	36%
Number of bonded IOBs	206	220	93%
Number of BUFG/BUFGCTRLs	6	32	18%
Number of DSP48Es	9	48	18%

3.2) Timing Result

Minimum Period	3.817ns (Maximum Frequency: 262.006MHz)
Minimum Input arrival time before clocks	3.900ns
Maximum output required time after clocks	2.775ns

Operation	Time taken by modules in ns
Addition	57.255ns (15 cycles)
Subtraction	57.255ns (15 cycles)
Multiplication	57.255ns (15 cycles)
Division	259.556ns (68 cycles)

3.3) Simulation Result- The simulation results of double precision floating point arithmetic unit (Addition, Subtraction, Multiplication and Division) is shown in fig 5, fig 6, fig 7, fig 8 respectively. It is calculated for the two input operands of 64 bits each. The reset signal is kept low throughout the simulation, so that operands are initialised all at once, then at the high of enable signal, operation of the two operands are calculated. After calculating the result, the result goes into fpu_round and then goes into fpu_exceptions. From fpu_exceptions the out signal gives the output. In the waveforms clock defines the applied frequency (262.006MHz) to the signals. Fpu_op defines the operation to be preformed that is 0=addition,1=subtraction, 2=multiplication and 3=division. Opa1 and Opa1 defines the input operand one and input operand two respectively. The r_mode signal defines the various rounding modes (00=Round to nearest even, 01=Round-to-Zero, 10=Round-Up, 11=Round-Down. Fpu_out defines the final output of the signals.

3.3.1) Simulation Result of floating point addition- It is calculated for the two input operands of 64 bits each. 15 clock cycles are required by floating point unit to complete addition process. As frequency is 262.006MHz so one clock cycle completes 3.82ns and 15 clock cycles completes in 3.82ns x 15 =57.3ns. Therefore the addition process completes in 57.3ns.

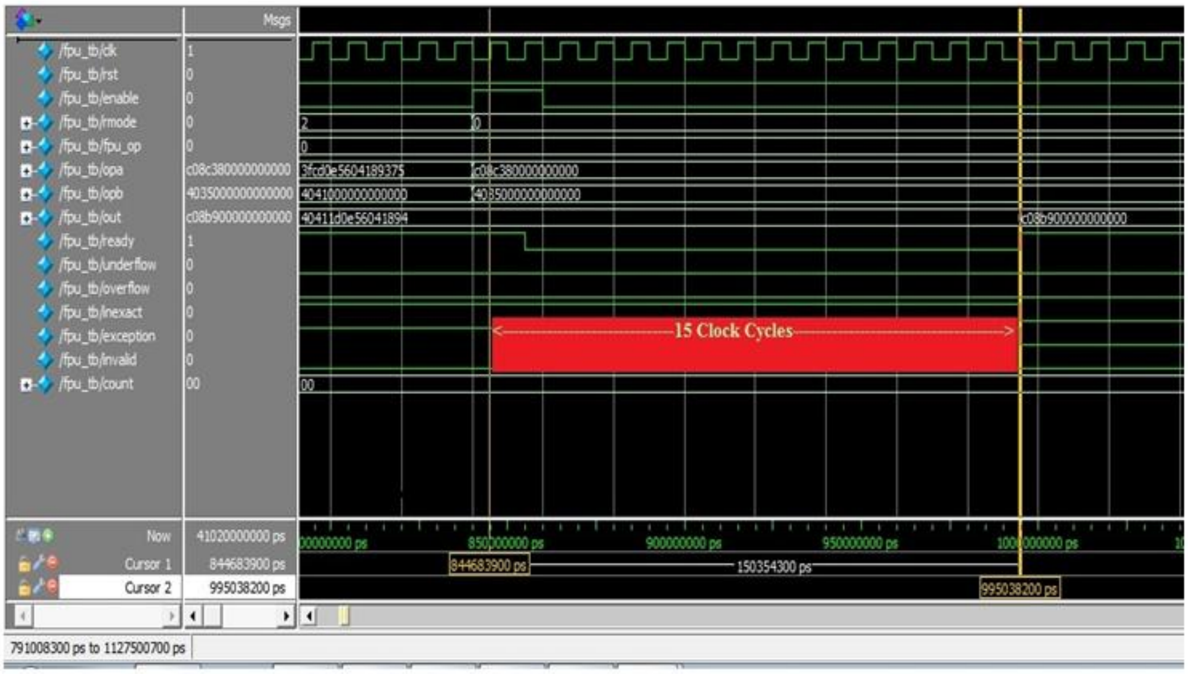

Fig 5: Simulation Result of Floating Point Addition

3.3.2) Simulation result of floating point subtraction- It is calculated for the two input operands of 64 bits each. 15 clock cycles are required by floating point unit to complete subtraction process. As frequency is 262.006MHz so one clock cycle completes 3.82ns and 15 clock cycles completes in 3.82ns x 15 =57.3ns. Therefore the subtraction process completes in 57.3ns.

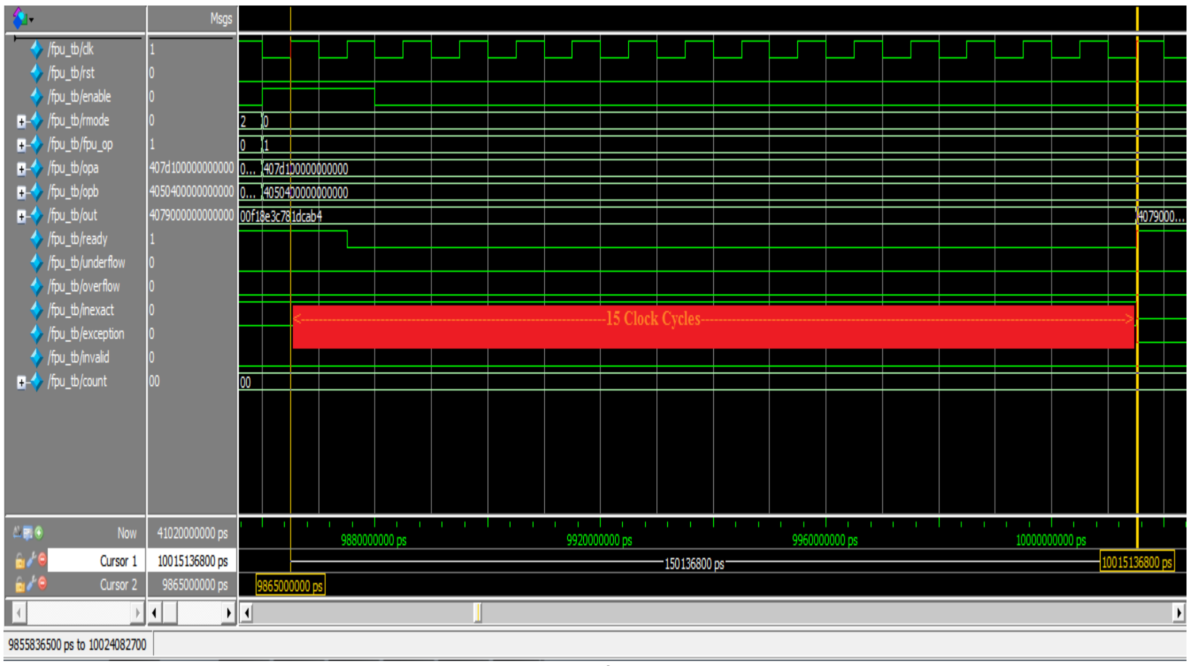

Fig 6: Simulation Result of Floating Point Subtraction

3.3.3) Simulation result of floating point multiplication- It is calculated for the two input operands of 64 bits each. 15 clock cycles are required by floating point unit to complete multiplication process. As frequency is 262.006MHz so one clock cycle completes 3.82ns and 15 clock cycles completes in 3.82ns x 15 =57.3ns. Therefore the multiplication process completes in 57.3ns.

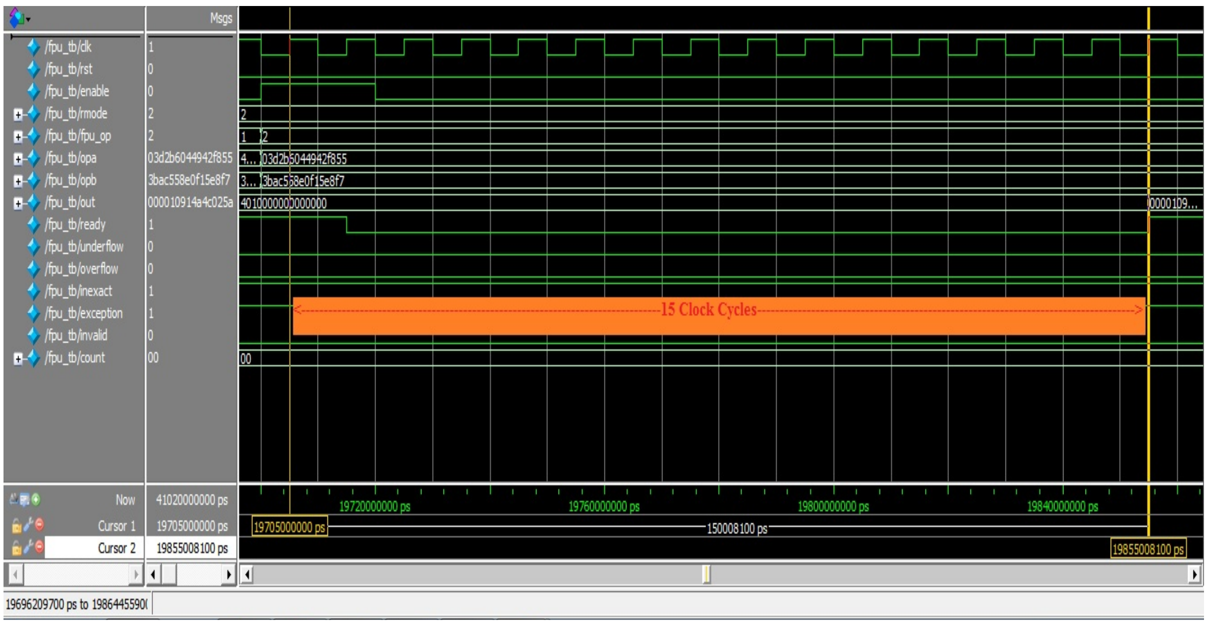

Fig 7: Simulation Result of Floating Point Multiplication

3.3.4) Simulation result of floating point division- It is calculated for the two input operands of 64 bits each. 68 clock cycles are required by floating point unit to complete division process. As frequency is 262.006MHz so one clock cycle completes 3.82ns and 68 clock cycles completes in 3.82ns x 68 =259.76ns. Therefore the division process completes in 259.76ns.

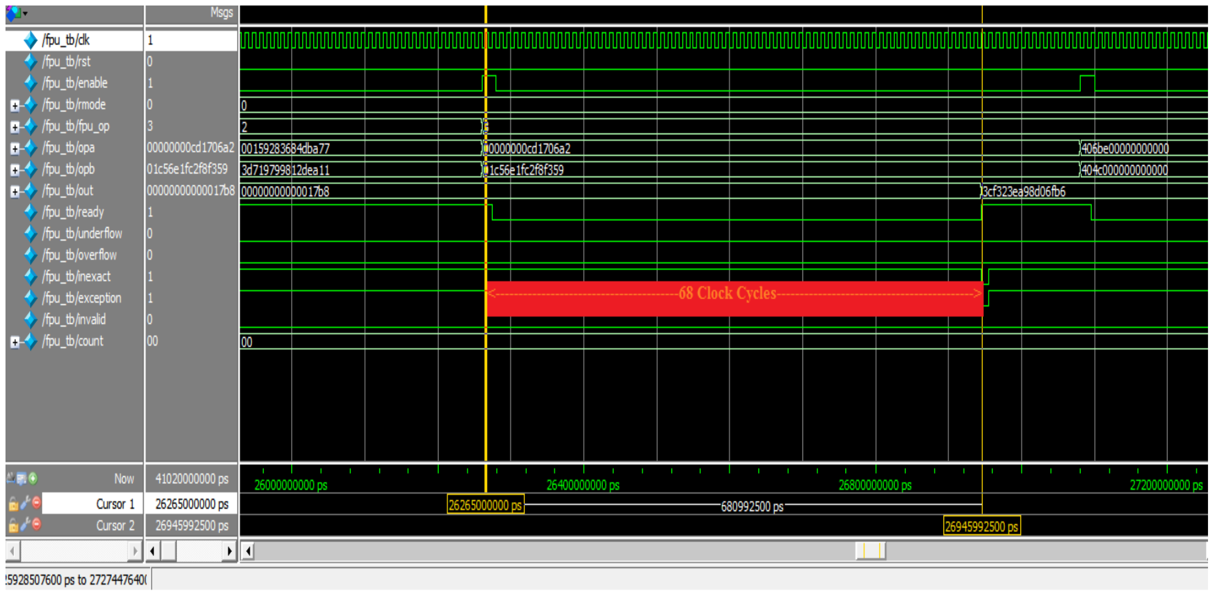

Fig 8: Simulation Result of Floating Point Division

IV. Conclusion

This paper presents a high performance implementation of double precision floating point arithmetic unit. The complete design is captured in Verilog Hardware description language (HDL), tested in simulation using Questa Sim, placed and routed on a Vertex 5 FPGA from Xilinx.It works on Maximum Frequency of 262.006MHz. When synthesized, this module used 16% number of slice registers, 22% Number of Slice LUTS, and 36% number of fully used LUT-FF pairs. The overall performance is increased in this design. The proposed work can be further proceed by adding some more modules like square root, logarithmic units to the floating point unit and also the complete design can be implemented on high performance vertex-6 FPGA.

References

[1]. Chaitanya a. Kshirsagar, P.M. Palsodkar “An FPGA implementation of IEEE - 754 double precision floating point unit using verilog” international journal of electrical, electronics and data communication, ISSN: 2320-2084 , volume-2, issue-6, june-2014

[2]. Paschalakis, S., Lee, P., “Double Precision Floating-Point Arithmetic on FPGAs”, In Proc. 2003 2nd IEEE International Conference on Field Programmable Technology (FPT ’03), Tokyo, Japan, pp. 352-358, 2003

[3]. Addanki Puma Ramesh, A. V. N. Tilak, A.M.Prasad “An FPGA Based High Speed IEEE-754 Double Precision Floating Point Multiplier Using Verilog” 2013 International Conference on Emerging Trends in VLSI, Embedded System, Nano Electronics and Telecommunication System (ICEVENT), pp. 1-5, 7-9 Jan. 2013

[4]. Ushasree G, R Dhanabal, Sarat Kumar Sahoo “Implementation of a High Speed Single Precision Floating Point Unit using Verilog” International Journal of Computer Applications National conference on VSLI and Embedded systems, pp.32-36, 2013

[5]. Pramod Kumar Jain, Hemant Ghayvat , D.S Ajnar “Double Precision Optimized Arithmetic Hardware Design For Binary & Floating Point Operands” International Journal of Power Control Signal and Computation (IJPCSC) Vol. 2 No. 2 ISSN : 0976-268X

[6]. Basit Riaz Sheikh and Rajit Manohar “An Operand-Optimized Asynchronous IEEE 754 Double-Precision Floating-Point Adder”, IEEE Symposium on Asynchronous Circuits and Systems (ASYNC), pp. 151 – 162, 3-6 May 2010

[7]. Ms. Anjana Sasidharan, Mr. M.K. Arun” Vhdl Implementation Of Ieee 754 Floating Point Unit” IJAICT, ISSN 2348 – 9928Volume 1, Issue 2, June 2014

[8]. 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

[9]. Tarek Ould Bachir, Jean-Pierre David “Performing Floating-Point Accumulation on a modern FPGA in Single and Double Precision” 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, pp.105-108, 2010

[10]. Geetanjali Wasson “IEEE-754 compliant Algorithms for Fast Multiplication of Double Precision Floating Point Numbers” International Journal of Research in Computer Science, Volume 1, Issue 1, pp. 1-7, 2011

[11]. KavithaSravanthi, Addula Saikumar “An FPGA Based Double Precision Floating Point Arithmetic Unit using Verilog” International Journal of Engineering Research & Technology ISSN: 2278-0181, Vol. 2 Issue 10, October - 2013

[12]. 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

[13]. H. Yamada, T. Hottat, T. Nishiyama, F. Murabayashi, T. Yamauchi, and H. Sawamoto “A 13.3ns Double-precision Floating-point ALU and Multiplier”, IEEE International Conference on Computer Design: VLSI in Computers and Processors, pp. 466 – 470, 2-4 Oct 1995

[14]. 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