Volume 2, Issue 10, October-2015, pp. 881-888 ISSN (O): 2349-7084

International Journal of Computer Engineering In Research Trends

Available online at: www.ijcert.org

Implementation of Double Precision Floating Point Multiplier in VHDL

[1] SUNKARA YAMUNA RANI, [2] BELLAM VARALAKSHMI

[1](M.Tech) VLSI, Dept. of ECE
[2]Assistant Professor, Dept. of ECE
Priyadarshini Institute of Technology & Management

Abstract:- Floating point arithmetic is widely used in many areas. IEEE Standard 754 floating point is the most common representation today for real numbers on computers. IEEE standards specify a set of floating point formats for single precision and double precision. Low power consumption and smaller area are some of the most important criteria for the fabrication of DSP systems and high performance systems. FPGAs are increasingly being used in the high performance and scientific computing community to implement floating-point based hardware accelerators. FPGAs are generally slower than their application specific integrated circuit (ASIC) counterparts, as they can't handle as complex a design, and draw more power. However, they have several advantages such as a shorter time to market, ability to re-program in the field to fix bugs, and lower nonrecurring engineering cost costs. Vendors can sell cheaper, less flexible versions of their FPGAs which cannot be modified after the design is committed. Double precision floating point multiplier using three stage pipelining technique achieved the maximum frequency of 489.045 MHz with minimum delay 2.045 ns and area of 888 slices. Double precision FPM targeted on a Xilinx Virtex-6 xc6vlx75t-3ff484 device. This pipelined floating point multiplier performs the rounding operation and also handles various exceptions conditions. The double precision floating point multiplier was simulated in ISE simulator and synthesized using Xilinx ISE 13.2 tool. Key Words: Single Precision

Keywords – Double precision, floating point, adder/subtractor, multiplier, FPGA, IEEE-754, Virtex-6

1. INTRODUCTION

Double precision floating point numbers are 64-bit binary numbers. The 64-bits are divided into 3 parts-sign, exponent and mantissa. The 52 least significant bits (LSBs) are used to represent the mantissa of the number. The next 11-bits are used to represent the exponent of the number. The most significant bit (MSB) of the number is used as a sign bit to represent the sign of the number.

- Sign bit '0' indicates positive number.
- Sign bit '1' indicates negative number.

Floating point number system is a common choice for many scientific computations due to its wide dynamic range feature. For instance, floating point arithmetic is widely used in many areas, especially in scientific computation, image processing and signal processing [1]. 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 or binary point can float. There are also representations in which the number of digits before and after the decimal or binary point is fixed, called fixed-point representations [2]. The advantage of floating-point representation over fixed point representation is that it can support a much wider range of values [3]. The float-ing point numbers is based on scientific notation. A scientific notation is just another way to represent very large or very small numbers in a compact form such that they can be easily used for computations.

Floating point number consists of three fields:

1. Sign (S): It used to denote the sign of the number i.e.0 represent positive number and 1 represent negative number.
2. Mantissa (M): Mantissa is part of a floating point number which represents the magnitude of the number.
3. Exponent (E): Exponent is part of the floating point number that represents the number of places that the decimal point (or binary point) is to be moved. Number system is completely specified by specifying a suitable base β, mantissa M, exponent E and sign S. Floating point number F has the value.

$$F = (-1)^S M \beta^E \quad (1)$$

The most common representation of exponent is as a biased exponent. A biased exponent is one that is obtained by adding a constant value to the original exponent.

According to which E is $E = E^{true} + bias$ (2)

where bias is constant and Etrue is true value of exponent. The choice of the bias is made depending on the number of bits available for representing exponents in the floating point for-mat used. If the number of bits allowed for exponent representation is n, the bias is 2n-1 - 1.The way floating point operations are executed depends on the data format of the operands. IEEE standards specify a set of floating point data formats, single precision and double precision. Double precision consists of 64 bits. Figure 1 shows the IEEE double precision data formats.

2. FLOATING POINT MULTIPLIER

Multiplication of two floating point numbers is a complex task and is carried out in a series of steps. Since a floating point number consists of 3 parts- sign, exponent and mantissa, calculations for all the parts are carried out separately.

2.1 Calculation of Sign

The sign bit of the resultant is obtained by carrying out the EXOR operation of the sign bits of the two operands. Sign bit '0' represents a positive sign and sign bit '1' represents a negative sign.

2.2. Calculation of Exponent

The exponents of both the operands are represented in the IEEE 754 format, i.e., a bias of 1023 is added to both the exponents. To calculate the exponent of the resultant the bias of 1023 must be removed from the exponents. After removal of bias from the exponents, both are added to give the resultant exponent. This resultant exponent is in unbiased form. So to represent it in IEEE 754 format, it should be converted to the biased form by adding bias of 1023 to it.

2.3. Calculation of Mantissa

Mantissa calculation is the most complex part of floating point multiplication. A 64-bit number contains 52-bit mantissa. The resultant mantissa[15] is calculated by multiplying the mantissas of both the operands. But before the multiplication is carried out, the mantissas of both the operands need to be normalized. Normalization is done in order to ensure that either of the numbers to be multiplied is not zero. If any one of the numbers is zero than the resultant will be zero. If both the numbers are zero than the resultant will be undefined or Not a Number (NaN). Normalization is done by adding a '1' as the MSB of the mantissa. By adding a '1' as MSB, the possibility of the number being a zero is eliminated. After normalization, the number of bits in the mantissa is increased by one, so the normalized mantissa contains 53-bits. The next step after normalization is multiplication of the normalized mantissas. Two 53-bits mantissas are multiplied and Mantissa calculation is the most complex part of floating point multiplication. A 64-bit number contains 52-bit mantissa. The resultant mantissa is calculated by multiplying the mantissas of both the operands. But before the multiplication is carried out, the mantissas of both the operands need to be normalized.

Normalization is done in order to ensure that either of the numbers to be multiplied is not zero. If any one of the numbers is zero than the resultant will be zero. If both the numbers are zero than the resultant will be undefined or Not a Number (NaN). Normalization is done by adding a '1' as the MSB of the mantissa. By adding a '1' as MSB, the possibility of the number being a zero is eliminated. After normalization, the number of bits in the mantissa is increased by one, so the normalized mantissa contains 53-bits. The next step after normalization is multiplication of the normalized mantissas.

Two 53-bits mantissas are multiplied and a resultant of 106-bits is obtained. There are several different algorithms which can be used to carry out the multiplication of the mantissas. As the size of the mantissa is very large it is convenient to use an algorithm rather than multiplying directly. This 106-bits resultant cannot be stored directly into the output because of its size. The output mantissa must contain only 52-bits. To obtain 52-bits mantissa, normalization of the 106-bits resultant is carried out. In normalized form the MSB of the number must be 1. Therefore all the '0' bits before the first '1' bit are discarded. Now the mantissa is in normalized form with a '1' as MSB. Now to extract the final 52-bits, de-normalization is carried out. This is done because the mantissa that is finally stored in IEEE 754 format is not in the normalized form as the integer part of the output is by default 1. So the first bit of the number, i.e. 1, is discarded and the next 52 bits are stored as the mantissa of the output, also discarding the remaining least significant bits.

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]:
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

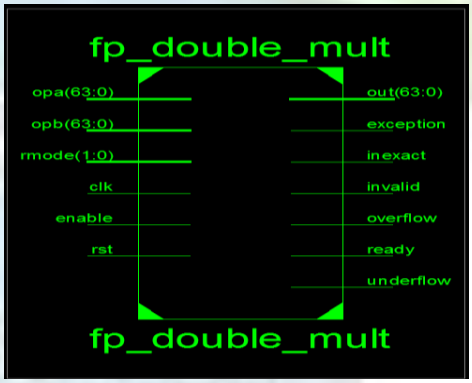
5. Normalizing the result; i.e. obtaining 1 at the MSB of the results significant

6. Rounding the result to fit in the available bits

7. Checking for underflow/overflow occurrence

The mantissa of operand A and the leading 1 (for normalized numbers) are stored in the 53-bit mulA signal. The mantissa of operand B and the leading 1 (for normalized numbers) are stored in the 53-bit mulB signal. Multiplying all 53 bits of mulA by 53 bits of mulB 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 double precision multiplier module breaks up the multiply into smaller 24-bit by 17-bit multiplies. The Xilinx Virtex-6 device contains DSP48E1 slices with 25 by 18 twos' complement multipliers, which can perform a 24-bit by 17-bit unsigned multiply. The breakdown of the multiply in multiplier module is done as follows:

Product a = mulA [23:0] x mulB [16:0]
Product b = mulA [23:0] x mulB [33:17]
Product c = mulA [23:0] x mulB [50:34]
Product d = mulA [23:0] x mulB [52:51]
Product e = mulA [40:24] x mulB [16:0]
Product f = mulA [40:24] x mulB [33:17]
Product g = mulA[40:24] x mulB[52:34]
Product h = mulA [52:41] x mulB [16:0]
Product i = mulA [52:41] x mulB [33:17]
Product j = mulA [52:41] x mulB [52:34]



The products (a-j) are added together, with the appropriate offsets based on which part of the mulA and mulB arrays they are multiplying. The final 106-bit product is stored in product signal. In this paper a floating point multiplier in which rounding support isn't implemented. Rounding support can be added as a separate unit that can be accessed by the multiplier or by a floating point adder, thus accommodating for more precision if the multiplier is connected directly to an adder in a MAC unit. Figure 1 shows the multiplier structure; Exponents addition, Significant and multiplication, and Results sign calculation are independent and are done in parallel. The significand multiplication is done on two 24 bit numbers and results in a 48 bit product, which we will call the intermediate product (IP). The IP is represented as (47 down to 0) and the decimal point is located between bits 46 and 45 in the IP [4].

2.4 Normalizer

The result of the significand multiplication (intermediate product) must be normalized to have a leading '1'just to the left of the decimal point. Since the inputs are normalized numbers then the intermediate product has the leading one at bit 104 or 105.The output will be left/right shifted if there is not a '1' in the MSB of product. If changes are made by shifting then corresponding changes has to be made in exponent also. Moving the radix point one place to the left increments the exponent by 1; moving one place to the right decrements the exponent by 1. The mantissa output from the double precision multiplier module is in 56-bit product result signal. The MSB[14] is a leading '0'.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 follow. Two extra bits follow the mantissa, and are used for rounding purposes.

2.5 Rounding Module

The result of floating point multiplication, however, must fit into the same n bits as the multiplier and the multiplicand. This, of course often leads to loss of precision. The IEEE standard attempted to keep this loss as minimal as possible with introduction of standard rounding modes [21]. The IEEE standard specifies four rounding modes round to nearest even, round to zero, round to positive infinity, and round to negative infinity [5].The rounding operation is performed in the rounding module. The inputs to the rounding module from the previous stage 1(multiplier module) and sign_r(1 bit), mantissa_r(56 bits), and exponent_r(12 bits) for double precision floating point multiplier.

The mantissa_r includes an extra '0' bit as the MSB, and two extra remainder bits as LSB's, and in the middle are the leading '1' and mantissa bits (52 bits) for double precision. The exponent_r has an extra '0' bit as the MSB so that an over-flow from the highest exponent (2047) will be caught. To per-form a rounding, rmode (1:0) signal and two extra bits follow the mantissa are used. Based on the rounding, a change to the mantissa result corresponding changes has to be made in the exponent result also. The output from the rounding module is a 64-bit value in the round_out register and 12 bit value in the exponent_final register. This is passed to the (stage 3) exceptions module.

2.6 Exceptions Module

Special cases are checked in exceptions module. The individual output signals of underflow, overflow, invalid, exception, and inexact will be high if the conditions for each case exist. The special cases are:

- Multiply 0 by infinity
- Multiply overflow
- Multiply underflow
- One or both inputs are QNaN
- One or both inputs are SNaN

3. Pipelined Double Precision Floating Point Multiplier

The aim in developing a floating point multiplier was to be pipeline that is each module in order to produce result at every clock cycle. In order to enhance the performance of the multiplier, three pipelining stages are used to divide the critical path thus increasing the operating frequency of the multiplier. 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. By pipelining the floating point multiplier, the speed increased, however, the area increased as well.

Different coding structures were tried in VHDL code in order to minimize size[13]. Multiplying two numbers in floating point for-mat is done by three main module i.e. multiplier module, rounding module and exceptions module. Figure 2 shows the Pipelined floating point multiplier with rounding and exceptions module. The multiplier, rounding and exceptions module that are implement independent and are done in parallel. All the modules in the FPM are realized using VHDL. In this pa-per, pipelined floating point multiplier structure organized in a three-stage. Stage 1 performs the addition of the exponents, the multiplication of the mantissas, and the exclusive-or of the signs. Stage 2 takes these results from stage 1 as inputs, per-forms the rounding operations. Stage 3 checks the various exceptions conditions like overflow, underflow, invalid and in-exact.

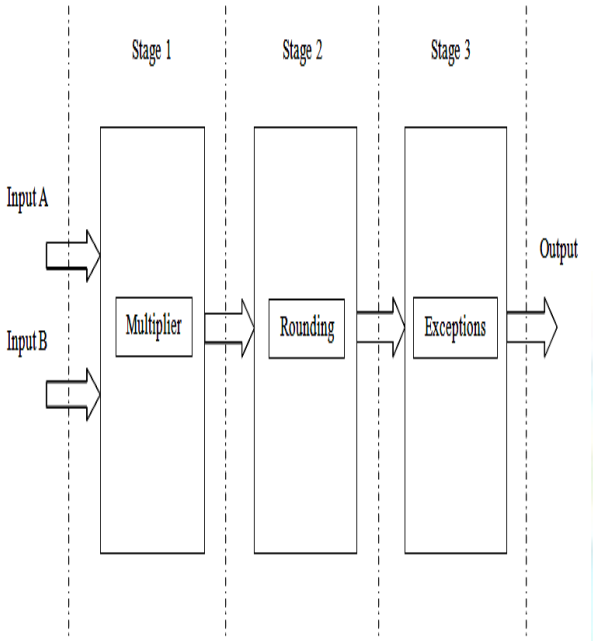


Fig 1: Pipelined Floating Point Multiplier with Rounding and Exceptions

3.1 Double Precision:

In case of double precision 64 bits format, Mantissa is represented in 52 bits, 1 bit is added to the MSB for normalization, Exponent is represented in 11 bits which is biased to 1023 and MSB of double precision is reserved for sign bit as shown in Figure 1 [7].

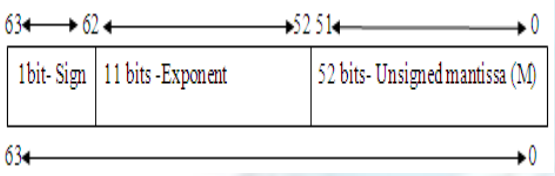


Fig 2: Single Precision Floating-Point IEEE Formats

The value of the floating point number represented in double precision format is $F= (-1)^S \ 1.M \ 2^{E-1023}$ (3) where 1023 is the value of bias in double precision data format. Exponent E ranges between 1 to 2046, and E = 0 and E = 2047 are reserved for special values. The double precision format offers range from 2-1023 to 2+1023, which is equivalent 10-308 to10308 [6].

4. PROPOSED METHODOLOGY

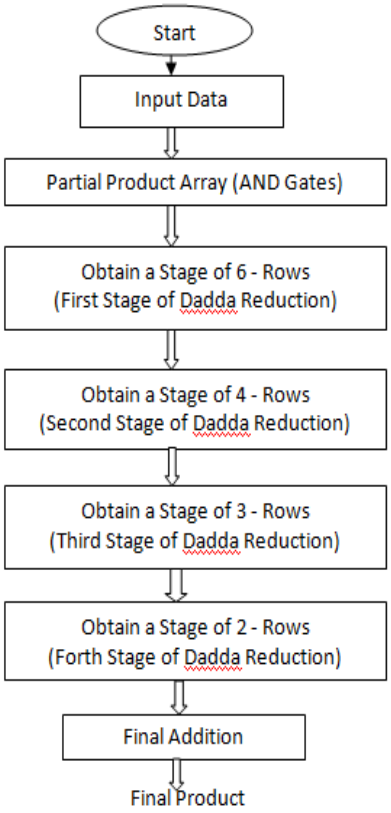


Fig.3 Flow Diagram of 8x8 Dadda Multiplier

We have followed a similar approach as [8] for designing the basic algorithm for this implementation. The floating point arithmetic in [9] is two stage pipe lined which are divided into two paths, namely "R-Path" and "N-Path". The two paths are selected on the basis of the exponent difference. Dadda proposed a sequence of matrix heights that are fixed to give the minimum number of reduction stages. For Dadda Multipliers there are N=8 bits. Dadda Multiplier uses partial product bits.

The calculation diagram for an 8X8 Dadda Multiplier [10] is shown in fig.4 the 8x8 multiplier takes 4 reduction stages, with matrix height 6, 4, 3 and 2. The reduction uses 35 (3, 2) counters (full adder), 7 (2, 2) counters (half adder) and a 14- bit carry propagate adder.

This algorithm is broken into two pipeline stages, which are executed in two different clock cycles. The advantage of the pipelining mechanism is that, despite having a higher input output sequential length, they offer an unmatched throughput by virtue of their

assembly line structure. An overview of the proposed algorithm is explained by Figure 4.

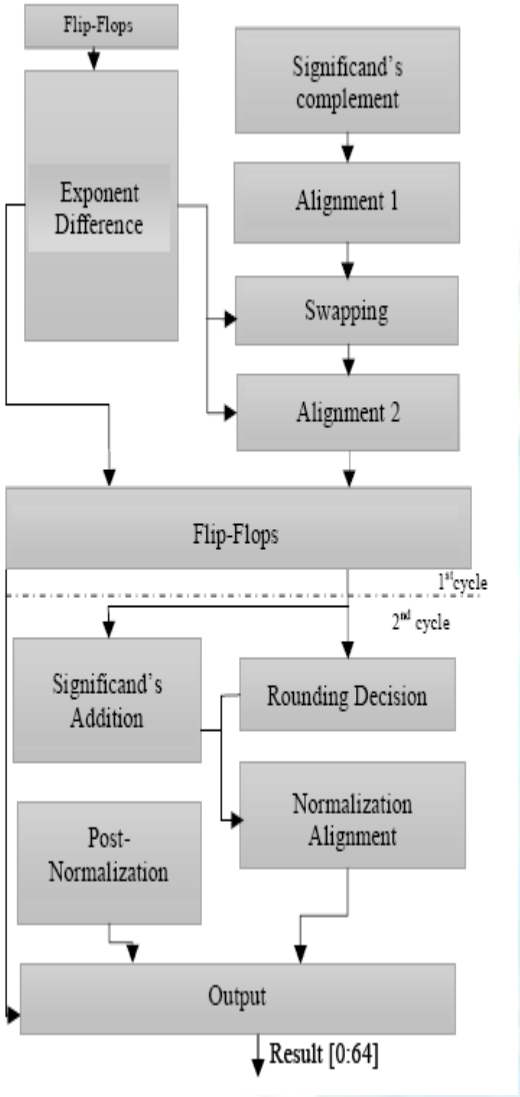


Fig 4. IEEE-754 double precision format

4.1. First Clock Cycle Operation

This is the first stage in the pipeline mechanism. The components of the Floating Point number, in terms of bit vector, are, (S, E [0:10], F [0:52]) The basic algorithm operates only with normalized FP numbers, i.e. f ε [1, 2]. The basic operation is performed within two clock stages, and is determined by the parameter. SOP ε {0, l} It is supplied as an input to the algorithm. The mathematical operation to be performed is determined by calculating the effective sign of operation. After this, some initial preprocessing operations are done before adding or subtracting the two numbers. The exponent difference is obtained and is represented as δ=ea-eb then the number with the smaller magnitude[11] is sorted out through various operations based on conditions derived from the effective sign and the resultant of the exponent difference. In case the exponent difference is in the range [-63, 64] the smaller significand is shifted by MAG_MED positions to the right.

4.2 Second Clock Cycle Operation

This is the second stage of the pipelining mechanism. The two "preprocessed" significands are added and the result is rounded in accordance with the IEEE standard rounding algorithm. Here the rounding algorithm from [12] has been implemented. At the end, it is normalized. The output result is a 64 bit binary floating point number. rnd (sum) = rnd ((-1)sa .2ea .fa+(-1)(SOP+sb) .2eb .fb (4)

4.3 Algorithm for Addition

Let s1; e1; f1 and s2; e2; f2 be the signs, exponents, and significands of two input floating-point operands, N1 and N2, respectively. Given these two numbers, Figure 4 shows the flowchart of the standard floating-point adder algorithm. A description of the algorithm is as follows.

1. The two operands, N1 and N2 are read in and compared for denormalization and infinity. If numbers are denormalized, set the implicit bit to 0 otherwise it is set to 1. At this point, the fraction part is extended to 53 bits.
2. The two exponents, e1 and e2 are compared using 8-bit subtraction. If e1 is less than e2, N1 and N2 are swapped i.e. previous f2 will now be referred to as f1 and vice versa.
3. The smaller fraction, f2 is shifted right by the absolute difference result of the two exponents' subtraction. Now both the numbers have the same exponent.
4. The two signs are used to see whether the operation is a subtraction or an addition.

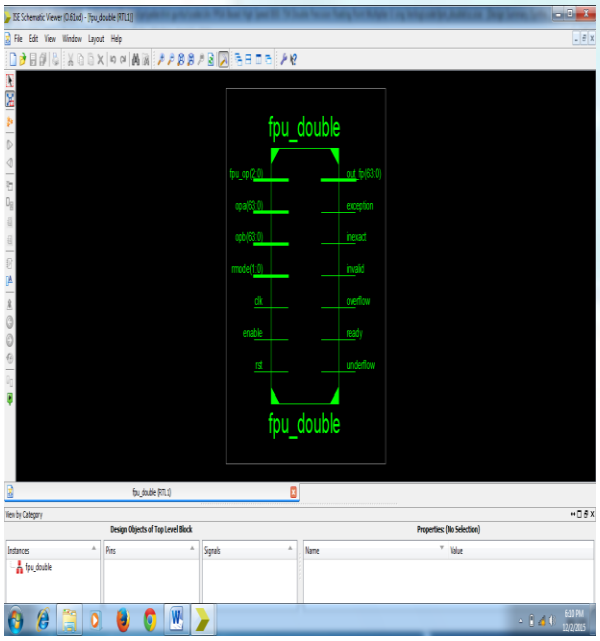
5. If the operation is a subtraction, the bits of the f2 are inverted.
6. Now the two fractions are added using a 2's complement adder.
7. If the result sum is a negative number, it has to be inverted and a 1 has to be added to the result.

8. The result is then passed through a leading one detector or leading zero counter. This is the first step in the normalization step.
9. Using the results from the leading one detector, the result is then shifted left to be normalized. In some cases, 1-bit right shift is needed.
10. The result is then rounded towards nearest even, the default rounding mode.
11. If the carry out from the rounding adder is 1, the result is left shifted by one.

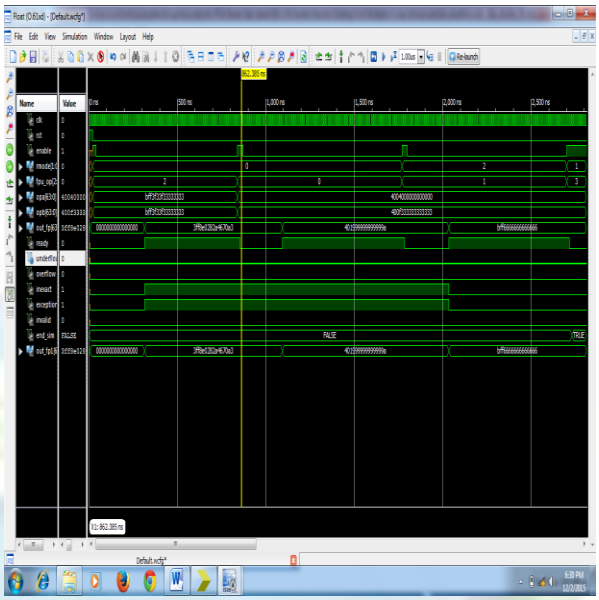
5. RESULTS

The previous sections have discussed design for floating-point Multiplier using Wallace tree multiplier. This is implemented in VHDL targeted on a Xilinx Virtex-6 device. The functionality of the implementations is verified by performing a simulation with some random input vectors. In order to evaluate designs, the area, critical path latency, throughput, and power consumption are verified. The double precision floating point adder/subtractor and multiplier designs were simulated in Modelsim 6.6c and synthesized using Xilinx ISE 12.2i which are mapped on to Virtex-6 FPGA. Where as in case of XiI in x core, it occupies an area of 266 slices and its operating frequency is 221.484 MHz. So the implemented design provides high operating frequency with more accuracy.

RTL SCHEMATIC



OUTPUT



6. Conclusion

This paper has successfully demonstrated an implementation of a high speed, IEEE 754, double precision floating point adder with a significant decrease in latency. This manifest in the fact that FPGA based embedded systems has a higher advantage of lower computational aspects. Also, an implementation work of this algorithm, on the Xilinx Spartan-3 FPGA would give results with further improvement. The double precision floating point adder/subtractor and multiplier supports the IEEE-754 binary interchange format, targeted on a Xilinx Virtex-6 xc6vlx75t-3ff484 FPGA. The designs achieved the operating frequencies of 363.76 MHz and 414.714 MFLOPs with an area of 660 and 648 slices respectively. This design handles the overflow, underflow, and truncation rounding mode.

REFERENCES

[1] V. 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, vol. 2, no. 1, (1994) March, pp. 124–128.

[2] P. Belanovic and M. Leeser, "A Library of Parameterized Floating-Point Modules and Their Use", in 12th International Conference on Field-Programmable Logic and Applications (FPL-02).

London, UK: Springer-Verlag, (2002) September, pp. 657–666.

[3] K. Hemmert and K. Underwood, "Open Source High Performance Floating-Point Modules", in 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM-06), (2006) April, pp. 349–350.

[4] A. Malik and S. -B. Ko, "A Study on the Floating-Point Adder in FPGAs", in Canadian Conference on Electrical and Computer Engineering (CCECE-06), (2006) May, pp. 86–89.

[5] D. Sangwan and M. K. Yadav, "Design and Implementation of Adder/Subtractor and

Multiplication Units for Floating-Point Arithmetic", in International Journal of Electronics Engineering, (2010), pp. 197-203.

[6] M. K. Jaiswal and R. C. C. Cheung, "High Performance FPGA Implementation of Double Precision Floating Point Adder/Subtractor", in International Journal of Hybrid Information Technology, vol. 4, no. 4, (2011) October.

[7] B. Fagin and C. Renard, "Field Programmable Gate Arrays and Floating Point Arithmetic", IEEE Transactions on VLSI, vol. 2, no. 3, (1994), pp. 365–367.

[8] 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), (1995), pp. 155–162.

[9] L. Louca, T. A. Cook and W. H. Johnson, "Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs", Proceedings of 83rd IEEE Symposium on FPGAs for Custom Computing Machines (FCCM" 96), (1996), pp. 107–116.

[10] A. Jaenicke and W. Luk, "Parameterized Floating-Point Arithmetic on FPGAs", Proc. of IEEE ICASSP, vol. 2, (2001), pp. 897-900.

[11] B. Lee and N. Burgess, "Parameterisable Floating-point Operations on FPGA", Conference Record of the Thirty-Sixth Asilomar Conference on Signals, Systems, and Computers, (2002).

[12] M. Al-Ashrafy, A. Salem, W. Anis, "An Efficient Implementation of Floating Point Multiplier".

[13] L. Louca, T. A. Cook, and W. H. Johnson, "Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs," Proceedings of 83rd IEEE Symposium on FPGAs for Custom Computing Machines (FCCM"96), pp. 107-116,1996.

[14] A. Jaenicke and W. Luk, "Parameterized Floating-Point Arithmetic on FPGAs", Proc. of IEEE lCASSP, 2001, vol. 2, pp. 897-900.

[15] B. Lee and N. Burgess, "Parameterisable Floating-point Operations on FPG A," Conference Record of the Thirty Sixth Asilomar Conference on Signals, Systems, and Computers, 2002.

