IEEE 754 Single Precision Floating Point Arithmetic Unit Using VHDL

1. Omendra Singh, 2.Asst.Prof.Ankit Trivedi

Student, Department of Electrical Engineering, Axis Institute of Technology&Management,Kanpur,UttarPradesh,India

Asst. Prof., Department of Electronic Engineering, Axis Institute of Technology&Management,Kanpur,UttarPradesh,India

Abstract:-We all know that Arithmetic is used such operation as addition, subtraction, multiplication and division. Many scientific applications require more accuracy in result. For that reason,we have explored Floating point unit, have different operations almost all of the algorithms implemented in FPGA(Field programmablegate array) kit by use of Xilinx ISE compiler & algorithm are design in VHDL language.Main advantage of floating-point representation is that it can support a much wider range of values, over fixed-point and integer representation Floatingpoint arithmetic unit with IEEE 754 Standard has been designed using VHDL code and all operations of addition,subtraction, multiplication and division are analyses on Xilinx.

Keywords: Floating point arithmetic, algorithm, IEEE754 format,simulation, VHDL**,** Single precision(32 bit)

1. Introduction:-

In, now days many fields of science, engineering and finance require manipulating real numbers efficiently. Since in the first computers appeared, many different ways of approximating real numbers on it have been introduced. In one of them, the floating point arithmetic, is clearly the most efficient way of representing real numbers in computers.The main concept of floating-point representation consist purely of significant is that over integer fixed-point numbers that expanding it with the exponent component achieves the greater range.The floating point actually refers to that a radix point of any number , (decimal point, or, more commonly in computers, binary point) can "float"; meaning that it can be laced anywhere in relation to the significant digits of the number.

In this paper, IEEE754-single precision (32-bit) high speed floating point arithmetic unit is designed using VHDL code. Flowcharts and algorithms have been presented and all operations of addition, subtraction, multiplication anddivision are tested on Xilinx and verified successfully. Thereafter, the new feature of creating test-bench forverification of VHDL code of that 32-bit Floating Point

Arithmetic Unit in Xilinx simulator has been explained and be givenoperation results of simulation are demonstrated.

2. The Standard IEEE 754

IEEE 754 standard have specifies formats and different methods in order to operate with floating point arithmetic. These methods used for computational with floating point numbers will give sameresult regardless the processing is done in hardware, software or a combinationfor the two or the implementation.

The standard specifies are given below.

• Formats for floating point data as binary and decimal for computation and data interchange.

· Such as different operations as addition, subtraction, multiplication and other operations.

· There are conversion between integer-floating point formats and the other way around.

· There are be given different properties to be satisfied when rounding numbers during arithmetic and conversions.

· Floating point exceptions and their handling (NaN, ∞ or zero)

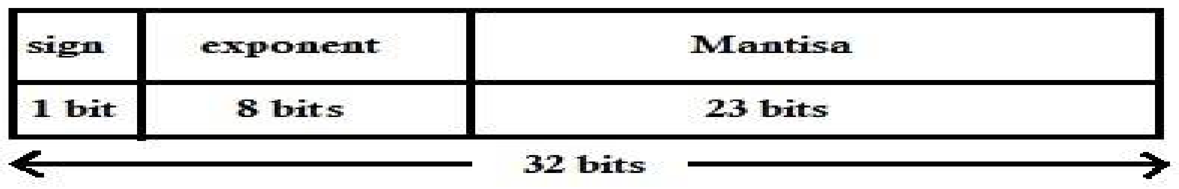
• IEEE 754 standardshave specifies four different formats to represent the floating point values.

· Single precision floating (32 bits)

· Double precision floating (64 bits)

3.IEEE754 standard floating point format

IEEE 754 standard give format by presents two different floating point formats that are Binary interchange format and Decimal interchange format. Where fig.1 shows the IEEE 754 single precision binary format representation; it consists of a one bit sign (S), an eight bit exponent (E), and a twenty three bit fraction (M or Mantissa). If the exponent is greater than 0 and smaller than 255, and there is 1 in the MSB of the significant then the number is said to be a normalized number.In this formats real number is represented by (1)



sign	exponent	Mantisa
1 bit	8 bits	23 bits

32 bits

Figure1. IEEE 754 single precision format

Sign: Where 1 bit is wide and used to denote the sign of the number and 0 bit indicate positive number and 1 represent negative number.

Exponent: In this 8 bit wide and signed exponent in excess-127 representation and exponent field represents both positive and negative exponents.

Mantissa: In this, 23 bits is wide and fractional component.

The single- precision floating-point number is calculated as **(-1) S × 1.F × $2^{(E-127)}$**

The classes of floating point single precision numbers are asfollows:

Normalized numbers: the bias is $2^8-1 = 127$; therange of the exponent is [−126:127], while its binary value is in the range [1:254]

Infinities &NaN: these special representation have abinary value of $2^8 - 1 = 256-1=255$ for the exponent.

FPU Architecture:-

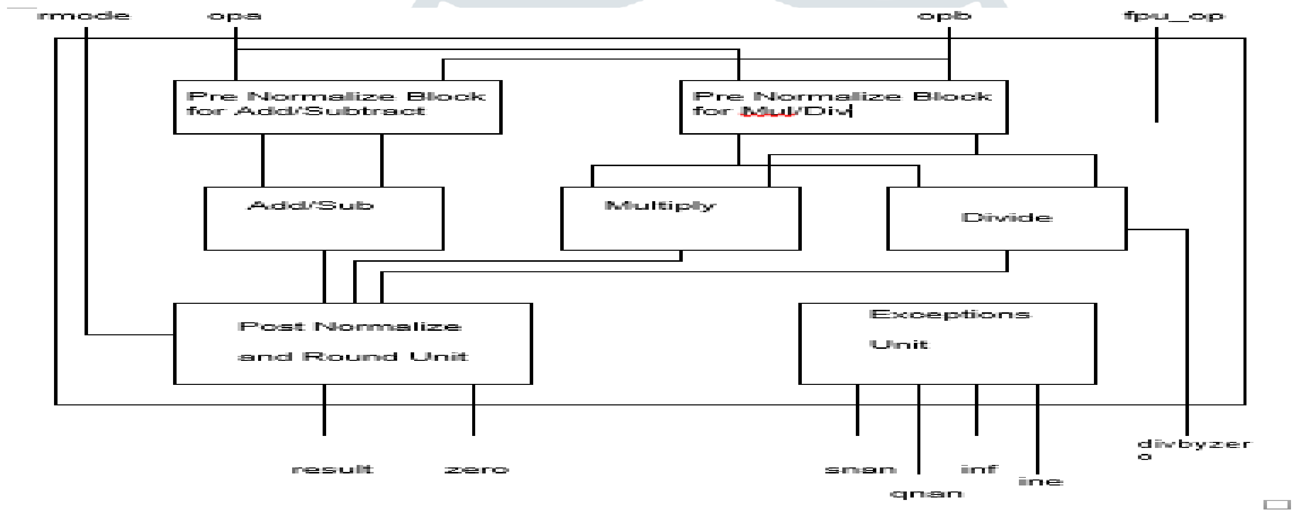

rmode
opa
opb
fpu_op
Pre Normalize Block for Add/Subtract
Pre Normalize Block for Mul/Div
Add/Sub
Multiply
Divide
Post Normalize and Round Unit
Exceptions Unit
result
zero
snan
qnan
inf
ine
divbyzero

Fig 2 :FPU Architecture

Figure 2 gives the architecture of floating point unit for single precision floating point unit. Two pre normalization units adjust the fractions. Where one is done for add and subtract operation and other for multiply and divide operation.

4. ALGORITHMS FOR FLOATING POINT ARITHMATIC UNIT

In the following section the algorithms are written by using flowchart for addition, subtraction, multiplication anddivision which are helpful in developing VHDL code for 32 bit single precision floating point format.

4.1. Floating Point Addition / Subtraction

The flowchart explains the algorithm for addition and subtraction. Depending on the sign of these numbers two cases arises.

In Case I- When both numbers are of same sign

Steps:-1:- Enter the two numbers in1 and in2 each of 32 bits. expo1, S1 and expo2, S2 indicates exponent and significant of in1 and in2 respectively.

2:- Check if expo1 or expo2 is zero, if it is zero then set hidden bit of respective input number '0', if expo1 and expo2 is not '0' check if expo2>expo1 swap in1 and in2 else keep them as it is.

3:- Now calculate the difference between expo1 and expo2.let's say this difference is d,

d= expo1-expo2. If difference'd' is zero then there is no change else left shift significant of in2 which is S2 by the amount 'd' along with the hiddenbit which us '1'

4:- Normalize the exponent of in2 by adding the amount d, expo2=expo2 (previous value) + d. now it is normalize because expo2 and expo1 are equal.

5:- Check if in1 and in2 have same sign by checking 32th bit of in1 and in2 which is sign bit.

6:- In this step add 24 bit significant (1 hidden bit and 23 mantissa) S1 and S2, i.e. S=S1+S2.

7:- Check for the carry whether carry is generated by significant addition; if it is generated then add '1' in either expo1 or updated value of expo2. After that shift right addition of significant 'S' by 1 bit drop the LSB and make new MSB'1'.

8:- If no carry is generated in addition then skip step 7 and previous exponent is final exponent.

9:- The sign of result is either of in1 or in2 as it is of same sign.

10:- pack the result in 32 bit format as follow: =sing_f (1) exponent (8) S (23).

In Case II- When both numbers are of different sign

Steps1:- Enter the two numbers in1 and in2 each of 32 bits. expo1, S1 and expo2, S2 indicates exponent and significant of in1 and in2 respectively.

2:- Check if expo1 or expo2 is zero, if it is zero then set hidden bit of respective input number '0', if expo1 and expo2is not '0' check if expo2>expo1 swap in1 and in2 else keep them as it is.

3:- Now calculate the difference between expo1 and expo2. Let's say this difference is d, d= expo1-expo2. If difference'd' is zero then there is no change else left shift significant of in2 which is S2 by the amount 'd' along with the hiddenbit which us '1'.

4:- Normalize the exponent of in2 by adding the amount d, expo2=expo2 (previous value) + d. now it is normalize because expo2 and expo1 are equal.

5:- Check if in2 and in1 have different sign, if yes;

6:- Add 2's compliment of S2 i.e. not (S2) +'1' with significant of in1 'S1'.

7:-If carry is generated by addition of significant then discard carry and left shift the result till there is '1' in MSB and count the number of shifting denoted by 'z'.

8:- Final exponent is calculated by subtracting 'z' from either expo1 or expo2

(expo_f = expo1-'z') and append 'z'number of 0's in LSB part of significant.

9:- If there is no carry generated in step number 6 converts into 2's compliment form and in this case MSB should be'1';.

10:- Sign of the result i.e. MSB = Sign of the larger number either MSB of N1 or it can be MSB of N2.

11:- pack the result in 32 bit format by excluding hidden bit as follow:-

sing_f (1bit) exponent (8bits) S(23bits).

4.2. Floating point Multiplication.

The figure 4 shows the flowchart of multiplication algorithm of multiplication is demonstrated by flowchart. In1 and in2 are two numbers sign1, expo1, S1 and sign2, expo2, S2 are sign bit, exponent and significant of in1 and in2 respectively.

Steps for multiplication are as follows.

1:- calculate sign bit. sign_f= sign1 XOR sign2, sign_f is sign of final result.

2:- add the exponents and subtract 127 to make adjustment in exponent (expo1+127+expo2+127)-127.

3:- Multiply the significant. S=S1*S2.

4:- check for overflow and underflow and special flag. When the value of biased exponent is less than 1 it shows occurrence of underflow, if exponent is greater than 254 then it shows overflow of floating point operation.

5:- Take the first 23 bits of 'S' and from left side and discard remaining bits.

6:- Arrange the results in 32 bit format. 1 sign bit followed by eight bits exponent followed by 23 bits mantissa.

4.3. Floating point division.

The figure 5 shows the flowchart of division, algorithm for division is explained through this flowchart. In1 and in2 are two numbers sign1, expo1, S1 and sign2, expo2, S2 are sign bit, exponent and significant of in1 and in2 respectively. It is assumed that in1 and in2 are in normalized form.

Steps for floating point division are as follows.

1:- calculate sign bit. sign_f= sign1 XOR sign2, sign_f is sign of final result.

2:- divide the significant S1 by S2 for division binary division method is used.

• Expand divisor and dividend to double of their size

- Expanded divisor = divisor (24 bits MSB) zeroes (24 bits LSB)

- Expanded dividend = zeroes (24 bits MSB) dividend (32 bits, LSB)

• For each step, determine if divisor is smaller than dividend

- Subtract divisor from dividend look at sign

- If result is greater than or equal to '0': dividend/divisor>=1, mark quotient as "1".

- If result is negative: divisor larger than dividend; make in quotient as "0"

• Shift quotient left and divisor right to cover next power of two.

3:- Subtract the expo2 from expo1.

4:- check for overflow and underflow flags

5. FLOWCHART FOR ALGORITHM

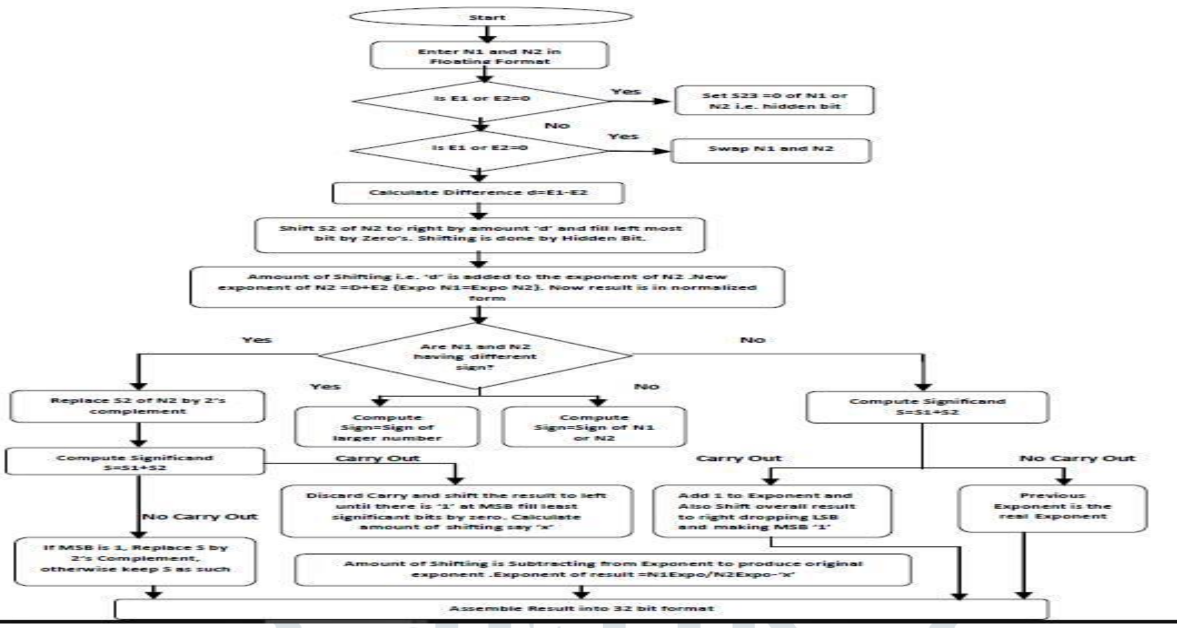

Start
Enter N1 and N2 in Floating Format
Is E1 or E2=0
Yes
Set S23 =0 of N1 or N2 i.e. hidden bit
No
Is E1 or E2=0
Yes
Swap N1 and N2
Calculate Difference d=E1-E2
Shift S2 of N2 to right by amount 'd' and fill left most bit by Zero's. Shifting is done by Hidden Bit.
Amount of Shifting i.e. 'd' is added to the exponent of N2 .New exponent of N2 =D+E2 {Expo N1=Expo N2}. Now result is in normalized form
Are N1 and N2 having different sign?
Yes
No
Replace S2 of N2 by 2's complement
Yes
No
Compute Sign=Sign of larger number
Compute Sign=Sign of N1 or N2
Compute Significand S=S1+S2
Compute Significand S=S1+S2
Carry Out
Carry Out
No Carry Out
Discard Carry and shift the result to left until there is '1' at MSB fill least significant bits by zero. Calculate amount of shifting say 'x'
Add 1 to Exponent and Also Shift overall result to right dropping LSB and making MSB '1'
Previous Exponent is the real Exponent
No Carry Out
If MSB is 1, Replace S by 2's Complement, otherwise keep S as such
Amount of Shifting is Subtracting from Exponent to produce original exponent .Exponent of result =N1Expo/N2Expo-'x'
Assemble Result into 32 bit format

Figure 3. Flowchart for ADDITION/SUBTRACTION

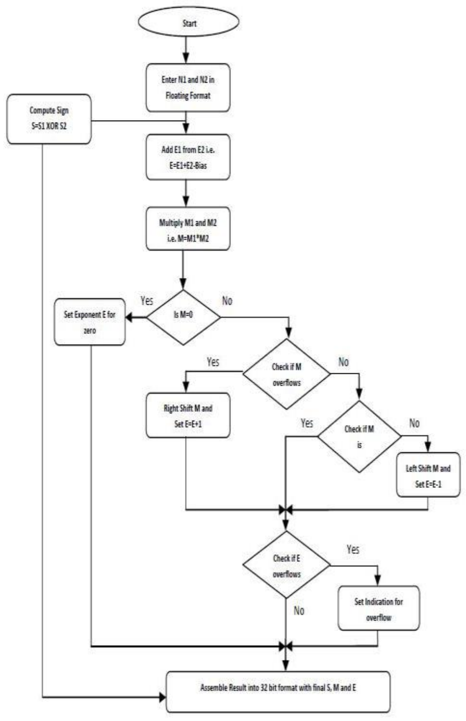

Start
Enter N1 and N2 in Floating Format
Compute Sign S=S1 XOR S2
Add E1 from E2 i.e. E=E1+E2-Bias
Multiply M1 and M2 i.e. M=M1*M2
Yes
No
Set Exponent E for zero
Is M=0
Yes
Check if M overflows
No
Right Shift M and Set E=E+1
Yes
Check if M is
No
Left Shift M and Set E=E-1
Check if E overflows
Yes
Set Indication for overflow
No
Assemble Result into 32 bit format with final S, M and E

Figure 4. Flowchart for MULTIPLICATION

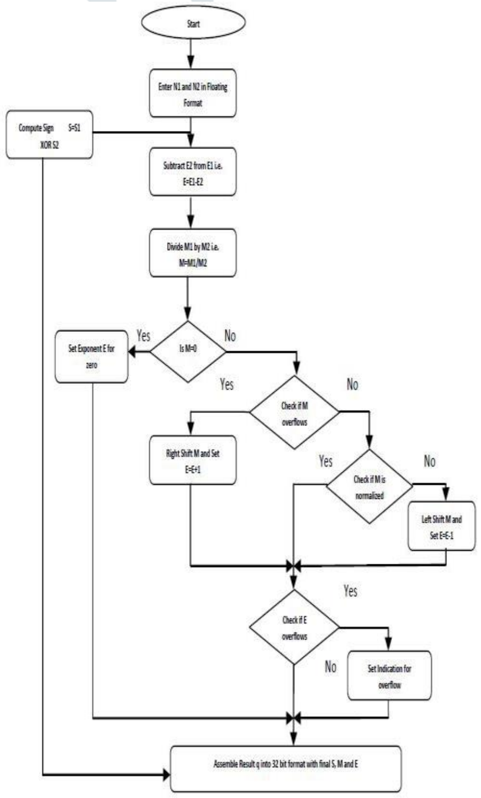
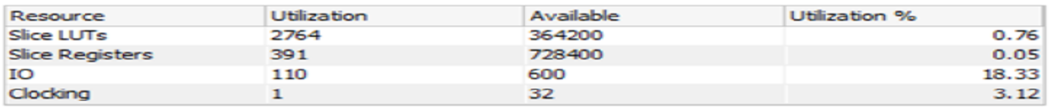

Start
Enter N1 and N2 in Floating Format
Compute Sign S=S1 XOR S2
Subtract E2 from E1 i.e. E=E1-E2
Divide M1 by M2 i.e. M=M1/M2
Yes
No
Set Exponent E for zero
Is M=0
Yes
No
Check if M overflows
Right Shift M and Set E=E+1
Yes
No
Check if M is normalized
Left Shift M and Set E=E-1
Yes
Check if E overflows
No
Set Indication for overflow
Assemble Result q into 32 bit format with final S, M and E

Figure 5. Flowchart for DIVISION

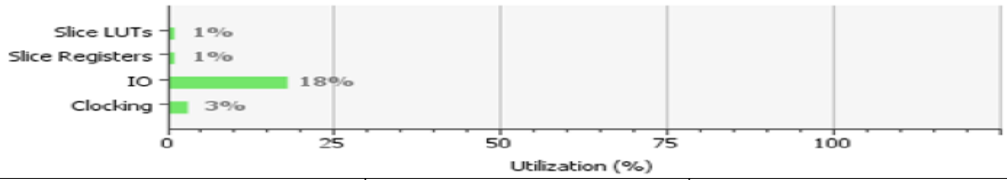
6. VERIFICATION OF RESULT

The floating point arithmetic unit is coded in VHDL, synthesized and simulated on Xilinx Vivado 2014.4 in virtex-7 simulator. The maximum combinational path delay of floating point multiplier is 25.494 ns .the various results of arithmetic operation and utilization report as shown below.

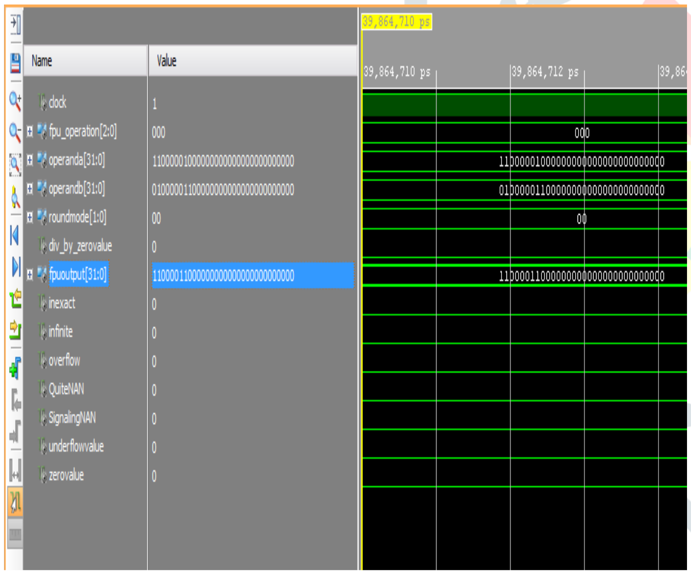
7. EXPERIMENTAL RESULTS



Resource	Utilization	Available	Utilization %
Slice LUTs	2764	364200	0.76
Slice Registers	391	728400	0.05
IO	110	600	18.33
Clocking	1	32	3.12

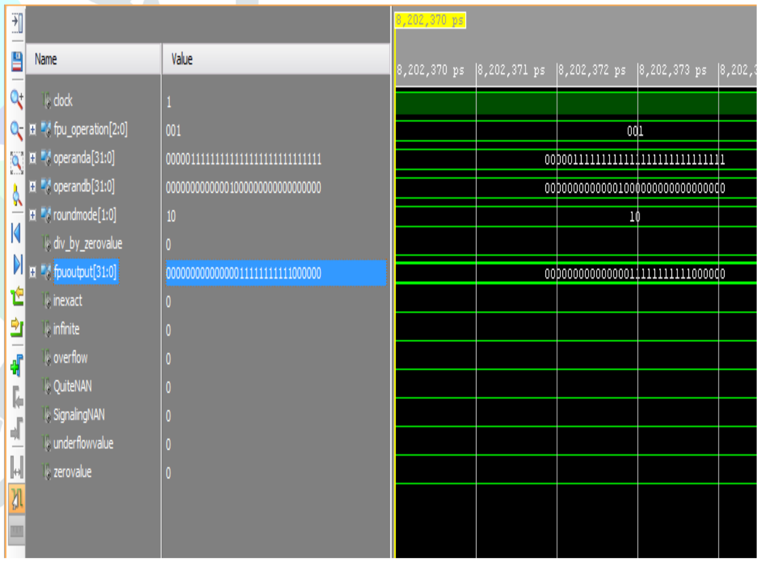


Device	Delay	Fmax
Virtex-7 xc7v585tffg1157-3	25.494ns	100Mhz



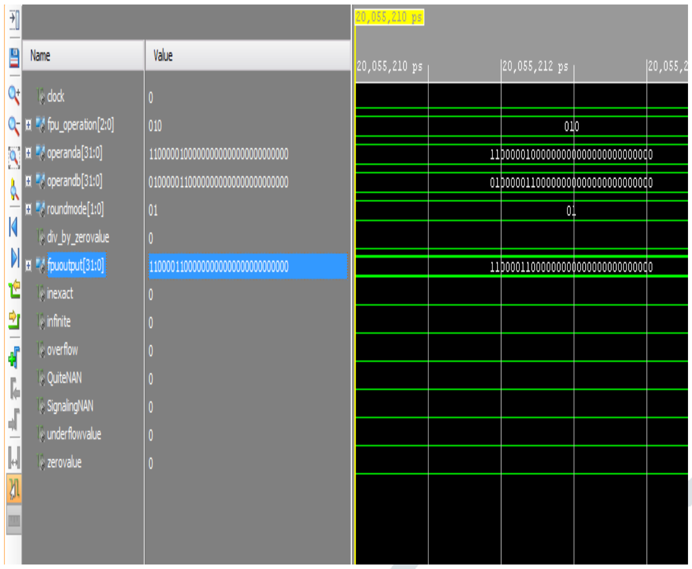
```
Name                  Value
clock                 1
fpu_operation[2:0]    000
operanda[31:0]        11000001000000000000000000000000
operandb[31:0]        01000001100000000000000000000000
roundmode[1:0]        00
div_by_zerovalue      0
fpuoutput[31:0]       11000011000000000000000000000000
inexact               0
infinite              0
overflow              0
QuiteNAN              0
SignalingNAN          0
underflowvalue        0
zerovalue             0
```

Simulation result of addition of two 32 bit floating point number. The round mode is 00 and the operation mode is 000



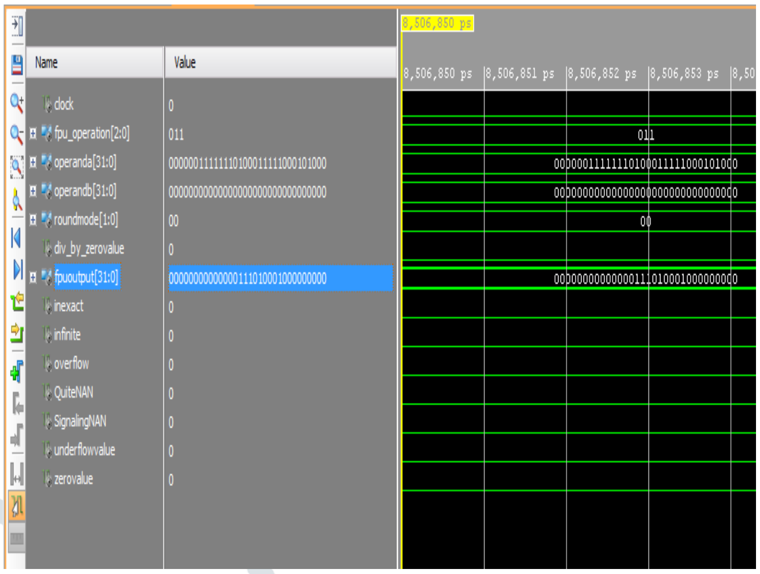
```
Name                  Value
clock                 1
fpu_operation[2:0]    001
operanda[31:0]        00000111111111111111111111111111
operandb[31:0]        00000000000001000000000000000000
roundmode[1:0]        10
div_by_zerovalue      0
fpuoutput[31:0]       00000000000000011111111111000000
inexact               0
infinite              0
overflow              0
QuiteNAN              0
SignalingNAN          0
underflowvalue        0
zerovalue             0
```

Simulation result of subtraction of two 32 bit floating point number. The round mode is 10 and the operation mode is 001



Name	Value
clock	0
fpu_operation[2:0]	010
operanda[31:0]	11000001000000000000000000000000
operandb[31:0]	01000001100000000000000000000000
roundmode[1:0]	01
div_by_zerovalue	0
fpuoutput[31:0]	11000011000000000000000000000000
inexact	0
infinite	0
overflow	0
QuiteNAN	0
SignalingNAN	0
underflowvalue	0
zerovalue	0

Simulation result of multiplication of two 32 bit floating point number. The round mode is 01 and the operation mode is 010



Name	Value
clock	0
fpu_operation[2:0]	011
operanda[31:0]	00000011111110100011111000101000
operandb[31:0]	00000000000000000000000000000000
roundmode[1:0]	00
div_by_zerovalue	0
fpuoutput[31:0]	00000000000000111010001000000000
inexact	0
infinite	0
overflow	0
QuiteNAN	0
SignalingNAN	0
underflowvalue	0
zerovalue	0

Simulation result of division of two 32 bit floating point number. The round mode is 00 and the operation mode is 011

8. CONCLUSION

This FPU has action as addition, subtraction multiplication, and division. Addition, subtraction and division have been implemented by conventional technique. The design of FPU has done using VHDL on Xilinx VIVADO 2014.4 and has implemented on Virtex-7 xc7v585tffg1157-3.We have proved that this FPU have requisite less memory but still we have a huge amount of work that can be set on this FPU to further make up the efficiency of the FPU by using other Vedic sutra.

9. REFERENCES

[1] IEEE 754 Standard for binary floating-point arithmetic,1985.

[2] “IEEE Standard for Binary Floating-point Arithmetic", ANSUIEEE Std 754-1985,
The Institute of Electrical and Electronics Engineers, August 12, 1985,New York.

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

[4] Michael L.Overton, “Numerical Computing with IEEE 754standard Floating Point Arithmetic,” Published by Society for Industrial and Applied Mathematics,2001.

[5] DhirajSangwan& Mahesh K. Yadav, “Design and Implementation ofFloating Point Arithmeticunit(Adder/Subtractor and multiplication)”, in International Journal of Electronics Engineering, 2(1), 2010, pp. 197-203.

[6] Prof J M Rudagi, VishwanathAmbli, VishwanathMunavali, RavindraPatil, VinaykumarSajjan "Design and implementation of Efficient multiplier using Vedic mathematics", on advances in recent technologies in Communication and Computing 2011

[7]Vikas Gupta, Anshuj Jain, BhartiChourasia," FPGA Based Implementation of Genetic Algorithm Using VHDL," International Journal ofElectronics Communication and Computer Technology (IJECCT) Volume 1 Issue 1 | September 2011.

[8]Sunita.s.malaj, S.B.Patil, Bhagappa.R.Umarane, "VHDL implementation of interval arithmetic algorithms for single precision floating point numbers" International Journal of Scientific & Engineering Research(IJSER) Volume 4, Issue3, March- 2013.

[9]Wikipedia "FPGA Implementation of single precisions floating point(32 bit)", IEEE(Institute of Electrical and Electronics Engineers) 754 standard.