

Design Of High Performance IEEE- 754 Single Precision (32 bit) Floating Point Adder Using VHDL

Preethi Sudha Gollamudi, M. Kamaraju

Dept. of Electronics & Communications Engineering

Gudlavalleru Engineering College, Andhra Pradesh, India-769008

Abstract

Floating Point arithmetic is by far the most used way of approximating real number arithmetic for performing numerical calculations on modern computers. The advantage of floating-point representation over fixed-point and integer representation is that it can support a much wider range of values. Addition/subtraction, Multiplication and division are the common arithmetic operations in these computations. Among them Floating point Addition/Subtraction is the most complex one. This paper implements an efficient 32bit floating point adder according to IEEE 754 standard with optimal chip area and high performance using VHDL. The proposed architecture is implemented on Xilinx ISE Simulator. Results of proposed architecture are compared with the existed architecture and have observed reduction in area and delay. Further, this project can be extendable by using any other type of faster adder in terms of area, speed and power.

Keywords: Floating point, Adder, FPGA, IEEE-754, VHDL, Single precision.

1. Introduction

Many fields of science, engineering and finance require manipulating real numbers efficiently. Since the first computers appeared, many different ways of approximating real numbers on it have been introduced. One of them, the floating point arithmetic, is clearly the most efficient way of representing real numbers in computers. Representing an infinite, continuous set (real numbers) with a finite set (machine numbers) is not an easy task: some compromises must be found between speed, accuracy, ease of

use and implementation and memory cost. Floating Point Arithmetic represent a very good compromise for most numerical applications. Floating point operations are hard to implement on reconfigurable hardware i.e. on FPGAs because of their complexity of their algorithms. On the other hand, many scientific problems require floating point arithmetic with high level of accuracy in their calculations. Therefore VHDL programming for IEEE single precision floating point adder in have been explored.

For implementation of floating point adder on FPGAs module various parameters i.e. clock period, latency, area (number of slices used), total number of paths/destination ports, combinational delay, modeling formats etc will be outline in the synthesis report. VHDL code for floating point adder is written in Xilinx 8.1i and its synthesis report is shown in Design process of Xilinx which will outline various parameters like number of slices used, number of slice flipflop used, number of 4 input LUTs, number of bonded IOBs, number of global CLKs. Floating point addition is most widely used operation in DSP/Math processors, Robots, Air traffic controller, Digital computers because of its raising application the main emphasis is on the implementation of floating point adder effectively such that it uses less chip area with more clock speed.

2. Floating point Number format

Floating point numbers are one possible way of representing real numbers in binary format; the IEEE 754 standard presents two different floating point formats, Binary interchange format and Decimal interchange format. 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 significand

then the number is said to be a normalized number; in this case the real number is represented by (1)

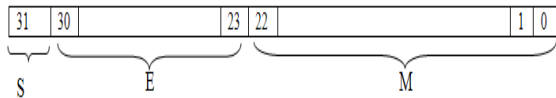


Figure 1. IEEE single precision floating point format

$$Z = (-1)^S * 2^{(E-bias)} * (1.M) \quad (1)$$

Where $M = m_{22} 2^{-1} + m_{21} 2^{-2} + m_{20} 2^{-3} + \dots + m_1 2^{-22} + m_0 2^{-23}$;

Bias=127

2.1 IEEE 754 Standard for Binary Floating-Point Arithmetic:

The IEEE 754 Standard for Floating-Point Arithmetic is the most widely-used standard for floating-point computation, and is followed by many hardware (CPU and FPU) and software implementations. The standard specifies:

- Basic and extended floating-point number formats
- Add, subtract, multiply, divide, square root, remainder, and compare operations
- Conversions between integer and floating-point formats
- Conversions between different floating-point formats

The different format parameters for single and double precision are shown in table 1.

Parameter	binary32	binary64
k , storage width in bits	32	64
p , precision in bits	24	53
$emax$, maximum exponent e	127	1023
<i>Encoding parameters</i>		
$bias$, $E - e$	127	1023
sign bit	1	1
w , exponent field width in bits	8	11
t , trailing significand field width in bits	23	52
k , storage width in bits	32	64

Table 1. Binary interchange format parameters

2.2 Conversion of decimal to floating point numbers

Conversion of Decimal to Floating point 32 bit format is explained with example. Let us take an example of a decimal number that how could it will be converted into floating format. Enter a decimal number suppose 129.85 before converting into floating format this number is converted into binary value which is 10000001.110111. After conversion move the radix point to the left such that there will be only one bit which is left of the radix point and this bit must be 1 this bit is known as hidden bit and also made above number of 24 bit including hidden bit which is like 1.00000011101110000000000 the number which is after the radix point is called mantissa which is of 23 bits and the whole number is called significand which is of 24 bits. Count the number of times the radix point is shifted say „x“. But in above case there is 7 times shifting of radix point to the left. This value must be added to 127 to get the exponent value i.e. original exponent value is $127 + „x“$. In above case exponent is $127 + 7 = 134$ which is 10000110. Sign bit i.e. MSB is „0“ because number is +ve. Now assemble result into 32 bit format which is sign, exponent, mantissa.

4. Existing Method

Floating point addition algorithm can be explained in two cases with example. Case I is when both the numbers are of same sign i.e. when both the numbers are either +ve or -ve means the MSB of both the numbers are either 1 or 0. Case II when both the numbers are of different sign i.e. when one number is +ve and other number is -ve means the MSB of one number is 1 and other is 0.

4.1 Case I: -When both numbers are of same sign

Step 1:- Enter two numbers N1 and N2. E1, S1 and E1, S2 represent exponent and significand of N1 and N2.

Step 2:- Is E1 or E2 =0“. If yes set hidden bit of N1 or N2 is zero. If not then check is $E2 > E1$ if yes swap N1 and N2 now contents of N2 in N1 and N1 in N2 and if $E1 > E2$ make contents of N1 and N2 same there is no need to swap.

Step 3:- Calculate difference in exponents $d = E1 - E2$. If $d = „0“$ then there is no need of shifting the significand and if d is more than „0“ say „y“ then

shift S2 to the right by an amount „y“ and fill the left most bits by zero. Shifting is done with hidden bit.

Step 4:- Amount of shifting i.e. „y“ is added to exponent of N2 value. New exponent value of E2= previous E2 + „y“. Now result is in normalize form because E1 = E2.

Step 5:- Is N1 and N2 have different sign „no“. In this case N1 and N2 have same sign.

Step 6:- Add the significands of 24 bits each including hidden bit $S=S1+S2$.

Step 7:- Is there is carry out in significand addition. If yes then add „1“ to the exponent value of either E1 or new E2 and shift the overall result of significand addition to the right by one by making MSB of S is „1“ and dropping LSB of significand.

Step 8:- If there is no carry out in step 6 then previous exponent is the real exponent.

Step 9:- Sign of the result i.e. MSB = MSB of either N1 or N2. Step 10:- Assemble result into 32 bit format excluding 24th bit of significand i.e. hidden bit .

4.2 Case II: - When both numbers are of different sign

Step 1, 2, 3 & 4 are same as done in case I.

Step 5:- Is N1 and N2 have different sign „Yes“.

Step 6:- Take 2's complement of S2 and then add it to S1 i.e. $S=S1+2$'s complement of S2.

Step 7:- Is there is carry out in significand addition. If yes then discard the carry and also shift the result to left until there is „1“ in MSB also counts the amount of shifting say „z“.

Step 8:- Subtract „z“ from exponent value either from E1 or E2. Now the original exponent is E1- „z“. Also append the „z“ amount of zeros at LSB.

Step 9:- If there is no carry out in step 6 then MSB must be „1“ and in this case simply replace „S“ by 2's complement.

Step 10:- Sign of the result i.e. MSB = Sign of the larger number either MSB of N1 or it can be MSB of N2. Step 11:- Assemble result into 32 bit format excluding 24th bit of significand i.e. hidden bit .

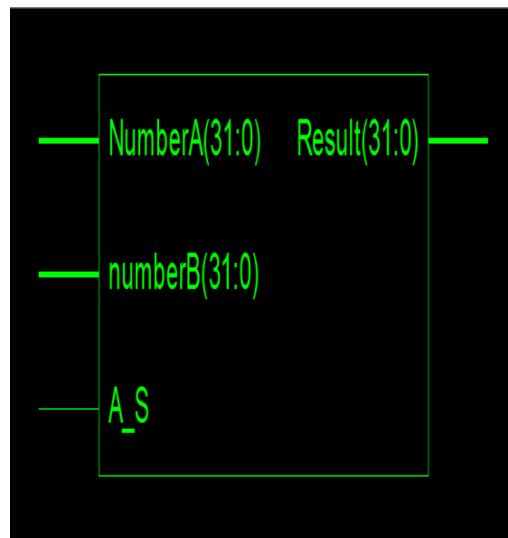
4.2 Problems Associated with floating point Addition

1. Some possible combinations of A and B have a direct result where there is no need of adder or some other resources, but in the existing work adder is used for direct result. Hence it is a time taking process.

2. IEEE 754 does not say anything about the operations between subnormal and normal numbers i.e. Mixed Numbers (one operand is normal and the other is subnormal).

5. Proposed Method

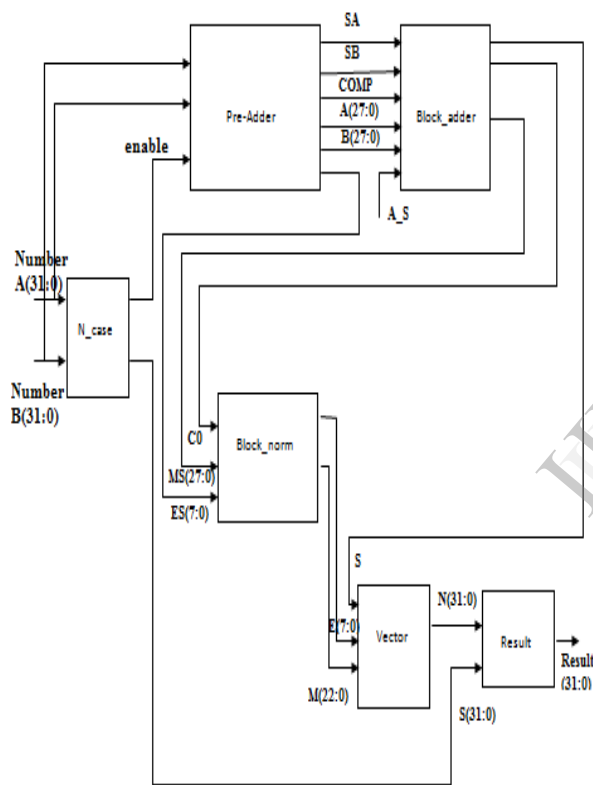
The black box view and block diagram of the single precision floating point Adder is shown in figures 2 and 3 respectively. The input operands are separated into their sign, mantissa and exponent components. This module has inputs opa and opb of 32-bit width. Before the operation Special cases are treated separately without Adder and some other resources. In addition to normal and subnormal numbers, infinity, NaN and zero are represented in IEEE 754 standard. Some possible combinations have a direct result, for example, if a zero and a normal number are introduced the output will be the normal number directly. Time and resources are saved implementing this block. The *n_case* block has been designed to run this behaviour.



“Figure 2. Black box view of single precision Adder”

The Block diagram has two branches:

- The special cases one is quiet simple because only the combination of the different exceptions are taken into account.
- The second one is more interesting. It includes the main operation of the adder. The different operations that should be done are divided in three big blocks: pre-adder, adder and normalizing block.



“Figure3.Block Diagram of Adder”

The inputs to the n_case block are A and B and outputs are vector S and enable. Vector S contains the result when there is a special case as shown in table[2] otherwise undefined. Finally, enable signal enables or disables the adder block if it is needed or not. If any normal or subnormal combination Occurs the enable signal is high, otherwise low.

Sign	outA	outB	Sign output	output
X	Zero	Number B	SB	Number B
X	Number A	Zero	SA	Number A
X	Normal/Subnormal	Infinity	SB	Infinity
X	Infinity	Normal/Subnormal	SA	Infinity
SA=SB	Infinity	Infinity	SX	Infinity
SA≠SB	Infinity	Infinity	1	NaN
X	NaN	Number B	1	NaN
X	Number A	NaN	1	NaN

“Table 2.Special cases output coded”

5.1 Pre-Adder block

The first subblock is the Pre-Adder block. Various functions of the block are:

1. Distinguishing between normal, subnormal or mixed (normal-subnormal combination) numbers.
2. Treating the numbers in order to be added (or subtracted) in the adder block.

- Setting the Output's exponent
- Shifting the mantissa
- Standardizing the subnormal number in mixed numbers case to be treated as a normal case.

5.1.1 Normal numbers case

The procedure is as follows:

1. Making a comparison between both A and B numbers and obtaining the largest number
2. Obtaining the output exponent (the largest one)
3. Shifting the smallest mantissa to equal both exponents.

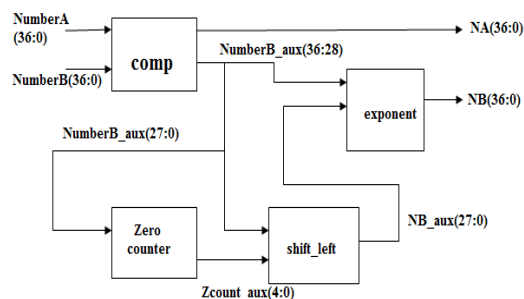
5.1.2 Subnormal numbers case

The operation using subnormal numbers is the easiest one. It is designed in just one block and the procedure is as follows:

1. Obtaining the two sign bits and both mantissas.
2. Making a comparison between both A and B numbers in order to acquire the largest number.
3. Fixing the result exponent in zero.

5.1.3 Mixed Numbers case

When there is a mixed combination of numbers (one subnormal and other normal) the subnormal one must have a special treatment in order to be added or subtracted to the normal one. The architecture of mixed numbers case is shown in figure. The comp block identifies the subnormal number and fixes it in NumberB_aux. Then the number of zeros on the beginning of the subnormal number are counted for shifting the vector and new exponent is calculated.



“Figure 4. Mixed numbers case architecture”

5.2 Adder Block

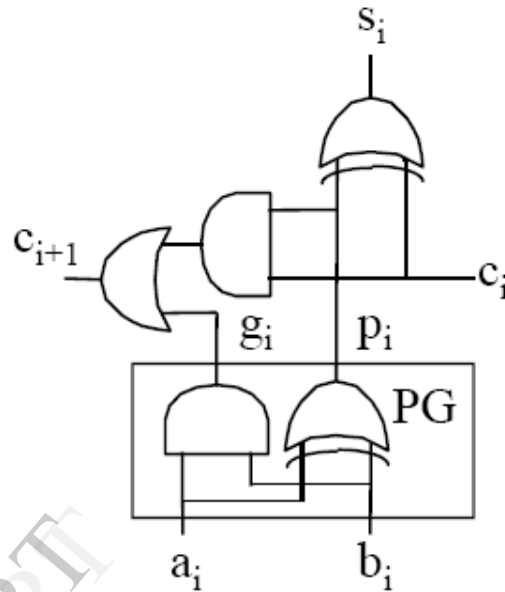
Adder is the easiest part of the blocks. This block only implements the operation (addition or subtraction). It can be said the adder block is the ALU (*Arithmetic Logic Unit*) of the project because it is in charge of the arithmetic operations.

Two functions are implemented in this part of the code:

1. Obtaining the output's sign
2. Implementing the desired operation

A Carry Look Ahead structure has been implemented. This structure allows a faster addition than other structures. It improves by reducing the time required to determine carry bits.

This is achieved by calculating the carry bits before the sum which reduces the wait time to calculate the result of the large value bits. The code has been designed implementing a 1-bit CLA structure and generating the other components up to 28 (the number of bits of the adder) by the function generate.



“Figure 5. Carry look ahead adder structure”

5.3 Standardizing Block

Finally the Standardizing Block takes the result of the addition/subtraction and gives it an IEEE 754 format. The procedure is as follows:

1. Shifting the mantissa to standardize the result
2. Calculating the new exponent according with the addition/subtraction overflow (carry out bit) and the displacement of the mantissa.

5.3.1 Round block

Round block provides more accuracy to the design. Four bits at the end of the vector had been added in the Pre-Adder block. Now it is time to use these bits in order to round the result. The process to round is chosen arbitrarily: if the round bits are greater than the value “1000” the value of the mantissa will be incremented by one. Otherwise the value keeps the same value.

Apart from that a vector block is used to group the final sign, exponent and mantissa of the result. The final Result is Selected between the Special case output and operated output based on inputs.

5.4 Algorithm for single precision floating point Adder

1. Extracting signs, exponents and mantissas of both A and B numbers.

2. Treating the special cases:

- Operations with A or B equal to zero
- Operations with ∞
- Operations with NaN

3. Finding out what type of numbers are given:

- Normal
- Subnormal
- Mixed

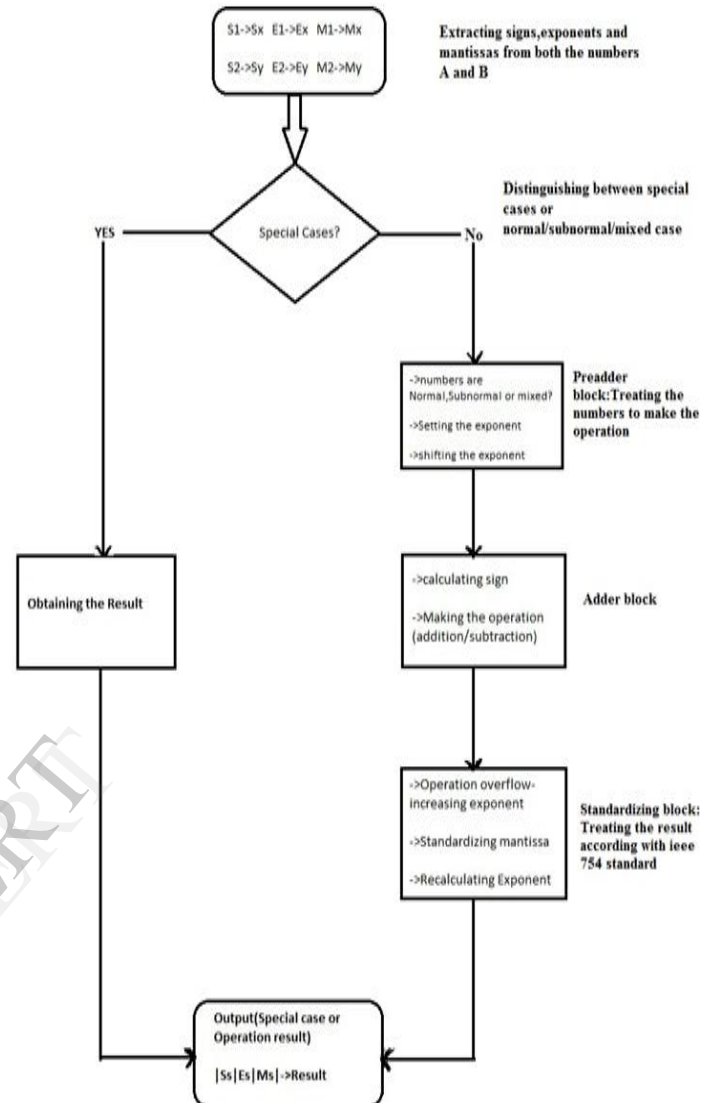
4. Shifting the lower exponent number mantissa to the right [$Exp1 - Exp2$] bits. Setting the output exponent as the highest exponent.

5. Working with the operation symbol and both signs to calculate the output sign and determine the operation to do.

6. Addition/Subtraction of the numbers and detection of mantissa overflow (carry bit).

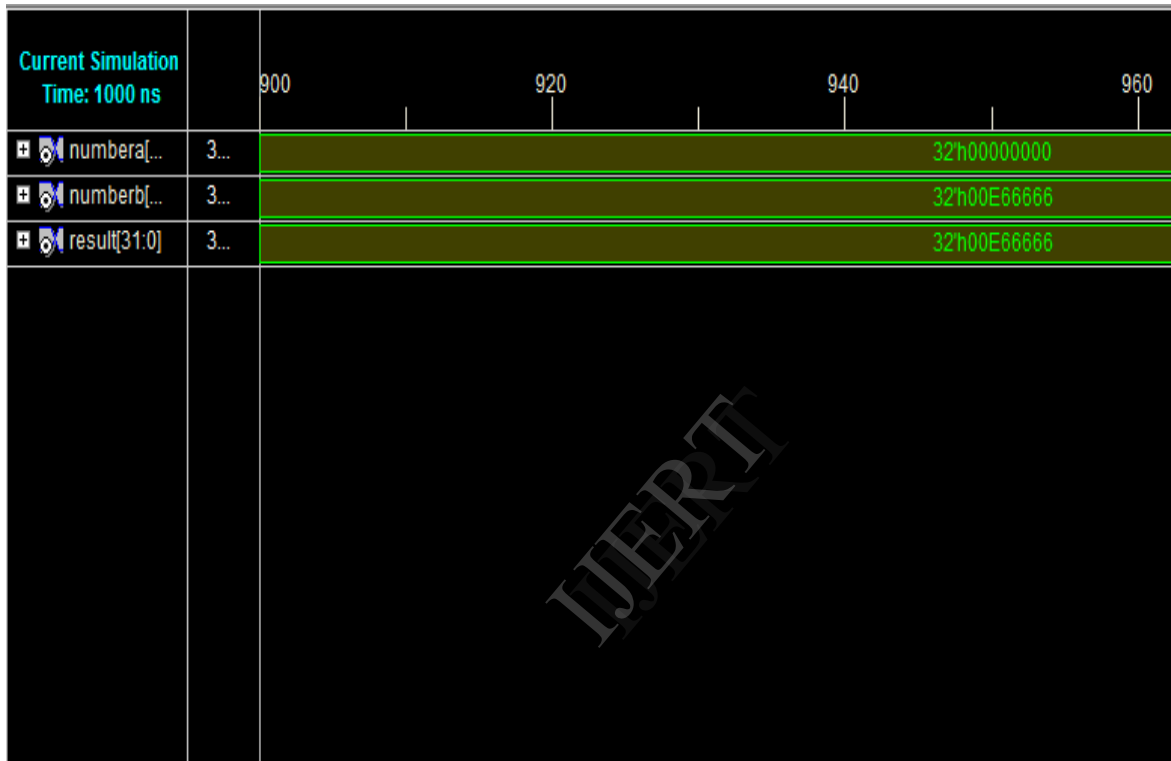
7. Standardizing mantissa shifting it to the left up the first one will be at the first position and updating the value of the exponent according with the carry bit and the shifting over the mantissa.

8. Detecting exponent overflow or underflow (result NaN or ∞)

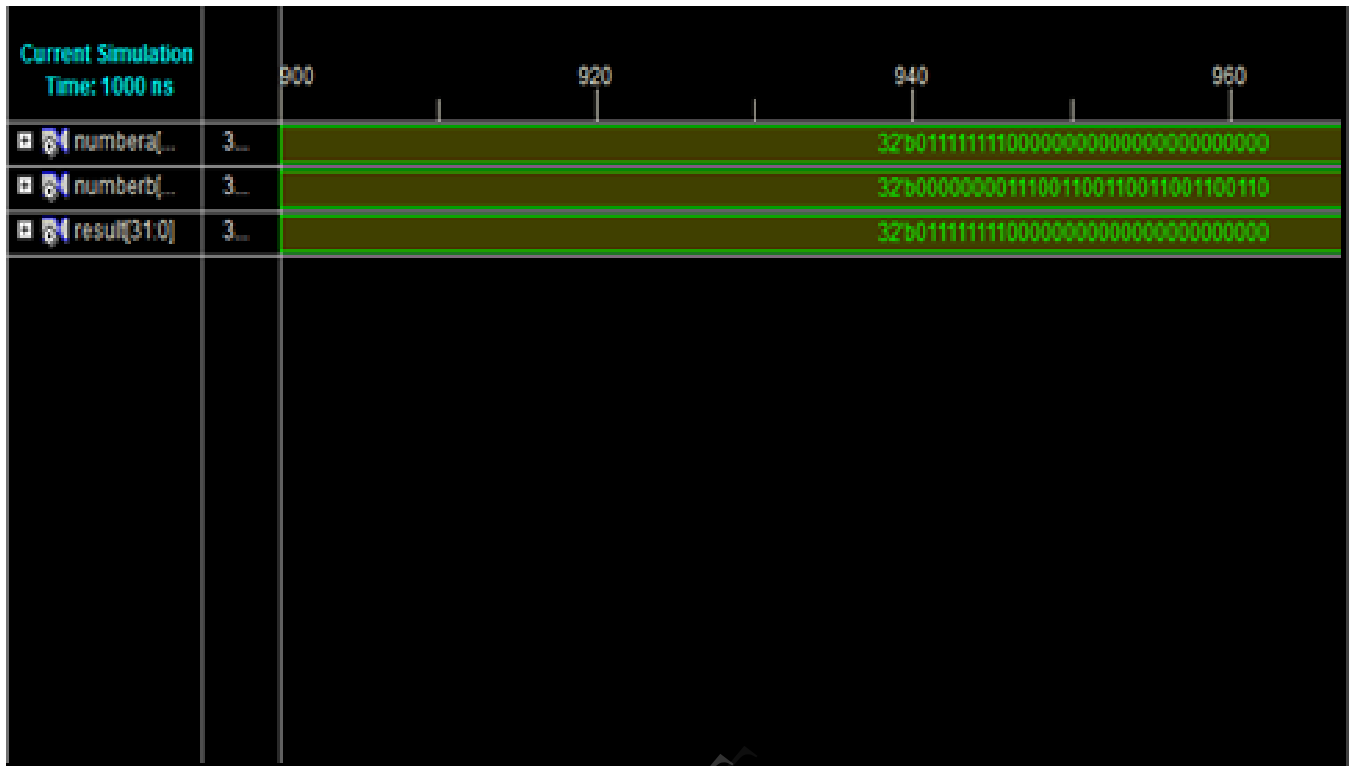


“Figure 6. Flow chart of single precision floating point addition”

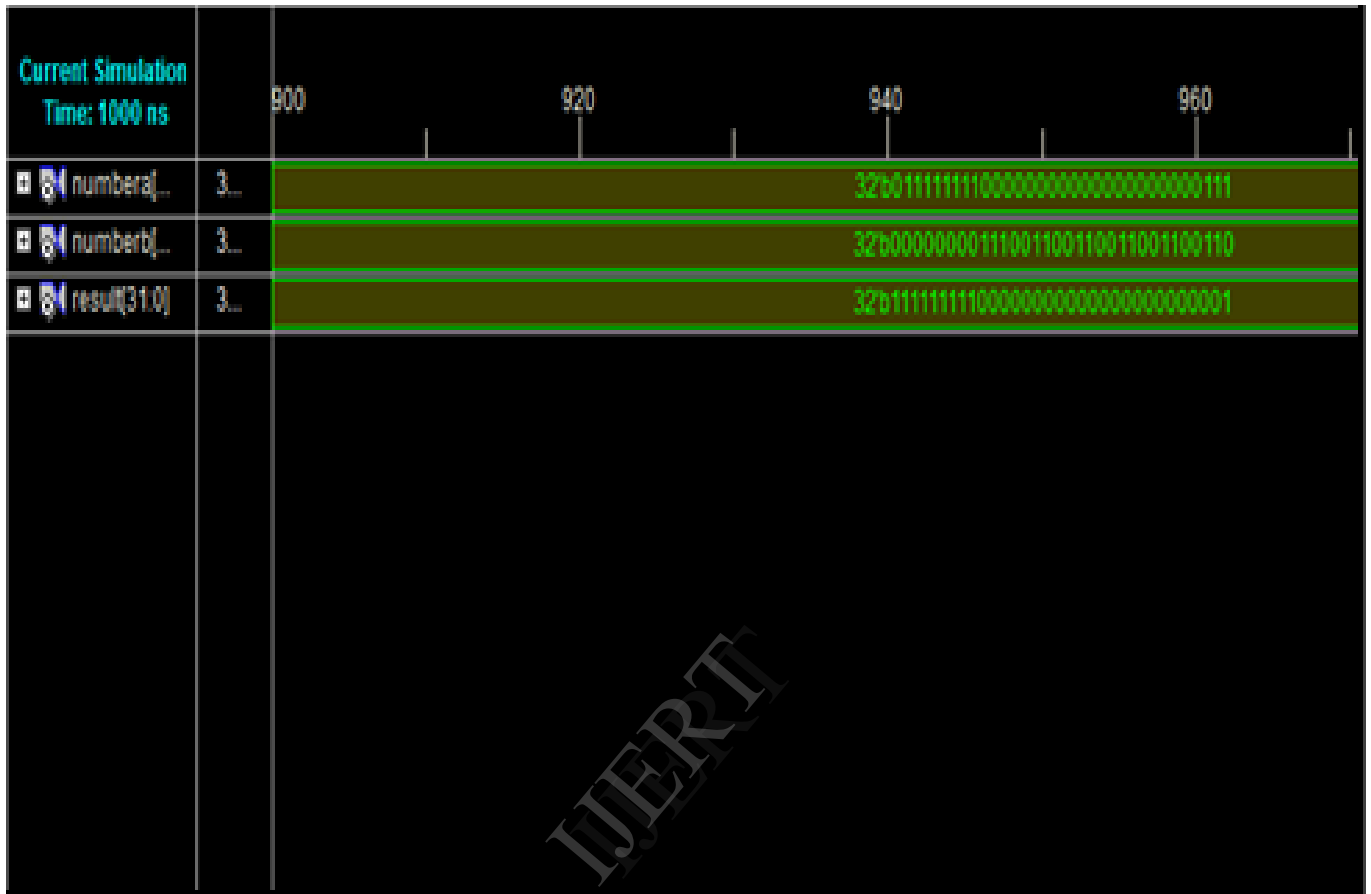
5.5 Simulation Results



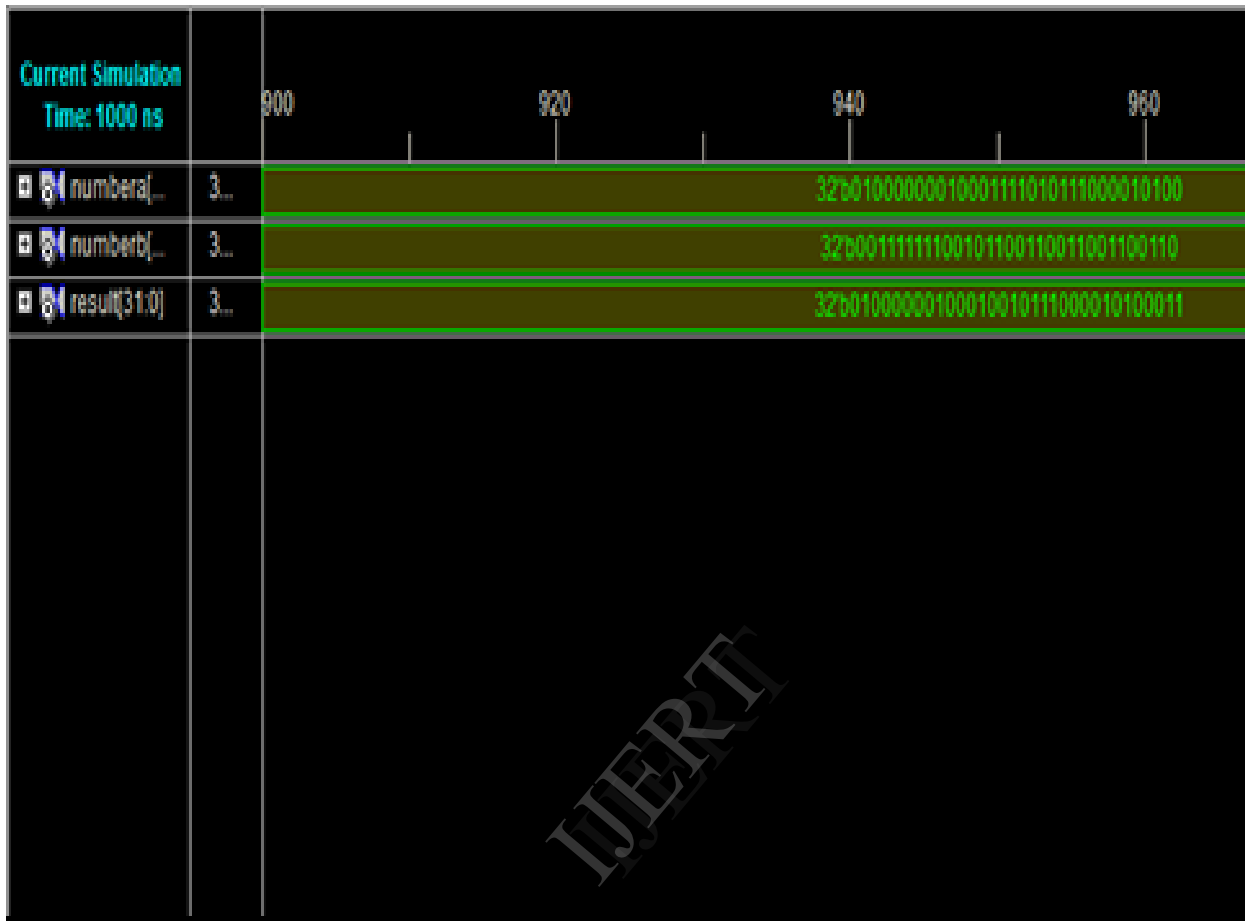
“Figure 7.Simulation result for special case-zero”



“Figure 8.Simulation Report for special cases-infinity”

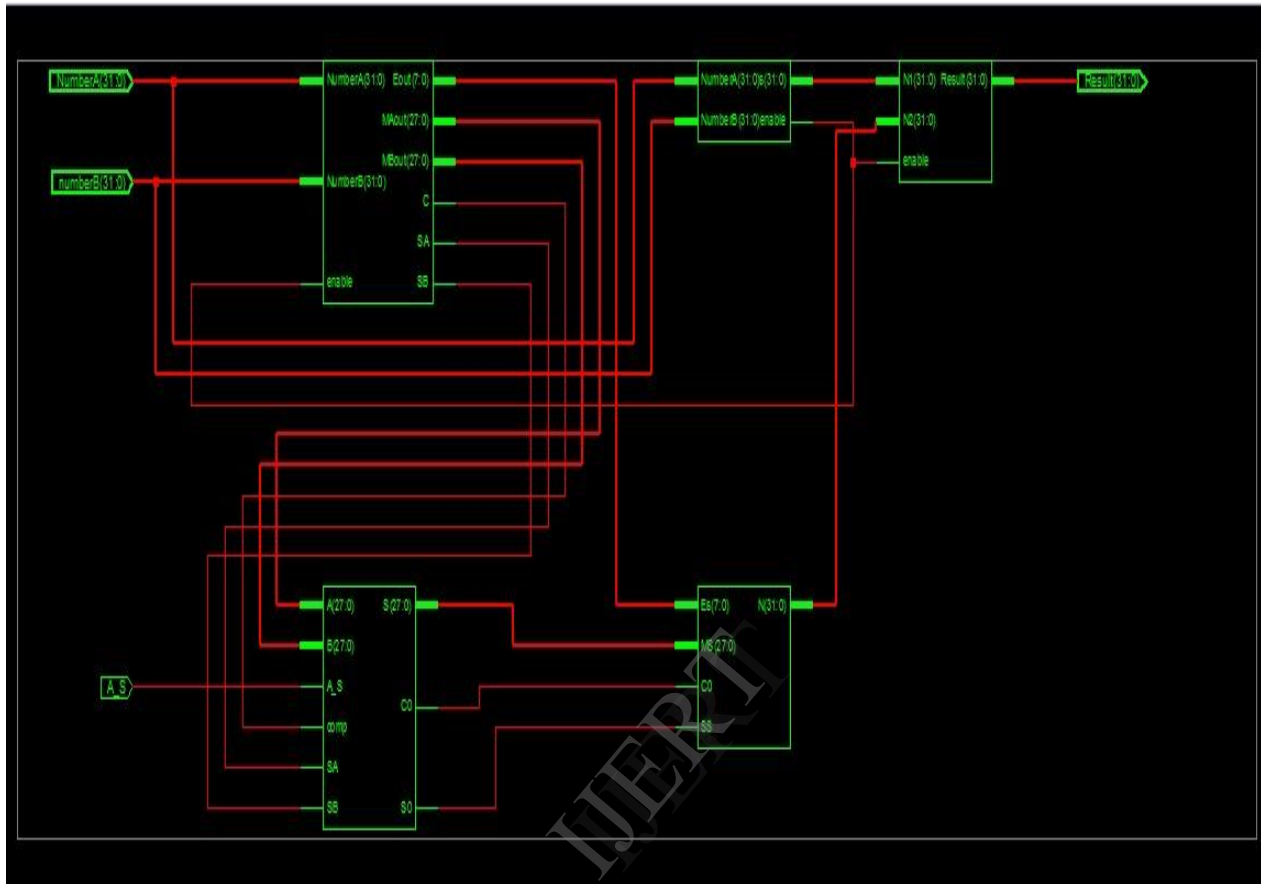


“Figure 9.Simulation result for special case-NaN”



“Figure 10.Simulation report for normal numbers case”

5.6 RTL Schematic



“Figure 11.RTL schematic of floating point adder”

In this work VHDL code is written for single precision floating point adder and implemented in Xilinx ISE simulator.

Table 3:Device utilization summary (6vlx75tff484-3) of single precision Floating point adder on Virtex 4 (xc4cfx12-12sf363) Speed Grade = -12

Logic Utilization	Existing Method	Proposed Method	Available
Number of Slices	401	281	5472
Number of Slice Flipflops	72	32	10944
Number of 4-input LUT's	710	504	10944
Number of bonded IOB's	99	97	240
Number of Global clk's	2	1	32

Table1 shows the device utilization for both existed method and proposed method . It is observed that the delay and chip area are reduced in the proposed method.

5.7 Combinational delay analysis

- Maximum combinational path delay for existing one:24.201ns
- Maximum combinational path delay for proposed one:16.988ns

6.Conclusion

The design of 32bit floating point adder has been implemented on virtex 4 FPGA .The entire code is written in VHDL programming language and

synthesized using xilinx 9.1.The proposed work consumes less chip area when compared to the existing one and has less combinationa delay.

7.References

- [1]Ali malik, Dongdong chenand Soek bum ko, "Design tradeoff analysis of floating point adders in FPGAs," Can. J. elect. Comput. Eng., ©2008IEEE.
- [2] IEEE standard for floating-point arithmetic(IEEE STD 754-2008),revision of IEEE std 754-1985.august (2008).
- [3] Karan Gumber,Sharmelee Thangjam "Performance Analysis of Floating Point Adder using VHDL on Reconfigurable Hardware" in International Journal of Computer Applications (0975 – 8887) Volume 46– No.9, May 2012
- [4] Loucas Louca, Todd A cook and William H. Johnson, "Implementation of IEEE single precision floating point addition and multiplication on FPGAs,"©1996 IEEE.
- [5] Ali malik, Soek bum ko , "Effective implementation of floating point adder using pipelined LOP in FPGAss," ©2010 IEEE.
- [6] Metin Mete, Mustafa Gok, "A multiprecision floating point adder" 2011 IEEE.
- [7] Florent de Dinechin, "Pipelined FPGA adders" ©2010 IEEE.
- [8] B. Fagin and C. Renard, "Field Programmable Gate Arrays and Floating Point Arithmetic," IEEE Transactions on VLSI, vol. 2, no. 3, pp. 365–367, 1994.
- [9] A. Jaenicke and W. Luk, "Parameterized Floating-Point Arithmetic on FPGAs", Proc. of IEEE ICASSP, 2001, vol. 2, pp.897-900.
- [10] 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