Volume 2, Issue 5, May 2012 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Implementation of Floating point Adder on Reconfigurable Hardware and Study its Effect on Chip Area

Karan Gumber[*] U.I.E.T Panjab university er.gumber88@gmail.com
Sharmelee Thangjam U.I.E.T Panjab university
Meenu Talwar CGC,Landran, PTU
Pardeep Sharma SLIET,Longowal

Abstract—**Floating point adders are hard to implement on reconfigurable hardware because floating point addition is the most complex operation since the alignment of mantissa is required before mantissa addition. Various parameters are outlined such as clock period, combinational delay, chip area i.e. number of slices, clock speed etc. when we implement floating point adder on reconfigurable hardware. Implementation of floating point adder on different FPGAs causes change in consumption of chip area on using different reconfigurable hardware. Implementation of floating point adder on Virtex 4 consumes only 7% a chip area i.e. consumes 401 slices out of 5472 slices with a offset delay of 27.891nsec while implementation of floating point adder on Spartan 2 consumes 52% of chip area causes a offset delay of 79.378nsec and the greatest utilization of chip area is required while using Virtex 2 for the implementation it utilizes 155% of chip area.**

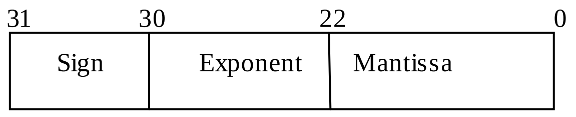
Keywords— **Floating point adder, FPGAs, Combinational delay, Chip area, Xilinx.**

I. INTRODUCTION

Addition is the most complex operation in a floating point unit and can cause major delay while requiring a significant area. Over the years, VLSI community has developed many floating point adder algorithm aimed primarily at reducing the overall latency. VHDL code for floating point adder is implemented on FPGA using Xilinx. Synthesis report in Xilinx provide us all the parameters like combinational delay, chip area, components used, number of paths, modelling format etc. Customizations were performed where this was possible in order to save chip area and to reduce latency of design. Floating point operations are hard to implement on FPGAs because of complexity of their algorithms. They usually require excessive chip area, a resource that is limited in FPGAs. This problem becomes harder if the 32 bit operations are required [1] [2]. On the other hand, many scientific problems require floating point arithmetic with high level of accuracy in their calculations. By the early 1980s, custom integrated circuits (ICs) were often designed to replace the large amount of glue logic in electronic devices and thus reducing manufacturing cost and system complexity. However, custom ICs are expensive to develop; they are viable for high volume products. To address this limitation, Xilinx produced Field programmable gate array technology in 1984 as an alternative to custom ICs. FPGAs is used to model a digital circuit ranging from gate level to very complex level it is a silicon chip with unconnected logic blocks. These blocks can be defined and redefined by the user at any time. FPGAs are increasingly being used for the applications which require high numerical stability and accuracy. FPGAs are becoming attractive solution for low volume application compared to application (ASICs) [1] [3]. Floating point adders are used in number of applications out of which the most familiar are DSP/math processors, air traffic control, digital computers, robots etc. The floating point addition is the most complex operation then the floating point multiplication or division since the alignment of mantissa is required before mantissa addition [3].

II. FLOATING NUMBER FORMAT

The advantage of floating point arithmetic over the fixed point arithmetic is the range of numbers that can be presented with the fixed number of bits [4]. Floating point number is composed of three fields and can be of 16, 18, 32 and 64 bit format. Figure 3 shows the 32 bit format of IEEE standard for floating point arithmetic.



1-bit sign, S: A value of '1' indicates a number is negative and '0' indicates positive number.
Bias-127exponent, e=E + Bias**:** This gives us an exponent range from E (min) = -126 to E (max) =127
Mantissa: The fractional part of a number. The fractional part must not be confused with the significand which is 1 plus the fractional part. The leading 1 in the significand is implicit. When performing operation with this format, the implicit bit is usually made explicit [5][6].

A. Conversion of Decimal to Floating 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 always '1' 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. 01000011000000011101110000000000. Now take another example which is totally different from above let us enter a decimal number -0.5 which is converted into binary value which is .000011. After conversion move the radix point to the right in this case 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 always '1' 1.10000000000000000000000 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 to the right say 'x'. In this case there is 5 times shifting of radix point to the right. This value must be subtracted to 127 to get the exponent value i.e. original exponent value is 127 - 'x'. In above case exponent is 127 - 5= 122 which is 01111010. Sign bit i.e. MSB is '1' because number is -ve. Now assemble result into 32 bit format which is sign, exponent, mantissa 10111101010000000000000000000000.

#Note: - Hidden bit is not included into 32 bit format this bit is implicit. When performing operation with this format this implicit bit is made explicit [5].

III. ADDITION ALGORITHM FOR FLOATING POINT NUMBERS

The floating point addition is the most complex operation then the floating point multiplication since the alignment of mantissa is required before mantissa addition. I would like to explain floating point addition algorithm in 2 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. The flowchart of the algorithm is given below in next page and it is explained in following steps with proper example.

A. *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 [6][7].

Example

Step 1: Enter N1 and N2.

N1=2.3=0 10000000 10010010000000000000000

N2=7.4=0 10000001 11101100000000000000000

E1= 10000000

E2= 10000001

S1=100100100000000000000000

S2= 111011000000000000000000

Step 2: If E2>E1. Yes then swap N1 & N2.

New N1=0 10000010 11101100000000000000000

New N2=0 10000000 10010010000000000000000

Step 3: Calculate d =E1-E2.

10000001-10000000 = 1

Step 4: Shifting of S2 to the right by one and also add 1 to E2.

N2 =0 10000000 100100100000000000000000(original)

N2= 0 10000000 010010010000000000000000 (one time shifted)

Shifting by 1 time means add '1' to exponent.

Step 5: New exponent value E2 = 10000001, new significand value S2 = 010010010000000000000000 here E1 = E2.

Step 6: S=S1+S2.

S1=111011000000000000000000

S2=010010010000000000000000

S=100110101000000000000000

Step 7: Here is carry out add '1' to exponent and shift result to the right by one bit and discard the LSB of 'S'.
Original exponent=10000010
Original significand=100110101000000000000000
Step 8: MSB of result is '0'.
Step 9: Assemble into 32 bit format.
0 10000010 00110101000000000000000

B. *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 N1or it can be MSB of N2.
Step 10:- Assemble result into 32 bit format excluding 24th bit of significand i.e. hidden bit [6][7].

Example

Step 1: Enter N1 and N2.
N1=128.5=0 10000110 10000000100000000000000
N2=-18.25=1 10000011 10010010000000000000000
E1=10000110
E2=10000011
S1=100000001000000000000000
S2=100100100000000000000000
Step 2: E1>E2 no need to swap.
Step 3: Calculate 'd'=E1-E2.
10000110-10000011=00000011=>3 in decimal.
Step 4: Shifting of S2 to the right by three and also add 3 to E2.
N2 = 0 10000011 100100100000000000000000 (original)
N2 = 0 10000100 010010010000000000000000 (1 time shifting)
N2 = 0 10000101 001001001000000000000000 (2 time shifting)
N2 = 0 10000110 000100100100000000000000 (3 time shifting)
Shifting by 1 time means add '1' to exponent.
Step 5: New exponent value E1 = 10000110, new significand value S2 = 000100100100000000000000 here E1=E2 i.e. result is in normalized form.
Step 6: Take 2's complement of S2 because S2 is -ve i.e. S=S1+2's complement of S2.
S1=100000001000000000000000
S2=111011011100000000000000

S=1011011100100000000000000

Step 7: Here is carry out add lets discard the carry and shift result to the leftt by one bit to make MSB '1' and then subtract the amount of shifting from E1 or E2 to form original exponent of result.
Original exponent=10000110-1=10000101
Original significand=110111001000000000000000
Step 8:- Sign bit of result i.e. MSB= Sign of 128.5 which is larger number.
Step 9: Assemble into 32 bit format.
0 10000101 11011100100000000000000

C. *Special Conditions*

There are some special conditions while implementing floating point adder which needs to be handle these are explained below
1: If N1 = N2 = '0' then overall result is '0'.
2: If E1=E2 and sign bit of E1 ≠ E2 then again overall result is '0'.
3: If E1= '0' and E2 ≠ '0' then overall result is equal to E2.
4: If E2= '0' and E1 ≠ '0' then overall result is equal to E1
5: If d= E1-E2 ≥24 then overall result is larger of E1 or E2 [3].

D. *Problems associated in addition*

There are two problems which occurs when we are going to add two floating point numbers
1: When the exponent of two numbers are different this can be solved by shifting the significand of smaller number to the right by an amount equal to exponent difference and this amount is added to exponent value of smaller number to make exponent of both the numbers are same means in normalized form
2: When there is carry out in significand addition if both the number are of different sign then add '1' to the exponent and shift the result of significand to the right by one discarding LSB and if both the number are of different sign then discard the carry and shift the result to the left until there is '1' at MSB the amount of shifting is subtracted from exponent to form real exponent [7][8].

IV. FINAL RESULTS/SYNTHESIS REPORT

Synthesis report shows the combinational delay when we are going to implement floating point adder on various hardware i.e. Spartan 2, Spartan 3, virtex 2, virtex 4 etc

TABLE I

RECONFIGURABLE HARDWARE	NUMBER OF SLICES USED OUT OF TOTAL AVAILABLE	UTILIZATION IN%
SPARTAN 2	402/768	52%
SPARTAN 2E	402/768	52%
SPARTAN 3	402/768	52%
SPARTAN 3E	401/960	41%
VIRTEX	402/768	52%

VIRTEX 2	399/256	155%
VIRTEX 4	401/5274	7%
VIRTEX E	402/768	52%

CONCLUSIONS

From the final results it is concluded that implementation of floating point adder on different reconfigurable hardware causes change in their utilization of chip area when we implement adder on virtex 4 it causes least utilization of chip area while implementation of adder on Virtex 2 causes great amount of utilization of chip area.

ACKNOWLEDGMENT

I would like to thanks the anonymous reviewers for their insightful comments.

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] Loucas Louca, Todd A cook and William H. Johnson, "Implementation of IEEE single precision floating point addition and multiplication on FPGAs,"©1996 IEEE.

[3] Alexandru, Mircea, Lucian and Oana, "Exploiting parallelism in double path adder structure for increase througput of floating point addition ," ©2007 IEEE.

[4] V. Y. Gorshtein, A. I Grushin, SR Shevtsov, "Floating point addtion method and apparatus," Sun microsystem U.S patent 5808926,1998.

[5] IEEE std. 1076-2002, "IEEE stsndard VHDL reference manual," Sponsored by Design Automation standards Committee published by IEEE.

[6] Metin Mete, Mustafa Gok, "A multiprecision floating point adder," ©2011 IEEE.

[7] Florent de Dinechin, "Pipelined FPGA adders," ©2010 IEEE.

[8] Ali malik, Soek bum ko , "Effective implementation of floating point adder using pipelined LOP in FPGAss," ©2010 IEEE.

[9] Allan, Wayne Luk, " Parametised floating point arithmetic on FPGA,"© 2001 IEEE

[10] Dr. John A. Eldon, Craig Robertson, " A floating point format for signal processing," ©2002 IEEE

[11] Asger, David, C. N. lyu, " An IEEE complaint floating point adder that conforms with the pipelined packet forwarding paradigm," ©2000 IEEE