FPGA Based 32-Bit Floating Point Arithmetic Unit for Floating Point Processors

Archana Yadav1, Bijender Mehandia2
1Student M.Tech. ECE (2013-2015) Gurgaon Institute of Technology and Management (M.D.U)
Bilaspur-Taura Road, Gurgaon (Haryana) India
2HOD ECE Gurgaon Institute of Technology and Management (M.D.U)
Bilaspur-Taura Road, Gurgaon (Haryana) India

Abstract — Most of the algorithms implemented in FPGAs used to be fixed-point. Floating-point operations are useful for computations involving large dynamic range, but they require significantly more resources than integer operations. With the current trends in system requirements and available FPGAs, floating-point implementations are becoming more common and designers are increasingly taking advantage of FPGAs as a platform for floating-point implementations. The rapid advance in Field-Programmable Gate Array (FPGA) technology makes such devices increasingly attractive for implementing floating-point arithmetic. Compared to Application Specific Integrated Circuits, FPGAs offer reduced development time and costs. Moreover, their flexibility enables field upgrade and adaptation of hardware to run-time conditions. A 32 bit floating point arithmetic unit with IEEE 754 Standard has been designed using VHDL code and all operations of addition, subtraction, multiplication and division are tested on Xilinx.

Keywords- Arithmetic Unit, Floating Point, VHDL, Simulink, Modelsim

1. Introduction

The floating point operations have found intensive applications in the various fields for the requirements for high precise operation due to its great dynamic range, high precision and easy operation rules. High attention has been paid on the design and research of the floating point processing units. With the increasing requirements for the floating point operations for the high-speed data signal processing and the scientific operation, the requirements for the high-speed hardware floating point arithmetic units have become more and more exigent. The implementation of the floating point arithmetic has been very easy and convenient in the floating point high level languages, but the implementation of the arithmetic by hardware has been very difficult. With the development of the very large scale integration (VLSI) technology, a kind of devices like Field Programmable Gate Arrays (FPGAs) have become the best options for implementing floating hardware arithmetic units because of their high integration density, low price, high performance and flexible applications requirements for high precious operation. Floating-point implementation on FPGAs has been the interest of many researchers. The use of custom floating-point formats in FPGAs has been investigated in a long series of work [3], [4], [8]. In most of the cases, these formats are shown to be adequate for some applications that require significantly less area to implement than IEEE formats [6] significant speedups for certain chosen applications. The earliest work on IEEE floating-point [7] focused on single precision although found to be feasible but it was extremely slow. Eventually, it was demonstrated [8] that while FPGAs were uncompetitive with CPUs in terms of peak FLOPs, they could provide competitive sustained floating-point performance. Since then, a variety of work [2], [5], [9]-[10] has demonstrated the growing feasibility of IEEE compliant, single precision floating point arithmetic and other floating-point formats of approximately same complexity. In [2], [5], the details of the floating-point format are varied to optimize performance. The specific issues of implementing floating-point division in FPGAs have been studied [10]. Early implementations either involved multiple FPGAs for implementing IEEE 754 single precision floating-point arithmetic, or they adopted custom data formats to enable a single-FPGA solution. To overcome device size restriction, subsequent single-FPGA implementations of IEEE 754 standard employed serial arithmetic or avoided features, such as supporting gradual underflow, which are expensive to implement.

In this paper, a high-speed IEEE754-compliant 32-bit floating point arithmetic unit designed using VHDL code has been presented and all operations of addition, subtraction, multiplication and division got tested on Xilinx and verified successfully along with that all the exceptions of floating point numbers are studied in detail. The simulation results of addition, subtraction, multiplication and division in Modelsim wave window.

The rest of the paper is organized as follows. Section 2 presents the general floating point architecture and conversion of decimal to floating point format. Section 3 explains the algorithms used to write VHDL codes for implementing 32 bit floating point arithmetic operations: addition/subtraction, multiplication and division. And the results are shown and discussed in its section 4 while section 5 concludes the paper with further scope of work.

2. Floating Point Architecture
Floating point numbers are one possible way of representing real numbers in binary format; the IEEE
754 [11] standard presents two different floating point formats, Binary interchange format and Decimal interchange format. This paper focuses only on single precision normalized binary interchange format. Figure 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.

32 bit Single Precision Floating Point Numbers
IEEE standard are stored as:
S EEEEEEEE MMMMMMMMMMMMMMMMMMM
S: Sign – 1 bit
E: Exponent – 8 bits
M: Mantissa – 23 bits Fraction

Sign bit

<table>
<thead>
<tr>
<th>8 bits</th>
<th>23 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Biased exponent</td>
<td>Significant</td>
</tr>
<tr>
<td>32 bits</td>
<td></td>
</tr>
</tbody>
</table>

Fig.1: IEEE 754 single precision binary format representation

The value of number V:
If E=255 and F is nonzero, then V= Nan ("Not a Number")
If E=255 and F is zero and S is 1, then V= - Infinity
If E=255 and F is zero and S is 0, then V= Infinity
If 0<E<255 then V= (-1)**S * 2 **(E-127) * (1.F) (exponent range = -127 to +128)
If E=0 and F is nonzero, then V= (-1)**S * 2 **(-126) * (0.F) ("un-normalized" values)
If E=0 and F is zero and S is 1, then V= -0
If E=0 and M is zero and S is 0, then V = 0

An extra bit is added to the mantissa to form what is called the significant. 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 case the real number is represented by (1)

\[ V = (-1)^S * 2^{(E - Bias)} * (1.M) \]  
(1)

Where \( M = m22 2^{-1} + m21 2^{-2} + m20 2^{-3} + \ldots + m1 2^{-22} + m0 2^{-23} \); Bias = 127.

2.1 Conversion of Decimal to Floating numbers
Conversion of Decimal to Floating point 32 bit formats is explained with example. Let us take an example of a decimal number that how could it 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 1.0000011101110000000000000 the number which is after the radix point is called mantissa which is of 23 bits and the whole number is called significant 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 1000110. Sign bit i.e. MSB is “0” because number is +ve. Now assemble result into 32 bit format which is sign, exponent, mantissa 01000011000000000110111000000000. 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” 1.100000000000000000000000 the number which is after the radix point is called mantissa which is of 23 bits and the whole number is called significant 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 01110110. Sign bit i.e. MSB is “1” because number is -ve. Now assemble result into 32 bit format which is sign, exponent, mantissa 10111110101000000000000000000000

3. Algorithms for Floating Point Arithmetic Unit

The algorithms using flow charts for floating point addition/subtraction, multiplication and division have been described in this section, that become the base for writing VHDL codes for implementation of 32-bit floating point arithmetic unit.

3.1 Floating Point Addition / Subtraction

The algorithm for floating point addition is explained. While adding the two floating point numbers, two cases may arise. Case I: when both the numbers are of same sign i.e. when both the numbers are either +ve or –ve. In this case 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. In this case the MSB of one number is 1 and other is 0.

Case I: - When both numbers are of same sign
Step 1:- Enter two numbers N1 and N2. E1, S1 and E2, S2 represent exponent and significant of N1 and N2 respectively.
Step 2:- Is E1 or E2 = “0”. If yes; set hidden bit of N1 or N2 is zero. If not; then check if E2 > E1, if yes
swap N1 and N2 and if E1 > E2; contents of N1 and N2 need not to be swapped.

Step 3:- Calculate difference in exponents d=E1-E2. If d= “0” then there is no need of shifting the significant. 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:- Check if N1 and N2 have different sign, if “no”;

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

Step 7:- Check if there is carry out in significant addition. If yes; then add “1” to the exponent value of either E1 or new E2. After addition, shift the overall result of significant addition to the right by one by making MSB of S as “1” and dropping LSB of significant.

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 significant i.e. hidden bit.

Case II: - When both numbers are of different sign

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

Step 5:- Check if N1 and N2 have different sign, if “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:- Check if there is carry out in significant addition. If yes; then discard the carry and also shift the result to left until there is “1” in MSB and also count 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 significant i.e. hidden bit.

In this algorithm three 8-bit comparators, one 24-bit and two 8-bit adders, two 8-bit subtractors, two shift units and one swap unit are required in the design.

First 8-bit comparator is used to compare the exponent of two numbers. If exponents of two numbers are equal then there is no need of shifting. Second 8-bit comparator compares exponent with zero. If the exponent of any number is zero set the hidden bit of that number zero. Third comparator is required to check whether the exponent of number 2 is greater than number 1. If the exponent of number 2 is greater than number 1 then the numbers are swapped.

One subtractor is required to compute the difference between the 8-bit exponents of two numbers. Second subtractor is used if both the numbers are of different sign than after addition of the significants of two numbers if carry appears. This carry is subtracted from the exponent using 8-bit subtractor.

One 24-bit adder is required to add the 24-bit significant of two numbers. One 8-bit adder is required if both the numbers are of same sign than after addition of the significant of two numbers if carry appears. This carry is added to the exponent using 8-bit adder. Second 8-bit adder is used to add the amount of shifting to the exponent of smaller number. One swap unit is required to swap the numbers if N2 is greater than N1. Swapping is normally done by taking the third variable. Two shift units are required one is shift left and second is shift right.

3.2 Floating Point Multiplication

The algorithm for floating point multiplication is explained through flow chart in Figure 3. Let N1 and N2 are normalized operands represented by S1, M1, E1 and S2, M2, E2 as their respective sign bit, mantissa (significant) and exponent. Basically following four steps are used for floating point multiplication.

1. Multiply significant, add exponents, and determine sign

M=M1*M2
E=E1+E2-
Bias
S=S1XORS2

2. Normalize Mantissa M (Shift left or right by 1) and update exponent E

3. Rounding the result to fit in the available bits

4. Determine exception flags and special values for overflow and underflow.

Sign Bit Calculation: The result of multiplication is a negative sign if one of the multiplied numbers is of a negative value and that can be obtained by XORing the sign of two inputs.

Exponent Addition is done through unsigned adder for adding the exponent of the first input to the exponent of the second input and after that subtract the Bias (127) from the addition result (i.e. E1+E2 - Bias). The result of this stage can be called as intermediate exponent. Significant Multiplication is done for multiplying the unsigned significant
and placing the decimal point in the multiplication product. The result of significant multiplication can be called as intermediate product (IP). The unsigned significant multiplication is done on 24 bit. The result of the significant multiplication (intermediate product) must be normalized to have a leading “1” just to the left of the decimal point (i.e. in the bit 46 in the intermediate product). Since the inputs are normalized numbers then the intermediate product has the leading one at bit 46 or 47. If the leading one is at bit 46 (i.e. to the left of the decimal point) then the intermediate product is already a normalized number and no shift is needed. If the leading one is at bit 47 then the intermediate product is shifted to the right and the exponent is incremented by 1.

Overflow/underflow means that the result’s exponent is too large/small to be represented in the exponent field. The exponent of the result must be 8 bits in size, and must be between 1 and 254 otherwise the value is not a normalized one .An overflow may occur while adding the two exponents or during normalization. Overflow due to exponent addition can be compensated during subtraction of the bias; resulting in a normal output value (normal operation). An underflow may occur while subtracting the bias to form the intermediate exponent. If the intermediate exponent < 0 then it is an underflow that can never be compensated; if the intermediate exponent = 0 then it is an underflow that may be compensated during normalization by adding 1 to it .When an overflow occurs an overflow flag signal goes high and the result turns to ±Infinity (sign determined according to the sign of the floating point multiplier inputs). When an underflow occurs an underflow flag signal goes high and the result turns to ±Zero (sign determined according to the sign of the floating point multiplier inputs).When an underflow occurs an underflow flag signal goes high and the result turns to ±Zero (sign determined according to the sign of the floating point multiplier inputs).

3.3 Floating Point Division

The algorithm for floating point multiplication is explained through flow chart in Figure 4. Let N1 and N2 are normalized operands represented by S1, M1, E1 and S2, M2, E2 as their respective sign bit, mantissa (significant) and exponent. If let us say we consider x=N1 and d=N2 and the final result q has been taken as “x/d”. Again the following four steps are used for floating point division.
1. Divide significant, subtract exponents, and determine sign M=M1/M2 E=E1-E2 S=S1XORS2
2. Normalize Mantissa M (Shift left or right by 1) and update exponent E
3. Rounding the result to fit in the available bits
4. Determine exception flags and special values

The sign bit calculation, mantissa division, exponent subtraction (no need of bias subtraction here), rounding the result to fit in the available bits and normalization is done in the similar way as has been described for multiplication.

4. RESULTS

The VHDL code written has been tested and verified on Xilinx ISE 8.1i for all operation. The design utilization summary has been shown in Figure 2.

Fig.2: Design Utilisation summary of floating point arithmetic unit on FPGA

Fig.3: Simulation results of decimal inputs 1.1 &1.1 for adder in modelsim wave window

Fig. 4: Simulation result of decimal inputs 2.5& 4.75 for adder in modelsim wave window
5. Conclusion and Future Scope of Work

The VHDL code written for complete 32-bit floating point arithmetic unit has been implemented and tested on Xilinx. The designed arithmetic unit operates on 32-bit operands. It can be designed for 64-bit operands to enhance precision. It can be extended to have more mathematical operations like trigonometric, logarithmic and exponential function.

References

1] N. Shirazi, A. Walters and P. Athanas,

ACKNOWLEDGEMENTS

I would like to thanks the anonymous users for their insightful comment.