ISSN: 2278 – 909X
International Journal of Advanced Research in Electronics and Communication Engineering (IJARECE)
Volume 4, Issue 7, July 2015

VHDL implementation of 32-bit floating point unit (FPU)

Nikhil Arora
M.Tech student
YMCA, Faridabad

Govindam Sharma
M.Tech student
YMCA, Faridabad

Sachin Kumar
M.Tech student
YMCA, Faridabad

***Abstract*—The IEEE Standard for Floating-Point Arithmetic (IEEE 754) is a technical standard for floating-point computation established in 1985 by the proposed work was implemented using the VHDL language and simulation was verified. This paper is focused on implementation of floating point arithmetic based on IEEE 754 standard.**

***Keywords*— IEEE754, Floating Point Unit, Floating point adder, Floating point subtracter, Floating point multiplier.**

I. INTRODUCTION

The term floating point is derived from the fact that there is no fixed number of digits before and after the decimal point. In fixed-point representations there are fixed number of digits before and after the decimal [1]. Floating point representations can handle a larger range of numbers. For example, a fixed point representation that has seven decimal digits, with the decimal point assumed to be positioned after the fifth digit, can represent the numbers 12345.67, 8765.43, 123.00, and so on, whereas a floating-point representation (such as the IEEE 754 decimal32 format) with seven decimal digits could in addition represent 1.234567, 123456.7, 0.00001234567, and 1234567000000. Floating-point is found in computer systems [2]. With the vast development in the very large scale integration (VLSI) circuit technology, many complex circuits have become easily implementable today [3].
This paper is organized as follows: section 2 consists of floating point arithmetic which explains floating point addition subtraction, multiplication, division. Section 3 gives the simulation results. Section 4 summaries the conclusion and future work.

II. FLOATING POINT ARITHMETIC

Floating point unit implemented here are addition, subtraction, multiplication, division. Pre-normalize an operand means that the operand is transformed into formats that make it easy and efficient to handle internally. Post-normalize the result is done if the result is not as per the IEEE format then it is Transformed into the specified IEEE format. Before performing arithmetic operations, the input variables Are first pre-normalized and then after performing the result is first checked whether an exception is occurred or not, and if not then result is post-normalized.

A. Floating point addition

The conventional floating-point addition algorithm consists of five stages - exponent difference, pre-alignment, addition, normalization and rounding. Given floating-point numbers X1 = (s1, e1, f1) and X2 = (s2, e2, f2), the stages for computing X1 + X2 are described as follows:

• If SIGN of two operands are same then SIGN of result is also same.
• If the two operands are having different signs then:
• SIGN of result is same as that of the operand with larger EXPONENT.
• If the two exponents are also equal then the SIGN of result is same as that of the operand with larger MANTISSA.
• Find EXPONENT difference d = e1-e2. If e1 < e2, swap position of MANTISSAS. Set larger EXPONENT as tentative exponent of result.
• Pre-align MANTISSAS by shifting smaller MANTISSA right by'd' bits.
• Add MANTISSAS to get tentative result for mantissa.
• Round MANTISSA result. If it overflows due to rounding, shift right and increment EXPONENT by 1-bit.

Normalization: If there are leading-zeros in the tentative result, shift result left and decrement EXPONENT by the number of leading zeros. If tentative result overflows, shift right and increment EXPONENT by 1 -bit.

Illustration of floating point addition:
A= 01101011111100111010000011000011
B= 01101000100011100101111100011100
Step - 1: compare the exponents:
Exponents are not equal. So, normalize the smaller exponent to the larger exponent. This will lead to:
Exponent (A) – Exponent (B)
215 - 209 = 6.
Shift B (number with smaller exponent) right 6 times.
B = 0.0000010001110010111110 0x2A 215.
The six least significant bits are truncated.
Now the two exponents are same.

Step - 2: add the mantissas:
1.11100111010000011000011
+0.00000100011100101111100
1.11101011101101000111111x 2A215
Step - 3: Result is in IEEE standard format. So, there is no need of post-normalizing the result.
01101011111101011101101000111111
The algorithm used for adding floating point numbers is shown in the figure 1.
The conventional floating-point subtraction algorithm consists of five stages – exponent difference, pre-alignment, subtraction, normalization and rounding. Given floating-point numbers X1 = (s1, e1, f1) and X2 = (s2, e2, f2), the stages for computing X1 - X2 are described as follows:

• If SIGN of two operands are same then SIGN of result is also same.
• If the two operands are having different signs then:
• SIGN of result is same as that of the operand with larger EXPONENT.
• If the two EXPONENTS are also equal then the SIGN of result is same as that of the operand with larger MANTISSA.

The algorithm used for adding floating point numbers is shown in the figure 1.

B. Floating point subtraction

The conventional floating-point subtraction algorithm consists of five stages – exponent difference, pre-alignment, subtraction, normalization and rounding. Given floating-point numbers X1 = (s1, e1, f1) and X2 = (s2, e2, f2), the stages for computing X1 - X2 are described as follows:

- If SIGN of two operands are same then SIGN of result is also same.
- If the two operands are having different signs then:
- SIGN of result is same as that of the operand with larger EXPONENT.
- If the two EXPONENTS are also equal then the SIGN of result is same as that of the operand with larger MANTISSA.
- Find EXPONENT difference d = e1-e2. If e1 < e2, swap position of mantissas. Set larger EXPONENT as tentative exponent of result.
- Pre-align MANTISSAS by shifting smaller MANTISSA right by d bits.
- Subtract MANTISSAS to get tentative result for MANTISSA.

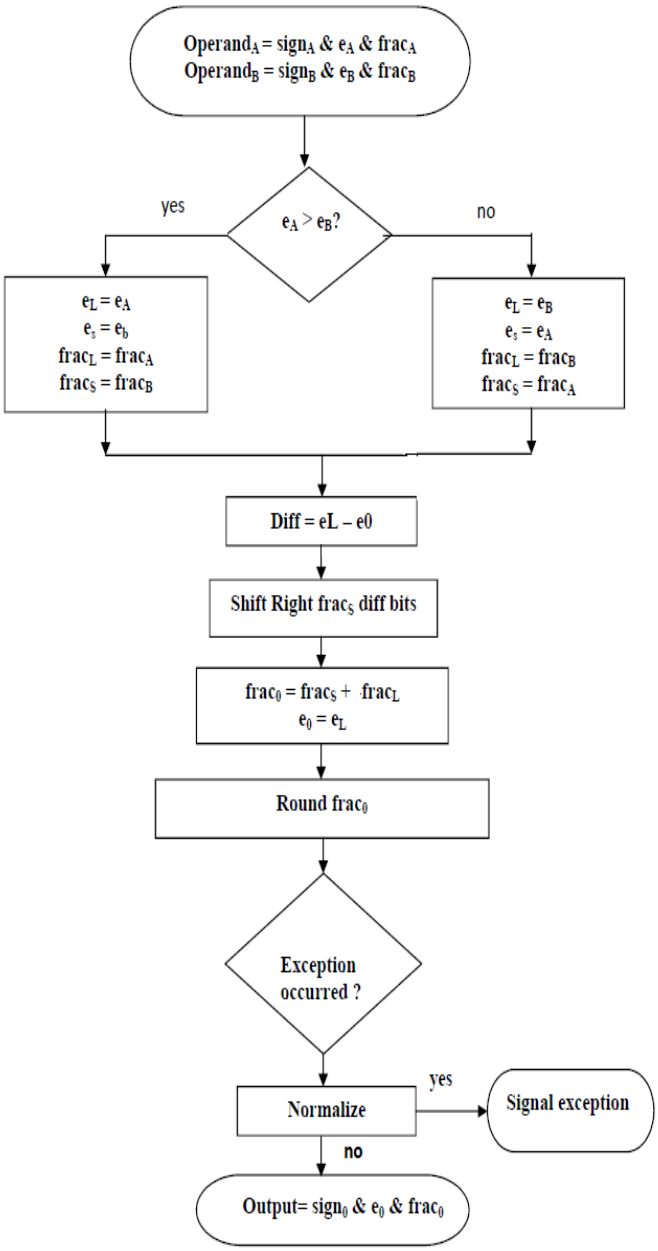


Figure 1: Block diagram of floating point addition.

- Normalization: If there are leading-zeros in the tentative result, shift result left and decrement EXPONENT by the number of leading zeros. If tentative result overflows, shift right and increment EXPONENT by 1-bit.
- Round MANTISSA result. If it overflows due to rounding, shift right and increment EXPONENT by 1-bit.

Illustration
A=01000001001011000000000000000000
B=01000000110100000000000000000000
Step - 1: compare the exponents:
Exponent(A) = 130 = 10000100.
Exponent(B) = 129 = 10000011.
Exponents are not equal. So, normalize the smaller exponent to the larger exponent. This will lead to:
EXPONENT(A) - EXPONENT(B)

130 – 129 = 6.Shift B (number with smaller exponent) right once.
B= 0.11010000000000000000000×2^ 130.
The least significant bit is truncated.
Now the two exponents are same.
Step-2: subtract the mantissas:
1.01011000000000000000000
+ 0.11010000000000000000000
0.10001000000000000000000 × 2^130
Step-3: post-normalize the result:
2^130 = 1.00010000000000000000000 × 2^129
1 right shift is required for this resulting into reducing 1 from the exponent. Our new exponent is 129. So, the result is: 01000000100010000000000000000000
The algorithm used for subtraction floating point numbers is shown in the figure 2.

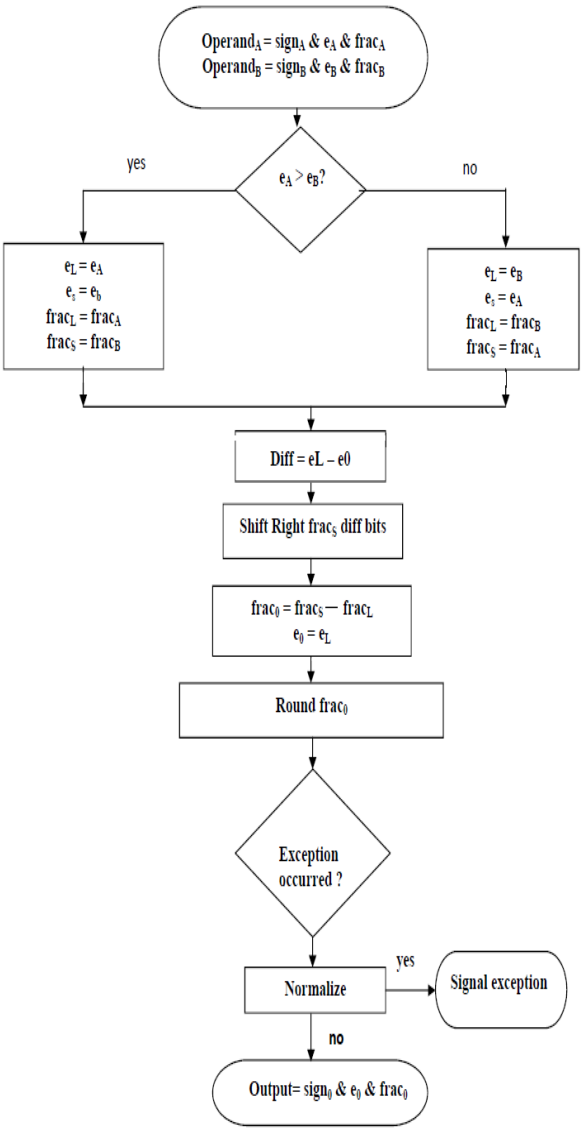


Figure 2: Block diagram of floating point subtraction.

C. *Floating-point multiplication*

In discussing floating-point multiplication, by complies with the IEEE 754 Standard, the two mantissas are to be multiplied, and the two exponents are to be added.

In order to perform floating-point multiplication, a simple algorithm is realized:

- XOR the SIGN bits.
- Add the exponents and subtract 127 (bias).
- Multiply the mantissas.
- Normalize the resulting value, if necessary.

Illustration:
A=01000001010010000000000000000000
B=11000001010010000000000000000000
EXPONENT(A) = 130 = 10000010.
EXPONENT(B) = 130 = 10000010.
STEP - 1: XOR the sign bit:
0 XOR 1 = 1.
Therefore the result will be negative.
Figure 3 depicts a generic flow chart for a floating-point multiplier.
STEP - 2: calculate the exponent of result by adding the two Exponents and subtract the bias:
130 + 130 - 127 = 133.
STEP - 3: multiply the mantissas:
1.10010000000000000000000
X1.10010000000000000000000
10.01110001000000000000000x 2^133
STEP - 4: post-normalize the result:
10.01110001000000000000000 X 2^133
1 is added to the exponent. So, our exponent is 134 i.e. 10000110.
So, the result is:
11000011000111000110000000000000

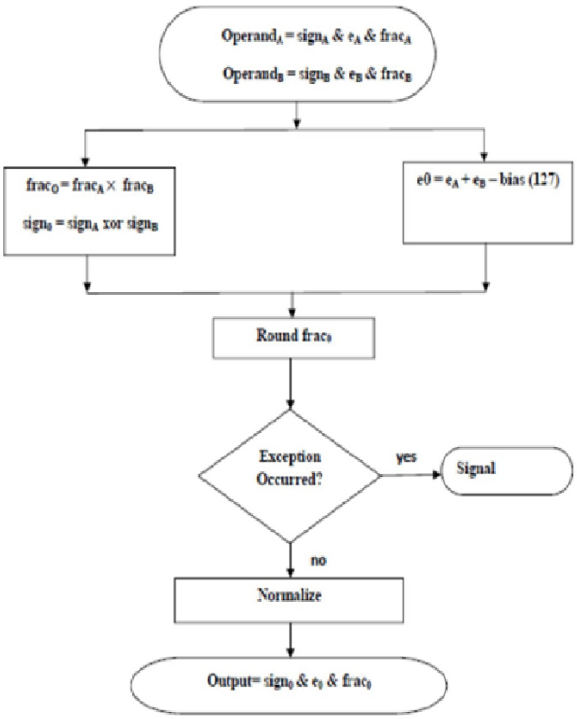


Figure 3: Block diagram of floating point multiplication.

OPCODE	OPERATION
00	MULTIPLICATION
01	DIVISION
10	ADDITION
11	SUBTARCTION

The result of multiplication may lie in the following range: $1 \leq \text{RESULT} \leq 4$.

D. *Floating point division:*

The algorithm for floating point division is as follows:
• XOR the SIGN bits.
• If divisor is 0 then result is INFINITE.
• Subtract the exponents and bias to it.
Else proceed as following:
• Divide the mantissa.
• Normalize the result if required.

The range for floating point division is given as:
0.5 < RESULT < 2.
A=01000001010010000000000000000000
B=01000001010010000000000000000000
EXPONENT(A) = 130 = 10000010.
EXPONENT(B) = 130 = 10000010.
Step - 1: XOR the sign bit:
0 XOR 1 = 1.
Therefore the result will be negative.
Step - 2: calculate the exponent of result by subtracting the Two exponents and adding the bias:
130 - 130 + 127 = 133.
Step - 3: divide the mantissas:
1.10010000000000000000000
- 1.10010000000000000000000
1.00000000000000000000000X 2^127
STEP - 3: Result is in IEEE standard format. So, there is no need of post-normalizing the result 01111111000000000000000000000000

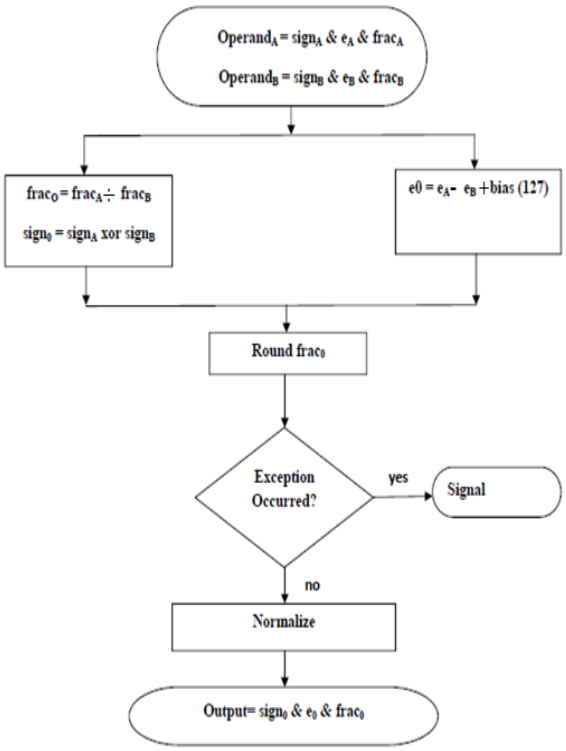


Figure 4: Block diagram of floating point division.

III. SIMULATION RESULTS

The arithmetic unit explained in last chapter has been coded in VHDL and simulated using ISE

Addition
Figure 5 shows the waveforms generated using ISE PROJECT NAVIGATOR while performing addition. The detailed description of the given inputs and the output generated is given further.

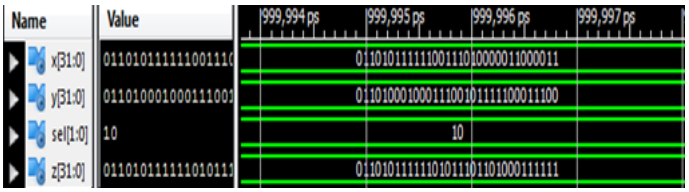


Figure 5: waveform for adder

Inputs:
X = 01101011111100111010000011000011
Y = 01101000100011100101111100011100
Sel = 10(opcode for addition).
Result:
Z = 01101011111101011101101000111111

Subtraction
Figure 6 shows the waveforms generated using ISE PROJECT NAVIGATOR while performing subtraction. The detailed description of the given inputs and the output generated is given further.

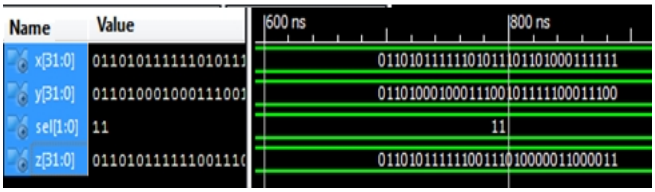


Figure 6: waveform for subtraction

Inputs:

X=01101011111101011101101000111111
y=01101000100011100101111100011100
Sel=11 (opcode for subtraction)
Result:
Z = 01101011111100111010000011000011

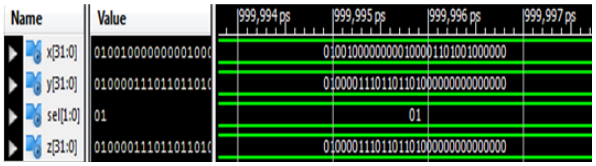
Multiplication
Figure 7 shows the waveforms generated using ISE PROJECT NAVIGATOR while performing multiplication. The detailed description of the given inputs and the output generated is given further.

Figure 7: waveform for multiplication

Inputs:
X = 01000011101101101000000000000000
Y = 01000011101101101000000000000000
Sel = 00(opcode for multiplication)

Result:
Z =01001000000000100001101001000000

Division
Figure 8 shows the waveforms generated using ISE PROJECT NAVIGATOR while performing division. The detailed description of the given inputs and the output generated is given further.



Inputs:

Figure 8: waveform for Division

X = 01001000000000100001101001000000
Y = 01000011101101101000000000000000
Sel = 01(opcode for division)

Result:
Z = 01000011101101101000000000000000

IV. CONCLUSION AND FUTURE SCOPE

Arithmetic unit has been designed to perform four arithmetic operations, addition, subtraction, multiplication and division on floating point numbers. IEEE 754 standard based floating point representation has been used. The unit has been coded in VHDL by using Xilinx 13.3.The designed arithmetic unit operates on 32-bit operands.
Future work consists of implementation of design for 64- bit operands to enhance precision. The unit can also be designed for different number systems like hexadecimal, decimal etc. It can be extended to have more mathematical operations like trigonometric, logarithmic and exponential functions.

REFERENCES

[1] Stehlé, D., Lefèvre, V., and Zimmermann, P., "Searching worst cases of a one-variable function", IEEE Transactions on Computers, 54(3), pp. 340–346, 2005.

[2] Bruguera, J. D., and Lang, T., "Floating-point Fused Multiply-Add: Reduced Latency for Floating- Point Addition", Proceedings of the 17th IEEE Symposium on Computer Arithmetic, ISBN 0-7695-2366-8,pp. 42–51, IEEE Computer Society, 2005.

[3] Muller, J.-M., Elementary Functions: Algorithms and Implementation, 2nd edition, Chapter 10, ISBN 0-8176-4372-9, Birkhäuser, 2006.

[4] Shuchita Pare*1 Dr. Rita Jain*2 "32 Bit Floating Point Arithmetic Logic Unit ALU Design and Simulation" international journal of emerging trend in electronics and computer science volume 1 issue, 2012

[5] IEEE floating point ,Wikipedia