CS M152A - Lab 1, Floating Point Conversion

Name: Sum Li

UID: 505146702

Due Date: October 25, 2020

TA: Mohit Garg

1) Introduction and Requirement

For this lab, the goal is to use the Xilinx ISE program to design and test a combinational circuit that functions as a floating point converter. The circuit was given a 13-bit linear encoding of an analog signal and converted into a closest 9-bit floating point number. For the purpose of simplicity, the output format of the floating point representation is splitted into 1 signed bit (S), 3-bits exponent (E) and 5-bits significand (F). The represented value is calculated based on the following formula.

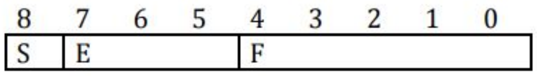


Figure 1: Output format of the floating point representation. S = Sign Bit; E = Exponent Bits; F = Significand / Mantissa Bits

$$V = (-1)^S \times F \times 2^E$$

Figure 2: Mathematical formula of the floating point representation

After the circuit is given the 13-bits linear encoding of a signal, it will make sure all the input numbers are positive in terms of decimal format. I will take the absolute value if the input number is negative in two complement format. The process is done by negating the negative number and adding one in order to count the number of leading zeroes. The number of leading zeros are then translated into the exponent value.

Leading Zeroes	Exponent
1	7
2	6
3	5
4	4
5	3
6	2
7	1
≥ 8	0

Figure 3: Encoding table from number of leading zeros in sign magnitude representation to exponent value in the output

For example, if I am given [0000_11010_0110] = 422 in two's complement positive representation, there are 4 leading zeros which then translate to exponent value equal to 4. The corresponding significand is [11010] which is 26 and sign bit is 0. However, if I am given [1_1110_0101_1010] = -422 in two's complement negative representation, then I have to first negate and add one in order to get the absolute value of the representation to count the number of leading zeros. For example, after negation and adding one, the representation becomes [0000_11010_0110] = 422, the corresponding significand is [11010] and sign bit is 1. However, there is error in the floating point representation due to the limited allocated bits for exponent and significand.

Due to the effect of the limited bits for the floating point representation, the designed circuit resolves this error by rounding the linear encoding to the nearest floating point encoding. The rounding rule depends on the 6th bit after the first 5 bits of the significand which is the bit followed after the last bit of the significand. If the 6th bit is 0, then the significand rounds down by staying the same. If the 6th bit is 1, then the significand rounds up by adding one.

Rounding Examples		
Linear Encoding	**Floating Point Encoding**	**Rounding**
0000001101100	[0 010 11011]	Down
0000001101101	[0 010 11011]	Down
0000001101110	[0 010 11100]	Up
0000001101111	[0 010 11100]	Up

Figure 4: Round up and Round down examples according to the 6th bit after the first 5 significand bits

However, there are two specific corner cases for the rounding stage of the floating point conversion. First, if all the bits in the significand are all ones which is [11111], it will cause overflow after 1 is being added. The designed circuit resolves the issue by shifting right a bit and increasing the exponent by 1. Therefore, the significand becomes [10000]. Second, if all the bits in the significand are all ones and as well as the bits in the exponent field are all ones which is 7, both the significand and the exponent field does not have enough bits to store the value after rounding. The problem is being resolved by using the largest possible floating point representation in 9 bits which means sign bit = [0/1], exponent bits = [111] and significand = [11111] when I am rounding larger linear encodings.

1) Design Description

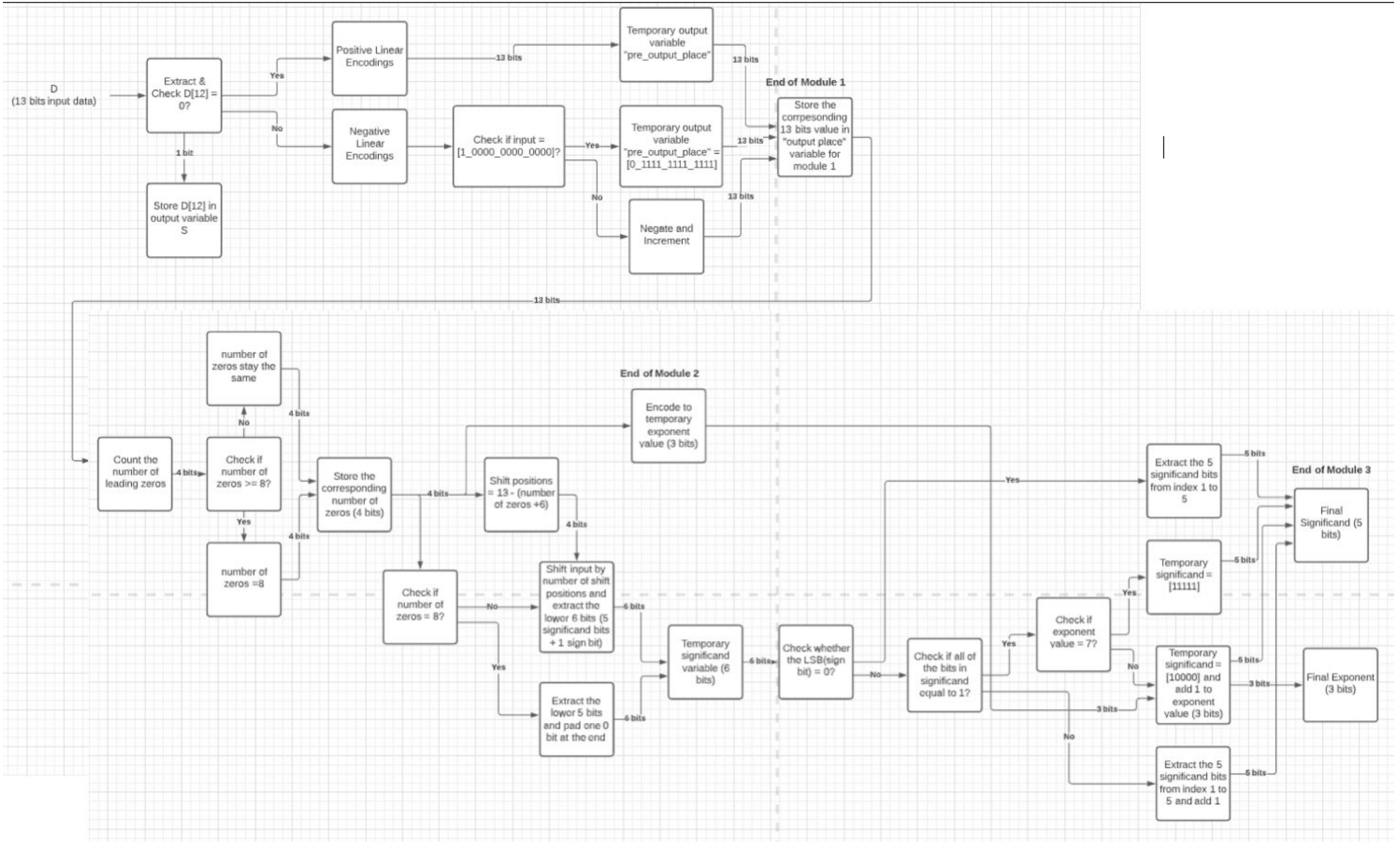

D
(13 bits input data)
Extract & Check D[12] = 0?
Yes
No
1 bit
Store D[12] in output variable S
Positive Linear Encodings
Negative Linear Encodings
13 bits
Temporary output variable "pre_output_place"
Check if input = [1_0000_0000_0000]?
Yes
No
Temporary output variable "pre_output_place" = [0_1111_1111_1111]
Negate and Increment
13 bits
13 bits
13 bits
End of Module 1
Store the corrpesonding 13 bits value in "output place" variable for module 1
13 bits
Count the number of leading zeros
4 bits
Check if number of zeros >= 8?
No
Yes
number of zeros stay the same
number of zeros =8
4 bits
4 bits
Store the corresponding number of zeros (4 bits)
4 bits
Shift positions = 13 - (number of zeros +6)
4 bits
Check if number of zeros = 8?
No
Yes
Shift input by number of shift positions and extract the lower 6 bits (5 significand bits + 1 sign bit)
Extract the lower 5 bits and pad one 0 bit at the end
6 bits
6 bits
End of Module 2
Encode to temporary exponent value (3 bits)
Temporary significand variable (6 bits)
6 bits
Check whether the LSB(sign bit) = 0?
Yes
No
Check if all of the bits in significand equal to 1?
Yes
No
Check if exponent value = 7?
Yes
No
Extract the 5 significand bits from index 1 to 5
Temporary significand = [11111]
Temporary significand = [10000] and add 1 to exponent value (3 bits)
Extract the 5 significand bits from index 1 to 5 and add 1
3 bits
5 bits
5 bits
5 bits
5 bits
3 bits
End of Module 3
Final Significand (5 bits)
Final Exponent (3 bits)

Figure 5: Overall design of the floating point conversion circuit

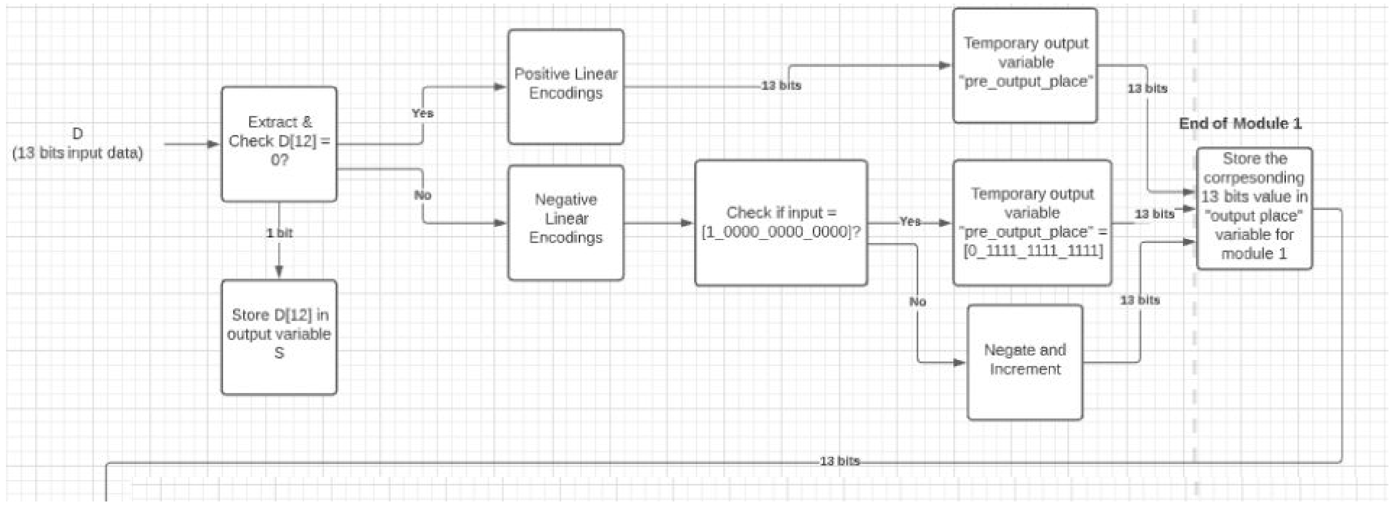

D
(13 bits input data)
Extract & Check D[12] = 0?
Yes
No
1 bit
Store D[12] in output variable S
Positive Linear Encodings
Negative Linear Encodings
13 bits
Temporary output variable "pre_output_place"
Check if input = [1_0000_0000_0000]?
Yes
No
Temporary output variable "pre_output_place" = [0_1111_1111_1111]
Negate and Increment
13 bits
13 bits
13 bits
End of Module 1
Store the corrpesonding 13 bits value in "output place" variable for module 1
13 bits

Figure 6: Close up on Module 1

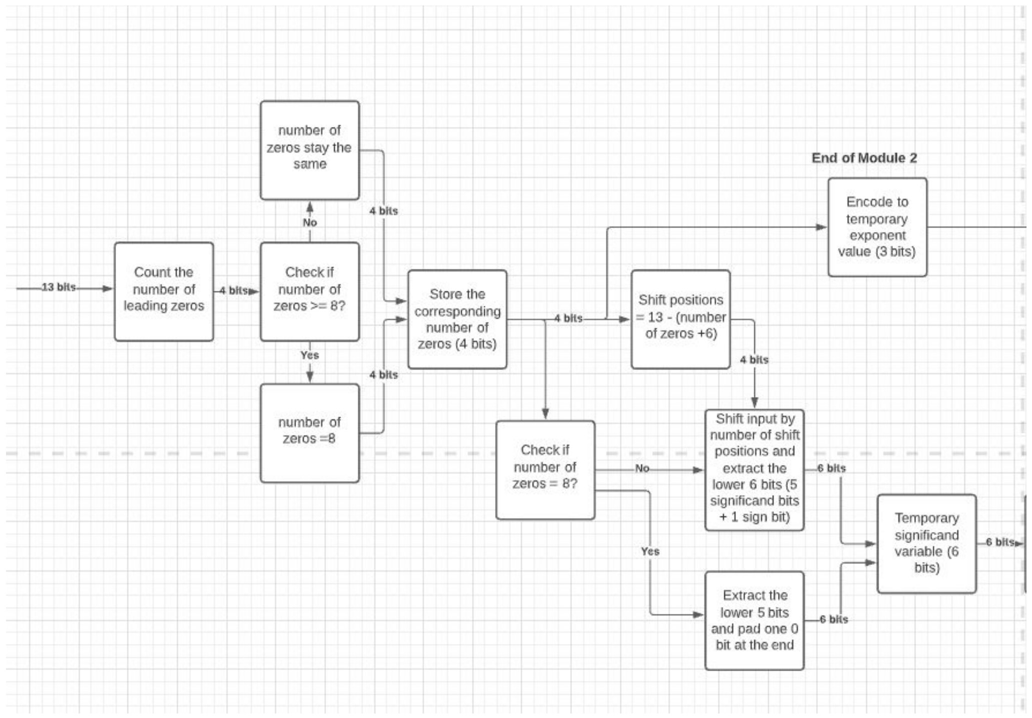

13 bits
Count the number of leading zeros
4 bits
Check if number of zeros >= 8?
No
number of zeros stay the same
Yes
number of zeros =8
4 bits
4 bits
Store the corresponding number of zeros (4 bits)
4 bits
End of Module 2
Encode to temporary exponent value (3 bits)
Shift positions = 13 - (number of zeros +6)
4 bits
Check if number of zeros = 8?
No
Yes
Shift input by number of shift positions and extract the lower 6 bits (5 significand bits + 1 sign bit)
Extract the lower 5 bits and pad one 0 bit at the end
6 bits
6 bits
Temporary significand variable (6 bits)
6 bits

Figure 7: Close up on Module 2

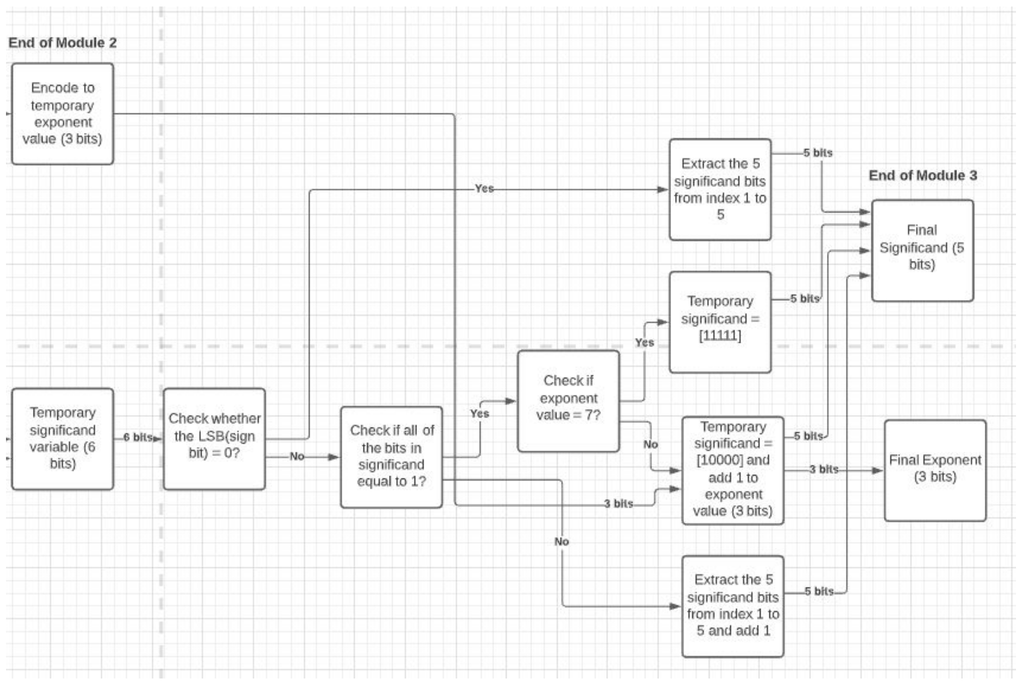

End of Module 2
Encode to temporary exponent value (3 bits)
Temporary significand variable (6 bits)
6 bits
Check whether the LSB(sign bit) = 0?
Yes
No
Check if all of the bits in significand equal to 1?
Yes
No
Check if exponent value = 7?
Yes
No
3 bits
Extract the 5 significand bits from index 1 to 5
Temporary significand = [11111]
Temporary significand = [10000] and add 1 to exponent value (3 bits)
Extract the 5 significand bits from index 1 to 5 and add 1
5 bits
3 bits
End of Module 3
Final Significand (5 bits)
Final Exponent (3 bits)

Figure 8: Close up on Module 3

The entire circuit is composed of three main blocks which are (1) two complement to sign magnitude conversion block, (2) counting leading zeros and extract leading zeros block and (3) rounding block.

(1) Two complement to Sign Magnitude Conversion Block

The first module takes in 13 bits input data in two complement representations called D. I need to decide whether the input is a positive or negative representation by extracting the D[12] which is the most significant bit(MSB) and store it to the output variable of sign bit which is variable S. If MSB is 0, then the input is a positive representation and it can be directly assigned to a temporary output variable called "pre_output_place" in this module. If MSB is 1, then the input is a negative representation and I need to check a corner case which is whether the input is equal to the most negative number (smallest number in two's complement). If the input does not equal to [1_0000_0000_0000], then I just negate the entire 13 bits input, add 1 and store the representation to the temporary variable "pre_output_place". If the input is indeed equal to [1_0000_0000_0000] = -4096, then I will set the temporary output variable equal to [0_1111_1111_1111] = +4095 which is the largest positive number in sign magnitude of 13 bits.

The reason for this replacement is the range of numbers that I can represent as 2's complement number in 13 bits is from -4096 to +4095. At the end, the temporary variable that stores the sign magnitude representation will be assigned to the final output variable ("output_place") in this module.

(2) Counting Leading Zeros and Extracting Leading Bits

The second module takes in 13 bits of sign magnitude value as input ("input_encoder") from the first module. I first find the number of leading zeros by checking the each bit/index whether it is equal to one or not starting from the most significant bit (input_encoder[12]) to the least significant bit (input_encoder[0]). If I know there is a 1 appearing at a specific index in the input, then I will know the corresponding number of leading zeros. The checking process is implemented by the conditional operator. For example, input_encoder[12] = 1, then leading zeros = 1; input_encoder[11], then leading zeros = 1; input_encoder[10], then leading zeros = 2; input_encoder[9], the leading zeros = 3, etc. The default case of the total number of leading zeros in the sign magnitude representation is 0. After that, the value will be stored into a 4 bits register that represents the number of leading zeros. Since the number of leading zeros will be encoded into the value of exponent, I need to check whether the number of leading zeros equal or exceed 8 and then assign 8 to the value. It is because any number exceeds 8, the value of the exponent will be encoded to 0. Next, I am encoding the number of leading zeros into the corresponding value of exponent according to Figure 2. For example, number of leading zeros = 1, then exponent = 7; number of leading zeros = 2, then exponent = 6; number of leading zeros = 3, then exponent = 5, etc.

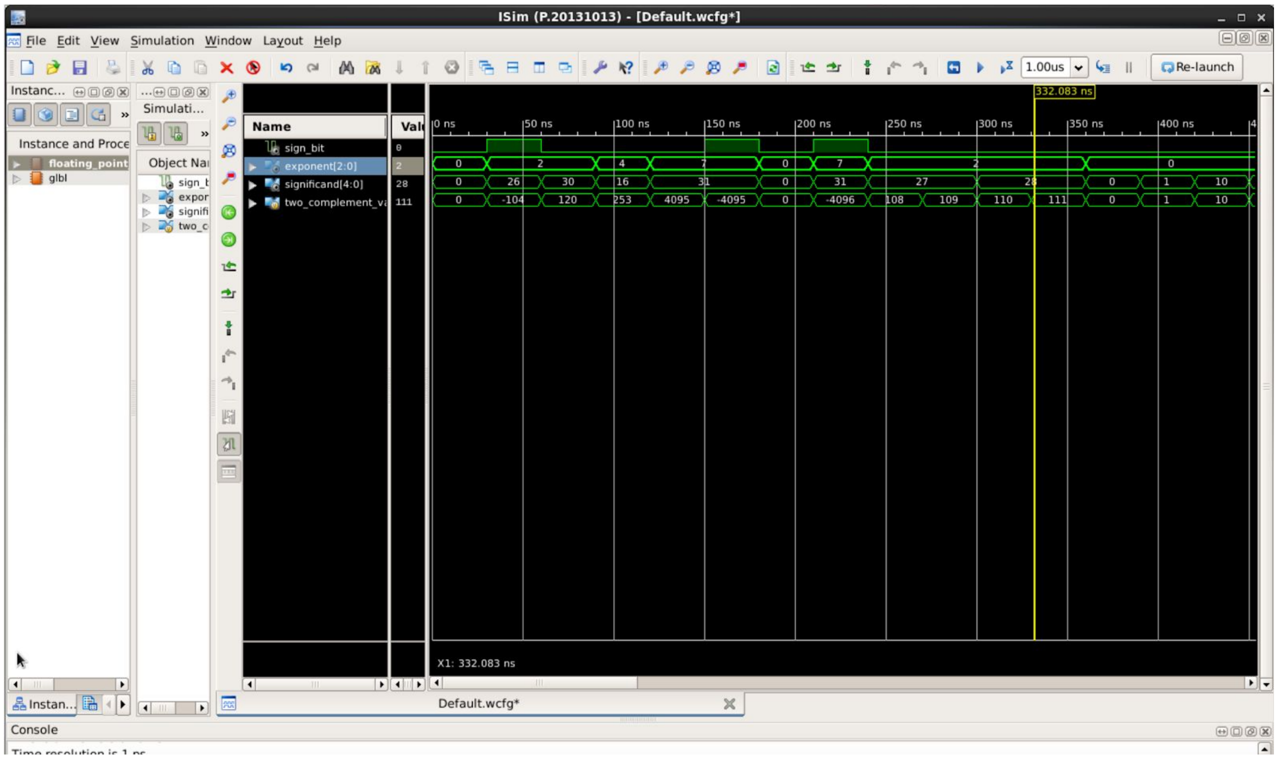
Then, I need to calculate the number of shifting places right after the last leading zero bit (the first bit of the 5 significand bits) in order to extract the 5 significant bits and the following 1 rounding bit. The formula that I have used: the number of times to shift = 13 - (number of leading zeros + 6). After that, I need to check if the number of zeros is not equal to 8 which also means less than 8 because I have already set the maximum number of leading zeros to 8 by the previous checking. If there are less than 8 leading zeros, then I shift the 13 input bits by the calculated shift positions and extract the least 6 significand bits (5 significand bits + 1 rounding bit) to store in the temporary significand variable. If there are exactly 8 leading zeros, I just extract the least 5 significand bits and then add an extra 0 at the end for rounding. Finally, I assign the temporary variable value to the significand (6 bits) and exponent (3 bits) output variable from this module.

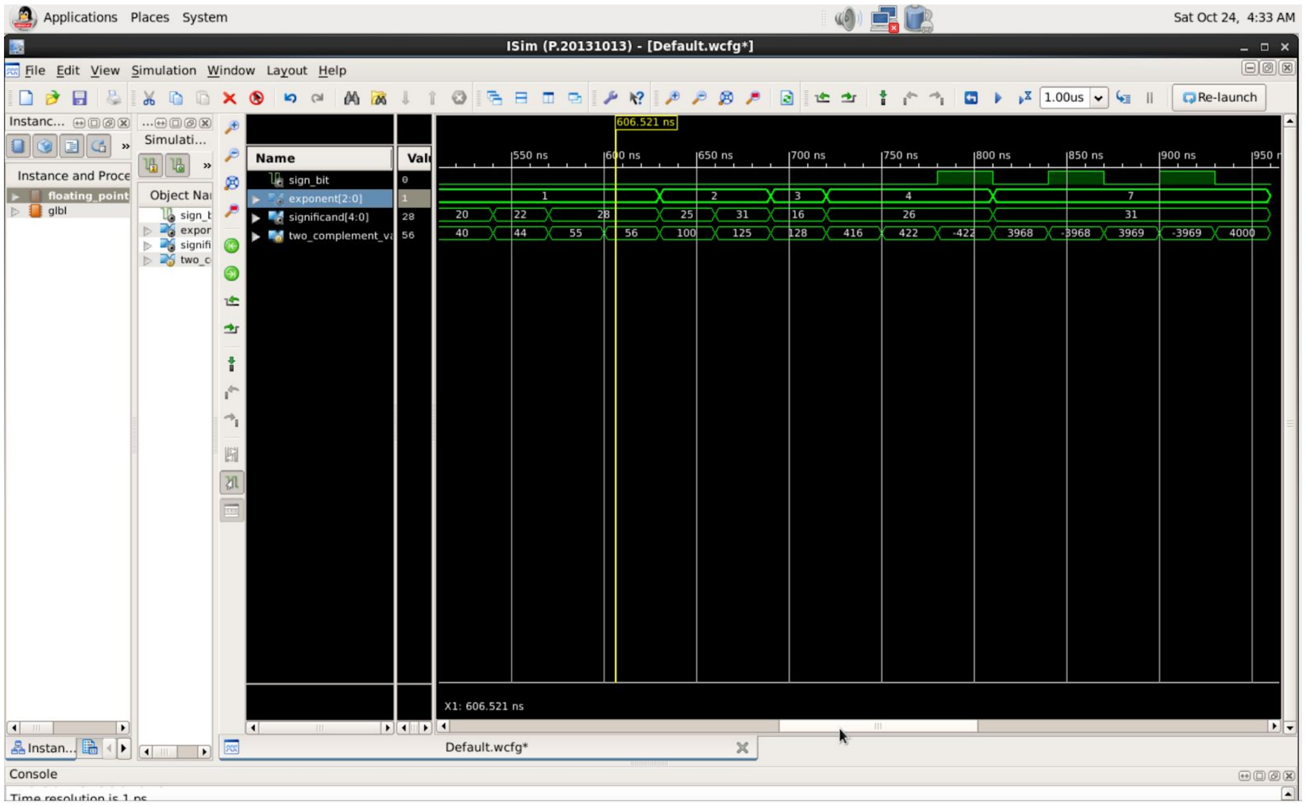
(3) Rounding

The third module takes in 3 bits exponent ("exponent_temp") and 6 bits pre-rounding significand ("significand_temp") as input from the second module. There are a total of 4 possible cases when it comes to rounding. First, I will check the rounding bit which is the least significant bit from the pre-rounding input whether it is 0 or not. (Case 1) If it is equal to 0, then it is a round down case which means the exponent value stays the same and the final 5 bits of significand obtained by extracting the bits starting from index 1 to index 5. If it is not equal to 0 which means 1, then it is a round up case. Then, I will check whether all the bits of the pre-rounding significand are 1 or not. (Case 2) If it is not all equal to 1 which means there exists at least one 0, then extract the 5 significand bits and add 1 to it while the value of the exponent stays the same. If all the bits of the pre-rounding significand are equal to 1, then I have to check whether the temporary exponent variable stores the maximum value which is 7. (Case 3) If the exponent value is 7 already and all the bits of significand are 1 before rounding up, then I can only use the largest possible floating point representation which is [0/1_111_11111]. (Case 4) If the exponent value is less than 7, then I can add 1 to the exponent value and the 5 significand bits becomes [10000]. Finally, I assign the temporary register variable that hold the output to the final output variable for exponent and significand.

2) Simulation Documentation

The goal of the lab is to Xilinx ISE software to design and test a combinational circuit which converts a 13 bits linear encoding of an analog signal into a compounded 9 bit floating point representation. One of the bugs that I have encountered is that my output sign bit shows up as Z (impedance) on the = simulation waveforms when I start running my testbench file. The reason is that I neglect to assign corresponding correct signs to the output variable when I first receive the 13 bits input. The following are the covered test cases and requirements for this lab.


ISim (P.20131013) - [Default.wcfg*]
File Edit View Simulation Window Layout Help
1.00us
Re-launch
Instance and Process
floating_point
glbl
Simulati...
Object Name
Name
Value
sign_bit 0
exponent[2:0] 2
significand[4:0] 28
two_complement_va 111
332.083 ns
0 ns
50 ns
100 ns
150 ns
200 ns
250 ns
300 ns
350 ns
400 ns
exponent: 0 2 4 7 0 7 2 0
significand: 0 26 30 16 31 0 31 27 28 0 1 10
two_complement: 0 -104 120 253 4095 -4095 0 -4096 108 109 110 111 0 1 10
X1: 332.083 ns
Default.wcfg*
Console
Time resolution is 1 ns


Applications Places System
Sat Oct 24, 4:33 AM
ISim (P.20131013) - [Default.wcfg*]
File Edit View Simulation Window Layout Help
1.00us
Re-launch
Instance and Process
floating_point
glbl
Simulati...
Object Name
Name
Value
sign_bit 0
exponent[2:0] 1
significand[4:0] 28
two_complement_va 56
606.521 ns
550 ns
600 ns
650 ns
700 ns
750 ns
800 ns
850 ns
900 ns
950 ns
exponent: 1 2 3 4 7
significand: 20 22 28 25 31 16 26 31
two_complement: 40 44 55 56 100 125 128 416 422 -422 3968 -3968 3969 -3969 4000
X1: 606.521 ns
Default.wcfg*
Console
Time resolution is 1 ns

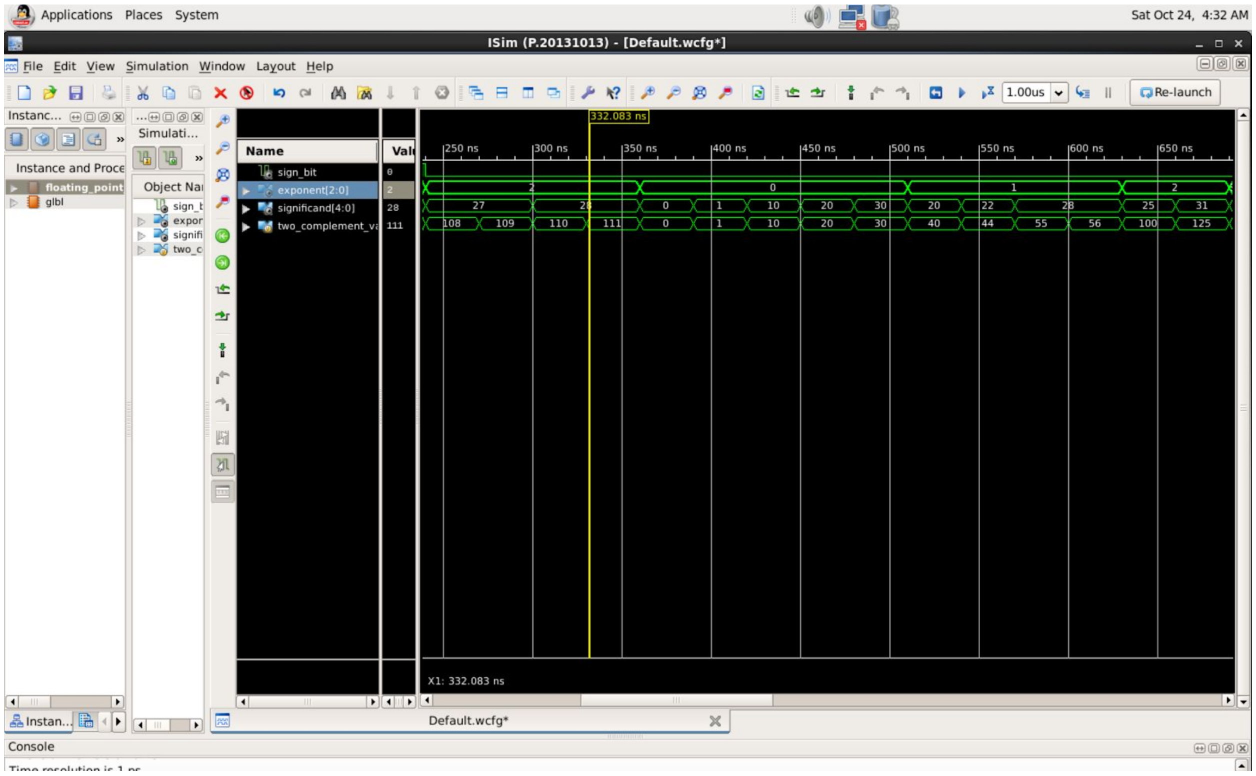


Figure 9: Simulation Waveform Diagram

Considered Test Cases:

1. **Positive input values that could be represent in 9 bits floating representation**

   For example, 13 bits input = [0_0000_0000_1010] = 10 and it will be represented as 9 bits output = [0_000 _01010] after the floating point conversion.

2. **Negative input values that could be represent in 9 bits floating representation**

   For example, 13 bits input = [1_1111_1001_1000] = -104, after complement increment, it becomes [0_0000_0110_1000], then after rounding stage, significand = 26 and exponent = 2, final output = [1_010_11010] = -26 * (2^2) = -104

3. **Positive input values that could be represent in closest 9 bits floating representation**

   For example, 13 bits input = [0_0000_0111_1101] = 125, after conversion, it becomes [0_010_11111] = 124, significand = 31 and exponent = 2, final output = 31 * (2^2) = 124

4. **Negative input values that could be represent in closest 9 bits floating representation**

   For example, 13 bits input = [1_1110_0101_1010] = -422, after complement increment, it becomes [0_0001_1010_0110], then after rounding stage, significand = 26 and exponent = 4, final output = [1_100_11010] = -26 * (2^4) = -422

5. **Round Down Case (normal)**

   For example, 13 bits input = [0_0000_0110_1100] = 108, the pre-significand (5 bits significand + 1 rounding bit) = [11011_0], after round down and conversion, significand = 27 and exponent = 2, final output = [0_010_11011] = 108

6. **Round Up Case (normal)**

   For example, 13 bits input = [0_0000_0110_1110] = 110, the pre-significand (5 bits significand + 1 rounding bit) = [11011_1], after round up and conversion, significand = 28 and exponent = 2, final output = [0_010_11100] = 112

7. **Round Up Case (significand overflow & exponent < 7)**

   For example, 13 bits input = [0_0000_1111_1101] = 253, the pre-significand (5 bits significand + 1 rounding bit) = [11111_1], after round up and conversion, significand = 16 and exponent = 4 , final output = [0_100_10000] = 256

8. **Round Up Case (significand overflow, exponent = 7) / Largest possible values**

   For example, 13 bits input = [0_1111_1111_1111] = 4095, the pre-significand (5 bits significand + 1 rounding bit) = [11111_1], after round up and conversion, significand = 31 and exponent = 7, final output = [0_111_11111] = 3968

9. **Smallest possible values**

   For example, 13 bits input = [1_0000_0000_0000] = -4096, after converting to sign magnitude, it becomes [0_1111_1111_1111], then after rounding stage, significand = 31 and exponent = 7, final output = [1_111_11111] = -31 * (2^7) = -3968

3) Conclusion

After I have finished this lab, I understand how to use Verilog and Xilinx ISE program to design and test a combinational circuit that does floating point conversion. I get to have hands-on experience to understand the way the underlying hardware modules connect with each other on certain calculation and design. One of the difficulties that I have encountered is to identify the corner cases that I need to consider when I am designing the circuit. Second, it takes a while to get adjusted to change my throughout process in terms of hardware and bits when I am writing Verilog.