Lab 2: Floating Point Conversion

CS M152A Fall 2020, Majid Sarrafzadeh

Lab 6: Rajas Mhaskar

Kenta Hayakawa

10/24/2020

Introduction and Requirement

The purpose of this lab was to use the Xilinx ISE software to create, simulate, and test a combinational circuit that takes in a 13-bit linear encoding of an analog signal, and converts it into the nearest compounded 9-bit floating point representation. For this lab, we used a simplified floating point representation, which consisted of a sign bit, a 3-bit exponent field, and a 5-bit significand field.

The following is the format we want our final floating point representation output to be in:

8	7	6	5	4	3	2	1	0
Sign	Exponent			Significand				

The value represented by the above format can be calculated as follows:

$$V = (-1)^{Sign} * 2^{Exponent} * Significand$$

The sign bit represents the sign of the number (0 for positive, 1 for negative). The 3-bit exponent ranges from values of 000=0 to 111=7, and the 5-bit significand ranges from values of 00000=0 to 11111=31. These ranges will become important when considering corner cases.

The number of leading zeros of the original 13-bit linear encoding input can be used to determine what the value of the 3-bit exponent field of the final floating point representation will be for most of the possible inputs. The following table displays this relationship:

Leading zeros of input	Exponent value
1	7
2	6
3	5
4	4
5	3
6	2
7	1
≥8	0

In case of a negative input number, the number should be negated first (flip all bits) then added 1 to put in positive two's complement form. Then count the leading zeros (the smallest possible 13-bit value, -2^{12}, however, is the only exception of all the possible negative numbers for a 13-bit encoding, and a special case had to be created).

The above rule, however, does not guarantee the most accurate floating point representation. Therefore, the combinational circuit must additionally round the input to the nearest floating point encoding. For this 13-bit input, the 5 bits following the last leading 0 is the significand, and bit following the last bit of the significand determines

whether to round up or down. If the bit is 0, the 5 bits of the significand are simply used as the significand for the final floating point format (rounding down). If the bit is 1, the 5-bit value obtained after adding 1 to the significand will be used as the significand for the final floating point format.

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

In the case of the significand value 11111, however, adding 1 will result in a 6-bit significand, so to compensate the 6-bit value must be shifted right one, and then the exponent value must be increased by 1. If the exponent value will exceed the max possible value after the exponent increase, simply return the largest possible floating point representation as the final output.

Design Description

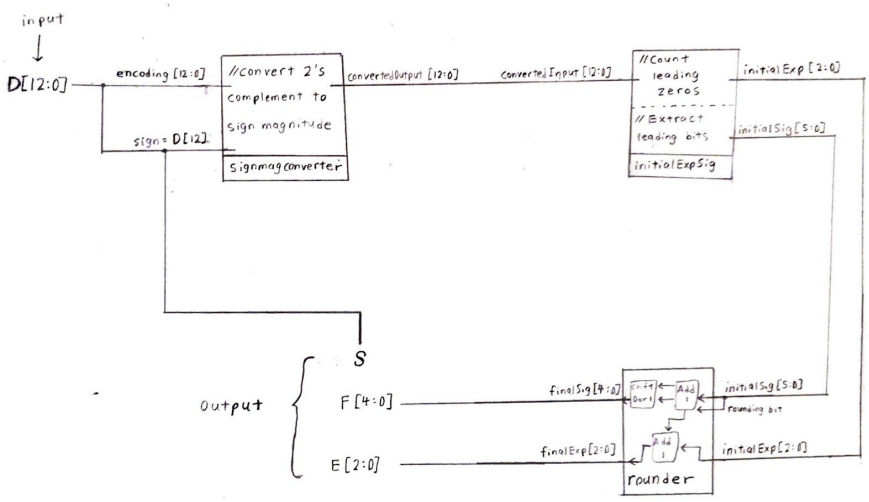
The basic design of the combinational circuit takes in a 13-bit two's complement input, denoted D[12:0], and uses three submodules to convert the input into a 9-bit floating point representation. The circuit has three final outputs:

-The sign bit, which is just the sign bit of the original input that is used as both the input to the first module and as one of the three final outputs of the whole circuit.

-The significand value, generated from the three submodules

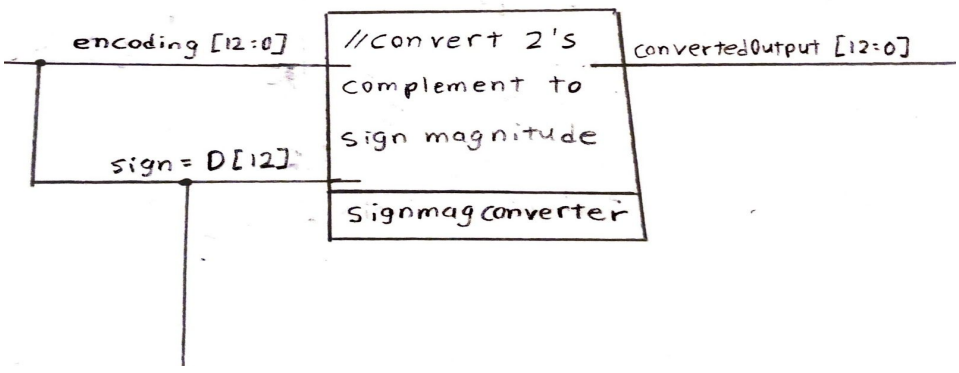
-The exponent value, generated from the three submodules

The schematic for this entire FPCVT module is:



The first module is a sign magnitude converter, which takes in the 13-bit input and converts it to sign magnitude form. First, the module checks the sign bit

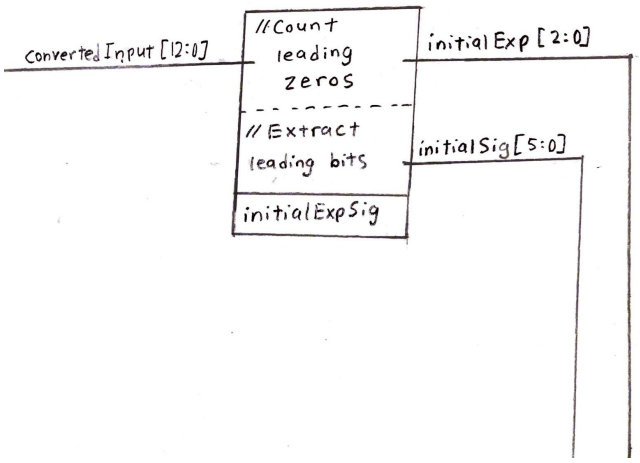
of the input (sign bit had already been assigned as the first bit of the 13-bit input in the top module), and if it is 0, simply keep the encoding and pass the value onto the next module, since leading zeros can already be counted. If it is 1, simply negate the value and add 1 to get the two's complement of that value. Now, leading zeros can be counted for that value. As previously mentioned, the smallest possible 13-bit value, -2^{12}, requires a special case to be created, since its two's complement form is just itself. For this, the converter simply uses the negated form of the value as the output, without the addition of 1. The converted output is sent to the next module, which counts the leading zeros obtained from this step, and extracts leading bits for future rounding purposes. The schematic for this converter is:



The second module is a module which takes in the converted sign magnitude form as input, and performs two main functions:

1) Counts the leading zeros and outputs an initial exponent value prior to rounding.
2) Finds an initial significand value prior to rounding, by using the number of leading zeros to identify the location of the significand in the input, and uses it as output.

The first function is done by going through the upper 8 bits of the input, and counting how many bits are zeros. Using this information and the relationship between leading zeros and floating point exponent values, an initial exponent value for the final floating point representation will be generated and used as one of the outputs of the module. For the second function, if there are less than 8 bits of leading zeros, simply shift the input by 13 - (number of zeros) + 6, and use the last 6 bits of the result as the initial significand to be passed as the other output into the final module. If there are greater than or equal to 8 bits of zeros, simply use the lower 5 bits of the input, and append another 0 to the least significant bit position to make it a 6 bit initial significand output (required because if the number of leading zeros is at least 8, that means the bit used to determine rounding will not be incorporated properly). The schematic for this module is given as follows:



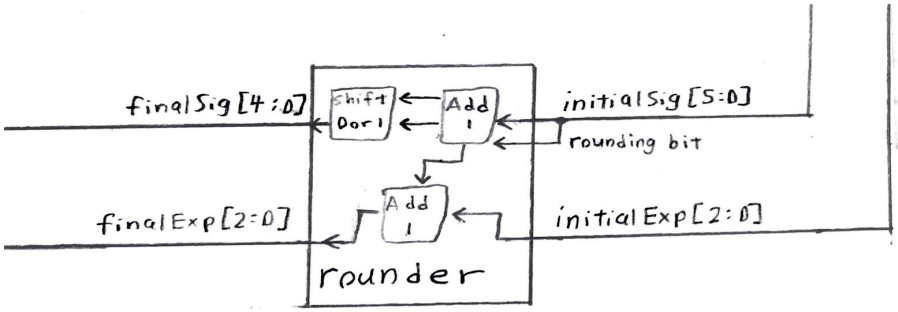
The third and final module takes in the initial exponent and initial significand values (prior to rounding) found in the previous module as input, and performs rounding to the nearest floating point encoding. The output of this module are the final exponent and final significand values to be used as part of the final floating point representation. If the least significant bit (or the rounding bit, in this case) of the initial significand value is 0, simply pass both inputs directly as the outputs, since rounding down means simply truncating the bits after the 5-bit significand without any alterations. If the least significant bit is 1, add 1 to the upper 5 bits of the initial significand, and use that value as the final significand output. The initial exponent will be outputted directly as the final exponent. Two special cases to consider are when:

1) All initial significand bits are 1s. In this case, adding 1 will create a 6-bit final significand, which is incompatible. To solve this, truncate a 0 from the

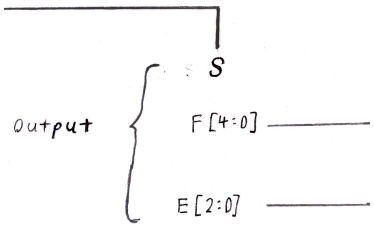
value to make it 5 bits again, but increase the exponent value by 1 to compensate for it.

2) If the above operation causes an exponent value to go above the max possible exponent value, use the largest possible floating point representation as the final significand, and directly output the initial exponent value as the final exponent value.

This is the schematic for this final rounding module:

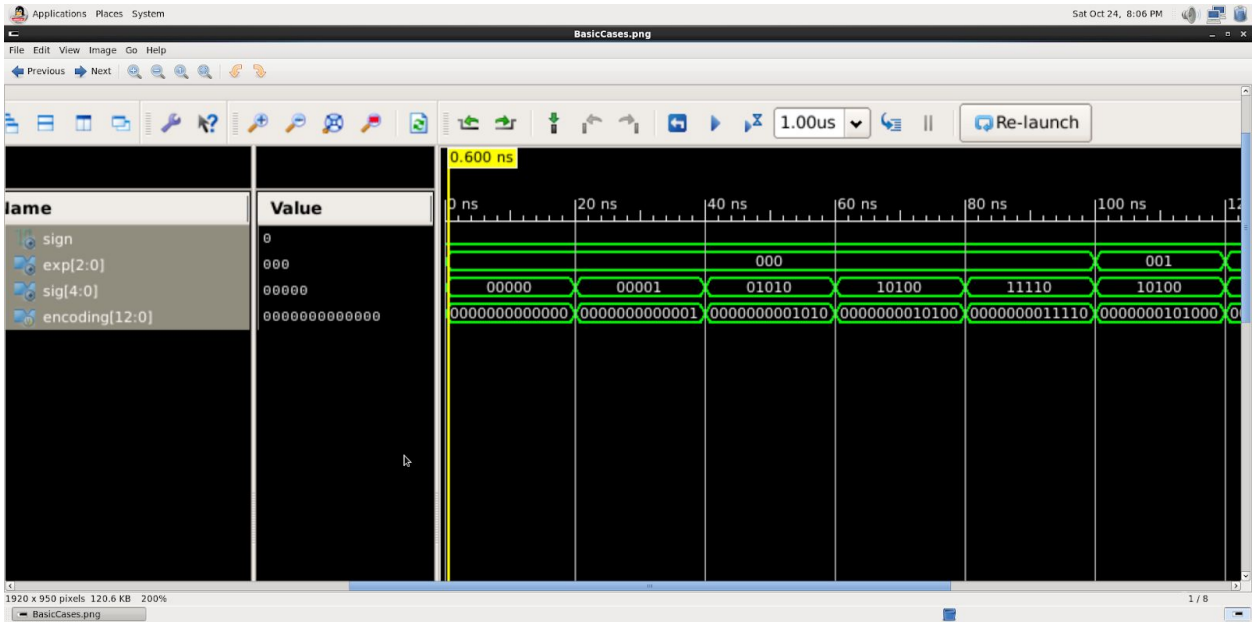


The final output should be in this format:



Simulation Documentation

For this lab, a testbench was drawn up from scratch, and this testbench was used to simulate the outputs generated from specific test cases. There were not an extensive amount of edge cases, but for those edge cases, special code was written. Since all three submodules had to be defined under the FPCVT.v top module, I simply performed tests on this one module. The linear encoding starts off with basic cases that can be converted without any special attention (values such as 1, 10, 20, 30, 0). The figure below shows the first few encodings simulated. The encoding of the waveform is the input, and the other three values: sign, exp, and sig, are outputs:

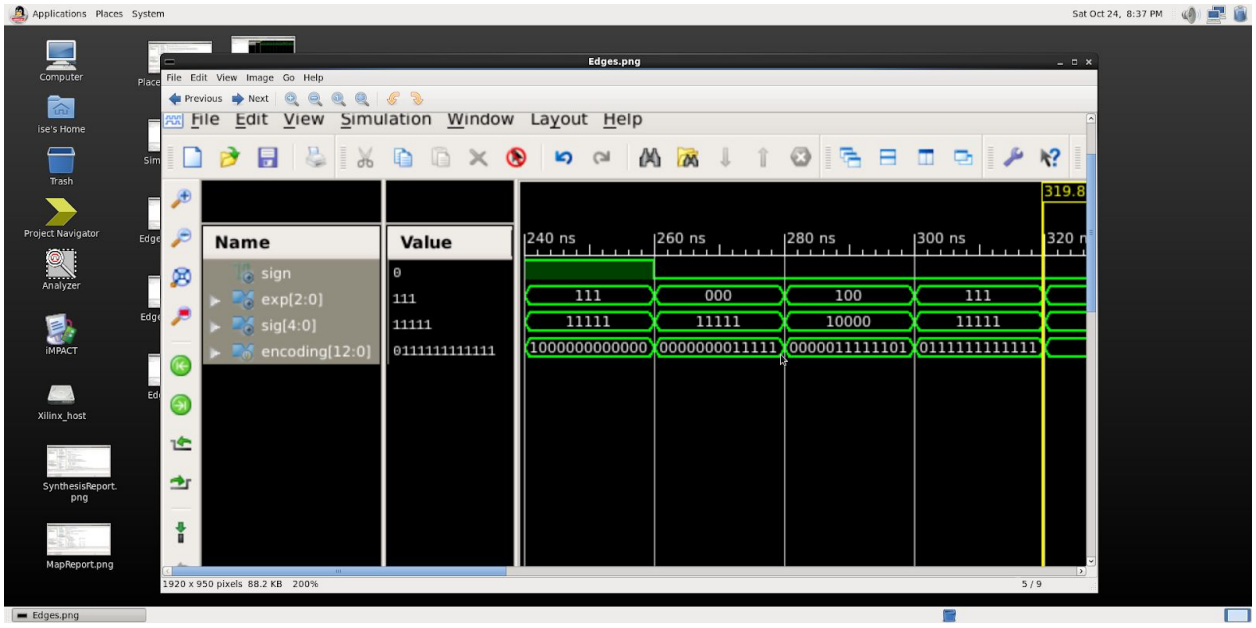


There are four test encodings of the testbench that are important edge cases to consider:

1) The encoding from 240 ns to 260 ns in the figure below shows the first edge case, when the input is the smallest negative number. Since the value's complement cannot be represented within 13 bits, we will just return a negative floating point number with the highest magnitude.
2) The encoding from 260 ns to 280 ns shows the second edge case, when there are ≥8 leading zeros. Here, a case is created where the output will be the lowest 5 bits of the encoding with an exponent of 0.
3) The encoding from 280 ns to 300 ns shows the third edge case, where an encoding with all 1s as the significand is rounded up. Rounding up will require the significand to add 1, which will cause an overflow because the significand will have 6 bits rather than 5. Therefore, the output will be formatted so that the LSB of the 6-bit significand will be truncated to make it 5 bits again, and the exponent value will be reduce by 1 to compensate for the truncation.
4) The last edge case from 300 ns to 320 ns shows when a very large encoding value is rounded up. Since the exponent may go beyond the max value of 7, and go beyond the 9-bit floating point capacity, the

output will need to be the largest possible floating point representation, since that is the closest a 9-bit floating point can get.

The figure below shows all the edge cases' simulations:



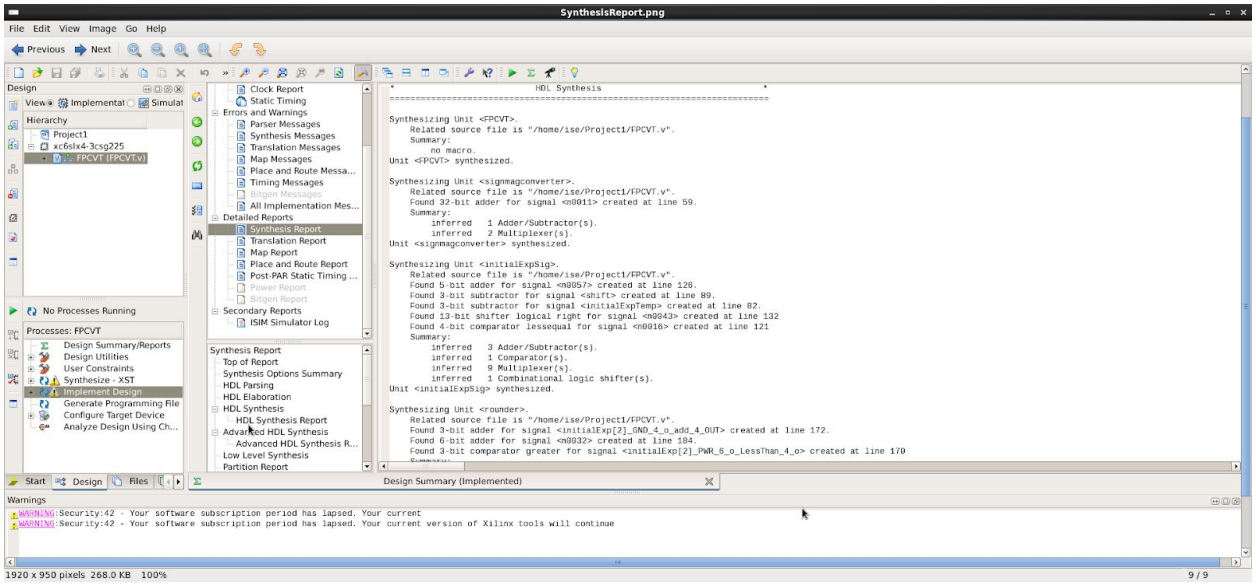
Notes: 240 ns - 260 ns = first edge case

260 ns - 280 ns = second edge case

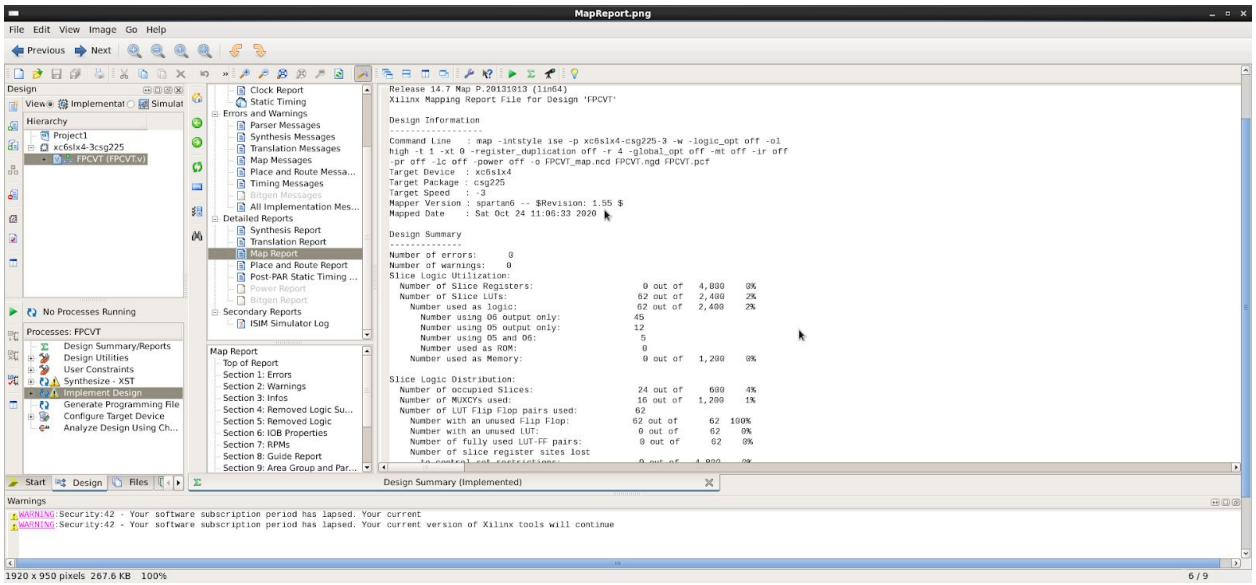
280 ns - 300 ns = third edge case

300 ns - 320 ns = fourth edge case

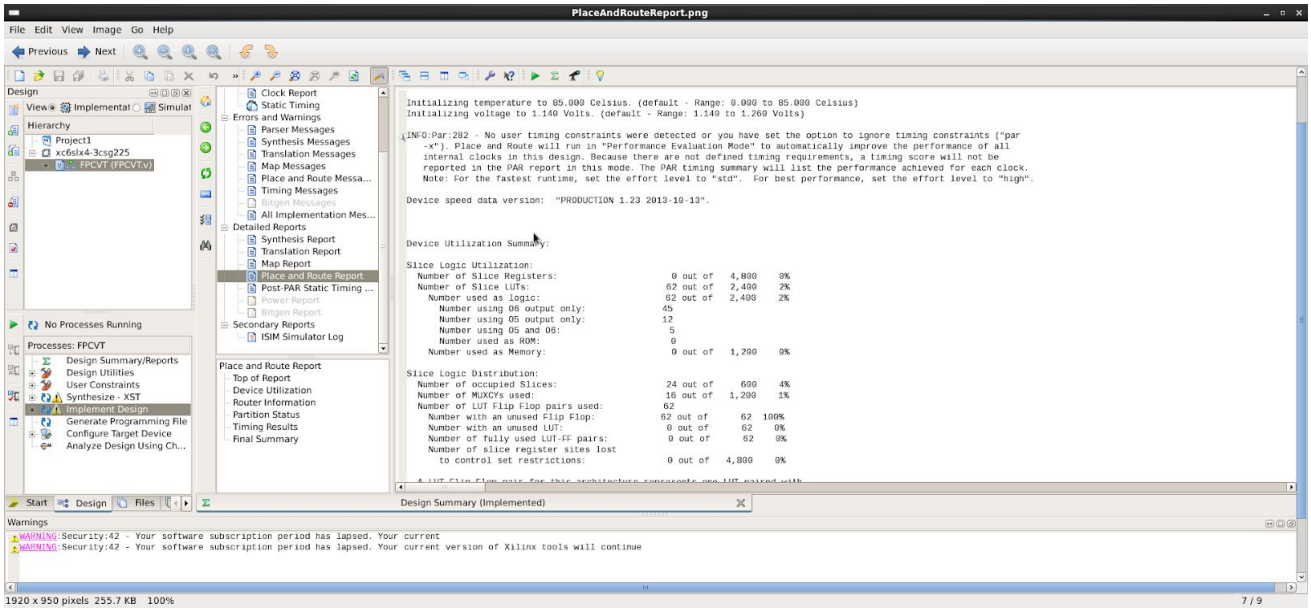
Simulation Report:



Map Report:



Place and Route Report:



Conclusion

The lab gave me first hands-on experience with Verilog coding and Xilinx simulations. We designed a floating point converter which used three submodules to process the actual conversion, and a top module to bring them all together. The final code reads in a 13-bit linear encoding in two's complement form, and outputs a converted, 9-bit floating point representation of that encoding.

The lab was simple enough to fully get comfortable with coding in Verilog. There were no excessive amounts of edge cases, so writing the testbench did not take too long. A lot of the difficulties came from the operation of the Xilinx ISE, such as learning how to perform simulations properly, or what buttons to click to synthesize, check behavioral syntax, etc. However, by the end of the project, I have gotten quite used to it. The error messages I got were usually syntax errors, so they were easy to debug.