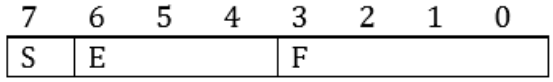
Lab 1 Report

Dennis Zang, 704766877
Partner: Hirday Gupta
Lab Section 5, October 14, 2019

Problem Statement:

The task at hand that was assigned is to use Verilog to design and model a combinational circuit that converts a 12-bit 2's-complement little-endian integer into a compounded 8-bit floating point representation, a variant of the standard binary32 IEEE format. Specifically, given a 12-bit integer a = -2^{11}(a[11]) + 2^{10}(a[10]) + 2^9(a[9]) + 2^8(a[8]) + 2^7(a[7]) + 2^6(a[6]) + 2^5(a[5]) + 2^4(a[4]) + 2^3(a[3]) + 2^2(a[2]) + 2(a[1]) + (a[0])), convert it into an 8-bit floating point representation FP such that S = FP[7] is the signed bit of FP, E[2:0] = FP[6:4] is the 3-bit exponent, and F[3:0] = FP[3:0] is the 4-bit significand. A visual representation of this is given below.

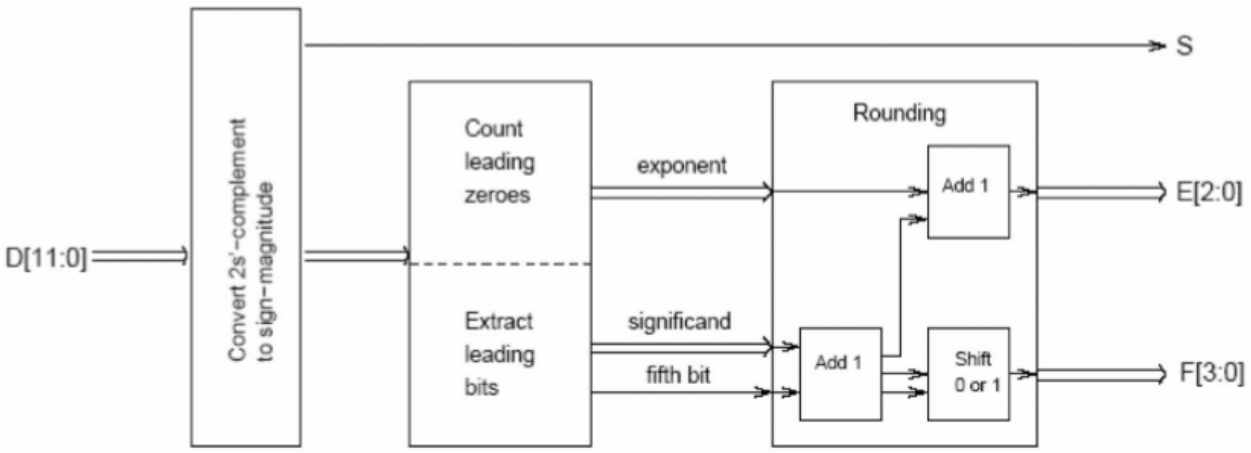
(8-bit Floating Point Representation)



Implementation/Algorithm:

For the design of our combinational circuit, we followed the proposed design given in the lab specifications. The flowchart of the individual modules are given below.

(Flowchart)



As we can see, the top-level module that does the overall conversion can be broken up into 3 separate submodules: one for getting the signed-bit S and the unsigned magnitude (absolute value); one for separating the unsigned magnitude into the exponent, significand, and the rounding bit; and one for performing the actual rounding with the rounding bit on the significand and the exponent. In our implementation, these submodules are named "SignedMagnitude", "Separator", and "Rounder" respectively, and are called by the top-level module "Convertor". The implementations for each are described below.

(SignedMagnitude)

For submodule “SignedMagnitude”, the input would be a 12-bit linear encoded integer “d”, and the outputs would be a signed bit “S” and an 11-bit linear integer encoded integer “m” representing the absolute value of d.

To calculate “S”, we would simply check if the most significant bit of “d” is 1. If it is, “S” = 1, else “S” = 0.

To calculate “m”, we simply need to check the number is negative. If the most significant bit is not one, then we can simply return the remaining 11 bits as the magnitude as positive numbers must be less than $2^{11} = 2048$. If this is not the case, then to get the magnitude of a negative number represented in 2’s complement, we take the bitwise-NOT of remaining 11 bits and add 1 (or ~d[10:0] + 1). However, one edge case we have to test for here is when “d” = -2048 = 12’b100000000000, because ~d[10:0] + 1 here would be equivalent to (11111111111) + 1 = 11’b00000000000, an integer overflow. So, for the second case for negative numbers, we need to check that d[10:0] != 11’b00000000000.

The input “d” would be the actual number to be converted into a float, “D” (see previous figure). The output “S” would become the signed bit of the final converted float, and the output magnitude “m” would be the input to the next submodule, “Separator”.

(Separator)

For submodule “Separator”, the input would be an 11-bit magnitude “m”, and the outputs would be the 3-bit exponent “exp”, the 4-bit significand “sig”, and an extra rounding bit “round” for the next submodule.

To calculate “exp”, we simply count the number of leading 0’s before the 1st bit that is on. Considering that the maximum value of “exp” is 7, “exp” would return 7 – (number of leading 0’s) if it evaluates to a non-negative value, otherwise “exp” returns 0. When counting the number of leading 0’s, we also need to get the position of the 1st set bit to calculate “sig” and “round” later.

To calculate “sig” and “round”, starting from the 1st set bit located from the previous part, we would load the next 4 bits into “sig” starting from the most significant bit to least significant bit. If the loading would go out of range of the 11 bits, then we would need to check how many bits there are left to load in “sig”, and left-shift the current buffered value of “sig” by that many positions. As for “exp”, we would return the 5th bit after the 1st set bit if it is still in range, otherwise we return 0.

The three outputs “exp”, “sig”, and “round” would be passed to the submodule “Rounder” to perform the necessary rounding operation and calculate the final values of “E” and “F”.

(Rounder)

For submodule “Rounder”, the input would be a 3-bit exponent “exp”, a 4-bit significand “sig”, and a rounding bit “round”. The outputs would be the 3-bit exponent “E” and the 4-bit significand “F” for the final floating point representation.

Rounding would depend if the “round” bit is set or not, s and if it is not set, we would simply have “E” return “exp” and “F” return “sig” as is.

If the “round” bit is set, then we can simply increment the current value of “sig”. However, an edge case to consider is when “sig” = 4’b1111. If we increment this, integer

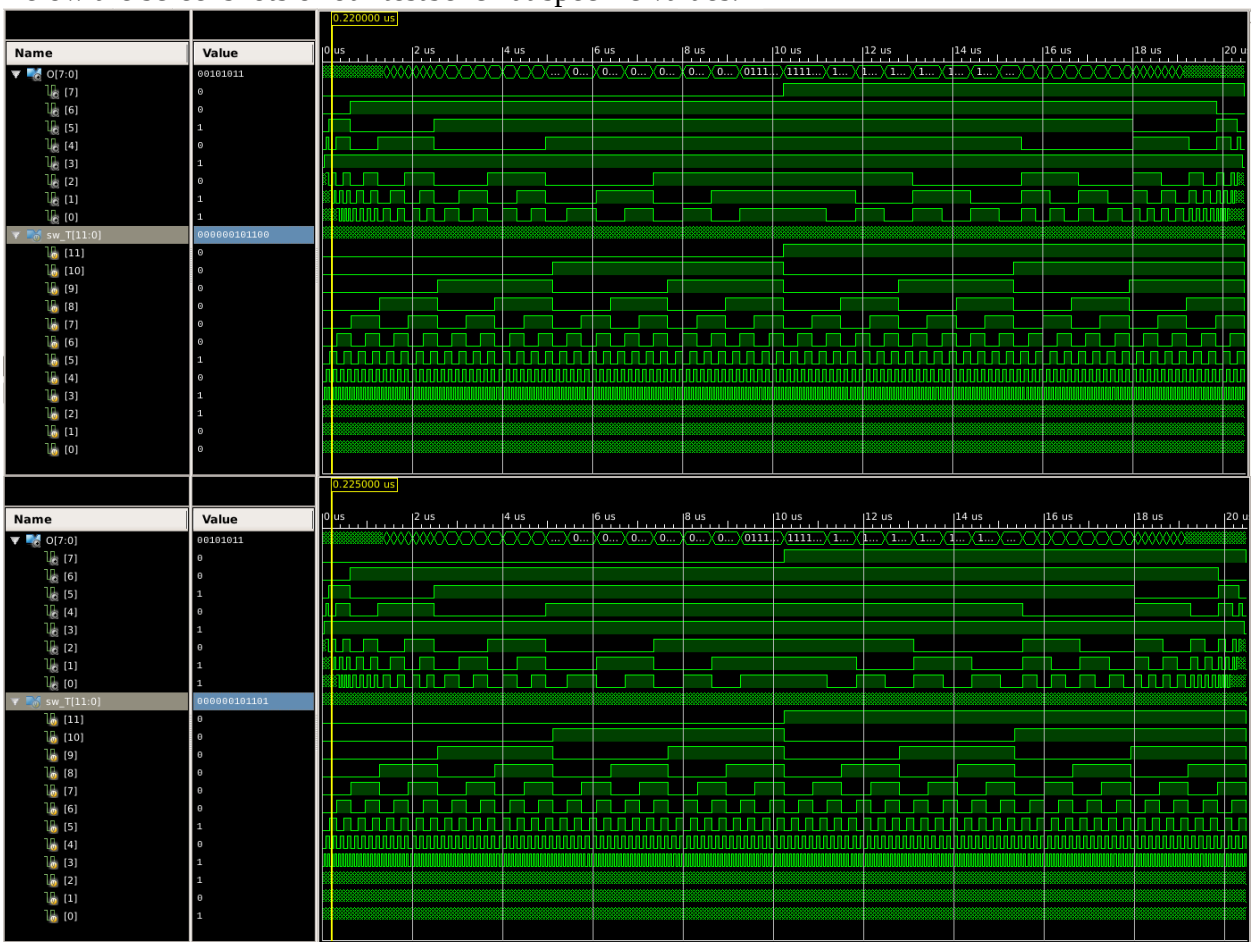
overflow would occur, resulting in 4’b0000. So, we check that “sig” != 4’b1111 before incrementing and having E return “exp” and “F” return “sig”.

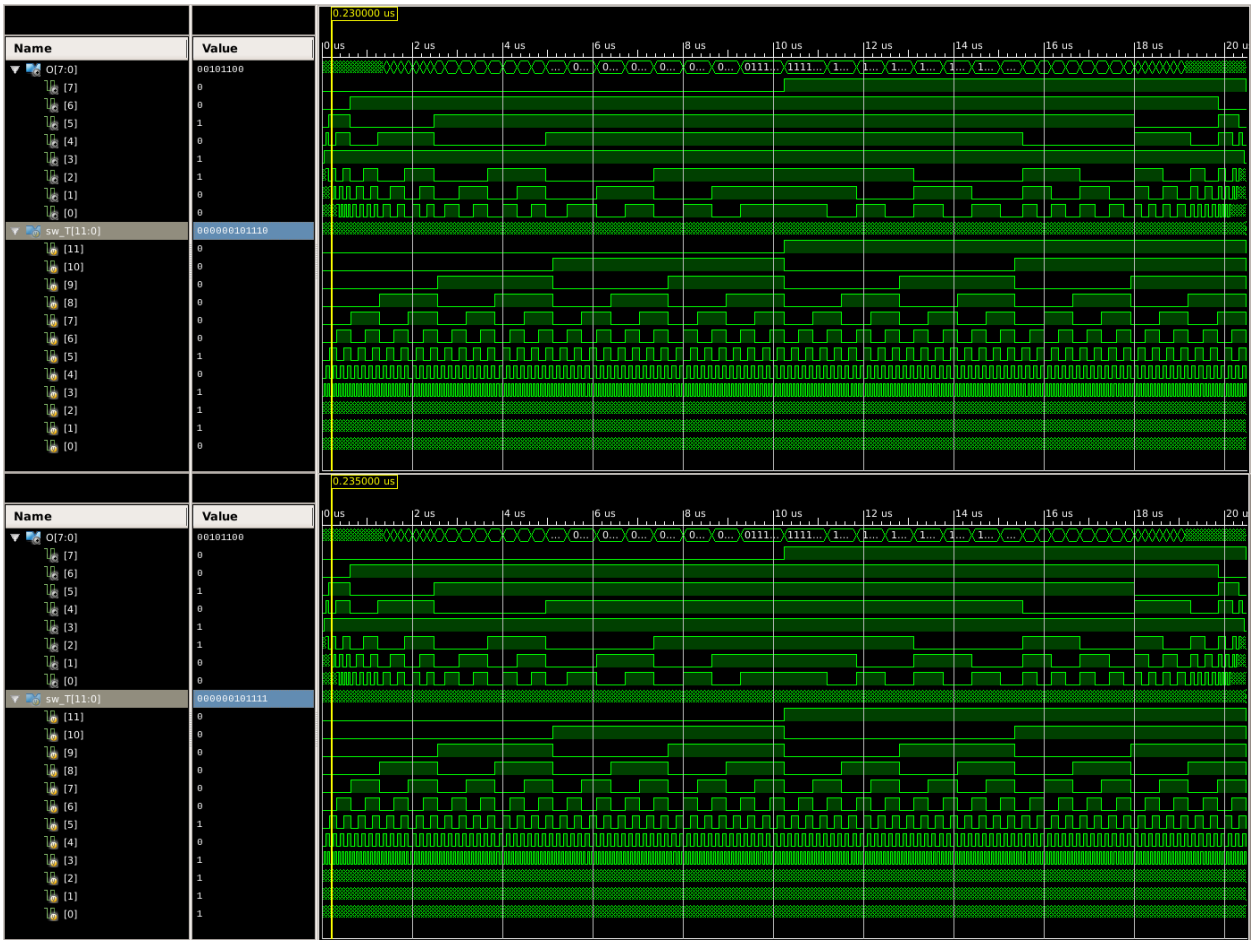
However, there is yet another edge case to consider: if we were to increment “exp” when “exp” = 3’b111, then we would habe 3’b000, another integer overflow. So, for the second case above, we run a second check to avoid this: if “sig” = 4’b1111 and “exp” != 3’b111, and we set “F” to 4’b1000, increment “exp”, and have “F” return “sig”. There is no complementary case for when “exp” = 3’b111, because there is nothing we can do when the maximum floating point is set.

Simulation Screenshots:

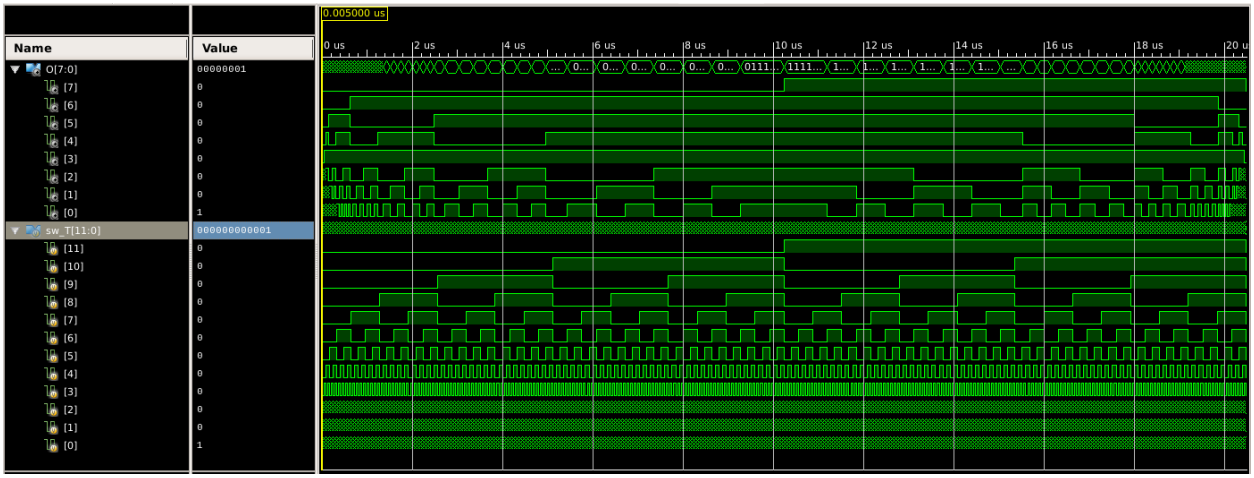
For our testbench, we simply wrote a simple program that would input 0 into “Convertor”, and increment the input “D” every 5 nanoseconds until integer overflow happens, going 1 full cycle before ending.

Below are screenshots of our testbench at specific values.

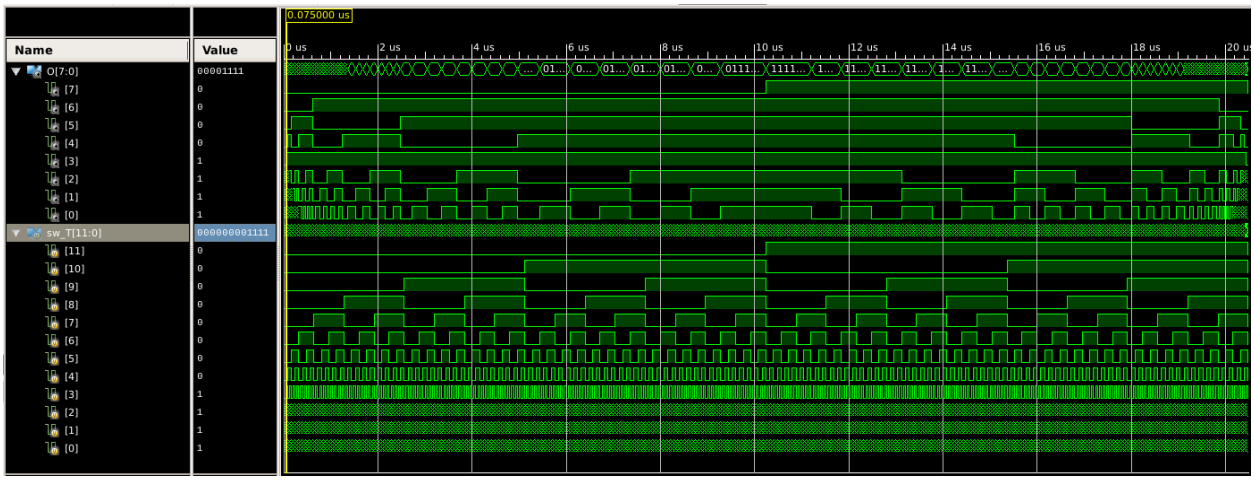




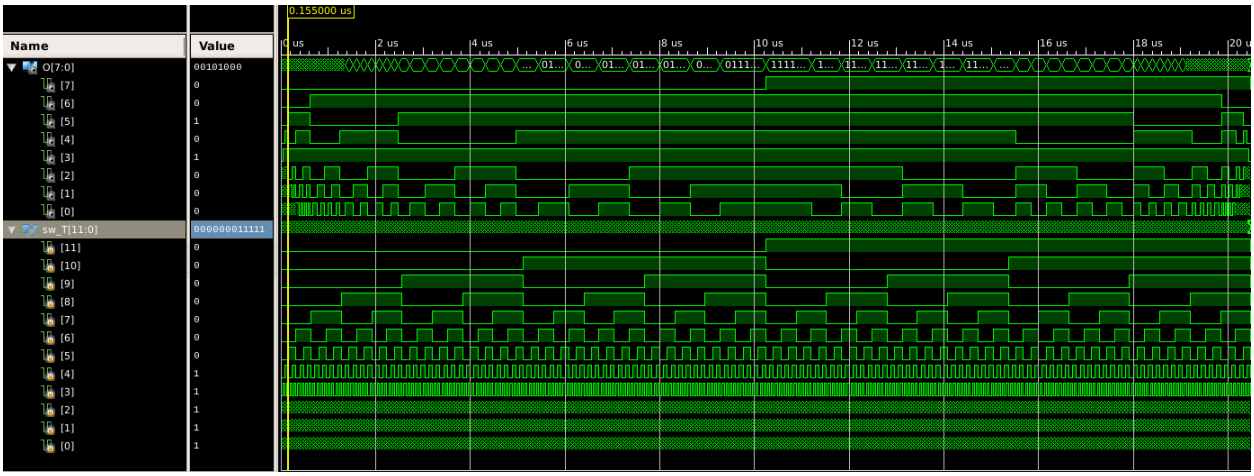
These 4 input/output pairs are the given examples in the specifications. These are as follows:
12’b000000101100 → 8’b00101011
12’b 000000101101 → 8’b 00101011
12’b 000000101110 → 8’b 00101011
12’b 000000101111 → 8’b 00101011



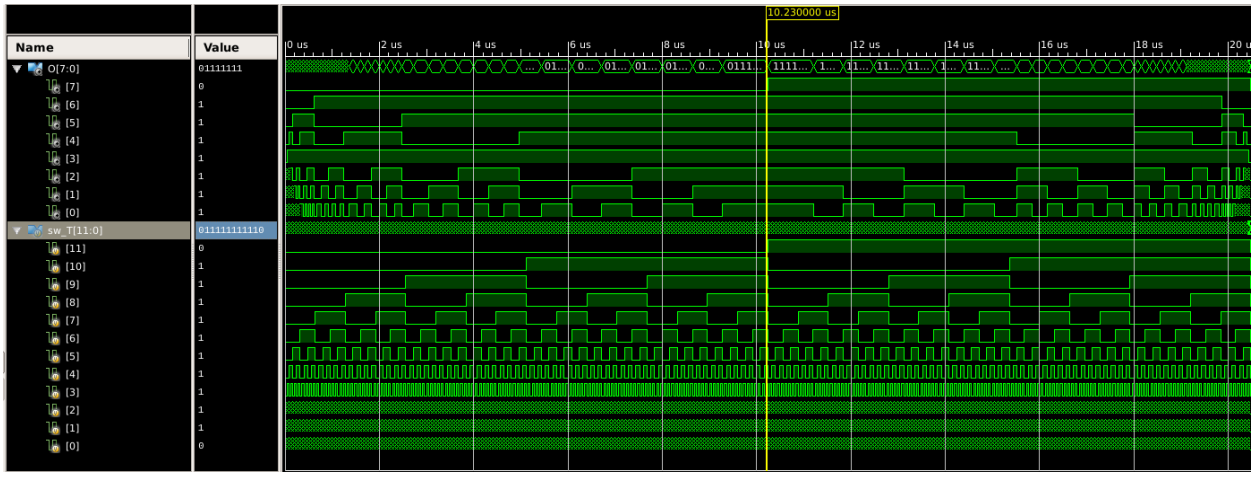
Here is the case when input “D” is equal to 12’b000000000001. The output is 8’b00000001.



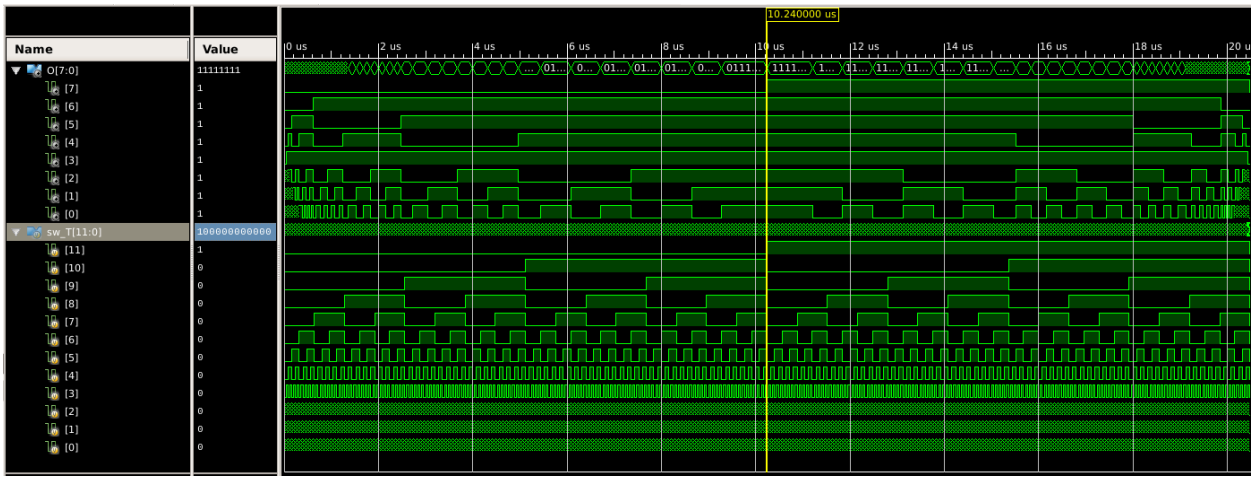
Here is the case when input “D” is equal to 12’b000000001111. The output is 8’b00001111.



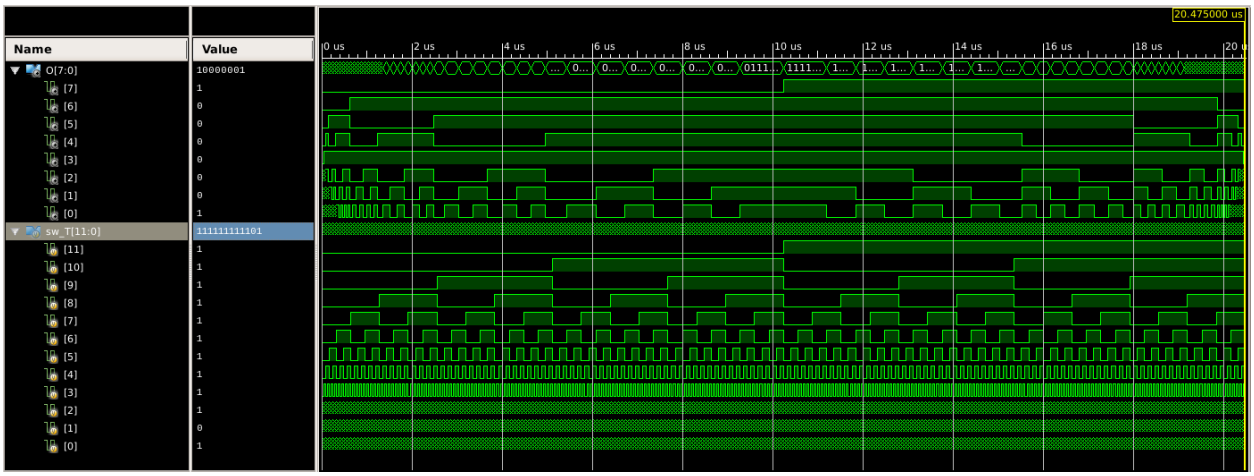
Here is the case when input “D” is equal to 12’b000000011111. The output is 8’b00101000.



Here is the case when input “D” is equal to 12’b011111111110. The output is 8’b01111111. (Note: There is a bug with the simulation in which we cannot get “D” = 12’b011111111111 directly. However, if we are to extend the simulation beyond the $finish, then we are able to access 12’b011111111111, and verify that the output is indeed 8’b01111111 as well.)



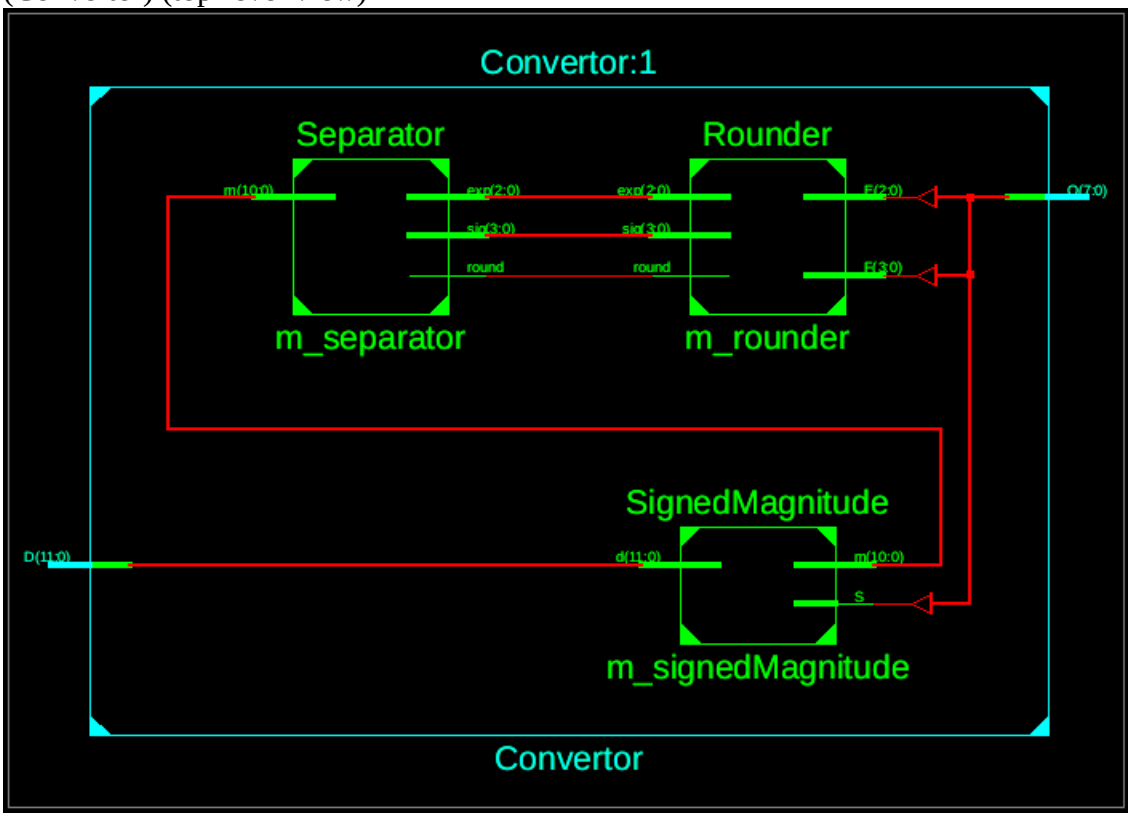
Here is the case when input “D” is equal to 12’b100000000000. The output is 8’b11111111.



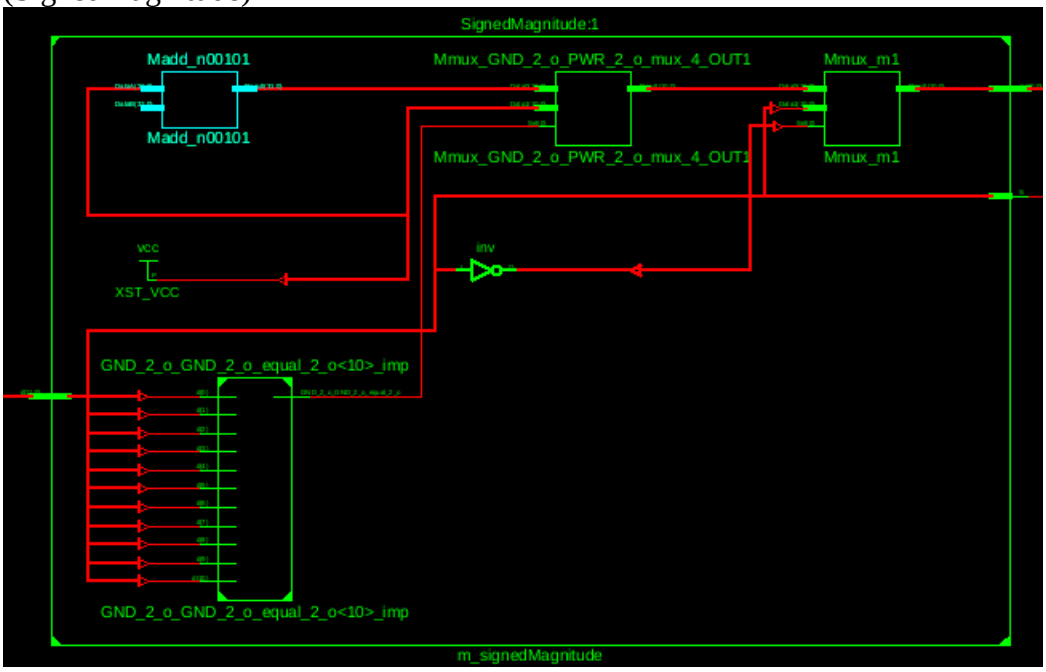
Here is the case when input “D” is equal to 12’b111111111101. The output is 8’b10000001. (Note: There is a bug with the simulation in which we cannot get “D” = 12’b111111111111 directly. However, if we are to extend the simulation beyond the $finish, then we are able to access 12’b111111111111, and verify that the output is indeed 8’b10000001 as well.)

Netlists:
Below are the results when we try to synthesize the schematics from the Verilog code.

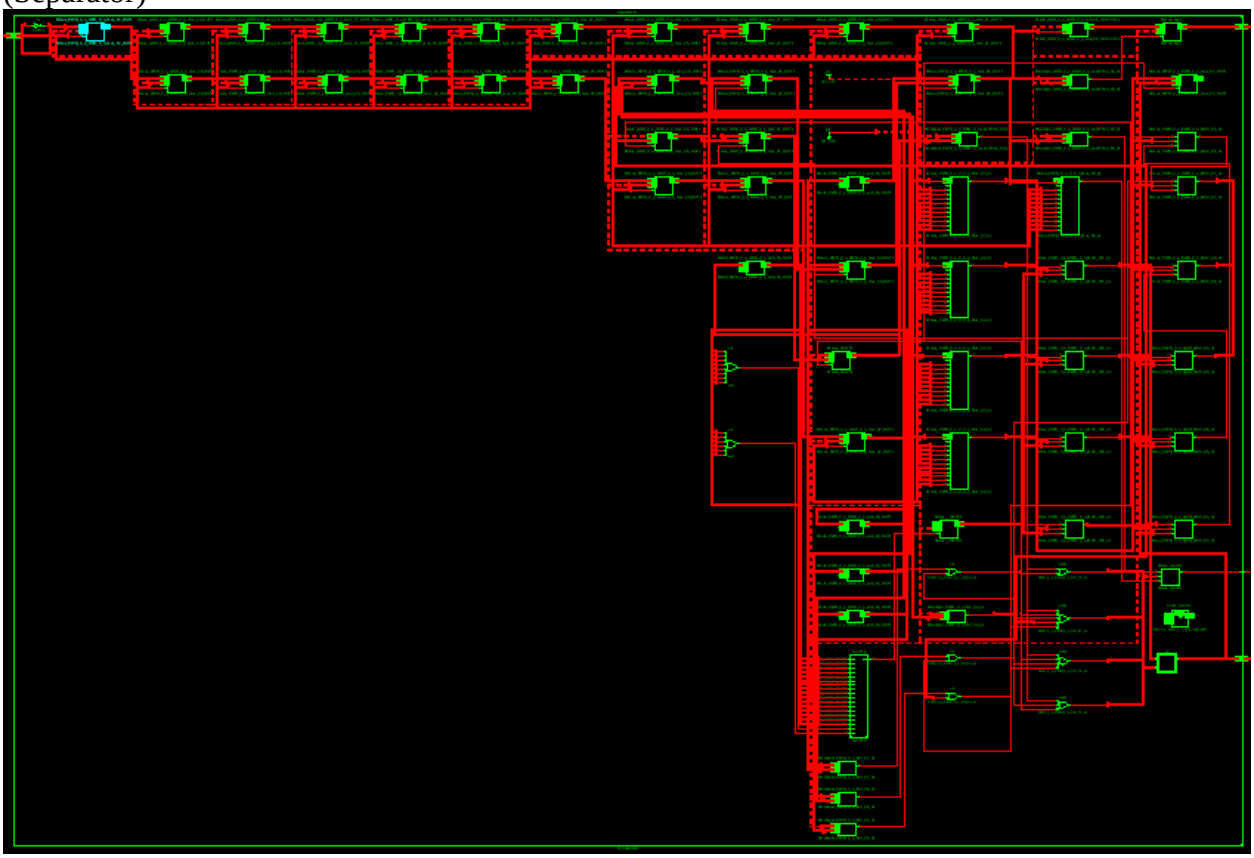
(Convertor) (top-level view)



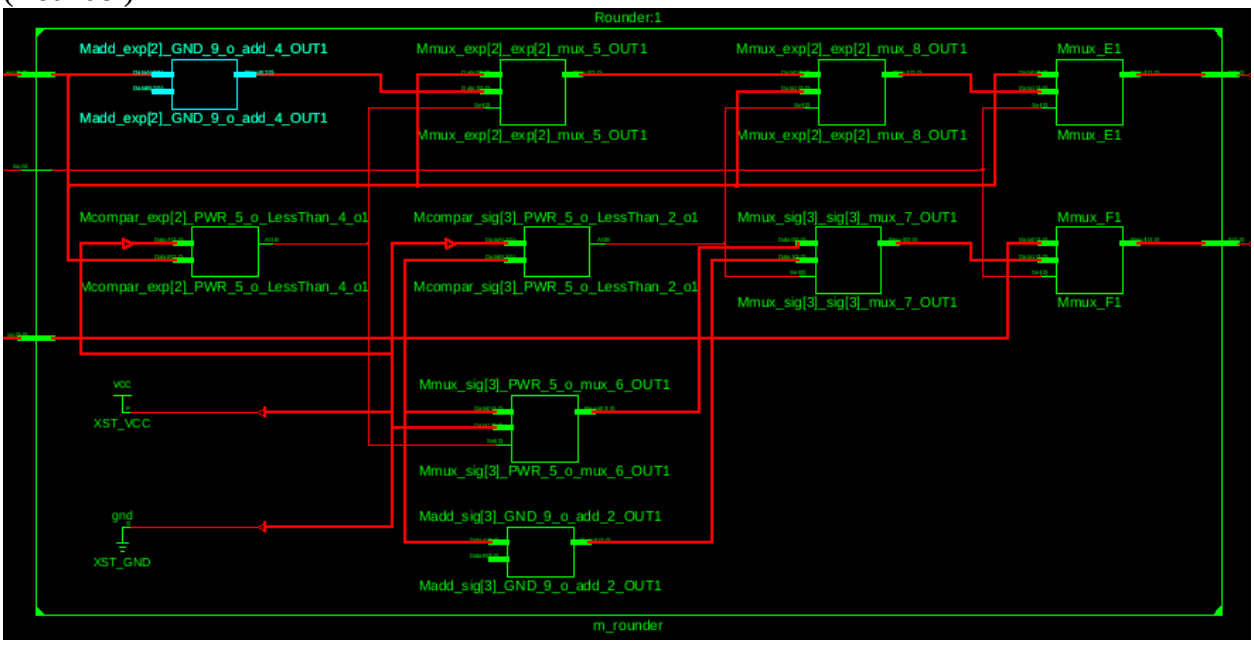
(SignedMagnitude)



(Separator)



(Rounder)



Inferences:

(Challenges)

The main challenges of this lab were figuring out the specifics of the Verilog language, and how it ties to hardware. Specifically for this lab, it was the implementing of various edge cases so that integer overflow wouldn't happen, like 12'b100000000000 in "SignedMagnitude"; bit-shifting in "Separator"; and the case of "exp" = 3b'111, "sig" = 4'b1111, and "round" = 1 in "Rounder".

(Mistakes)

In this lab, because of little to no background in Verilog, we wrote the code without thinking too much about the underlying hardware. Since conditionals are heavily tied to multiplexers, in the "Separator" submodule, because of unrestricted use of while-loops instead of regular if-else-if conditionals, the complexity of the generated netlist rose exponentially compared to the other submodules. In the future, we should focus more on how the Verilog corresponds to hardware and avoid using while-loop like the TA said.