Joshua Aymett

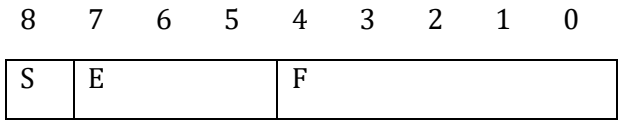
COM SCI M152A-6

01/24/2021

Project 1: Floating Point Conversion Lab Report

Behavioral Summary

The circuit takes one 13-bit two's complement input and outputs one 9-bit floating point representation with (1) sign bit, (3) exponent bits, and (5) significand bits. Two's complement is a signed linear numeric format while floating point is a signed exponential representation of the form <SIGN>_<EXPONENT>_<SIGNIFICAND>. Many different floating point representations exist; however, for this project, the following format will be used:



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

Source: CS M152A Lab1

While floating point is capable of storing numbers of greater magnitude with fewer bits, rounding errors occur for numbers that exceed the precision of its significand. As a result, not every integer across the number line may be precisely represented. In the case of this project, 1 bit is reserved for the sign (S), 3 bits for the exponent (E), and 5 bits for the significand (F). Inputs are of the following format:

FPCVT Pin Descriptions	
D [12 : 0]	Input data in Two's Complement Representation. D0 is the Least Significant Bit (LSB). D12 is the Most Significant Bit (MSB).
S	Sign bit of the Floating Point Representation.
E [2 : 0]	3-Bit Exponent of the Floating Point Representation.
F [4 : 0]	5-Bit Significand of the Floating Point Representation.

Source: CS M152A Lab1

Schematics

****For detailed descriptions, see "Logical Summary" below**

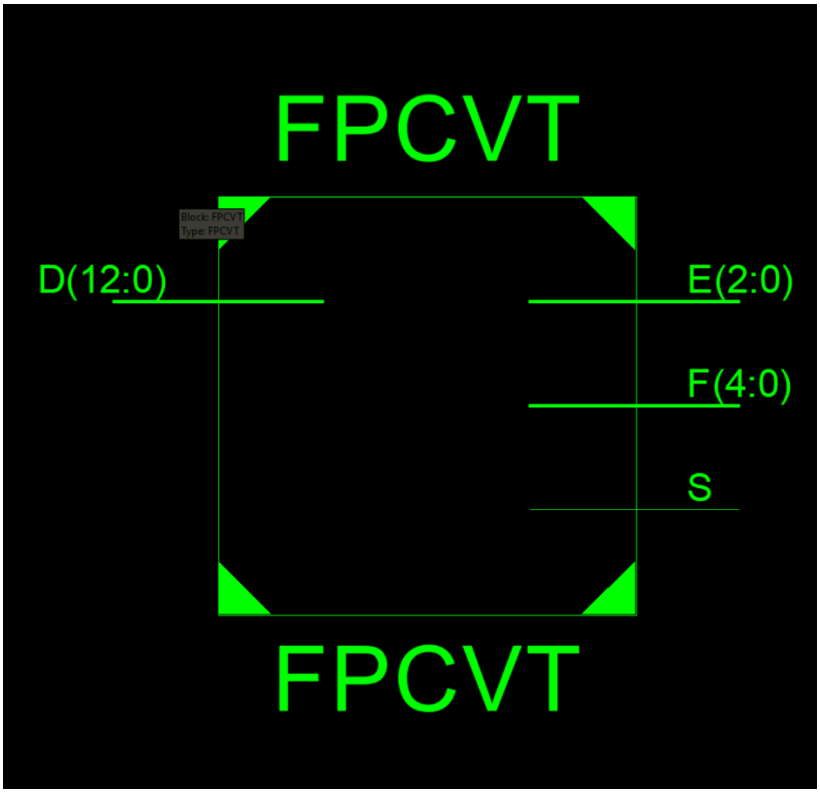


Diagram of main module, FPCVT, as a black box. A two's complement number is entered through the data input, D, and the floating point representation is outputted through E, F, and S.

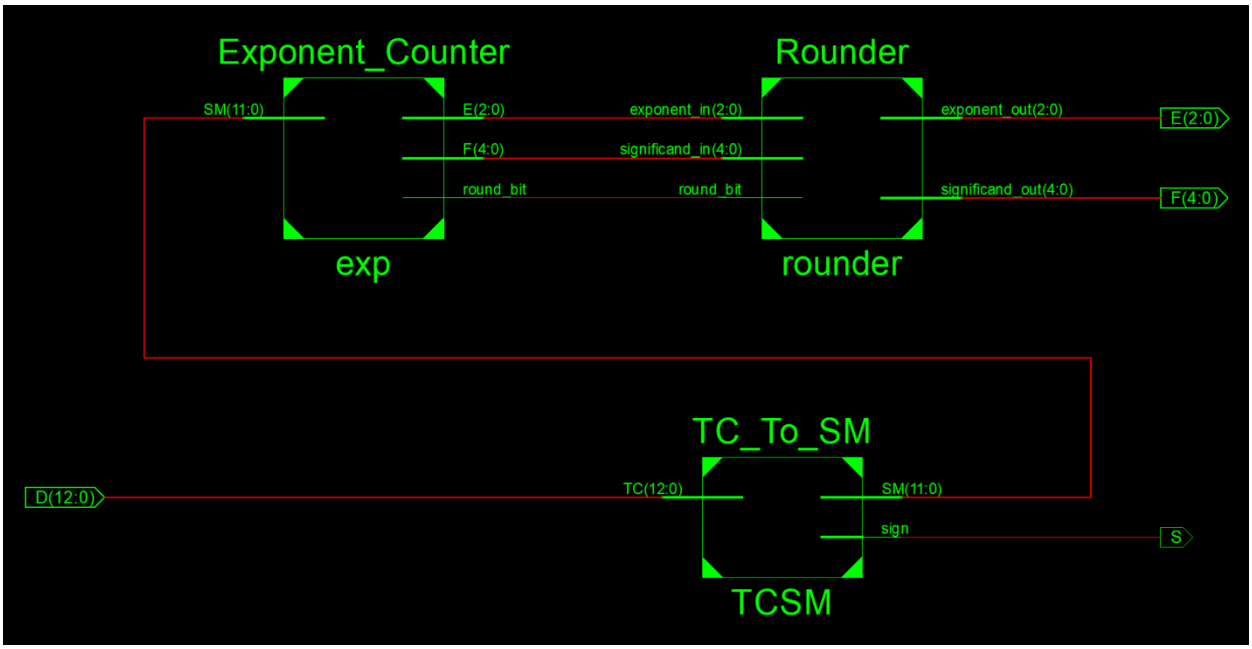


Diagram of FPCVT submodule connections.

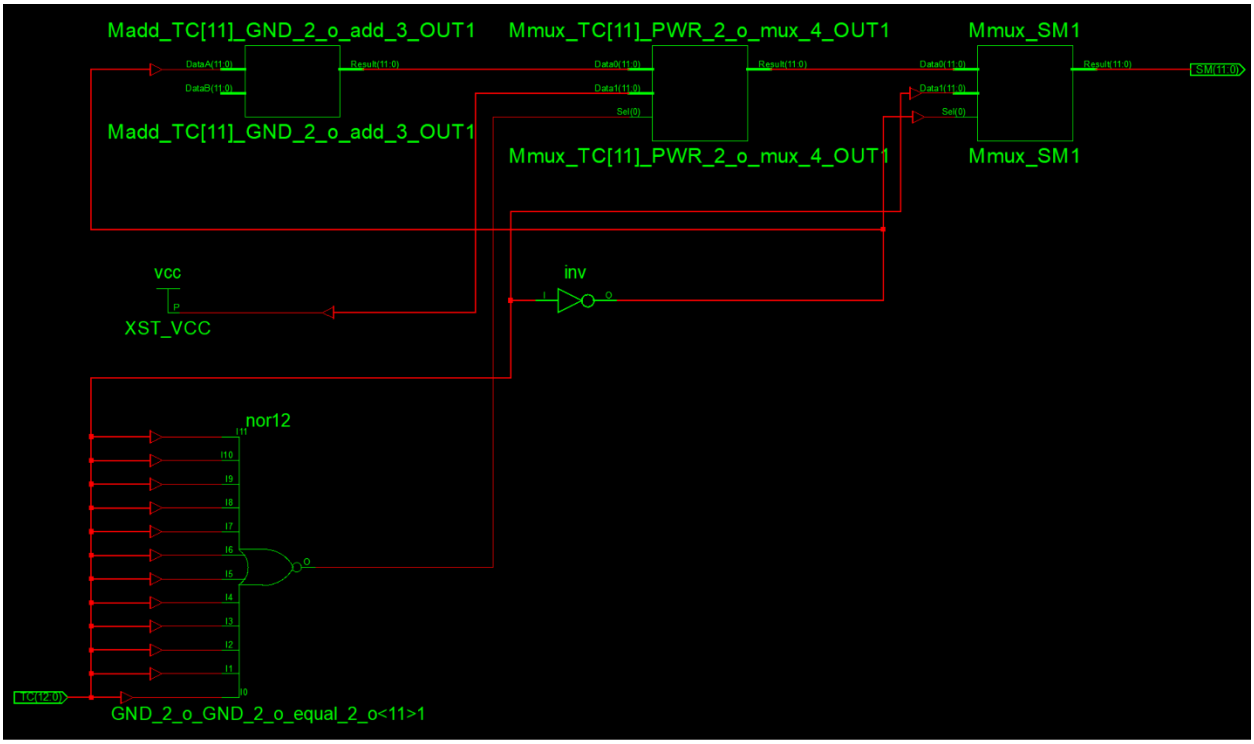


Diagram of TC_To_SM primitives.

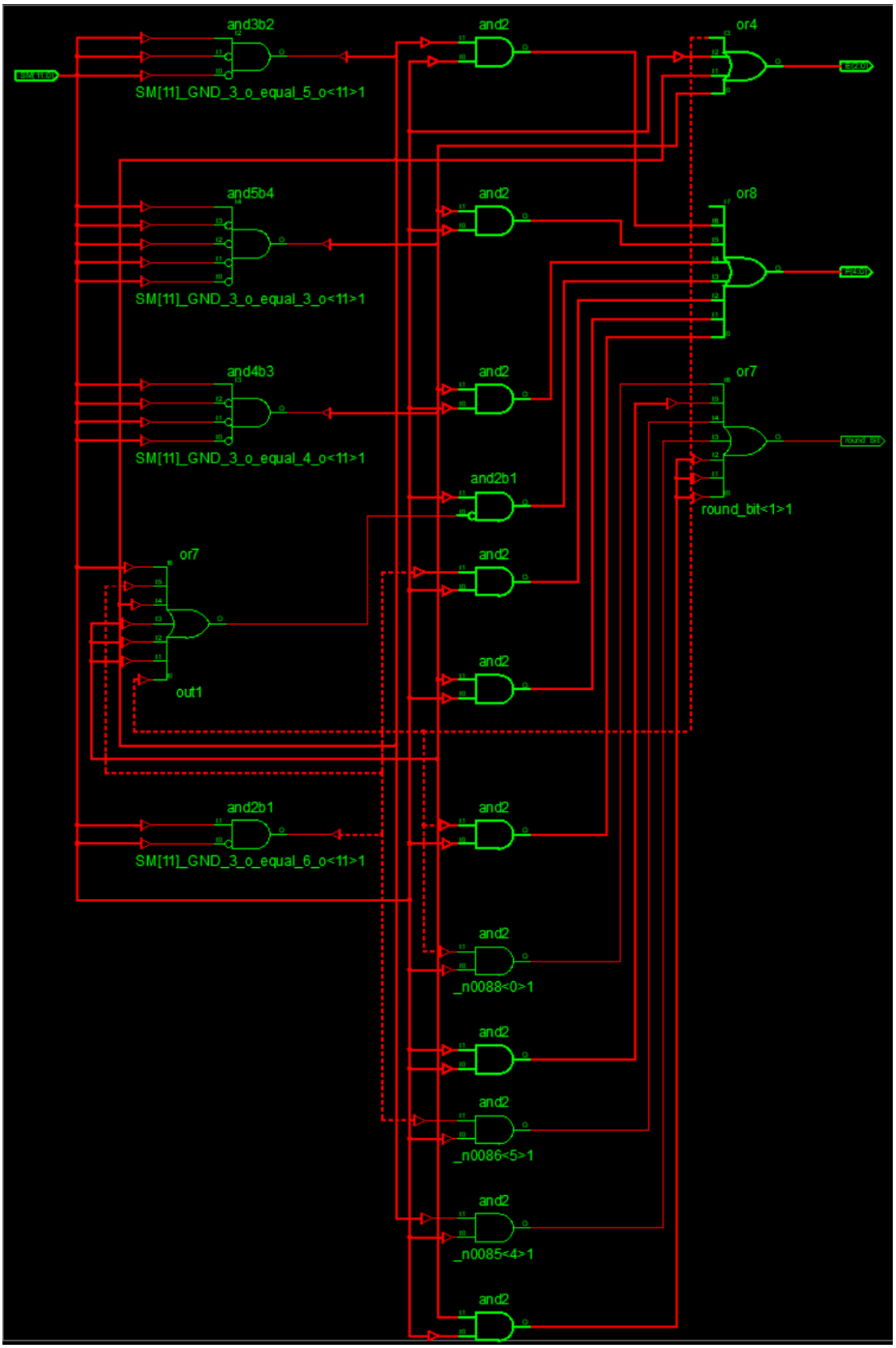


Diagram of Exponent_Counter primitives.

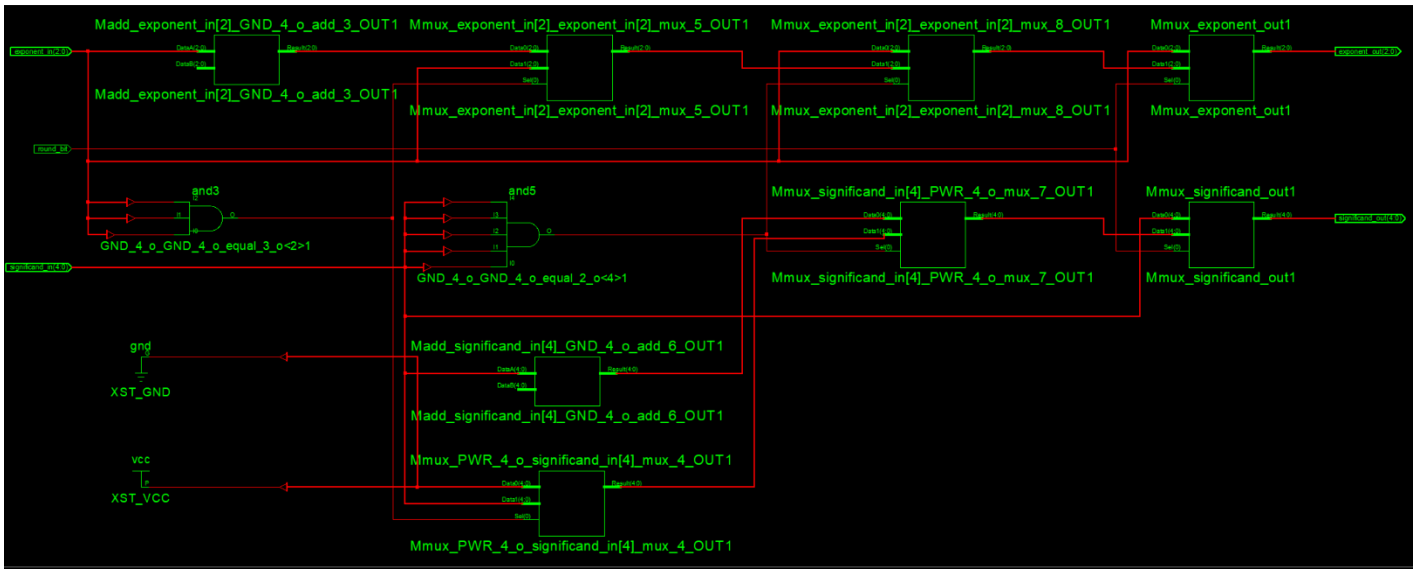


Diagram of rounder primitives.

FPCVT Project Status (01/24/2021 - 12:51:10)			
Project File:	Project1.xise	**Parser Errors:**	No Errors
Module Name:	FPCVT	**Implementation State:**	Placed and Routed
Target Device:	xc6slx16-3csg324	**• Errors:**	No Errors
Product Version:	ISE 14.7	**• Warnings:**	No Warnings
Design Goal:	Balanced	**• Routing Results:**	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	**• Timing Constraints:**	
Environment:	System Settings	**• Final Timing Score:**	0 (Timing Report)

Device Utilization Summary				[-]
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	0	18,224	0%	
Number of Slice LUTs	64	9,112	1%	
Number used as logic	63	9,112	1%	
Number using O6 output only	50			
Number using O5 output only	11			
Number using O5 and O6	2			
Number used as ROM	0			
Number used as Memory	0	2,176	0%	
Number used exclusively as route-thrus	1			
Number with same-slice register load	0			
Number with same-slice carry load	1			
Number with other load	0			
Number of occupied Slices	23	2,278	1%	
Number of MUXCYs used	12	4,556	1%	
Number of LUT Flip Flop pairs used	64			
Number with an unused Flip Flop	64	64	100%	
Number with an unused LUT	0	64	0%	
Number of fully used LUT-FF pairs	0	64	0%	
Number of slice register sites lost to control set restrictions	0	18,224	0%	
Number of bonded IOBs	22	232	9%	
Number of RAMB16BWERs	0	32	0%	
Number of RAMB8BWERs	0	64	0%	
Number of BUFIO2/BUFIO2_2CLKs	0	32	0%	
Number of BUFIO2FB/BUFIO2FB_2CLKs	0	32	0%	
Number of BUFG/BUFGMUXs	0	16	0%	
Number of DCM/DCM_CLKGENs	0	4	0%	
Number of ILOGIC2/ISERDES2s	0	248	0%	
Number of IODELAY2/IODRP2/IODRP2_MCBs	0	248	0%	
Number of OLOGIC2/OSERDES2s	0	248	0%	
Number of BSCANs	0	4	0%	
Number of BUFHs	0	128	0%	
Number of BUFPLLs	0	8	0%	
Number of BUFPLL_MCBs	0	4	0%	
Number of DSP48A1s	0	32	0%	
Number of ICAPs	0	1	0%	
Number of MCBs	0	2	0%	
Number of PCILOGICSEs	0	2	0%	
Number of PLL_ADVs	0	2	0%	
Number of PMVs	0	1	0%	
Number of STARTUPs	0	1	0%	
Number of SUSPEND_SYNCs	0	1	0%	
Average Fanout of Non-Clock Nets	4.16			

Performance Summary				[-]
Final Timing Score:	0 (Setup: 0, Hold: 0)	**Pinout Data:**	Pinout Report	
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report	
Timing Constraints:				

Detailed Reports					[-]
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**
Synthesis Report	Current	Sun Jan 24 13:15:46 2021	0	0	0
Translation Report	Current	Sun Jan 24 15:41:46 2021	0	0	0
Map Report	Current	Sun Jan 24 15:41:54 2021	0	0	6 Infos (0 new)
Place and Route Report	Current	Sun Jan 24 15:42:01 2021	0	0	2 Infos (0 new)
Power Report					
Post-PAR Static Timing Report	Current	Sun Jan 24 15:42:06 2021	0	0	4 Infos (0 new)
Bitgen Report					

Secondary Reports			[-]
Report Name	**Status**	**Generated**	
ISIM Simulator Log	Current	Sun Jan 24 15:23:43 2021	
Post-Synthesis Simulation Model Report	Out of Date	Sun Jan 24 12:53:35 2021	

Date Generated: 01/24/2021 - 15:42:11

Logical Summary

FPCVT is the central module which converts from two's complement to floating point representation. It is composed of three submodules (detailed below) and takes a 13-bit two's complement data input (D) while outputting a 1-bit sign (S), 3-bit exponent (E), and 5-bit significand (F).

```
module FPCVT(
    input [12:0] D,      // Data input
    output S,            // Sign output
    output [2:0] E,      // Exponent output
    output [4:0] F       // Significand output
    );
```

The submodules are connected as follows:

input -> TC_To_SM -> Exponent_Counter -> Rounder -> output

TC_To_SM takes the data input, passing the sign bit directly out and the sign-magnitude representation to Exponent_counter. Exponent_Counter takes the aforementioned input and passes the exponent, significand, and round bit to Rounder. Rounder then rounds its inputs and its outputs are assigned to S, E, and F as the final outputs from FPCVT.

-

TC_To_SM converts from two's complement to sign-magnitude representation of the form <SIGN_BIT>_<MAGNITUDE BITS>. This module takes the two's complement representation as a 13-bit input (TC) and outputs a 1-bit sign (sign) and 12-bit magnitude (SM).

```
module TC_To_SM(
    input [12:0] TC,      // Two's complement input
    output sign,          // Output for sign value
    output reg [11:0] SM    // Sign magnitude representation
    );
```

This module uses the following algorithm:

```
output sign bit
if (positive) -> output the magnitude, TC[11:0];
else -> check for special case where -4096 results in overflow.
    if (special case) -> output 1111_1111_1111
    else -> output the two's complement of TC[11:0]
```

Exponent_Counter converts the sign-magnitude representation to unrounded floating point format. It takes the 12-bit sign-magnitude representation from TC_To_SM and outputs a 3-bit exponent (E), 5-bit significand (F), and 1-bit rounding flag (round_bit).

```
module Exponent_Counter (
        input [11:0] SM,       // Sign magnitude input
        output reg [2:0] E,    // Unrounded exponent output
        output reg [4:0] F,    // Unrounded significand output
        output reg round_bit   // Bit to determine if rounding should occur
        );
```

A casex statement counts the leading zeros in the sign-magnitude input. If 7 or more of the leading bits from the 12-bit sign magnitude input are 0, the last 5 bits become the significand and the exponent is 000. Don't care conditions are used to match bits following the first 1.

-

Rounder rounds the outputs of Exponent_Counter to the nearest floating point approximation of the sign-magnitude representation. It takes a 3-bit unrounded exponent (exponent_in), 5-bit unrounded significand (significand_in), and 1-bit round flag (round_bit) as inputs.

```
module Rounder (
        input [2:0] exponent_in,         // Unrounded exponent input
        input [4:0] significand_in,      // Unrounded significand input
        input round_bit,                 // Round bit to deterimine if rounding should occur
        output reg [2:0] exponent_out,   // Rounded exponent output
        output reg [4:0] significand_out // Rounded significand output
        );
```

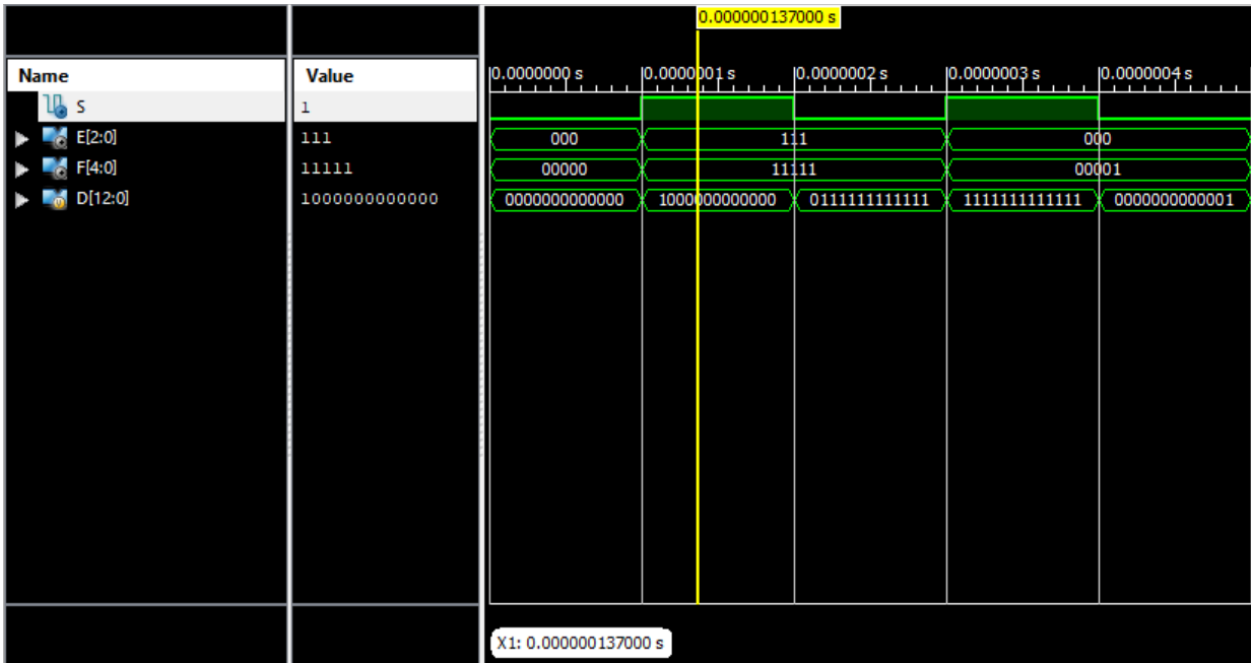
This module uses the following algorithm:

```
if (rounding is necessary)
    if (significand overflow will occur)
        if (exponent overflow will occur) -> output unrounded exponent and significand (largest possible number)
        else -> output significand=10000 and incremented exponent_in
    else -> output incremented significand_in and unrounded exponent_in
else -> output unrounded values (rounding already accomplished through truncation)
```

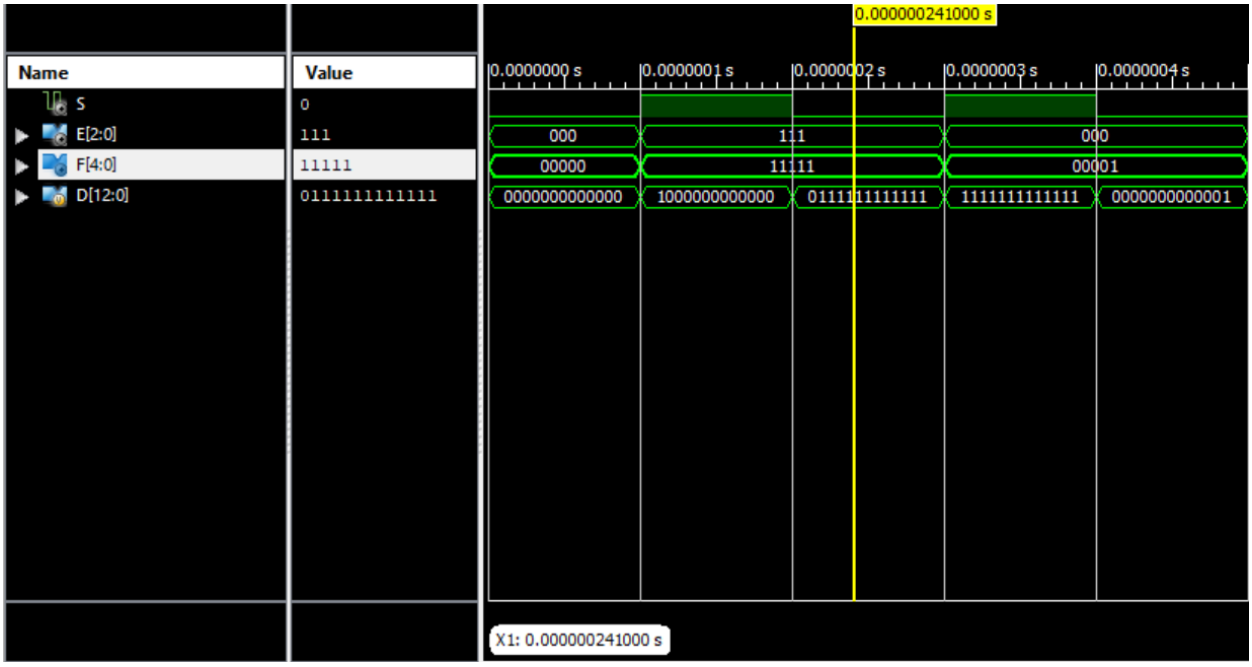
Simulation Output

Selected test cases include smallest number, greatest number, 0, exact positive and negative representations, rounding up, and rounding down. These cases account for potential errors in two's complement to sign-magnitude conversion, exponent counting, and rounding. Descriptions follow the pattern "Test <number_or_test_case_tested> (<data_input>); Expected output: <expected_output>" where <number_or_test_case_tested> describes either an exact number that should be represented or the description of a potential edge case.

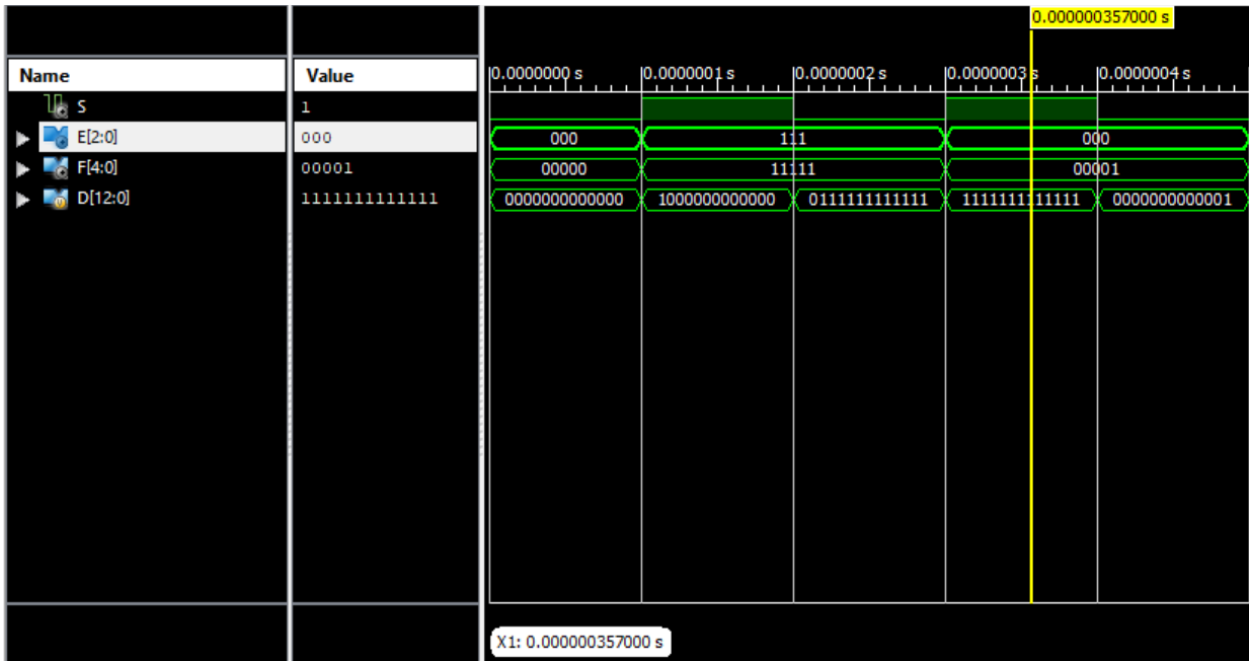
Test 0 (D = 'b0_0000_0000_0000); Expected output: 0_000_00000



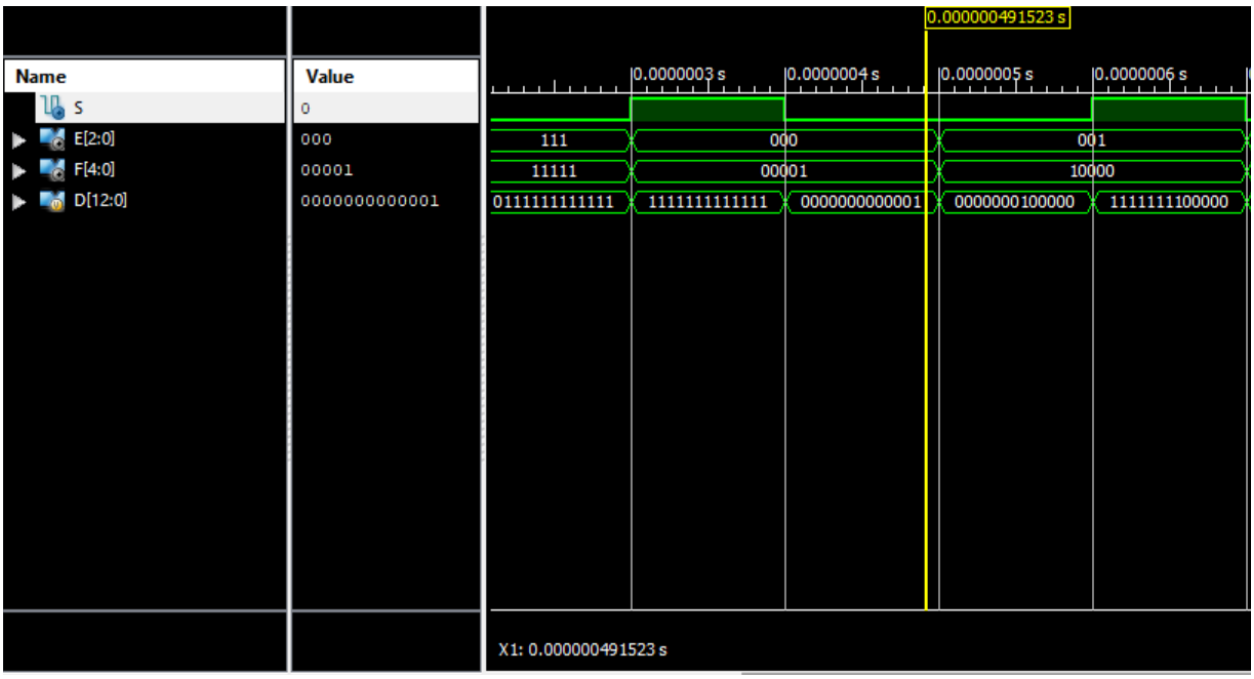
Test -4096 (D = 'b1_0000_0000_0000); Expected output: 1_111_11111



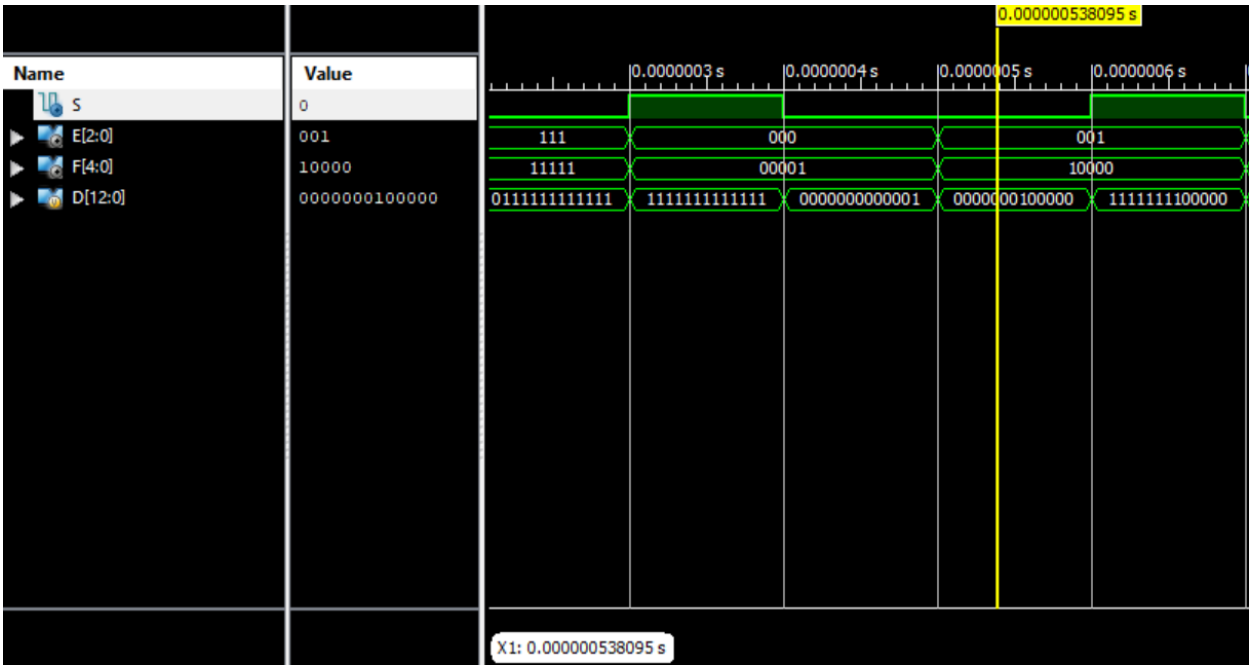
Test 4095 (D = 'b0_1111_1111_1111); Expected output: 0_111_11111



Test -1 (D = 'b1_1111_1111_1111); Expected output: 1_000_00001

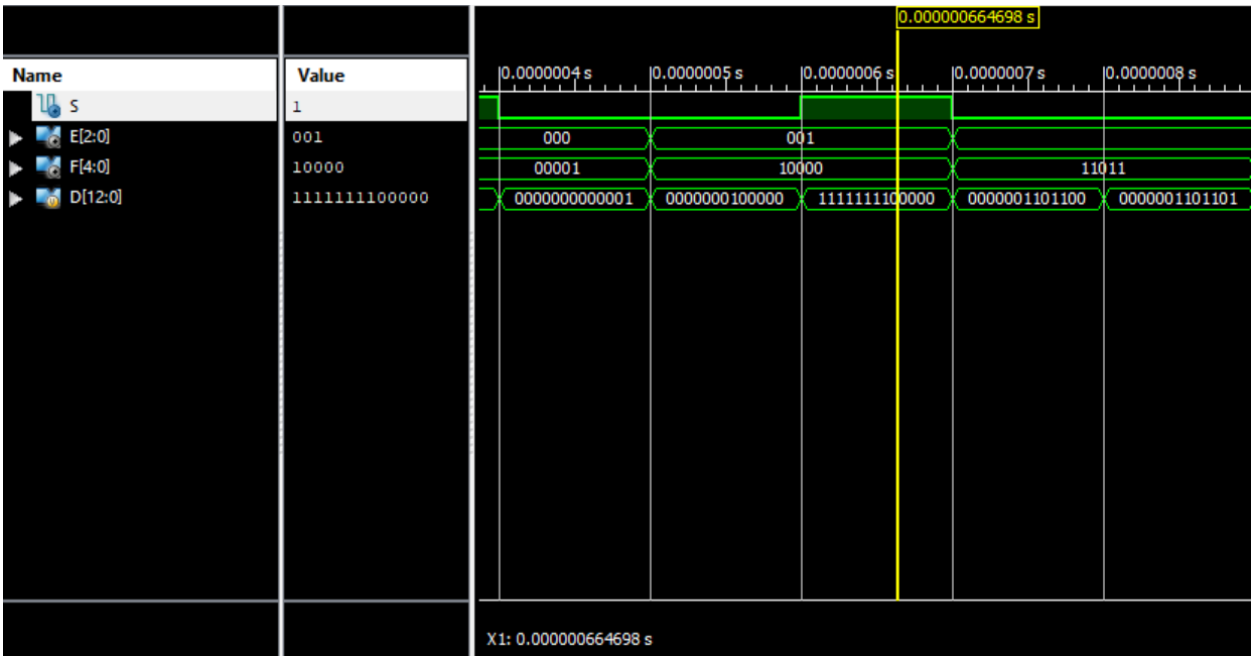


Test 1 (D = 'b0_0000_0000_0001); Expected output: 0_000_00001



Test positive exact representation with exponent value (32, D = 'b0_0000_0010_0000);

Expected output: 0_001_10000

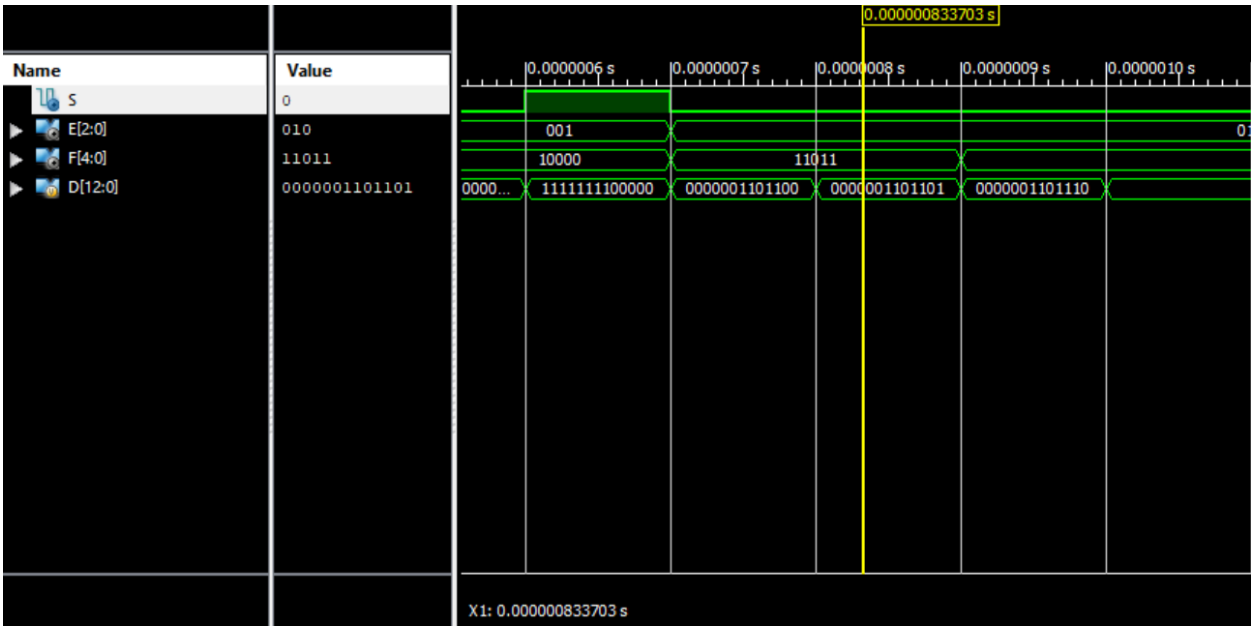


Test negative exact representation with exponent value (-32, D = 'b1_1111_1110_0000)

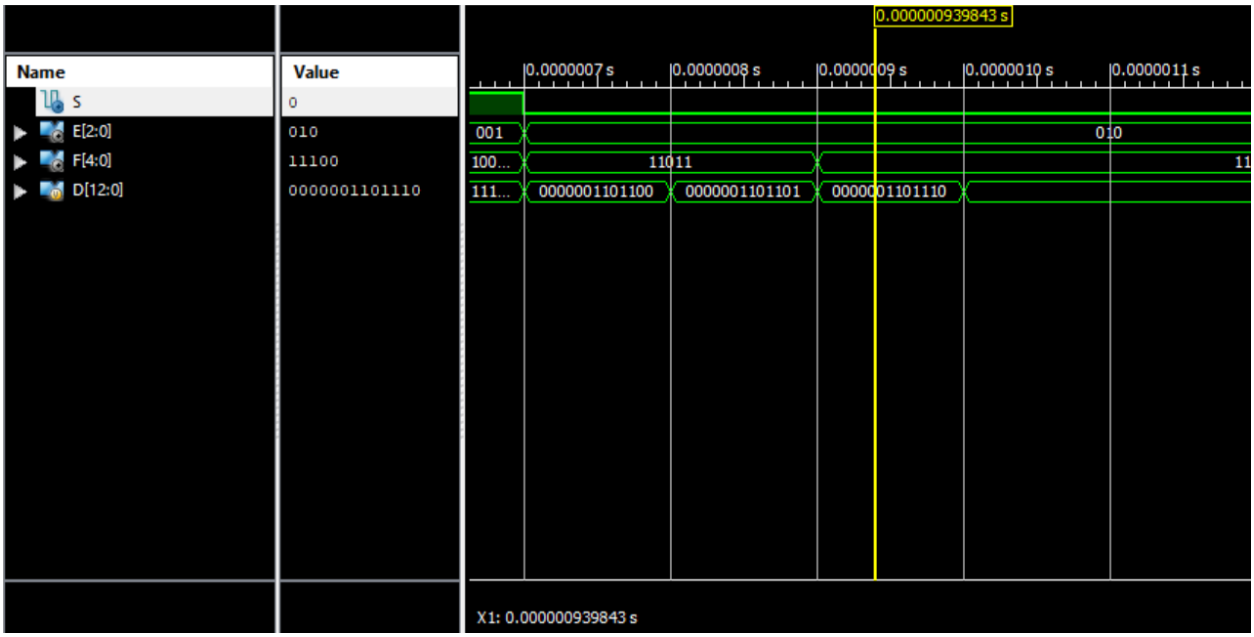
Expected output: 1_001_10000

Test cases from the project assignment:

Test round down (D = 'b0_0000_0110_1100); Expected output: 0_010_11011



Test round down (D = 'b0_0000_0110_1101); Expected output: 0_010_11011



Test round up (D = 'b0_0000_0110_1110); Expected output: 0_010_11100

Test round up (D = 'b0_0000_0110_1111); Expected output: 0_010_11100

Post-Project Evaluation

For this project, I took the recommended approach of one main module with three submodules. I found this to be a very simple and effective method when implementing the circuit design and as a result, I encountered no significant obstacles. My intention in designing the circuit was to create a modular and testable set of modules that I could connect together while ensuring functionality. To test them, I created a testbench with a variety of edge cases and normal cases (detailed above). Values were outputted using temporary outputs that would display debug values as new submodules were added. This proved to be very effective as I could design testing procedures up front and track functionality as submodules were connected. Because of my design process, I was able to catch errors before I added new modules which allowed me to pinpoint the root cause. For example, when designing the rounder, I added logic for the exponent overflow, but forgot to added logic for a significand overflow as well. Because I had already designed my testbench, I was able to immediately catch the problem and fix it.

I found this project to be an excellent way to practice Verilog and by the end, I was easily writing modules without documentation. The assignment was practical and of reasonable difficulty which allowed me to finish the project having learned useful skills. To improve the project, I recommend that the assignment description be updated to include a checklist of grading criteria. Being new to Xilinx and Verilog, vocabulary can sometimes be confusing. It would be helpful to have a checklist that describes exactly what components are in each part of the submission instead of having to dig through lectures to double check my notes.