**Design Description:**

The design has 4 major modules:

* FPCVT
* twos\_complement\_to\_sign\_magnitude
* linear\_to\_floating\_point
* rounding

The main module is FPCVT, and it instantiates the other 3 modules. The logic flow within these three modules is shown in Figure #.

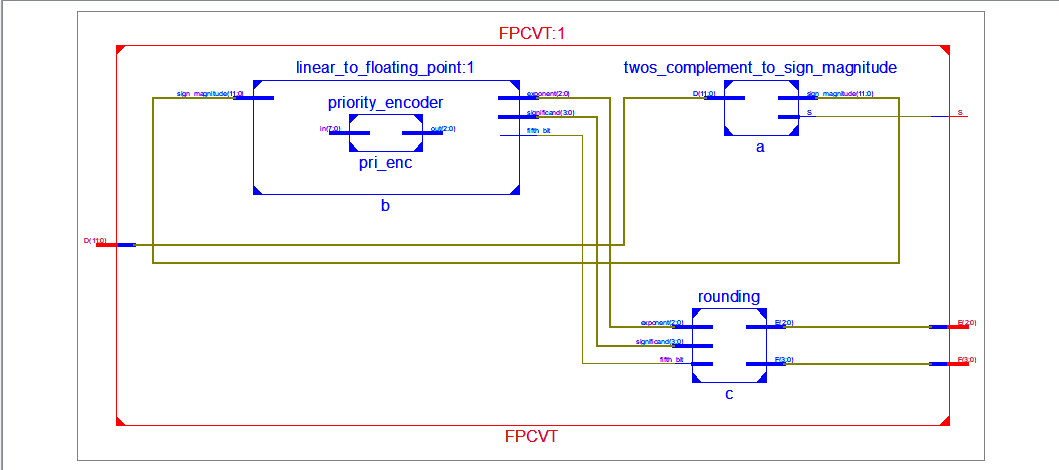


Figure #: The above figure shows the general schematic of the Verilog code. In the middle left, the input D enters FPCVT and connects to the input of the twos\_complement\_to\_sign\_magnitude in the top right corner. One of the outputs of this module S is the direct output of FPCVT. The other output sign\_magnitude gets sent as input to linear\_to floating\_point, which is the module on the top left. The three outputs of linear\_to\_floating\_point are all provided as inputs to rounding. Finally, the two outputs of rounding, E and F, are outputs of the FPCVT module.

Each module has its own function as defined below. The first step in the process is to extract the sign bit and the magnitude of the two’s complement number. The sign bit is outputted as is, and the magnitude is sent to the “linear\_to\_floating\_point” module. This module converts it to an approximate floating-point number with some rounding errors. It outputs the approximate exponent, fraction and the fifth bit; all of which is fed to the rounding module. The rounding module corrects the fraction as described in the section below. The schematic in Figure # shows the modular architecture of the Verilog code.

FPCVT:

|  |  |
| --- | --- |
| Input | Output |
| * D | * S * E * F |

Table #: The above table shows the interface of the FPCVT module. This is the outermost module. It takes in a 12-bit two’s complement integer as input. It compresses it to a 8-bit floating point number. It produces a 1-bit output that is the sign (S), a 3-bit exponent (E) and a 4-bit significand (F).

This module is the outermost module for our design. It takes as input a 12-bit 2’s complement integer, and produces a compressed 8-bit floating-point representation of this number. FPCVT does this by instantiating the 3 modules: twos\_complement\_to\_sign\_magnitude, linear\_to\_floating\_point, and rounding.

twos\_complement\_to\_sign\_magnitude:

|  |  |
| --- | --- |
| Input | Output |
| * D | * S * sign\_magnitude |

Table #: The above table shows the interface of the twos\_complement\_to\_sign\_magnitude module. The module takes a 12-bit two’s complement integer (D) as input. It then produces as output, a one bit number that is the sign of the two’s complement number (S) and a 12-bit number which is the magnitude of the two’s complement number (sign\_magnitude).

This module works by extracting the MSB of the input D. If the MSB is a 0, then S is assigned a 0, because our input is positive, and so should our final floating point number. In this case, sign\_magnitude remains unaffected and the output sign\_magnitude is just D. If the MSB is a 1, then S is assigned a 1, because our input is negative, and our final output should also be negative. In this case, sign\_magnitude is calculated by negating D, according to two’s complement rules (flipping all the bits and adding 1).

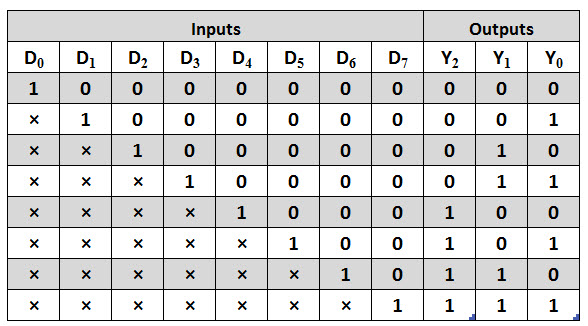
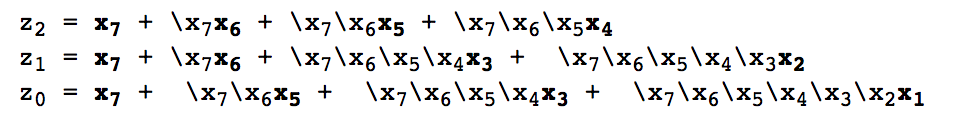
The output sign\_magnitude is then fed into the module linear\_to\_floating\_point. S becomes the output of the outermost module, FPCVT.

priority\_encoder:

|  |  |
| --- | --- |
| Input | Output |
| * in | * out |

Table #: The above table shows the interface of the priority encoder module. It takes in as input an 8 bit number (in) and outputs a 3 bit number (out).

This module is a 8🡪3 priority encoder. It takes an 8 bit number as input, and outputs the position of the most-significant high bit. The truth table and logic for this is shown in Figure #.

<http://www.electronicshub.org/wp-content/uploads/2015/06/Octal-to-Binary-Priority-Encoder.jpg>

https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Comb/encoder.html

linear\_to\_floating\_point:

|  |  |
| --- | --- |
| Input | Output |
| * sign\_magnitude | * exponent * significand * fifth\_bit |

Table #: The above table shows the interface of the module linear\_to\_floating\_point. The module takes in a 12-bit unsigned number (sign\_magnitude). It produces as output a 3-bit number (exponent), a 4-bit output (significand) and a 1 bit number (fifth\_bit). The way these outputs is produced is described below.

The module works by first extracting the number of leading zeros in the sign\_magnitude number. This is done using a priority encoder, which is described above. Once we know the number of leading zeros, we set the exponent according to the mapping given in Figure #.

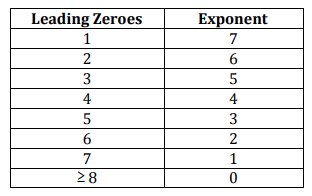


Figure #: The above figure describes the mapping between the number of leading zeros in the unsigned number and the value of exponent. (Reference Lab Manual)

Once we know the exponent, we extract the next 4 bits, and set that as our significand. If there is a 5th bit, we extract that and set that as fifth\_bit, else we set fifth\_bit as 0. In the special case that our input is -2048 (in decimal), we set exponent and significand to the maximum possible value, that is 3’b111 for exponent and 4’b1111 for significand and set fifth\_bit as 0.

The outputs exponent, significand and fifth\_bit are pipelined to the rounding module.

rounding:

|  |  |
| --- | --- |
| Input | Output |
| * exponent * significand * fifth\_bit | * E * F |

Table #: The above table describes the interface of the rounding module. It takes as input a 3-bit exponent, a 4-bit significand and a 1-bit number that represents the fifth bit. It produces as output, the final exponent E (3 bits), and the final significand F (4 bits).

The rounding module performs rounding of the floating-point number as described in the Lab Manual. It takes as input the exponent, significand and the fifth bit as produced in linear\_to\_floating\_point, and outputs the final exponent and floating point after performing rounding.

To perform rounding, we follow these steps. If the fifth bit is 0, we set E as exponent and F as significand, without making any changes. If the fifth bit is a 1, we add it to the significand. In the case that the addition doesn’t overflow, we set F as this new significand value and set E as exponent. However, if it does overflow, we right shift the significand by 1, add 1 to the exponent, and assign E and F as the updated exponent and significand respectively. If adding 1 to the exponent, results in overflow, we set E and F as all the maximum possible value. (E = 3’b111, F = 4’b1111).

**Conclusion**

The design described above successfully met the specifications and worked for all edge cases. Dividing up the code into separate modules enabled us to unit test and makes it possible to modify a module in the future without affecting any of the other modules.

We faced an obstacle in getting used to how Verilog simultaneously executes multiple statements that are not separated by time delays. We faced errors while testing the modules that included a display statement. The output printed to the console didn’t match the waveform that the code created. We worked through the code multiple times to figure out the bug. Another challenge we faced was modifying the standard priority encoder to an encoder give us the leading number of 0s instead of the position of the most significant high bit. We dealt with this issue by working through a couple of examples on paper, seeing the output of a standard priority encoder and what the output of the modified priority encoder should be. This enabled us to come up with a conversion that worked well.

Overall, this lab was a good introduction to writing complex modules in Verilog by breaking them into smaller modules.