Danning Yu, 305087992
CS M152A, Lab 2
TA: Logan Kuo

Project 2 Report

Introduction

The purpose of this lab is to build a combinational module that performs compression by converting a 13 bit two's complement number into a floating point format approximation with a 1 bit sign, 3 bit exponent, and 5 bit significand (mantissa). The conversion process can be split into individual steps that can each be represented as a module, so we will learn how to implement submodules and then combine them together to form a top level module that represents the converter as a whole. After designing the module, test bench code will be written to test it in Xilinx Isim, and then synthesis and implementation will be carried out to view the design summary, which will show the physical resource usage if this was implemented on an actual FPGA board.

Note: In this report, binary numbers are written within square brackets. Spacing may be added between the bits for easier readability.

Design

The decimal equivalent of a 13 bit two's complement number, represented by $b_{12}b_{11}b_{10}b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0$, is given by the following formula below, which is the standard formula for a two's complement number.

$$-b_{12} \cdot 2^{12} + \sum_{i=0}^{11} b_i \cdot 2^i$$

The 9 bit floating point number is comprised of a 1 bit sign value $S = s$, 3 bit exponent $E = e_2e_1e_0$, and a 5 bit significand $F = f_4f_3f_2f_1f_0$, and they are packed together to form a 9 bit value $se_2e_1e_0f_4f_3f_2f_1f_0$. Its decimal equivalent is given by the formula below, where S, E, and F have been interpret them as unsigned binary numbers and converted into their decimal equivalent.

$$\text{Number} = (-1)^S \cdot F \cdot 2^E$$

Floating point representations of numbers have some properties that two's complement numbers do not have, and these properties must be taken into consideration when converting a number from two's complement to floating point. The first is that two's complement can represent every integer between its minimum and maximum values, but floating point cannot. For example, 355 is [0 0001 0110 0011] in two's complement, but has no floating point representation that evaluates to this value; the closest values are [0 100 10110] = 352 and [0 100 10111] = 368. Thus, we must account for this decreased resolution when carrying out the conversion by representing a two's complement number with the closest possible floating point number. Furthermore, each two's complement number uniquely corresponds to a decimal number, but a number can have multiple floating point representations. For example, 352 can be represented as either [1 100 10110] or [1 101 01011]. For this converter, we will favor setting the most significant bit of the significant whenever possible. For example, for 352, [1 100 10110]

would be the preferred representation. Finally, the minimum possible input value is –4096 and the maximum possible value is 4095. However, for the floating point representation, the minimum is [1 111 11111] = –3968 and [0 111 11111] = 3968. Thus, for any input value that falls outside this range, if it is smaller than –3968, we will output [1 111 11111]; if it is larger than 3968, we will output [0 111 11111].

Floating Point Converter (FPCVT, Top Level Module)

This is the top level module whose function is to take in a 13 bit input, representing a two's complement number, route it through various submodules to convert it to a 9 bit compounded floating point representation, and then output that representation. It also reads the input to directly output the sign bit of the floating point result. A summary of the inputs and outputs are given below.

Input/Output	Description
`D[12:0]`	Input: 13 bit two's complement number
`S`	Output: 1 bit sign of the floating point representation
`E[2:0]`	Output: 2 bit exponent of the floating point representation
`F[4:0]`	Output: 5 bit significand of the floating point representation

Table 1: A summary of the input and outputs of the `FPCVT` module and their functions.

To convert the input to a floating point number, first extract the most significant bit and get the magnitude of the input. The most significant bit directly corresponds to `S`, and the magnitude will be used to determine the exponent `E` and significand `F`. Then, by determining how many leading zeroes are in the binary representation of the magnitude, the values of the exponent and significand can be determined. Counting leading zeroes to extract the significand ensures the most significant bit of the extracted significand is 1 for all cases where there is more than 1 possible floating point representation of the input. Finally, to account for the fact that the floating point representation will sometimes only be an approximation of the input value, we apply rounding rules to the exponent and significand. Rounding also takes care of situations where the input is greater than the maximum possible floating point value or more negative than the minimum possible floating point value. Each step will be described in more detail in their respective submodules. This process is summarized in the figure on the next page.

(project report continues on next page)

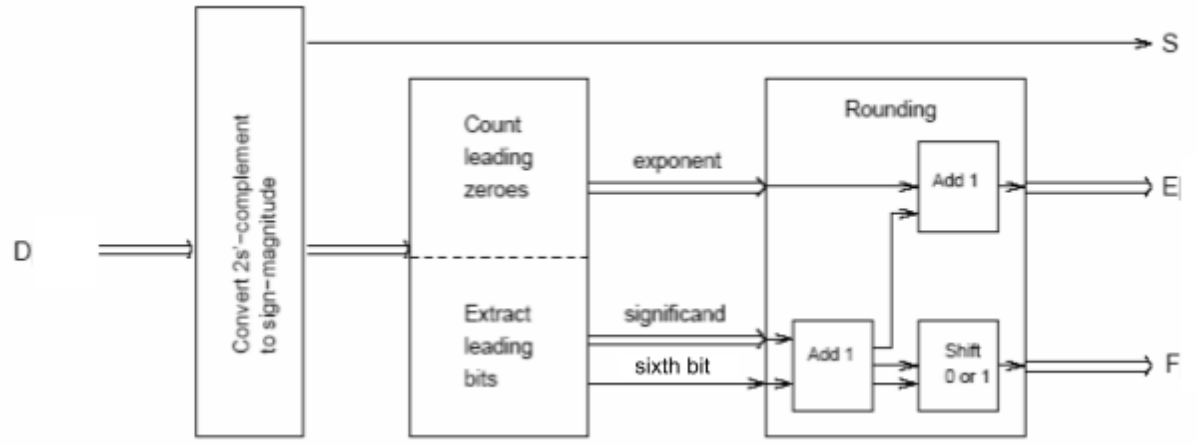

Figure 1: The high level design for converting a two's complement number to a floating point representation. The input `D` is converted to sign-magnitude, the number of leading zeroes is counted to determine the exponent, the leading bits are extracted to find the significand, and then rounding is applied to the exponent and significand in conjunction with a sixth bit value. Figure courtesy of the Project 2 specification document.

From this high level design schematic, various submodules were created in Verilog to implement each step, as shown in the image below. First, the most significant bit of the input `D`, which is the sign of the input, is directly read and passed on to the output as `S`. The full 13 bit input is passed into a `getMagnitude` module, which determines the 13 bit magnitude of the input, outputting it as `magnitude`.

The resulting 13 bit output `magnitude` is passed into a module `countZeroes`, which counts the number of leading zeroes and outputs a 3 bit result `unroundedPower`. This output, along with `magnitude`, is the input for `getMantissa`, which uses the number of leading zeroes (directly correlated to `unroundedPower`) to determine which 5 bits to extract from `magnitude` and output as the 5 bit value `unroundedF`. The names `F`, mantissa, and significand all stand for the same thing. `unroundedPower` and `magnitude` are also inputs to `getSixthBit`, which gets the next lowest bit immediately after the significand (`unroundedF`) and outputs it as the 1 bit value `sixthBit`.

Next, a rounding module, `roundEF`, applies the appropriate rounding to `unroundedPower` and `unroundedF` based on their values, as well as the value of `sixthBit`. It outputs two values, a 5 bit `roundedF` and a 6 bit `roundedPower`. Finally, to handle the edge case of –4096 being passed in, `roundedF` and `roundedPower` are passed in to the module `handle4096` along with an 1 bit input `detected4096` that signals whether –4096 was inputted or not. The value `detected4096` is the output of the module `detect4096`, which simply takes in `magnitude` and determines if it is 4096.

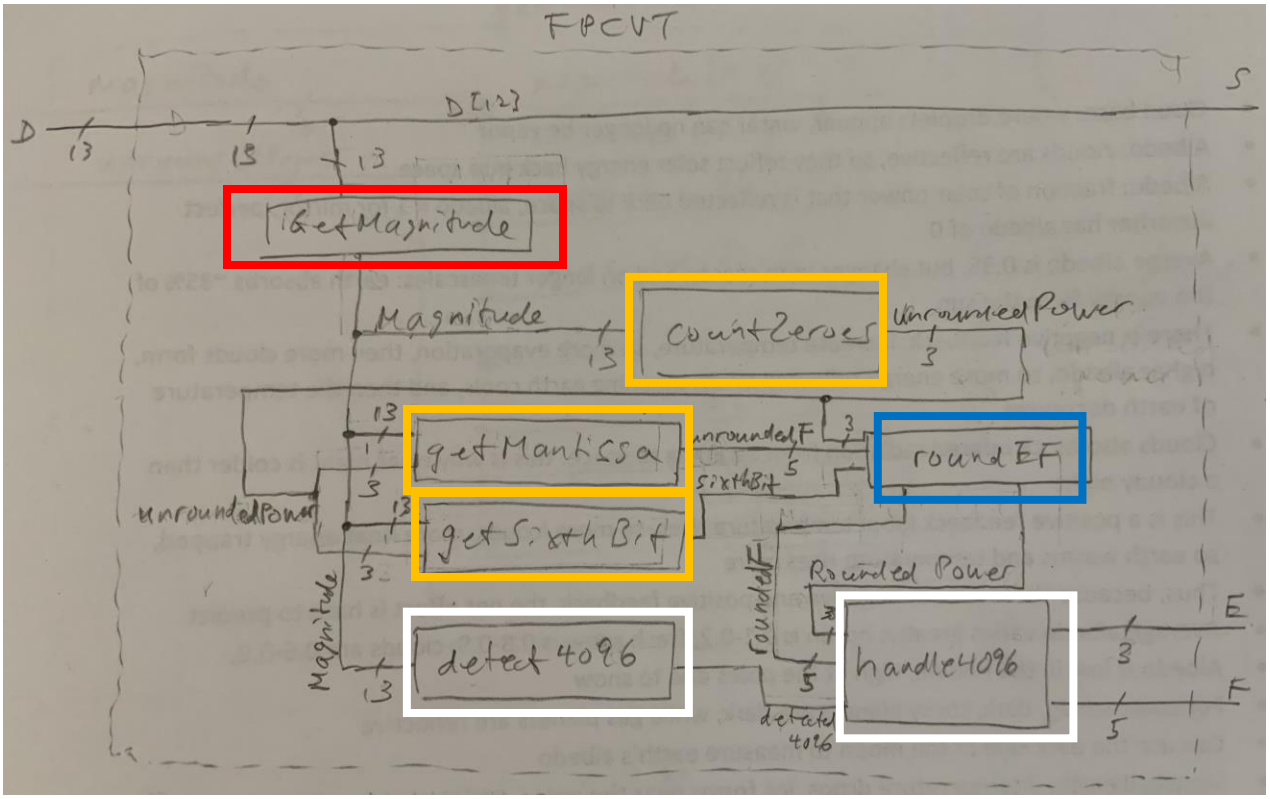

Figure 2: A module level schematic showing the submodules that make up the top level `FPCVT` module. A 13 bit input `D` is passed in and sent through a series of modules to ultimately output a sign bit `S`, exponent value `E`, and significand value `F`. The red box contains the module that implement the "convert 2's complement to sign magnitude" part of Figure 1. The orange box contains the count leading zeroes and extract leading bits step of Figure 1, and the blue box implements the rounding aspect of Figure 1. Finally, the white box handles the edge case of –4096 being passed in.

Using Figure 2, the top level Verilog module `FPCVT` was written. In the code excerpt from the module on the next page, one can see it consists primarily of modules, except for one line of code that automatically extracts the sign bit. The variables `D`, `S`, `E`, and `F` correspond to the same variables described in Table 1. Wires are used for intermediate values to pass between the modules.

(project report continues on next page)

```
--------------------------FPCVT Module Code--------------------------
wire [12:0] magnitude;
wire [2:0] unroundedPower;
wire [4:0] unroundedF;
wire sixthBit;

wire [2:0] roundedPower;
wire [4:0] roundedF;

wire detected4096;

assign S = (D[12] == 1'b0) ? 1'b0 : 1'b1;

getMagnitude magnitudeModule(.inVal(D), .magnitude(magnitude));
countZeroes countZeroesModule(.magnitude(magnitude),
                              .unroundedPower(unroundedPower));
getMantissa mantissaModule(.magnitude(magnitude),
                           .unroundedPower (unroundedPower),
                           .unroundedF(unroundedF));
getSixthBit sixthBitModule(.magnitude(magnitude),
                           .unroundedPower(unroundedPower),
                           .sixthBit(sixthBit));
roundEF roundingModule(.sixthBit(sixthBit), .unroundedF(unroundedF),
                       .unroundedPower(unroundedPower),
                       .roundedPower(roundedPower),
                       .roundedF(roundedF));
is4096 detect4096Module(.magnitude(magnitude),
                        .detect4096(detected4096));
handle4096 handle4096Module(.detected4096(detected4096),
                            .roundedPower(roundedPower),
                            .roundedF(roundedF), .E(E), .F(F));
----------------------------------------------------------------------
```

(project report continues on next page)

getMagnitude Module

The function of this module is to take in a 13 bit two's complement number `D` and output its 13 bit magnitude as `magnitude`. It does this by using the property that for a two's complement number *x*, its negative can be found by a bitwise complement (complement each bit individually) and then adding one: $\sim x + 1$. Thus, if `D` is positive, it simply passes `D` through; if it is negative, then it takes `D`'s bitwise complement and adds 1 using a 13 bit adder. The sign bit of `D` is used as the input to a 2 to 1 multiplexer that selects between these two values to output the correct value. This implementation is shown in the circuit diagram below.

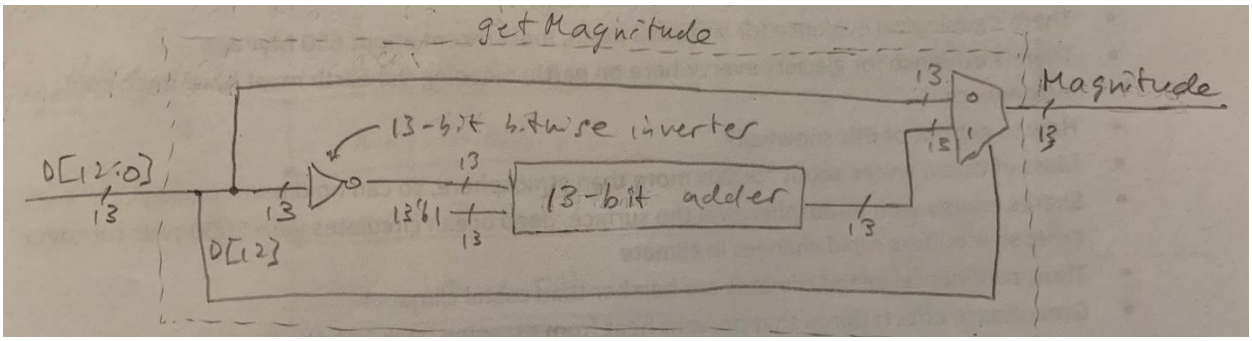

Figure 3: The `getMagnitude` module, which takes in the 13 bit input from the top level module, `D`, and uses its most significant bit `D[12]`, which is the sign bit, to pick between `D` and its negated version to output as 13 bit `magnitude`.

The corresponding Verilog code for this module is given below. It takes in a 13 bit input `D` and outputs a 13 bit value `magnitude`.

```
------------------------getMagnitude Module Code----------------------
always @(*) begin
  if(D[12] == 1'b1)
    magnitude = ~D + 13'b1;
  else
    magnitude = D;
end
----------------------------------------------------------------------
```

countZeroes Module

The function of this module is to count the number of leading zeroes in the magnitude of the value passed in to the top level `FPCVT` module and output an exponent value corresponding to the number of leading zeroes. To achieve this, it takes in a 13 bit input `magnitude` from the `getMagnitude` module and then uses a priority encoder-like logic to output a 3 bit value `unroundedPower` representing the exponent that corresponds to the number of leading zeroes. The mapping between number of leading zeroes and the power is shown in Table 2 below. The priority encoder is implemented as a case statement in Verilog that looks through each bit of magnitude, starting from the most significant, to find the first bit that is a 1, and thus the number of leading zeroes. When implemented, the result is a cascading series of multiplexers that take in

2 3-bit values and use each successive bit of `magnitude` to determine where the first 1 is located in `magnitude`, thus determining the number of leading zeroes. Each multiplexer, except for the first one, has one input as the output from the previous multiplexer, and the other input as the `unroundedPower` value corresponding to the value for if its selector bit, which is a bit from magnitude, is 1. This is shown in the circuit diagram in Figure 4.

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

Table 2: Mapping of number of leading zeroes to the corresponding exponent value. This mapping is used to create the code below and circuit diagram in Figure 4.

```
------------------------countZeroes Module Code----------------------
always @(*) begin
  if(magnitude[12]== 1'b1 ||
          magnitude[11] == 1'b1) unroundedPower = 3'b111;
  else if(magnitude[10] == 1'b1) unroundedPower = 3'b110;
  else if(magnitude[9] == 1'b1)  unroundedPower = 3'b101;
  else if(magnitude[8] == 1'b1)  unroundedPower = 3'b100;
  else if(magnitude[7] == 1'b1)  unroundedPower = 3'b011;
  else if(magnitude[6] == 1'b1)  unroundedPower = 3'b010;
  else if(magnitude[5] == 1'b1)  unroundedPower = 3'b001;
  else                           unroundedPower = 3'b000;
end
-----------------------------------------------------------------------
```

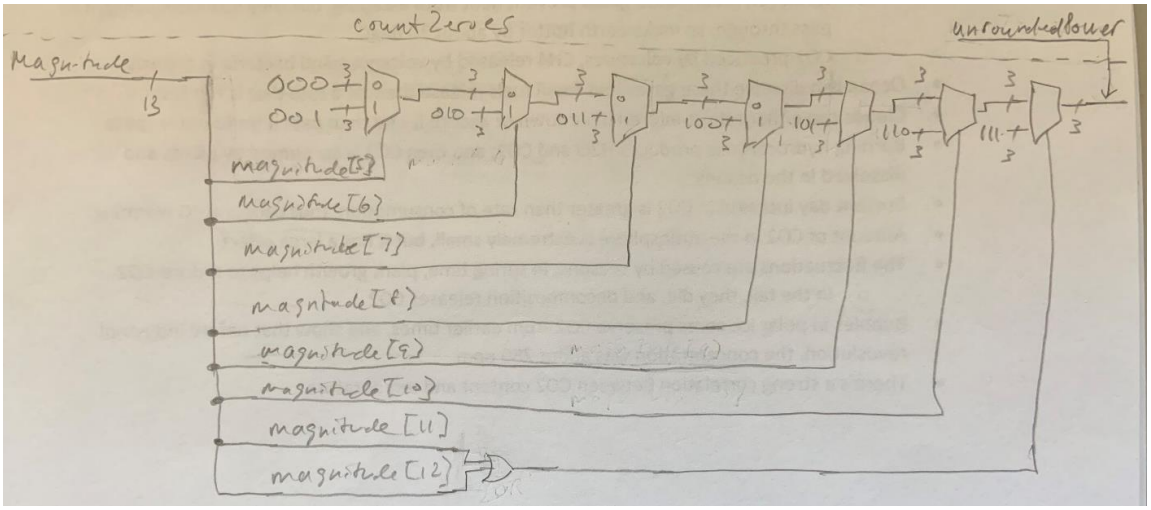

Figure 4: A circuit diagram of the `countZeroes` module. It takes in a 13 bit input `magnitude`, and then uses a series of 2 to 1 3 bit multiplexers to count the number of leading zeroes, outputting a 3 bit result `unroundedPower`, which directly corresponds to the number of leading zeroes as given in Table 2.

getMantissa Module

The function of this module is to extract a 5 bit significand (mantissa) from the 13 bit magnitude of the input. The 5 bits to extract are the first 5 bits after all the leading zeroes, and by doing this, it is guaranteed that for most cases, the most significant bit of the mantissa will be 1, as required by the specification. The only case where this will not occur is if the magnitude is less than 16, but in this case, there is only 1 possible floating point representation for such a number, so the value of the mantissa's most significant bit does not matter. The module achieves its function by taking in 3 bit input `unroundedPower` from `countZeroes` module and a 13 bit input magnitude. Using Table 2, there is a direct mapping between the value of `unroundedPower` and number of leading zeroes, and using that information, the module knows which 5 bit range to extract from `magnitude` as the unrounded significand.

As shown in the Verilog code below, the module was implemented using a case statement on `unroundedPower`. When synthesized, this code was translated into an 8 to 1 multiplexer, where the 3 bit input `unroundedPower` was used the selection bits and there were 8 5 bit inputs to the multiplexer, representing different 5 bit "chunks" of the `magnitude` input.

```
-----------------------getMantissa Module Code----------------------
always @(*) begin
  case(unroundedPower)
    3'b000: unroundedF = magnitude[4:0];
    3'b001: unroundedF = magnitude[5:1];
    3'b010: unroundedF = magnitude[6:2];
    3'b011: unroundedF = magnitude[7:3];
    3'b100: unroundedF = magnitude[8:4];
    3'b101: unroundedF = magnitude[9:5];
    3'b110: unroundedF = magnitude[10:6];
    3'b111: unroundedF = magnitude[11:7];
  endcase
end
--------------------------------------------------------------------
```

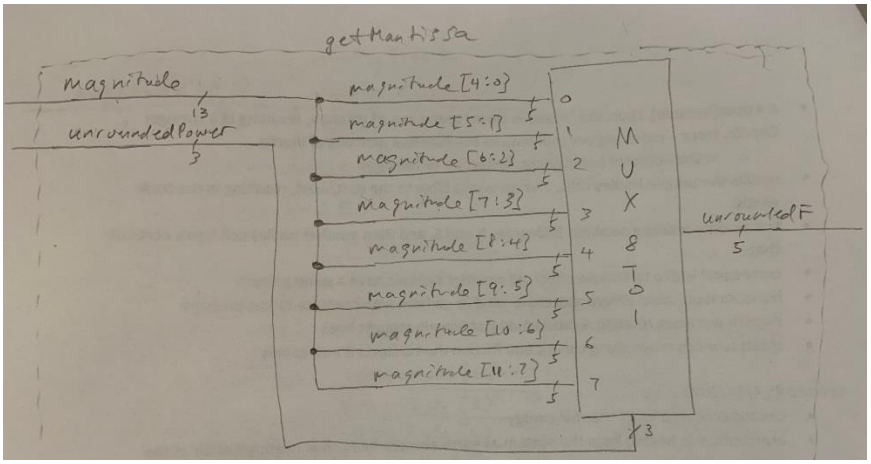

Figure 5: Circuit diagram of `getMantissa` module. This module uses an 8 to 1 multiplexer to pick a 5 bit "chunk" from magnitude depending on the value of `unroundedPower`, which is used as the selector for the multiplexer.

getSixthBit Module

The function of this module is to extract the next bit immediately following the 5 bits extracted for the significand so that it can be used later for rounding purposes. Thus, its design is extremely similar to the `getMantissa` module. It takes a 13 bit input magnitude and a 3 bit input `unroundedPower`, and using `unroundedPower`, it knows how many leading zeroes there are, and thus where the sixth bit after the last leading zero is, which it outputs as `sixthBit`. This module was implemented in Verilog by using a case statement on `unroundedPower`, and can be represented by a circuit diagram as an 8 to 1 multiplexer, where the 8 inputs are bits from various positions of `magnitude` and the selector is `unroundedPower`.

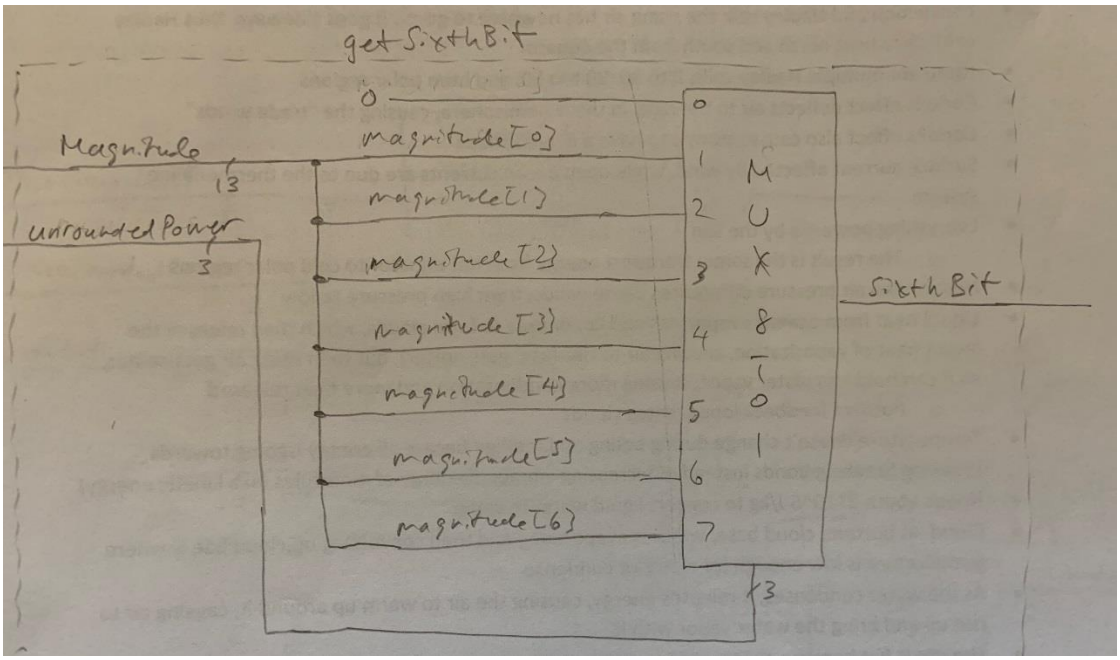

Figure 5: Circuit diagram of the `getSixthBit` module. The module uses an 8 to 1 multiplexer to pick between 8 different bits of the 13 bit input `magnitude`, and the 3 bit input `unroundedPower` determines which bit from `magnitude` to output.

```
------------------------getSixthBit Module Code-----------------------
always @(*) begin
  case(unroundedPower)
    3'b000: sixthBit = 1'b0; // no need to round
    3'b001: sixthBit = magnitude[0];
    3'b010: sixthBit = magnitude[1];
    3'b011: sixthBit = magnitude[2];
    3'b100: sixthBit = magnitude[3];
    3'b101: sixthBit = magnitude[4];
    3'b110: sixthBit = magnitude[5];
    3'b111: sixthBit = magnitude[6];
  endcase
end
----------------------------------------------------------------------
```

roundEF Module

The function of this module is to carry out any needed rounding of unrounded power and significand values as determined by the value of the sixth bit after the last leading 0 in the magnitude. It must also handle cases of overflow, which will be described in detail. It does this by taking in a 3 bit input `unroundedPower` from the `countZeroes` module, 5 bit input `unroundedF` from the `getMantissa` module, and 1 bit input `sixthBit` from the `getSixthBit` module, and outputting a 3 bit value `roundedPower` and 5 bit value `roundedF`.

The overall rounding rule is that if the sixth bit is 1, then the significand should be rounded up by 1. This helps the resulting floating point value be closer to the actual two's complement value that was passed in. However, if the significand is already 31 (5'b11111), then rounding it up would result in overflow, and so in this case, the significand is rounded up and then divided by 2 to have it fit, and the divide by 2 is compensated for by increasing the value of exponent to 1. However, if the exponent is also already at its maximum value, 7 (3'b111), then this indicates that the maximum possible magnitude of the floating point representation is currently being expressed, and it is not possible to increase it any more. In this case, where both the exponent and significand would overflow, no rounding will occur even if the sixth bit is 1'b1.

To implement this set of rounding rules in Verilog, a series of if-else statements were used, starting from the most restrictive case to the least restrictive. In the code below, where the inputs are 3 bit `unroundedPower`, 5 bit `unroundedF`, and 1 bit `sixthBit`, and the outputs are 3 bit `roundedPower` and 5 bit `roundedF`, it first checks if `unroundedF` is 5'b11111 via a bit-reduction AND, `sixthBit` is 1'b1, and `unroundedPower` is 3'b111. If so, then rounding will cause overflow, and thus no rounding should occur. The next less restrictive condition is if `unroundedF` is 5'b11111 and `sixthBit` is 1'b1 but `unroundedPower` is not 3'b111: then rounding up will result in `unroundedF` overflowing, so we increment `unroundedPower` by 1 using a 3 bit adder to get `roundedPower` and set `roundedF` to 5'b10000. Next, if `sixthBit` is 1'b1, then we should increment `unroundedF` by 1 using a 5 bit adder to get `roundedF` and preserve the value of `unroundedPower` and output it as `roundedPower`. Finally, if `sixthBit` is 1'b0, then no rounding should occur and we simply pass on both `unroundedPower` and `unroundedF` as their rounded versions.

(project report continues on next page)

```
-------------------------roundEF Module Code-------------------------
always @(*) begin
  if(&unroundedF == 1'b1 && sixthBit == 1'b1) begin
    if(unroundedPower == 3'b111) begin
      roundedPower = unroundedPower;
      roundedF = unroundedF;
    end
    else begin
      roundedPower = unroundedPower + 3'b1;
      roundedF = 5'b10000;
    end
  end
  else if (sixthBit == 1'b1) begin
    roundedPower = unroundedPower;
    roundedF = unroundedF + 1'b1;
  end
  else begin
    roundedPower = unroundedPower;
    roundedF = unroundedF;
  end
end
-----------------------------------------------------------------------
```

The circuit diagram for this module is shown below.

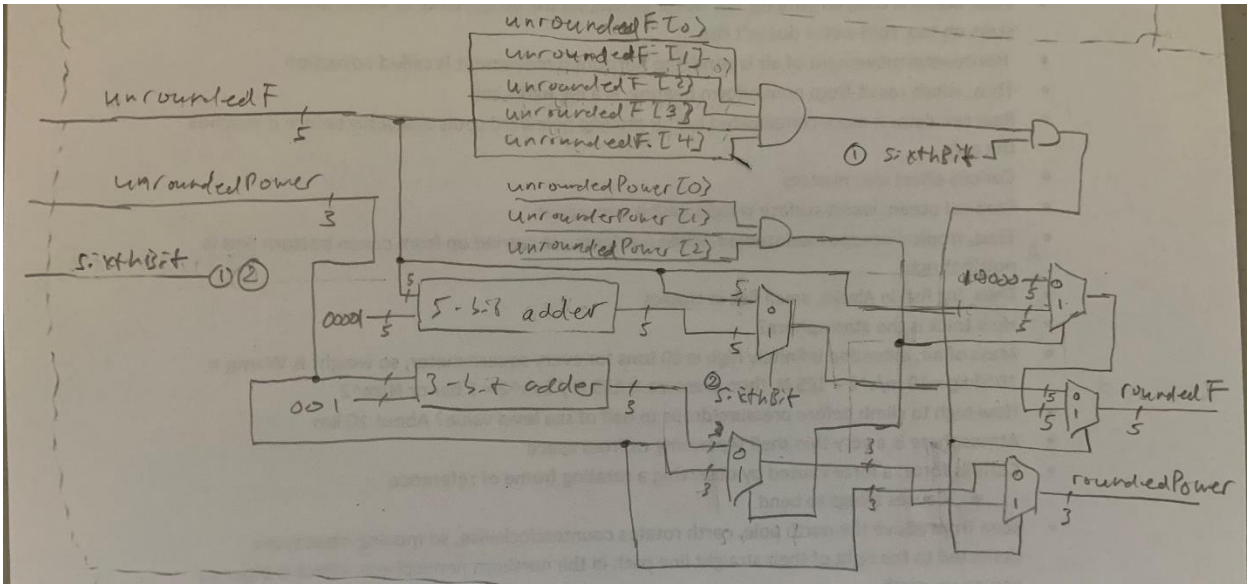

Figure 6: Circuit diagram for the `roundEF` module. The branching behavior is implemented using 2 to 1 multiplexers, which choose between `roundedF` being `unroundedF`, 5'b10000, or `unroundedF` + 1'b1. `roundedPower` is chosen to be either left unrounded or incremented by 1. These choices are all made based on the value of `sixthBit`, if `unroundedF` is 5'b11111 (the five input AND gate at the top) and if `unroundedPower` is 3'b111 (the 3 input AND gate).

detect4096 Module

The function of this module is to detect if the magnitude of the number passed in is equal to 4096. Special attention is given to the value 4096 because it is an edge case that requires additional handling to ensure the correct floating point representation is outputted. The details of why are given in the description of the `handle4096` module. This module achieves its desired functionality by taking in a 13 bit input `magnitude` from the `getMagnitude` module and checking if is equal to 13'b100000000000, outputting the result as a 1 bit result `detected4096`. At the gate level, it is simple, only using AND, NOR, and NOT gates. NOR gates were used for simplicity, as 12 out of the 13 bits in 4096 are 0. In the interest of keeping the fan-in low to be more realistic and better suited for real world implementation, a maximum fan-in of 4 was used for the gates for the circuit diagram. This does not necessarily correspond with how the circuit will be synthesized.

```
------------------------detect4096 Module Code-----------------------
always @(*) begin
  if(magnitude == 13'b1000000000000)
    detect4096 = 1'b1;
  else
    detect4096 = 1'b0;
end
---------------------------------------------------------------------
```

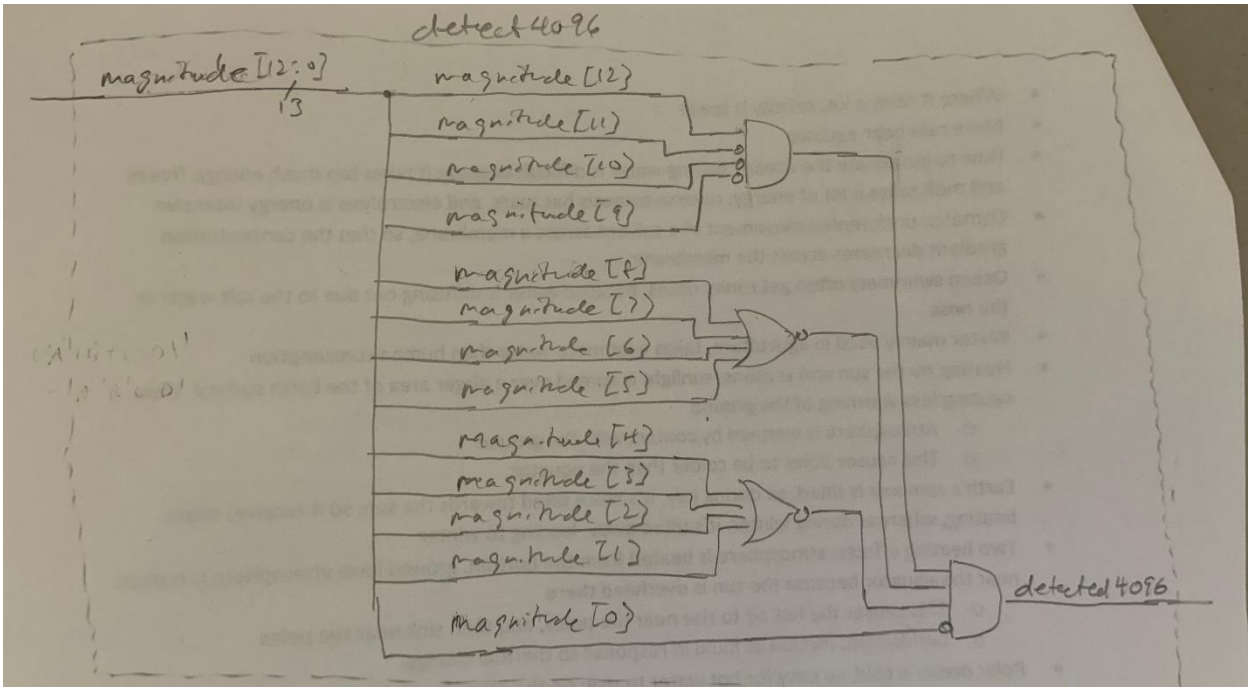

Figure 7: Circuit diagram of the `detect4096` module. The module takes in a 13 bit magnitude and checks bit by bit if it is equal to 13'b1000000000000, outputting the result as detected4096.

handle4096 Module

The purpose of this module is to handle the special case of the input having a magnitude of 4096. To do this, it takes in from the `roundEF` module a 3 bit input `roundedPower` and a 5 bit input `roundedF`, as well as a 1 bit input `detected4096` from the `detect4096` module. It outputs a 3 bit value `E` and a 5 bit output value `F`, which are the outputs of the same name of the top level module. This module is needed because a number with magnitude 4096 does not give the correct value after passing through all modules up to and including the `roundEF` module. A magnitude of 4096 arises from the input –4096, and thus the output should be [1 111 11111]. However, after negating –4096 to get a magnitude of 4096 (13'b1000000000000 in unsigned binary), the modules see that 4096 has no leading zeroes, so the unrounded exponent is 3'b111, the unrounded significand is 5'b10000, and the sixth bit is 0. The sixth bit is 0, so no rounding is performed, and thus the `roundEF` module outputs the incorrect answer. Thus, `handle4096` performs the necessary postprocessing of values to account for an input of –4096. It does this by using two 2 to 1 multiplexers, with `detect4096` being the selection bit, to choose between outputting `roundedPower` or 3'b111 and `roundedF` or 5'b11111. These are outputs are directly passed on to be the outputs of the top level `FPCVT` module. In Verilog code, this is implemented using an if-statement, and Figure 8 below shows the circuit diagram.

```
------------------------handle4096 Module Code----------------------
always @(*) begin
  if(detected4096 == 1'b1) begin
    E = 3'b111;
    F = 5'b11111;
  end
  else begin
    E = roundedPower;
    F = roundedF;
  end
end
--------------------------------------------------------------------
```

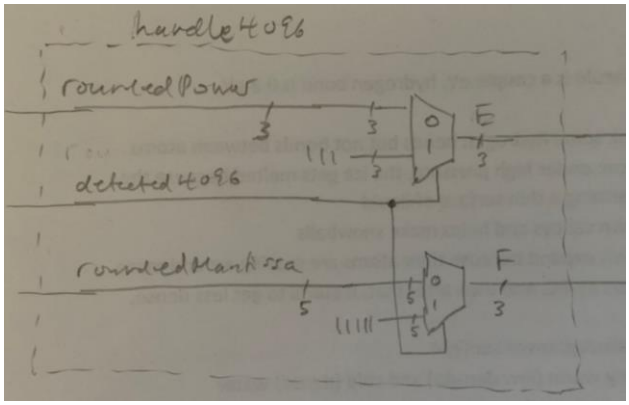

Figure 8: Circuit diagram of the handle4096 module. It takes in `roundedPower`, `roundedMantissa`, and `detected4096`, and then uses `detected4096` to determine whether to output `roundedPower` and `roundedMantissa` unchanged, or to output 3'b111 and 5'b11111 for the case of the `magnitude` being 4096.

Simulation Documentation

The requirement is that the top level `FPCVT` module outputs the appropriate 9 bit floating point representation of a 13 bit two's complement input. The floating point representation's value may not match the input value exactly, so appropriate rounding rules should be applied. For testing, the `FPCVT` module with all its submodules was tested as a whole, and no submodules were individually tested. It would have been better to test each submodule individually before putting them together, but due to the relatively simple ability to verify the overall output, only the top level module was tested.

The testing consisted of 2 phases: first, a variety of edge cases were tested manually to ensure the module worked as intended. The cases used for testing are listed in Table 3 below. Then, as there are only 8192 possible inputs, it is feasible to test them all, so automated testing was implemented to check the output for all possible inputs.

To test these cases manually in the test bench, I simply set `D` to these values every 100 ns in an always block, as shown in the code extract below:

```
--------------Test Bench Code for FPCVT Module (Manual)---------------
initial begin
  D = 13'b0;
  #100;
end

always begin
  #100;
  D = 0;
  #100;
  D = 1;
  #100;
  D = 8;
  #100;
  D = 16;
  #100;
  D = 31;
  // and so on for all values in Table 3, including negative numbers
end
-----------------------------------------------------------------------
```

(project report continues on next page)

Test Input	Expected Result	Reason for Testing
0 (13'b0000000000000)	[0 000 00000]	All 0's
1 (13'b0000000000001)	[0 000 00001]	`E` = 0 and `F` = 1
8 (13'b0000000001000)	[0 000 01000]	More than 8 leading zeroes
16 (13'b0000000010000)	[0 000 10000]	Exactly 8 leading zeroes
31 (13'b0000000011111)	[0 000 11111]	Last number where exponent is 0
32 (13'b0000000100000)	[0 001 10000]	Exactly 32: first number where exponent is not 0
33 (13'b0000000100001)	[0 001 10001]	Rounding: sixth bit is 1, so 5'b10000 should be rounded up to 5'b10001
34 (13'b0000000100010)	[0 001 10001]	Check rounding does not occur, as sixth bit is 0
63 (13'b0000000111111)	[0 010 10000]	Rounding with overflow: sixth bit is 1'b1, previous 5 bits are 5'b11111, so significand should be 5'b10000, exponent should be incremented from 3'b001 to 3'b010
3967 (13'b0111101111111)	[0 111 11111]	Approaching the maximum possible value that can be represented by the floating point representation; check for only 1 leading 0, sixth bit is 1 so significand should be rounded from 5'b11110 up to 5'b111111
3968 (13'b0111110000000)	[0 111 11111]	The maximum value that can be represented by the floating point representation
4032 (13'b0111111000000)	[0 111 11111]	A number above the maximum floating point value, where the sixth bit is 1'b1, but because exponent is 3'b111 and significand is 5'b11111, no rounding occurs
4095 (13'b0111111111111)	[0 111 11111]	The maximum possible two's complement output
–4096 (13'b1000000000000)	[1 111 11111]	Special case where the magnitude is 13'b1000000000000, which requires special handling; check sign bit is 1
–4095 (13'b1000000000001)	[1 111 11111]	Check that the negative of the largest possible two's complement input (4095) is handled properly; check sign bit is 1
–4032 (13'b1000001000000)	[1 111 11111]	Same reasoning as 4032 case, but also sign bit should be 1
–3968 (13'b1000010000000)	[1 111 11111]	Same reasoning as 3968 case, but also sign bit should be 1
…test negative counterparts of the remaining positive numbers above…		Same reasoning as the positive cases but also sign bit should be 1

Table 3: Edge cases for converting numbers from 13 bit two's complement to floating point. They aim to cover various cases of rounding, including where the significand and exponent may overflow, as well inputs at the extreme edges (minimum and maximum) of the two's complement and floating point representations.

After running the simulations in Isim, the following waveforms below in Figures 9 and 10 were generated for the manual test cases.

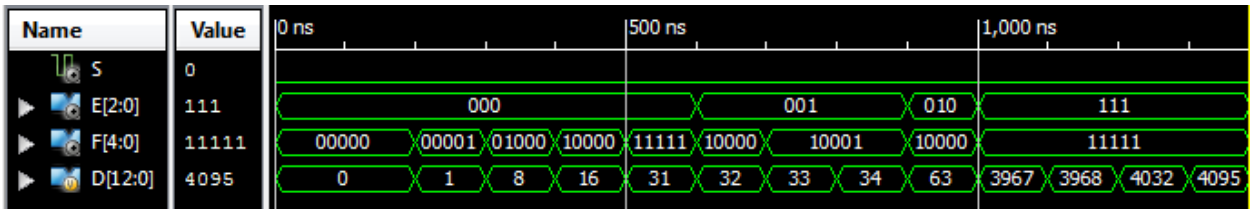

Figure 9: Simulation waveform results for the positive inputs listed in Table 3. Comparing them to the expected values show that they all match, thus verifying the design for these test cases.

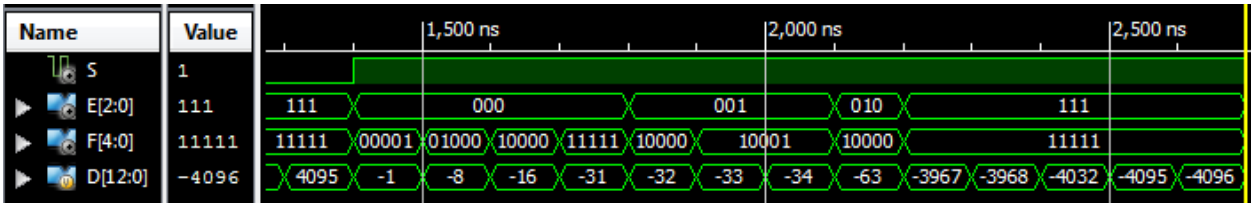

Figure 10: Simulation waveform results for negative inputs listed in Table 3. Comparing them to the expected values show that they all match, thus verifying the design for these test cases.

(project report continues on next page)

Next, to perform automated testing, a Python script was written to generate the expected results for all possible inputs according to the conversion instructions given in the project document. These results were saved in a file called `progConversion.txt`. Then, test bench code was written to open this file. As the test bench looped through all the possible values of the input `D`, the resulting outputs `S`, `E`, and `F` were checked against the expected output in the text file by using an assertion macro. If any did not match, a message was outputted to the ISim console with details of which input failed. The test bench code used to do this is given below:

```
------------Test Bench Code for FPCVT Module (Automatic)--------------
// macro to define an assert function
`define assert(signal, value, testcase) \
  if (signal !== value) begin \
  $display("ASSERTION FAILED in %m for %d: %b != %b", \
           $signed(testcase), signal, value); \
  end

integer file; // Variables for file IO
integer scan_file;
reg [12:0] data;

initial begin
  file = $fopen("progConversion.txt", "r");
  if(!file) $display("Failed to open file");
  else      $display("Successfully opened file");
  D = 13'b0;
  #100;
end

always begin
  //automated testing for all 8192 possible inputs
  #100;
  scan_file = $fscanf(file, "%b\n", data);
  `assert(D, data, D);
  scan_file = $fscanf(file, "%b\n", data);
  `assert(S, data, D);
  scan_file = $fscanf(file, "%b\n", data);
  `assert(E, data, D);
  scan_file = $fscanf(file, "%b\n", data);
  `assert(F, data, D);

  D = D + 1'b1;
  if(D == 0) begin
    $fclose(file);
    $finish;
  end
end
-------------------------------------------------------------------------
```

This test bench code was run and no error messages were produced, thus indicating that all test cases were passed; the module's output matched what was expected. However, one must be careful to note that this is not a foolproof way to test, as it is possible that in both the Verilog code and Python script, the same error was made, causing a test case to pass where it should fail. However, all 8192 cases matching for `S`, `E`, and `F` gives us high confidence that the module is implemented correctly.

After synthesis, the text report indicated that no registers were used, which makes sense, as this is a purely combinational circuit. 60 slice LUTs were used out of 9112 available, and out of the 60, all of them had an unused flip flop, which makes sense, as this is a combinational circuit. After advanced HDL synthesis, there were 3 adders and 16 multiplexers used. The number of multiplexers used differs slightly from the number shown on the circuit diagrams due to optimizations made during the synthesis step. The details of each component and their presumed use are listed in Table 4 below. There are 22 bonded IOBs, which corresponds to the 13 bit input and 9 bit output, which is 9% of the 232 bonded IOBs available. Detailed extracts from the synthesis report are included in Appendix A at the end of this report.

Combinational Element	Quantity	Corresponding Module
13 bit adder	1	getMagnitude
3 bit adder	1	roundEF, for rounding exponent
5 bit adder	1	roundEF, for rounding significand
1 bit 8 to 1 multiplexer	1	getSixthBit
13 bit 2 to 1 multiplexer	1	getMagnitude
3 bit 2 to 1 multiplexer	9	countZeroes, roundEF, handle4096
5 bit 2 to 1 multiplexer	4	roundEF, handle4096
5 bit 8 to 1 multiplexer	1	getMantissa

Table 4: Adder and multiplexer usage by the synthesized circuit. The details of how each component are used are described in its corresponding module's section in the Design section of this report.

After implementing the design, the mapping and place & route reports show the number of slice LUTs used was reduced down to 50 of 9112. Of these 50 slice LUTs, all of them are used as logic and none are used as memory, which makes sense for a combinational circuit. Of the 50 LUT flip flop pairs used, all 50 have an unused flip flop. The number of bonded IOBs remains the same, at 22 out of 232. Detailed extracts from the mapping report are included in Appendix B at the end of this report. The place & route report has the exact same information concerning slice LUTs, so it is not included in this report.

Overall, the LUT usage is low for the synthesis and mapping reports, so this is a feasible design to implement on the actual board.

Conclusion

In this lab, a combinational module to convert a 13 bit two's complement number to a 9 bit floating point number with a 1 bit sign, 3 bit exponent, and 5 bit mantissa was implemented using multiple submodules that were then linked together to form the top module. The module was then manually tested with selected edge cases and automatically tested with all 8192 possible input cases, and all passed, thus verifying the design. From this lab, I learned how to write submodules in Verilog and combine them together into other modules. Also, I learned how to do automatic testing of modules in test benches through the use of an assertion macro, as well as opening a file to read data from and write data to.

The main difficulty I encountered was figuring out how to incorporate multiple submodules into one top level module. This wasn't really covered in class, but the slides on CCLE showed me how to do this. I also initially kept running into errors with using wires where I should be using registers and the other way around. The distinction between the two wasn't covered very well in class, so I did my own research to figure out the difference between the two and when it is legal to use each one and how to use them together as module inputs and outputs.

Overall, the lab was quite straightforward with a clear project specification, so it was easy to understand and implement. The only suggestion would be that the specification mentions that very large linear encodings should be encoded with the largest possible floating point representation, but it does not explicitly say what to do for very negative linear encodings. It was easy to figure out that I should simply use the most negative floating point representation, but having it clear would be nice.

Appendix

Note: Only sections pertaining to slice registers, slice LUTs, gates, and IO are included. Duplicate information within the same report may be omitted (for example, if "HDL Synthesis Report" and "Advanced HDL Synthesis Report" are the same).

Appendix A: Synthesis Report

The warnings in the HDL Synthesis section are due to the fact that the full 13 bits of `magnitude` from the `getMagnitude` module were passed in to all the modules, but not all modules made use of all 13 bits, thus causing the warning to appear.

```
=========================================================================
*                            HDL Synthesis                              *
=========================================================================
Synthesizing Unit <FPCVT>.
    Related source file is "C:\CS M152A\Project2\FPCVT.v".
    Summary:
      no macro.
Unit <FPCVT> synthesized.

Synthesizing Unit <getMagnitude>.
    Related source file is "C:\CS M152A\Project2\FPCVT.v".
    Found 13-bit adder for signal <inVal[12]_GND_2_o_add_2_OUT>
created at line 155.
    Summary:
      inferred   1 Adder/Subtractor(s).
      inferred   1 Multiplexer(s).
Unit <getMagnitude> synthesized.

Synthesizing Unit <countZeroes>.
    Related source file is "C:\CS M152A\Project2\FPCVT.v".
WARNING:Xst:647 - Input <magnitude<4:0>> is never used. This port will
be preserved and left unconnected if it belongs to a top-level block
or it belongs to a sub-block and the hierarchy of this sub-block is
preserved.
    Summary:
      inferred   6 Multiplexer(s).
Unit <countZeroes> synthesized.

Synthesizing Unit <getMantissa>.
    Related source file is "C:\CS M152A\Project2\FPCVT.v".
WARNING:Xst:647 - Input <magnitude<12:12>> is never used. This port
will be preserved and left unconnected if it belongs to a top-level
block or it belongs to a sub-block and the hierarchy of this sub-block
is preserved.
```

```
    Found 5-bit 8-to-1 multiplexer for signal <mantissa> created at
line 105.
    Summary:
      inferred   1 Multiplexer(s).
Unit <getMantissa> synthesized.

Synthesizing Unit <getSixthBit>.
    Related source file is "C:\CS M152A\Project2\FPCVT.v".
WARNING:Xst:647 - Input <magnitude<12:7>> is never used. This port
will be preserved and left unconnected if it belongs to a top-level
block or it belongs to a sub-block and the hierarchy of this sub-block
is preserved.
    Found 1-bit 8-to-1 multiplexer for signal <sixthBit> created at
line 136.
    Summary:
      inferred   1 Multiplexer(s).
Unit <getSixthBit> synthesized.

Synthesizing Unit <roundEF>.
    Related source file is "C:\CS M152A\Project2\FPCVT.v".
    Found 3-bit adder for signal <unroundedPower[2]_GND_6_o_add_4_OUT>
created at line 83.
    Found 5-bit adder for signal <unroundedF[4]_GND_6_o_add_8_OUT>
created at line 89.
    Summary:
      inferred   2 Adder/Subtractor(s).
      inferred   5 Multiplexer(s).
Unit <roundEF> synthesized.

Synthesizing Unit <is4096>.
    Related source file is "C:\CS M152A\Project2\FPCVT.v".
    Summary:
      no macro.
Unit <is4096> synthesized.

Synthesizing Unit <handle4096>.
    Related source file is "C:\CS M152A\Project2\FPCVT.v".
    Summary:
      inferred   2 Multiplexer(s).
Unit <handle4096> synthesized.

=========================================================================
                        HDL Synthesis Report
=========================================================================
Macro Statistics
# Adders/Subtractors                                   : 3
 13-bit adder                                          : 1
 3-bit adder                                           : 1
```

```
 5-bit adder                                            : 1
# Multiplexers                                          : 16
 1-bit 8-to-1 multiplexer                               : 1
 13-bit 2-to-1 multiplexer                              : 1
 3-bit 2-to-1 multiplexer                               : 9
 5-bit 2-to-1 multiplexer                               : 4
 5-bit 8-to-1 multiplexer                               : 1

=========================================================================
*                           Low Level Synthesis                         *
=========================================================================
Optimizing unit <FPCVT> ...

Optimizing unit <getMagnitude> ...

Mapping all equations...
Building and optimizing final netlist ...
Found area constraint ratio of 100 (+ 5) on block FPCVT, actual ratio
is 0.

=========================================================================
*                            Design Summary                             *
=========================================================================
Top Level Output File Name         : FPCVT.ngc

Primitive and Black Box Usage:
------------------------------
# BELS                             : 89
#      GND                         : 1
#      INV                         : 12
#      LUT1                        : 1
#      LUT2                        : 2
#      LUT3                        : 13
#      LUT4                        : 1
#      LUT5                        : 6
#      LUT6                        : 25
#      MUXCY                       : 12
#      MUXF7                       : 2
#      VCC                         : 1
#      XORCY                       : 13
# IO Buffers                       : 22
#      IBUF                        : 13
#      OBUF                        : 9
```

```
Device utilization summary:
---------------------------

Selected Device : 6slx16csg324-3

Slice Logic Utilization:
 Number of Slice LUTs:                   60  out of   9112     0%
    Number used as Logic:                60  out of   9112     0%

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:     60
   Number with an unused Flip Flop:      60  out of     60   100%
   Number with an unused LUT:             0  out of     60     0%
   Number of fully used LUT-FF pairs:     0  out of     60     0%
   Number of unique control sets:         0

IO Utilization:
 Number of IOs:                          22
 Number of bonded IOBs:                  22  out of    232     9%
================================================================================
```

Appendix B: Mapping Report

```
Design Summary
--------------
Number of errors:      0
Number of warnings:    0
Slice Logic Utilization:
  Number of Slice Registers:                 0 out of  18,224    0%
  Number of Slice LUTs:                     50 out of   9,112    1%
    Number used as logic:                   50 out of   9,112    1%
      Number using O6 output only:          28
      Number using O5 output only:          12
      Number using O5 and O6:               10
      Number used as ROM:                    0
    Number used as Memory:                   0 out of   2,176    0%

Slice Logic Distribution:
  Number of occupied Slices:                16 out of   2,278    1%
  Number of MUXCYs used:                    16 out of   4,556    1%
  Number of LUT Flip Flop pairs used:       50
    Number with an unused Flip Flop:        50 out of      50  100%
    Number with an unused LUT:               0 out of      50    0%
    Number of fully used LUT-FF pairs:       0 out of      50    0%
    Number of slice register sites lost
      to control set restrictions:           0 out of  18,224    0%

IO Utilization:
  Number of bonded IOBs:                    22 out of     232    9%
```