Charles Zhang

28 October 2021

COM SCI M152A, Lab 5

Lab 2: Floating Point Conversion Report

1 Introduction

In this lab, we developed a floating point converter, which took in a 12-bit two's complement value, and returned that value in an 8-bit floating point encoded in the following form:

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

Here, `S` is the sign bit, `E` is a 3-bit exponent, and `F` is a 4-bit significand. In addition to this basic conversion, we also need to handle rounding of values that our floating-point representation is not precise enough to handle. Past our implementation, we also had to implement a testbench that covered a variety of test cases for our module to ensure its correctness.

2 Design Description

For our design of our top module, we followed the recommended design set out by the spec:

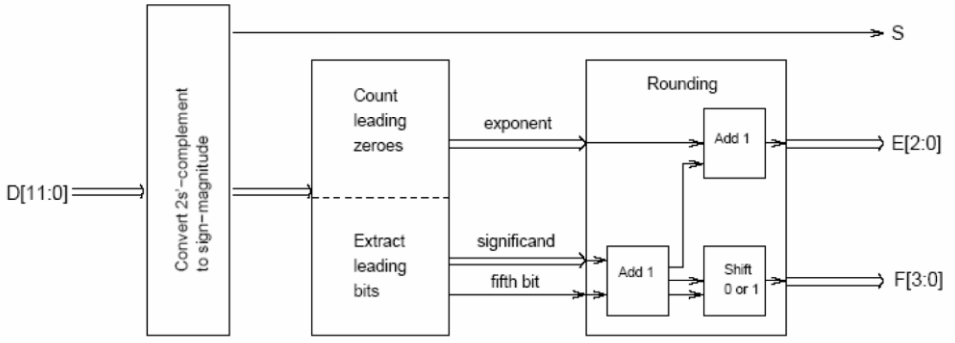

Figure 1: Block diagram of top module `FPCVT.v`

This design is made up of three modules: `to_signed_magnitude`, `leading_zeroes`, and `handle_round`. These are all brought together in our top module `FPCVT`, which takes in a 12-bit linear encoded value `D`, and outputs `S`, `E`, and `F`, the sign bit, exponent, and significand of the floating point representation, respectively.

The first module, `to_signed_magnitude`, is responsible for the extraction of the sign bit and the conversion of the 12-bit two's complement value to a 12-bit signed magnitude value. This module takes in a 12-bit two's complement value `D`, and outputs the sign bit `S` and a 12-bit signed magnitude value `signed_mag`. To implement the module, we check the most significant bit of the input `D`, and assign its value to the sign bit `S`. This `S` is then passed as the output of `FPCVT`. Next, if the sign bit was `1` (`D` is negative), we negate the value of `D` and pass it as the output `signed_mag`. One edge case we handle here is if `D` $= T_{min}$, which cannot be negated, as $|T_{min}|$ cannot be represented in the desired format. Instead, under this condition, we manually set `signed_mag` to the highest possible value that can be stored in a 12-bit signed magnitude format. If the value of `D` wasn't negative, we simply passed on its value to `signed_mag`.

The next module, `leading_zeroes`, is responsible for the counting of leading zeroes in the 12-bit signed magnitude encoding. From that count, we can then determine the unrounded exponent and significand, as well as extract the bit we need to check for rounding. This module takes in a 12-bit signed magnitude value `D`, and outputs a 3-bit unrounded exponent `E`, a 4-bit unrounded significand `F`, and the bit we need to check for rounding `fifth_bit`. To implement this module, we start by iterating through the value from left to right, decrementing the exponent from `8`, until we find a `1`. This allows us to get our output `E`, which is just `8` minus the number of leading zeroes. From there, we simply loop through the next four bits to build our significand, `F`. Finally, we take the bit following the significand for our output `fifth_bit`. Here, we must handle the edge case where `E = 0`, meaning the last four bits of `D` make up `F`. Since there is no bit following the significand, we simply assume that this value will not be rounded in the future and set `fifth_bit` to `0`.

The final module, `handle_round`, is responsible for rounding the values obtained from `leading_zeroes`, according to the spec. This module takes in a 3-bit unrounded exponent

E_in, a 4-bit unrounded significand F_in, and fifth_bit, which is used to check for rounding. This module outputs a 3-bit rounded exponent E_out and a 4-bit rounded significand F_out. For our converter, we round the value such that the magnitude of the value increases if the bit following the significand is 1. To implement this, we started by checking if fifth_bit was equal to 1, which would mean we would need to round. If this was the case, we would check if F_in was equal to 1111. If it was not, we increment it by 1 and output the original exponent with the modified significand. If it was, then incrementing it by 1 would make it overflow, so instead, we attempt to increment the exponent by 1, setting the significand to 1000 to compensate. Then, the modified exponent and significand could be output. An edge case we have to handle here is that it is possible for the exponent to also overflow if E_in = 111. In this case, we simply do not modify the inputs, as if no rounding took place, as the inputs already represent the maximum values representable in our encoding.

3 Simulation Documentation

To test our converter, we developed a testbench that was designed to test all the edge cases we could think of. This included generic cases for positive and negative numbers, rounding cases for positive and negative numbers, T_{min} and T_{max}, 0, −1, cases where the significand overflowed, cases where $E = 0$, and cases where $E \geq 8$. Our simulation could then show us how our top module and submodules behaved in each case.

A generic case in our testbench was D = 000000101100. Here, we can see that this is a positive number with six leading zeroes, which tells us that the exponent is 2 and the significand is 1011. In addition, the bit following the significand was 0, so no rounding needed to occur. The output of our top module is shown below:

Figure 2: Waveform diagram for `FPCVT` for the case `D = 000000101100`

Breaking things down, we can see the waveforms for our first module in the same test case:

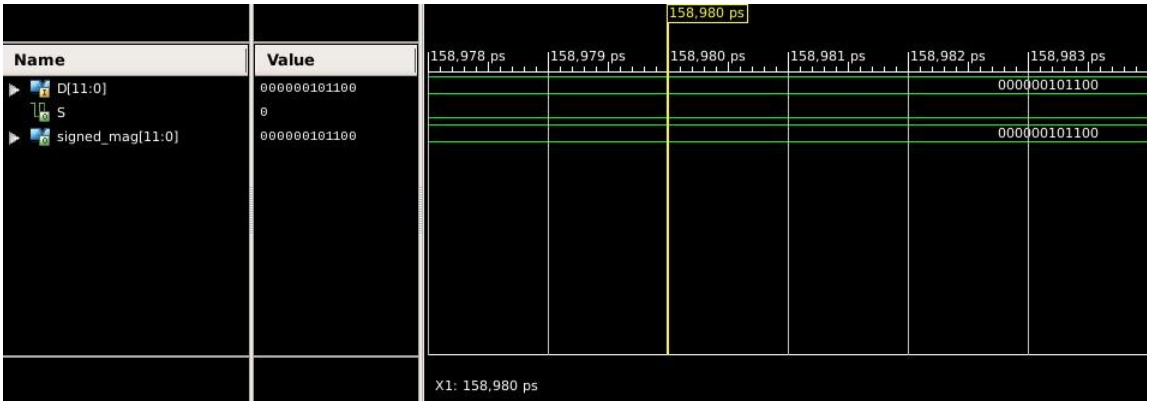

Figure 3: Waveform diagram for `to_signed_magnitude` for the case `D = 000000101100`

In this module, we see that the sign bit `0` was extracted from `D`, and, since `D` was positive, no change needed to occur to convert the value to signed magnitude. We can then look at the waveforms for the next module:

Figure 4: Waveform diagram for `leading_zeroes` for the case `D = 000000101100`

In this module, we can see that we take in the signed magnitude value from the previous module and then translate it to exponent and significand values `E` and `F`. Since there are six leading zeroes, this module tells us that `E = 2`, and then sets `F = 1011`. Finally, the module extracts the bit following the significand, outputting it as `fifth_bit`. Finally, we can then pass these values to the final module:

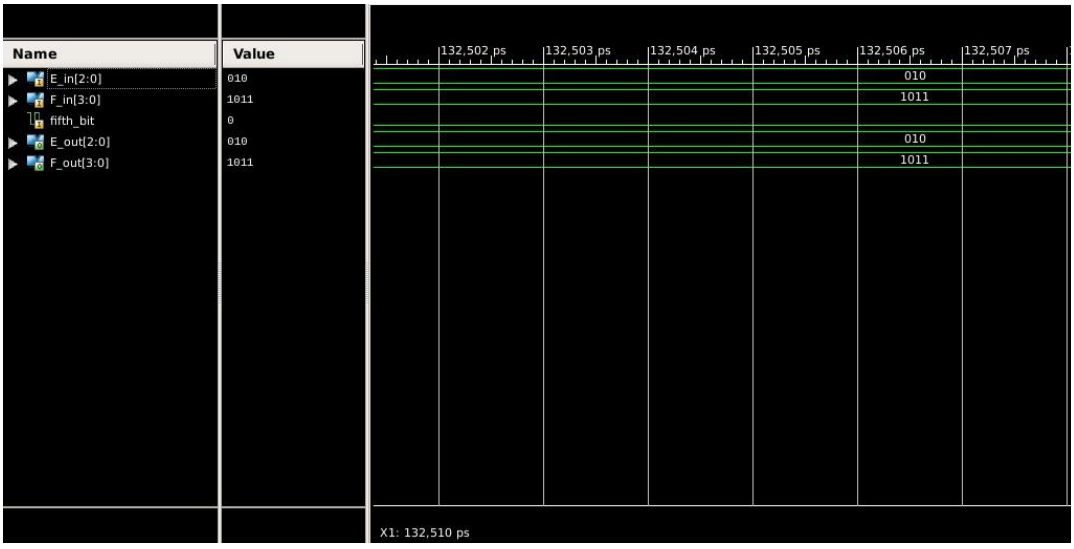

Figure 5: Waveform diagram for `handle_round` for the case `D = 000000101100`

Here, we can see that the module takes in the unrounded `E` and `F` values, as well as the value for `fifth_bit`. Since the value is `0`, the outputted values of `E` and `F` are exactly the same as the inputted ones.

In addition to generic cases, we can see these waveforms for edge cases. One such edge case is when `E = 0`. In this case, we know that the last four bits must be the significand, and there's no bit that tells us if we should round. The output of one such case is shown below:

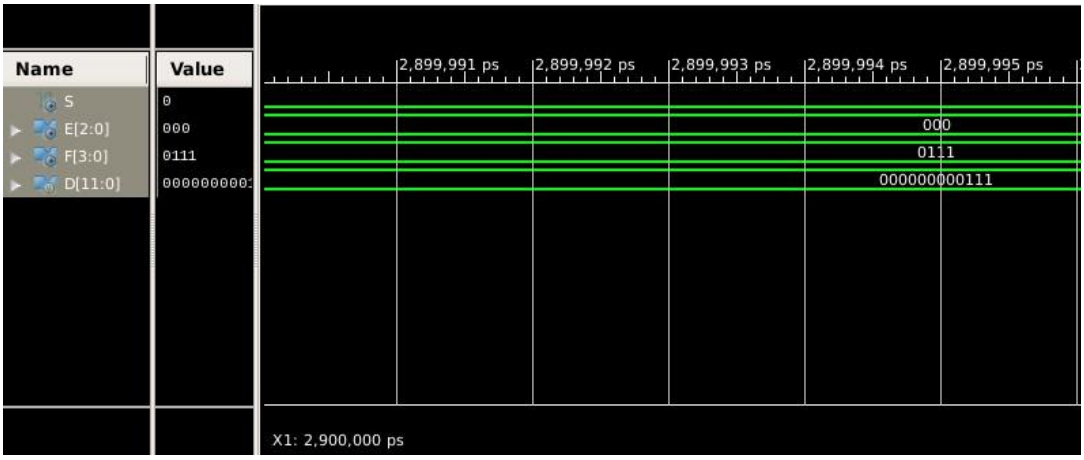

Figure 6: Waveform diagram for `FPCVT` for the case `D = 000000000111`

Here, we can see that our top module properly extracts the last four bits as the significand and sets the exponent to `0`. We can see the start of this process in our first module:

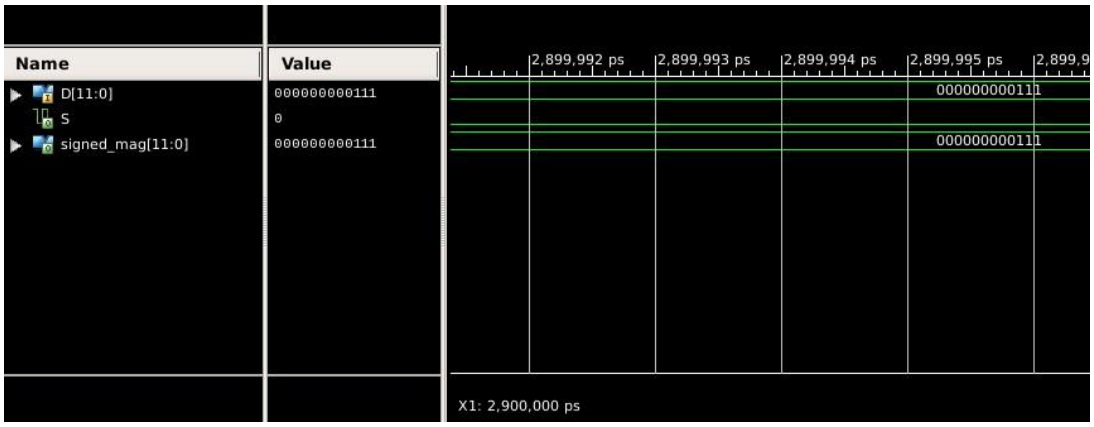

Figure 7: Waveform diagram for `to_signed_magnitude` for the case `D = 000000000111`

Here, we see that the sign bit is extracted properly and that the same value is passed on to `signed_mag`. Next, we can see:

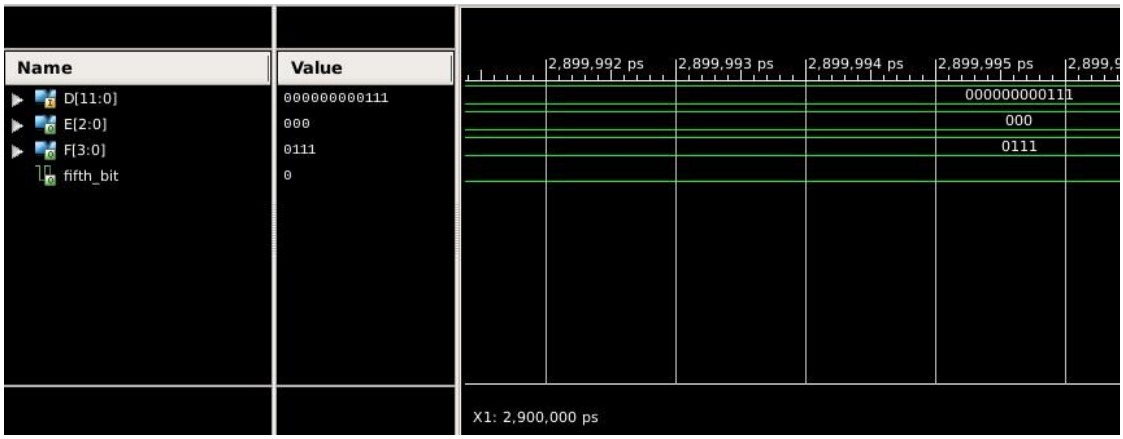

Figure 8: Waveform diagram for `leading_zeroes` for the case `D = 000000000111`

In this waveform, we see that `E` was properly set to `0` without error, and that `0111` was selected as the significand, despite it starting with a `0` bit. We can also see that, despite there being no bits behind the significand, `fifth_bit` is still set to `0`, telling us not to round the value. Finally, we can see our last module:

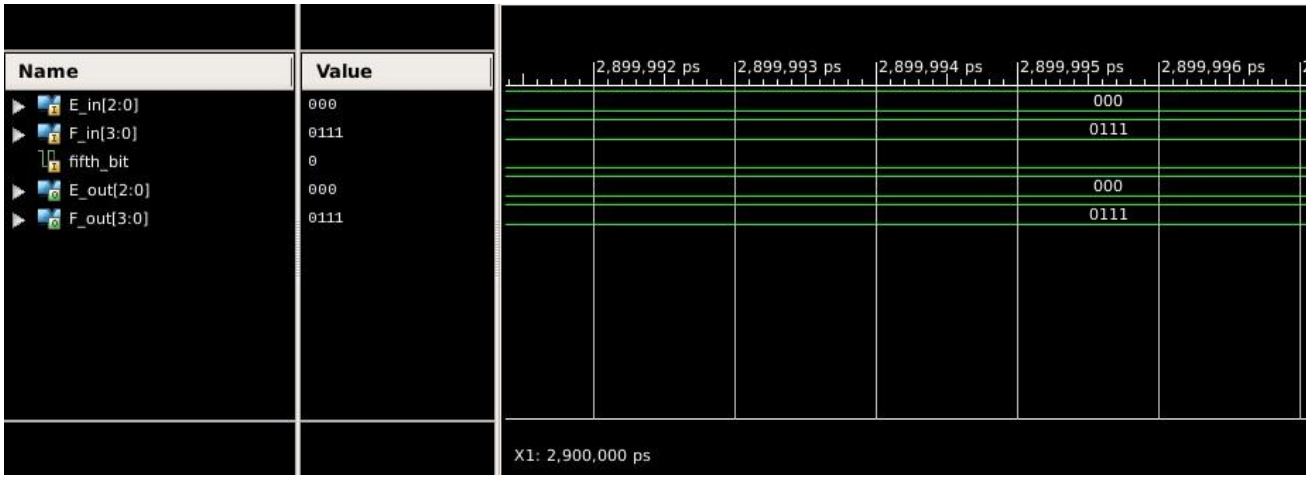

Figure 9: Waveform diagram for `handle_round` for the case `D = 000000000111`

In these waveforms, we can see that `fifth_bit` is `0`, and therefore, the inputted `E` and `F` values are simply outputted.

Another possible edge case is T_{min}, or `D = 10000000000`. Here, we cannot simply negate the number to convert it to signed magnitude, as it will overflow. The top module's waveform is given below:

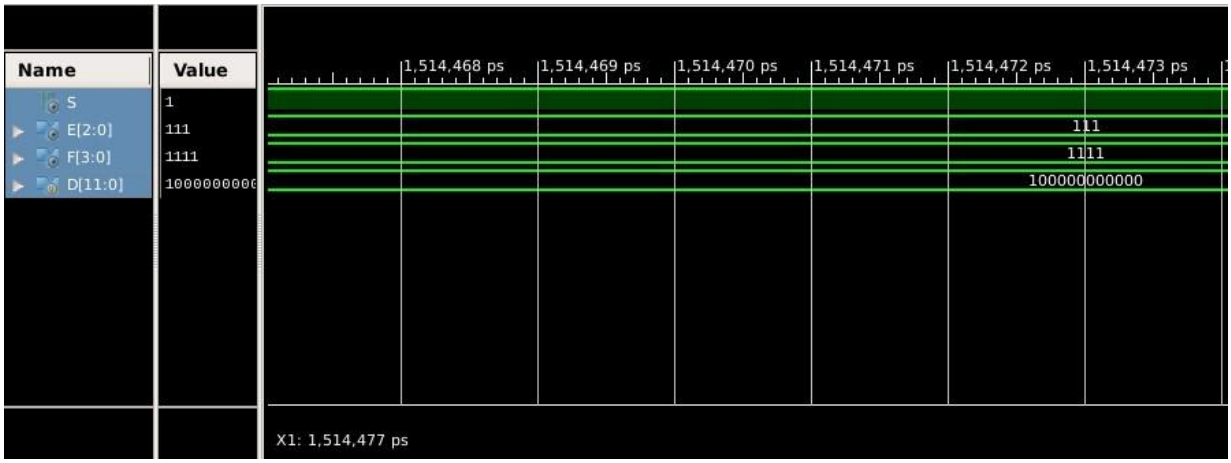

Figure 10: Waveform diagram for `FPCVT` for the case `D = 10000000000000`

We can see from the waveforms that the desired output is simply the maximum magnitude we can represent with our floating point encoding. To see how we get there, we can look at the waveform for our first module:

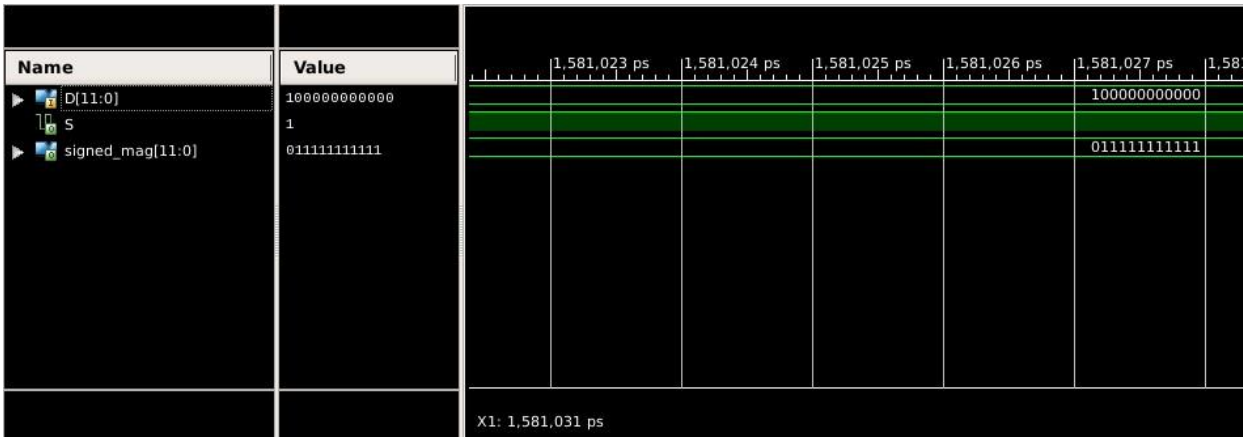

Figure 11: Waveform diagram for `to_signed_magnitude` for the case `D = 100000000000`

Here, we see that we properly extract the sign bit, `1`. In addition, since the value is negative, we see that the module tries to negate it. However, since it overflows, our module manually sets the

signed magnitude encoding to the highest possible value: `011111111111`. We then pass this to the next module:

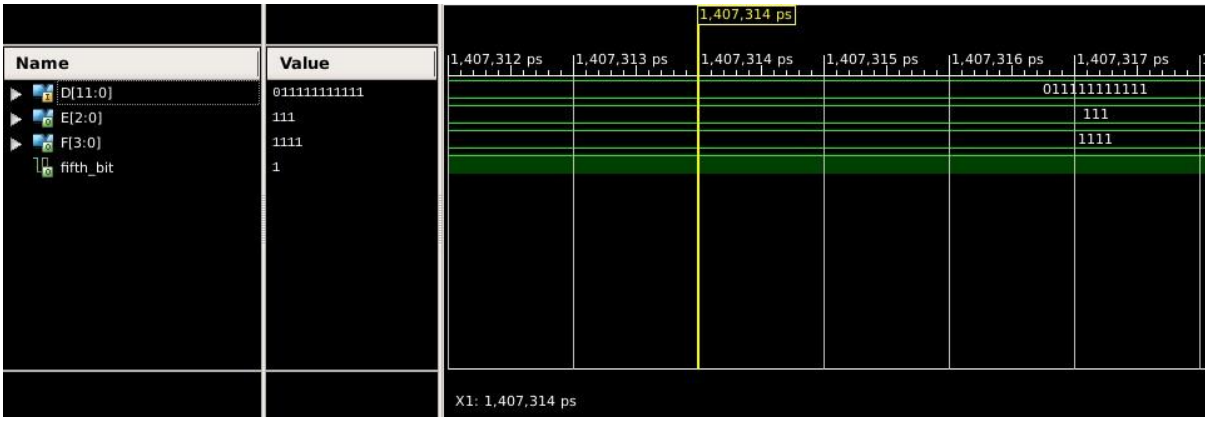

Figure 12: Waveform diagram for `leading_zeroes` for the case `D = 100000000000`

Here, we successfully calculate that the exponent is `7` and that the significand is `1111`. We also correctly set the `fifth_bit` to `1`, which can then be used in the following waveforms:

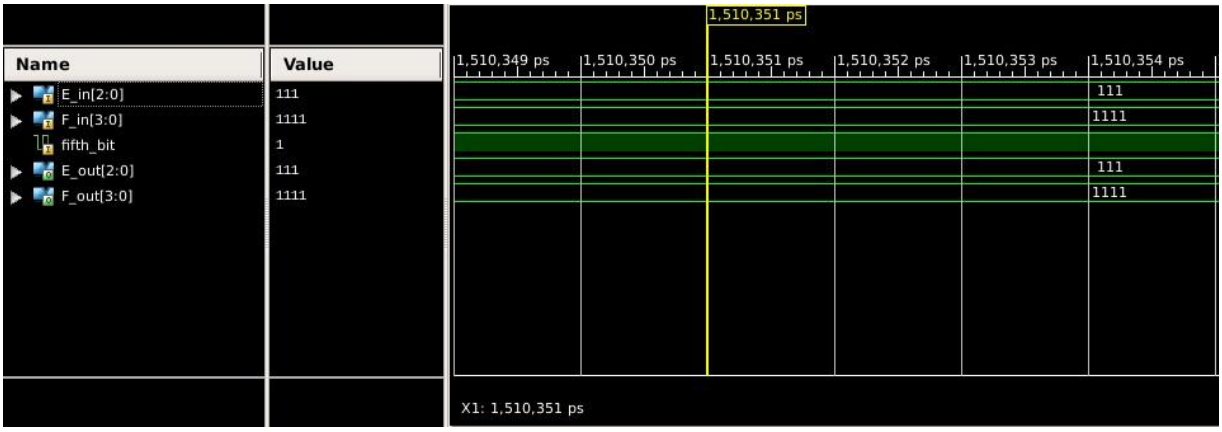

Figure 13: Waveform diagram for `handle_round` for the case `D = 100000000000`

In this module, we can see that `fifth_bit` is `1`, which means we must round. However, we can see that both the inputted exponent and significand would overflow if we attempted to round up. As per the spec, we simply leave the exponent and significand as they were inputted.

4 Conclusion

In this lab, we developed three modules that came together into a top module that converted a 12-bit two's complement number into an 8-bit floating point representation. To accomplish this, we set up one module that extracted the sign bit of the value, one module that calculated the exponent and significand, and a final module that rounds these values.

The biggest difficulty we encountered was figuring out how to establish a connection between our modules and the top module. Once we figured out how to translate inputs to outputs of various modules, the logic was fairly simple, and we just had to work out a few edge cases that caused issues in our testing.