CS M152A Lab 2: Floating Point Converter

Group: Deven Agrawal, Shaan Mathur, Garvit Pugalia

UID: 104646996, 904606576, 504628127

Date Completed: Monday, 5/1/2017

TA: Hongxiang Gu

Introduction

The aim of this project was to implement a pre-designed floating point converter using Verilog modules. The input is a 12-bit number in the two's complement representation, which is converted into three values: a 4-bit significand (F), a 3-bit exponent (E) and a sign bit (S). The conversion works in the following way:

$$F.P. = (-1)^S * F * 2^E$$

As an example, the input D = 111111111110, which represents -2, will be outputted as $F.P. = (-1)^1 * 2 * 2^0$. The suggested design consisted of three modules (described in the next section) and various tests were conducted to ensure that our specific implementation of the modules worked as expected in a simulation environment.

Design Description

The overall implementation of the floating point converter was divided into three submodules: convert_twos.v, floating-rounding.v and round.v. As shown in the schematic below, the 12-bit input was first passed to the convert_twos module which converted the two's complement representation into a sign magnitude representation. This sign magnitude number was passed into the floating-rounding.v module which consists of two key procedures. Firstly, the number of leading zeros were counted as the foundation for the final value of the exponent. Secondly, the leading bits were extracted (along with a special fifth bit used for rounding) to represent the significand. The last module dealt solely with cases involving rounding and outputted the final significand and exponent values.

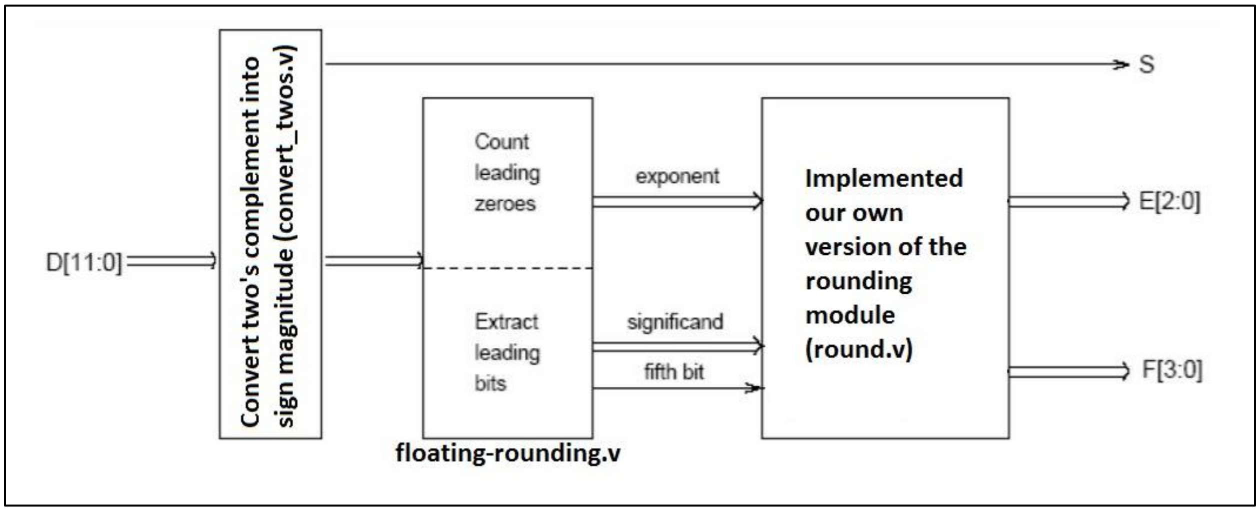
D[11:0]
Convert two's complement into sign magnitude (convert_twos.v)
Count leading zeroes
Extract leading bits
floating-rounding.v
exponent
significand
fifth bit
Implemented our own version of the rounding module (round.v)
S
E[2:0]
F[3:0]

Figure 1: The overall modular design of the floating point converter, including the names of the specific Verilog files.

Convert to sign magnitude

This submodule converted the two's complement representation into sign magnitude, for example, IN = 111111111110 (-2) would be converted to OUT = 100000000010. The processing of the input requires just one major step - if the number is negative, flip the bits and add one (negate the number), otherwise, return the same number. However, since 12-bit INT_MIN cannot be represented in 12-bit sign magnitude, the overflow issue was individually taken care of by adding one to the input (in case of INT_MIN), returning INT_MIN + 1 which is the most negative number that can be represented in sign magnitude. This entire procedure is displayed in the schematic below, with the individual handling of INT_MIN highlighted on the left.

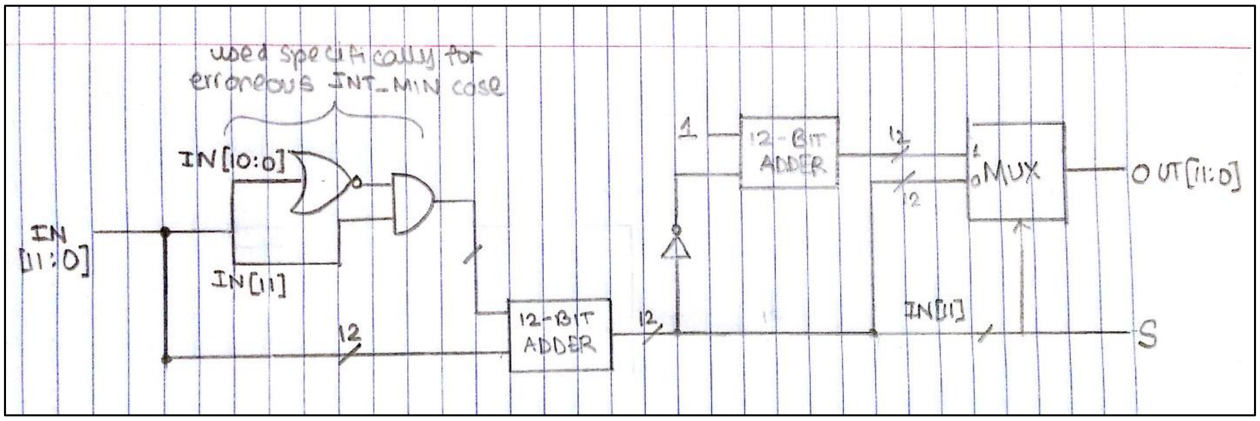

Figure 2: The conversion from two's complement (IN) to sign magnitude (OUT). The module on the top left represents the handling of the erroneous INT_MIN edge case. The two cases are represented in the MUX on the right, where the sign of the inputted number (IN[11]) acts as the select line.

Extracting bits for floating point representation

The next major module was essentially responsible for the conversion of the signed magnitude input into the approximation of the floating point version (prior to rounding). To implement this module in an elegant way, we also describe a submodule for this module, which was implemented in Verilog as the body of a for loop. The order of execution did indeed matter since we wanted to count the leading zeroes as well as the index of the first 1 bit.

What happens in this module, which we call "Loop Kernel," is the following. The input *index* is used in a MUX to index into and select the appropriate bit in our signed magnitude input. We logically AND this with an *adder* input, which defines what we add to our current *count* of zeroes; this assures that the new *adder* value will be 1 up until the first 0 is reached. After adding this to *count*, we have the *adder* and *count* outputs which can be used in the next module. Additionally, to keep track of the first index of a 1, we initialize to input *j* to 1111, an invalid index. Thus ANDing all the bits in *j* yields one if and only if *j* is still uninitialized to a proper value. If we then AND this result with the current bit of interest in our signed magnitude input, we will have a value of 1 if and only if the current bit is the first instance of a 1. Thus we can use a MUX with this value to select the current value of *j* if 0, or the current value of *index* if 1. We then output this *j* and the current signed magnitude input for the next module in the loop.

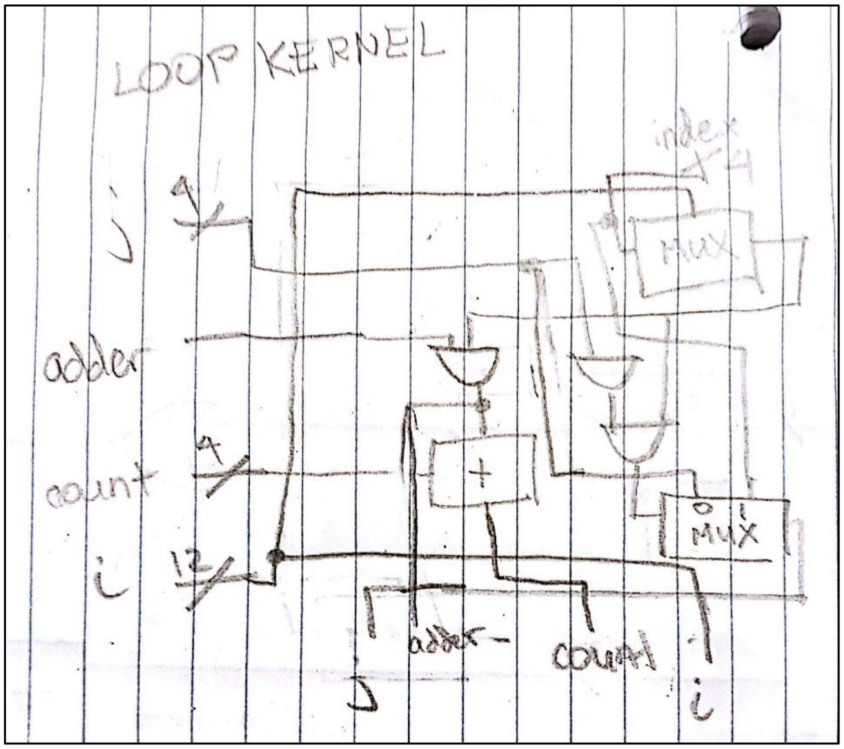

Figure 3: This is the Loop Kernel module that was used to model the body of our for loop in Verilog. We are keeping track of the current index, the current count of leading zeroes, how much we are adding to the count, and the signed magnitude input. We also output all these values to pass as input to the next iteration/module.

Now we can use this “Loop Kernel” module 11 times successively to incrementally walk through the bits to finally get the number of zeroes (*count*) and the index of the first 1 (*j*). This is shown in Figure 4. As input for the first “iteration” we have the invalid value for *j* of 1111, an *adder* value of 1, the signed magnitude input, and a zero-extended complement of the sign bit for the *count*. This last input is used to account for the fact that we actually need to count the leading zeroes in two’s complement form, so we add accordingly. After the successive Loop Kernel modules, we subtract from 8 the sign of the value, and compare to see if that is greater than the count. If it is not, this implies that we will by default take the output as *exp* and *fifth* having a 0 value and taking the last four bits as the *signif*. If it is, we then subtract off count to yield our *exp* value, use a chain of subtractions and MUX’s to select the next four bits starting at index *j*, and take the fifth bit after that as our *fifth*. Between these two options, we use a MUX to select them based on the aforementioned comparison.

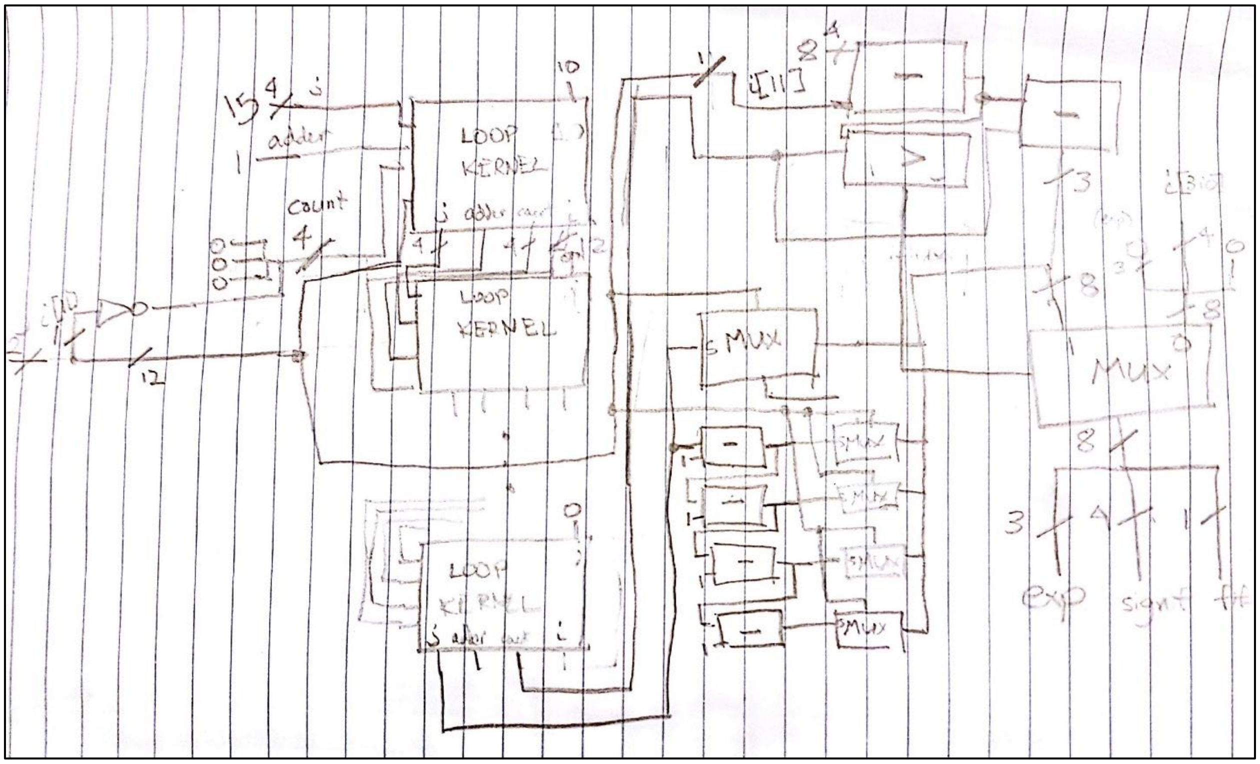

Figure 4: This is the module for converting signed magnitude to the (not rounded) approximate floating point representation. The Loop Kernel modules in tandem provide us the number of leading zeroes as well as the index of the first 1 bit, which allows us to decide what exponent value to use based on a simple formula as well as how to choose the next five bits after the first 1 bit for the four-bit significand and fifth bit output.

Rounding and final values of exponent and significand

This module dealt with the final touches of rounding the previously acquired values of exponent and significand. The main idea behind this submodule is the use of the fifth bit (calculated in the previous module) to determine the rounding of the floating point number.

If the fifth bit is indeed a 0, then no rounding is necessary (essentially we are just truncating), and thus we can just immediately pass through the input *i_signif* and *i_exp* as *F* and *E* respectively. This is demonstrated with a MUX in Figure 5. However if it is indeed a 1, then rounding must be done. The following logical bitwise operations are used to handle every rounding case.

We begin by adding one to *i_signif* since we are rounding up. Overflow occurs if and only if the significand overflows to 0000. Thus the value 1 will indicate overflow if we invert the bits and AND the bits. If there is indeed overflow, we want to set our *signif* to 1000, otherwise keeping it the same. Both of these cases can be cleanly executed by adding the prior overflow boolean to our current significand and shifting by 3 times that boolean (thus remaining the same if the boolean is 0 but becoming 1000 if the boolean is 1). A similar approach is now taken to round up the exponent, which is added to the aforementioned boolean so that it will be incremented only if overflow in the significand occurred. To detect overflow here, we know it will occur if the *exp* is 000 and there was overflow in the significand; thus we flip the bits in the exponent, AND the bits, and then AND with the boolean. Now we have a new boolean value that says whether the *exp* overflowed or not. If overflow occurs, we want *exp* and *signif* to consist of all 1's. Thus we subtract from *exp* the latter boolean, which would cause an overflow back to 111 if there was overflow; we then OR the results of *signif* with the negation of the boolean (which would be a pattern of all 1's if overflow occurred, thus resulting in 1111). Thus we are finished.

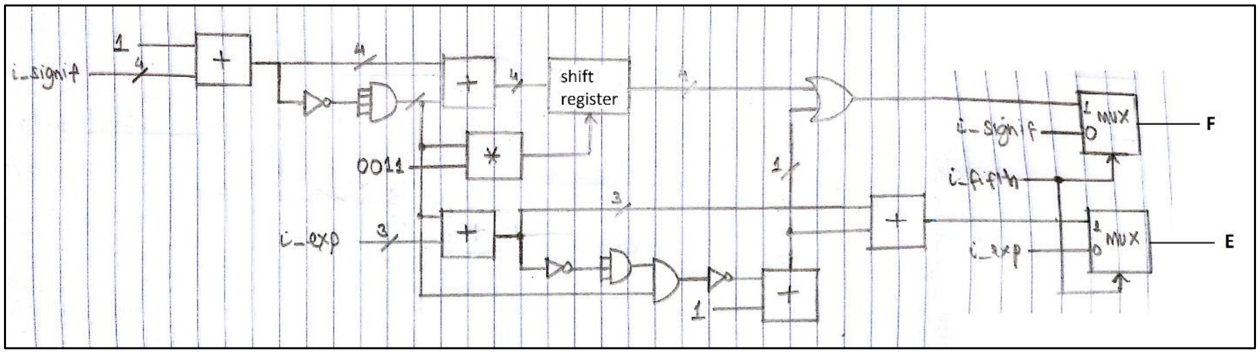

Figure 5: This module is purposed with rounding the current floating point representation based upon the fifth bit value. We truncate if the fifth bit is 0, and round up by incrementing the significand if the fifth bit is 1. In the latter case, a series of bitwise operations are used to, by means of boolean logic and bit vectors, determine how precisely to round the significand and exponent, even elegantly handling the extreme cases as well.

Simulation Documentation

The complete program, including the four sub-modules discussed above, was tested for most of the edge cases, such as INT_MIN and INT_MAX. As the program was mostly self-explanatory, the tests were minimal and only concerned with the complete, compiled program, as shown in the table and associated waveform below.

Test case	Expected result	Pass/No Pass

D = 0
(000000000000)
	S = 0
	F = 0000
	E = 000
		PASS

D = INT_MIN
(100000000000)
	S = 1
	F = 1111
	E = 111
		This was a trickier case, and needed to be hardcoded as explained in the schematics earlier.

D = INT_MAX
(011111111111)
	S = 0
	F = 1111
	E = 111
		PASS

D = -1
(111111111111)
	S = 1
	F = 0001
	E = 000
		PASS

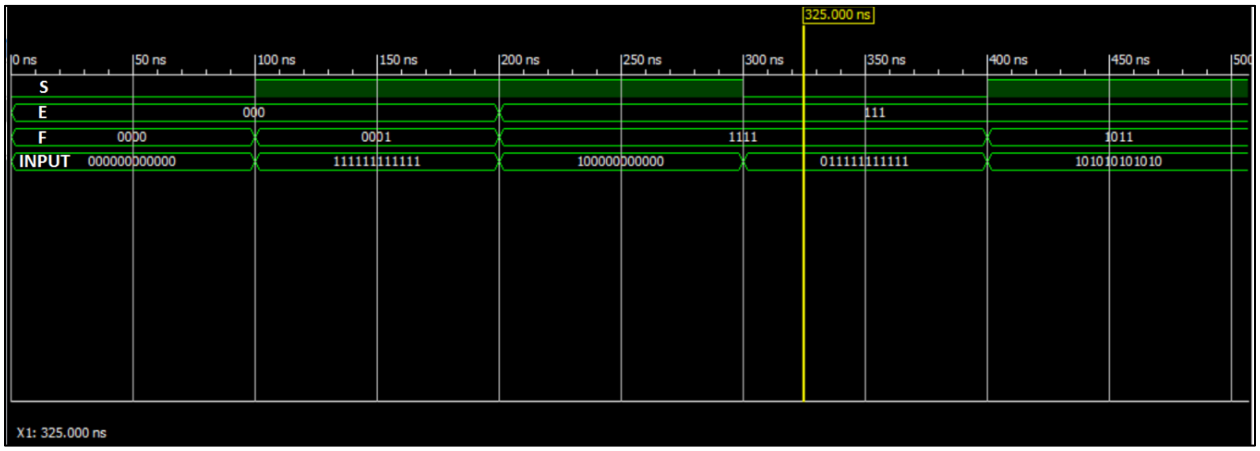

Figure 6: The waveform shows the input and the corresponding output for four edge cases and a regular case. The program passed all the testbench cases and the provided grading testbench.

Conclusion

The modular design of the floating point converter worked cohesively and allowed for a better, cleaner implementation. The key learning points from the project were the fundamentals of modular design and the conversion of interface specification into implementation. While mostly easy to understand, there was an obstace faced in terms of the representation of INT_MIN. However, this was quickly resolved by hardcoding the processing for this specific case. Overall, the project was a success and worked well as a preparation for harder things to come.