M152A - Lab 3
Stopwatch

Markus NOTTI - 904269231
Kyle BAKER - 604273748
Niels PINEDA - 604272353

February 23, 2016

Introduction

In this lab, we implemented a basic stopwatch, and in the process, went through the complete FPGA design flow. This stopwatch is capable of counting seconds and minutes up to 99 minutes, and displaying the time on a seven segment display. We also designed the stopwatch with pause, adjustment, and direction capabilities.
The stopwatch itself was designed with 5 primary inputs: ADJ, SEL, DIR, PAUSE, and RESET.

ADJ was controlled by a switch that, when set high, would put the stopwatch in adjustment mode, meaning that the stopwatch would count at a rate of 2 Hz, and depending on SEL, and DIR, either only the minutes or only the seconds would change. This adjustment mode is the way for a user to change the stopwatch's value to a desired value.

ADJ	Action
0	Stopwatch counts normally
1	Stopwatch enters adjustment mode and counts at 2 Hz

Figure 1: ADJ Switch

SEL was controlled by another switch, which when set high would change seconds in adjustment mode, and when set low would change minutes in adjustment mode. Note that if ADJ is set to low, the value of SEL makes no difference.

SEL	Action
0	During adjustment mode, minutes change at rate of 2 Hz
1	During adjustment mode, seconds change at rate of 2 Hz

Figure 2: SEL Switch

DIR was controlled by yet another switch, which when set high, would cause the stopwatch to count up from zero, and when set to low would cause the stopwatch to count down to zero. Note that this direction controls the direction of the counting regardless of whether or not the stopwatch is in adjustment mode or not.

DIR	Action
0	Stopwatch counts down to zero
1	Stopwatch counts up from zero

Figure 3: DIR Switch

PAUSE was controlled by a button, which when pressed would pause the stopwatch. In order to filter out noise and correctly pause the stopwatch, we had to implement debounce logic for this functionality which will be talked about in the design section of the lab.

RESET was controlled by another button, which when pressed would reset the stopwatch's count to 00:00.

While taking 5 inputs, the stopwatch had one output, the seven segment display. This output, however, was a little bit more complex than a single output. The seven segment display was split into 4 digits and 8 segments per digit (really making it a 32 segment display). The main challenge we had to deal with when outputting the correct time on the display was the fact that you could only send 1 combination of segments to the display. This meant that if I wanted to display separate numbers on different digits, I could not physically do it. With the segmented display we used on our board, it is physically impossible to display two different numbers on two different digits.
To get around this, we displayed one digit every time a 500 Hz clock was high, cycling through all four of the digits more than 125 times a second. This gave the illusion that the segmented display was displaying 4 different numbers simultaneously, though it wasn't really.

Design Description

When designing the lab, we created a module for each section in figure 3. We named these modules convertToSignMag, countLeadZeros, extractLeadBits, and rounding to correspond to each of the sections in the figure. We then created a main module called

fpcvt to tie all the other modules together. The implementation of the modules is described in the sections below.

fpcvt

We then created a main module we named fpcvt which took in as input D, a bus of 12 wires and outputted a sign bit S, the exponent bus E, and the floating point bus F. The functionality of fpcvt essentially just followed the flow of figure 3. We decided to follow this flow because it seemed the most logical and modular approach to solving the problem. The input D was passed into convertToSignMag, which fed its output to both countLeadZeros and extractLeadBits. The output of extractLeadingBits (the significand and the fifth bit) and the output of countLeadingZeros (the exponent) were passed into the rounding module. We then assigned the output of the rounding function to E and F, the outputs of fpcvt.

Figure 4: fpcvt design diagram from lab.

The circuit diagrams for each of the sections in the figure above will be depicted in the following sections.

convertToSignMag

Our convertToSignMag module took in an 12 bit input named D2 and and has S2 and MAG as 1 and 11 bit outputs respectively. D2 represents the original 12 bit input into the fpcvt module, S2 represents the sign bit to be outputted in fpcvt, and MAG represents the unsigned magnitude of the 12 bit integer passed in. The first step in this module was creating a 12 bit reg called tempMAG which we assigned to have the same value as D2 in order to operate on D2. We then checked if D2 was equal to the bit stream 100000000000, which is the most negative number. We needed to check this because the next step is, if the number is negative, inverting the number by taking its complement and adding one. But since the integer can only represent numbers in the range -2048 to 2047, overflow will occur. In this case, we set our sign bit to 1 and our magnitude to 11111111111 and return from there. Otherwise, we checked if the sign bit was negative. If it was, we inverted tempMAG as previously mentioned. We then used a for loop to extract the 11 least significant bits in order to extract the magnitude and ignore the sign bit. In the end, we assigned the output MAG to be tempMAG and the sign bit output S2 to be D2[0] (the first bit of the input integer).

Figure 5: convertToSignMag circuit diagram

In this circuit, the input is 12 bits (represented in hex on the diagram) and the output

is one sign bit and one bus of 11 bits which is the unsigned magnitude. The 11 least significant bits are fed through a NAND gate to find out of they are all zeros. That signal is then fed to an AND gate along with the sign bit. The purpose of this is to find out if the input is the most negative integer, 100000000000. In this case, we have the output be 111111111111, or fff in hex on the diagram. Otherwise, we use a mux to choose between the inverted version of the integer and the regular version of the integer depending on whether the sign bit is 1 or 0. At the end, we extract all the bits but the sign bit to get the unsigned magnitude.

countLeadZeros

The countLeadZeros module takes in MAG3 as an 11 bit input and specifies E3 as a 3 bit output. MAG3 represents the magnitude of the integer passed into the fpcvt module and E3 represents the exponent of the integer. The exponent E3 was calculated by counting the number of leading zeros, then setting E3 to the corresponding exponent value in the table below.

Figure 6: Leading zeros to exponent conversion table taken from the lab.

To count the number of leading zeros, we used a priority encoder. We utilized a switch statement to implement the priority encoder. We first checked the first leading bit. If that bit was 1, then there was 1 leading zero (including the sign bit). If it was zero, then we check the next bit. If that bit was 1, then there were 2 leading zeros, and so on. With this information, we assigned the proper exponent value to E3.

Figure 7: countLeadZeros circuit diagram

The countLeadingZeros module in the circuit diagram above takes an 11 bit magnitude which is immediately split into its 11 bits. These 11 bits then are sent into a cascade of muxes which essentially examine each bit one at a time, starting with the most significant. As soon as a 1 bit is found, the cascade shuts down, and only the appropriate number, corresponding the appropriate amount of leading zeros, is chosen by a mux and finds its way to the 7 input OR gate, and then to the output as the 3 bit exponent.

extractLeadBits

This module takes in MAG4 as a 11 bit input and specifies SIG4 as a 4 bit output and FB4 as a 1 bit output where MAG4 is the same as MAG3 in countLeadZeros. SIG4 represents the significand and FB4 represents the fifth bit. To calculate SIG4 and FB4, we first needed to left shift MAG4 by the number of leading zeros (up to 7 times). We then used another priority encoder (as described under countLeadZeros) to count the

number of leading zeros and to decide how many times to left shift MAG4. We then assigned SIG4 to the four must significant bits of the shifted MAG4 and FB4 to be the fifth most significant bit of the shifted MAG4.

Figure 8: extractLeadBits circuit diagram

In the circuit diagram for extractLeadBits, we used a chain of ANDs with one input negated in order to count get a high signal on all the leading zeros and a low signal on all bits including and after the first 1 in the input. We only needed to count up to seven because seven is the max amount you can shift by. These signals are fed into a bit extender in order to make them 4 bits long to make them compatible with the adder in Logisim. All the signals are then added, thus giving a count of the number of leading zeros. After this, we feed the input and the count to a shifter and left the input shift by the count. From there, we extract the 4 most significant bits to get the significand and the fifth most significant bit to get the fifth bit.

rounding

Our rounding module takes in E5 as a 3 bit input, SIG5 as a 4 bit input, and FB5 as a one bit input. E5 represents the exponent bus outputted by countLeadZeros and SIG5 and FB5 represent the significand bus and the fifth bit, both of which are outputted by extractLeadBits. The rounding module also specifies E6 as a 3 bit output and F6 as a 4 bit output. E6 and F6 represent the final exponent and significand values to be outputted by fpcvt. We began the module by checking if the significand was 1111 and the fifth bit was 1. This is important because when you add the two you'd get overflow, so we set a special overflow flag to be 1. In this case, we also checked to see if the exponent had the value 111 because when you add 1 to the exponent in this case you would overflow the exponent. If this case were true, we set a special flag we called "allOnes" to be true. If allOnes was true, we left the SIG5 and E5 unaltered because it was beyond what floating point could represent. If FB5 is zero, we also leave SIG5 and E5 unaltered because you just add by zero. If FB5 was one and overflow is zero, we simply added one to SIG5. If FB5 was one and the overflow bit was also one, then we shifted SIG5 right by one, added one to the exponent, and added one to the already shifted SIG5. We had to shift before we added to SIG5 because if we added first, overflow would happen and zero out SIG5. At the end, we assigned E6 and F6 to the manipulated E5 and SIG5 values.

Figure 9: rounding circuit diagram

The rounding module in the circuit diagram above takes as input a 3 bit temporary exponent, a 4 bit temporary significand, and a 1 bit “fifth bit”. The exponent and

significand are immediately incremented and sent into muxes each with their original un-incremented selves where the fifth bit input is chosen as the selector for the significand, telling the mux to use the incremented significand if 1. The overflow of that addition is then used as the selector for the mux with the exponent and its incremented self as input. If the significand's overflow is then also used as the selector for another mux which, if overflow is 1, sets its output equal to the original significand right shifted and incremented. Another mux then handles the case of the exponent and significand both overflowing, resetting the outputs to all 1's.

Simulation

The simulation portion of the lab involved several test bench files, one for each individual module, with the most thorough of the test cases falling under the overarching FPCVT. By understanding what each module was intended to do, and thus knowing the expected output for each one, the testing simply consisted of: creating a test bench file via adding a "testing fixture" source, then writing example instances along with a delay in said file, and finally running through the iSIM Waveform while checking accuracy by hand. Our individual modules, which more or less acted as "sub/helper-functions" in order to implement the overall module of FPCVT, consisted of: convertToSignMag, countLeadZeros, extractLeadBits, and rounding. While it may not have been the best idea, we made very simple and basic test cases for these smaller modules in order to allow us to continue to progress in a timely fashion, while still working at a timely rate.

For convertToSignMag, the input was the input data in the form of two's complement. The two instantiations we tested were the integer values of 1 and -1, or D2 = 12'b000000000001 and D2 = 12'b111111111111 respectively. We felt as though these were good, simple test cases as they were several corner cases which also tested if both our negative and positive conversion was working correctly. As expected, the outcomes were correctly stored in S and MAG: for 1 S=0 and MAG=00000000001 and for -1 S=1 and MAG=00000000001.

Figure 10: Example testbench simulation testing convertToSignMag module

The following simulation for countLeadZeros was by far the trickiest for our group, as we initially had trouble understanding exactly what the output was to look like; consequently, this led to us believing our first attempt was correct which made our overall converter fail. However, after clarifying the process for this module, we were able to produce the correct output, in which we checked many cases including: MAG3 = 11'b00000000000; MAG3 = 11'b00000000001; MAG3 = 11'b00000000010; MAG3 = 11'b00000000100; MAG3 = 11'b00000001000; MAG3 = 11'b00000010000; MAG3 = 11'b00000100000; MAG3 = 11'b10100100000; and MAG3 = 11'b11111111111. Our initial approach led to undefined behavior, but after the fix, we were very thorough

in testing this module. All of these test cases simply represented every different number of leading zeros (from 0 to > 8) which would led to different exponents.

Figure 11: Example testbench simulation testing countLeadingZeros module

Module extractLeadBits was relatively straightforward, and similarly, so were the test cases. Because of the simplicity of the code itself, which merely consisted of a priority encoder, we only used two test cases, which were passing in the magnitude (MAG4) as MAG4 = 11'b01101000101 and MAG4 = 11'b00000101111. These represented relatively simple cases that assured our priority encoder was working, and both of these resulted in the expected outcomes.

Figure 12: Example testbench simulation extractLeadBits module

The final module (aside from the converter) was used for rounding of the floating point conversion. We carefully tested the various corner cases (in this case, the various types of overflow) after testing a couple of very basic cases. The cases we tested were based off of changing the exponent (E5), significand (Sig5) and fifth-bit (FB5); the cases included: [E5 = 3'b001; SIG5 = 4'b0001; FB5 = 0;] ; [E5 = 3'b001; SIG5 = 4'b0001; FB5 = 1;] ; [E5 = 3'b001; SIG5 = 4'b1111; FB5 = 1;] and [E5 = 3'b111; SIG5 = 4'b1111; FB5 = 1;]. The first two instantiations were very simple cases, simply assuring if the basic rounding was correct. Once those were verified, we tested two major overflow conditions, one in which the exponent could had room to carry over, and the other where there was no room for adding to any input. These covered the major corner cases, all of which eventually passed with some adjustments to the code to account for those particular cases.

Figure 13: Example testbench simulation testing rounding module

Lastly, and most importantly, we tested the entire converter, FPCVT. The major trouble we had testing this came when we believed all of our sub-modules were working correctly, when in fact countLeadZeros was not outputting what it was supposed to. After that adjustment, our test simulations were generally correct aside from a single corner case (which was the most negative number). We had a variety of test cases including: D = 12'b 000000101100; D = 12'b 000000101101; D = 12'b 000000101110; D = 12'b 000000101111; D = 12'b 111111111111; D = 12'b 011111111111; D = 12'b 011111000000; D = 12'b 100000000000; D = 12'b 111001011010, where D was the input. As indicated by the images below, these test cases checked for multi-

ple conditions including: the base case of a generic number (000000101110), negative rounding (111001011010), negative one (111111111111), the most positive number (011111111111), and the edge case (which is the one we initially missed) of 100000000000.

Figure 14: Example testbench simulation testing generic number on fpcvt module

Figure 15: Example testbench simulation testing negative rounding on fpcvt module

Figure 16: Example testbench simulation testing negative 1 on fpcvt module

Figure 17: Example testbench simulation testing most positive number on fpcvt module

Figure 18: Example testbench simulation testing edge case on fpcvt module

Conclusion

Overall, the implementation of the converter went relatively smoothly, especially with the help of the lab manual. By breaking down the the overall design and module of FPCVT into several modules, in our case: convertToSignMag, countLeadZeros, extractLeadBits, rounding, then using all of them in the, the process was made much clearer and easier. Initially, we fed in the data in two's complement representation into the convertToSignMag. That module's output was used as the input to countLeadZeros and extractLeadBits. These output the exponent, significand, and fifth bit, which fed into the rounding process, which eventually led to the overall output.

There were, however, several difficulties throughout the implementation. For starters, simply trying to understand Verilog, most notably when to use wires or registers along with dealing with assignments and always blocks, took a decent amount of time. After understanding this and implementing our first module, we wrote our own test bench

instead of simply adding a “test fixture” source file, which led to even more time being lost. Our final problem with ISE came with a minor mistake, but we often would run the simulation on the uut file accidentally, which led to undefined behavior.

Regarding the actual writing of code, the problems came from either counting the leading zeros or the various corner cases. Most notably, the description in the lab manual for counting the leading zeros was worded oddly, which led to a confusion on what the module should actually do. The only test case we initially failed was the case of 100000000000, but we were able to fix this in the convertToSignMag module. Aside from those issues, the lab was relatively straightforward.