CompArch Lab 2

Logan Sweet & Maggie Jakus

2 November 2017

1 Implementation

For this midpoint checkin, our top level implementation followed the layout specified in Figure 1. It is made up of three input conditioners and a shift register, which are explained in more detail in the sections below. At a higher level, this module (pushed as midpoint.v) uses one input conditioner to clean up a serial signal, one to create a positive clock edge signal, and one to create a negative clock edge signal. These datalines are passed to a shift register, which converts the serial data to parallel data and vice versa.

This will serve as a building block for our final SPI memory module. The most significant component we will need to build for the full SPI memory will be the finite state machine.

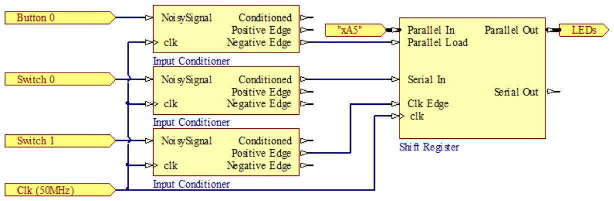


Figure 1: Our top level implementation for the midpoint check-in

2 Input Conditioning

To begin this lab, we built the Input Conditioning subcircuit. Most of this was already built - all we had to do was understand the code in "inputconditioner.v" and add signals to detect when there are positive and negative edges of the external signal. We added edge detection by writing if statements that would use a nonblocking assignment to turn on "positiveedge" and "negativeedge". This involved some amount of troubleshooting, as we did not understand at first where we should reassign these values to 0.

The Input Conditioning subcircuit must do three things. First, it should synchronize the potentially noisy input to the internal clock domain. Second, it should debounce the input. Third, it should detect the positive and negative edges of the output signal. We were initially confused about how to test input synchronization. We asked about this on Piazza and were told that it could not realistically be done in Verilog. If you look at our waves (figure 2), you can see that the pin (noisy input signal) does not immediately get transferred to the conditioned output. If you include the synchronizer0 and synchronizer1 wires, you can see the steps more clearly. This caused us to believe that the input synchronizer was working. We can see that debouncing is working in figure 2. When the noisy input glitches (is momentarily switched, and then switched back), the output does not change. One example of this is around 300s, where pin briefly turns on, but there is no corresponding reaction in the output. Finally, we can visually confirm that the rising and falling edges are detected and only present for one clock cycle.

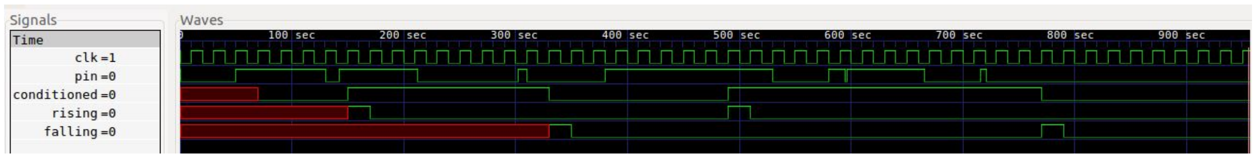


Figure 2

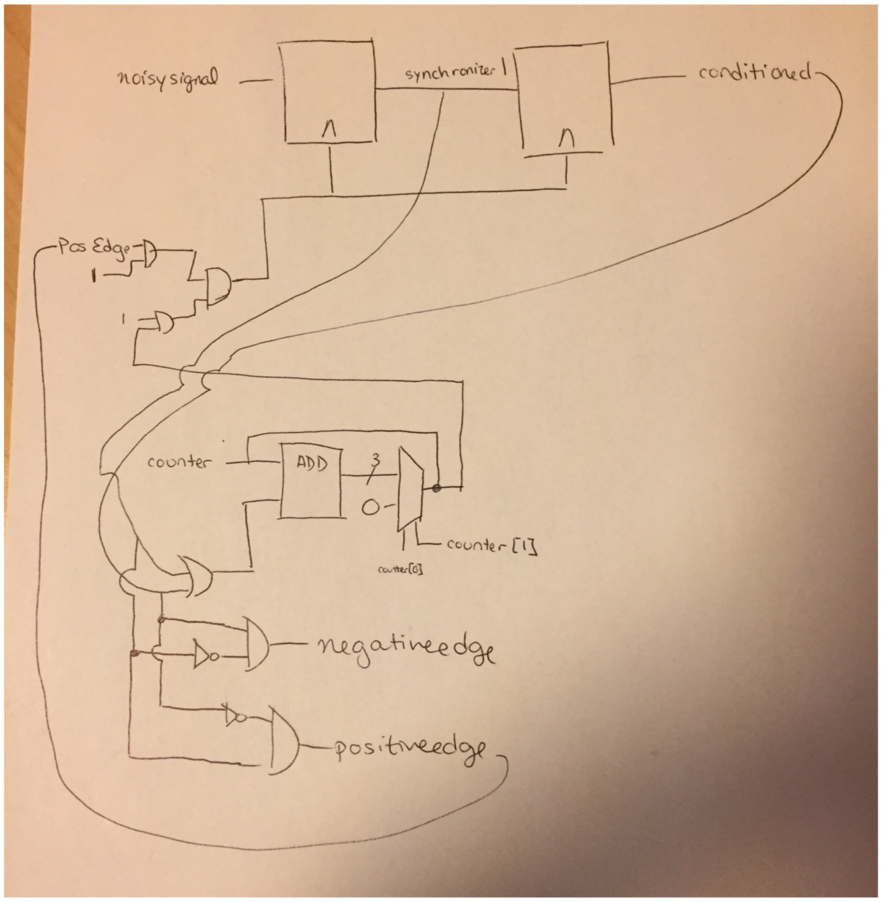


Figure 3: An artistic rendition of our input conditioner circuit.

If the main system clock is running at 50MHz, the maximum length input glitch that will be suppressed for a waittime of 10 is 13 cycles, or 260 nanoseconds. This assumes that the noisy signal entered the system right at the positive clock edge, so that it had to wait one entire cycle before it was passed on to synchronizer0. After one cycle, the noisy signal gets passed to synchronizer0, and one cycle later, it gets passed from synchronizer0 to synchronizer1. When conditioned is not equal to synchronizer1, the counter begins. However, it waits one clock cycle after synchronizer1 changes before counting begins. This will count until counter equals wait time, which in this case is ten clock cycles. At the end of the tenth, counter equals waittime, and any noisy signal that is still present will get passed to conditioned. In total, there will be thirteen 13 cycles between the noisy signal coming in and it being passed to the conditioned output.

3 Shift Register

We began working on the shift register by reading about what a shift register does and what the difference between Parallel In, Serial Out ("PISO") and Serial In, Parallel Out ("SIPO") is. Once we got that figured out, we built the two in behavioral verilog, and the results can be seen in figure 4.

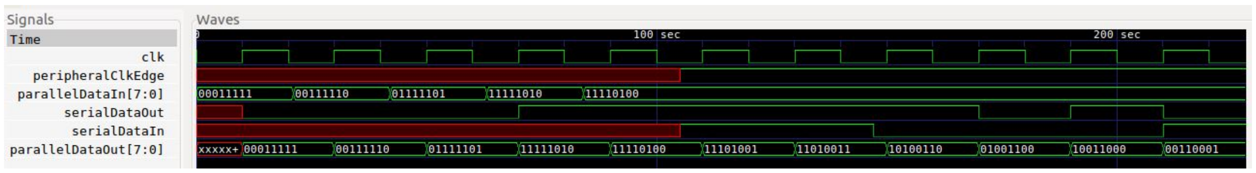
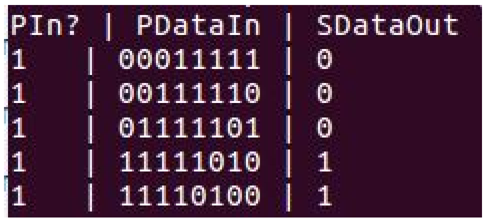


Figure 4: This shows the clock, peripheral clock, parallel load, and SIPO and PISO results.

We first tested PISO. As you can see, the data gets shifted to serial data out on the next positive clock edge. This, coupled with our results seen in the command line (figure 5) convinced us that our shift register was working.

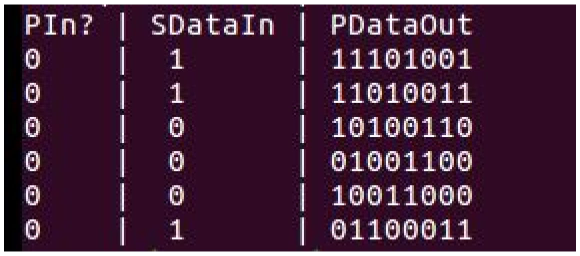


```
PIn? | PDataIn | SDataOut
1    | 00011111 | 0
1    | 00111110 | 0
1    | 01111101 | 0
1    | 11111010 | 1
1    | 11110100 | 1
```

Figure 5: We expect the values in "SDataOut" to match the most significant bits in "PDataIn," and they do!

Next, we tested SIPO. This doesn't become valid until "parallelLoad" turns to 0. Similarly, peripheralClkEdge doesn't become valid until parallelLoad is 0, either. This is because its value is suppressed in the if statements we set up. We did this to determine which shift register (PISO or SIPO) would win if both were activated. This way, SIPO always "wins."

The value in serial data in gets input into parallel data out on the positive clock edge after it is input. Because of our results in figure 4 and our command line results in figure 6, we believe our SIPO shift register is working correctly.



```
PIn? | SDataIn | PDataOut
0    |  1      | 11101001
0    |  1      | 11010011
0    |  0      | 10100110
0    |  0      | 01001100
0    |  0      | 10011000
0    |  1      | 01100011
```

Figure 6: We expect to see the value of 'SDataIn" in the least significant bit of "PDataOut", and we do.

4 FPGA Test Sequence

To test our FPGA, we came up with the following procedure:

To test PISO, we need to push the button. We have a value for parallel data in set in the lab 2 wrapper file. When we push the button, we should see all the values in the shiftregistermem (which is parallel data out). We should see the most significant bit reflected in the one LED on the FPGA that is set to Serial Out.

To test SIPO, we use Switch 0 to set the serial input and Switch 1 as the peripheral clock edge. We tested this by setting Switch 0 to 1, then switching Switch 1 on and off. This should cause the first LED to light up. We then set Switch 0 to 0, and turned Switch 1 on and off. This should cause the first light to move down one, and for the first LED to be off. We repeated this process so that eventually we had the

LEDs as “on - off - on - off”, from most to least significant bit. We should observe the lights “shifting” as we flip Switch 1. We should also see the most significant bit reflected in the LED on the FPGA that is set to Serial Out.

5 SPI Memory

We chose to use the SPI memory structure presented in the lab (figure 7).

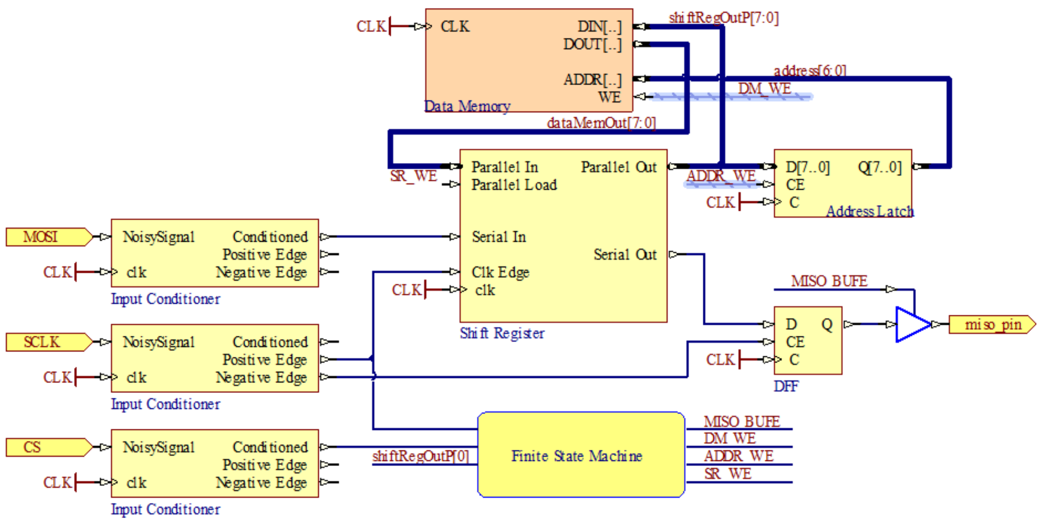


Figure 7: We followed this diagram to structure our spimemory.v file.

6 Finite State machine

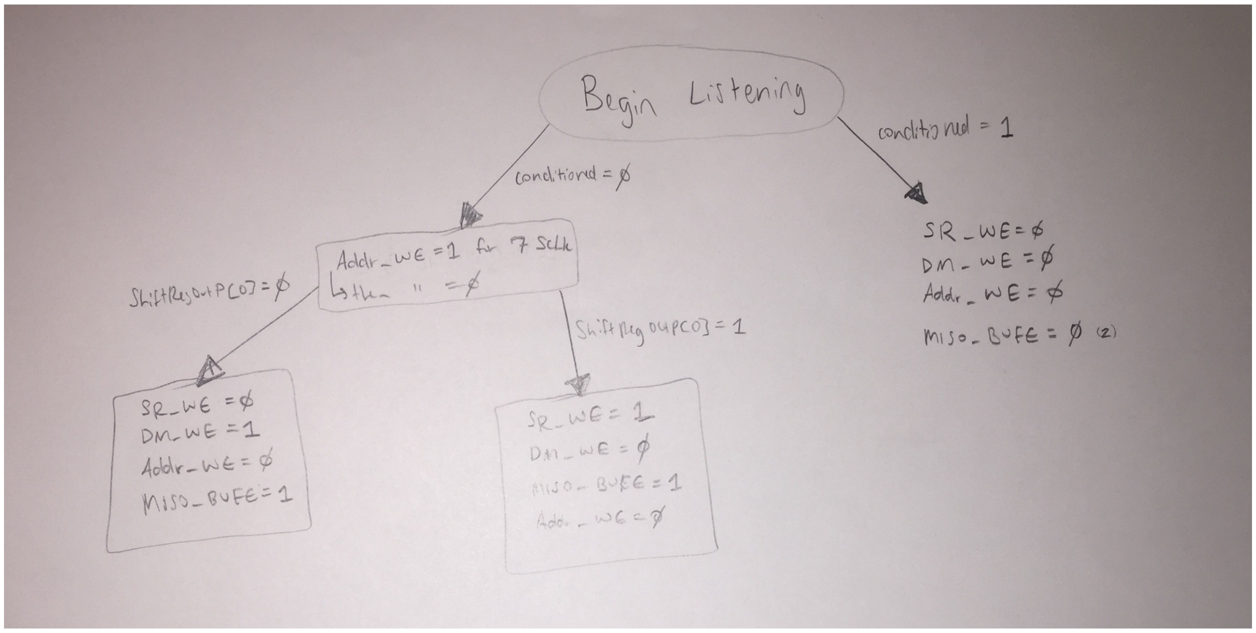


Figure 8: Our FSM structure.

Action	SCLK	CS	Addr WE	Data Mem WE	Shift Reg WE	MISO Buff.
Write to	0-7	0	0	0	0	0
Data Memory	8	0	1	1	0	0
	9-15	0	0	1	0	0
Read from	0-7	0	0	0	0	0
Data Memory	8	0	1	0	1	1
	9-15	0	0	0	1	1
Do Nothing	0-15	1	0	0	0	0

Table 1: The finite state machine follows this pattern. This is implemented in our code using if statements.

7 Testing the SPI

Our test strategy was as follows.

We began by testing the simplest case: when chip select is 1. When this is true, all other FSM controls should be set to 0. As we can see from figure 9, this works!

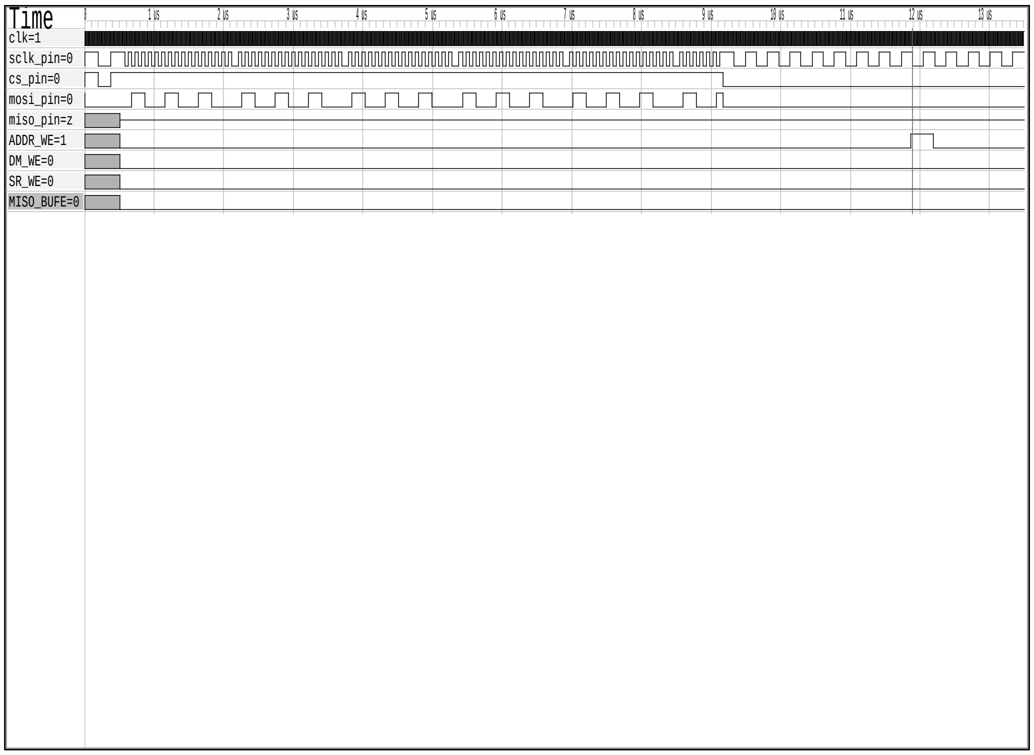


Figure 9: As you can see, while chip select is high, ADDRWE, DMWE, SRWE, and MISOBUFE are all zero. Once chip select is low, the counter begins again, and eventually ADDRWE gets set to one for one clock cycle.

Next, we tested writing to the data memory. We began by writing to a specific address ("0000100"). The first seven bits, therefore, are 0000100, then we sent the value 0 (for "write"), then we sent the value to write ("00011000"). This can be seen in figure 10, where shiftRegOutP becomes "00011000" soon after DMWE goes high.

We then tested reading from the data memory (figure 11). We input the same address as before, then, after ADDRWE comes on, SRWE and MISOBUFE both turn on. The output value on misopin is that most significant value of the parallelDataOut. We can see the correct value ("00011000") in parallelDataOut. Because of the way we have structured our code, only the most significant bit will be passed out. We also have input counters that only allow one value to pass through at a time. For a better understanding of how our shift register works, please see our earlier explanation.

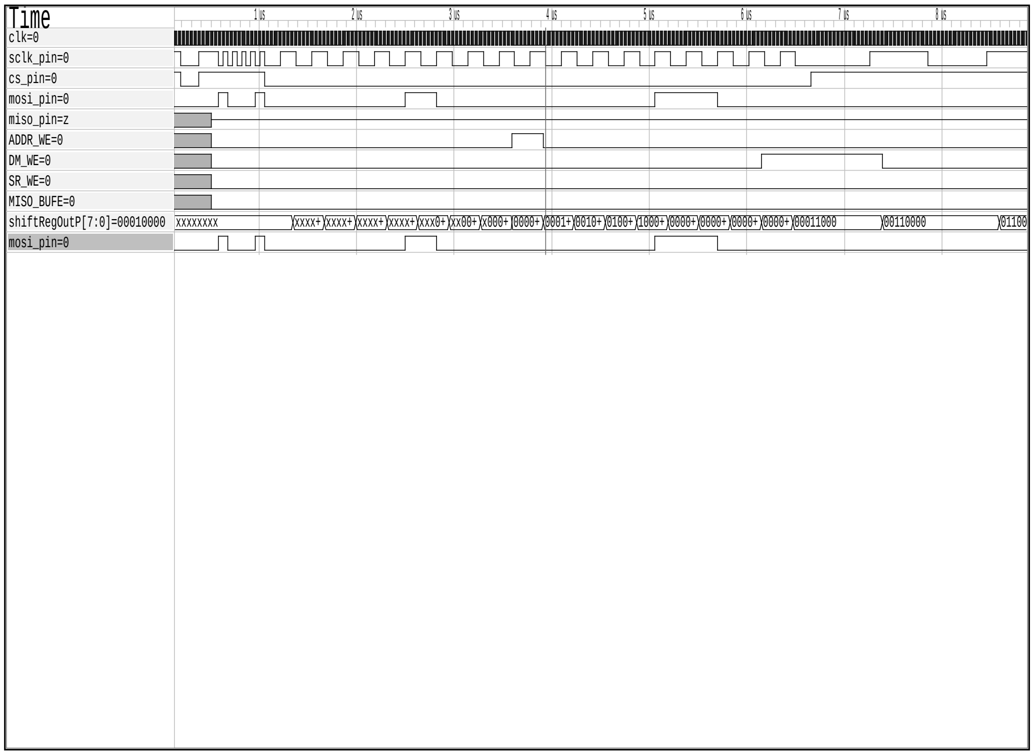


Figure 10: Look! It works!

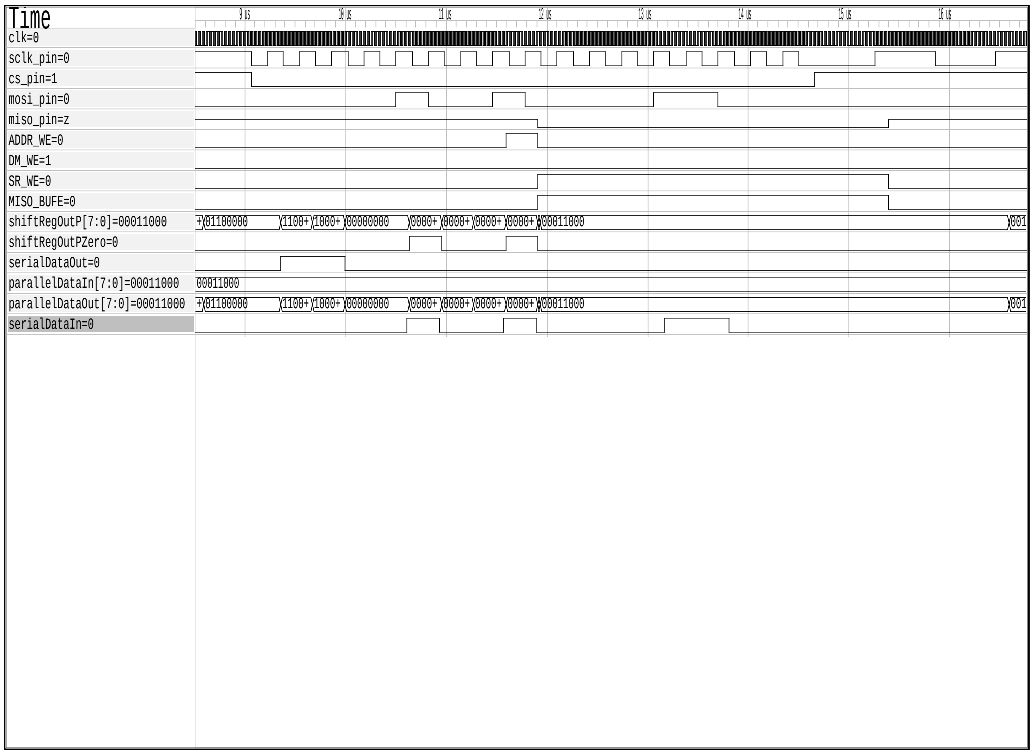


Figure 11: Once chip select goes low, reading begins. First, the address gets passed in. Next, the shift register outputs the most significant value of the data on misopin.

8 Work Plan Reflection

In our work plan, we estimated that this would take us a total of 27 hours. We did not do a great job of tracking how long each task we listed took us since we went back and forth on things (we would often test and think we were done with one part, only to find that we had a bug we didn't see). In total, each of us spent around 20 hours on this lab.

Overall, the FPGA took us much longer than expected, and the finite state machine was quicker than we planned.