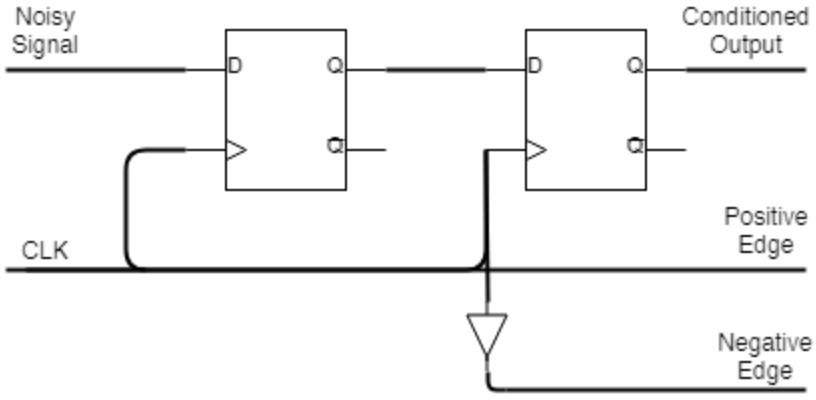
Comp Arch: Lab 2
10/10/2018
Vicky McDermott
Minju Kang
Allison Basore

Comp-Arch: Lab 2

For the first part of this lab, we built a shift register, input conditioner, address latch, finite state machine, and tristate buffer to make a slave SPI Memory module using behavioral Verilog.

Input Conditioner

The first part of the SPI Memory is the is Input Conditioner. This is for taking in a noisy signal, using a clk and several D-flip-flops to output a conditioned signal from the input. The input here could be a button or a switch that has a chance of being metastable or being in a debouncing state. By doing so, the input signal noise was filtered while being synchronized with the clock signal. The following is a circuit diagram of the input conditioner:



For the input conditioner test, we can observe the behavior with waveforms as well as with several Verilog tests. We wanted to double check that the signal we inputted was correct with the conditioned output. We also wanted to double check that the positive edge and negative edge make sense for each test. The two major error cases we wanted to check for are debouncing and a metastable state. While we cannot test a metastable state in Verilog, we can physically test debouncing and the overall system with the FPGA board.

Below is a capture of our waveforms for the input conditioner. Our tests included the following:

1) Two conditioned input value tests
2) Two synchronizer value tests
3) Debouncing test
4) Positive edge test
5) Negative edge test

As you can see in the waveform, a simple input and output functionality is tested. Around 370 seconds, the debouncing test ran. Then the negative and positive edges were tested.

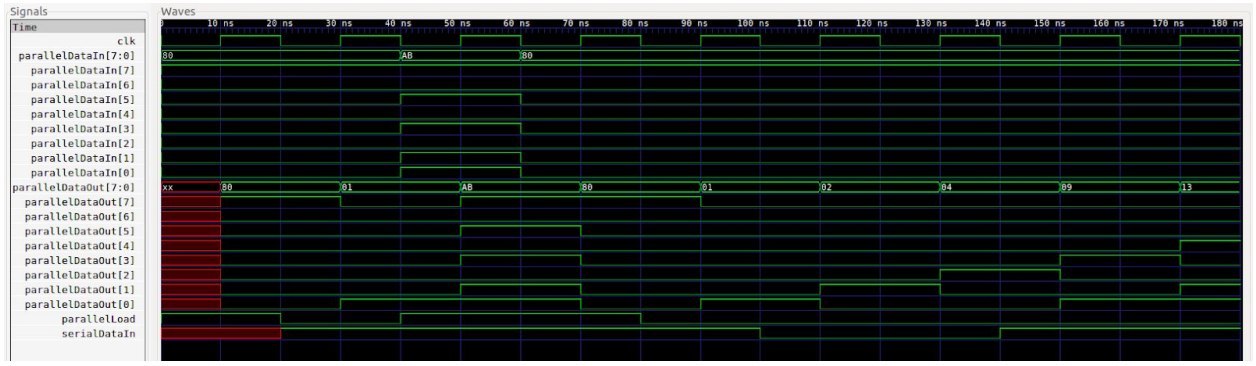
Comp Arch: Lab 2
10/10/2018
Vicky McDermott
Minju Kang
Allison Basore

If the main system clock is running at 50MHz, what the maximum length input glitch that will be suppressed by this design for a wait time of 10 is 20 units of time. This comes from the fact that a noisy signal would have to be unstable for the wait time of 10 for both the d flip-flops in the input conditioner. If an input value was in a metastate for more than 20 units of time, then the input conditioner would not be able to catch the error.

Shift Register

The shift register works by taking in serial data and appending it to the end of the shift register when the peripheral clock has an edge. Alternatively, it outputs the input data when the parallel load is asserted.

Below is a capture of the shift register waveform test. In the waveform, it's obvious that 8th bit is shifted to 7th, 6th, 5th... so on for each clock cycle.

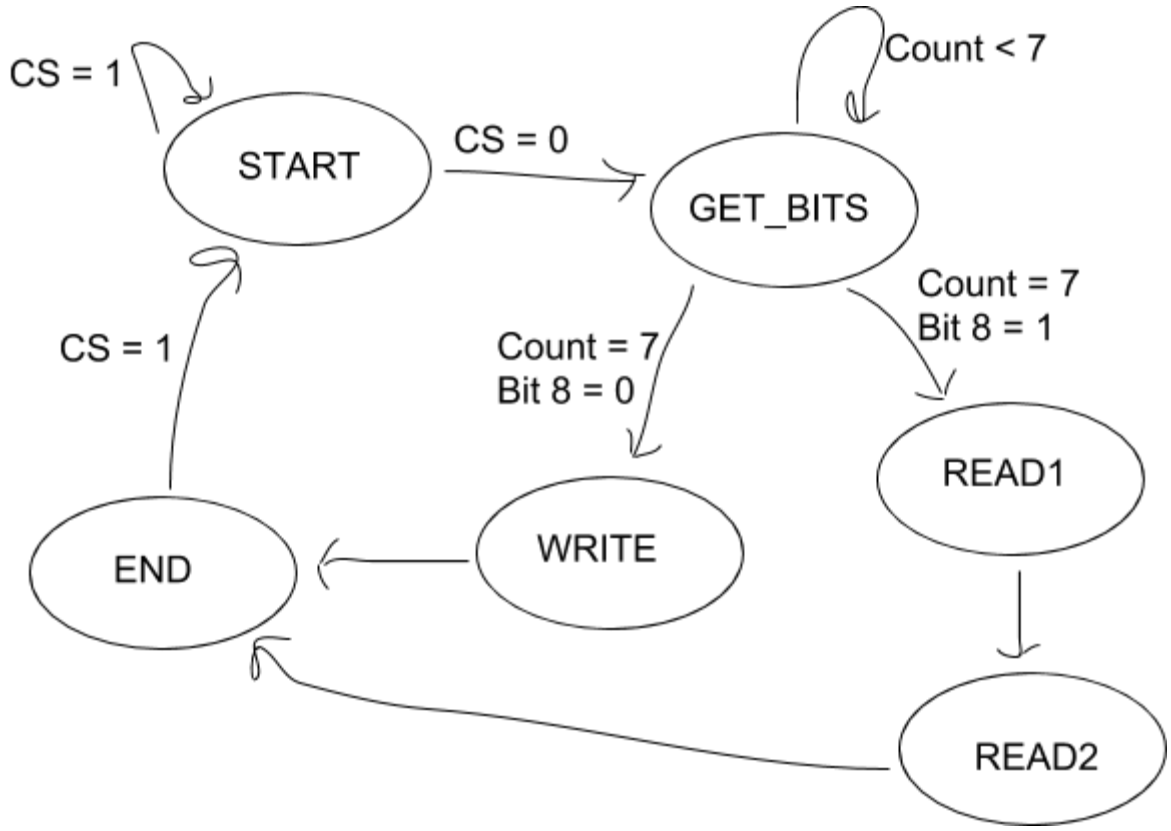


After building the shift register and input conditioner, we uploaded and tested functionality on the FPGA board. Here is a link to a video of our modules functioning. The test shows a preloaded value of 1010101. Then we set the values with the switch. Lastly, we reset the value. Video Link: https://photos.app.goo.gl/TTiFzbUgh1sw8GcH9

Comp Arch: Lab 2
10/10/2018
Vicky McDermott
Minju Kang
Allison Basore

Finite State Machine

We created a finite state machine to output the appropriate control values depending on what state we are in. The states are represented in the following flow diagram:

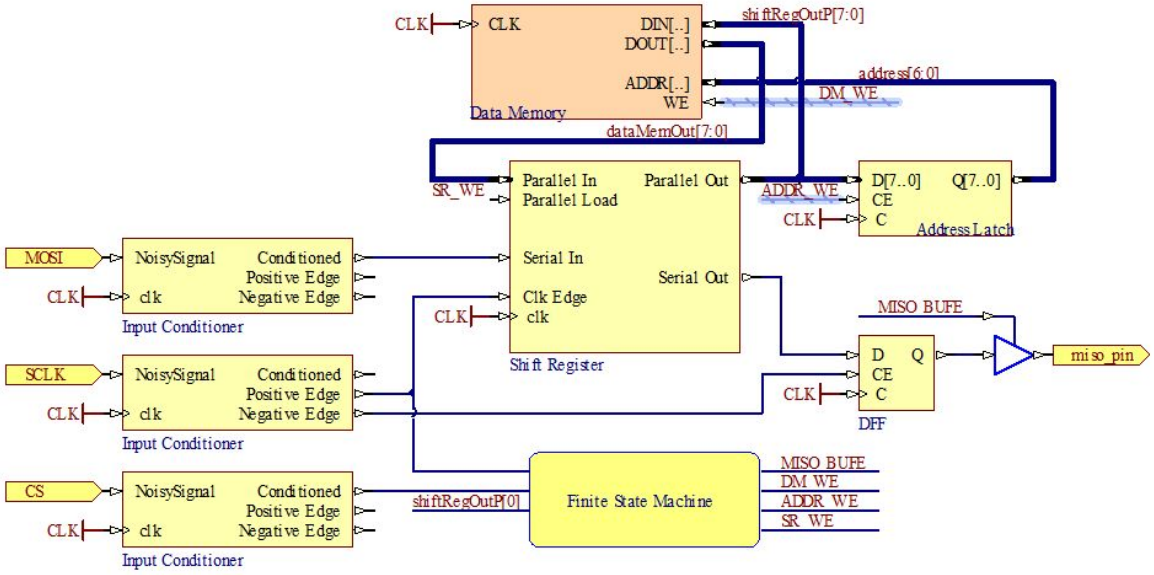


During the START state, we want all of our control values in our SPI memory to be 0. During the GET_BITS state we need to make sure parallel load in our shift register (sr_we) is 0 so that we can take in serial inputs instead of just reading from parallel data in. We also want to set the address that we will want to read or write from in data memory using the address latch at this stage. Therefore, we will want to set addr_we to 1. During the WRITE state, we need to set the data memory write enable (dm_we) to 1 so that we write the current value to the data memory. During our first READ1 state we set shift register write enable (sr_we) to 1 so that parallel load is asserted and the value coming into the shift register from data memory is sent out to parallel data out. In the next clock cycle (now that we have read the value from data memory) we can assert miso_bufe to 1 which effectively outputs the now loaded value of MISO. Our full control table can be seen below:

State	SR_WE	ADDR_WE	DM_WE	MISO_BUFE
START	0	0	0	0
GET_BITS	0	1	0	0
WRITE	0	0	1	0
READ1	1	0	0	0
READ2	0	0	0	1
END	0	0	0	0

SPI Memory

The last step was to put together all the pieces we described in the steps above to make our fully functioning SPI Memory. We used the following schematic to construct our SPI Memory:



Testing Strategy

For testing, we wanted to check each part of the process. A great way to accomplish this is with the handy finite state machine that handles the change in states. Our tests included the following process: check if in the start state -> set chip select high -> check in GET_BITS state -> add 11000000 bitstring -> check in Write state -> check the data value in the data memory -> check output of shift register -> write values 00110000 -> set Start state -> set GET_BITS state -> enter Read state.

Comp Arch: Lab 2
10/10/2018
Vicky McDermott
Minju Kang
Allison Basore

We found many bugs and solutions via our test bench. The various tests we laid out were designed to monitor the states of the SPI memory and the values of certain key input and output variables during certain states. One important note is that the SPI memory currently has a bug that is yet to be solved. Our last test, test #7. Here we were testing to check for our expected read value of 00110000. However, we found an error in one particular case of our code when we are transiting out of the write state and into the read state. In this particular case, we found an off by one clock cycle error where the read indicator value was not in the read/write variable digit. Instead, it was shifted by one after a clock cycle output by the shift register. This is in the process of further evaluation.

Work Plan Reflection

According to our original work plan, we were a day and a half behind on finishing the midpoint check-in. Otherwise, our timing was fairly well planned. Our estimations for certain things were less well calculated. Many of our estimation issues arose when we had to backtrack to find an error after attempted implementations. We realized two major areas for improvement. Firstly, we would like to work on running tests as we build the module. Often we run into time draining errors when we try to code the whole module, integrate and then test afterwards. Secondly, we plan to preload effort into understanding a module or part of the circuit before we start coding it. This will allow us to not make a large error that will have to be totally changed after we discover a bug. Altogether, we feel satisfied with the conclusion of this lab and adhering to the work plan.