Lab 2: SPI Memory

Emma Westerhoff and Samantha Young

October 23, 2018

1 Input Conditioning

The input conditioner has three jobs: input synchronization, input debouncing, and edge detection. Below we've included our circuit diagrams for the structural input conditioner.

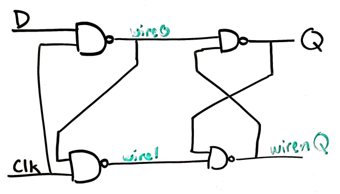

Figure 1: Latched D Flip-Flop

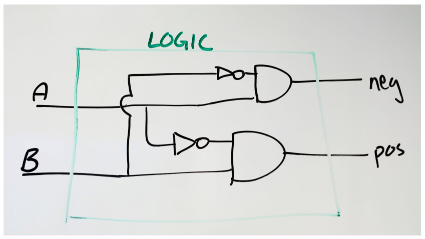

Figure 2: Top-Level Module Diagram

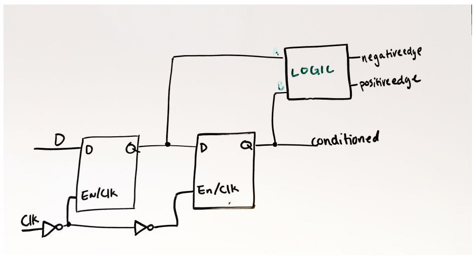

Figure 3: Edge Signal Logic Block

If the main system clock is running at 50MHz, what is the maximum length input glitch that will be suppressed by this design for a waittime of 10?

For the worst case, we assume a signal that starts directly after a positive clock edge. When finding this duration of the signal, it's start time can be the clock edge. This design waits 10 rising clock edges to ensure that there isn't a glitch. By the first clock edge, an entire period has passed. On the ninth clock edge, nine periods have passed. If we assume the signal changes again just before the tenth edge, then this glitch will be suppressed. The length of this glitch is just under 10 periods long.

For a 50 MHz clock, there is an associated period of 20 nanoseconds. For ten periods, this means that the maximum suppressed input glitch can be just under 200 nanoseconds.

2 Shift Register

The first case that we test is PIPO, where parallel load is always asserted. The shift register has some initial 8 bits stored in it. We change parallelDataIn, then observe the output. We test three cases, one of which is all zeros. We observe that PIPO is working as intended.

Our second case is SISO and SIPO, which are checked independently. Here, the parallelLoad stays low for the entire transaction. We test that serial data shows the LSB for both ones and zeros, and that parallel data shows the new serial inputs in the correct positions.

Our third and fourth cases are combined. We have independently tested each of the modes of operation of our device, but want to ensure that it can operate under changing conditions. Parallel load is toggled, and we change the data staggered from the clock edges. We observe that, in the output, the changes are read and outputed correctly on the clock edges.

3 Finite State Machine

We modeled our FSM and SPI memory off of the provided schematic. There are two major changes, the FSM takes inputs "falling edge of serial clock" and "MOSI conditioned"

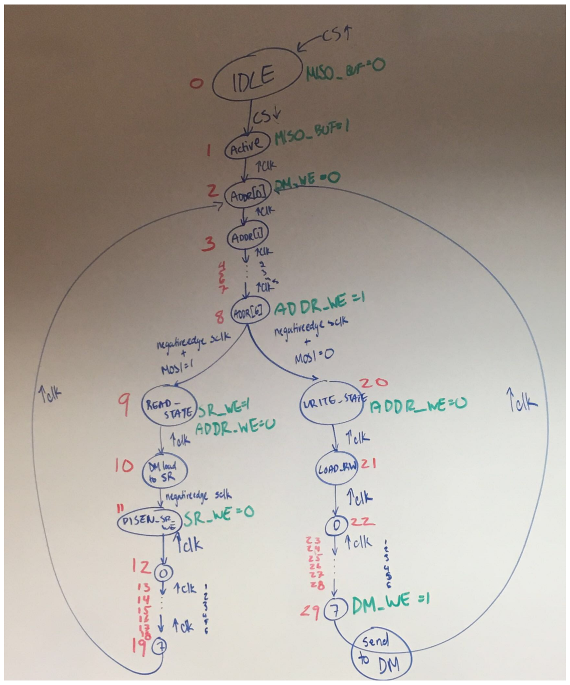

Figure 4: FSM Diagram

Our control table is embedded in our FSM diagram. We note the changing of states in green. Unless otherwise denoted, the other three control signals stay what they were set to in an earlier state.

One control signal that is not shown is something we called MISO-EN. This control signal is high for states 11-19, and low otherwise. The purpose of this control signal is to force the miso pin to zero when we are not reading data onto it.

4 Final SPI Testing

We tested a sequence of events for our SPI memory. Our first testing step was testing the FSM. We gave the test bench a series of 16 bit inputs, with both read and write modes. We watched the FSM step through the states as predicted.

We tested functionality of the fully constructed SPI memory slowly. We first tested the chip select functionality: when chip select was high, the miso pin was floating, as the protocol demanded.

We then tested the writing implementation by setting up a test bench to feed a simplistic 16 bit write command. We found an unconnected wire, and several other discrepancies in our code like this. However, this process moved fairly quickly. We noticed that during the state in which the SPI memory was receiving an address, the miso pin was copying that address. We created MISO-EN, which sets the output to zero in all cases except for when we are reading data to miso pin.

From here, we tested the reading implementation. We quickly discovered that if the memory is instructed to read from an empty memory, our output is undefined. To test the read implementation, we first wrote to an address. Then we tried to read from the same address. After some minor debugging, we were able to get the functionality working. However, our output signal is set high for a single clock cycle where it should be low. This occurs only after a read operations. We are currently still in the process of debugging this.

Our next step was writing to multiple address, and being able to overwrite what was already in an address. We also tested back to back requests. Again, functionality was achieved except for the blip on the miso pin.

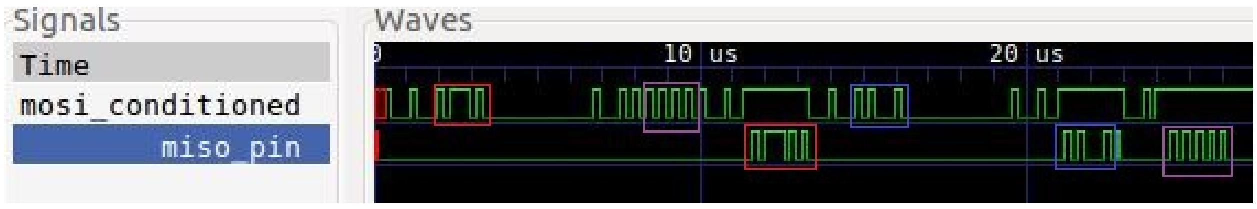

Figure 5: GTKWAVE simulation

5 Work Plan Reflection

1. Input Conditioning Module
   (a) Expected: 4.5 hours
   (b) Actual: 1.5 hours to write behavioral and structural implementations, 1 hour to write test bench, 1 hour spent debugging
2. Shift Register Module
   (a) Expected: 3 hours
   (b) Actual: 30 minutes to write .v file, 1 hour to write test bench, 3.5 hours spent debugging
3. Midpoint FPGA Load
   (a) Expected: 2 hours
   (b) Actual: 2 hours
4. SPI Memory
   (a) Expected: 3 hours
   (b) Actual: 5 hours to plan FSM, 6 hours to implement

5. Report
    (a) Expected: 4 hours
    (b) Actual: 3 hours

In end, we stuck very closely to our predicted amounts at the beginning of the project. We had to finish this project early, due to attending the SWE Conference. It was very important that we accurately plan out the amount of time each step would take, and stick to the plan. The SPI memory was the only real deviation from this plan. We set out 3 hours for drafting and implementation. Drafting alone took 5 hours due to multiple drafts and understanding how to create a state machine that was easily implementable. The implementation itself went fairly quickly: the time consuming step in this process was building a test bench and ensuring it worked as needed.