Computer Architecture Lab 2: SPI Memory

David Zhu, Bonnie Ishiguro, Zarin Bhuiyan

October 2016

Input Conditioner

Circuit Diagram

Figure 4 demonstrates some abstractions we've made to make the overall input conditioner schematic, shown in Figure 2, legible.

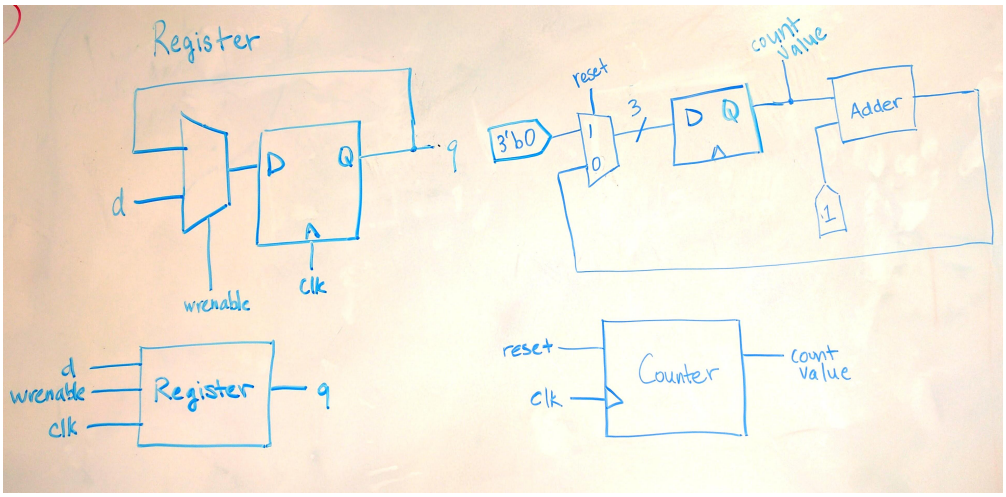

Figure 1: This illustrates the abstractions that we made.

Maximum Length Input Glitch

The maximum length input glitch that will be suppressed by this diagram for a waittime of 10 when the system clock is running at 50MHz is:

10 / 50MHz - epsilon <= 200ns

This is because any noise signal that is longer than this length would be treated as actual signal. At 200ns, our input conditioner's counter would hit our waittime and we would accept the noisy signal as valid.

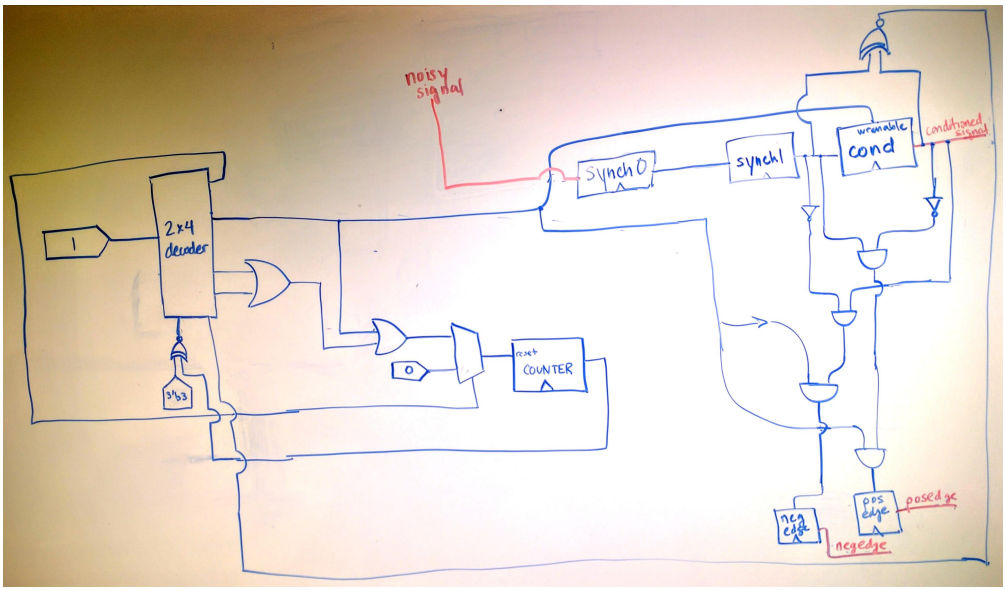

Figure 2: This shows our input conditioner schematic.

Shift Register

To create a shift register that supports both "Serial In, Parallel Out" and "Parallel In, Serial Out" modes of operation, we created a module in which the shift register is clocked by the main system oscillator clock that is running at 50 MHz. We designed for four behaviors:

1) When the peripheral clock edge is high, the shift register advances a single position, allowing for serialDataIn to be loaded into the least significant bit and the rest of the bits to move up by one.

2) When parallelLoad is asserted, the shift register assumes the value of parallelDataIn.

3) The serialDataOut is always the most significant bit of the shift register.

4) The parallelDataOut is always the entirety of the contents of the shift register.

We also made it so that if a parallel load and a serial shift happen in the same clock edge, the shift register ignores the posedge of the clk and High of the peripheralClkEdge, and on the next posedge of the clk sets the shiftregistermem to be the parallelDataIn.

Test Bench Strategy

We designed three tests to cover the "Serial In, Parallel Out," "Parallel In, Serial Out," and "Parallel In, Parallel Out" cases. For each case we pass in 8 bits of data and check to see if the input data matches the output of the shift register. To simulate "Serial In," we loop through the bit sequence, passing the current bit into serialDataIn and ticking the peripheral clock. To simulate "Parallel In," we pass the whole bit sequence into parallelDataIn.

The test bench can be found at ‘shiftregister.t.v‘.

Midpoint

FPGA Test Sequence

To test our top-level module for the shift register, we load it onto the FPGA and ran through a testing process listed here:

1) Press Button 0 to set the input mode to ”Serial In.”

2) Set Switch0 high.

3) Toggle Switch1 from low to high to write ”1” to LED0.

4) Repeat step 3 to serially write 1 bit into the shiftregister. Since Switch0 is set to high, we will be writing 1 bit on the positive clock edge (through toggling Switch0).

5) Set Switch0 low.

6) Repeat step 3 to serially write 0 bit into the shift register. This is because Switch0 is zero, meaning our input signal is low.

7) Press Button 0 (which stands for parallelIn) to write all data bits into the shiftregister. Since we’ve hardcoded a number (10100101), this number will be parallel loaded, and remove all prior serial loading.

The demo of this test can be found here: https://www.youtube.com/watch?v=XQpx6vsjJXQ

FSM Module

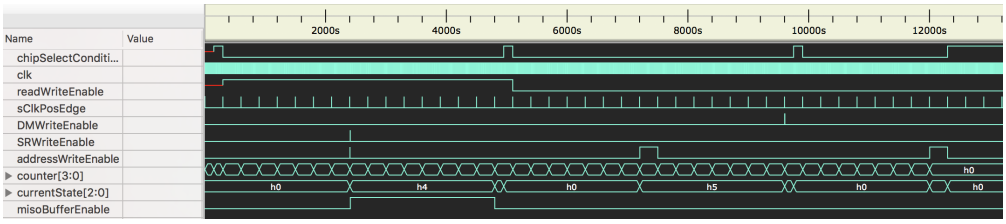

Figure 3: This is our GTKWave waveform for the finite state machine module.

Test Bench Strategy

To test our FSM, we decided to black box the module and test its expected outputs when we provide a certain timed sequence into the input. As we drive a fast internal clock, we pass in simulated

blips for the sClkPosEdge to trigger internal FSM events.

The three main tests we've written checks for whether reading, writing, or resetting works for our FSM. To write these tests, we simulate 8 sClkPosEdge blips and then compare the outputs of misoBufferEnable, DMWriteEnable, addressWriteEnable, and SRWriteEnable to our expected outputs. On our reset test, we set the chipSelectConditioned signal back to high and wait for a short time before taking the pulse of our outputs, which would then all be reset to 0.

It was difficult to write these tests because the blips for the DMWriteEnable, SRWriteEnable, addressWriteEnable, and sClkPosEdge were tiny compared to the other waveforms (2 seconds).

SPI Memory

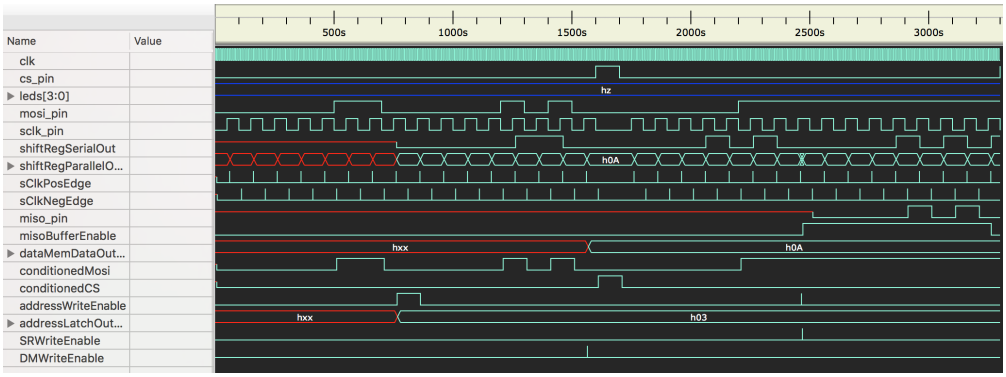

Figure 4: This is our GTKWave waveform for our SPI module.

Test Bench Strategy

We originally and ambitiously wanted to test our code directly through C using Vivado. However, after being unsuccessful at running our write and read commands for our SPI, we had to do some more Verilog debugging for all of our subcomponents.

We wrote additional test files for spimemory, addresslatch, datamemory, d-flipflop, and revisited the FSM. By converting the provided C test into Verilog for 'spimemory.v', we were able to eventually detect that the wiring connections for the subcomponents and the FSM's timing were wrong.

For the FSM, we revisited what type of output each state provided. One issue we faced is that the clocked FSM always exihibited its outputs 1 clock cycle after it reaches that state. After using GTKWave and better test timing checks, we were able to align the FSM to the proper behavior.

Within the Verilog tests for the SPI memory, we tested to see if it could read and write to an address, which it was successful in.

The default test for the SPI memory on Vivado was unsuccessful.

Workplan Reflection

Since this lab contained more steps than before, we decided to use Trello to manage our sub-tasks. Trello turned out to be less useful than we expected, because a lot of tasks blocked each other. It made more sense for the team as a whole to learn each step of the lab construction, and we worked together in this regard.

One area that we spent more time than expected was constructing the FSM module for providing correct control signals for the rest of our components. It took us a long time to realize that the module was driven internally by a master clock that was running a few orders of magnitude faster than our peripheral clock. Also, the correct control outputs happen at very precise time periods depending on which state we were in, which made it hard to test the expected output if our expected wait time was wrong. Since the tests for the FSM were not high quality at first, debugging the system later became much trickier. We had to revisit this module several times before getting the behavior correct.

Other areas that took considerable time are loading the code onto the FPGA and correctly debugging misbuilt circuits. There were many steps involved in loading in our Verilog code and building and compiling it for our FPGA. At times, we were lost in which buttons we forgot to press, and we had to be unblocked by a NINJA. Finally, C code could not tell us what was broken in our circuit implementation, only that something was broken. We not only had to revisit and write more lower level tests, we had to put trust in areas that were much more difficult to test - for instance the connections and timings between submodules.

Ultimately, the lab took much more time than we've