Lab 2: SPI Lab

Ryan Eggert, Meg McCauley, Shane Skikne

9 November 2015

1 Input Conditioner

If we are given a 50MHz clock signal and a desired `waittime` of 10, we can set the `counterwidth` parameter to 4 by the given relationship $\texttt{counterwidth} \geq \log_2 \texttt{waittime}$. This ensures that our counter has sufficient bits to count up to 10 clock cycles. By setting `waittime` to 10, we are instructing our conditioner to count for 10 clock cycles upon recognition of a change to the synchronized input. If, throughout this delay, the synchronized input has remained stable, then the input conditioner will adjust the conditioned output signal accordingly. At a 50MHz clock, one clock cycle is 20ns, so any input glitch of 200ns $[{}^{20\text{ns}}/_{10\text{ cycles}} * 10 \text{ cycles} = 200\text{ns}]$ or less will be suppressed.

A circuit diagram of our input conditioner can be seen in Figure 1 on page 3.

2 Shift Register

To test our shift register, we broke tests down into 4 categories of loading data and tested each to make sure the output was correct. The 4 types are: Setting everything to 0, Parallel Loading, Serial Loading and Conflict between Parallel and Serial Loading. First, we tested just instantiating and loading all zeros into the register. We then simply ensure that the register has correctly loaded all zeros and isn't completely dysfunctional. This doesn't really test much behavior, but did allow us to make sure the Shift Register would compile.

The second and third test ensured that our serial loading and parallel loading, respectively, worked. We chose not to test Parallel In, Serial Out and Serial In, Parallel Out directly and rather just checked that both Parallel Out and Serial Out were correct at all times. These tests just checked that whatever data was loaded into both Parallel Out and Serial Out were correct.

Finally, we test how the shift register responds to a load conflict, where we use Serial In and Parallel In at the same time. In our design of shift register, we decided that Parallel In would take precedence in situations like this. Because of this, we ensure that that the Parallel In data is correctly loaded into Serial Out (which should be 1) and Parallel Out (which should be 11111111_2).

3 SPI Memory Testing

We perform one large test check that our SPI memory is functioning correctly. We comprehensively check our ability to read and write values to every address. To do this, we write an 8-bit value to an address and then immediately ensure that we can read that value back from the address. This test is repeated by writing every possible 8-bit value in sequence to an address and then by performing this over every possible 7-bit address. When this test passes, we are fairly confident that our SPI memory is functional. This test, while it ensures that writing and reading works for each address, doesn't check whether writing or reading other addresses affects the overall memory in any unexpected ways.

To make our tests more comprehensive, it would be nice to create a second test in which we first write a unique value to every address. Then we would read each address and ensure it is what we expected. Unlike the first test, this checks that the values we write are still saved after we either write to or read from every other address. With this test, we could assume that working with data from one address won't have any effect on data from a different address.

4 Fault Injection

Although we ran out of time on this lab to implement our fault injection, we have come up with a plan for what we would have broken in our code if we had the time. We would choose to break the address write enable (ADDR_WE) that is outputted from the finite state machine and inputted into the address latch (see Figure 2 on page 3). If we had broken the address write enable as indicated, it would always be set low and therefore, the finite state machine would not be able to indicate to the address latch that it was trying to tell the address to store the data. A specific test that could be run in order to check this failure mode is attempting to store any piece of data to any address other than zero. If the read of that address did not come back correctly, it would show that our fault injection was completed correctly.

5 Work Plan Reflection

For this lab, we found our work plan generally helpful until we got to SPI Memory and loading it onto the FPGA. Up until that point, our plan was generally accurate, with the pieces being broken in pretty even ways. Once we got done with the finite state machine, though, things started to take significantly longer from our work plan expectations as we had to battle everything from Verilog to C to the FPGA/Vivado. We were certainly not expecting any of the challenges there. Our SPI C tests were delayed by an interesting (and not-well-understood) behavior in which the inclusion of a `xil_printf` statement caused our tests to fail. Our fault injector plans were delayed by unexpected trouble finding the correct syntax/methods to connect a physical switch on the board to a packaged module through the "new" block diagram interface. In the future, we think we will aim to finish a few days earlier so we have more time to handle unexpected issues like this. Additionally, we will give a little more thought to our work plan and give each component 10 minutes of initial thought and work in order to get a better idea of how big it is and what it will entail. In the past, we'd just read the lab and estimated, but spending a few minutes in the beginning discussing our plan of attack will allow us to have a better idea of what we are going to accomplish.

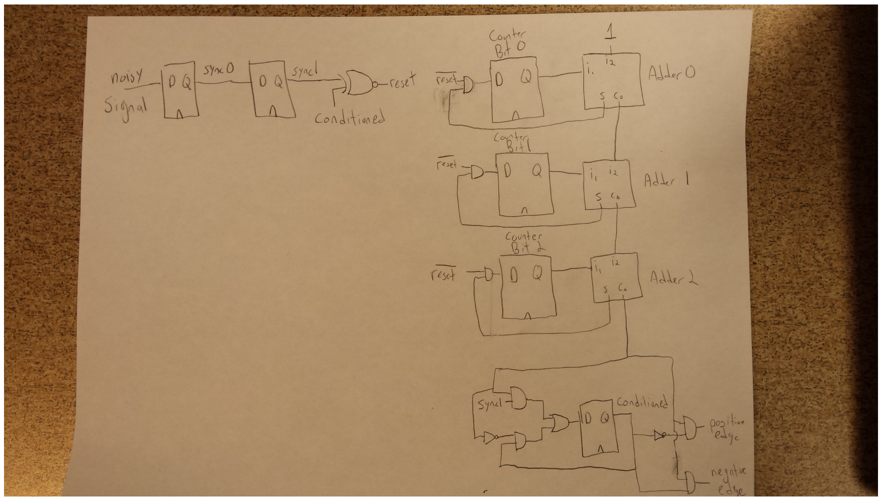

Figure 1: *A circuit diagram of the structural circuit of our input conditioner.*

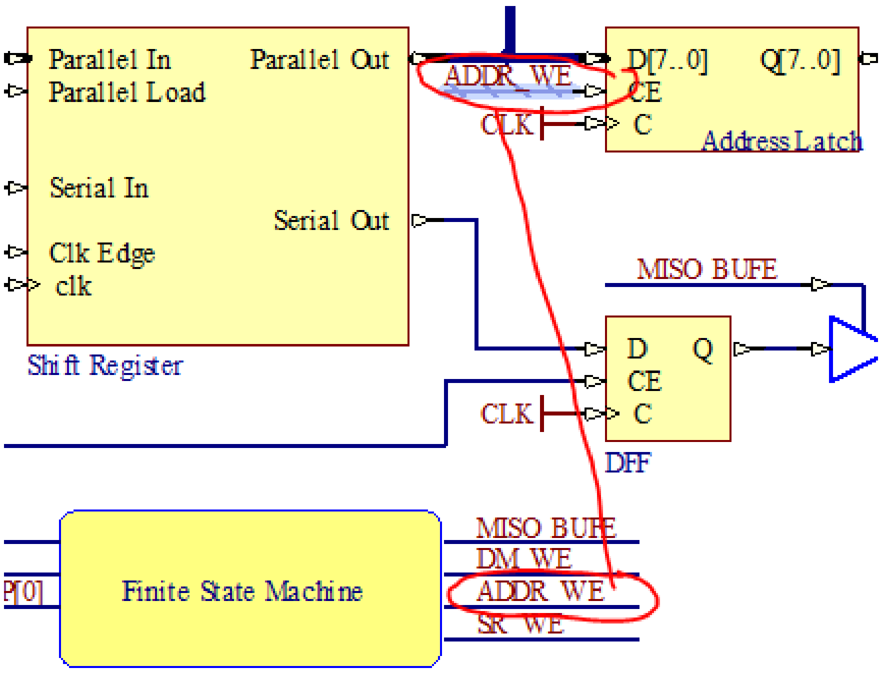

Figure 2: *A schematic detailing the fault that we are inserting. See the ADDR_WE circled in red.*