CompArch Lab 2: SPI Memory

Jacob Riedel, Jennifer Vaccaro, Jennifer Wei

November 9, 2015

1 Input Conditioner

1.1 Circuit Diagram

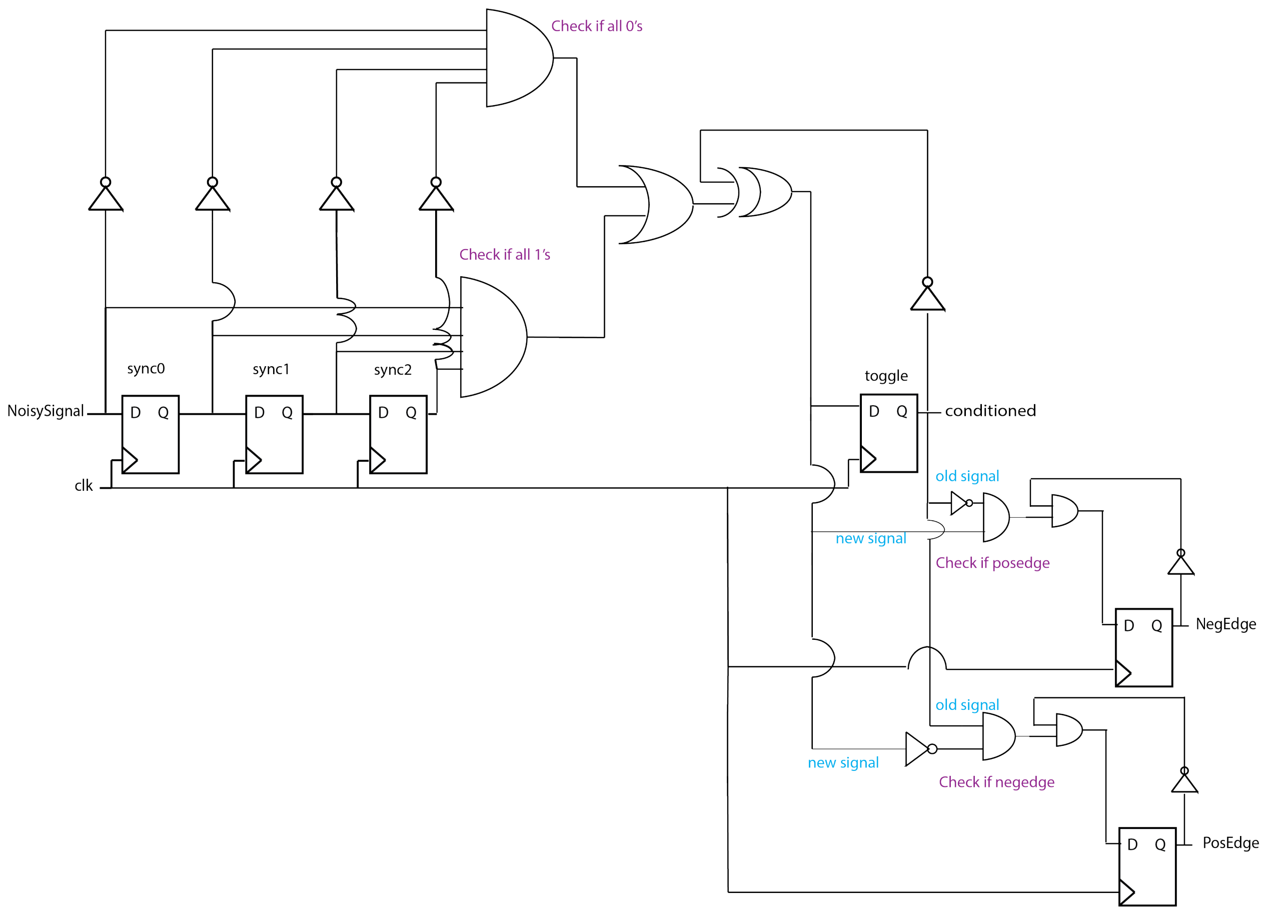

Figure 1: Circuit Diagram of Structural Input Conditioner (variable names in black, descriptors in blue, and checks in purple)

An input conditioner acts as an input synchronizer and debouncer with edge detection. In Figure 1, there are essentially three parts, the checker, the toggle, and the edge detectors. For the checker, the noisy signal goes through three clocks since the wait time set for this lab is three clock cycles. The four inputs/outputs of these clocks are then passed through two gates, an `AND` gate and a NOTed `AND` gate, and the output from those two `AND` gates go through an `OR` gate. This is to check if all the values are '1's or '0's as just using one

`AND` gate will pass a '0' if all '0's are passed in, which is not what we want - we want the output to check if all four values are the same to account for the debouncing and syncing.

From there, the checker output becomes the input to a T "toggle" flip-flop. If `T` is high, the value changes. Otherwise, `Q` stays the same. To create the T flip-flop, we took a `D` flip-flop and the inverse of the conditioned output, `Q`, and used an `XOR` between that and the output from the check.

Lastly, the conditioned output `Q` is compared to the input of the toggle, `D` using `AND` gates where `D` is the new signal and `Q` is the old one (`D`=1 and `Q`=0 for `posedge` and the opposite for `negedge`). These are then fed into modified D flip-flops such that the edge flag only exists for one clock cycle.

1.2 Maximum Length Input Glitch

In the current system we have a `waittime` of 3 clock cycles with the clock running at 50 MHz, so the period is 1/frequency (or 1/50 MHz) or 20 ns per clock cycle. For a wait time of 10 clock cycles, the total time would be 200 ns. Thus, the maximum length input glitch that would be suppressed by this is one less than the total time, which is a glitch of 199 ns.

2 Shift Register Test Strategy

The behavior of the shift register is as follows:

- at `peripheralClkEdge` edge → register advances one position: `serialDataIn` is the LSB and all other bits shift left by one
- when `parallelLoad` is high → shift register takes the value of `parallelDataIn`
- `serialDataOut` always presents the MSB of the shift register
- `parallelDataOut` always presents all content of the shift register

With that in mind, to test the shift register, we test the two cases: parallel input and serial input - which are determined based off the state of `parallelLoad` - and the `peripheralClkEdge` toggle to ensure that `serialDataOut` does not change with `serialDataIn` if `peripheralClkEdge` is low.

In our Verilog file, we set up several test cases, looking at the generated waves to check that each one works as intended.

In our first test case, we test serial where we check that a bit can be passed in. When `peripheralClkEdge` set back to low, we attempt to pass more bits into `serialDataIn` but those do not show up in `parallelDataOut`, which is what we expect.

In our second test case, we test parallel by checking that passing in an 8-bit works. We then toggle `peripheralClkEdge`, test that another 8-bit passed in via `parallelDataIn` shows up in `parallelDataOut`. Finally, we set `peripheralClkEdge` back low and pass more 8-bits in to ensure that `parallelDataOut` equals that of `parallelDataIn`

In our third test case, we do another test of serial, toggling `peripheralClkEdge` between the high and low states to ensure that the bit shifting works as intended, shifting the bits through from LSB to MSB on each clock edge.

2.1 SPI ARM Processor Test Strategy

In order to test our full SPI module, we used the ARM processor on Vivado. As we will note in our reflection, this process gave us the most difficulty in terms of time needed to succeed. After the SPI memory module was written, packaged and ready to go from Vivado, problems started to occur. We were able to connect the module to the wrapper fairly easily, but our team lacks any knowledge of `C`. This made programing the SDK harder than it should have been. After a couple of hours, we had code that should have been fully functional, but somehow, there was an error. Every read operation returned the same hex value, regardless of what had been written. We talked with ninjas but they were just as confused as we were. Eventually, they compared their written `C` function to ours but saw no differences. After several more hours, by swapping

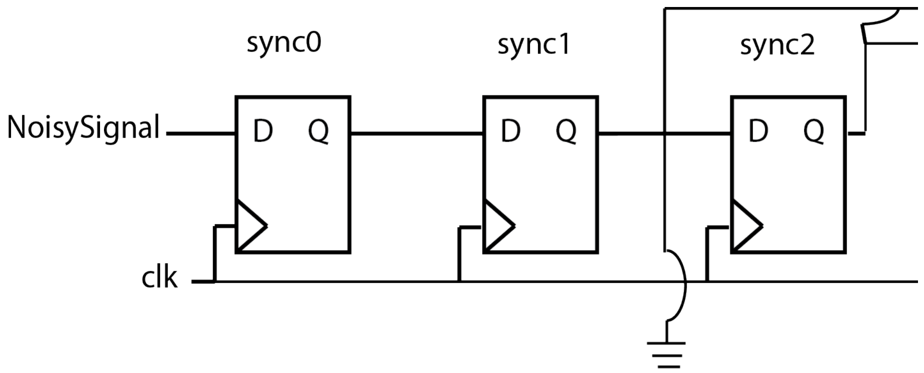

Figure 2: This snippet of the schematic above shows where the `D` flip-flop is grounded, so that the input conditioner only outputs zero.

the 2 lines for the definitions of the send and receive buffers (i.e. `RecvBuf` first then `SendBuf`) the code final worked. We have no idea as to why these two lines, which seem unrelated caused such a strange issue.

Another major issue that we ran into was state saving in the SDK. When running the working `C` code, if we wrote to `Register A` in data memory, then read from `A`, we received the value that we wrote. If we then tried to read from `A` again, even just as the next operation after the last read from `A`, the value would be 0. It seemed that somehow the value in any register could only be read once, even though it should be held until it is overwritten. We know that our SPI memory module works correctly because we were able to show that we can read from the same register more than once and get back the correct value during fault injection testing in ModelSim.

The final issue that arose was the fault injection pin hook-up. Somehow, we were unable to hook up the `fault_pin` to the switch in Vivado. Through testing in ModelSim (see below) we know that fault injection works in our system, but the code seemed unresponsive during the attempts to get it working in the SDK. We are unsure as to why it did not seem to affect the code, as we did propagate the signal correctly through the multiple layers of Verilog.

3 Fault Injection Testing

Our injectable failure mode is that the output to a synchronizing `D` flip-flop in our Input Conditioner is set permanently to zero. This could occur is a wire connected to the output of the `D` flip-flop was shorted to ground. We simulate this by setting the output `sync1` to zero every clock cycle. This can be identified in testing by demonstrating that the output to our Input Conditioner is always zeros.

3.1 Schematic

Figure 2 shows a zoom-in of the second D flip-flop, `sync1`, that we ground to trigger a failure mode.

3.2 Specific Test Pattern Description

To show that the fault injector on the input conditioner worka as intended, we setup the `Sclk` input conditioner as the faulty conditioner. When a fault occurs, the Sclk is held at ground, and no operations can occur that normally use this clock, including adding data to the shift register through serial read, and incrementing the counter for get, read and write operations on the finite state machine.

The following waveforms (Figures 3-7) demonstrate our fault injection. The composite image was unclear, so we broke the test into 4 segments. Each caption explains the test.

Unfortunately, the implementation of our Fault Injection is not functional on the Zybo at this stage in the design. This seems to be an error on the Vivado hardware level, not on the ModelSim simulation level.

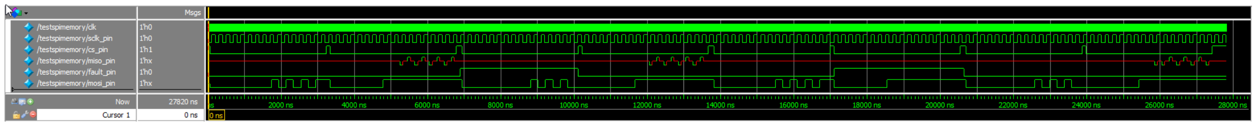

Figure 3: This show the whole test bench of the `SPImemory` module with fault injection.

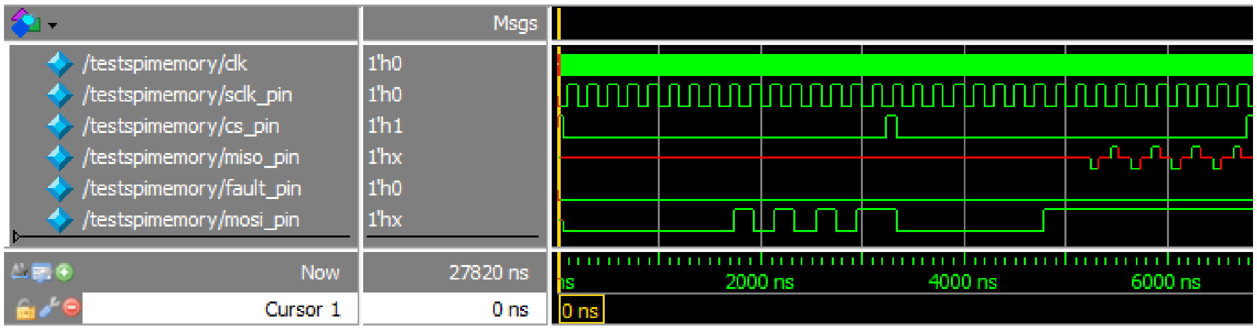

Figure 4: This show the first test bench of the `SPImemory` module with no active fault injection.

The first breakup of the waveform in Figure 4 shows a fully operational SPImemory module. This is expected because the fault pin is low and no faults should be injected. Everything runs well.

The second breakup of the waveform in Figure 5 has an active fault injection on the write operation of the SPI memory. This blocks the write from actually writing to the requested address in data memory. As a result, instead of writing a new byte to the address, nothing is written. The fault is then corrected before the read stage and the read pulls the old byte of data from the address. This also happens to show that our data memory holds state between different read and write operations.

The third breakup of the waveform in Figure 6 has an active fault injection during the read operation of the SPI memory. This prevents any data from being returned from the requested address during the read. The write operation is unhindered, as the fault injection is not asserted during this operation, so we can assume that the write function worked.

The forth and final breakdown of the waveform in Figure 7 has no active faults and shows what data should have been written during the second test, if the fault injection had not blocked the write operation. This new data is then read correctly through the read operation and the module seems to work flawlessly(except for the intended fault).

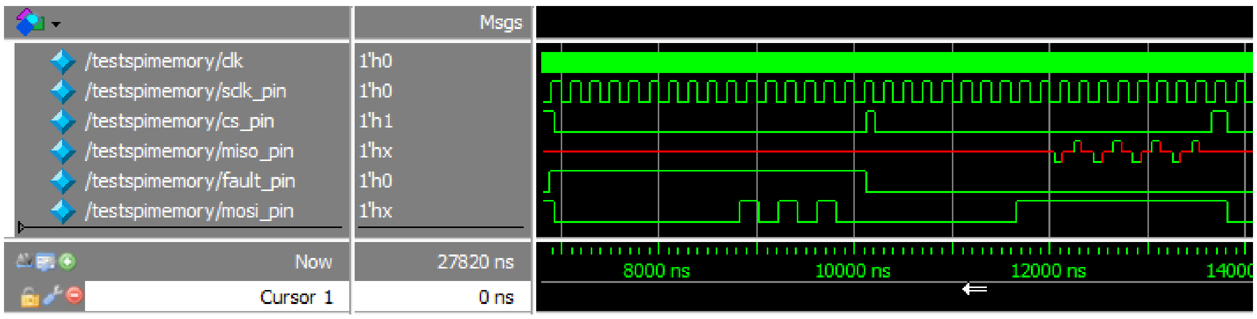

Figure 5: This show the whole test bench of the `SPImemory` module with fault injection during the write operation.

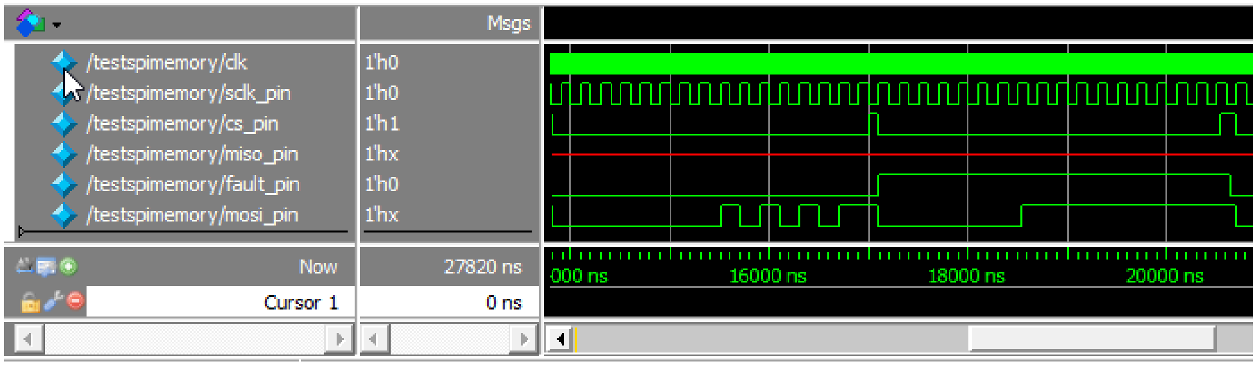

Figure 6: This show the whole test bench of the `SPImemory` module with fault injection during the read operation.

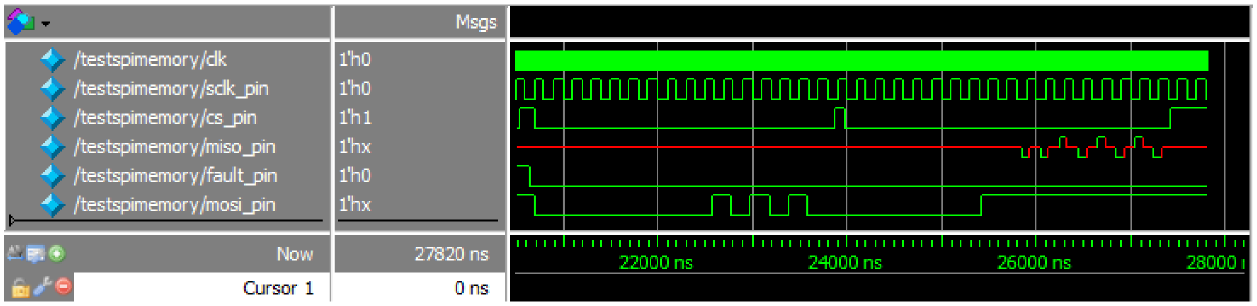

Figure 7: This show the whole test bench of the `SPImemory` module with no fault injection and a different write sequence.

4 Reflection

For this lab, we were on schedule up until the midpoint check-in - we even got checked off the night before! However, we hit some roadblocks when trying to test SPI Memory that caused us the fall a bit behind schedule both in terms of when we estimated to be done and also with our time estimations for how long each task would take. This was partially due to not being as thorough with earlier tasks, but we were able to resolve it in the end.

After that, we struggled a lot with the FPGA hardware upload and testing. We were eventually able to debug a unit to the success level of Kyle's design, and ended up having a working version without fault injection. We have a waveform which shows that the fault injection works in ModelSim, though we were unable to get it functional on the Zybo.