Comparch Lab 2: SPI Memory

Lisa Hachmann, Tom Heale, Anisha Nakagawa

October 30, 2016

1 Introduction

For this lab, we implemented SPI (Serial Peripheral Interface) memory on the FPGA using Verilog. For the first half, we created input conditioner and shift register modules, while for the second half we created a finite state machine and connected the modules shown in Figure 1 to create a SPI Memory module that will act as a slave to the FPGA master.

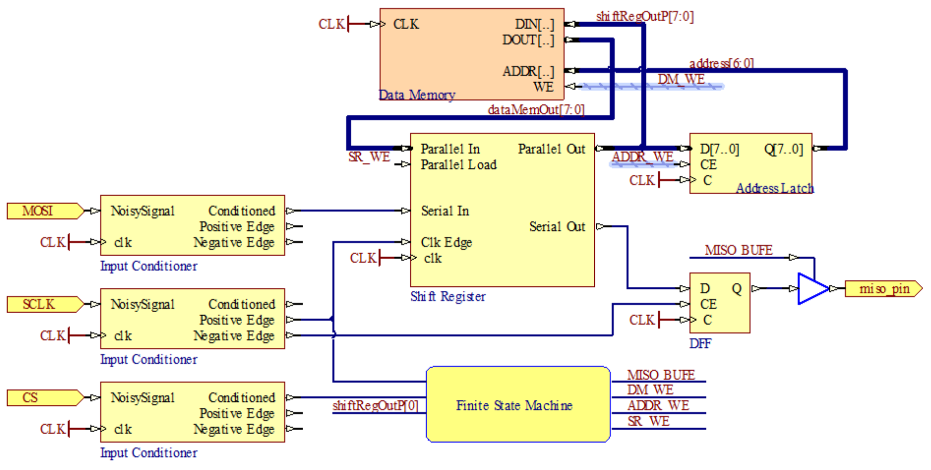

Figure 1: Schematic for the implemented SPI. The only changes made (not represented) were changing the CE for the DFF from negative edge of sclk to the miso buffer control signal and wiring fast clk into the FSM.

1.1 Finite State Machine Behavior

The finite state machine controls all the control signals to propagate the correct behavior of the SPI memory module. The state machine can be represented by the block diagram shown in Figure 2.

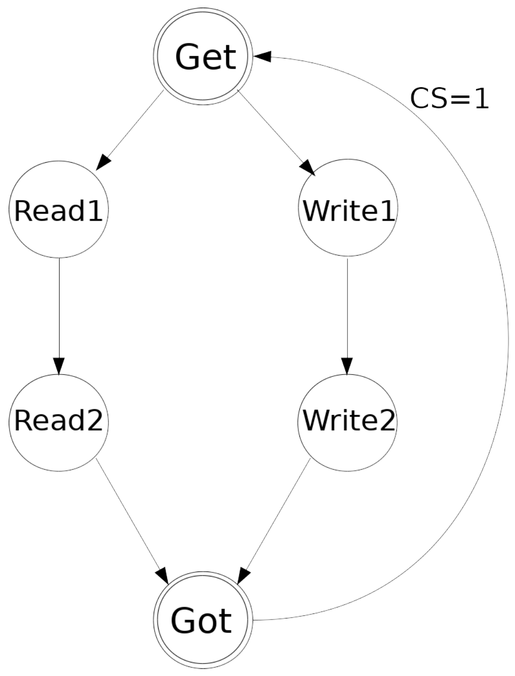

Figure 2: Finite State Machine Implementation

The process of going through the states is triggered when chip select goes low. The first state is the Get Address state, where the slave reads the 8 bit instruction from master. The first 7 bits are the address, and the least significant bit is the read/write flag. The SPI module stores this address to the address port of the Data Memory module, by setting the pin of Address Write Enable to true.

In order to read the address, it has to wait for 8 clock cycles as one bit of the address comes in at a time. This is controlled by the counter, which gets updated on the positive edge of each peripheral clock cycle.

The LSB of the instruction is the read/write flag. When this flag is high, it means that the SPI should go to the Read states. In the Read1 state, the address write-enable flag is set low so that we stop writing to the address. We also set ParallelLoad from the shift register so that we read in the data from data memory via parallel data input. Then in the Read2 state, we shift out that value of data through the serial out pin of the shift register. It takes 8 peripheral clock cycles to shift those 8 bits out, so we wait for the counter to count to 8.

When the read/write flag is low, we go to the write states of the finite state machine. In the Write1 state, we send the signal to stop writing to the address. In the Write2 state, we take each bit of the data from master and write it to the shift register serial input. We store the 8 bit parallel output from the shift register into the data memory module. This takes another 8 cycles of the peripheral clock, which is controlled by waiting for the counter to reach 8 cycles.

Finally, both the Read2 and Write2 states transition to the done state. This state resets all the writing pins, and just waits until Chip Select goes high - which starts off another cycle of the FSM.

2 Input Conditioner Testing

In Figure 4 on page 4, the wave forms of the input conditioner are shown. A noisy signal is put into the input conditioner, and it ”debounces” the signal correctly so that it can deduce if the signal is meant to be high or low. In the first half of the wave form, the noisy signal is a perfect square, and it is reflected in the conditioned signal a few clock cycles later. In the second half, we show a very noisy signal (it drops to 0 at times) that is then translated into a perfect square on the conditioned signal line by the input conditioner.

In Figure 3 on the next page is a structural representative schematic of our Input Conditioner Module. It has 2 inputs, *noisysignal* and *clk*, and 3 outputs, *conditioned*, *posedge*, and *negedge*. There are also several immediates (constant values) and an intermediate wire value, counter. *Counter* iterates using a 2 bit adder

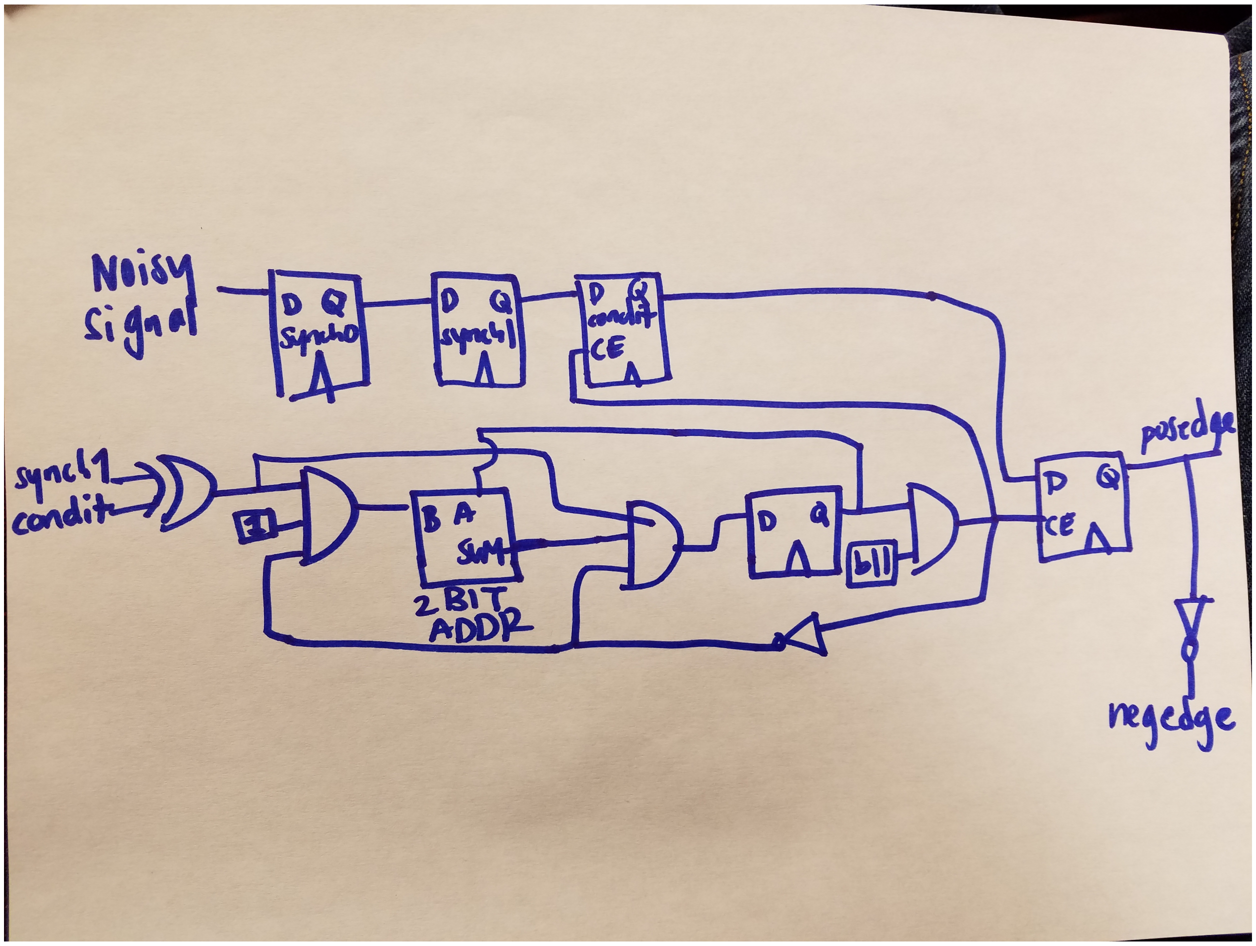

Figure 3: Circuit diagram for input conditioner

and a D-flipflop to give us the specified delay in our signal processing unit. In our circuit, *counter* acts as an enable pin to the flipflops holding the values of *conditioned*, *posedge*, and *negedge*, which blocks the outputs from updating until *noisysignal* has been fully "conditioned."

2.1 Waittime question

Q: 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? Include the analysis in your report.

A: If our clock ran at 50MHz, a single clock cycle would take 20ns to complete. If our *waittime* variable were 10, then our input conditioner would suppress any changes to noisy signal that occur in the 13 clock cycles following an initial change in the signal. This is because it will take 2 cycles to modify *synchronizer1* and compare it to *conditioned* to start the counter. Once the counter counts for 10 clock cycles, there is one more cycle before the value of *synchronizer1* is written to *conditioned*, our output. Thus, our Input Conditioner Module could suppress a glitch of less than 260ns.

The suppression of a glitch in *noisysignal* can be seen in Figure 4 on the following page. In this waveform, *conditioned* goes high at 3 separate points and holds for several clock cycles. In contrast, *noisysignal* goes low twice during the second *conditioned* plateua. These purposeful glitches were ignored by the Input Conditioner during output.

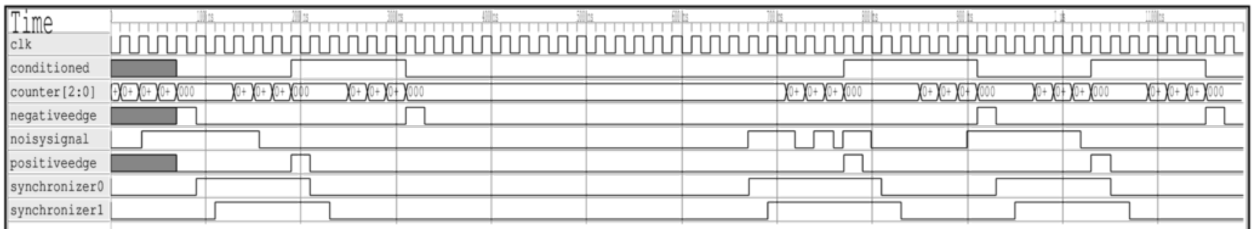

Figure 4: Waveform for the input conditioner test analysis

3 Shift Register Testing

The shift register has two main modes of operation. It can either take in an 8-bit parallel input all at once, or take in a 1-bit serial input over 8 clock cycles. The mode is controlled using the Parallel Load pin, where a high value means that the shift register will read in from parallel data in. In this case, the 8-bit value will immediately be written to the internal memory. In the case that Parallel Load is low, the value of the serial input will be written to internal memory in the least significant bit location. The rest of the bits of internal memory will shift left, which was implemented using concatenation.

The internal memory is written to the 8-bit parallel data out. Serial out is always the most significant bit of the internal memory. When serial bits are shifted in during clock cycles, the bits of internal memory are also shifted out one at a time, and can be read through serial data out.

We tested the shift register under both cases: when it takes parallel data in, and when it takes serial data in. We made the design decision to always write the parallel data if parallel load is high. Serial data will only be written on the rising edge of the clock, when parallel load is false.

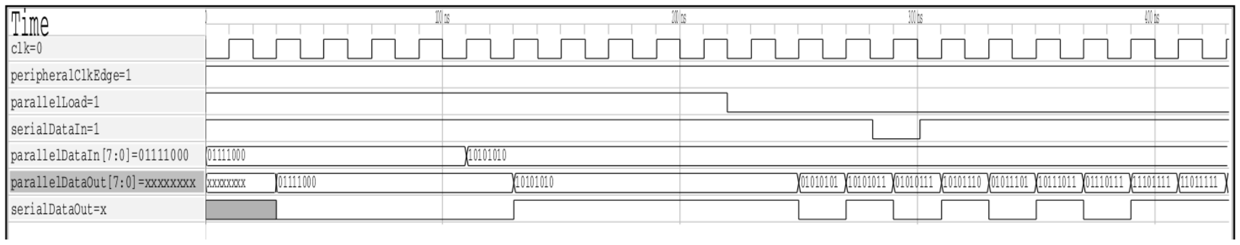

Figure 5: Waveform of the shift register module under both parallel in and serial in conditions.

The waveform of one of these tests is shown in Figure 5. In the first half of the waveform, parallel load is high, so the entire contents of parallel data in gets written to parallel data out. There is a delay of one clock cycle as expected, because the data is only processed once on the rising edge of clock. In this case, serial data out is just the least significant bit of parallel data out.

The second half of the waveform demonstrates the serial-in capability, when parallel load is low. In this case, the value of the serial-in bit gets written to the least significant bit of internal memory, which is the same value as parallel data out. With each clock cycle, these bits move to the left. We can see this represented in the 8-bit binary value of parallel data out. As we shift the data bit-by-bit, we can also see the serial-out pin output the previously stored memory one bit at a time. We ran multiple tests with different numbers to more fully test a variety of values. Both the parallel-in and serial-in cases work as expected, which verifies the functionality of this module.

4 SPI Memory Testing

To test the SPI Memory module that we created, we first made test benches in verilog to look at the signal simulation in GTKwave. Then we tested three main failure modes before moving to further testing on the FPGA.

4.1 Test benches in Verilog

In verilog, our test benches covered a couple cases. It tested and simulated: writing to an address then reading to an address, writing while chip select is high, writing to two different addresses and having addresses with edge case values (ex. b'1000001 or 65 in decimal).

The simulation of the first test case, writing and then reading from the same address, is shown below in figure 6. This test case was important to verify first, as it shows that both write and read are working, at least under normal conditions. Notice that data in the MOSI signal shown during state 40 (writing) is shown to be read on MISO during state 8 (read).

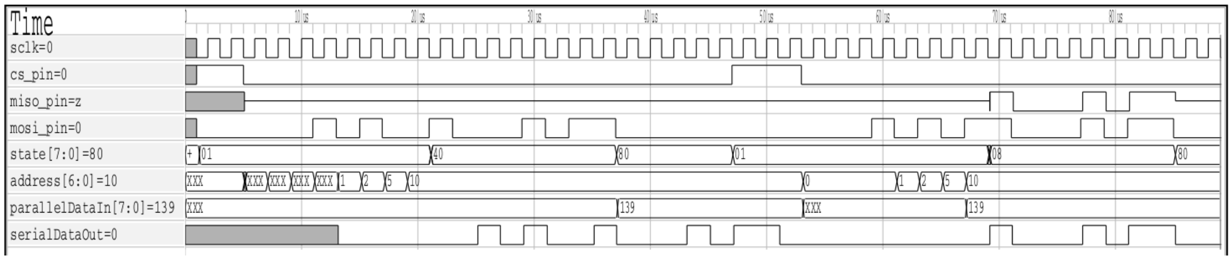

Figure 6: Waveform of the module for the test case where we write the value b'10001011 to the 10th address, and then read from the 10th address. Inspecting the miso_pin shows that the SPI outputs the correct signal for b'10001011 when the module is set to read (state 08 here). We chose the value of b'10001011 because it demonstrates that our module can correctly read and write the LSB and MSB of the value.

The second test case was to check if the slave would ignore all messages if chip select was asserted. Slave's behavior is to only listen when the chip select signal is low, and thus not read/write while chip select is high. Figure 7 shows that no response is invoked from MOSI's attempt to write to address 2 or read from address 2.

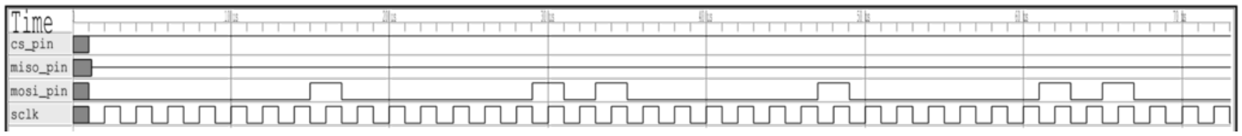

Figure 7: Testing MISO output when chip select is high. It should not listen when chip select is asserted, as shown. MISO stays in the high impedance state

The third test was made to show the ability to change addresses and have proper performance. This was done in figure 8 by writing to 2 different addresses: 2 and 65. This test bench also subtly tested that writing to an address with an edge case value (like the MSB and LSB's as 1's) still functions. With the major failure modes covered, we moved onto testing on the FPGA with C code.

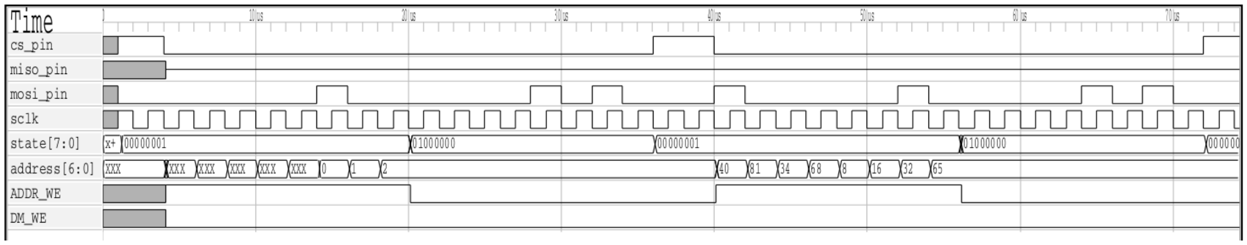

Figure 8: Testing writing to two different addresses

4.2 Tests on the PFGA with C code

Once we verified the behavior of the SPI module using verilog, we then tested it on the FPGA using code written in C. The test code communicates with the SPI slave using functions that allow the master to read from the slave. Both these functions use the same transfer protocol, where two bytes are sent to the SPI module, and two bytes are returned . In the spi_write function, two bytes are sent to the FPGA: the 7-bit address with a 0 flag for writing, and an 8-bit data value to write. In the read function, the address is sent to the SPI module, and an 8-bit data memory from the SPI is returned.

We tested out module by writing a known data value to an address of the SPI module, and then reading back the data from that same address. Since we know the value of the data that was written to the SPI memory, we want to confirm that we read the same value back. The SPI module would fail this test if these two values were different. We also looped through all possible addresses of the SPI, and we wrote a sweep of values to the SPI to test the range of its functionality. For readability and ease of debugging, we wrote the value of the address to that address (so we wrote the value 0 to address 0, wrote the value 1 to address 1, etc). Our SPI implementation passed all these tests.

5 Reflection

In our work plan, we had allocated 5 hours to General Confusion, which proved to be very accurate. For the next lab, we perhaps will even add time to this estimate. Our time estimate for the FPGA testing was accurate because our FPGA was broken. Once we got a better FPGA, this part of the lab went way more quickly. We were not as efficient as possible because we decided to work together as much as possible. However, we think this helped our collective understanding of all parts of the lab. Therefore, it was worth it.