SPI Memory

Ariana Olson and Prava Dhulipalla

Input Conditioner

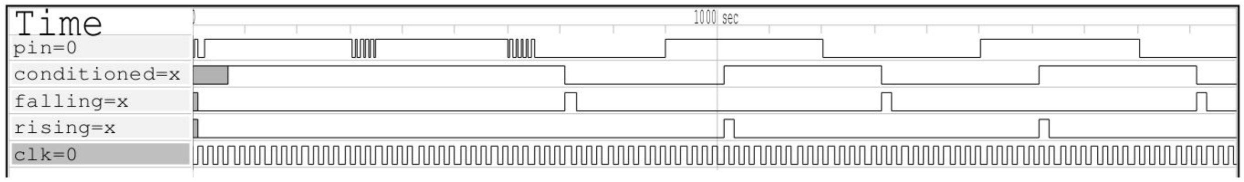
Implementation

See `inputconditioner.v` and `inputconditioner.t.v` for the Verilog module and test bench implementation.

The input conditioner module itself was fairly simple to instantiate as most of the code was already written - all we had to was add conditional statements to detect a negative and positive edge of the conditioned signal.

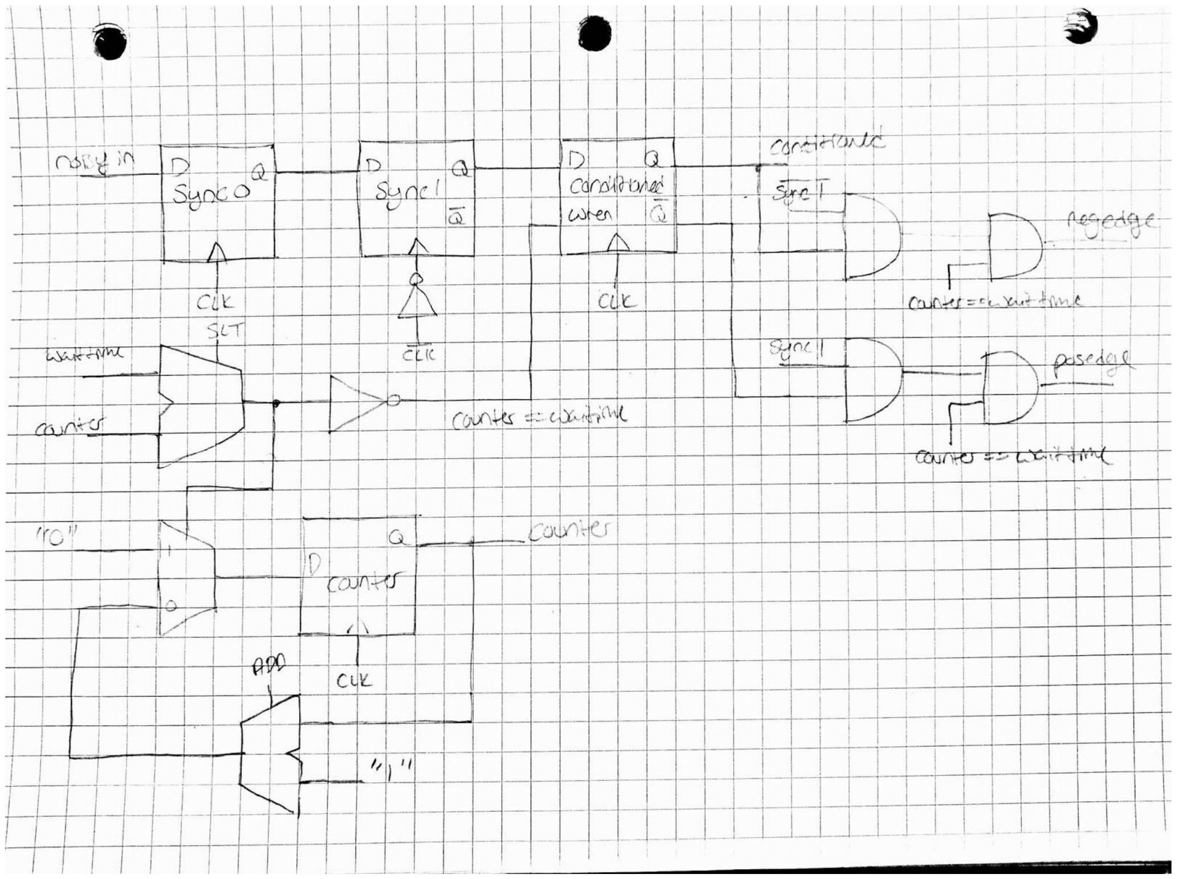
The test bench had three requirements - it had to demonstrate the three input conditioner functions - synchronization, debouncing, and edge detection. Synchronization could be tested by simply making sure that the inputted noisy signal was in phase with the internal clock domain (essentially everything was acting in accordance with the clock signal). Input debouncing could be tested by passing in a noisy signal and seeing if the conditioned signal was ‘cleaning up’ the noisy signal. Edge detection included making sure that the positive and negative edge pulses were actually occurring at the positive and negative edge of the conditioned signal.

Waveform



As can be seen from a waveform, the conditioned output is in phase with the clock signal. In addition, when a noisy signal is passed in, the conditioned output ‘cleans up’ the signal (output after a delay) and doesn’t include the noise. Also, the rising and falling edge pulses matched up with the rising and falling edge of the conditioned output.

Circuit diagram



We have three main blocks to our block diagram. One of our blocks is the main input conditioner circuit. The noisy signal is propagated through three D-flip flops, with the final conditioned flip-flop having some write enable logic. Note that *synchronizer1* is clocked when the clock signal is low. To find the negative edge and the positive edge, *synchronizer1* and *conditioned* are compared to each other and only set high if *synchronizer1* and *conditioned* are different in the proper order. The second block involves our write enable logic for the conditioned D flip flop. It is only activated when counter is equal to waittime, which we implement (in our circuit design) using the SLT operation of an ALU. The counter logic itself is the 'final' block, using a register to store count, an ALU to increment count by 0 or 1, and a mux to determine if we want to reset counter or if we want to keep incrementing.

Waittime Analysis

If the main system clock is running at 50 MHz, the maximum length input glitch that will supressed by this design for a *waittime* of 10 is 260 ns. For a clock running at 50 MHz, each clock cycle is going to be 20 ns. In the input conditioner design, it will take two clock cycles for *synchronizer1* to modify its value and then be compared to *conditioned*. The counter counts for *waittime* clock cycles, which in this case is 10, in order to reach the next part of the design. It then takes one clock cycle for *synchronizer1* to be written to *conditioned*. *Conditioned* is the final output to our system. The total number of clock cycles is 13 clock cycles (for a *waittime* of 10). 13 clock cycles would take 260 ns. Thus, the maximum length input glitch able to be supressed with a waittime of 10 us 260 ns.

Shift Register

Test Bench Strategy For the Shift Register:

The test bench had to demonstrate the four functions of the shift register (SR):
1) The shift register advances one position on a peripheral clock edge.
   - serialDataIn loaded into the LSB.
   - The rest of the bits shift up by one position.
2) When parallelLoad is asserted, the shift register will take the value of parallelDataIn.
   - Data is not loaded into the shift register from the serialDataInPort if parallelLoad is true.
3) serialDataOut will always present the MSB of the shift register.
4) parallelDataOut always presents the entirety of the contents of the shift register

We decided to use 8 test cases to test the various functions:
Test Case 0: Load data into the SR and check to make sure that the data in the shiftregister memory (a register in the shiftregister module) matched the output port parallelDataOut. If it does, this confirms that function 4 is true.

Test Case 1: Successfully serially load data into the SR. We loaded the SR by presenting alternating data bits on the peripheral clock edges to the serialDataInPin. We then ensured that the sequence we loaded matched the values at the parallelDataOut port. This confirms function 4.

Test Case 2: Successful bit shifting. In Test Case 1, we serially loaded bits into an empty shift register. In this test case, we loaded one additional bit serially to check if the bits from the load performed for Test Case 1 were shifted over and that the least significant bit took on the value of the new bit loaded in. This further confirms function 1.
Test Case 3: Successful parallel load into SR. We enabled parallelLoad and presented data at the ParallelDataIn. We then confirmed that the data presented matched the data at parallelDataOut. This confirms function 2.

Test Case 4: Serial loading is blocked when parallelLoad is enabled. We attempted to serially load data into the SR while parallelLoad was high. If the shift register memory doesn't change from this loading, we have successfully blocked serial loading. This test confirms the subcondition of function 2.

Test Case 5: SerialDataOut takes the value of the most significant bit of the shift register memory. We parallel loaded a known bit sequence into the SR and then ensured that the MSB of the shift register memory matched the value of serialDataOut. This confirms function 3.

Test Case 6: Only parallel load when parallelLoad is high. We presented data at the parallelDataIn port while parallelLoad was low and checked to make sure that this data was not stored in the SR memory. This further confirms function 2.

Midpoint Check In

We were able to create a top-level module according to the structure given to us in the Lab 2 guidelines and load it onto the FPGA. We then demonstrated our successful operation. The following was the test sequence we designed and executed to demonstrate our working top-level module:

PIPO:

1. Press button 0 (parallelLoad) to load parallelDataIn (specified as 0xA5, or b10100101).

2. Press button 1 to display the least significant bits. An LED ‘on’ state represents when a bit is 1 (and when it is off, it represents when a bit is 0). LEDs 0 and 2 should be on and LEDs 1 and 3 should be off in order to represent least significant bits ‘0101’.

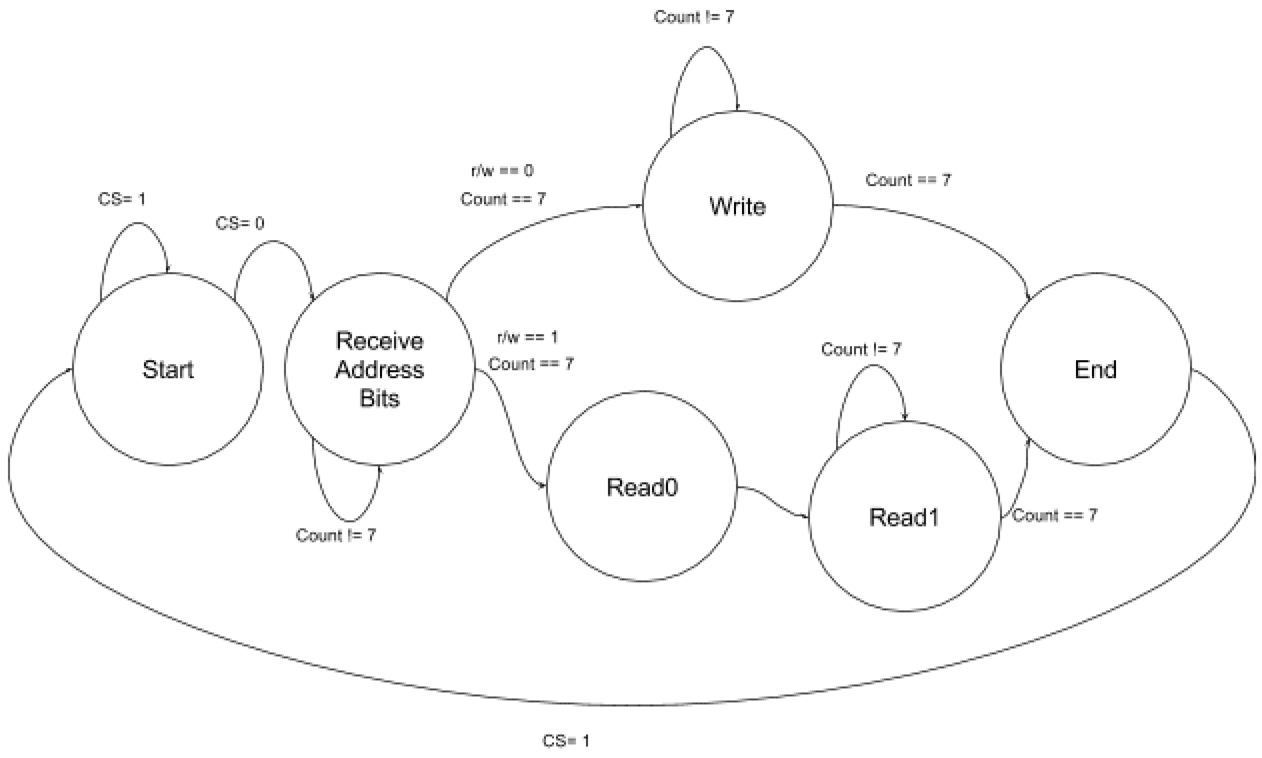
3. Press button 2 to display the most significant bits. LEDs 0 and 2 should be off and LEDs 1 and 3 should be on in order to represent the most significant bits ‘1010’.

SIPO

1. Place switch 0 in the “low” position. This represents a 0 value bit. When switch 2 is toggled from low to high, the 0 bit will be serially loaded, and the bits in the shift register will shift up one degree from their previous places.
2. Press button 1. LEDs 0 and 2 will now be off, and LEDs 1 and 3 will be on.
3. Press button 2. LEDs 0, 1, and 3 will be off, and LED 2 will be on.
4. Place switch 0 in the high position, which represents a 1 value bit. Toggle switch 1 from low to high to serially load a 1 into the shift register.
5. Press button 1. LEDs 0 and 2 will now be on, and LEDs 1 and 3 will be off.
6. Press button 2. LEDs 1 and 2 will be off, and LEDs 0 and 4 will be on.

Finite State Machine

Diagram



Tables

State Logic

		signal	
condition	**state**	**count**	**CS**
CS = 0	recieve_addr_bits	(for receive) 0	0
CS = 1	start	X	1
Receive_count = 7 AND A[0] = 0	write	(for receive) 7	0
Receive_count = 7 AND A[0] = 1	read0	(for receive) 7	0
(previous state is	read1	(for receive) 0	0

read0)			
Receive_count != 7	receive_addr_bits	(for receive) count += 1	0
Write_count = 7	end	(for write) 7	1
Write_count != 7	write	(for write) count += 1	0
Read_count = 7	end	(for read) 7	1
Read_count != 7	write	(for read) 7	0

Output Logic

	Outputs			
State	**addr_we**	**dm_we**	**miso_buff**	**sr_we**
start	0	0	0	0
receive_addr_bits	1	0	0	0
write	0	1	0	0
read0	0	0	0	1
read1	0	0	1	0
end	0	0	0	0

Implementation

To implement the SPI memory in accordance to the recommended structure provided in the Lab 2 guidelines, we first had to design and implement a finite state machine. This finite state machine would allow the SPI memory to perform the right operation at the when necessary.

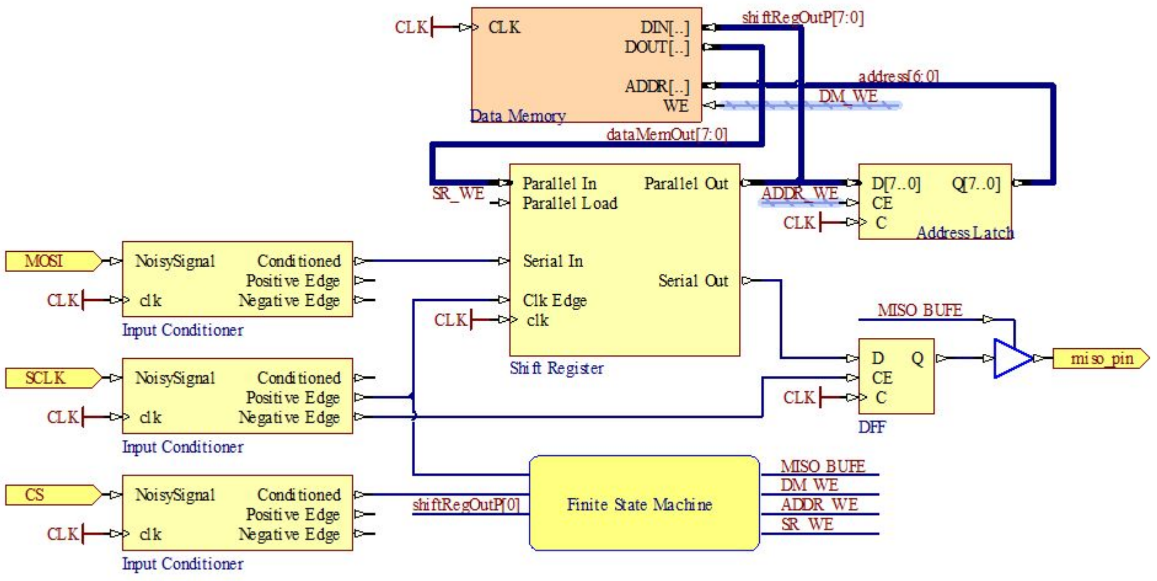
We started off with an FSM diagram for the state logic (which we then also put into a table). First, we had the start state. The system would stay in this state when chip select (*CS*) was asserted, but would move onto the receive_addr_bits state when chip select was 0. In this state, a counter was needed (in order to receive the address bits, as the name implied). When count was less than 7, it would continue receiving. When the count was 7, then (not reflected in the FSM diagram) that bit would be read - this bit is the R/W bit. When count is 7 and the R/W bit is 0, then the system goes into the write state. This write state receives more bits (the bits to be written to the address bits specified), and so a counter is used again. When count is 7, the system goes into the end state. If CS goes high, then the system goes back into the start state.

From receive_addr_bits, there is another state it could go to: when count is 7 and the R/W bit is 1, then the state becomes read0. The state then becomes read1. (This seems arbitrary but is actually necessary for the output logic.) Then, another counter occurs in order for the data memory bits to be read. When count is 7, the state goes to end.

The output logic for the system is fairly straightforward. In the start and end states, everything is deasserted. The reason that these states aren't the same is because we wanted to make sure that end only goes to the state state if CS is asserted. (This is so that the state logic doesn't occur infinitely.) In the receive_addr_bits state, *addr_we* is asserted - the write enable to the address latch, so that the input address bits can be written to the address latch. In the write state, everything else is deasserted except for *dm_we*, the write enable to data memory. This allows the inputted bits to actually be written to memory. In the read0 state, the *sr_we*, the *parallelLoad*, is asserted with everything else deasserted. This is so the data memory can output the bits stores in the address into the shift register, which will then become serial out. This needs to be only asserted for when data memory is outputting its bits and no more - otherwise the serial output will not output correctly. In the read1 state, *miso_buff*, essentially an enable into a buffer before the *miso_pin*, is asserted with everything else deasserted. This is so the value gets put into *miso_pin* only when reading.

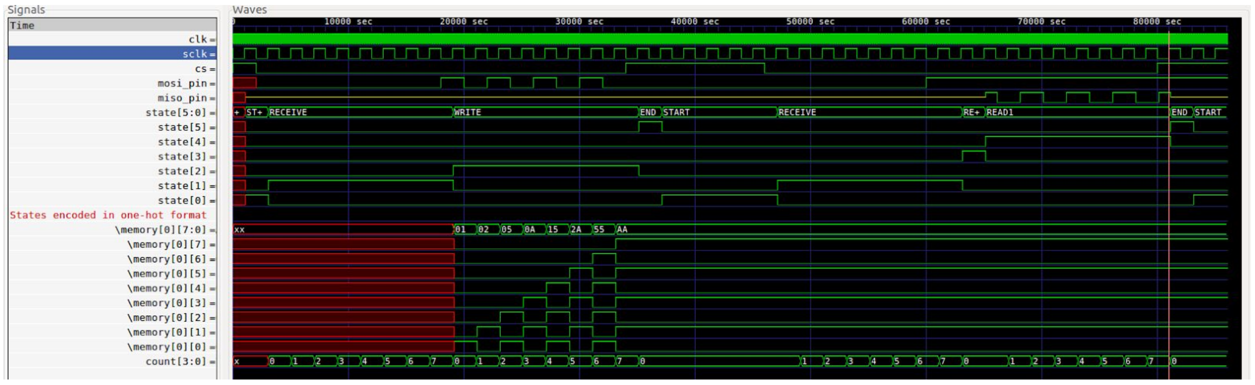
SPI Memory

Implementation



We chose to use the suggested implementation of the SPI memory, which is shown in the above block diagram. This implementation makes use of a shift register to load and push data to the output of the system and a data memory that stores 8 bit sequences at specified addresses. The SPI memory module was implemented in `spimemory.v`.

Waveform



Testing

There were four main things that we wanted to test for in our test bench:

1. State: The logic in the SPI Memory is implemented within the FSM, so it's essential that the FSM is working correctly in conjunction with the rest of the system. In order to test this, we decided to check whether the SPI was in the correct state whenever there was a new state entered. The FSM implementation is just state logic (making sure we toggle between states) and output logic (making sure that the right things are asserted) - however output logic is essentially checked by the rest of the rests and making sure they were working right. Thus, we didn't feel the need to check whether the actual outputs were correct and working (we also tested our FSM and were confident that it was asserting and deasserting the necessary signals at the appropriate states). However, the state logic was important to check because it allowed us to see if responses were happening at an acceptable delay, synced up correctly, and whether the signals we were writing (expecting it to do a certain action) was actually performing that action.
2. Address: It was necessary to test if we were writing to the expected address in our memory because it is easy to change one bit in the address by presenting data at the wrong clock edges. To do this, we presented the bits of a specific memory address and then checked at the address input of the data memory at the end of the receive bits state to ensure that the correct address would be written to.
3. Write: It is similarly important to the address testing to ensure that the expected bit sequence is being written to the data memory. Presenting data on the wrong clock cycle will result in the wrong data being written. To test this, we presented a specified sequence of bits and then checked the data memory at the address we wrote to confirm that the data was the same as what we were trying to write.
4. Read: It was also necessary to test whether the data bits being read were the data bits we expected - in this case, the data bits we had previously written to the same address

location. We in fact caught an error through this - it turned out we were originally overwriting whatever was stored in our data memory with more address bits (that we sent in to test out read). At each point in the *mosi_pin* output, then, we make sure to check whether the output of that pin matches what we expect - which in this case, was the corresponding bit of whatever we initially wrote. When this test passed, we then knew that we were actually reading from data memory, and doing so correctly.

We implemented our tests by running through a complete spi memory cycle (writing to an address and reading from an address) checking conditions throughout the run. We also extensively used gtkwave for diagnosing problems when our tests failed. When writing bits to the memory, we chose to use an alternating pattern of 1s and 0s in order to minimize wrongly passing tests due to phase offsets.

Work Plan

Here is a table showing our original work plan and the actual reality of what occurred.

Task	Expected Breakdown of Time	Actual Breakdown of Time
Input Conditioner	6 hours Finish by 10/24/17	3 hours Finished by 10/22/17
Shift Register	4.25 hours FInish by 10/24/17	2 hours Finished by 10/22/17
Midpoint Check-In	4 hours Finish by 10/24/17	2 hours Finished by 10/24/17
SPI Memory	3.5 hours Finish by 10/27/17	2.5 hours Finished by 10/29/17
SPI Memory Testing	3.5 hours Finish by 10/28/17	1 hour “Finished” (see debugging) by 10/29/17
Debugging	4+ hours Finish by 11/1/17	5 hours Finish by 11/1/17
Polishing Code	1.5 hours Finish by 11/1/17	0.5 hours Finish by 11/2/17
Final Report	0.25 hours Finish by 11/2/17	2 hours Finished by 11/2/17
Total Time	27.5 hours Finish by 11/2/17	18 hours Finished by 11/2/17

As one can see, the budgeted time that we had laid out in our work plan was overscoped, and we actually spent significantly less time overall on our SPI memory. The primary reason we overscoped is primarily because 1) we initially thought, as a two-person team, we may be at a disadvantage for a lab that specified ‘teams of about three people’, and 2) with our read-through of what was required, we had anticipated more difficulty getting things to work properly. In reality, most things took less time. One reason that a lot of tasks took less time than anticipated, other than the fact they were easier, was because we didn’t actually end up working on the lab report in conjunction, which increased the final lab report time but decreased the time per certain tasks. Note as well that SPI memory was completed after the date we set but under time (with a few bugs in the FSM and test bench, so that time carried over into debugging) due to the fact that overlapping times we could both meet limited our ability to complete that part according to the date we set.

Error Debugging

A large amount of time went into debugging. The two main issues we had were with the test bench and with the FSM. With the test bench, we had to be more thoughtful about how we were setting the pins and at what clock edge they were being set up. This ended up being fixed by simply making sure every input pin was defined when we wanted it to be (waveforms were very useful in this aspect). Secondly, in our FSM, the fact that the read state was asserting two different output signals caused issues with data being overwritten - s that had to be separated into two. We also had to be thoughtful about deasserting other signals to make sure that no other write enables were asserted.