Computer Architecture: Lab b002

Lauren Anfenson
Anusha Datar
Josh Deng

October 19, 2018

1 Our Architecture

1.1 Input Conditioning

Our input conditioner handles time synchronization, input signal debouncing, and generation of a positive and negative clock edge. Synchronization is handled with a series of D-Flip Flops that only passes a signal on the clock's rising edge. Our wait time debounces the input by forcing the signal to be high for some amount of time before it can be passed to the conditioned output – this is done with a counter the increments on the rising edge of the clock that has to reach some value before the final D-Flip Flop is enabled. To detect the positive and negative edges of the conditioned output, we store the value of the output from one clock cycle before the current output with a D-Flip Flop. We then AND the inverted stored signal with the current output to produce the rising clock edge, and AND the inverted current signal and the stored signal to produce the falling clock edge.

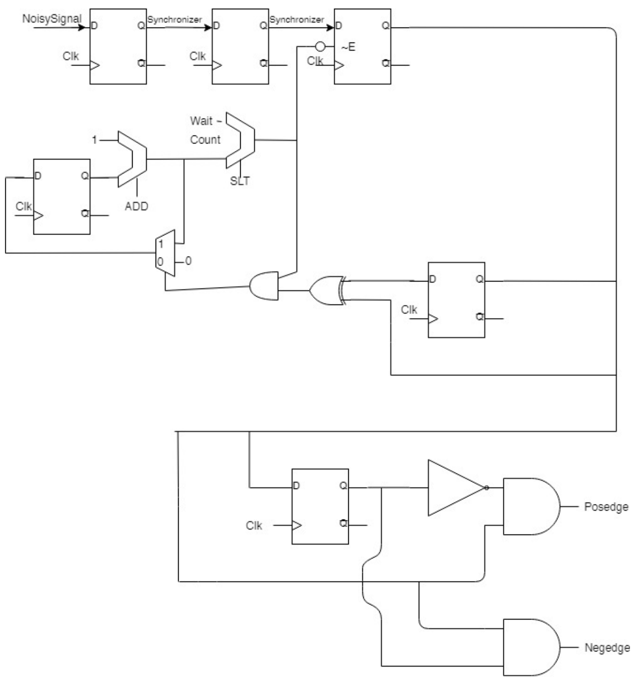

Figure 1: The schematic of the input conditioner circuit.

clk	cs	sclk	counter(0..15)	R/∼W	addr_en	dm_en	s_r	miso_en
not ↑	x	x	counter$_{last}$	x	x	x	x	x
↑	1	x	15	0	0	0	0	0
↑	0	↑	counter++(15)	x	0	0	0	0
↑	0	↑	counter++ (0..5)	x	1	0	0	0
↑	0	↑	counter++ (6..14)	0	0	1	0	0
↑	0	↑	counter++ (6..14)	1	0	0	1	1

We tested our input conditioner by inputting a few longer high signals to make sure edge detection was working. Then we added a few short, high frequency signals to make sure the debouncing behavior prevented them from reaching the conditioned output.

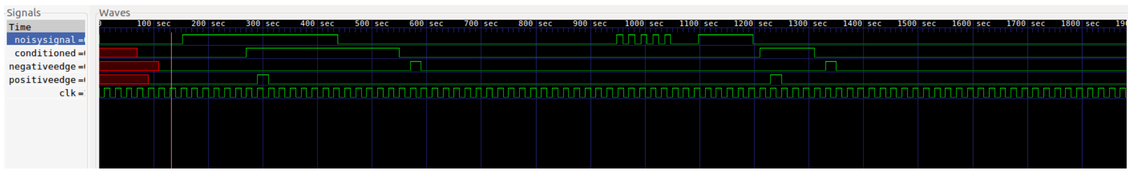

Figure 2: The waveforms associated with the input conditioner. We can see that high frequency glitches in the input are debounced and the noisy signal is turned into a conditioned signal that is synchronized with the clock.

If the main system clock is running at 50MHz, the maximum length input glitch the design could suppress when $waittime = 10$ will be $\frac{11}{50{,}000{,}000}$ seconds, or $220ns$. The reason that this is the case is because the actual counter will count ten clock cycles before enabling the final flip-flop, and then it will hold state for an additional clock cycle prior to pushing the final output.

1.2 Shift Register

With our shift register testing, we worked to test each function of the shift register: Parallel Out, Serial Out and Serial In, and Parallel Load. We then checked the waveforms to ensure that the system reacted as expected. When testing Serial In, we input a 1 and then watched as the shift register propagated this 1 through the entire Parallel Output. To test Parallel Load we loaded a value (0xAA) and then watched as Parallel Out changed to the correct value and Serial Out incremented through all 8 bits from MSB to LSB.

1.3 Finite State Machine

Our finite state machine sets the enable pins for our memory so the system can keep track of whether it's receiving an address, reading data, or writing data. It does this by counting the positive edges of the serial clock when chip select is low, setting address enable to HIGH for the first 7 bits, then either setting serial read enable to HIGH and enabling MISO if we're reading data from a register (8th bit is 1), or setting data memory enable high if we're writing data to a register (8th bit is 0). A diagram of these transitions is shown below.

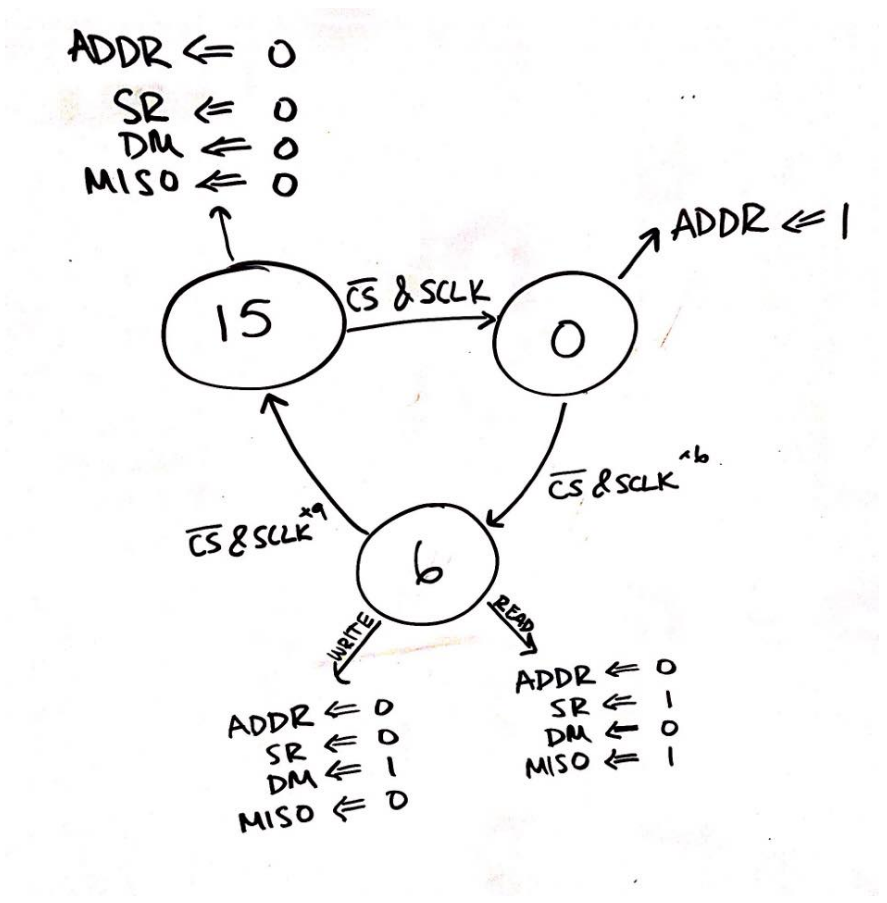

Figure 3: A whiteboard capture of our finite state machine.

1.4 SPI Memory

We tested each of our systems that we used to build up to the SPI memory. Some of the larger ones, such as the shift register and input conditioner, had associated testbench files. Otherwise, we simply tried a few cases as we integrated and examined the gtkwave output to confirm that the system behaved as anticipated. To test our SPI memory, we developed a generalized testing framework : we chose an address, transmitted the appropriate address bits, made sure that the data memory had the matching address value, wrote some data to that area, and then we confirmed that it was written. Afterwards, we read from the same address and ensured that the reading matched what we had written to that address prior. Once we had this framework, which effectively tested each piece of our system, we tried a variety of standard and edge cases for both the address and data bits : we ensured that the module was not sensitive to endianness, failed when reading from the wrong address or reading for the wrong data, could handle unique sequences such as a full string of 0's and 1's, etc. By testing edge cases and more standard inputs, we could be confident about the functionality and robustness of our module without exhaustively testing it.

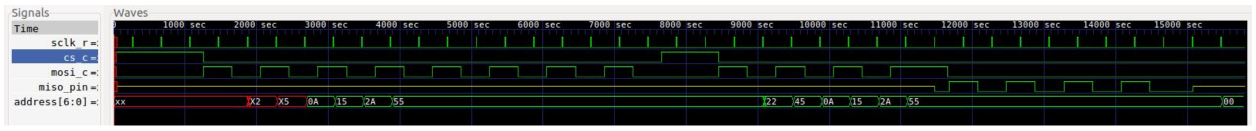

Figure 4: A waveform capture of first a WRITE to SPI Memory address 55_{16} and then a READ from SPI Memory address 55_{16}. Data read and written is AA_{16}.

2 Reflection

2.1 Work Plan

Our work plan time estimates ended up being fairly inaccurate when it came to the integration and testing of our SPI memory. We ran into some unexpected bugs and spent several hours testing and debugging that we did not account for when writing the work plan. So, our implementation and testing were finished on Thursday night rather than Wednesday night like we had hoped. Because of this, we didn't end up spending the 1.5 hours on exploring testing with an external device that we had planned to.