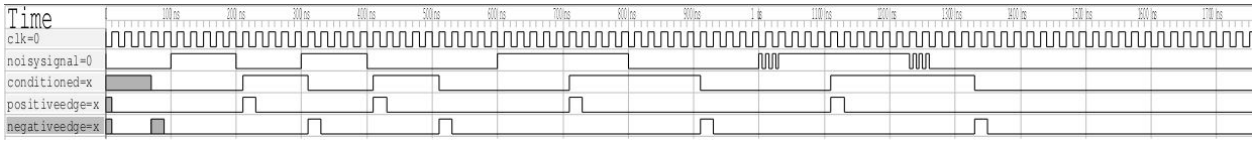
Comp Arch Lab 2
Nick, Noah, Yichen
10/17/2018

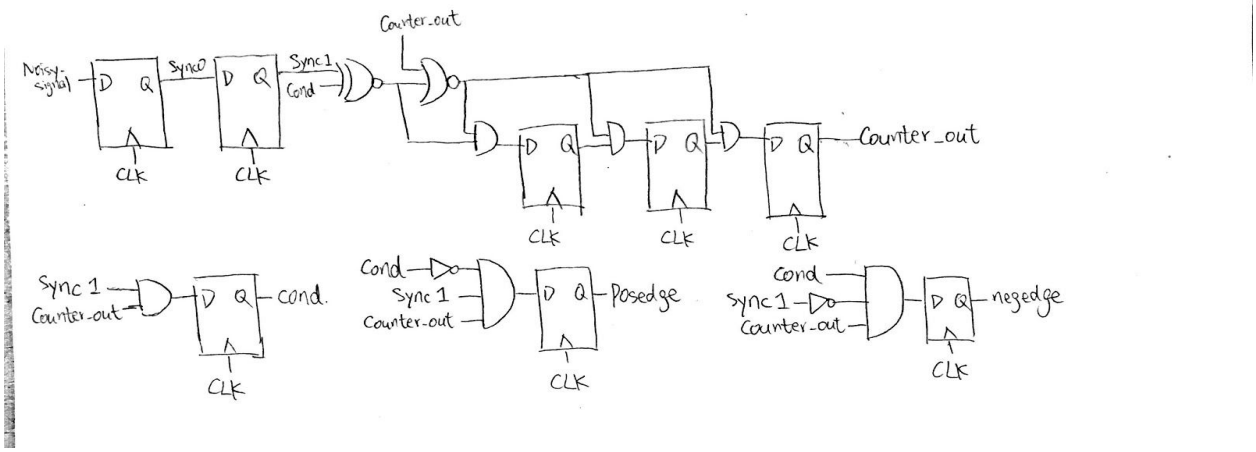
Section 1. Input Conditioner

The input conditioner has three major purposes, namely input synchronization, edge detection and input debouncing. We ran test cases that involved those cases in our test bench and produced the following waveform.



The first part of the waveform demonstrates the input synchronization and edge detection functions of the input conditioner, and the second half demonstrates input debouncing. According to the result, the input conditioner is functioning as expected.

The structural circuit of an input conditioner is shown below:



Because the system clock runs at 50 MHz, each clock cycle takes 0.02 ns to complete. With the input conditioner of waittime=10, the maximum length suppressible input glitch can last for 0.2ns, and any input glitch longer than this will occur on the conditioned output line.

Section 2. Shift Register

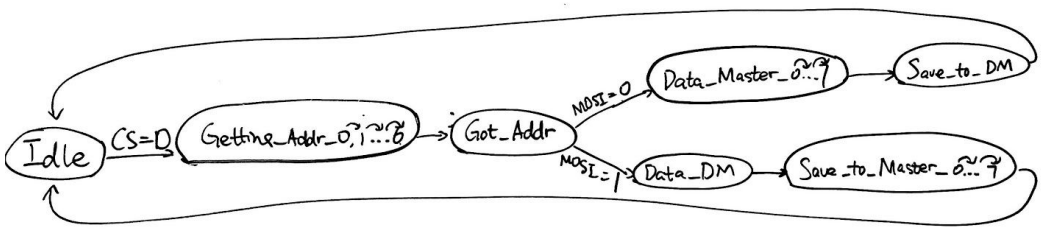
The shift register has two modes, PISO (parallel in, serial out) and SIPO (serial in, parallel out). If the parallel load input is high it is in PISO mode. It will set the contents of the shift register to an 8-bit input, and output the MSB on the serial port.

In SIPO mode it will output the content of the entire shift register. It accepts serial input on positive edge detection. When the input is accepted the contents of the register will be shifted by one, and the serial input will be moved into the LSB position.

Our test bench for the shift register tests both SIPO and PISO modes. Our tests are designed to isolate particular operations on the shift register. For example, the first test only checks the parallel read operation. We then test the other operations one at a time.

Section 3. Finite State Machine

The diagram and control table of our finite state machine are shown below. The 6 GETTING_ADDR states, 7 DATA_MASTER states and SAVE_TO_MASTER states are expanded in the actual implementation, but are concatenated together in the diagram for viewability.



States	MISO_BUFF	DM_WE	ADDR_WE	SR_WE
IDLE	0	0	0	0
GETTING_ADDR_0..6	0	0	0	0
GOT_ADDR	0	0	1	0
DATA_MASTER_0..7	0	0	0	0
SAVE_TO_DM	0	1	0	0
DATA_DM	0	0	0	1
SAVE_TO_MASTER_0..7	1	0	0	0

Section 4. SPI Memory

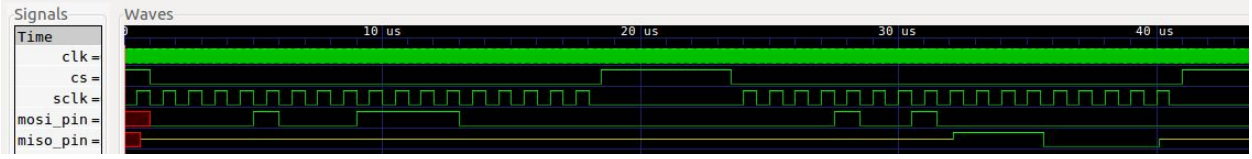
Our test sequence for the SPI memory is as follows:

- Using the appropriate inputs, we first determine the first register being checked is at address 0000100.
- We will first write the value 11110000 to this address. During this, we will check the FSM state and ensure that we are writing to the right register.

- We then read the value at the address. At this point, we are ensuring the correct write/read pattern.
- Throughout this process, there are test statements that will throw the flag “dutpassed” to 0 if any test is failed.
- This process is then repeated for the second register 0001100, with the used value being 00001111. This will ensure that we are only writing to one register.

By following through with this strategy, we are able to test every state for multiple registers in the SPI memory. We decided to pseudo-randomly select two registers to test, as exhaustively testing every register should not be necessary as all registers rely on the same code. We also used this as a way to reconfirm that the other parts of our code base were working (the full SPI memory testing caught a few issues with the FSM, which were quickly resolved).

The first test can be seen below. The MISO pin is giving out the correct output from the data memory as expected, so this confirms that our SPI memory implementation is correct.
In order to run the SPI memory tests, run make in the directory our lab is saved in.



Section 5. Reflection/Analysis of Work Plan

Our work plan accurately described the first part of the project, up until the midpoint deliverable. After that however, reality quickly diverged from the work plan. The individual work still generally only took as long as expected, but integrating it was more time consuming than anticipated. During integration a number of difficulties occurred. These difficulties were in part due to small timing errors. An additional problem arose as a result of our using the example circuit diagram provided in the github repository. In the circuit diagram the input to the FSM was mislabeled, leading to a timing discrepency where the read/write bit was delayed by an additional clock cycle.. Changing the input from the shift register output to the conditioned input resolved the difficulty. Once the integration problems were resolved we returned to our original work plan, and completed the remaining tasks in the anticipated time.