Computer Architecture Lab 02 : SPI Memory

Yoonyoung Cho, Haozheng Du, Shruti Iyer

October 13 2016

1 Introduction

In this Lab, we implemented SPI (Serial Peripheral Interface) Memory in behavioral Verilog, which incorporated components of, largely, *Input Conditioner*, *Shift Register*, and *Finite State Machine*. After conducting the initial tests in Verilog and verifying that the module was fully functional, we implemented our design on the ZYBO FPGA board, where the internal ARM processor acted at the *master* and the FPGA acted as the *slave*.

2 Input Conditioner

The input conditioner was a critical component of our system, as it synchronized the input with the clock, removed the noise by debouncing[1], and detected the positive and negative edges of the signal.

2.1 Schematic

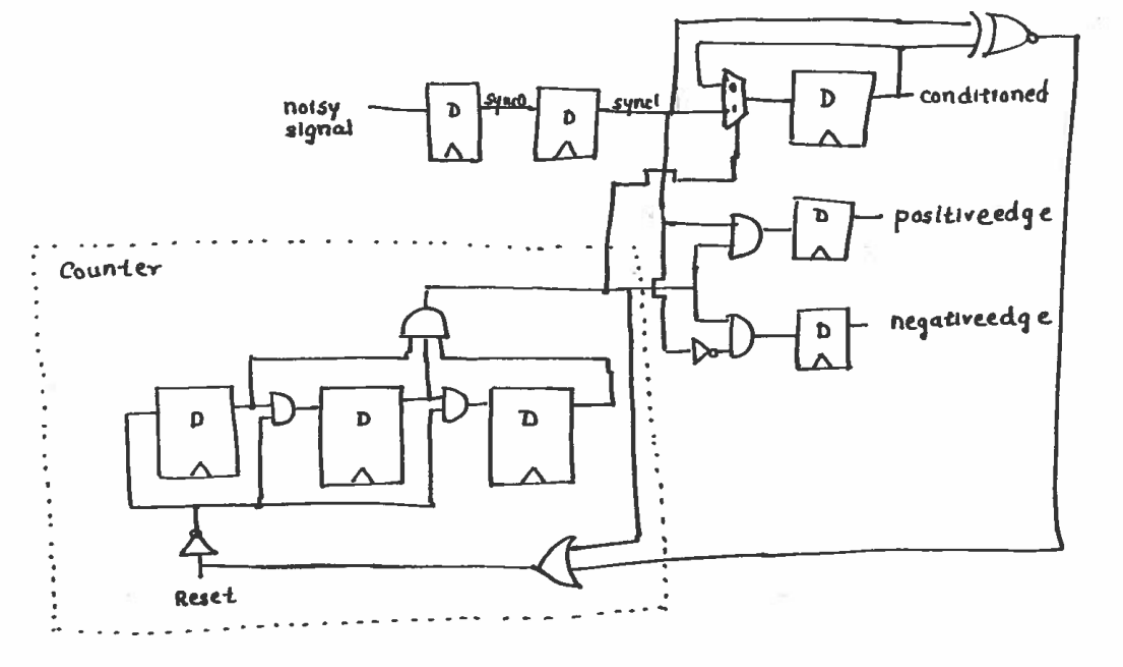

Figure 1: Structural circuit of the input conditioner

[1] in this implementation, we debounced the circuit simply by examining the consistency of input over a fixed time-frame.

2.2 Waveform

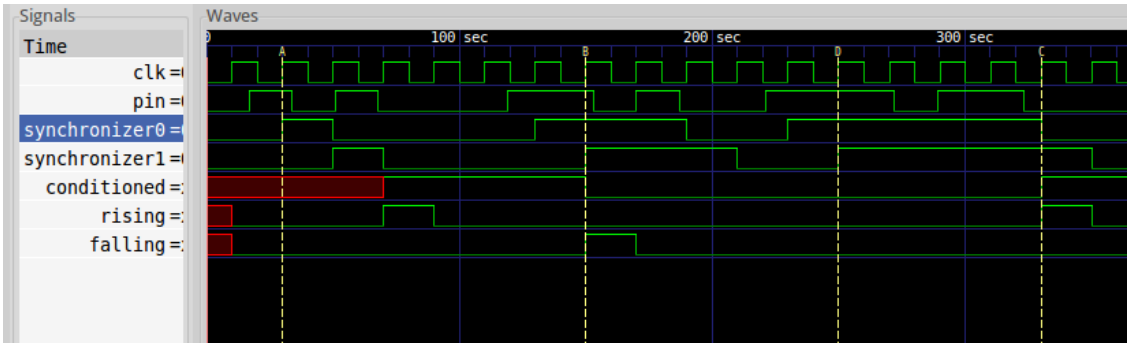

Figure 2: Waveforms generated by the input conditioner

Marker A : SYNCHRONIZATION Marker B : FALLING Marker C : RISING Marker D : DEBOUNCING

At Marker A, the pin value is high and the counter (not shown) equals 1. At the third clk pulse from Marker A, when the counter equals 3, the conditioned signal goes high.

At Marker B, the falling signal goes high because the conditioned signal goes low. At Marker C, the rising signal goes high because the conditioned signal goes high.

Between Marker D and C (at around Time = 260 sec), there is a glitch in the pin value where it goes low momentarily for time less than 3 clk cycles. The corresponding conditioned outputs signal without a glitch.

2.3 Analysis

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?

$$\begin{aligned} f &= 50MHz \\ t &= 1/f \\ &= 20ns \\ 10t &= 200ns \end{aligned}$$

If the glitch is shorter than 200 nanoseconds, it will be suppressed by the input conditioner[2].

3 Shift Register

As our module communicated with the master with two pins (MISO and MOSI pins), the shift register played a critical role in the implementation, as it allowed for serialized communication between the two units.

In order to test the shift register, we generated a set of random bits as address and data input to the shift register; we first verified that the register was shifted at each positive edge of the peripheral clock, and then also tested that the whole data bus was loaded when *parallelLoad* flag was high.

[2]To be more precise, the glitch -shorter than 180 nanoseconds is guaranteed to be suppressed, irrespective of its initiation; the longest glitch that may be suppressed is just shy of 200 nanoseconds.

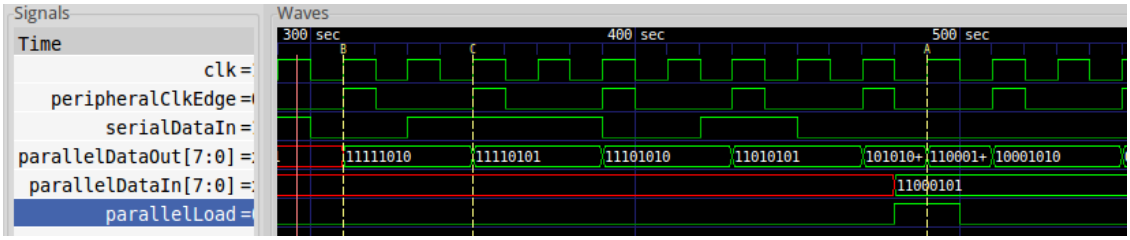

Figure 3: Waveforms generated by the shift register; the dotted yellow lines correspond to the markers B,C,A from the left.

Seen above are the three markers that demonstrate each of the functions of the shift register; marker B shows the time at which the LSB[3] of the shift register was set to be 0, the remaining bits shifted by one towards the left. Marker C indicates its complement, where the LSB of the shift register was set to be 1. Finally, Marker A demonstrates loading the data input from the parallel bus and setting the value to the internal memory of the shift register.

4 Midpoint

As there are only four LEDs available on the board, we use switch[2] to decide which four bits to display. If switch[2] is HIGH, the lowest four bits will be displayed else the highest four bits will be displayed. Check out our test here: `https://www.youtube.com/watch?v=Vn69aocfGD0`

4.1 Parallel In

For the purposes of mid-point check-in, we used a hardcoded string "xA5"(b10100101) for parallel load. On pressing buttton[0] on the ZYBO loads the hardcoded hex string. If switch[2] is HIGH, the LEDs displays "0101", else it displays "1010".

4.2 Serial In

Value of switch[0] is the bit that gets loaded for Serial In. On the rising edge of switch[1], the value of switch[0] is written to the lowest bit of the shift register. If you want to write "1" to the lowest bit, set switch[1] (clock) to LOW and switch[0] (value that is going to be written) to HIGH. On setting switch[1] to HIGH, the value "1" will be written to the lowest bit of the shift register. Before writing the next bit, you have to set switch[1] back to LOW.

5 Serial Peripheral Interface

5.1 Test Bench in Verilog

Prior to implementing our design on the ZYBO board, we sought to validate our design by writing a test bench in Verilog – doing so exposed a myriad of faults in our design (including clocking the transition on the serial clock, rather than the fast internal clock), and allowed us to test different scenarios without having to undergo the overhead incurred by implementing it in Vivado[4]

[3] least significant bit

[4] the process can be excruciatingly slow, especially since the software is run within the virtual machine.

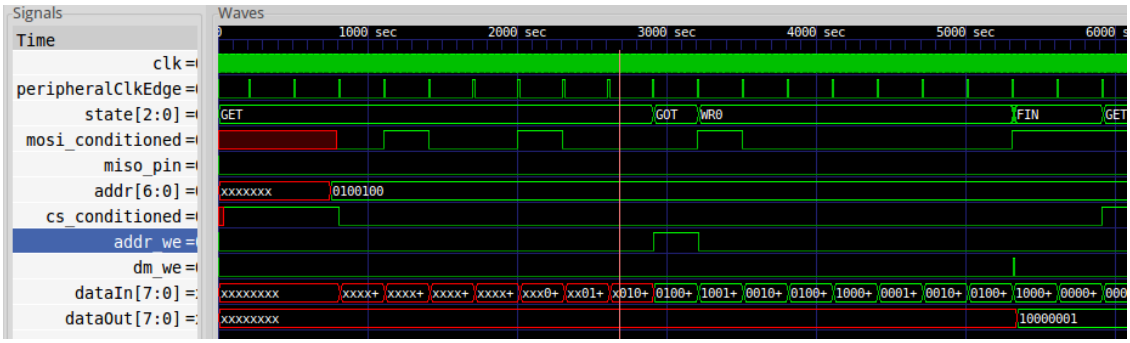

Figure 4: Waveform for writing to SPI

Initially, the cs_pin is set to HIGH for an indefinite amount of time, in order to simulate an idle state. Then, in order to write to SPI memory, cs gets de-asserted. While the state is GET, data at the mosi_conditioned pin clocks in the address to which the data needs to be written to. After receiving all 8 bits from mosi_conditioned, the state changes to WRITE0. The actual data (10000001) that needs to be written gets passed in the same signal, but this happens quickly according to the internal clock, so the transition occurs quickly to FIN: i.e. after getting all the 8 bits of the actual data, the state changes to WRITE1 and then FIN[5].

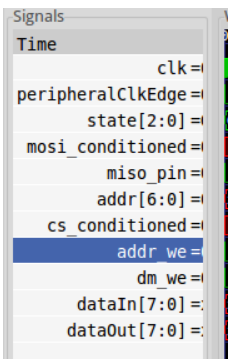

Figure 5: Signals

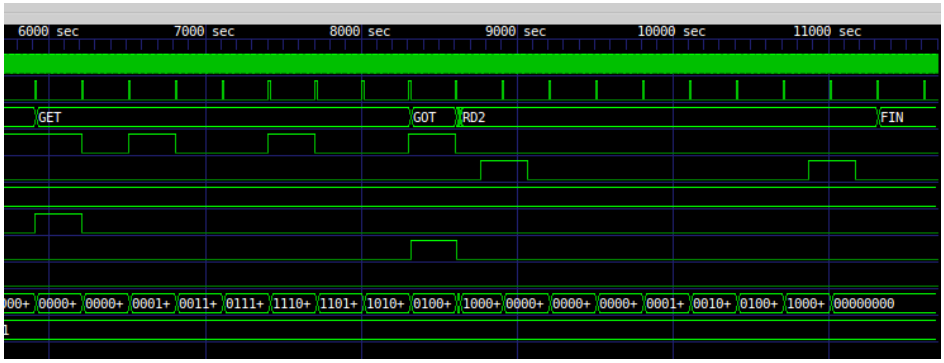

Figure 6: Waveform for reading from SPI

In order to read, the cs gets asserted and, in a similar fashion, loads the address unto the address latch. Thence, the state of the FSM transitions from GET to GOT and dataIn gets read. Because rw == 1, the state changes to READ0 → READ1 → READ2. (The waveform output doesn't show states READ0 and READ1 because they are not dependent on the serial clock and, consequently, immediately transfer to the next). The data at 0100100 read (10000001) can be seen in miso_pin.

5.2 Testing on the ZYBO Board

We have undergone numerous attempts to implement our design on the ZYBO board, each time in vain; the result from spi_read() always returned 0, regardless of the value written to the datamemory. In fact, even when miso_pin was configured to always output 1, spi_read() reported the value to be 0. Due to these inconsistencies, we suspected a disconnect in the communication link between the internal ARM processor and the ZYBO board.

As such, we were incapable of conducting thorough tests on the ZYBO board; as we were instructed – after an attendance during one of the NINJA hours and extensive testing –[6], below is a hypothetical test

[5]On the waveform, the output does not show state WRITE1 because it is not dependent on the counter and immediately changes to the next.

[6]There *was* an occasion in which the debugging procedure completed without failure, but we're currently suspecting that the breakpoint was not set in that run.

sequence that should demonstrate the validity of our module.

```
void spi_write(XSpi *SpiInstancePtr, u8 addr, u8 value){
  u8 addr_byte = (addr << 1); //Shifted, because we have 7 bits of addr, 1 bit of
      R/W
  //addr_byte |= 0x1; // not necessary, set to 0 anyways
  u8 RcvBuf[2];
  u8 SendBuf[2];
  SendBuf[0] = addr_byte; // send address + flag at first byte
  SendBuf[1] = value; // send value at second byte
  XSpi_Transfer(SpiInstancePtr, SendBuf, RcvBuf, 2);
  return;
}
```

Listing 1: Implementation of spi_write()

We have provided spi_write() above, in order to demonstrate that the module is functional;

```
int test_module(XSpi *SpiInstancePtr){
  // exhaustive test
  unsigned int addr, value;
  for(addr=0; addr < (1 << 7); ++addr){
      for(value=0; value < ( 1 << 8 ); ++value){
            spi_write(SpiInstancePtr, addr, value);
            u8 res = spi_read(SpiInstancePtr, addr);
            if(res != value){
              // breakpoint here
              return -1; // indicate failure
            }
        }
    }
  return 0; // indicate success
}
```

Listing 2: Hypothetical test sequence

Successfully completing an exhaustive test through all 7 bits of the address and 8 bits of value should be indicative of a thorough implementation of the module.

6 Work Plan Reflection

While making our workplan, the group decided to err on the side of overestimation of the work so that we are prepared to put in more work if we hit a roadblock. In our work plan, we estimated the number of the group spends working together to be around 15 hrs and 30 mins.

For Input Conditioner, we spent 3 hrs 30 mins on understanding the purpose, editing the given module and writing the testbench. The time spent was 2 hrs 30 mins less than estimated. For the Midpoint checkin, we spent 3 hours total on the shiftregister, midpoint and FPGA demo. The time mostly matched our estimation. We also spent 6 hrs, just like we estimated to write, test and debug the SPI memory.

We hit a roadblock while trying to load our SPI onto the FPGA; the initial expectation of 6 hours was proven to be very naive. Designing the module itself was not quite as difficult, nor close to as frustrating as were the efforts to guess-and-check the particularities of the ZYBO board: we had three meetings solely on the subject of debugging the elusive cause of the read result always returning zero, a problem that was seen in a majority of the other teams as well. The root cause of this still remains unclear, but ultimately the efforts spent[7] simply trying to work out the particular module exceeded the benefits it might provide towards our learning experience.

[7]More than 24 man-hours of sheer confusion, with more than nine attempts to load the project unto the board.