# ECE 272 Lab 5
Fall 2018

# Serial Peripheral Interface (SPI)

Phi Luu

November 14th, 2018
Grading TA: Edgar Perez
Lab Partner: Benjamin Geyer

# 1 Introduction

*Serial Peripheral Interface* (*SPI*) is a method of transmitting multiple bits of data using minimal hardware. When two pieces of hardware communicates with each other via SPI, they need to share a clock. At every rising edge of the clock, the data is transmitted from one device to the other.

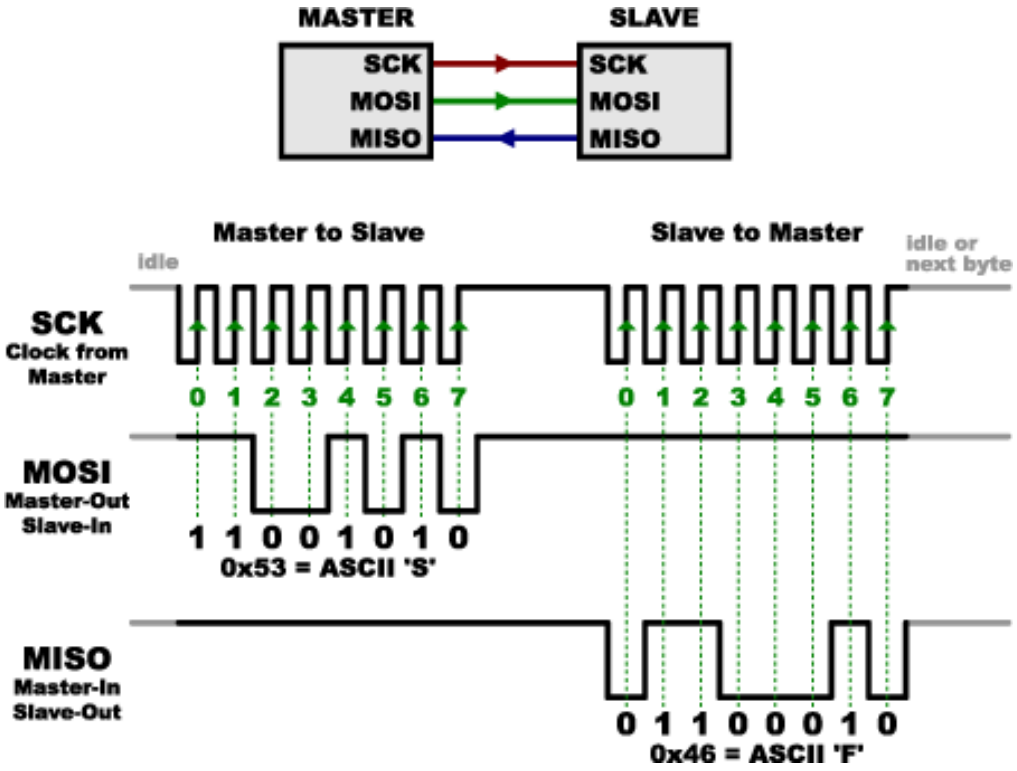


Figure 1: The waveform diagram of the master and the slave in SPI communication protocol [1]

In Figure 1, at every rising edge of *SCK* (clock from master), the data *MOSI* (master-out slave-in) is transmitted from the master device to the slave device. After 8 rising edges, an 8-bit data will have been transmitted—in this case, 11001010 from master to slave.

When the data goes over the designated number of bits, in this case 8 bits, the SPI will always "forget" the left most bit and transmit the latest chunk of bits.

In this lab, I and Ben designed a SPI protocol using the DE10-Lite FPGA. We used a button as a clock signal, a switch as the data line, and six seven-segment displays as the indicator of the data transmission from the SPI.

# 2 Design

The system takes in two inputs: one for the clock signal, controlled by a push button, and one for the data, controlled by a switch. This 2-bit input then goes to the SPI. The SPI protocol always outputs an 8-bit binary into a parser to pass into a seven-segment display decoder. Finally, the decoder outputs to the LEDs, indicating the current data being transmitted by the SPI protocol, as illustrated by the block diagram in Figure 2 below.

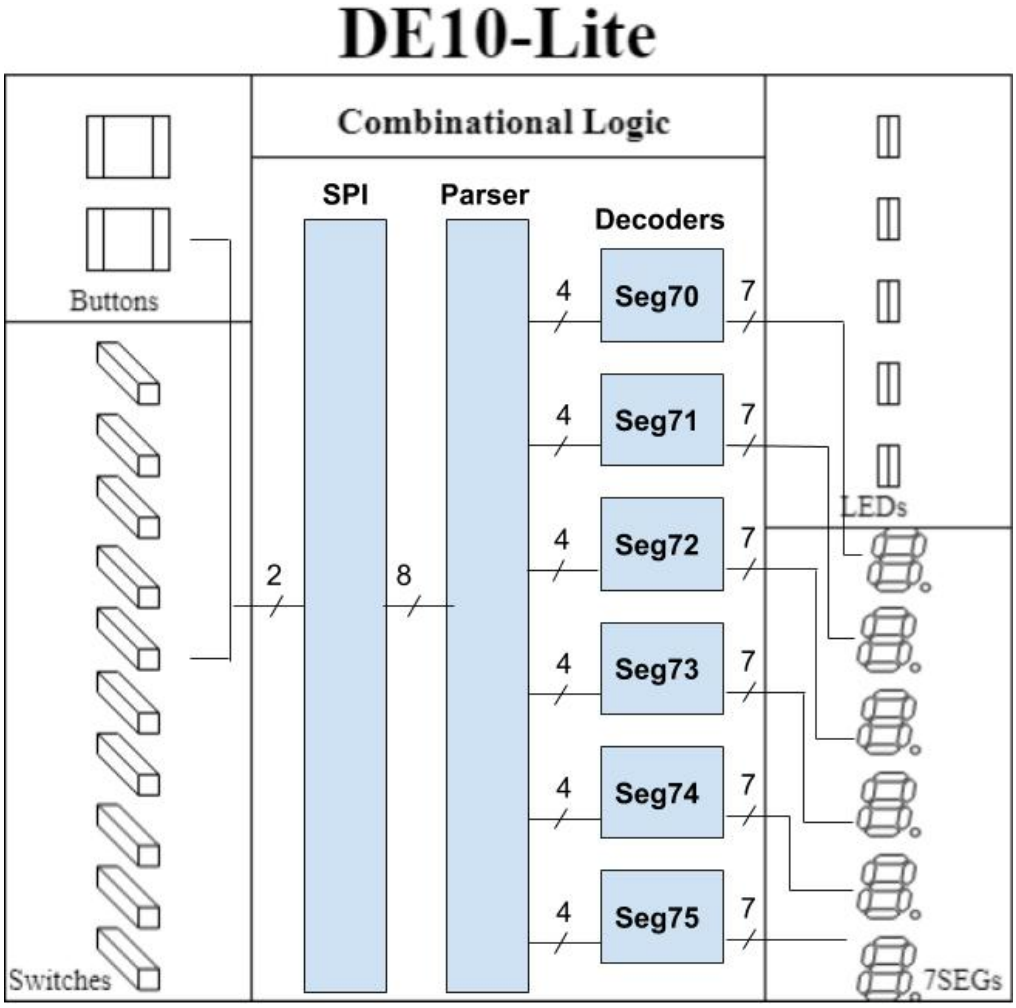


Figure 2: Block diagram

The SPI module takes in the clock signal and the data. It outputs an 8-bit to the parser, and every time the clock rises, it shifts the current value of the output 1 bit to the left and then add the input bit (data) to the right.

$$output = (output << 1) + data \tag{1}$$

For example, let the initial value of the output is 00000000, if the data is 1 when the clock rises, the output then becomes

$$output = (00000000 << 1) + 1 = 00000001 \tag{2}$$

Suppose for the next 7 times the clock rises, the data is 1100110, then $output = 11100110$.

If the clock rises again, *output* will be overflow. In this case, the SPI will "forget" the oldest bit (the most significant bit), the first 1. Suppose when the clock rises again, the data is a 1. Then, $output = 11001101$.

The SPI module has the same design as explained above. We reuse the parser and the decoder modules, thus there is nothing new in these modules. However, for the sake of clarity, I will explain

the functionality of the parser and the decoder below.

The parser takes in an 8-bit binary number, and SystemVerilog automatically converts numbers between binary, hex, and decimal systems. Since the SPI result is always an 8-bit number, its maximum value is a 3-digit decimal 255. Therefore, we only need the first three seven-segment displays. The parser outputs three 4-bit numbers, which are then taken in as inputs for the decoders. The decoder module uses the values in Table 1 to convert these numeric inputs to LED signals.

See the Appendix for the code.

| **Decimal** | **Binary** | **Seg$_A$** | **Seg$_B$** | **Seg$_C$** | **Seg$_D$** | **Seg$_E$** | **Seg$_F$** | **Seg$_G$** |
|---|---|---|---|---|---|---|---|---|
| 0 | 0000 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0001 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |
| 2 | 0010 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 3 | 0011 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| 4 | 0100 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| 5 | 0101 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0110 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 7 | 0111 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 8 | 1000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 9 | 1001 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

Table 1: Conversion table between decimal, 4-bit binary, and seven-segment signals

# 3 Results

We compile and simulate the project on ModelSim and obtain the following waveforms:

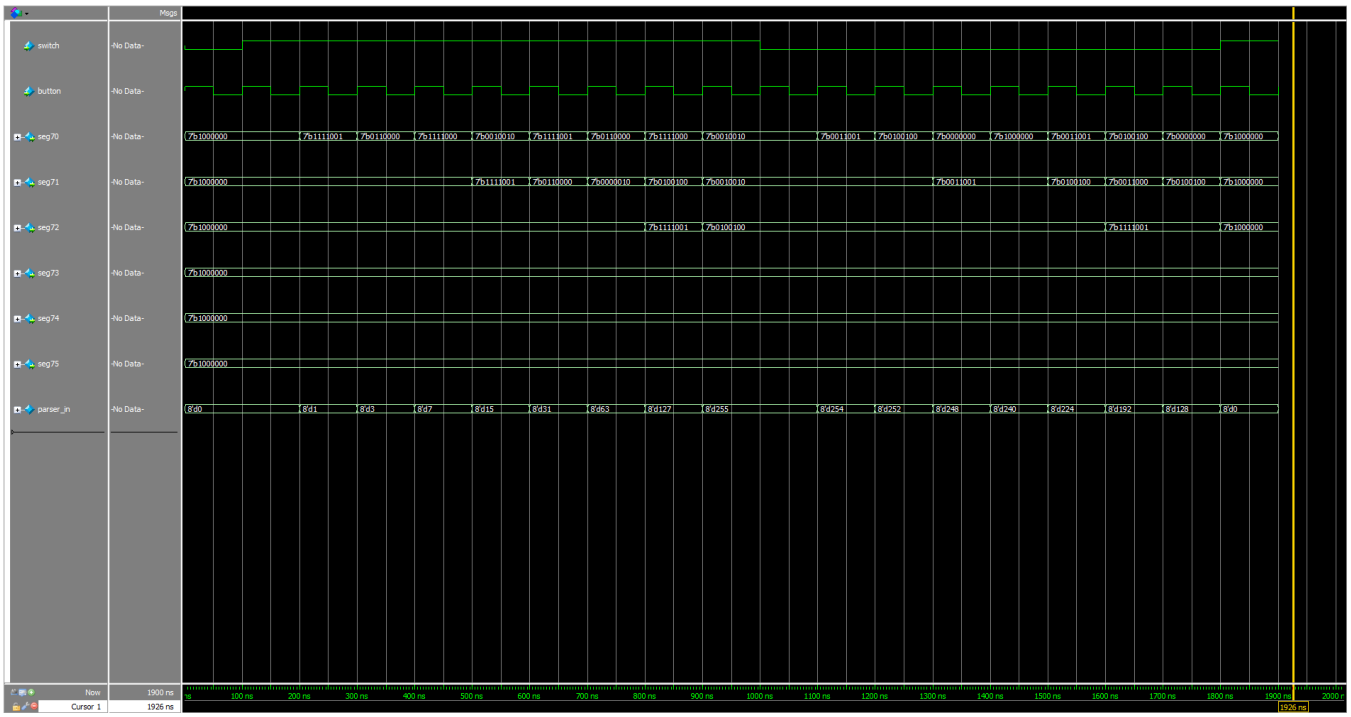


Figure 3: Simulation waveforms of a few test examples. High-resolution image

The button is the clock, and the switch is the data. From 200 ns to 1000 ns, the output of the SPI (e.g. *parse_in*) increases from 1, 3, 7, 15, 31, 63, 127, to 255. This is consistent with the expected SPI output (1, 11, 111, 1111, 11111, 111111, 1111111, 11111111) when adding 1 on each clock edge.

Similarly, from 1000 ns to 1900 ns, the system keeps receiving 0, and so the SPI module keeps adding 0 to the least significant position and "forgetting" the most significant bit. Specifically, the output of the SPI changes as follows: 11111111 (255), 11111110 (254), 11111100 (252), 11111000 (248), 11110000 (240), 11100000 (224), 11000000 (192), 100000000 (128), 00000000 (0). From these simulation results, we believe our source code is working as intended.

We upload the project to the real FPGA. We do thorough tests on the real board, and everything also seems to be working as intended. Therefore, we have successfully implemented an SPI transmission protocol using SystemVerilog.

# 4 Experiment Notes

## Reflection

This lab turned out to be easier than I expected. I spent a lot of time trying to implement the SPI module. But thanks to the bit shifting operator available in SystemVerilog, I was able to implement the SPI module using only one line of sequential logic (see the Appendix).

I had issues when I tried to simulate the project using ModelSim. For some reasons, the waveforms of the output didn't change when I changed the input. Ben helped me find a solution for the issue, though, by initializing the local variables to 0 right after declaring them. For example,

```
logic [7:0] tmp_out = 0;
```

## Study Questions

1. In your own words, describe the functionality of 4 wire SPI.

   A 4-wire SPI system connects the master and the slave(s) via four wires: *SCK*, *MOSI*, *MISO*, and *SS*. **SCK** is the **S**erial **C**loc**K** signal—generated by the master—which runs through all devices, sharing the same clock to everything else in the system. **MOSI** is **M**aster-**O**ut **S**lave-**I**n, which transmits the data from the master to a slave. Conversely, **MISO** (**M**aster-**I**n **S**lave-**O**ut) transmits the data from a slave to the master. **SS** (**S**lave **S**elect) is more tricky because it enables the master to communicate with multiple slaves. In this 4-wire SPI setup, *SS* of the master connects to all *SS* pins of the slaves. By this way, to transmit the data to a certain slave, the master must overflow the data. The *first* chunk of data will be tramistted to the *last* slave.

   The 4-wire SPI setup can be summarized by Figure 4 below:

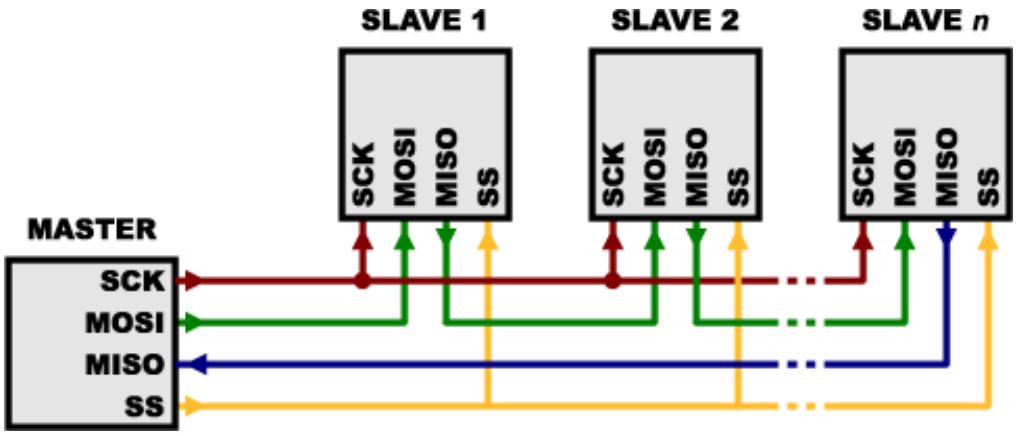


Figure 4: An implementation of a 4-wire SPI with $n$ slaves [1]

# Appendix

**lab5.qsf** (Pin Assignment)

---

```
set_location_assignment PIN_C14 -to seg70[0]
set_location_assignment PIN_E15 -to seg70[1]
set_location_assignment PIN_C15 -to seg70[2]
set_location_assignment PIN_C16 -to seg70[3]
set_location_assignment PIN_E16 -to seg70[4]
set_location_assignment PIN_D17 -to seg70[5]
set_location_assignment PIN_C17 -to seg70[6]
set_location_assignment PIN_C18 -to seg71[0]
```

```
46  set_location_assignment PIN_D18 -to seg71[1]
47  set_location_assignment PIN_E18 -to seg71[2]
48  set_location_assignment PIN_B16 -to seg71[3]
49  set_location_assignment PIN_A17 -to seg71[4]
50  set_location_assignment PIN_A18 -to seg71[5]
51  set_location_assignment PIN_B17 -to seg71[6]
52  set_location_assignment PIN_B20 -to seg72[0]
53  set_location_assignment PIN_A20 -to seg72[1]
54  set_location_assignment PIN_B19 -to seg72[2]
55  set_location_assignment PIN_A21 -to seg72[3]
56  set_location_assignment PIN_B21 -to seg72[4]
57  set_location_assignment PIN_C22 -to seg72[5]
58  set_location_assignment PIN_B22 -to seg72[6]
59  set_location_assignment PIN_F21 -to seg73[0]
60  set_location_assignment PIN_E22 -to seg73[1]
61  set_location_assignment PIN_E21 -to seg73[2]
62  set_location_assignment PIN_C19 -to seg73[3]
63  set_location_assignment PIN_C20 -to seg73[4]
64  set_location_assignment PIN_D19 -to seg73[5]
65  set_location_assignment PIN_E17 -to seg73[6]
66  set_location_assignment PIN_F18 -to seg74[0]
67  set_location_assignment PIN_E20 -to seg74[1]
68  set_location_assignment PIN_E19 -to seg74[2]
69  set_location_assignment PIN_J18 -to seg74[3]
70  set_location_assignment PIN_H19 -to seg74[4]
71  set_location_assignment PIN_F19 -to seg74[5]
72  set_location_assignment PIN_F20 -to seg74[6]
73  set_location_assignment PIN_J20 -to seg75[0]
74  set_location_assignment PIN_K20 -to seg75[1]
75  set_location_assignment PIN_L18 -to seg75[2]
76  set_location_assignment PIN_N18 -to seg75[3]
77  set_location_assignment PIN_M20 -to seg75[4]
78  set_location_assignment PIN_N19 -to seg75[5]
79  set_location_assignment PIN_N20 -to seg75[6]
80
81  set_global_assignment -name FAMILY "MAX 10"
82  set_global_assignment -name DEVICE 10M50DAF484C7G
83  set_global_assignment -name TOP_LEVEL_ENTITY SpiTopLevel
84  set_global_assignment -name ORIGINAL_QUARTUS_VERSION 18.0.0
85  set_global_assignment -name PROJECT_CREATION_TIME_DATE "11:04:13  OCTOBER 31, 2018"
86  set_global_assignment -name LAST_QUARTUS_VERSION "18.0.0 Lite Edition"
87  set_global_assignment -name PROJECT_OUTPUT_DIRECTORY output_files
88  set_global_assignment -name MIN_CORE_JUNCTION_TEMP 0
89  set_global_assignment -name MAX_CORE_JUNCTION_TEMP 85
90  set_global_assignment -name ERROR_CHECK_FREQUENCY_DIVISOR 256
91  set_global_assignment -name POWER_PRESET_COOLING_SOLUTION "23 MM HEAT SINK WITH 200 LFPM
    ↪  AIRFLOW"
```

```
set_global_assignment -name POWER_BOARD_THERMAL_MODEL "NONE (CONSERVATIVE)"


set_location_assignment PIN_C10 -to switch
set_location_assignment PIN_B8 -to button
set_global_assignment -name SYSTEMVERILOG_FILE SpiTopLevel.sv
set_global_assignment -name SYSTEMVERILOG_FILE Spi.sv
set_global_assignment -name SYSTEMVERILOG_FILE Parser.sv
set_global_assignment -name SYSTEMVERILOG_FILE Decoder.sv
set_global_assignment -name PARTITION_NETLIST_TYPE SOURCE -section_id Top
set_global_assignment -name PARTITION_FITTER_PRESERVATION_LEVEL PLACEMENT_AND_ROUTING
    ↪  -section_id Top
set_global_assignment -name PARTITION_COLOR 16764057 -section_id Top
set_instance_assignment -name PARTITION_HIERARCHY root_partition -to | -section_id Top
```

---

### SpiTopLevel.sv

---

```
module SpiTopLevel(
    input logic switch,
    input logic button,
    output logic [6:0] seg70, seg71, seg72, seg73, seg74, seg75
);

    logic [7:0] parser_in = 0;
    logic [3:0] parser_out0, parser_out1, parser_out2 = 0;

    Spi s(
        .clk(button),
        .data_in(switch),
        .out_parser(parser_in)
    );

    Parser p(
        .in(parser_in),
        .out0(parser_out0),
        .out1(parser_out1),
        .out2(parser_out2)
    );

    Decoder d0(
        .num_in(parser_out0),
        .segments(seg70)
    );
    Decoder d1(
        .num_in(parser_out1),
```

```
        .segments(seg71)
    );
    Decoder d2(
        .num_in(parser_out2),
        .segments(seg72)
    );
    Decoder d3(
        .num_in(0),
        .segments(seg73)
    );
    Decoder d4(
        .num_in(0),
        .segments(seg74)
    );
    Decoder d5(
        .num_in(0),
        .segments(seg75)
    );

endmodule
```

---

## Spi.sv

---

```
module Spi(
    input logic clk,
    input logic data_in,
    output logic [7:0] out_parser
);

    logic [7:0] out = 0;

    always_ff @(posedge clk)
        out <= (out << 1) + data_in;

    assign out_parser = out;

endmodule
```

---

## Parser.sv

---

```
module Parser(
    input logic [7:0] in,
```

```
    output logic [3:0] out0, out1, out2
);

    always_comb begin
        out0 = in % 10;
        out1 = (in / 10) % 10;
        out2 = (in / 100) % 10;
    end

endmodule
```

---

**Decoder.sv**

---

```
module Decoder(
    input logic [3:0] num_in,
    output logic [6:0] segments
);
    always_ff @(*)
        case(num_in)
            0: segments <= 7'b100_0_000;
            1: segments <= 7'b111_1_001;
            2: segments <= 7'b010_0_100;
            3: segments <= 7'b011_0_000;
            4: segments <= 7'b001_1_001;
            5: segments <= 7'b001_0_010;
            6: segments <= 7'b000_0_010;
            7: segments <= 7'b111_1_000;
            8: segments <= 7'b000_0_000;
            9: segments <= 7'b001_1_000;
            default: segments <=  7'b111_1111;
        endcase

endmodule
```

---

# References


[1] M. Grusin, "Serial peripheral interface (spi)." https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi.