Spring 2022

California State University, Northridge

Department of Electrical & Computer Engineering

Final Project Report
SPI Protocol

ECE 526

May 16, 2022

Written By: Angel Rosas De Luis

1. ***Purpose:***

The purpose of this project was to design and implement a serial peripheral interface (SPI) protocol. As well as, to investigate and familiarize ourselves with how this type of synchronous communication protocol is used.

2. ***Software Used:***

MS Office 2016
MobaXterm 21.5

3. ***Background***

For this project, research was first conducted to learn about the SPI protocol before beginning the design process. An SPI protocol can be described as a synchronous communication protocol that can provide a full duplex communication between a microcontroller and its peripherals. Full duplex communication allows for data to be transmitted and received simultaneously.

In an SPI protocol, one master node is allowed per design but many slave nodes or peripherals can be connected to the one master node. An example of how an SPI protocol behaves is shown in Figure 1. In this particular design, four wires control the transfer of data between a processor and one peripheral. The MasterOutSlaveIn (MOSI) wire sends data from the master node to the slave node. The MasterInSlaveOut (MISO) wire transmits back date from the slave node to the master node that is controlling the particular slave node. The SPI clock (SCK) wire is the clock signal that synchronizes the data transfer of the MISO and MOSI wires. The Slave select (SS or CS) wire controls which peripheral should the master slave communicate to if many slaves or peripherals are connected. To replicate this function I design a state machine diagram to implement the SPI protocol in Verilog shown in Figure 2.

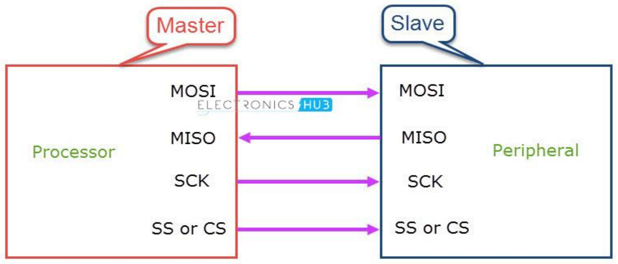


Figure 1: Sample design of an SPI protocol

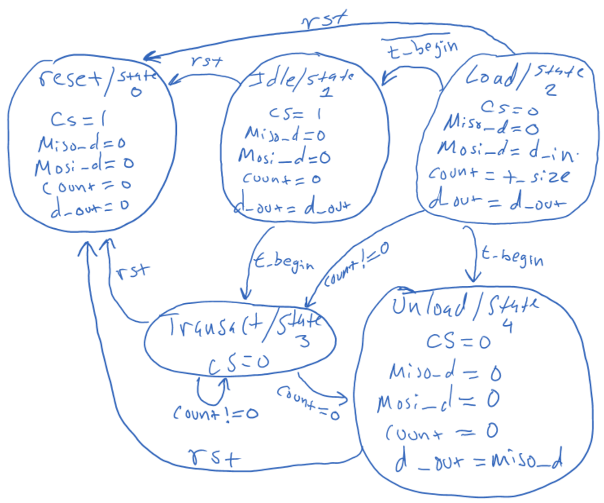
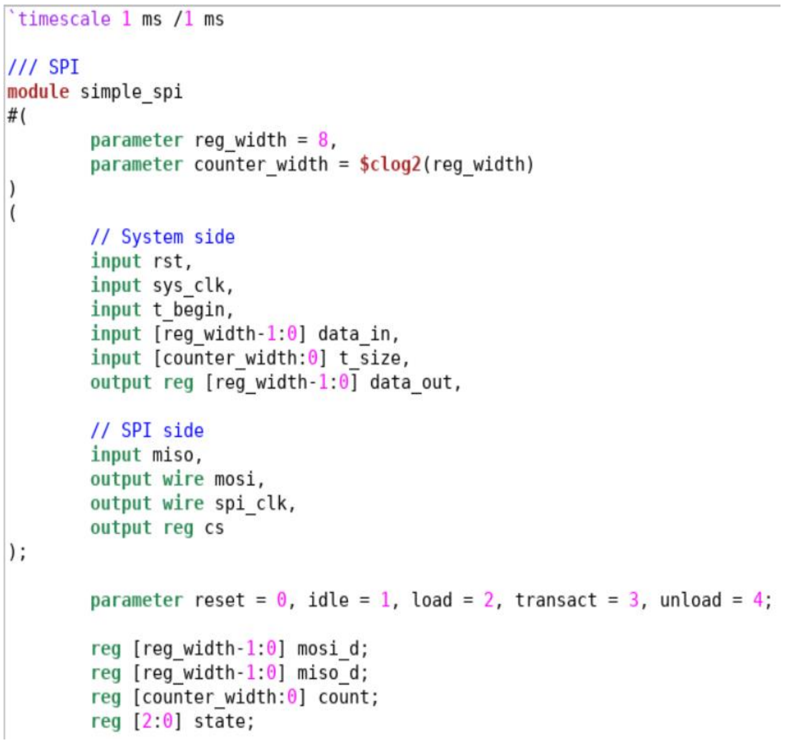


Figure 2: State machine diagram for SPI protocol

4. ***Procedure and Results:***

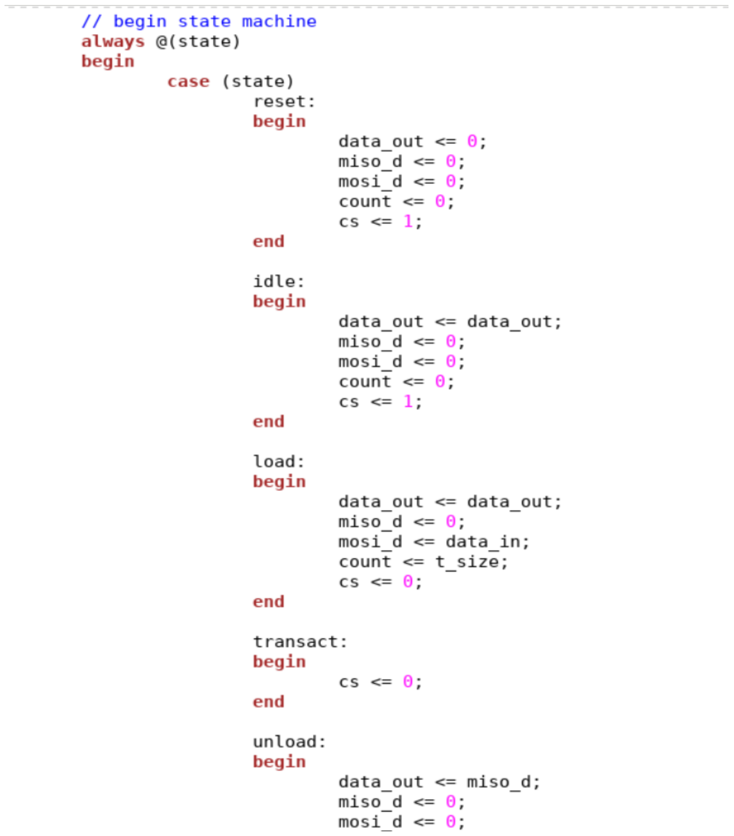
Step 1: For this lab, I first logged onto MobaXterm to start session dcd142. After logging in, I created a new directory called 526LSP22_SPI_Project using mkdir in the command window and changed into the new directory using the cd command.

Step 2: Next, I opened a text editor window by typing gedit & on the command window. Next, I was able to create one Verilog file named simple_SPI..v. A screenshot of the codes are shown in Figure 1-?.



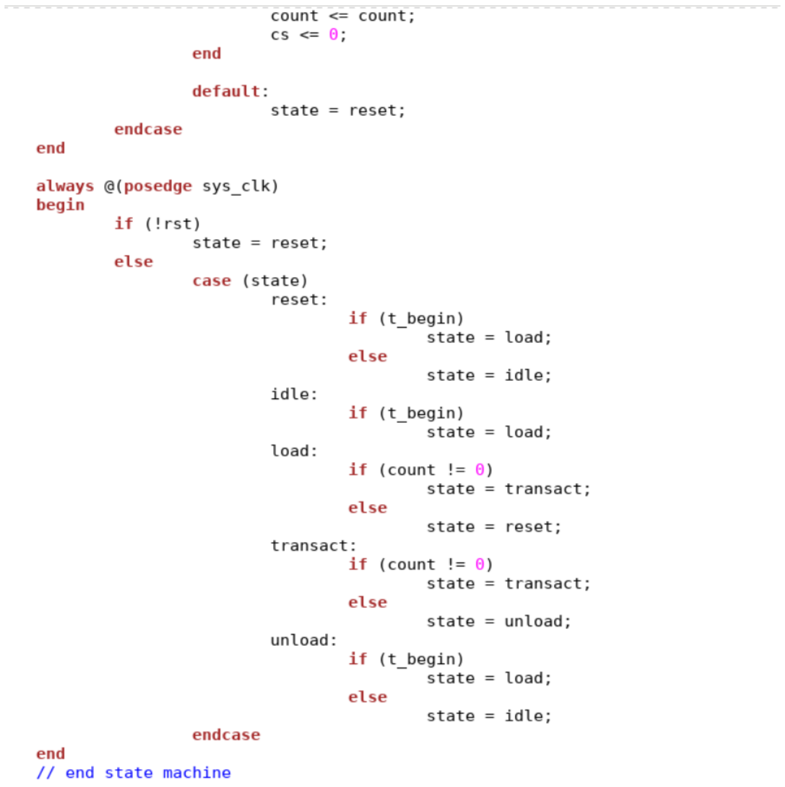
```
`timescale 1 ms /1 ms

/// SPI
module simple_spi
#(
        parameter reg_width = 8,
        parameter counter_width = $clog2(reg_width)
)
(
        // System side
        input rst,
        input sys_clk,
        input t_begin,
        input [reg_width-1:0] data_in,
        input [counter_width:0] t_size,
        output reg [reg_width-1:0] data_out,

        // SPI side
        input miso,
        output wire mosi,
        output wire spi_clk,
        output reg cs
);

        parameter reset = 0, idle = 1, load = 2, transact = 3, unload = 4;

        reg [reg_width-1:0] mosi_d;
        reg [reg_width-1:0] miso_d;
        reg [counter_width:0] count;
        reg [2:0] state;
```

Figure 3: Screenshot of parameters set for simple_SPI.v file



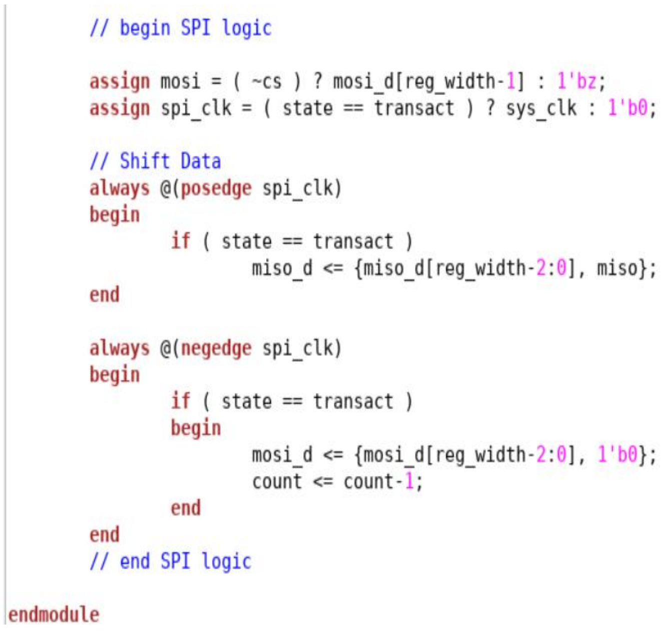
```
        // begin state machine
        always @(state)
        begin
                case (state)
                        reset:
                        begin
                                data_out <= 0;
                                miso_d <= 0;
                                mosi_d <= 0;
                                count <= 0;
                                cs <= 1;
                        end

                        idle:
                        begin
                                data_out <= data_out;
                                miso_d <= 0;
                                mosi_d <= 0;
                                count <= 0;
                                cs <= 1;
                        end

                        load:
                        begin
                                data_out <= data_out;
                                miso_d <= 0;
                                mosi_d <= data_in;
                                count <= t_size;
                                cs <= 0;
                        end

                        transact:
                        begin
                                cs <= 0;
                        end

                        unload:
                        begin
                                data_out <= miso_d;
                                miso_d <= 0;
                                mosi_d <= 0;
```

Figure 4: Screenshot of outputs for state machine from simple_SPI.v file



```
                                count <= count;
                                cs <= 0;
                        end

                        default:
                                state = reset;
                endcase
        end

        always @(posedge sys_clk)
        begin
                if (!rst)
                        state = reset;
                else
                        case (state)
                                reset:
                                        if (t_begin)
                                                state = load;
                                        else
                                                state = idle;
                                idle:
                                        if (t_begin)
                                                state = load;
                                load:
                                        if (count != 0)
                                                state = transact;
                                        else
                                                state = reset;
                                transact:
                                        if (count != 0)
                                                state = transact;
                                        else
                                                state = unload;
                                unload:
                                        if (t_begin)
                                                state = load;
                                        else
                                                state = idle;
                        endcase
        end
        // end state machine
```

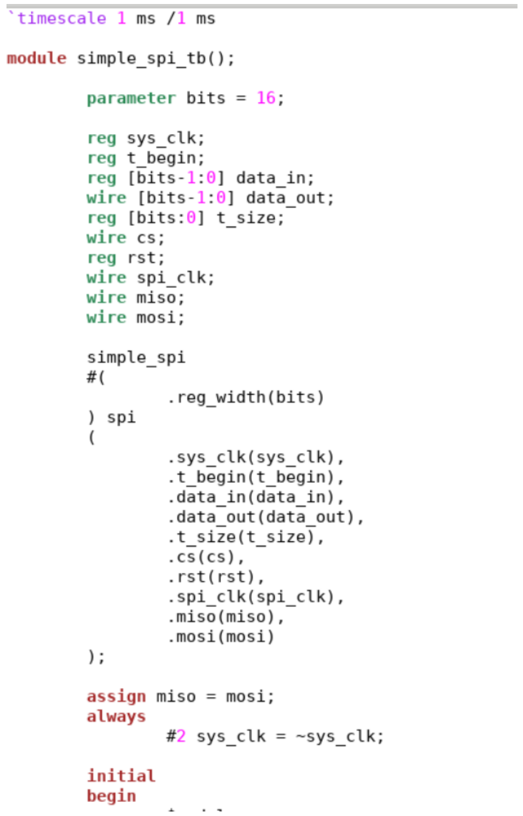
Figure 5: Screenshot of controls for state transitions inside simple_SPI.v file



```
        // begin SPI logic

        assign mosi = ( ~cs ) ? mosi_d[reg_width-1] : 1'bz;
        assign spi_clk = ( state == transact ) ? sys_clk : 1'b0;

        // Shift Data
        always @(posedge spi_clk)
        begin
                if ( state == transact )
                        miso_d <= {miso_d[reg_width-2:0], miso};
        end

        always @(negedge spi_clk)
        begin
                if ( state == transact )
                begin
                        mosi_d <= {mosi_d[reg_width-2:0], 1'b0};
                        count <= count-1;
                end
        end
        // end SPI logic

endmodule
```

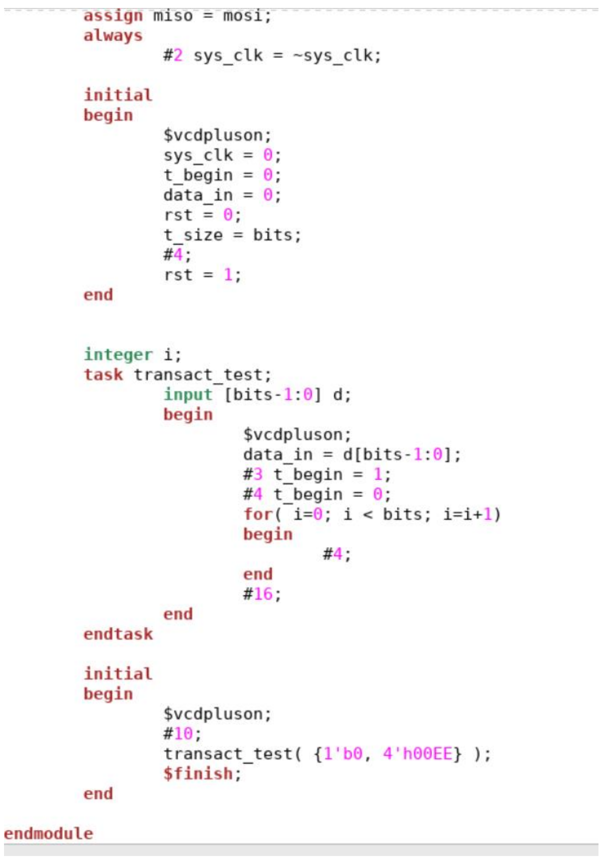
Figure 6: Screenshot of shift registers for SPI transmission in simple_SPI.v

Step 3: Next I created a testbenches in gedit window called simple_SPI_tb.v shown in Figure . In the SPI testbench, I needed to implement my simple_SPI.v design and show that it can read and implement the state machine that I designed for this project shown in Figure 2.



```
`timescale 1 ms /1 ms

module simple_spi_tb();

        parameter bits = 16;

        reg sys_clk;
        reg t_begin;
        reg [bits-1:0] data_in;
        wire [bits-1:0] data_out;
        reg [bits:0] t_size;
        wire cs;
        reg rst;
        wire spi_clk;
        wire miso;
        wire mosi;

        simple_spi
        #(
                .reg_width(bits)
        ) spi
        (
                .sys_clk(sys_clk),
                .t_begin(t_begin),
                .data_in(data_in),
                .data_out(data_out),
                .t_size(t_size),
                .cs(cs),
                .rst(rst),
                .spi_clk(spi_clk),
                .miso(miso),
                .mosi(mosi)
        );

        assign miso = mosi;
        always
                #2 sys_clk = ~sys_clk;

        initial
        begin
```

Figure 5: Part 1 of screenshot of simple_SPI_tb.v



```
        assign miso = mosi;
        always
                #2 sys_clk = ~sys_clk;

        initial
        begin
                $vcdpluson;
                sys_clk = 0;
                t_begin = 0;
                data_in = 0;
                rst = 0;
                t_size = bits;
                #4;
                rst = 1;
        end


        integer i;
        task transact_test;
                input [bits-1:0] d;
                begin
                        $vcdpluson;
                        data_in = d[bits-1:0];
                        #3 t_begin = 1;
                        #4 t_begin = 0;
                        for( i=0; i < bits; i=i+1)
                        begin
                                #4;
                        end
                        #16;
                end
        endtask

        initial
        begin
                $vcdpluson;
                #10;
                transact_test( {1'b0, 4'h00EE} );
                $finish;
        end

endmodule
```

Figure 6: Part 2 of screenshot of simple_SPI_tb.v

Step 4: In order to be able to compile and run my testbench I used the common compiler command,vcs -debug -full64 simple_SPI.v simple_SPI_tb.v, on the command window of MobaXterm. Once all my files are compiled, I ran a simulation using “simv” on the command window. Next, by using “dve -full64,” I was able to open the DVE simulation window. Then, I located the file tab and then the open database tab to locate the file named “vcdplus.vpd” to open. Once the file has been opened, I selected all the inputs and outputs that I needed to create the waveforms shown in Figure 7.

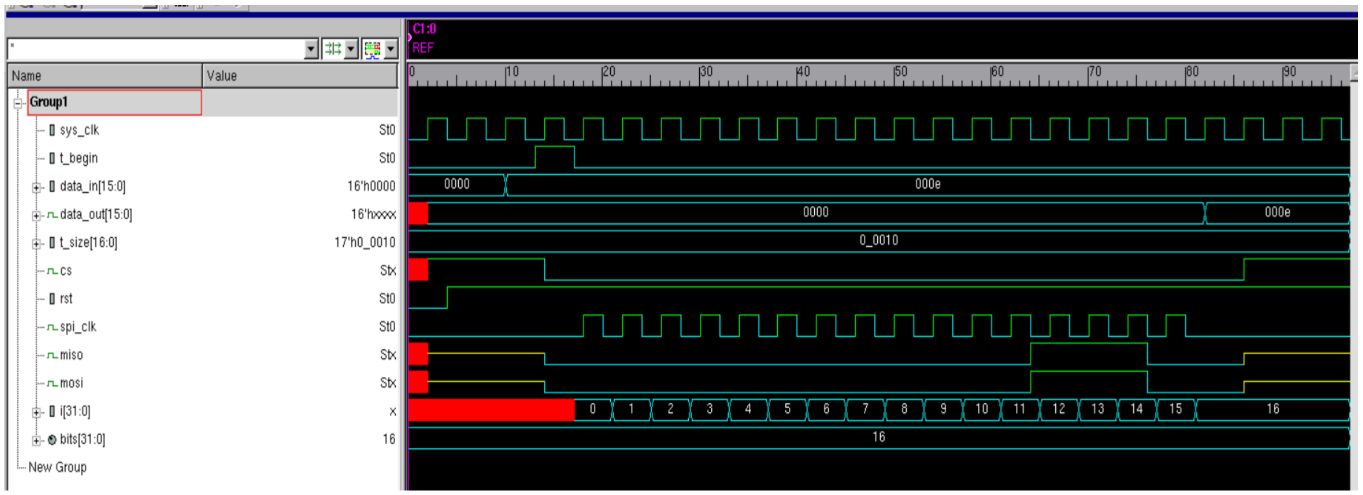


Figure 7: Screenshot of waveforms from simple_SPI_tb.v

5. *Discussion/Analysis:*

For this project, I needed to research and design an SPI protocol that is capable of full duplex communication between one microcontroller and one peripheral. While designing the source code for the SPI protocol, there were many states that need to be executed before the shift registers begin transmitting data across the MISO and MOSI communication wires. Some difficulties that I ran into was writing a correct test bench for my SPI protocol design. From my simulation shown in Figure 7, I observed that I was unable to achieve full transmission of 8 bits and show that only 4 bits are read and transmitted to and from the MISO and MOSI transmission wires. However, the SPI protocol seems to be operating properly as it begins transmission of the data from the master node when the clocks from the SPI and system clocks synchronize. Some future fixes and troubleshooting is need to have the correct of amount of bits to be read and transmitted by the master and slave nodes from the SPI protocol.

6. *Conclusion:*

In this project, I had to practice writing different cases statements and definitions in order to replicate the function of the state machine that I designed for a simple SPI protocol that would achieve communication between only one microcontroller and one peripheral. In general, this communication protocol is best used for communication of peripheral to a microcontroller that are located on the same board.

I hereby attest that this lab report is entirely my own work. I have not copied either code or text from anyone, nor have I allowed or will I allow anyone to copy my work.

Name (printed) Angel Rosas De Luis

Name (signed) [illegible] Date 05/16/22