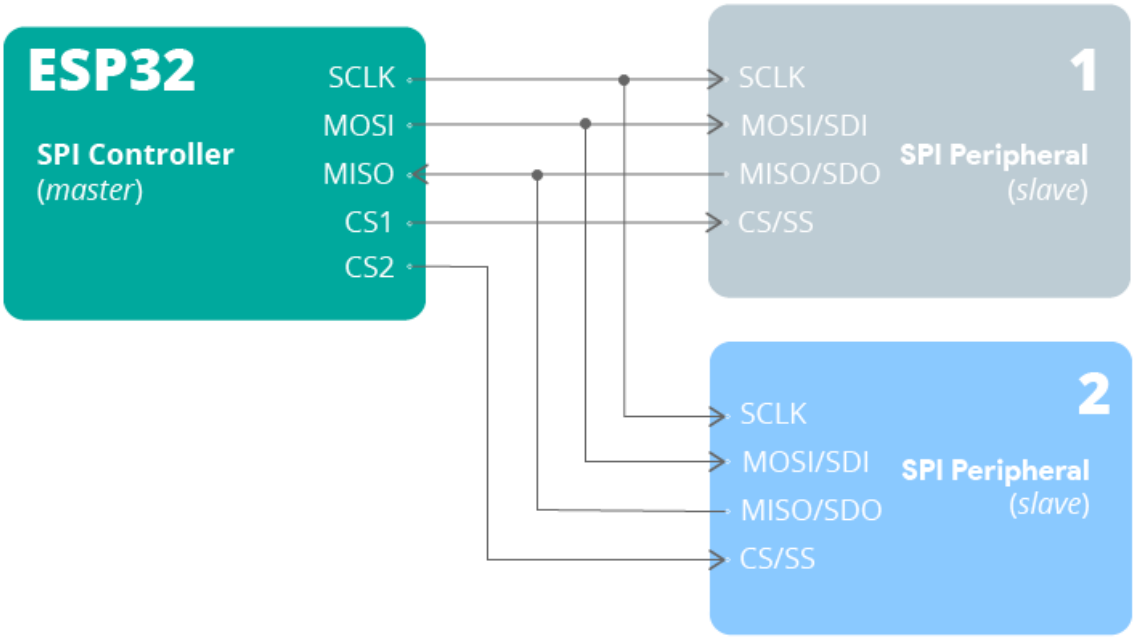
SPI PROTOCOL IMPLEMENTATION USING VERILOG HDL



NAVYA P VERNEKAR

PGDVD 15

U2023032930854

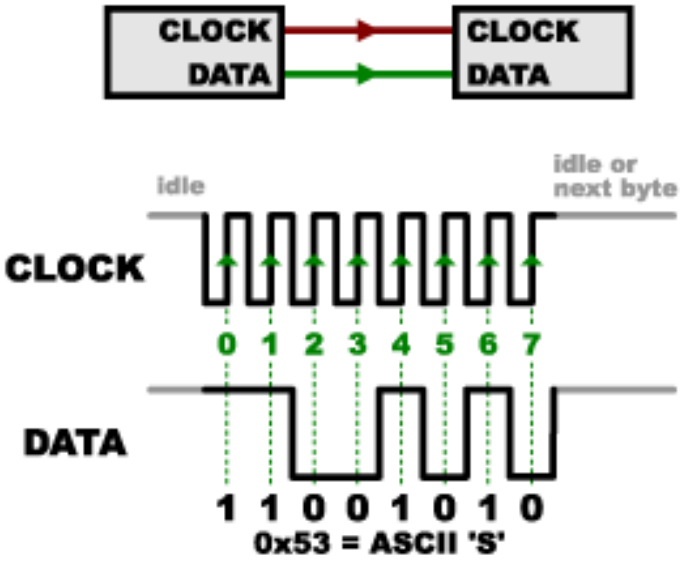
INTRODUCTION

The Serial Peripheral Interface (SPI) protocol is a widely used synchronous serial communication protocol that enables efficient data exchange between microcontrollers, sensors, and other digital devices. It plays a crucial role in facilitating communication between integrated circuits within embedded systems. In this section, we will provide a comprehensive introduction to the SPI protocol and highlight its significance in digital communication.

Serial Peripheral Interface (SPI) Protocol Overview:

The SPI protocol was originally developed by Motorola (now Freescale) as a means to connect multiple peripheral devices to a microcontroller or microprocessor. It has since become a de facto standard for interconnecting digital devices due to its simplicity, flexibility, and efficiency.

The SPI protocol operates in a master-slave architecture, where a master device controls the communication and one or more slave devices respond to the master's commands. The protocol employs a synchronous communication scheme, meaning that data transmission is tightly synchronized to a clock signal.



Significance in Digital Communication:

The SPI protocol holds significant importance in various domains of digital communication and embedded systems for several reasons:

1. **High Speed and Efficiency:** SPI communication is known for its high data transfer rates compared to other serial communication protocols. This makes it suitable for applications that require rapid data exchange, such as real-time sensor data acquisition, memory interfacing, and display control.
2. **Simple and Versatile:** The SPI protocol's simplicity and flexibility make it well-suited for a wide range of applications. It uses a minimal number of signal lines (typically four: SCLK, MOSI, MISO, and SS/CS), making it efficient in terms of wiring and PCB layout.
3. **Full-Duplex Communication:** SPI supports full-duplex communication, meaning that data can be transmitted and received simultaneously, eliminating the need for time-consuming handshaking mechanisms often required in other protocols.
4. **Device Interfacing:** SPI is commonly used to interface microcontrollers with various peripheral devices such as sensors (temperature, pressure, accelerometers), EEPROMs, flash memory, DACs (Digital-to-Analog Converters), ADCs (Analog-to-Digital Converters), and display controllers.
5. **Configuration and Control:** SPI is frequently employed to configure and control peripheral devices. For instance, a microcontroller can use SPI to set operating parameters of sensors or configure the behaviour of display modules.
6. **Compact Communication Protocol:** The SPI protocol's minimal overhead and clear synchronization make it ideal for communication within resource-constrained systems where efficiency is paramount.
7. **Hardware Simplicity:** The SPI protocol's implementation requires relatively simple hardware compared to other communication protocols like I2C or UART, which can be beneficial in terms of both hardware design and power consumption.

BACKGROUND AND THEORY

Overview of the SPI Protocol:

The Serial Peripheral Interface (SPI) protocol is a synchronous serial communication standard that facilitates the exchange of data between a master device and one or more peripheral devices. It operates on a full-duplex communication model, allowing data to be simultaneously transmitted and received. SPI is widely used in embedded systems, microcontroller interfacing, and communication with various peripherals like sensors, memory devices, and display controllers.

Basic Working Principle:

The SPI protocol is based on a master-slave architecture, where one device (the master) initiates and controls the communication, while the other device(s) (the slave(s)) respond to the master's commands. The protocol uses a clock signal (SCLK) to synchronize data transfer between the devices. The data is transferred in a series of binary bits, usually in 8-bit chunks, and each bit is synchronized to the rising or falling edge of the clock signal.

Signal Lines in SPI Communication:

1. **SCLK (Serial Clock):** This line carries the clock signal generated by the master device. It synchronizes the data transmission between the master and the slave devices. All data transfers occur at the edges (rising or falling) of the clock signal.
2. **MOSI (Master Output Slave Input):** This line is used by the master to send data to the slave. The master drives the MOSI line with the data it wants to transmit, and the slave reads this data on its end.
3. **MISO (Master Input Slave Output):** This line is used by the slave to send data back to the master. The slave drives the MISO line with the data it wants to transmit, and the master reads this data on its end.
4. **SS/CS (Slave Select/Chip Select):** This line is used to select the specific slave device with which the master wants to communicate. In systems with multiple slave devices, the master asserts the SS/CS line of the desired slave to indicate that it is the target of the communication.

Significance of Each Signal Line:

SCLK: The Serial Clock line ensures synchronous communication between the master and slave devices. It provides a common timing reference, ensuring that both devices are aligned in terms of data transmission and reception.

MOSI: The Master Output Slave Input line allows the master to send data to the slave. This is essential for transmitting commands, data, or configuration settings from the master to the slave device.

MISO: The Master Input Slave Output line enables the slave to send data back to the master. This is crucial for returning responses, measurements, or data requested by the master.

SS/CS: The Slave Select or Chip Select line is used to enable a specific slave device for communication. When the master asserts the SS/CS line for a particular slave, it indicates that the communication is intended for that specific slave.

DESIGN ARCHITECTURE

The Verilog code represents an implementation of an SPI master module. Let's break down the key components and their interactions:

Input and Output Ports:

- clk: Clock signal input.
- rst: Reset signal input.
- input_data: 16-bit input data to be transmitted over SPI.
- MISO: Master Input Slave Output signal (received data from slave).
- MOSI: Master Output Slave Input signal (transmitted data to slave).
- spi_cs: Chip Select signal output.
- spi_sclk: Serial Clock signal output.
- counter: 5-bit counter value indicating the current state.
- data_out_mosi: 16-bit output data representing data transmitted over MOSI.
- data_out_miso: 16-bit output data representing data received over MISO.

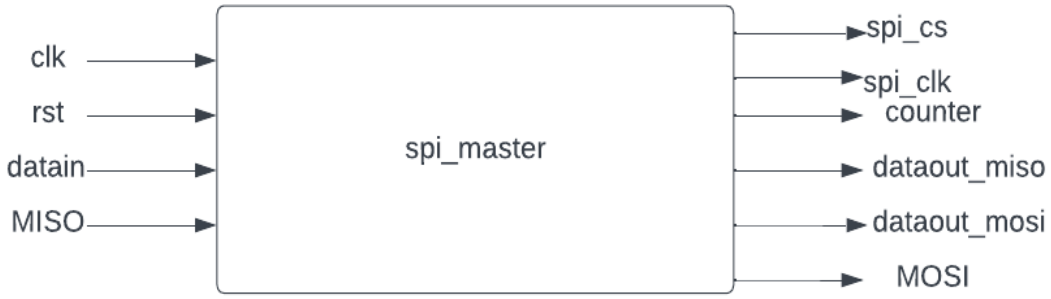
Registers and State Machines:

- mosi: 1-bit register representing the data to be transmitted (MOSI).
- count: 5-bit counter representing the data bit index being transmitted.
- count1: 5-bit counter for receiving data from the slave.
- cs: 1-bit register controlling the Chip Select line.
- sclk: 1-bit register controlling the Serial Clock line.
- state: 3-bit register representing the current state of the state machine.

Always Blocks:

- The first always block handles data reception from the slave. It operates on the rising edge of spi_sclk or when spi_cs is asserted. It reads data from the MOSI line when the Chip Select line is not active and stores it in the mem and mem1 registers based on the count1 value.
- The second always block handles the state machine that controls SPI communication. It operates on the rising edge of clk or when rst is asserted. It goes through different states (0, 1, and 2) to manage the data transmission process, the Serial Clock, and Chip Select signals.

Block Diagram :



Input Path:

input_data feeds data to be transmitted over the MOSI line.

The state machine controls the selection of data bits and the transmission process.

State Machine:

The state machine (state) manages the SPI communication process.It progresses from state 0 (idle) to state 1 (transmitting data) and then to state 2 (managing SCLK) before resetting back to state 0.

Synchronous Clock and Reset:

The clock signal (clk) and reset signal (rst) synchronize and control the operations of the state machine.

Serial Clock and Chip Select:

The sclk signal controls the Serial Clock line.The cs signal controls the Chip Select line.

Data Transmission and Reception:

Data is transmitted to the slave using the MOSI line based on the current count.Received data from the slave is stored in the mem1 register using the MISO line.

Output Path:

The output paths involve assigning the respective internal signals and registers to the output ports (spi_cs, spi_sclk, MOSI, counter, data_out_mosi, data_out_miso).

Verilog Implementation:

Signal Handling:

The code handles four primary signals: spi_cs, spi_sclk, MOSI, and MISO.

- spi_cs and spi_sclk are generated internally and connected to output ports.
- MOSI is driven by the master to send data to the slave.
- MISO is used to receive data from the slave.

Data Shift Registers:

The master transmits and receives data using shift registers. The mosi register holds the data bit to be transmitted over MOSI.The mem register stores transmitted data bits for data_out_mosi. The mem1 register stores received data bits for data_out_miso.

Synchronization:

The clk signal is used to synchronize the operations.The state machine progresses through different states based on the clock signal.The rising edge of clk triggers the state transitions.

Key Code Snippets:

Signal Handling:

```
always @(posedge spi_sclk or spi_cs) begin
    if(spi_cs) begin
        mem <= 16'b0;
        mem1 <= 16'b0;
        count1 <= 16'd16;
    end
    else begin
        mem[count1] = MOSI;
        mem1[16 - count1] = MISO;
        count1 = count1 - 1;
    end
end
```

State Machine:

```
always @(posedge clk or posedge rst) begin
    if(rst) begin
        mosi <= 0;
        count <= 5'd16;
        cs <= 1'b1;
        sclk <= 1'b0;
    end
    else begin
        case(state)
            0: begin
                sclk <= 1'b0;
                cs <= 1'b1;
                state <= 1;
               end
            1: begin
                sclk <= 1'b0;
                cs <= 1'b0;
                mosi <= input_data[count - 1];
                count <= count - 1;
                state <= 2;
               end
            2: begin
                sclk <= 1'b1;
                if(count > 0)
                    state <= 1;
                else begin
                    count <= 16;
                    state <= 0;
                end
              end
            default: state <= 0;
        endcase
    end
end
```

Output Assignments:

```
assign spi_cs = cs;
assign spi_sclk = sclk;
assign MOSI = mosi;
assign counter = count;
assign data_out_mosi = mem;
assign data_out_miso = mem1;
```

Explanation:

1. Handling Data Reception:

- The first always block, which is triggered by either a rising edge of spi_sclk or when spi_cs is asserted, manages the reception of data from the slave device. Let's break down how this block works:
- When spi_cs is asserted, it indicates that a new communication session is starting. As a result, the mem and mem1 registers, which hold received data bits, are reset to zeros, and the counter count1 is initialized to 16.
- When spi_cs is not asserted (i.e., when data reception is ongoing), the block captures the incoming data bit from the MISO line, which represents data sent by the slave. The incoming data bit is stored in the appropriate position within the mem1 register using the value of count1.
- The count1 counter is then decremented by 1 in each cycle, effectively moving to the next bit position within the mem1 register. This allows the code to capture each bit of received data and store it in reverse order.This block ensures that the received data bits are properly captured and stored in the mem1 register, ready for later output.

2. State Machine for SPI Communication:

The second always block is responsible for managing the state machine that controls the SPI communication process. This state machine progresses through different states (0, 1, and 2) to handle various aspects of the communication sequence.

- State 0 (Idle): In this state, the Serial Clock (sclk) is set to low (0), and the Chip Select (cs) line is asserted (1). This signifies that the communication is not active.
- State 1 (Transmitting Data): This state involves transitioning the state machine to send data to the slave. The Serial Clock remains low (0), and the Chip Select line is de-asserted (0), indicating that data transmission is about to start. The data bit to be transmitted is taken from the input_data based on the value of the count counter. This data bit is sent over the MOSI line (MOSI <= input_data[count - 1]), and the counter is decremented.
- State 2 (Serial Clock): In this state, the Serial Clock is toggled to high (1), indicating that the data bit is being shifted out or received by the slave. This state checks whether there are more bits to be transmitted by evaluating count > 0. If there are more bits to send, the state machine transitions back to state 1 to continue transmitting. Otherwise, if all bits have been transmitted, the counter is reset to 16, and the state machine goes back to state 0 (Idle).

3. Assign Statements and Output Ports:

The assign statements are used to connect internal registers and signals to the module's output ports. These statements ensure that the values of the signals, counters, and data registers are reflected at the module's outputs and can be accessed externally.

Simulation and Verification:

Software Platform:

Vivado is being utilized as the primary tool for simulating and verifying the SPI protocol implementation. Vivado, developed by Xilinx, is a comprehensive design environment that encompasses various tools for FPGA and digital design, including simulation capabilities.

Testbench Strategy:

The testbench strategy involves meticulously evaluating the correctness of the SPI protocol implementation using a well-structured simulation scenario. Here is a breakdown of the strategy:

Signal Initialization:

The testbench initializes key signals such as clock (clk), reset (rst), and input data (datain) to their initial values at the beginning of the simulation.

Clock Generation:

A designated always block is responsible for generating the clock signal (clk). It toggles the clock at regular intervals of 5 time units, creating a consistent clock waveform.

Reset Initialization:

Initially, the reset signal (rst) is asserted to establish a known starting point for the design. After a brief period of time (#10), the reset signal is de-asserted, allowing the design to begin its operations.

Data Transmission and Monitoring:

The testbench simulates data transmission scenarios and simultaneously monitors the reception of data (MISO) from the SPI master module.

Different rounds of data transmission are simulated with controlled delays (#335).

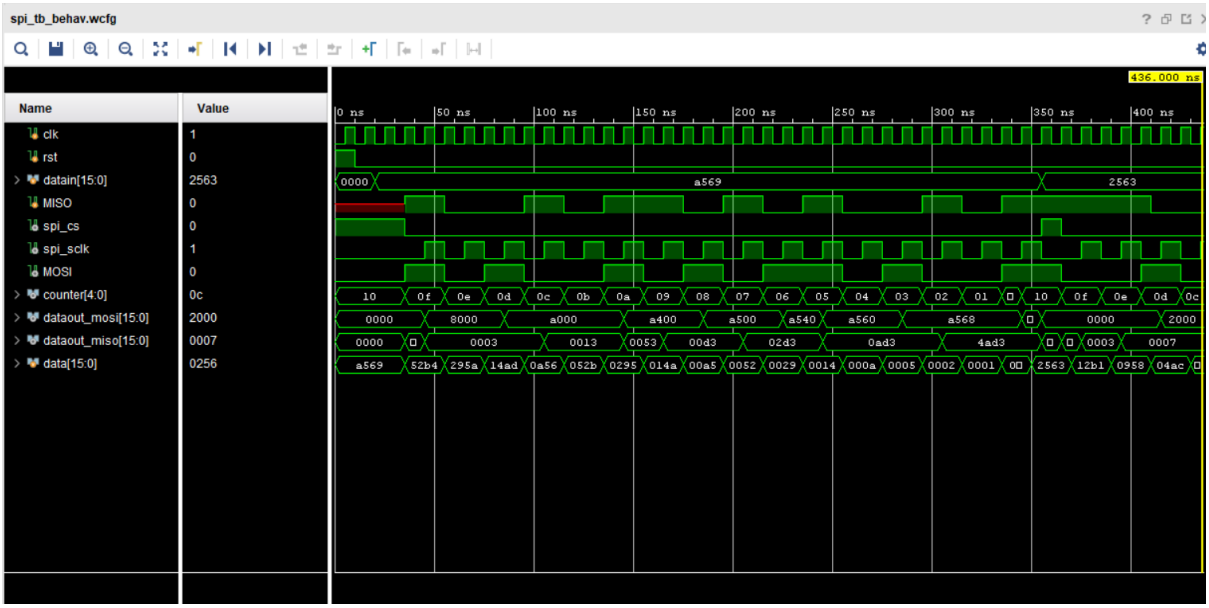
Within each transmission round, individual data bits (MOSI) are transmitted and subsequently received (MISO) using a repeat loop structure.

Simulation Termination:

Upon completion of the test scenarios, the simulation is concluded using the $stop command after a specified duration (#500), indicating the desired simulation time.

Simulation Waveforms:

The simulation waveforms showcase the dynamic behaviour of relevant signals during the simulation process, offering visual insights into the SPI protocol's successful operation. The waveforms exhibit the following aspects:



Clock Signal (clk): A continuous waveform demonstrating the clock signal's oscillation at a predefined frequency.

Reset Signal (rst): The reset signal initially remains asserted and subsequently transitions to de-asserted after a delay (#10).

Input Data (datain): The waveform illustrates changing input data values (datain) at specific simulation time points.

Serial Clock (spi_sclk): The waveform corresponds to the Serial Clock signal, visually representing data transmission timing.

Master Output Slave Input (MOSI) and Master Input Slave Output (MISO): The waveforms show the transmission of data from the master to the slave (MOSI) and the reception of data from the slave to the master (MISO).

Chip Select (spi_cs): The waveform exhibits Chip Select activation and deactivation events.

By carefully examining these simulation waveforms, it is verified that the SPI protocol implementation effectively manages data transmission and reception, adhering to the specified timing and synchronization requirements.

CONCLUSION

The project was successfully executed, encompassing the implementation and simulation of an SPI protocol using Verilog HDL within the Vivado environment. The core objective of developing a functional SPI master module proficient in managing data transmission, synchronization, and reception was achieved. The simulation results effectively substantiated the accuracy of the protocol's execution, thereby validating the project's implementation.

APPENDIX

Design:

```
module
spi_master(clk,rst,input_data,MISO,MOSI,spi_cs,spi_sclk,counter,data_out_mosi,data_out_miso);

input clk,rst,MISO;

input [15:0] input_data;

output spi_cs,spi_sclk,MOSI;

output [15:0] data_out_mosi;

output [4:0] counter;

output [15:0] data_out_miso;


reg mosi;

reg [4:0] count;

reg [4:0] count1;

reg cs;

reg sclk;

reg [2:0] state;

reg [15:0] mem;

reg [15:0] mem1;


always @(posedge spi_sclk or spi_cs) begin

if(spi_cs)

  begin mem <= 16'b0;

       mem1 <= 16'b0;

       count1 <= 16'd16;

       end

else begin

   mem[count1] = MOSI;

   mem1[16 - count1] = MISO;

   count1 = count1 -1;
```

```
    end
end
always @(posedge clk or posedge rst)
if(rst) begin
    mosi <= 0;
    count <= 5'd16;
    cs <= 1'b1;
    sclk <= 1'b0;
  end
else begin
    case(state)
    0: begin
        sclk <= 1'b0;
        cs <= 1'b1;
        state <= 1;
       end
    1: begin
        sclk <= 1'b0;
        cs <= 1'b0;
        mosi <= input_data[count -1 ];
        count <= count -1;
        state <= 2;
       end
   2: begin
        sclk <= 1'b1;
        if(count > 0)
            state <=1 ;
        else begin
            count <= 16;
            state <= 0;
```

```
        end
    end
  default: state <=0;
  endcase
end

assign spi_cs = cs;
assign spi_sclk = sclk;
assign MOSI = mosi;
assign counter = count;
assign data_out_mosi = mem;
assign data_out_miso = mem1;
endmodule
```

Testbench:

```
module spi_tb();
reg clk,rst;
reg [15:0] datain;
reg MISO;
wire spi_cs,spi_sclk;
wire  MOSI;
wire [4:0] counter;
wire [15:0] dataout_mosi;
wire [15:0] dataout_miso;
reg  [15:0] data;
spi_master dut (clk,rst,datain,MISO,MOSI,spi_cs,spi_sclk,counter,dataout_mosi,dataout_miso);

initial begin
   clk = 0;
   rst = 1;
   datain = 0;
   end
```

```
always #5 clk=~clk;

initial begin
#10 rst=1'b0;
#10 datain=16'hA569;
#335 datain=16'h2563;
#335 datain=16'h6A61;
#335 datain=16'h7635;
#335 datain=16'hB614;
#500 $stop;
end

initial begin
data = 16'hA569;
#15 repeat (16) begin
    #20 MISO = data[0];
    data = data >>1;
    end
 #15
data = 16'h2563;
   repeat (16) begin
        #20 MISO = data[0];
        data = data >>1;
        end

data = 16'h6A61;
      repeat (16) begin
           #20 MISO = data[0];
           data = data >>1;
           end
  #15
```

```
data = 16'h7635;
        repeat (16) begin
            #20 MISO = data[0];
            data = data >>1;
            end
  #15 ;
end

endmodule
```