VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

Serial Peripheral Interface Design

A PROJECT REPORT

submitted in partial fulfillment of the requirements

for

VLSI Design Internship

By

AYUSH KUMAR (22BEE1292)

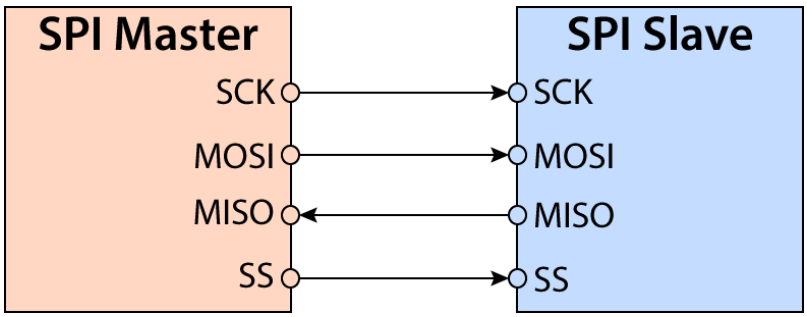
INTRODUCTION

This project report presents a comprehensive account of the design and implementation of the SPI (Serial Peripheral Interface) Master Core using Verilog. The primary objective was to acquire practical experience in digital circuit design and implementation with Verilog, along with an in-depth understanding of the SPI protocol and its applications.

The report begins with an overview of the project, outlining its objectives and scope. This is followed by an exploration of the SPI protocol, detailing its features, advantages, and disadvantages compared to other serial communication protocols. It also includes a comparison with other synchronous serial protocols such as I2C and Microwire/Plus.

The design and implementation of the SPI Master Core in Verilog are explained in detail, covering the Verilog code, IO ports, registers, and external connections used in the design. The testing process for verifying the functionality of the SPI Master Core and the results obtained are also discussed.

ABOUT SPI

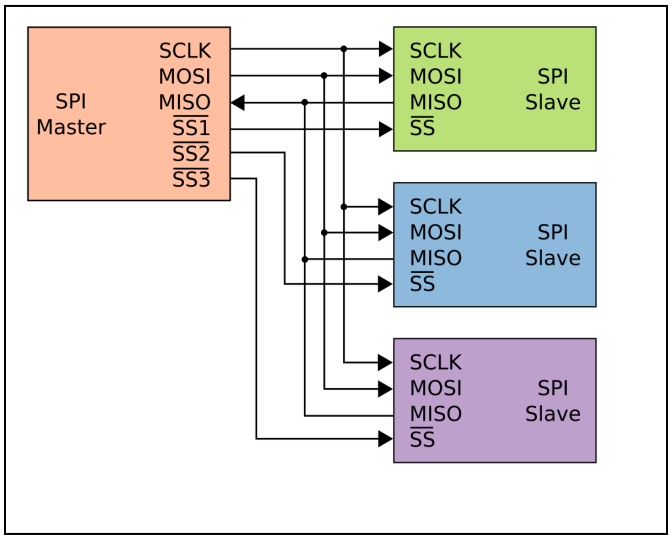


Serial Peripheral Interface (SPI) is a master – slave type protocol that provides a simple and low-cost interface between a microcontroller and its peripherals ICs such as sensors, ADC's, DAC's, shift register, SRAM and others. In SPI protocol, there can be only one master but many slave devices.

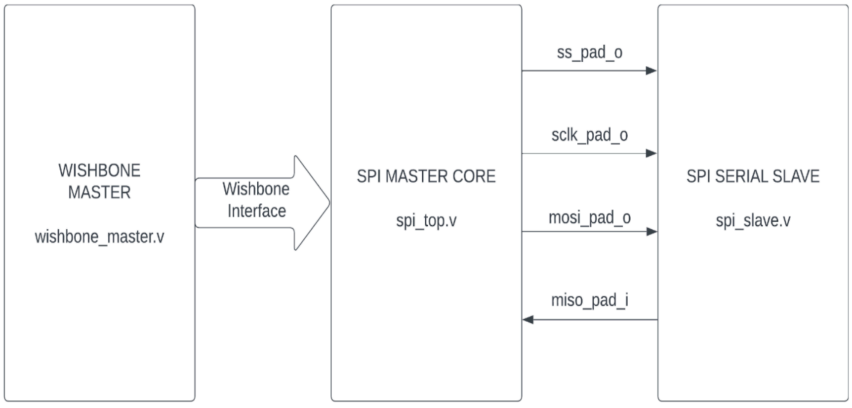
The SPI bus consists of 4 signals or pins. They are

- Master – Out / Slave – In (MOSI)
- Master – In / Slave – Out (MISO)
- Serial Clock (SCLK)
- Slave Select (SS)

Data is exchanged simultaneously between devices in a serial manner (shifted out serially and shifted in serially). Serial clock line synchronizes shifting and sampling of information on two serial data lines. A slave select line allows individual selection of slave SPI devices.

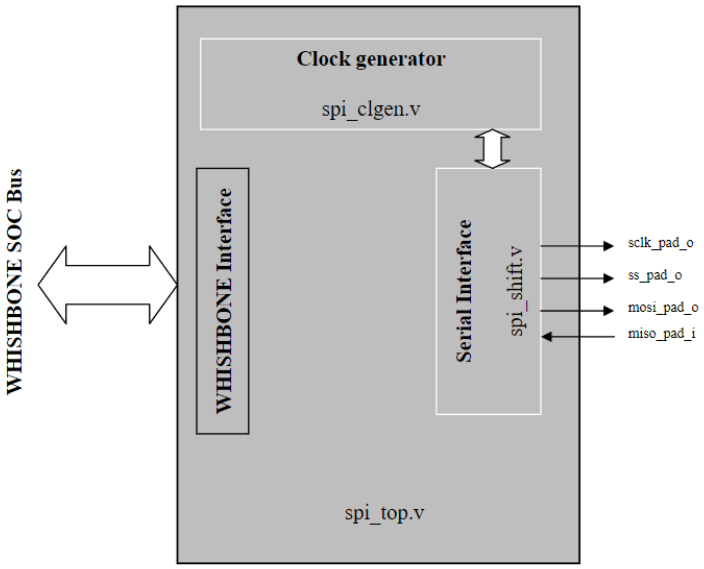


ARCHITECTURE



SPI Master Core

The SPI Master core consists of three parts shown in the following figure:



Features of SPI Master Core:

- Full duplex synchronous serial data transfer
- Variable length of transfer word up to 128 bits
- MSB or LSB first data transfer
- Rx and Tx on both rising or falling edge of serial clock independently
- 8 slave select lines
- Fully static synchronous design with one clock domain
- Technology independent Verilog
- Fully synthesizable

IO PORTS

WISHBONE INTERFACE SIGNALS

Port	Width	Direction	Description
wb_clk_i	1	Input	Master clock
wb_rst_i	1	Input	Synchronous reset, active high
wb_adr_i	5	Input	Lower address bits

wb_dat_i	32	Input	Data towards the core
wb_dat_o	32	Output	Data from the core
wb_sel_i	4	Input	Byte select signals
wb_we_i	1	Input	Write enable input
wb_stb_i	1	Input	Strobe signal/Core select input
wb_cyc_i	1	Input	Valid bus cycle input
wb_ack_o	1	Output	Bus cycle acknowledge output
wb_err_o	1	Output	Bus cycle error output
wb_int_o	1	Output	Interrupt signal output

SPI EXTERNAL CONNECTIONS

Port	Width	Direction	Description
/ss_pad_o	8	Output	Slave select output signals
sclk_pad_o	1	Output	Serial clock output
mosi_pad_o	1	Output	Master out slave in data signal output
miso_pad_i	1	Input	Master in slave out data signal input

REGISTERS

CORE REGISTERS LISTS

Name	Address	Width	Access	Description
Rx0	0x00	32	R	Data receive register 0
Rx1	0x04	32	R	Data receive register 1
Rx2	0x08	32	R	Data receive register 2
Rx3	0x0c	32	R	Data receive register 3
Tx0	0x00	32	R/W	Data transmit register 0

Tx1	0x04	32	R/W	Data transmit register 1
Tx2	0x08	32	R/W	Data transmit register 2
Tx3	0x0c	32	R/W	Data transmit register 3
CTRL	0x10	32	R/W	Control and status register
DIVIDER	0x14	32	R/W	Clock divider register
SS	0x18	32	R/W	Slave select register

DATA RECEIVE REGISTERS [RxX]

Bit #	31:0
Access	R
Name	Rx

DATA TRANSMIT REGISTERS [TxX]

Bit #	31:0
Access	R/W
Name	Tx

CONTROL and STATUS REGISTER [CTRL]

Bit #	31:14	13	12	11	10	9	8	7	6:0
Access	R	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W
Name	Reserved	ASS	IE	LSB	Tx_NEG	Rx_NEG	GO_BSY	Reserved	CHAR_LEN

DIVIDER REGISTER [DIVIDER]

Bit #	31:16	15:0
Access	R	R/W
Name	Reserved	DIVIDER

SLAVE SELECT REGISTER [SS]

Bit #	31:8	7:0
Access	R	R/W
Name	Reserved	SS

DESIGN BLOCKS and WAVEFORM

SPI_DEFINES.V

The Verilog code "*spi_defines.v*" defines various parameters and constants for configuring an SPI (Serial Peripheral Interface) module. Here is a summary of the key aspects covered in this code:

Clock Generator Divider Length: The code provides options for setting the SPI divider length, which determines the clock frequency for SPI communication. You can choose between different divider lengths, such as 8, 16, 24, or 32.

Max Number Of Bits That Can Be Sent/Received At Once: It defines the maximum number of bits that can be sent or received in a single SPI transfer. Options are available for different character lengths, including 8, 16, 24, 32, 64, or 128 bits.

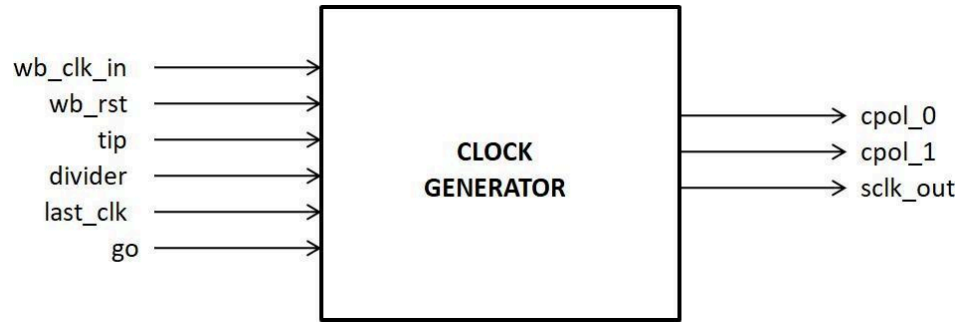
Number of Devices Select Signals: The code defines the number of devices select signals (SS) that can be used in the SPI interface. You can choose from options like 8, 16, 24, or 32 device select signals.

Register Offsets: The code provides register offsets for various SPI registers, such as RX (Receive), TX (Transmit), Control (CTRL), Divide (for clock division settings), and SS (Device Select).

No. of Bits in CTRL Register: It specifies the number of bits in the Control (CTRL) register, which is set to 14 bits.

Control Register Bit Position: The code defines the bit positions within the Control (CTRL) register for various control signals and settings, such as ASS (Auto Slave Select), IE (Interrupt Enable), LSB (Least Significant Bit First), TX/RX edge selection, GO (Start Transfer), and character length configuration.

CLOCK GENERATOR



SPI_CLGEN.V

Module Declaration: Defines the module spi_clgen with input and output ports.

- Inputs:
  - wb_clk_in: Input clock signal.
  - wb_rst: Reset signal.
  - go: Signal indicating the start of the transfer.
  - tip: Signal indicating whether a transfer is in progress.
  - last_clk: Signal indicating the last clock edge of the transfer.
  - divider: Input value determining the clock divider for generating the SPI clock.
- Outputs:
  - sclk_out: Output SPI clock signal.
  - cpol_0: Output signal indicating the pulse marking the positive edge of the SPI clock.
  - cpol_1: Output signal indicating the pulse marking the negative edge of the SPI clock.

The clock generator block generates two clock signals: rx_clk and tx_clk. These clock signals are used to control the receive and transmit operations of the SPI shift register. The generation of these clocks is based on the input signals rx_negedge, tx_negedge, last, sclk, cpol_0, and cpol_1.

The value given to the divider field is the frequency divider of the system clock wb_clk_i to generate the serial clock on the output sclk_pad_o. The desired frequency is obtained according to the following equation:

$$f_{sclk} = \frac{f_{wb\_clk}}{(DIVIDER+1)*2}$$

The rx_clk signal is generated by combining the rx_negedge input signal and the logical OR of last and sclk. It is used as the receive clock enable. The tx_clk signal is generated by

combining the tx_negedge input signal and the logical AND of last and the selected clock polarity (cpol_0 or cpol_1). It is used as the transmit clock enable.

The clock generator block ensures that the clock signals are generated appropriately based on the specified clock polarities and the transfer status (last). This enables proper synchronization and timing control for the SPI shift register module during data transmission and reception.

CODE:

```
`include "spi_defines.v"

module spi_clgen (wb_clk_in,  wb_rst,tip, go, last_clk, divider, sclk_out, cpol_0, cpol_1);

 input                   wb_clk_in,  wb_rst,  tip,   go,  last_clk,
 input   [`SPI_DIVIDER_LEN-1:0] divider;
 output                   sclk_out,  cpol_0, cpol_1;

 reg                     sclk_out,cpol_0,cpol_1;

 reg [`SPI_DIVIDER_LEN-1:0]    cnt;

  always@(posedge wb_clk_in or posedge wb_rst)
 begin
  if(wb_rst)
    begin
     cnt <= {{`SPI_DIVIDER_LEN{1'b0}},1'b1};
    end
```

```
  else if(tip)
    begin
     if(cnt == (divider + 1))
       begin
         cnt <= {{`SPI_DIVIDER_LEN{1'b0}},1'b1};
       end
      else
       begin
         cnt <= cnt + 1;
       end
    end
  else if(cnt == 0)
    begin
     cnt <= {{`SPI_DIVIDER_LEN{1'b0}},1'b1};
    end
 end

 always@(posedge wb_clk_in or posedge wb_rst)
 begin
  if(wb_rst)
    begin
     sclk_out <= 1'b0;
    end
  else if(tip)
    begin
     if(cnt == (divider + 1))
       begin
         if(!last_clk || sclk_out)
           sclk_out <= ~sclk_out;
       end
    end
 end
endmodule
```

SPI_CLGEN_TB.V

The Verilog module "*spi_clgen_tb*" is a testbench for the "*spi_clgen*" module, which is a clock generator for an SPI (Serial Peripheral Interface) interface. This testbench defines the test environment and simulates the behavior of the clock generator. Here's a brief conclusion of this testbench code:

Signal Declarations: The testbench defines various signals to simulate the clock generator module, including "*wb_clk_in*", "*wb_rst*" for reset, "*go*" for transfer initiation, "*tip*" for transfer-in-progress indication, "*last_clk*" for tracking the last clock cycle, and "*divider*" for configuring the clock divider. It also declares "*sclk_out*", "*cpol_0*", and "*cpol_1*" as wires to observe the output signals from the clock generator.

Instantiation of DUT: The testbench instantiates the Device Under Test (DUT), which is the "*spi_clgen*" module, and connects the input and output signals accordingly.

Clock Generation: The testbench generates a clock signal "*wb_clk_in*" using an "*always*" block. It toggles the clock every 5-time units, simulating the system clock.

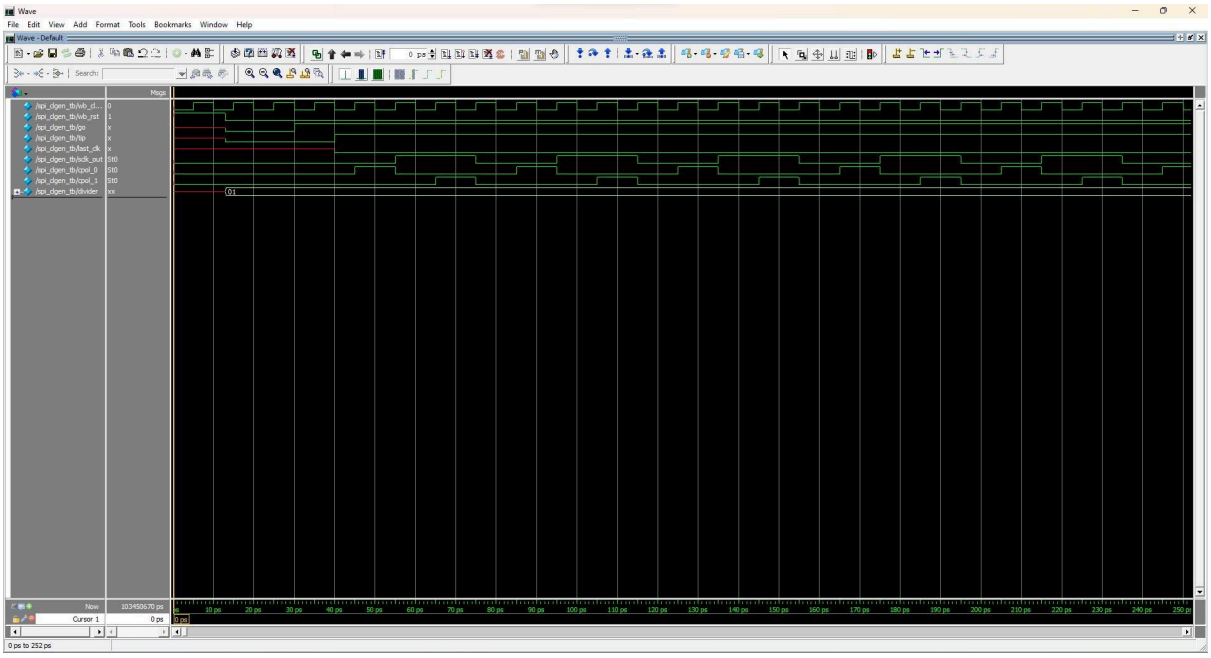
Signal Initialization: In the "*initial block*", the testbench initializes various signals to simulate different conditions. It initially sets the reset signal, "*wb_rst*", high and then lowers it after a delay. It sets the "*divider*", "*tip*", and "*go*" signals to specific values to simulate a transfer initiation with a certain divider value. It also sets "*tip*" and "*last_clk*" to observe the behavior of the clock generator under these conditions.

CODE:

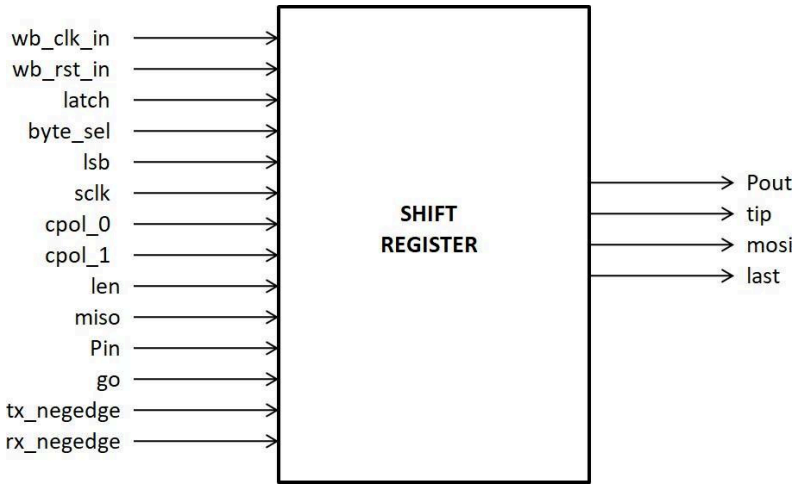
```
`include "spi_defines.v"

module spi_clgen_tb;

 // Define the signals for the clock generator module
 reg wb_clk_in;
 reg wb_rst;
 reg go;
 reg tip;
 reg last_clk;
 wire sclk_out;
 wire cpol_0;
 wire cpol_1;
 reg [`SPI_DIVIDER_LEN-1:0] divider;

 // Instantiate the Device Under Test (DUT)
 spi_clgen dut (
  .wb_clk_in(wb_clk_in),
  .wb_rst(wb_rst),
  .go(go),
  .tip(tip),
  .last_clk(last_clk),
  .divider(divider),
  .sclk_out(sclk_out),
  .cpol_0(cpol_0),
```

```
  .cpol_1(cpol_1)
 );

 // Clock generation
 always begin
  #5 wb_clk_in = ~wb_clk_in; // Toggle the clock every 5 time units
 end

 // Initialize signals
 initial begin
  wb_clk_in <= 0;
  wb_rst <= 1;
  #13;
  wb_rst <= 0;
  divider <= 1;
  tip <= 0;
  go <= 0;
  #17;
  go <= 1;
  #10;
  tip <= 1;
  last_clk <= 0;
 end

 initial begin
  #200;
 end

endmodule
```

CLOCK GENERATOR WAVEFORM



SHIFT REGISTER



SPI_SHIFT_REG.V

Module Declaration: Defines the module spi_shift_register with input and output ports.

- Input:
  - rx_negedge: Input signal, indicates the negative edge of the receive clock.

    - tx_negedge: Input signal, indicates the negative edge of the transmit clock.
    - byte_sel: 4-bit input signal used for byte selection.
    - latch: Input signal used for latch control.
    - len: 32-bit input signal representing the length of the character.
    - p_in: 32-bit input signal for the incoming data.
    - wb_clk_in: Input clock signal (system clock).
    - wb_rst: Input reset signal.
    - go: Input signal to initiate the transfer.
    - miso: Input signal for the master input slave output.
    - lsb: Input signal indicating the least significant bit.
    - cpol_0: Input signal for the pulse marking the positive edge of the sclk_out.
    - cpol_1: Input signal for the pulse marking the negative edge of the sclk_out.
- Output:
    - p_out: Output signal representing the shifted data from the shift register. It is SPI_MAX_CHAR bits wide.
    - last: Output signal indicating whether the last character is being transmitted or received.
    - mosi: Output signal representing the serial output data from the shift register.
    - tip: Output signal indicating if a transfer is in progress or not.
    - rx_clk: Output signal representing the receive clock enable.
    - tx_clk: Output signal representing the transmit clock enable.
- Internal Registers:
    - char_count: Register used for counting the number of bits in a character.
    - master_data: Register representing the shift register holding the data.
    - tx_bit_pos: Register indicating the next bit position for transmission.
    - rx_bit_pos: Register indicating the next bit position for reception.

The shift register module takes various inputs such as clock signals, data selection, data input, and control signals. The module includes internal registers for storing data, bit positions for data shifting, and a counter for character bit tracking.

It calculates the transfer in progress and the last character indicator. It also calculates the serial output based on the clock signals and the current bit position.

The module handles the calculation of the bit positions for both transmission and reception based on the configuration. The output is the shifted data from the shift register. The module also supports data latching.

CODE :

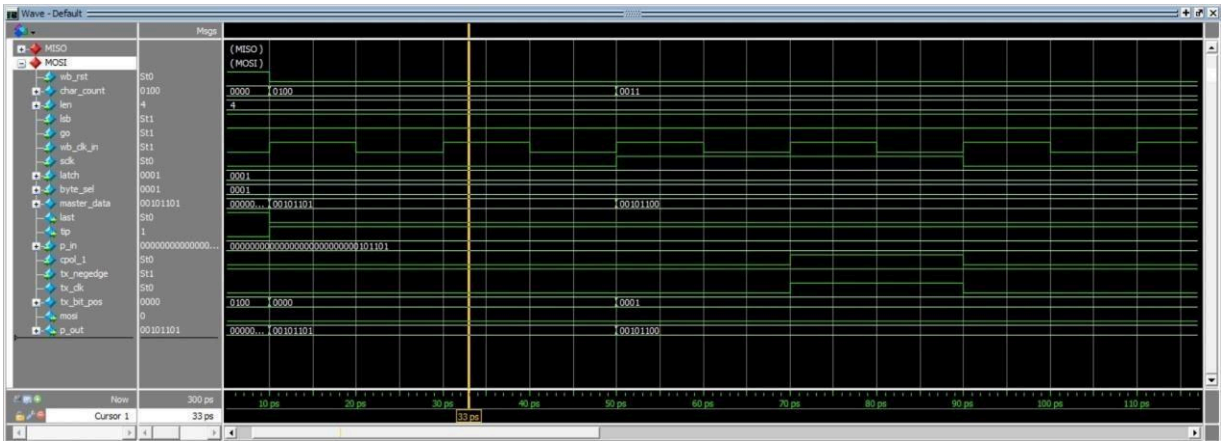
```
`include "spi_defines.v"

module spi_shift_reg(rx_negedge, tx_negedge,
byte_sel,latch,len,p_in,wb_clk_in,wb_rst,go,miso,lsb,sclk,cpol_0,cpol_1,p_out,last,mosi,tip);

   input   rx_negedge,tx_negedge,wb_clk_in,wb_rst,go, miso,lsb,sclk,cpol_0, cpol_1;

   input [3:0] byte_sel, latch;
   input [`SPI_CHAR_LEN_BITS-1:0] len;
   input [31:0] p_in;

   output [`SPI_MAX_CHAR-1:0] p_out;
   output reg tip, mosi;
   output last;

   reg [`SPI_CHAR_LEN_BITS:0] char_count;
   reg [`SPI_MAX_CHAR-1:0] master_data;    // shift register
   reg [`SPI_CHAR_LEN_BITS:0] tx_bit_pos;  // next bit position
   reg [`SPI_CHAR_LEN_BITS:0] rx_bit_pos;  // next bit position
   wire rx_clk;                        // rx clock enable
   wire tx_clk;                        // tx clock enable

   // Character bit counter
   always @(posedge wb_clk_in or posedge wb_rst) begin
      if (wb_rst) begin
         char_count <= 0;
      end else begin
         if (tip) begin
            if (cpol_0) begin
               char_count <= char_count - 1;
            end
         end else begin
            char_count <= {1'b0, len}; // This stores the character bits other than 128 bits
         end
      end
   end

   // Calculate transfer in progress
   always @(posedge wb_clk_in or posedge wb_rst) begin
      if (wb_rst) begin
         tip <= 0;
```

```
    end else begin
      if (go && ~tip) begin
        tip <= 1;
      end else if (last && tip && cpol_0) begin
        tip <= 0;
      end
    end
  end

  // Calculate last
  assign last = ~(|char_count);

  // Calculate the serial out
  always @(posedge wb_clk_in or posedge wb_rst) begin
    if (wb_rst) begin
      mosi <= 0;
    end else begin
      if (tx_clk) begin
        mosi <= master_data[tx_bit_pos[`SPI_CHAR_LEN_BITS-1:0]];
      end
    end
  end

  // Calculate tx_clk, rx_clk
  assign tx_clk = ((tx_negedge) ? cpol_1 : cpol_0) && !last;
  assign rx_clk = ((rx_negedge) ? cpol_1 : cpol_0) && (!last || sclk);

  // Calculate TX_BIT Position
  always @(lsb, len, char_count) begin
    if (lsb) begin
      tx_bit_pos = ({~{|len}, len} - char_count);
    end else begin
      tx_bit_pos = char_count - 1;
    end
  end1

  // Calculate RX_BIT Position based on rx_negedge as miso depends on rx_clk
  always @(lsb, len, rx_negedge, char_count) begin
    if (lsb) begin
      if (rx_negedge) begin
        rx_bit_pos = {~(|len), len} - (char_count + 1);
      end else begin
        rx_bit_pos = {~(|len), len} - char_count;
      end
    end else begin
      if (rx_negedge) begin
        rx_bit_pos = char_count;
      end else begin
        rx_bit_pos = char_count - 1;
```

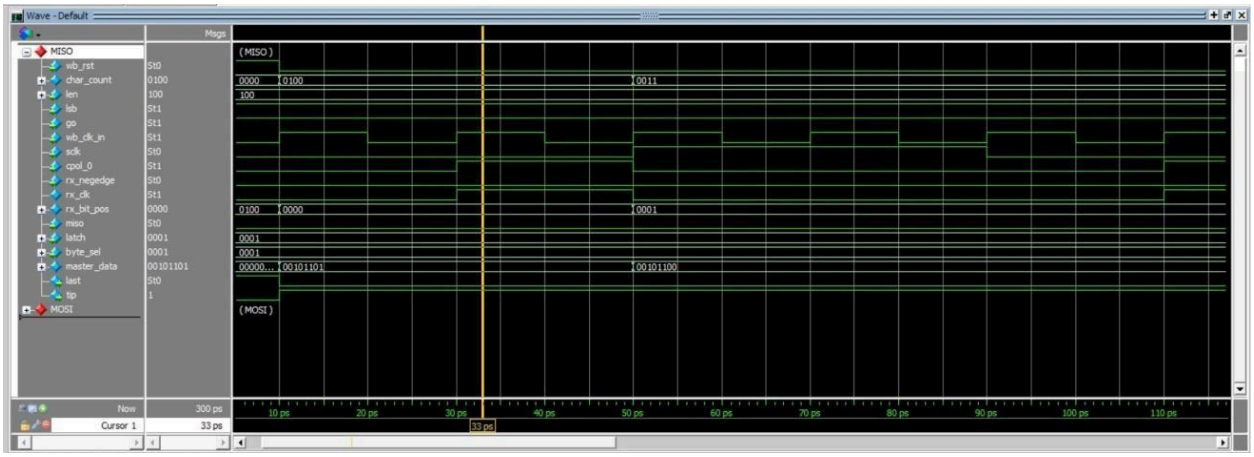
```
        end
      end
    end

    // Calculate p_out
    assign p_out = master_data;

    // Latching of data
    always @(posedge wb_clk_in or posedge wb_rst) begin
      if (wb_rst) begin
        master_data <= {`SPI_MAX_CHAR{1'b0}};
      end else if (latch[0] && !tip) begin
        if (byte_sel[0]) begin
          master_data[7:0] <= p_in[7:0];
        end
      end else begin
        if (rx_clk) begin
          master_data[rx_bit_pos[`SPI_CHAR_LEN_BITS-1:0]] <= miso;
        end
      end
    end
endmodule
```

SHIFT REGISTER WAVEFORM

MOSI



MISO



SPI_SHIFT_REG_TB.V

The Verilog module "*spi_shift_reg_tb*" is a testbench designed to test the functionality of the "*spi_shift_reg*" module, which appears to be an implementation of a shift register for an SPI (Serial Peripheral Interface) interface. Here's a brief conclusion of this testbench code:

Signal Declarations: The testbench defines numerous signals to control and monitor the behavior of the DUT ("*spi_shift_reg*"), including clock edges ("*rx_negedge*" and "*tx_negedge*"), system clock "*wb_clk_in*", reset signal "*wb_rst*", transfer initiation signal "*go*", "*miso*" for input data, "*lsb*" for least significant bit first, "*sclk*" for the serial clock, and "*cpol_0*" and "*cpol_1*" for clock polarity configuration. It also defines signals for byte selection, latch control, character length "*len*", and parallel input data "*p_in*". Output signals include "*p_out*", "*tip*", "*mosi*", and "*last*".

Parameters: The testbench defines parameters such as "*T*" for the clock period and "*divider_value*" to configure the divider length, which influences the serial clock ("*sclk*") generation.

DUT Instantiation: The testbench instantiates the Device Under Test (DUT), which is the "*spi_shift_reg*" module, and connects the input and output signals accordingly.

Clock Generation: The testbench generates the system clock wb_clk_in and the serial clock "*sclk*". It toggles these clocks based on the configured time intervals and the "*divider_value*". The clock generation ensures that the DUT operates in a timed environment.

Tasks for Edge Configuration: The testbench defines tasks ("*t1*" and "*t2*") to set the "*rx_negedge*" and "*tx_negedge*" signals, allowing for the testing of different clock edge configurations.

Initialization: The "*initialize*" task is used to set various input signals to their initial values before testing. It sets parameters such as character length, data values, and control signals to specific initial states.

Simulation Scenario: The testbench defines a specific simulation scenario where it configures input signals, initiates a reset, sets various parameters, and monitors the behavior of the DUT. It models data transmission ("*miso*"), clock polarity configuration, and other aspects of the SPI interface. The scenario includes delays to simulate data transfer and clock behavior.

CODE:

```
`include "spi_defines.v"

`timescale 1us/1ns

module spi_shift_reg_tb;

 reg   rx_negedge,
       tx_negedge,
       wb_clk_in,
       wb_rst,
       go,
       miso,
       lsb,
       sclk,
       cpol_0,
       cpol_1;

 reg [3:0] byte_sel,latch ;
 reg [`SPI_CHAR_LEN_BITS-1:0] len;
 reg [`SPI_MAX_CHAR-1:0] p_in;

 wire [`SPI_MAX_CHAR-1:0]p_out;
 wire tip,mosi;
 wire last;

 parameter T = 10;
 parameter [`SPI_DIVIDER_LEN-1:0] divider_value = 4'b0010;

 // Instantiate the DUT
 spi_shift_reg DUT (rx_negedge,
                    tx_negedge,
                    byte_sel,
                    latch,
                    len,
                    p_in,
                    wb_clk_in,
                    wb_rst,
                    go,
                    miso,
                    lsb,
                    sclk,
```

```
            cpol_0,
            cpol_1,
            p_out,
            last,
            mosi,
            tip);

initial
begin
  wb_clk_in = 1'b0;
  forever
    #(T/2) wb_clk_in = ~wb_clk_in;
end

initial
begin
  sclk = 1'b0;
  forever
    begin
      repeat(divider_value + 1)
        @(posedge wb_clk_in);
        sclk = ~sclk;
    end
end

task rst();
  begin
    wb_rst = 1'b1;
    #13;
    wb_rst = 1'b0;
  end
endtask

initial
begin
  cpol_1 = 1'b0;
  forever
    begin
      repeat(divider_value*2 + 1)
        @(posedge wb_clk_in);
      cpol_1 = 1'b1;
      @(posedge wb_clk_in)
      cpol_1 = 1'b0;
        repeat(divider_value*2 + 1)
```

```
        @(posedge wb_clk_in);
      cpol_1 = 1'b1;
      @(posedge wb_clk_in)
      cpol_1 = 1'b0;
    end
end

initial
begin
  cpol_0 = 1'b0;
    repeat(divider_value)
      @(posedge wb_clk_in);
    cpol_0 = 1'b1;
    @(posedge wb_clk_in)
    cpol_0 = 1'b0;
    forever
      begin
        repeat(divider_value*2 + 1)
          @(posedge wb_clk_in);
        cpol_0 = 1'b1;
        @(posedge wb_clk_in)
        cpol_0 = 1'b0;
      end
end

task t1;
  begin
    @(negedge wb_clk_in)
    rx_negedge = 1'b1;
    tx_negedge = 1'b0;
  end
endtask

task t2;
  begin
    @(negedge wb_clk_in)
    rx_negedge = 1'b0;
    tx_negedge = 1'b1;
  end
endtask

task initialize;
  begin
    len = 3'b000;
```

```
    lsb = 1'b0;
    p_in = 32'h0000;
    byte_sel = 4'b0000;
    latch = 4'b0000;
    go = 1'b0;
    miso = 1'b0;
    cpol_1 = 1'b0;
    cpol_0 = 1'b0;
  end
 endtask

 initial
 begin
  initialize;
  rst;
  @(negedge wb_clk_in)
  len = 32'h0004;
  lsb = 1'b1;
  p_in = 32'haa55;
  latch = 4'b0001;
  byte_sel = 4'b0001;
  t1;
  #10;
  go = 1'b1;
  #40;
  miso = 1'b1;
  #20;
  miso = 1'b0;
  #20;
  miso = 1'b1;
  #20;
  miso = 1'b0;
  #20;
  miso = 1'b1;
  #60;
  miso = 1'b0;
  #30;
  #100;
 end

endmodule
```

SPI_TOP.V

It interfaces with a Wishbone bus and provides communication with a master device using the SPI protocol.

- Inputs:
  - wb_clk_in: Clock input for the Wishbone bus.
  - wb_rst_in: Reset input for the Wishbone bus.
  - wb_adr_in: Address input for the Wishbone bus.
  - wb_dat_in: Data input for the Wishbone bus.
  - wb_sel_in: Select input for the Wishbone bus.
  - wb_we_in: Write enable input for the Wishbone bus.
  - wb_stb_in: Strobe input for the Wishbone bus. ❖ wb_cyc_in: Cycle input for the Wishbone bus. ❖ miso: Master Input Slave Output (MISO) signal

- Outputs:
  - wb_dat_o: Data output for the Wishbone bus.
  - wb_ack_out: Acknowledge output for the Wishbone bus.
  - wb_int_o: Interrupt output for the Wishbone bus.
  - sclk_out: Clock output for the SPI interface.
  - mosi: Master Output Slave Input (MOSI) signal.
  - ss_pad_o: Slave select output for the SPI interface.

The code instantiates two sub-modules: "spi_clgen" and "spi_shift_reg".

- "spi_clgen" generates the SPI clock signal based on the input clock and control signals.
- "spi_shift_reg" handles the data shifting and synchronization for SPI communication.

It decodes addresses to read from specific registers and provides output data and acknowledgment signals. It also supports interrupt generation and slave device selection. Overall, it serves as a complete SPI controller for data transfer between a master and multiple slave devices.

CODE:

```
`include "spi_defines.v"

//SPI Master Core

module spi_top(wb_clk_in,
               wb_rst_in,
               wb_adr_in,
               wb_dat_o,
               wb_sel_in,
               wb_we_in,
               wb_stb_in,
               wb_cyc_in,
               wb_ack_out,
               wb_int_o,
               wb_dat_in,
               ss_pad_o,
               sclk_out,
               mosi,
               miso);

input wb_clk_in,
    wb_rst_in,
    wb_we_in,
    wb_stb_in,
    wb_cyc_in,
    miso;

input [4:0]  wb_adr_in;
input [31:0] wb_dat_in;
input [3:0]  wb_sel_in;

output reg [31:0] wb_dat_o;

output wb_ack_out,wb_int_o,sclk_out,mosi;

reg wb_ack_out,wb_int_o;

output [`SPI_SS_NB-1:0] ss_pad_o;

//Internal signals.....................................

wire rx_negedge;                    //miso is sampled on negative edge
```

```
wire tx_negedge;                    //mosi is driven on negative edge
wire [3:0] spi_tx_sel;              //tx_1 register selected
wire [`SPI_CHAR_LEN_BITS-1:0] char_len; //char len
wire go,ie,ass;                     //go
wire lsb;
wire cpol_0,cpol_1,last,tip;
wire [`SPI_MAX_CHAR-1:0] rx;
wire spi_divider_sel,spi_ctrl_sel,spi_ss_sel;
reg  [`SPI_DIVIDER_LEN-1:0] divider;    //Divider register
reg  [31:0] wb_temp_dat;
reg  [`SPI_CTRL_BIT_NB-1:0] ctrl;       //Control and status register
reg  [`SPI_SS_NB-1:0] ss;               //Slave select register

//Instantiate the SPI_CLK_GENERATOR Module
spi_clgen SC(wb_clk_in,
          wb_rst_in,
          go,
          tip,
          last,
            divider,
          sclk_out,
          cpol_0,
          cpol_1);

//Instantiate the SPI shift register
spi_shift_reg SR(rx_negedge,
              tx_negedge,
              wb_sel_in,
              (spi_tx_sel[3:0] & {4{wb_we_in}}),
              char_len,
              wb_dat_in,
              wb_clk_in,
              wb_rst_in,
              go,
              miso,
              lsb,
              sclk_out,
              cpol_0,
              cpol_1,
              rx,
              last,
              mosi,
              tip);
```

```
  //Address decoder
  assign spi_divider_sel = wb_cyc_in & wb_stb_in & (wb_adr_in == (5'b10100));
  assign spi_ctrl_sel    = wb_cyc_in & wb_stb_in & (wb_adr_in == (5'b10000));
  assign spi_ss_sel      = wb_cyc_in & wb_stb_in & (wb_adr_in == (5'b11000));
  assign spi_tx_sel[0]   = wb_cyc_in & wb_stb_in & (wb_adr_in == (5'b00000));
  assign spi_tx_sel[1]   = wb_cyc_in & wb_stb_in & (wb_adr_in == (5'b00100));
  assign spi_tx_sel[2]   = wb_cyc_in & wb_stb_in & (wb_adr_in == (5'b01000));
  assign spi_tx_sel[3]   = wb_cyc_in & wb_stb_in & (wb_adr_in == (5'b01100));

  //Read from registers
  always@(*)
  begin
    case(wb_adr_in)
      `ifdef SPI_MAX_CHAR_128
        `SPI_RX_0 : wb_temp_dat = rx[31:0];
        `SPI_RX_1 : wb_temp_dat = rx[63:32];
        `SPI_RX_2 : wb_temp_dat = rx[95:64];
        `SPI_RX_3 : wb_temp_dat = rx[127:96];
      `else
      `ifdef SPI_MAX_CHAR_64
        `SPI_RX_0 : wb_temp_dat = rx[31:0];
        `SPI_RX_1 : wb_temp_dat = rx[63:32];
        `SPI_RX_2 : wb_temp_dat = 0;
        `SPI_RX_3 : wb_temp_dat = 0;
      `else
        `SPI_RX_0 : wb_temp_dat = rx[`SPI_MAX_CHAR-1:0];
        `SPI_RX_1 : wb_temp_dat = 32'b0;
        `SPI_RX_2 : wb_temp_dat = 32'b0;
        `SPI_RX_3 : wb_temp_dat = 32'b0;
      `endif
    `endif
      `SPI_CTRL    : wb_temp_dat = ctrl;
      `SPI_DIVIDE  : wb_temp_dat = divider;
      `SPI_SS      : wb_temp_dat = ss;
       default     : wb_temp_dat = 32'dx;
    endcase
  end

  //WB data out
  always@(posedge wb_clk_in or posedge wb_rst_in)
  begin
    if(wb_rst_in)
      wb_dat_o <= 32'd0;
    else
```

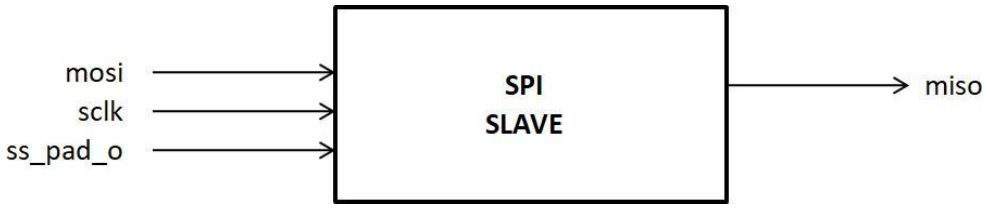
```
      wb_dat_o <= wb_temp_dat;
end

//WB acknowledge
always@(posedge wb_clk_in or posedge wb_rst_in)
begin
  if(wb_rst_in)
    begin
      wb_ack_out <= 0;
    end
  else
    begin
      wb_ack_out <= wb_cyc_in & wb_stb_in & ~wb_ack_out;
    end
end

//Interrupt
always@(posedge wb_clk_in or posedge wb_rst_in)
begin
  if (wb_rst_in)
    wb_int_o <= 1'b0;
  else if (ie && tip && last && cpol_0)
    wb_int_o <= 1'b1;
  else if (wb_ack_out)
    wb_int_o <= 1'b0;
end

//Selecting Slave device from a group of 32 slave devices
assign ss_pad_o = ~((ss & {`SPI_SS_NB{tip & ass}}) | (ss & {`SPI_SS_NB{!ass}}));

//Divider register
always@(posedge wb_clk_in or posedge wb_rst_in)
begin
  if(wb_rst_in)
    begin
      divider <= 0;
    end
  else if(spi_divider_sel && wb_we_in && !tip)
    begin
      `ifdef SPI_DIVIDER_LEN_8
        if(wb_sel_in[0])
          divider <= 1;
      `endif
      `ifdef SPI_DIVIDER_LEN_16
```

```
            if(wb_sel_in[0])
              divider[7:0] <= wb_dat_in[7:0];
            if(wb_sel_in[1])
              divider[15:8] <= wb_dat_in[`SPI_DIVIDER_LEN-1:8];
          `endif
          `ifdef SPI_DIVIDER_LEN_24
            if(wb_sel_in[0])
              divider[7:0] <= wb_dat_in[7:0];
            if(wb_sel_in[1])
              divider[15:8] <= wb_dat_in[15:8];
            if(wb_sel_in[2])
              divider[23:16] <= wb_dat_in[`SPI_DIVIDER_LEN-1:16];
          `endif
          `ifdef SPI_DIVIDER_LEN_32
            if(wb_sel_in[0])
              divider[7:0] <= wb_dat_in[7:0];
            if(wb_sel_in[1])
              divider[15:8] <= wb_dat_in[15:8];
            if(wb_sel_in[2])
              divider[23:16] <= wb_dat_in[23:16];
            if(wb_sel_in[3])
              divider[31:24] <= wb_dat_in[`SPI_DIVIDER_LEN-1:24];
          `endif
        end
    end

    //Control and status register
    always@(posedge wb_clk_in or posedge wb_rst_in)
    begin
      if(wb_rst_in)
        ctrl <= 0;
      else
        begin
          if(spi_ctrl_sel && wb_we_in && !tip)
            begin
              if(wb_sel_in[0])
                ctrl[7:0] <= wb_dat_in[7:0] | {7'd0, ctrl[0]};
              if(wb_sel_in[1])
                ctrl[`SPI_CTRL_BIT_NB-1:8] <= wb_dat_in[`SPI_CTRL_BIT_NB-1:8];
            end
          else if(tip && last && cpol_0)
            ctrl[`SPI_CTRL_GO] <= 1'b0;
        end
    end
```

```
assign rx_negedge = ctrl[`SPI_CTRL_RX_NEGEDGE];
assign tx_negedge = ctrl[`SPI_CTRL_TX_NEGEDGE];
assign lsb = ctrl[`SPI_CTRL_LSB];
assign ie  = ctrl[`SPI_CTRL_IE];
assign ass = ctrl[`SPI_CTRL_ASS];
assign go  = ctrl[`SPI_CTRL_GO];
assign char_len = ctrl[`SPI_CTRL_CHAR_LEN];

//Slave select
always@(posedge wb_clk_in or posedge wb_rst_in)
begin
  if(wb_rst_in)
    begin
      ss <= 0;
    end
  else
    begin
      if(spi_ss_sel && wb_we_in && !tip)
        begin
          `ifdef SPI_SS_NB_8
            if(wb_sel_in[0])
              ss <= wb_dat_in[`SPI_SS_NB-1:0];
          `endif

          `ifdef SPI_SS_NB_16
            if(wb_sel_in[0])
              ss <= wb_dat_in[7:0];
            if(wb_sel_in[1])
              ss <= wb_dat_in[`SPI_SS_NB-1:8];
          `endif

          `ifdef SPI_SS_NB_24
            if(wb_sel_in[0])
              ss <= wb_dat_in[7:0];
            if(wb_sel_in[1])
              ss <= wb_dat_in[15:8];
            if(wb_sel_in[2])
              ss <= wb_dat_in[`SPI_SS_NB-1:16];
          `endif

          `ifdef SPI_SS_NB_32
            if(wb_sel_in[0])
              ss <= wb_dat_in[7:0];
```

```
                if(wb_sel_in[1])
                  ss <= wb_dat_in[15:8];
                if(wb_sel_in[2])
                  ss <= wb_dat_in[23:16];
                if(wb_sel_in[3])
                  ss <= wb_dat_in[`SPI_SS_NB-1:24];
              `endif
            end
        end
    end

    endmodule
```

SPI_SLAVE.V



The Verilog module "*spi_slave*" appears to represent a simple SPI (Serial Peripheral Interface) slave device, which receives and transmits data as part of an SPI communication protocol. Here's a brief conclusion regarding this module's functionality:

Signal Declarations: The module has input signals, including "*sclk*" for the serial clock and "*mosi*" for the master's output data. It also receives the "*ss_pad_o*" signal, which is a combination of slave select signals for selecting this particular slave device. The "*miso*" output signal represents the data transmitted by this slave.

Data Registers: The module includes two data registers, "*temp1*" and "*temp2*", each 128 bits wide, which store data being received from and transmitted to the SPI master. These registers are shifted in synchronization with the serial clock "*sclk*".

Control Signals: The module uses two control signals, "*rx_slave*" and "*tx_slave*", to determine whether the slave is in receive or transmit mode. The "*rx_slave*" signal is set when the slave is receiving data, and "*tx_slave*" is set when it's transmitting data.

Data Shifting: The module shifts the received and transmitted data into the "*temp1*" and "*temp2*" registers, respectively, based on the rising and falling edges of the serial clock "*sclk*". It does this by checking the state of "*rx_slave*" and "*tx_slave*". Data is shifted into the registers one bit at a time, with the most significant bit (MSB) being the first.

MISO Generation: The module generates the MISO (Master In Slave Out) signal, which represents the data that this slave sends back to the master. It does this by extracting the MSB from "*temp1*" when in

receive mode and from "*temp2*" when in transmit mode. These MSB values are combined with a logical OR operation and assigned to the "*miso*" output signal.

CODE:

```
`include "spi_defines.v"

module spi_slave (input sclk,mosi,
                  input [`SPI_SS_NB-1:0]ss_pad_o,
                  output miso);

  reg rx_slave = 1'b0; //Slave receiving from SPI_MASTER
  reg tx_slave = 1'b0; //Slave transmitting to SPI_MASTER

  //Initial value of temp is 0
  reg [127:0]temp1 = 0;
  reg [127:0]temp2 = 0;

  reg miso1 = 1'b0;
  reg miso2 = 1'b1;

  always@(posedge sclk)
  begin
    if ((ss_pad_o != 8'b11111111) && ~rx_slave && tx_slave) //Posedge of the Serial Clock
      begin
        temp1 <= {temp1[126:0],mosi};
      end
  end

  always@(negedge sclk)
  begin
    if ((ss_pad_o != 8'b11111111) && rx_slave && ~tx_slave) //Negedge of the Serial Clock
      begin
        temp2 <= {temp2[126:0],mosi};
      end
  end

  always@(negedge sclk)
  begin
    if (rx_slave && ~tx_slave) //Posedge of the Serial Clock
      begin
        miso1 <= temp1[127];
      end
```

```
  end

  always@(negedge sclk)
 begin
    if (~rx_slave && tx_slave) //Posedge of the Serial Clock
       begin
          miso2 <= temp2[127];
       end
  end

  assign miso = miso1 || miso2;
endmodule
```

WISHBONE_MASTER.V

The Verilog module "*wishbone_master*" appears to be an implementation of a Wishbone master interface. It is designed to initiate Wishbone bus cycles, which include single read and write operations. Below is a concise summary of this module's functionality:

Input and Output Signals: The module takes several input signals, including the system clock ("*clk_in*"), reset ("*rst_in*"), acknowledge (*"ack_in"*), data input (*"dat_in"*), and control signals (*"adr_o"*, *"cyc_o"*, "*stb_o*", "*we_o*", "*dat_o*", and "*sel_o*"). These signals are used to control and monitor the Wishbone bus operation.

Internal Signals: Internal signals like "*adr_temp*", "*sel_temp*", "*dat_temp*", "*we_temp*", "*cyc_temp*" and "*stb_temp*" are declared and used for managing the Wishbone bus cycles and storing temporary values during the operations.

Initialize Task: The module includes an "*initialize*" task, which resets all the internal signals to zero. This task is used to initialize the state of the module, setting it to a known starting condition.

Single Write Task: The "*single_write*" task is responsible for initiating a single write operation on the Wishbone bus. It is used to set the address, data, select lines, and control signals to perform the write operation. The task waits for an acknowledgment ("*ack_in*") and then clears the control signals to end the operation.

Signal Assignments: The module includes several "always" blocks that assign values to the output signals based on the values of internal signals. These assignments include the address ("*adr_o*"), write enable ("*we_o*"), data output (*"dat_o"*), select lines (*"sel_o"*), cycle (*"cyc_o"*), and strobe (*"stb_o"*) signals.

Reset Handling: The module includes reset handling for the "cyc_*o*" and "*stb_o*" signals to ensure they are de-asserted during a reset condition ("*rst_in*").

CODE:

```
module wishbone_master(input clk_in, rst_in,ack_in,err_in,
                       input [31:0]dat_in,
                       output reg [4:0]adr_o,
```

```
            output reg cyc_o,stb_o,we_o,
            output reg [31:0]dat_o,
            output reg [3:0]sel_o);

//Internal Signals
integer adr_temp,sel_temp,dat_temp;
reg we_temp,cyc_temp,stb_temp;

//Initialize task
task initialize;
   begin
      {adr_temp,cyc_temp,stb_temp,we_temp,dat_temp,sel_temp} = 0;
   end
endtask

//Wishbone Bus Cycles Single Read/Write
task single_write;
input [4:0]adr;
input [31:0]dat;
input [3:0]sel;
begin
   @(negedge clk_in);
   adr_temp = adr;
   sel_temp = sel;
   we_temp  = 1;
   dat_temp = dat;
   cyc_temp = 1;
   stb_temp = 1;
   @(negedge clk_in);
   wait(~ack_in)
   @(negedge clk_in);
   adr_temp = 5'dz;
   sel_temp = 4'd0;
   we_temp  = 1'b0;
   dat_temp = 32'dz;
   cyc_temp = 1'b0;
   stb_temp = 1'b0;
end
endtask

always@(posedge clk_in)
begin
   adr_o <= adr_temp;
end
```

```
  always@(posedge clk_in)
  begin
    we_o <= we_temp;
  end

  always@(posedge clk_in)
  begin
    dat_o <= dat_temp;
  end

  always@(posedge clk_in)
  begin
    sel_o <= sel_temp;
  end

  always@(posedge clk_in)
  begin
    if(rst_in)
      cyc_o <= 0;
    else
      cyc_o <= cyc_temp;
  end

  always@(posedge clk_in)
  begin
    if(rst_in)
      stb_o <= 0;
    else
      stb_o <= stb_temp;
  end

endmodule
```

TB.V

The provided Verilog module "tb" is a testbench that assesses the functionality of an SPI communication system that includes a Wishbone interface, a SPI master, and a SPI slave. It utilizes a set of test scenarios to validate different configurations of the system. Here's a brief conclusion regarding the key aspects of this testbench code:

Module Instantiation: The testbench instantiates three main modules: “*wishbone_master*”, “*spi_top*”, and “*spi_slave*”. These modules represent the Wishbone master, the SPI master, and the SPI slave,

respectively. The interconnection of these modules allows for comprehensive testing of the SPI communication system.

Clock Generation: The testbench generates a system clock ("*wb_clk_in*") with a period of "*T*" (as specified by the parameter) and utilizes this clock for simulating the operation of the system.

Reset Task: The "*rst*" task is provided to apply a reset signal to the system by asserting and de-asserting the "*wb_rst_in*" signal. This task ensures that the system starts in a known state before each test scenario.

Test Scenarios: The testbench includes multiple initial blocks, each representing a different test scenario with specific configuration settings for the SPI master. These test scenarios configure various parameters such as clock polarity, clock edge, least significant bit (LSB) order, and character length. After configuration, they allow the simulation to run for a certain duration (through the "*repeat*" and "*wait*" statements), and then the simulation is finished.

Clock Generation and Timing Control: The testbench continuously toggles the clock signal with half the specified period to ensure that the system operates in a timed environment. It also uses wait statements to control the timing and duration of each test scenario.

Completion and Termination: At the end of each test scenario, the "*$finish*" statement is used to terminate the simulation.

CODE:

```
`include "spi_defines.v"

module tb;

  reg wb_clk_in, wb_rst_in;
  wire wb_we_in, wb_stb_in, wb_cyc_in, miso;

  wire [4:0]wb_adr_in;
  wire [31:0]wb_dat_in;
  wire [3:0]wb_sel_in;
  wire [31:0]wb_dat_o;

  wire wb_ack_out, wb_int_o, sclk_out, mosi;

  wire [`SPI_SS_NB-1:0]ss_pad_o;

  parameter T = 20;

  wishbone_master MASTER(wb_clk_in,
                         wb_rst_in,
                         wb_ack_out,
                         wb_err_in,
                         wb_dat_o,
```

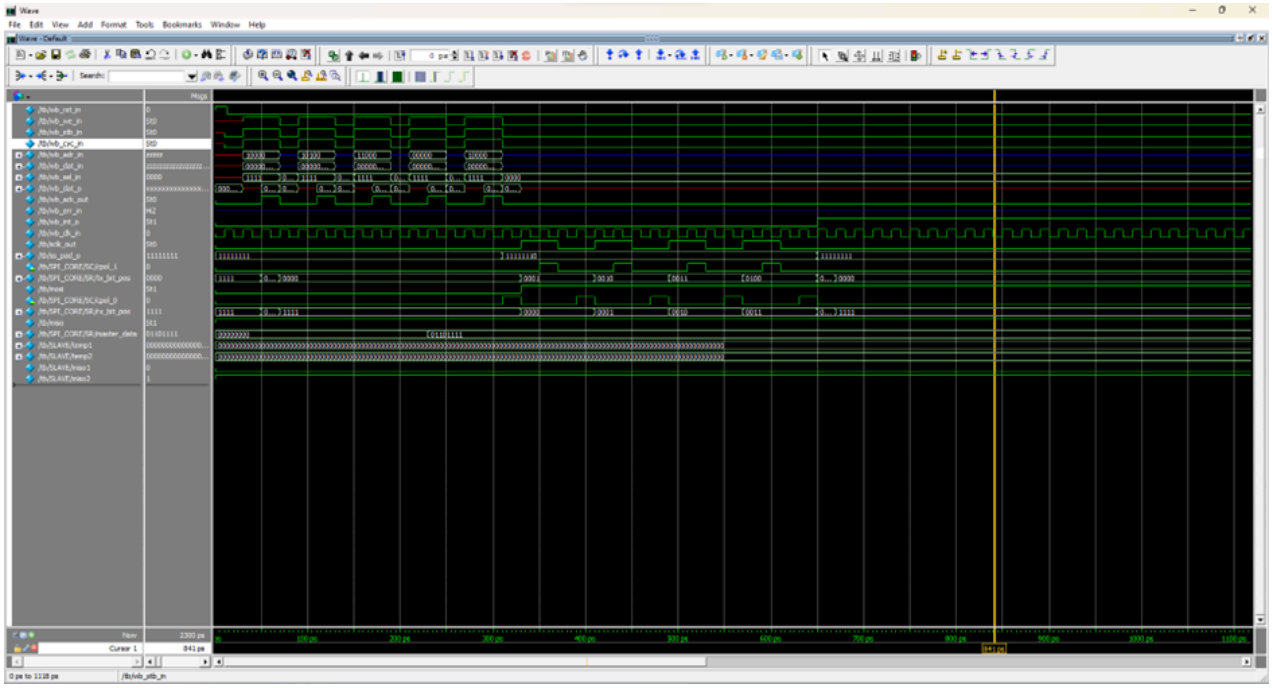
```
                wb_adr_in,
                wb_cyc_in,
                wb_stb_in,
                wb_we_in,
                wb_dat_in,
                wb_sel_in);

  spi_top SPI_CORE(wb_clk_in,
                   wb_rst_in,
                   wb_adr_in,
                   wb_dat_o,
                   wb_sel_in,
                   wb_we_in,
                   wb_stb_in,
                   wb_cyc_in,
                   wb_ack_out,
                   wb_int_o,
                   wb_dat_in,
                   ss_pad_o,
                   sclk_out,
                   mosi,
                   miso);

  spi_slave SLAVE(sclk_out,
                  mosi,
                  ss_pad_o,
                  miso);

  initial
  begin
    wb_clk_in = 1'b0;
    forever
      #(T/2) wb_clk_in = ~wb_clk_in;
  end

  task rst();
    begin
      wb_rst_in = 1'b1;
      #13;
      wb_rst_in = 1'b0;
    end
  endtask

  //tx_neg=1, rx_neg=0, LSB=1, char_len=4
```

```
/*initial
begin
   rst;
   //initialize the WISHBONE output signals
   MASTER.initialize;
   //configure control register with go_busy being low
   MASTER.single_write(5'h10,32'h0000_3c04,4'b1111);
   //configure divider with go_busy being low
   MASTER.single_write(5'h14,32'h0000_0004,4'b1111);
   //configure slave register with go_busy being low
   MASTER.single_write(5'h18,32'h0000_0001,4'b1111);
   //configure tx register with go_busy being low and processor is sending 4 bits
   MASTER.single_write(5'h00,32'h0000_236f,4'b1111);
   //configure control register with go_busy beingg high
   MASTER.single_write(5'h10,32'h0000_3d04,4'b1111);
   repeat(100)
   @(negedge wb_clk_in);
   $finish;
       #10000 $finish;
end*/

//tx_neg=1, rx__neg=0, LSB=0, char_len=4
/*initial
begin
   rst;
   //initialize the WISHBONE output signals
   MASTER.initialize;
   //configure control register with go_busy being low
   MASTER.single_write(5'h10,32'h0000_3404,4'b1111);
   //configure divider with go_busy being low
   MASTER.single_write(5'h14,32'h0000_0004,4'b1111);
   //configure slave register with go_busy being low
   MASTER.single_write(5'h18,32'h0000_0001,4'b1111);
   //configure tx register with go_busy being low and processor is sending 4 bits
   MASTER.single_write(5'h00,32'h0000_236f,4'b1111);
   //configure control register with go_busy beingg high
   MASTER.single_write(5'h10,32'h0000_3504,4'b1111);
   repeat(100)
   @(negedge wb_clk_in);
   $finish;
       #10000 $finish;
end*/

//tx_neg=0, rx__neg=1, LSB=1, char_len=4
```

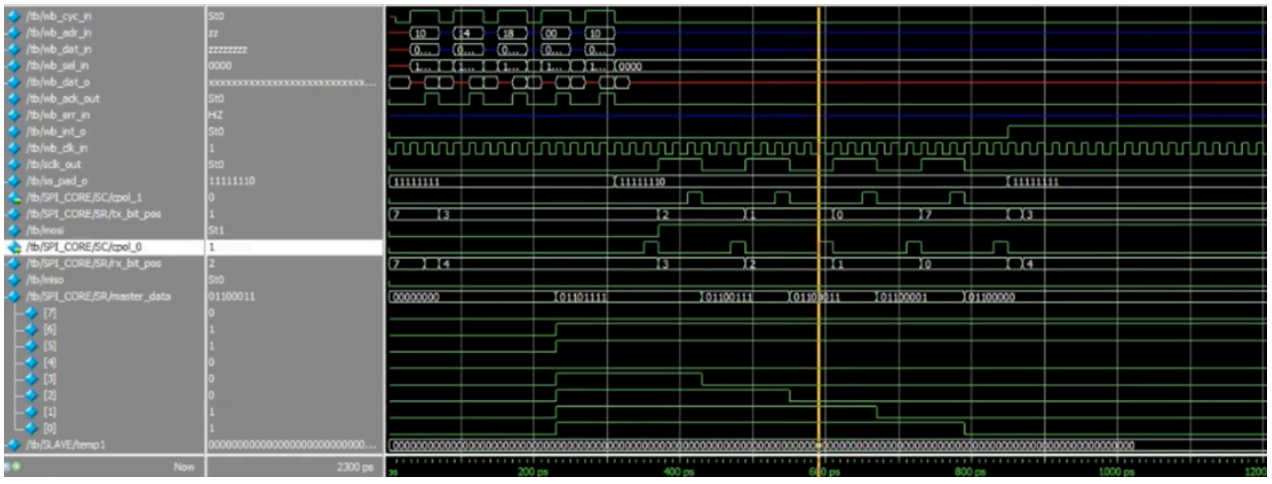
```
  initial
  begin
    rst;
    //initialize the WISHBONE output signals
    MASTER.initialize;
    //configure control register with go_busy being low
    MASTER.single_write(5'h10,32'h0000_3A04,4'b1111);
    //configure divider with go_busy being low
    MASTER.single_write(5'h14,32'h0000_0004,4'b1111);
    //configure slave register with go_busy being low
    MASTER.single_write(5'h18,32'h0000_0001,4'b1111);
    //configure tx register with go_busy being low and processor is sending 4 bits
    MASTER.single_write(5'h00,32'h0000_236f,4'b1111);
    //configure control register with go_busy being high
    MASTER.single_write(5'h10,32'h0000_3B04,4'b1111);
    repeat(100)
    @(negedge wb_clk_in);
    $finish;
        #10000 $finish;
  end


// tx_neg =0,rx_neg = 1 LSB 0, char_len = 4
initial
begin
    rst;
    //initialize the WISHBONE output signals
    MASTER.initialize;
    //configure control register with go_busy being low
    MASTER. single_write (5'h10, 32' h0000_3204,4' b1111);
    // configure divider with go_busy being low
    MASTER. single_write (5'h14, 32' h0000_0002,4'b1111);
    // configure slave register with go_busy being low
    MASTER. single_write (5'h18, 32' h0000_0001,4'b1111);
    //configure tx register with go busy being low and processor is sending 4 bits
    MASTER. single_write (5'h00, 32'h0000_236f, 4 b1111);
    //configure control register with go_busy being high
    MASTER. single_write (5'h10, 32' h0000_3304,4'b1111);
    repeat (100)
@(negedge wb_clk_in);
Sfinish;
        #10000 $finish;
end
endmode
```

TESTCASE MODULE WAVEFORM

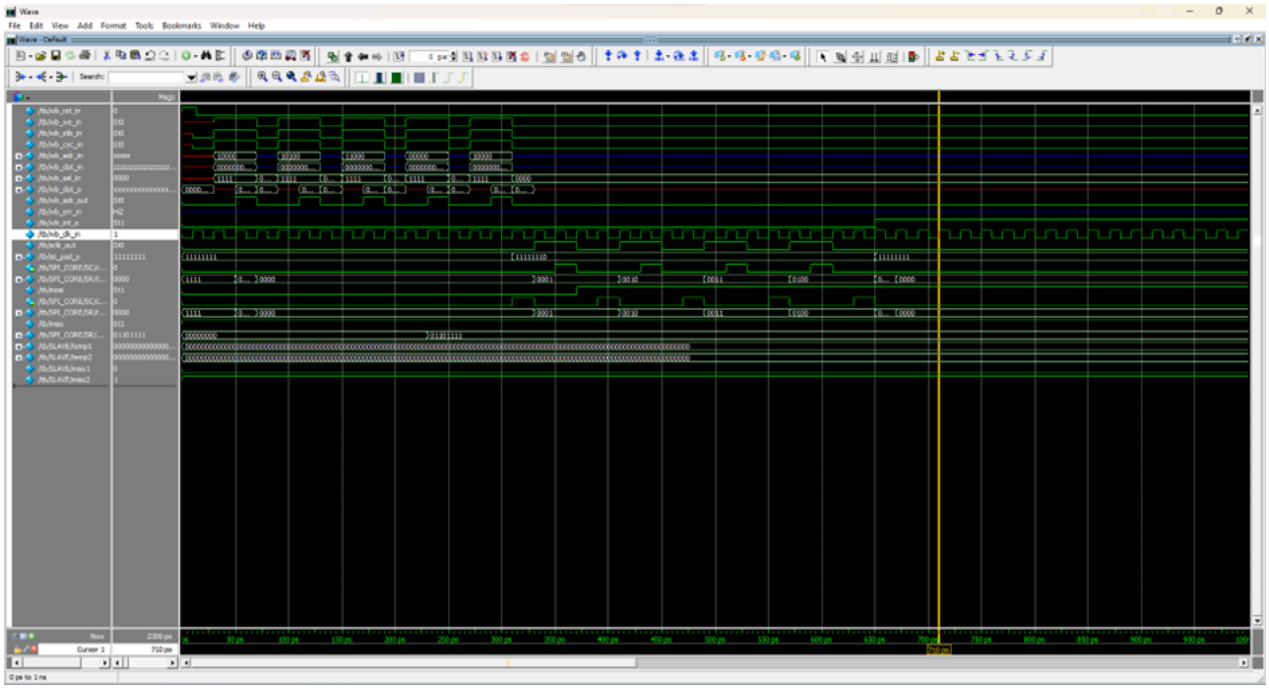
SET 1: TX_NEG = 0, RX_NEG = 1, LSB = 1, CHAR_LEN = 4



SET 2: TX_NEG = 0, RX_NEG = 1, LSB = 0, CHAR_LEN = 4



SET 3: TX_NEG = 1, RX_NEG = 0, LSB = 1, CHAR_LEN = 4



SET 4: TX_NEG = 1, RX_NEG = 0, LSB = 0, CHAR_LEN = 4