NAME:SAKETHVARMA SRINADHARAJU

REGISTER NUMBER:21BEC1416

COLLEGE:Vellore Institute of Technology-CHENNAI

Introduction

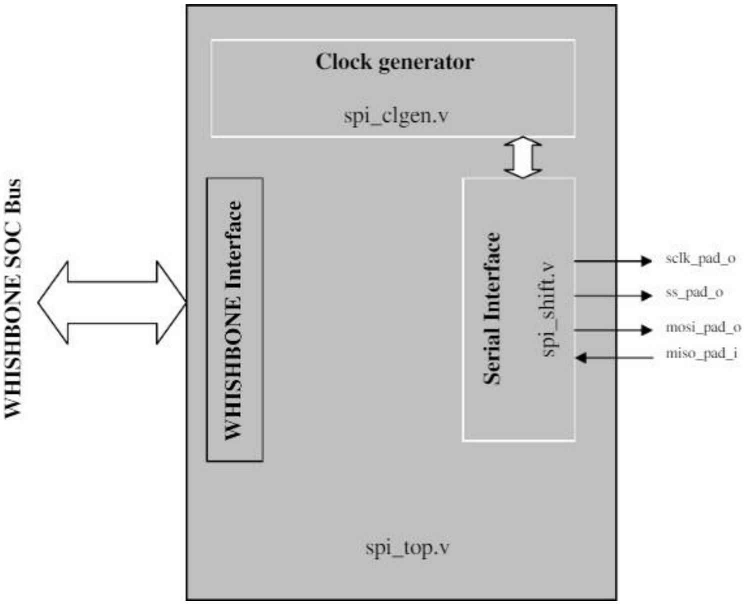
Serial Peripheral Interface (SPI) is a master – slave type protocol that provides a simple and low cost interface between a microcontroller and its peripherals ICs such as sensors, ADC's, DAC's, shift register, SRAM and others. In SPI protocol, there can be only one master but many slave devices.

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

- Master – Out / Slave – In (MOSI)
- Master – In / Slave – Out (MISO)
- Serial Clock (SCLK) and
- 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.

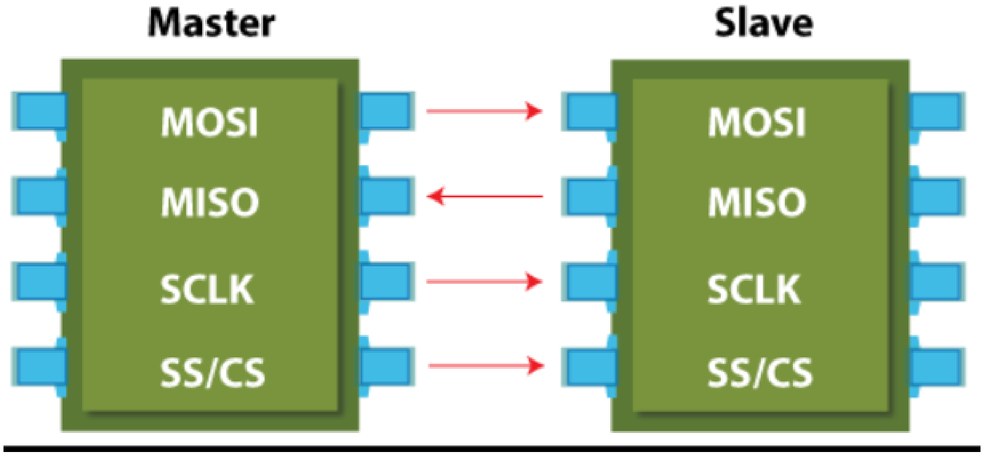
Architecture of SPI Master Core



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

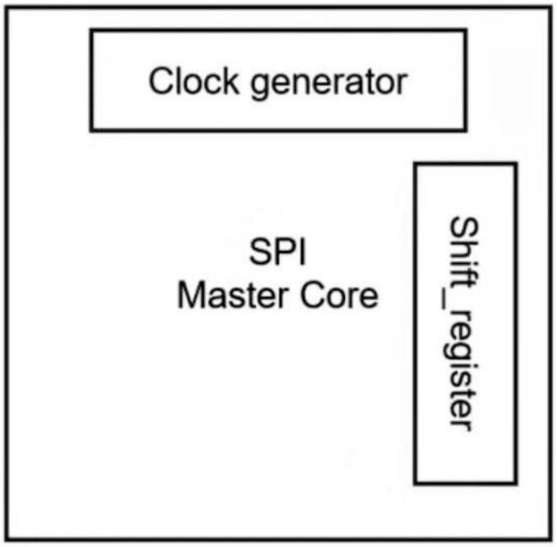
Features of SPI Master Core:

- 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



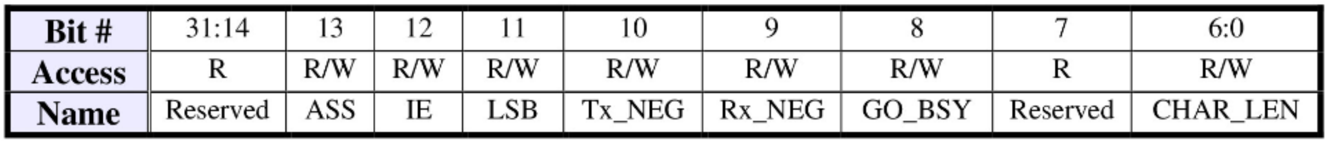
SPI Master Core Interface:

SPI Master Core module is a combination of clock generator and shift register blocks.



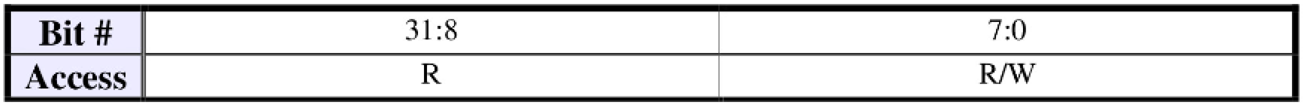
Module “spi.defines.v” is used to initialize all the variable values related to registers (CTRL, SS, DIVIDER and SPI Registers).

Control and Status Register bus is as follows:



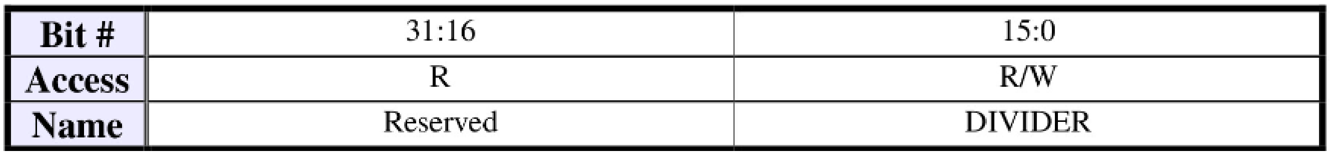
Bit #	31:14	13	12	11	10	9	8	7	6:0
Access	R	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W
Name	Reserved	ASS	IE	LSB	Tx_NEG	Rx_NEG	GO_BSY	Reserved	CHAR_LEN

Slave Select Register bus is as follows:



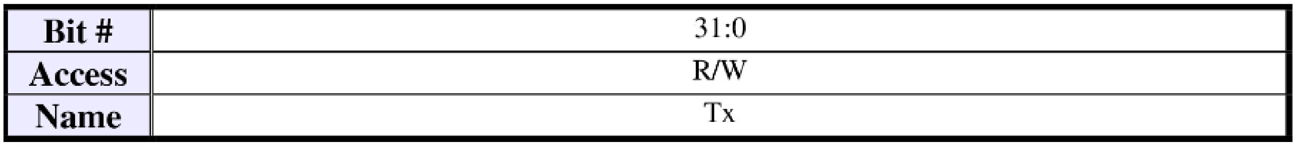
Bit #	31:8	7:0
Access	R	R/W

Divider Register bus is as follows:

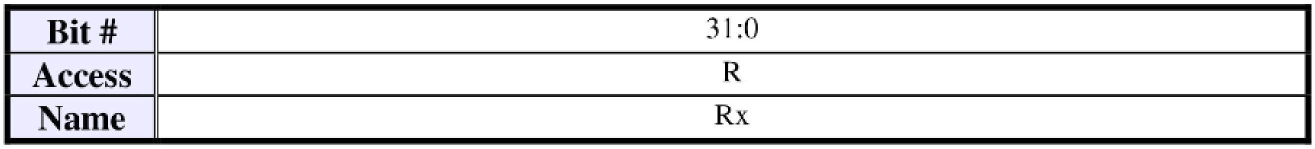


Bit #	31:16	15:0
Access	R	R/W
Name	Reserved	DIVIDER

TxX and RxX Register bus are as follows:

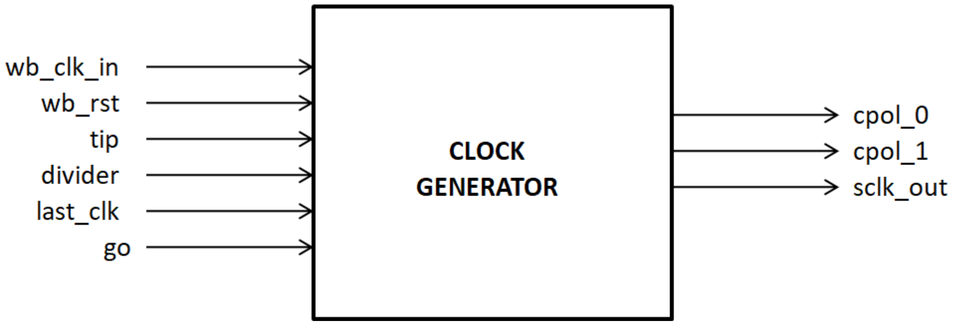


Bit #	31:0
Access	R/W
Name	Tx



Bit #	31:0
Access	R
Name	Rx

Clock Generator Block:



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 ofthe 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 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 basedon 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.

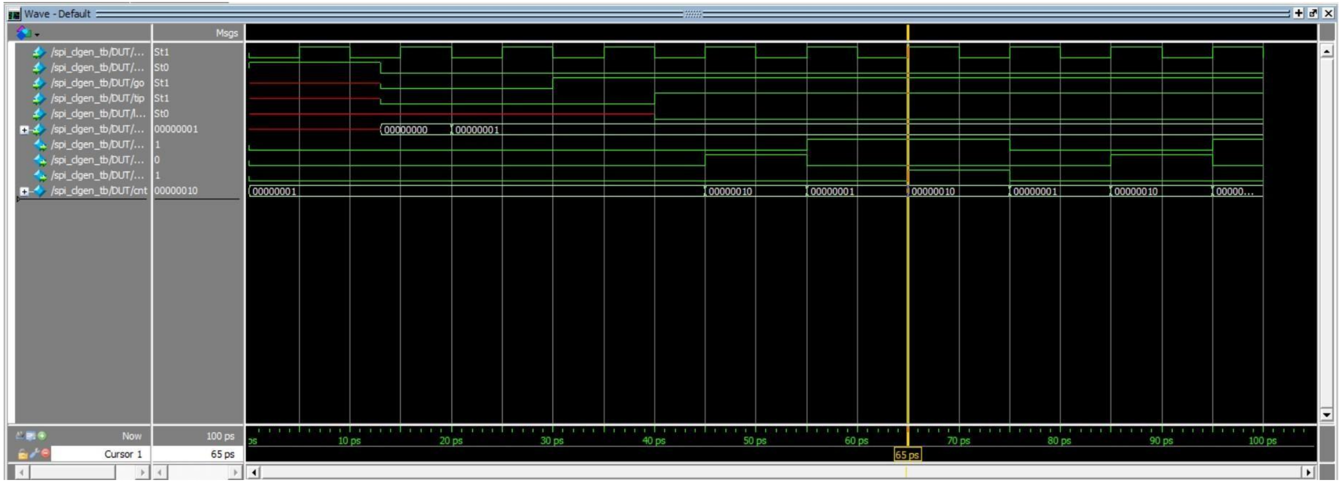
Verilog Code for clock generator block:

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

```
reg [`SPI_DIVIDER_LEN-1:0]cnt;
//Counter counts half period
always @(posedge wb_clk_in or posedge wb_rst)
begin
 if(wb_rst)
  cnt<={{`SPI_DIVIDER_LEN{1'b0}},1'b1};
 else if(tip)
  begin
   if(cnt==(divider+1))
    cnt<={{`SPI_DIVIDER_LEN{1'b0}},1'b1};
   else
    cnt<=cnt+1;
  end
 else if(cnt==0)
  cnt<={{`SPI_DIVIDER_LEN{1'b0}},1'b1};
end
//Generation of Serial clock
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
//Posedge and Negedge detection of sclk
always @(posedge wb_clk_in or posedge wb_rst)
begin
```

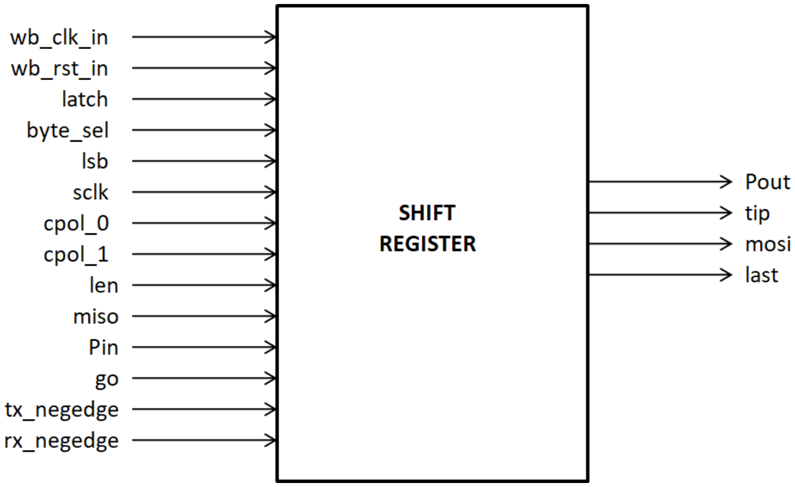
```
      if(wb_rst)
       begin
        cpol_0<=1'b0;
        cpol_1<=1'b0;
       end
      else
       begin
        cpol_0<=1'b0;
        cpol_1<=1'b0;
        if(tip)
         begin
          if(~sclk_out)
           begin
            if(cnt==divider)
             begin
              cpol_0<=1;
             end
           end
         end
         if(tip)
          begin
           if(sclk_out)
            begin
             if(cnt==divider)
              begin
               cpol_1<=1;
              end
            end
          end
       end
     end
endmodule
```

Clock generator Module Waveform



```
Wave - Default
                              Msgs
/spi_clgen_tb/DUT/...         St1
/spi_clgen_tb/DUT/...         St0
/spi_clgen_tb/DUT/go          St1
/spi_clgen_tb/DUT/tip         St1
/spi_clgen_tb/DUT/l...        St0
/spi_clgen_tb/DUT/...         00000001
/spi_clgen_tb/DUT/...         1
/spi_clgen_tb/DUT/...         0
/spi_clgen_tb/DUT/...         1
/spi_clgen_tb/DUT/cnt         00000010
Now                           100 ps
Cursor 1                      65 ps
```

Shift Register Block:



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.

Verilog Code for Shift Registers:

```
module
spi_shift_reg(tx_negedge,rx_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 tx_negedge,rx_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;

reg [`SPI_CHAR_LEN_BITS:0] tx_bit_pos;

reg [`SPI_CHAR_LEN_BITS:0] rx_bit_pos;

wire rx_clk;

wire tx_clk;
```

```
//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

          char_count <= {1'b0,len};

      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
 end
//Calculate RX_BIT Position ......... based on rx_negedge as miso depends on rxclk ..........
always @(lsb, len, rx_negedge, char_count)
 begin
  if(lsb)
   begin
    if(rx_negedge)
     rx_bit_pos = {~{|len},len}-(char_count+1);
    else
     rx_bit_pos = {~{|len},len}-char_count;
   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)
      master_data <= {`SPI_MAX_CHAR{1'b0}};
    //Receiving bits from the parallel line............


    `ifdef SPI_MAX_CHAR_128


    else if(latch[0] && !tip) //TX0 is selected..
      begin
        if(byte_sel[0])
          begin
            master_data[7:0] <= p_in [7:0];
          end
        if(byte_sel[1])
      begin
       master_data[15:8] <= p_in [15:8];
      end
     if(byte_sel[2])
      begin
       master_data[23:16] <= p_in [23:16];
      end
     if(byte_sel[3])
      begin
       master_data[31:24] <= p_in [31:24];
      end
    end

   else if(latch[1] && !tip) //TX1 is selected
    begin
     if(byte_sel[0])
      begin
       master_data[39:32] <= p_in [7:0];
      end
     if(byte_sel[1])
      begin
       master_data[47:40] <= p_in [15:8];
      end
      if(byte_sel[2])
        begin
          master_data[55:48] <= p_in [23:16];
        end
      if(byte_sel[3])
        begin
          master_data[63:56] <= p_in [31:24];
        end
    end

  else if(latch[2] && !tip) //TX2 is selected..
    begin
      if(byte_sel[0])
        begin
          master_data[71:64] <= p_in [7:0];
        end
      if(byte_sel[1])
        begin
          master_data[79:72] <= p_in [15:8];
        end
      if(byte_sel[2])
        begin
          master_data[87:80] <= p_in [23:16];
```

```
   end
  if(byte_sel[3])
   begin
    master_data[95:88] <= p_in [31:24];
   end
 end

else if(latch[3] && !tip) //TX3 is selected..
 begin
  if(byte_sel[0])
   begin
    master_data[103:96] <= p_in[7:0];
   end
  if(byte_sel[1])
   begin
    master_data[111:104] <= p_in[15:8];
   end
  if(byte_sel[2])
   begin
    master_data[119:112] <= p_in[23:16];
   end
  if(byte_sel[3])
   begin
```

```
        master_data[127:120] <= p_in[31:24];
      end
  end

`else

`ifdef SPI_MAX_CHAR_64

else if(latch[0] && !tip) //TX0 is selected..
  begin
    if(byte_sel[0])
      begin
        master_data[7:0] <= p_in[7:0];
      end
    if(byte_sel[1])
      begin
        master_data[15:8] <= p_in[15:8];
      end
    if(byte_sel[2])
      begin
        master_data[23:16] <= p_in[23:16];
      end
    if(byte_sel[3])
```

```
      begin
        master_data[31:24] <= p_in[31:24];
      end
  end

else if(latch[1] && !tip)//TX1 is selected..
  begin
    if(byte_sel[0])
      begin
        master_data[39:32] <= p_in[7:0];
      end
    if(byte_sel[1])
      begin
        master_data[47:40] <= p_in[15:8];
      end
    if(byte_sel[2])
      begin
        master_data[55:48] <= p_in[23:16];
      end
    if(byte_sel[3])
      begin
        master_data[63:56] <= p_in[31:24];
      end
```

```
   end

`else
  else if(latch[0] && !tip)//TX0 is selected..
   begin

    `ifdef SPI_MAX_CHAR_8
    if(byte_sel[0])
     begin
      master_data[7:0] <= p_in[7:0];
     end
    `endif

    `ifdef SPI_MAX_CHAR_16
    if(byte_sel[0])
     begin
      master_data[7:0] <= p_in[7:0];
     end
    if(byte_sel[1])
     begin
      master_data[15:8] <= p_in[7:0];
     end
    `endif
```

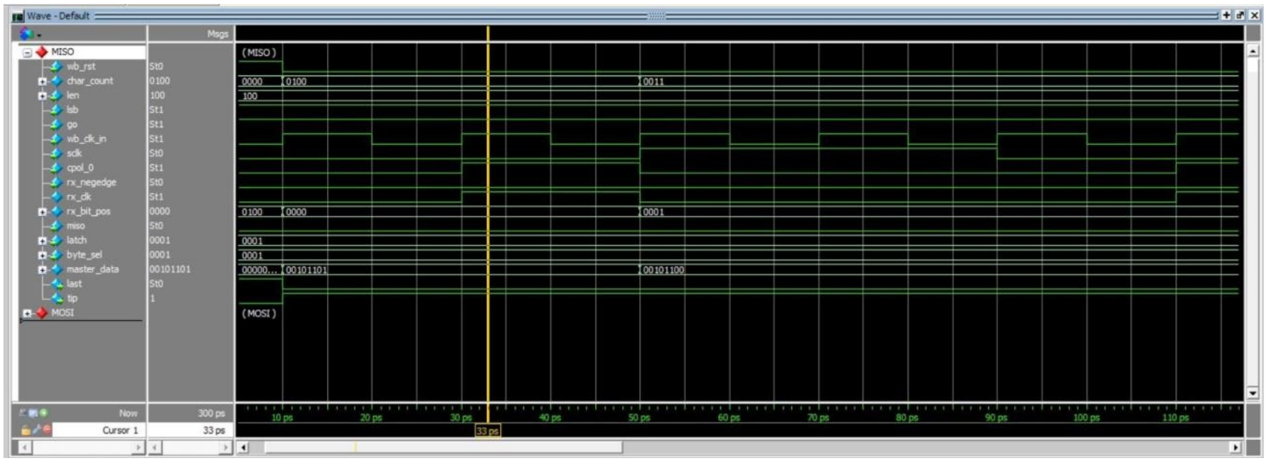
```
`ifdef SPI_MAX_CHAR_24

if(byte_sel[0])

  begin

    master_data[7:0] <= p_in[7:0];

  end


if(byte_sel[1])

  begin

    master_data[15:8] <= p_in[7:0];

  end

if(byte_sel[2])

  begin

    master_data[23:16] <= p_in[23:16];

  end


`endif


`ifdef SPI_MAX_CHAR_32


if(byte_sel[0])

  begin
```

```
      master_data[7:0] <= p_in[7:0];
     end
    if(byte_sel[1])
     begin
      master_data[15:8] <= p_in[7:0]; //I think here it should be 15:8
     end
    if(byte_sel[2])
     begin
      master_data[23:16] <= p_in[23:16];
     end
    if(byte_sel[3])
     begin
      master_data[31:24] <= p_in[31:24];
     end

    `endif
   end


  `endif
  `endif


//Receiving bit from serial line................
  else
```

```
      begin
        if(rx_clk)
          begin master_data[rx_bit_pos[`SPI_CHAR_LEN_BITS-1:0]] <= miso;
          end
      end
    end
  endmodule
```

Shift Register Module Waveforms

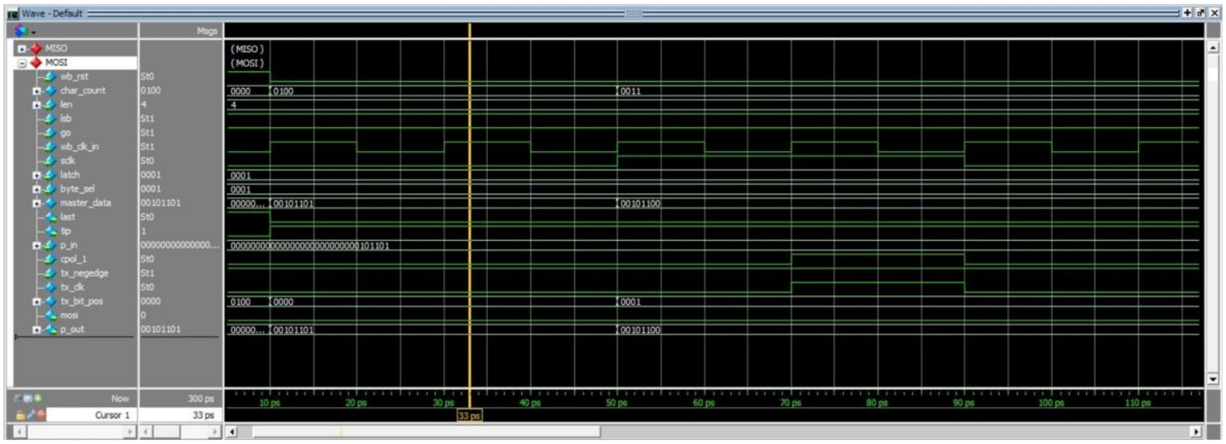
MISO



Wave - Default

	Msgs
MISO	
wb_rst	St0
char_count	0100
len	100
lsb	St1
go	St1
wb_clk_in	St1
sclk	St0
cpol_0	St1
rx_negedge	St0
rx_clk	St1
rx_bit_pos	0000
miso	St0
latch	0001
byte_sel	0001
master_data	00101101
last	St0
tip	1
MOSI	
Now	300 ps
Cursor 1	33 ps

MOSI



Wave - Default

	Msgs
MISO	
MOSI	
wb_rst	St0
char_count	0100
len	4
lsb	St1
go	St1
wb_clk_in	St1
sclk	St0
latch	0001
byte_sel	0001
master_data	00101101
last	St0
tip	1
p_in	00000000000000...
cpol_1	St0
tx_negedge	St1
tx_clk	St0
tx_bit_pos	0000
mosi	0
p_out	00101101
Now	300 ps
Cursor 1	33 ps

SPI Top Block

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.

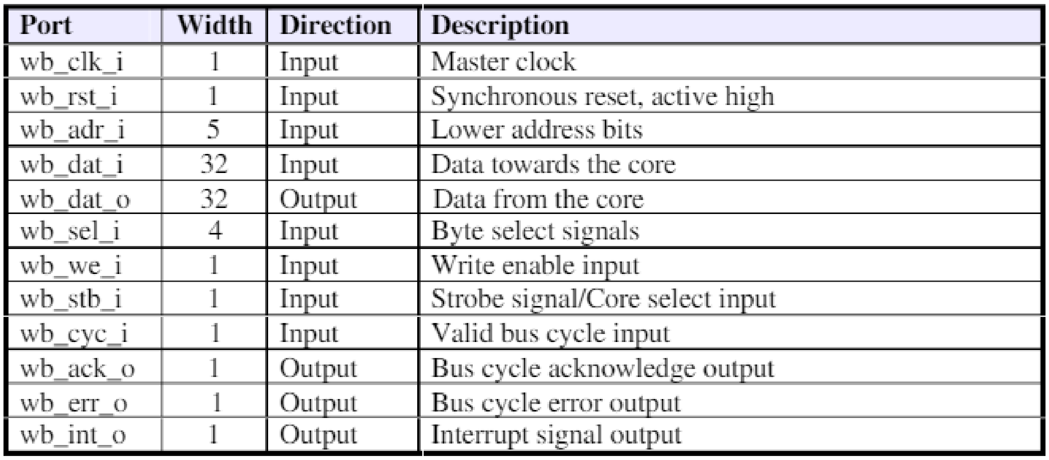
The code instantiates two sub-modules: "spi_clgen" and "spi_shift_reg".

- "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.

Wishbone Master Module Interface

SPI Master acts as a slave to Wishbone Interface. Wishbone master communicates with the host. Bus signals of Wishbone master are as follows:



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

Table 1: Wishbone interface signals

All output WISHBONE signals are registered and driven on the rising edge of wb_clk_i. All input WISHBONE signals are latched on the rising edge of wb_clk_i.

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

- Inputs:
    - clk_in: The clock input for the module.
    - rst_in: The reset input for the module.
    - ack_in: The acknowledgment input from the slave device.
    - err_in: The error input from the slave device.
    - dat_in: The data input for write operations.

- Outputs:
    - ❖ adr_o: The address output for the Wishbone master.
    - ❖ cyc_o: The cycle output for initiating a Wishbone transaction.
    - ❖ stb_o: The strobe output for indicating the start of a Wishbone transaction.
    - ❖ we_o: The write enable output for write operations.
    - ❖ dat_o: The data output for read operations.
    - ❖ sel_o: The select output for selecting the active byte lanes.

The module contains a task called "initialize" for initializing the internal signals of the module. It also contains a task called single_write for performing a single write operation. It sets the internal signals adr_temp, sel_temp, we_temp, dat_temp, cyc_temp, and stb_temp based on the provided inputs. It waits for the acknowledgment signal (ack_in) to go low before resetting the internal signals.

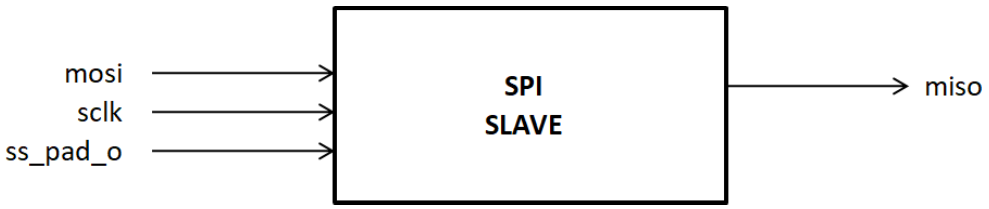
The module uses sequential blocks (always@(posedge clk_in)) to assign values to the output signals based on the internal signals. The rst_in signal is used to handle reset conditions for cyc_o and stb_o.

Overall, the wishbone_master module provides control and data signals for interacting with a Wishbone slave device, allowing for read and write operations.

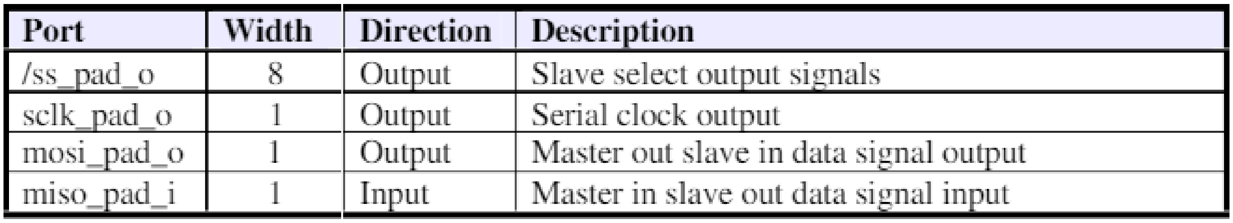
Slave Module Interface

The master can choose which slave it wants to talk to by setting the slave's SS line to a low voltage level. In the idle, non-transmitting state, the slave select line is kept at a high voltage level. Multiple SS pins may be available on the master, which allows for multiple slaves to be wired in parallel. If only one SS pin is present, multiple slaves can be wired to the master by daisy-chaining.

SPI Slave module here just behaves as a "slave" to test for the code results. It is not an actual slave.



Bus Signals for SPI slave module are as follows:



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

Table 2: SPI external connections

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

- Inputs:
  - sclk: The serial clock input for synchronization.
  - mosi: The input signal representing the data received from the SPI master.
  - ss_pad_o: A multi-bit output signal representing the chip select lines for the SPI slave.

- Output:
  - miso: The output signal representing the data to be transmitted from the SPI slave.

- Internal signals:
  - rx_slave: A register indicating whether the SPI slave is in receive mode.
  - tx_slave: A register indicating whether the SPI slave is in transmit mode.
  - temp1: A 128-bit register used for temporary storage of received data.
  - temp2: A 128-bit register used for temporary storage of received data.

On the posedge of sclk, if the chip select lines are not all high (ss_pad_o != 8'b1111111) and the slave is in receive mode (rx_slave is high) and not in transmit mode (tx_slave is low), the received data mosi is concatenated with the existing data in temp1.

On the negedge of sclk, if the slave is in receive mode (rx_slave is high) and not in transmit mode (tx_slave is low), the value in temp1 is assigned to miso1.

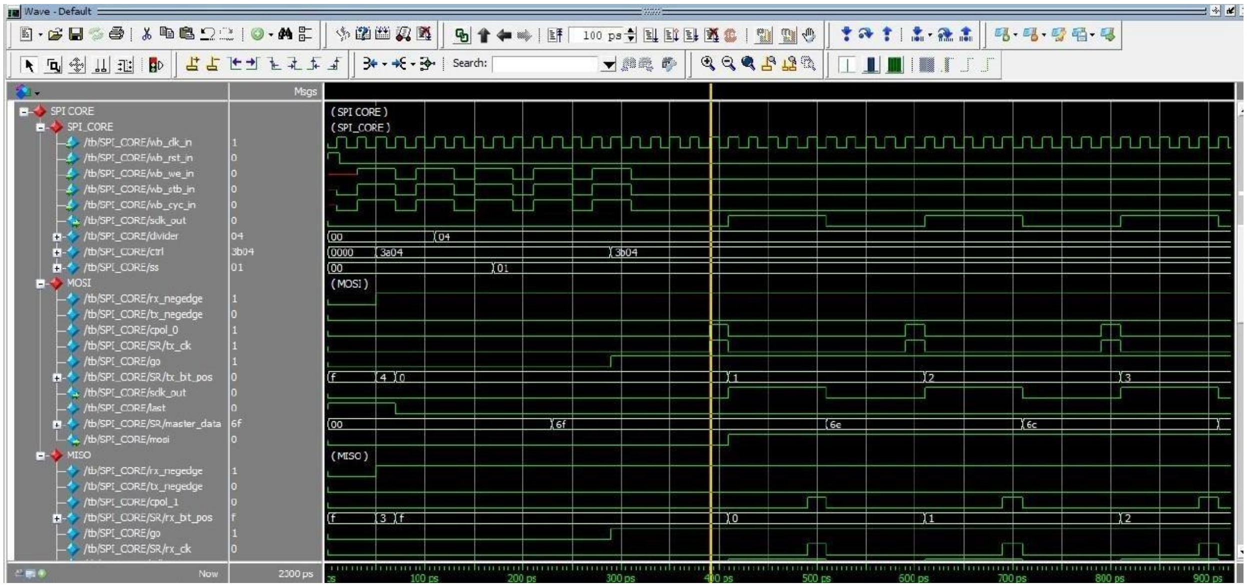
On the posedge of sclk, if the slave is in receive mode (rx_slave is high) and not in transmit mode (tx_slave is low), the value in temp2 is assigned to miso2.

The miso signal is assigned as the logical OR of miso1 and miso2, which represents the data to be transmitted from the SPI slave.

Overall, the spi_slave module receives data from the SPI master on the mosi line, processes and stores it in temp1 and temp2, and provides the output miso for transmitting data back to the SPI master. The chip select lines (ss_pad_o) are used to control the slave operation.

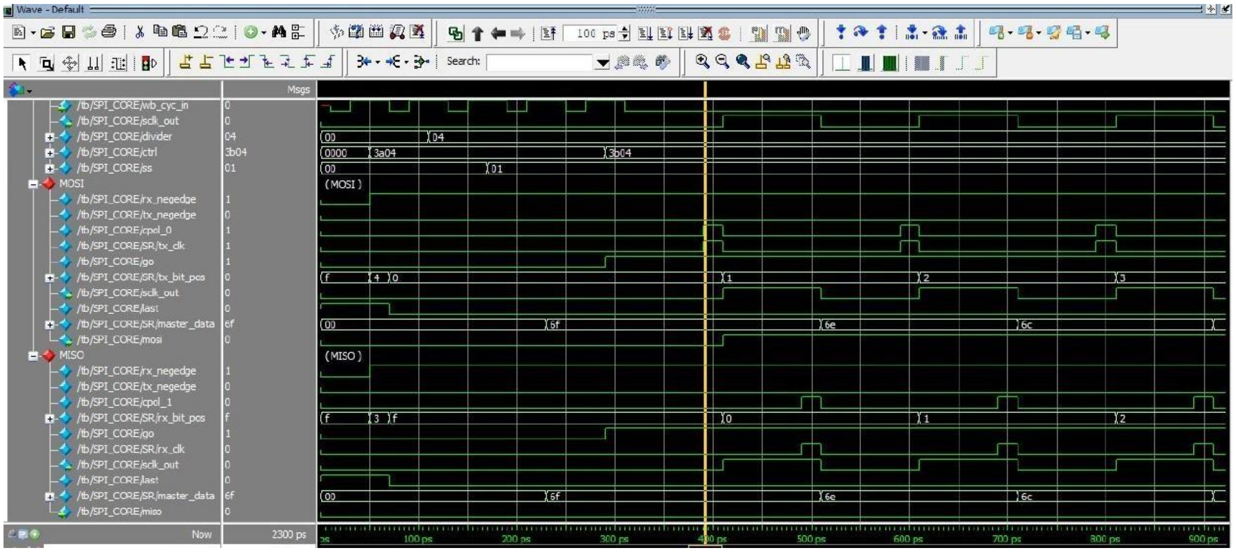
Test Case Module Waveforms

A. RX negedge = 1 and TX negedge = 0



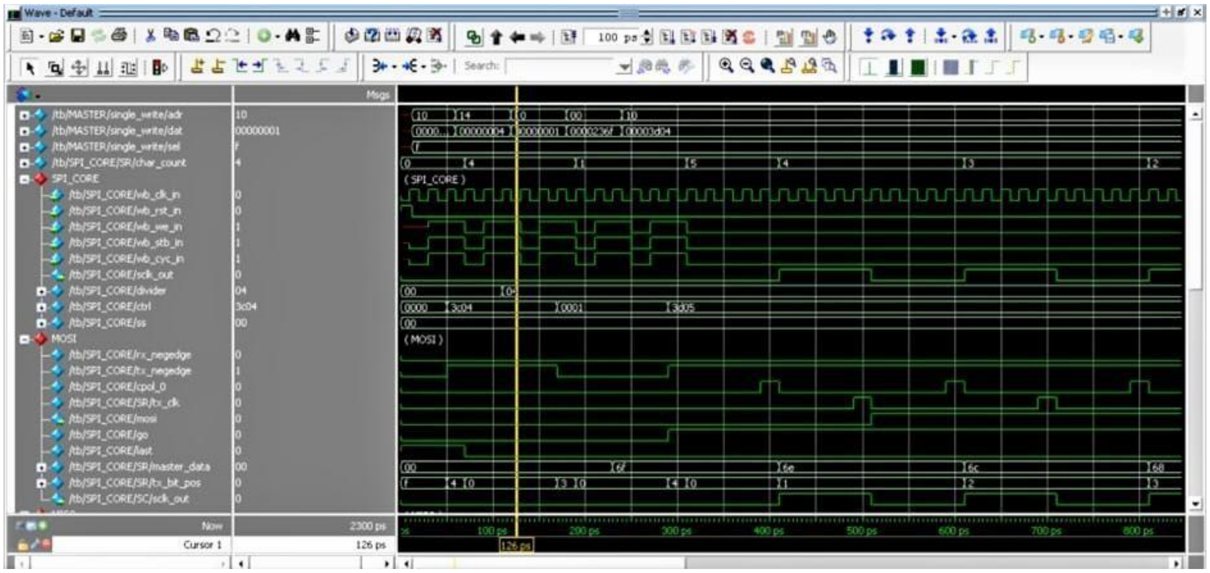
Wave - Default

	Msgs
SPI CORE	
SPI_CORE	
/tb/SPI_CORE/wb_clk_in	1
/tb/SPI_CORE/wb_rst_in	0
/tb/SPI_CORE/wb_we_in	0
/tb/SPI_CORE/wb_stb_in	0
/tb/SPI_CORE/wb_cyc_in	0
/tb/SPI_CORE/sclk_out	0
/tb/SPI_CORE/divider	04
/tb/SPI_CORE/ctrl	3b04
/tb/SPI_CORE/ss	01
MOSI	
/tb/SPI_CORE/rx_negedge	1
/tb/SPI_CORE/tx_negedge	0
/tb/SPI_CORE/cpol_0	1
/tb/SPI_CORE/SR/tx_clk	1
/tb/SPI_CORE/go	1
/tb/SPI_CORE/SR/tx_bit_pos	0
/tb/SPI_CORE/sclk_out	0
/tb/SPI_CORE/last	0
/tb/SPI_CORE/SR/master_data	6f
/tb/SPI_CORE/mosi	0
MISO	
/tb/SPI_CORE/rx_negedge	1
/tb/SPI_CORE/tx_negedge	0
/tb/SPI_CORE/cpol_1	0
/tb/SPI_CORE/SR/rx_bit_pos	f
/tb/SPI_CORE/go	1
/tb/SPI_CORE/SR/rx_clk	0
Now	2300 ps

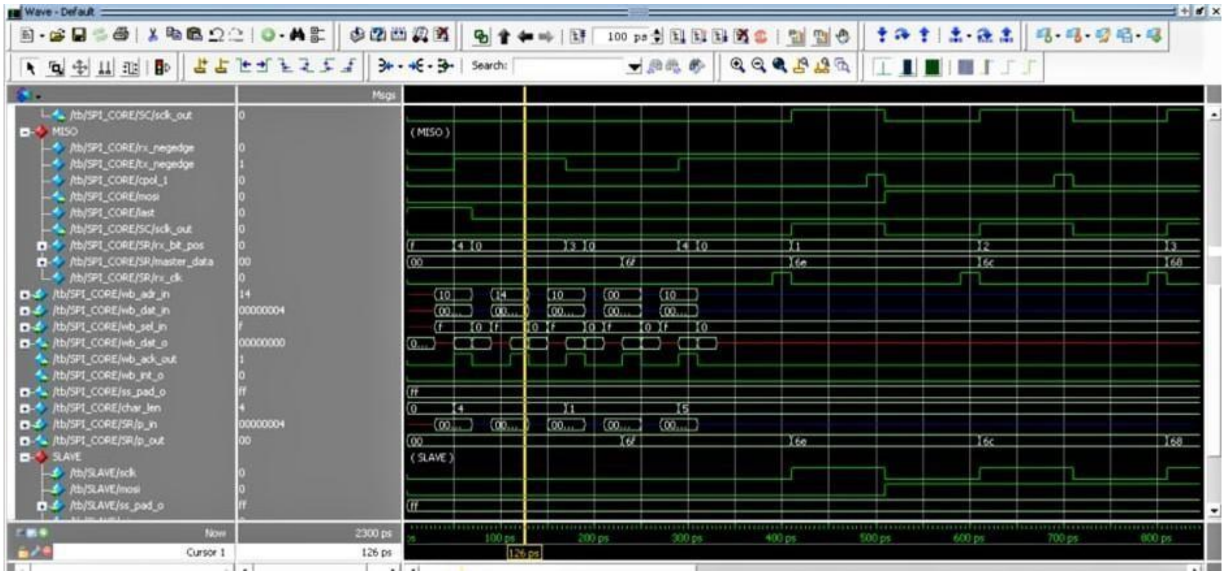


```
Wave - Default
                                         Msgs
    /tb/SPI_CORE/wb_cyc_in               0
    /tb/SPI_CORE/sclk_out                0
  + /tb/SPI_CORE/divider                 04
  + /tb/SPI_CORE/ctrl                    3b04
  + /tb/SPI_CORE/ss                      01
- MOSI
    /tb/SPI_CORE/rx_negedge              1
    /tb/SPI_CORE/tx_negedge              0
    /tb/SPI_CORE/cpol_0                  1
    /tb/SPI_CORE/SR/tx_clk               1
    /tb/SPI_CORE/go                      1
  + /tb/SPI_CORE/SR/tx_bit_pos           0
    /tb/SPI_CORE/sclk_out                0
    /tb/SPI_CORE/last                    0
  + /tb/SPI_CORE/SR/master_data          6f
    /tb/SPI_CORE/mosi                    0
- MISO
    /tb/SPI_CORE/rx_negedge              1
    /tb/SPI_CORE/tx_negedge              0
    /tb/SPI_CORE/cpol_1                  0
  + /tb/SPI_CORE/SR/rx_bit_pos           f
    /tb/SPI_CORE/go                      1
    /tb/SPI_CORE/SR/rx_clk               0
    /tb/SPI_CORE/sclk_out                0
    /tb/SPI_CORE/last                    0
  + /tb/SPI_CORE/SR/master_data          6f
    /tb/SPI_CORE/miso                    0
                                  Now    2300 ps
```

B. RX negedge =0 and TX negedge = 1



```
Wave - Default
                                         Msgs
  + /tb/MASTER/single_write/adr          10
  + /tb/MASTER/single_write/dat          00000001
  + /tb/MASTER/single_write/sel          f
  + /tb/SPI_CORE/SR/char_count           4
- SPI_CORE
    /tb/SPI_CORE/wb_clk_in               0
    /tb/SPI_CORE/wb_rst_in               0
    /tb/SPI_CORE/wb_we_in                1
    /tb/SPI_CORE/wb_stb_in               1
    /tb/SPI_CORE/wb_cyc_in               1
    /tb/SPI_CORE/sclk_out                0
  + /tb/SPI_CORE/divider                 04
  + /tb/SPI_CORE/ctrl                    3c04
  + /tb/SPI_CORE/ss                      00
- MOSI
    /tb/SPI_CORE/rx_negedge              0
    /tb/SPI_CORE/tx_negedge              1
    /tb/SPI_CORE/cpol_0                  0
    /tb/SPI_CORE/SR/tx_clk               0
    /tb/SPI_CORE/mosi                    0
    /tb/SPI_CORE/go                      0
    /tb/SPI_CORE/last                    0
  + /tb/SPI_CORE/SR/master_data          00
  + /tb/SPI_CORE/SR/tx_bit_pos           0
    /tb/SPI_CORE/SC/sclk_out             0
                                  Now    2300 ps
                             Cursor 1    126 ps
```



```
/tb/SPI_CORE/SC/sclk_out          0
MISO
  /tb/SPI_CORE/rx_negedge         0
  /tb/SPI_CORE/tx_negedge         1
  /tb/SPI_CORE/cpol_1             0
  /tb/SPI_CORE/mosi               0
  /tb/SPI_CORE/last               0
  /tb/SPI_CORE/SC/sclk_out        0
  /tb/SPI_CORE/SR/rx_bit_pos      0
  /tb/SPI_CORE/SR/master_data     00
  /tb/SPI_CORE/SR/rx_clk          0
/tb/SPI_CORE/wb_adr_in            14
/tb/SPI_CORE/wb_dat_in            00000004
/tb/SPI_CORE/wb_sel_in            f
/tb/SPI_CORE/wb_dat_o             00000000
/tb/SPI_CORE/wb_ack_out           1
/tb/SPI_CORE/wb_int_o             0
/tb/SPI_CORE/ss_pad_o             ff
/tb/SPI_CORE/char_len             4
/tb/SPI_CORE/SR/p_in              00000004
/tb/SPI_CORE/SR/p_out             00
SLAVE
  /tb/SLAVE/sclk                  0
  /tb/SLAVE/mosi                  0
  /tb/SLAVE/ss_pad_o              ff
Now                               2300 ps
Cursor 1                          126 ps
```

Verilog code for spi Wishbone Master:

```
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;
//Initialise 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
```

Verilog Code for Slave:

```
module spi_slave(sclk,mosi,ss_pad_o,miso);
input sclk,mosi;
input [`SPI_SS_NB-1:0]ss_pad_o;
output miso;
reg rx_slave =1'b0;
reg tx_slave=1'b0;

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)
begin
temp1<={temp1[126:0],mosi};
end
end

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

always@(negedge sclk)
begin
if(rx_slave && ~tx_slave)
begin
miso1<=temp1[127];
end
end

always@(posedge sclk)
begin
if(~rx_slave && tx_slave)
begin
miso2<=temp2[127];
end
end
```

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

SPI TEST BENCH:

```
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 being 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 slave register 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 the tx register with go busy being low and processor is sending 4 bits
MASTER.single_write(5'h00, 32'h0000_26ff, 4'b1111);
//configure control register with go_busy 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 low
MASTER.single_write(5'h10,32'h0000_3A04,4'b1111);
//configure divider with go_busy 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;*/



//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 low
MASTER.single_write(5'h10,32'h0000_3204,4'b1111);
//configure divider with go_busy 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_3304,4'b1111);
repeat(100)
@(negedge wb_clk_in);
$finish;
#10000
$finish;
end
```

```
initial
begin
/*//tx_neg=1, rx_neg = 0
SLAVE.rx_slave=0;
SLAVE.tx_slave=1;*/

//tx_neg=0, rx_neg=1
SLAVE.rx_slave=1;
SLAVE.tx_slave=0;
end
endmodule
```