SPI

(SERIAL PERIPHERAL INTERFACE)

Batch - SPI Design - VIT - June 2024

SUDHARSAN S

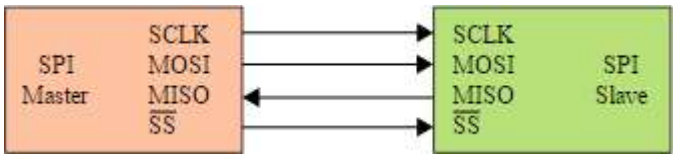
21BLC1079

CONTENTS

Introduction to the Serial Peripheral Interface (SPI) Protocol:

The Serial Peripheral Interface (SPI) is a synchronous serial communication protocol commonly used for short-distance communication between microcontrollers and peripheral devices. It is a full-duplex protocol, meaning that data can be transferred simultaneously in both directions. SPI is a four-wire protocol, utilizing four signal lines:

1. **Master-In Slave-Out (MOSI):** Data line for sending data from the master to the slave device.
2. **Master-Out Slave-In (MISO):** Data line for sending data from the slave to the master device.
3. **Serial Clock (SCLK):** Clock signal generated by the master device to synchronize data transfer.
4. **Slave Select (SS):** Chip select signal used by the master to select a specific slave device for communication.



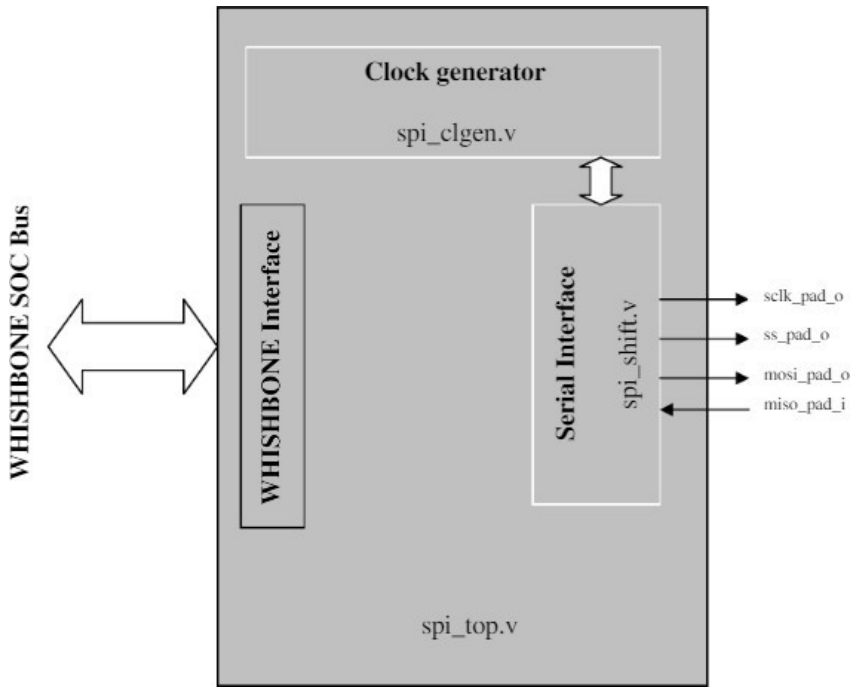
SPI communication is initiated by the master device, which generates the clock signal and controls the data transfer. The master selects a slave device by asserting its corresponding SS signal. Data is then transferred between the master and slave devices on the MOSI and MISO lines, synchronized by the SCLK signal.

SPI is a versatile protocol that supports various data transfer modes, including full-duplex, half-duplex, and slave-initiated modes. It is widely used in embedded systems due to its simplicity, speed, and flexibility.

SPI is a popular choice for applications requiring high-speed communication between microcontrollers and peripherals, such as sensors, memory devices, and display modules. Its simplicity and flexibility make it a versatile protocol for a wide range of embedded system application.

Architecture of SP 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

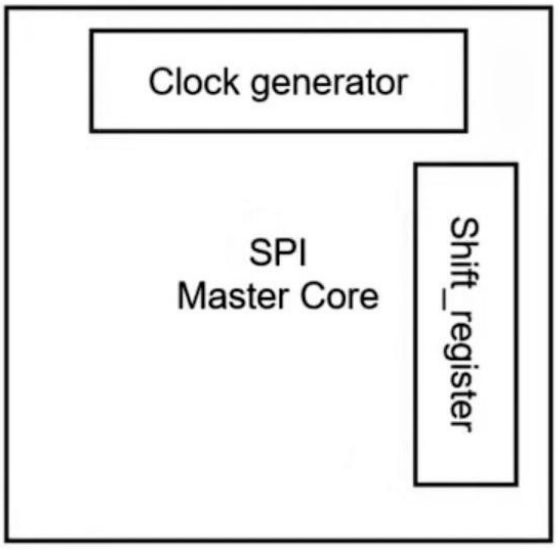
The architecture of an SPI Master Core typically consists of four main components:

1. **Clock Generator:** The clock generator typically includes a configurable clock divider to generate the desired SCLK frequency based on the system clock. It also generates the CS signals for each SPI slave device, ensuring proper timing and synchronization during SPI communication.
2. **Shift Register:** The shift register is typically a multi-bit register that can be loaded with data from the Wishbone interface. It shifts data bits out to the serial interface during transmission and captures incoming data bits from the serial interface during reception.

3. **Serial Interface:** The serial interface manages the actual bit-level communication with the SPI slave devices. It transmits data bits on the MOSI (Master Out Slave In) line and receives data bits on the MISO (Master-In Slave-Out) line, following the SPI protocol's timing and data framing.
4. **Wishbone Interface:** The Wishbone interface provides a standardized bus connection for the SPI Master Core to interact with the system's memory or processor. It allows for control and data transfers between the SPI Master Core and other system components, enabling the SPI Master Core to receive data for transmission and store received data from the SPI slave devices.

In summary, the SPI Master Core architecture comprises a clock generator for generating timing signals, a shift register for data manipulation, a serial interface for SPI communication, and a Wishbone interface for system integration. These components work together to enable efficient and reliable SPI communication between the master device and SPI slave devices

SPI Master Core Interface:



Module “spi.defines.v” is used to initialize all the macros 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
Name	Reserved	SS

Divider Register bus is as follows:

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

TxX Register bus is as follows:

Bit #	31:0
Access	R/W
Name	Tx

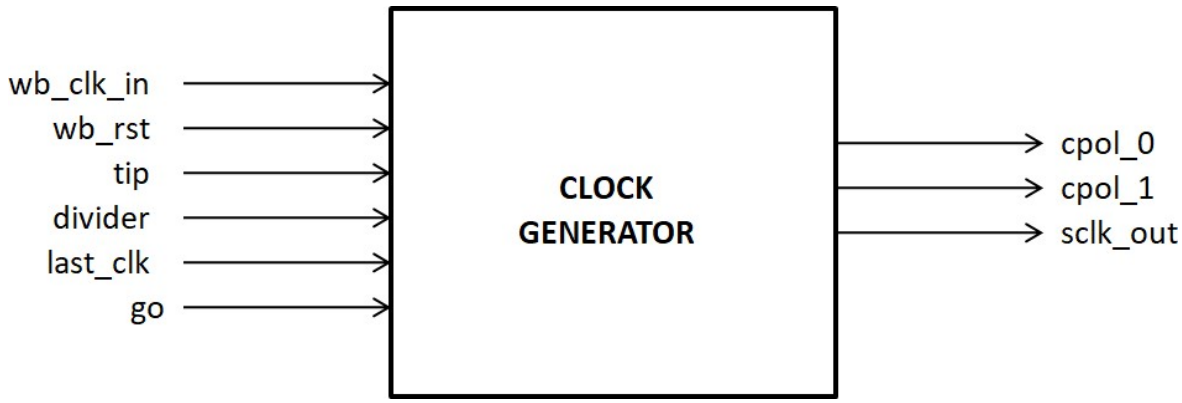
RxX Register bus is as follows:

Bit #	31:0
Access	R
Name	Rx

Clock Generator:

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.
- **Output:**
  - *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.

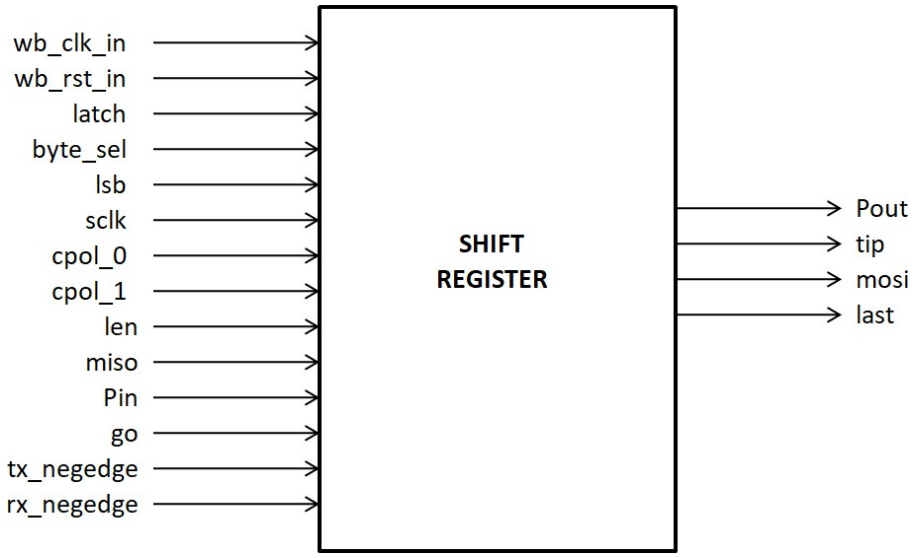


The clock generator block generates two clock signals: *rx_clk* and *tx_clk*. These clock signals are used to control the receive and transmit operations of the SPI shift register. The generation of these clocks is based on the input signals *rx_negedge*, *tx_negedge*, *last*, *sclk*, cpol_0, and *cpol_1*.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

Shift Register:



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.

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

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.

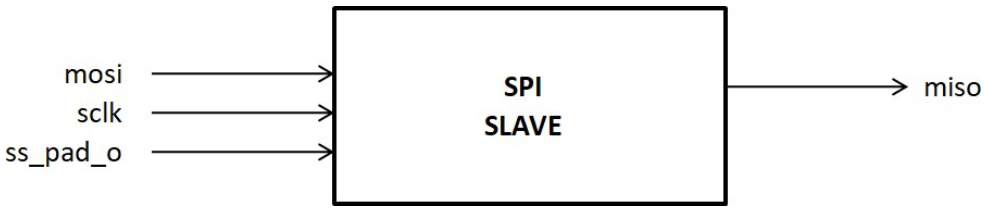
The module uses sequential blocks (always@ (posedge clk_in)) to assign values to the output signals based on the internal signals. The *rst_in* signal is used to handle reset conditions for *cyc_o* and *stb_o*.

Overall, the *wishbone_master* module provides control and data signals for interacting witha 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 anactual 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

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.

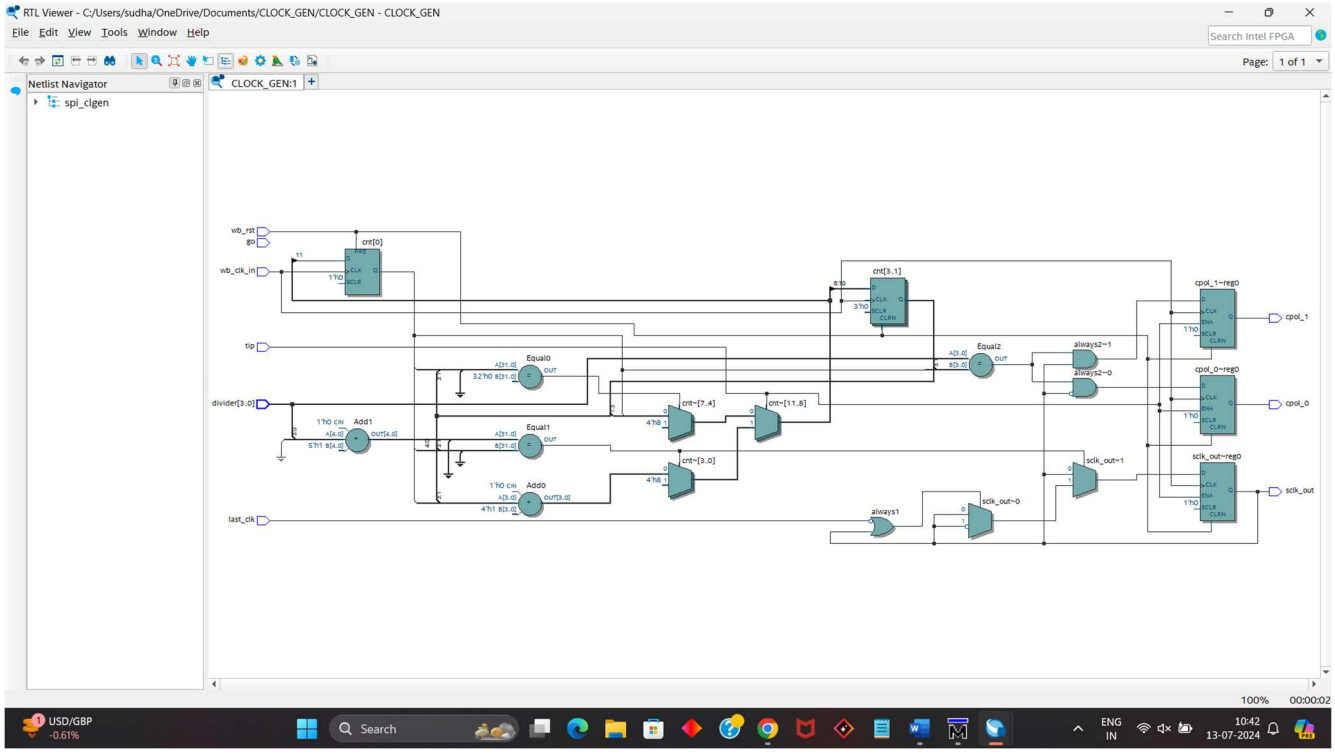
On the posedge of *sclk*, if the slave is in receive mode (*rx_slave* is high) and not in transmit mode (*tx_slave* is low), the value in temp2 is assigned to miso2.

The miso signal is assigned as the logical OR of miso1 and miso2, which represents the datato 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.

SYNTHESIS:

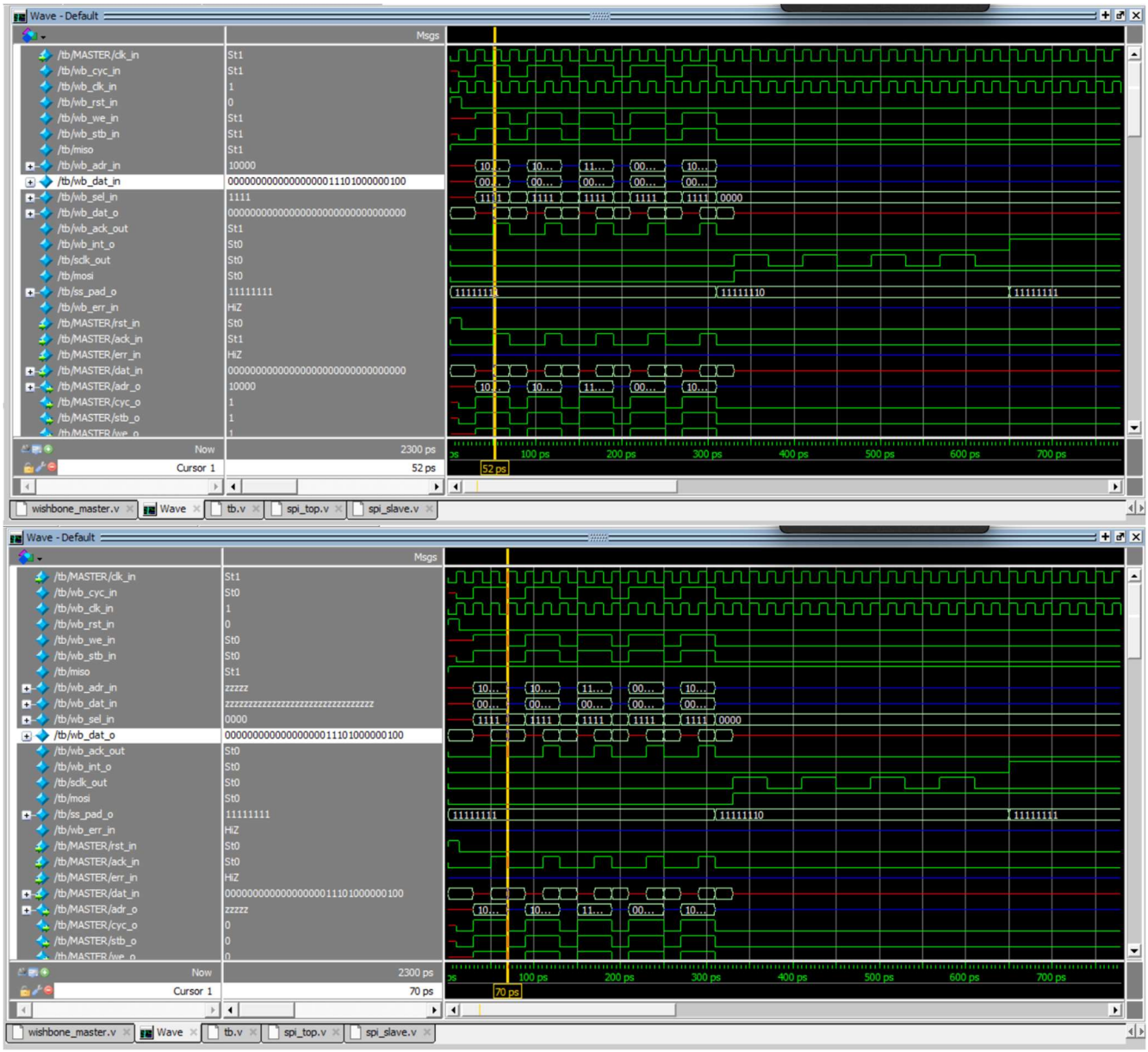
CLOCK GENERATOR



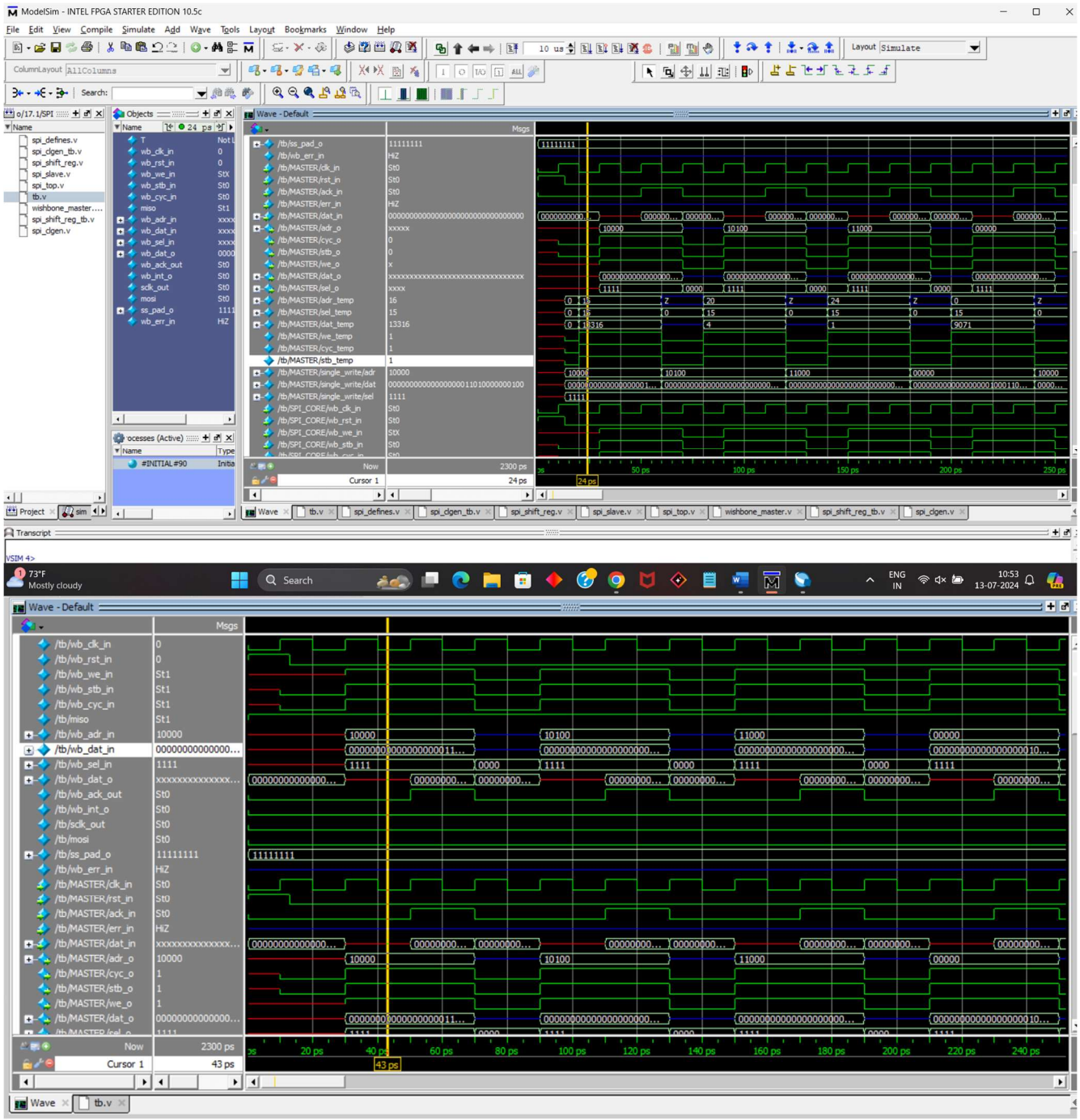
Test Case Module Waveforms:

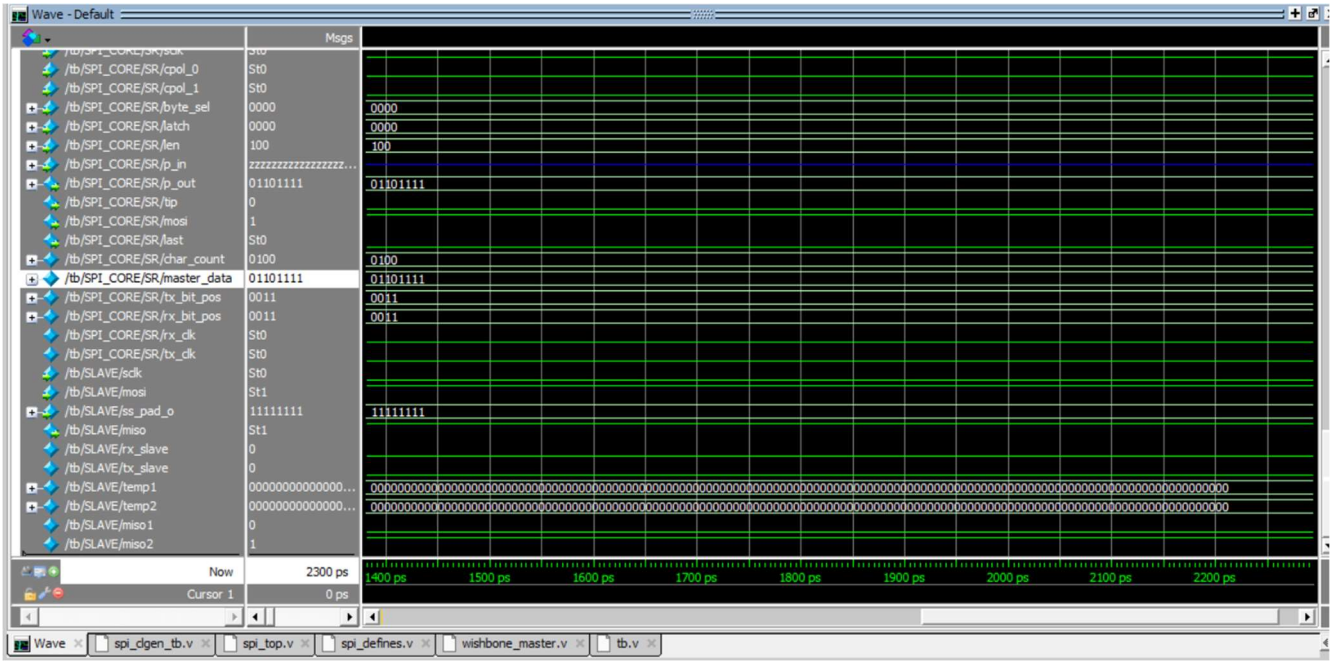
MASTER

A. **TX_NEG = 1, RX_NEG = 0, LSB = 1, CHAR_LEN = 4**

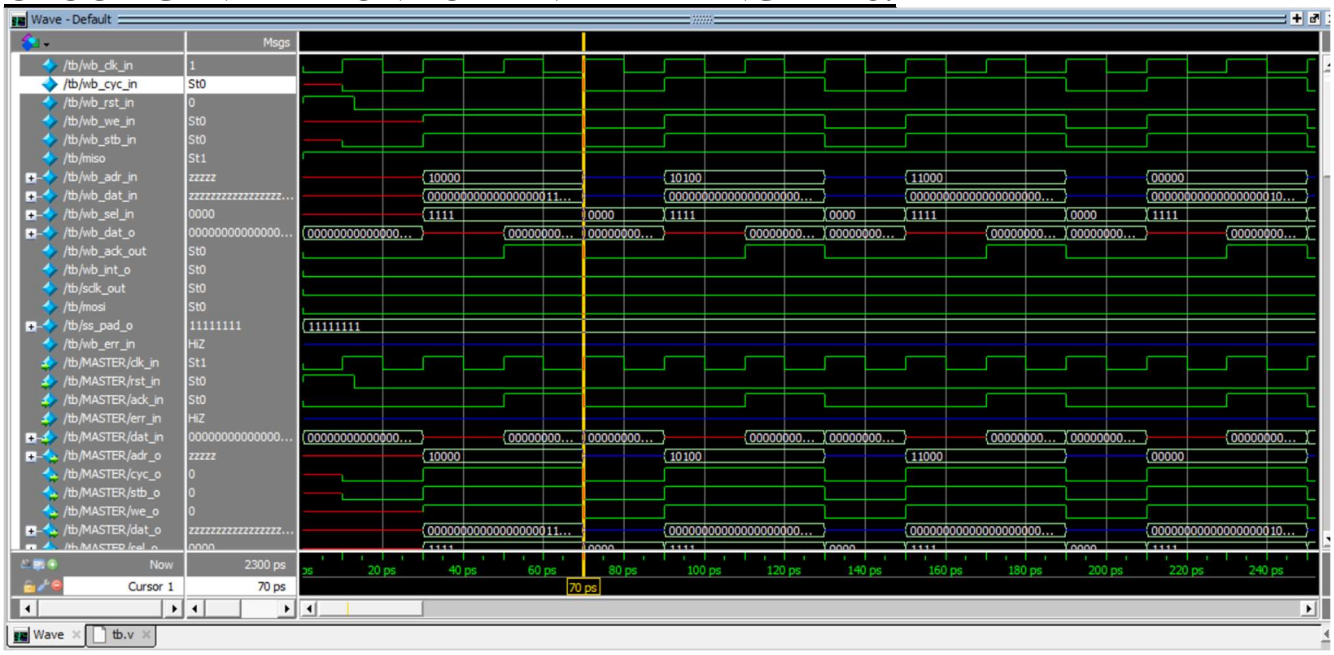


SET 2: TX_NEG = 1, RX_NEG = 0, LSB = 0, CHAR_LEN = 4

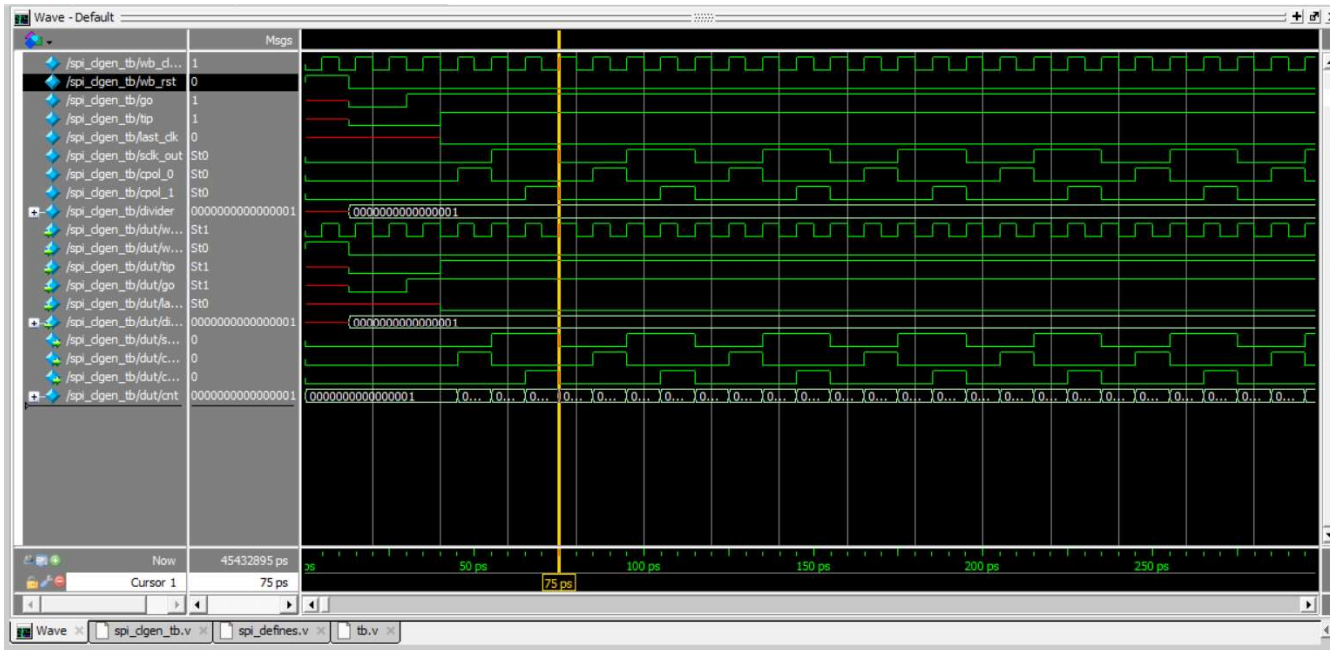

ModelSim - INTEL FPGA STARTER EDITION 10.5c
File Edit View Compile Simulate Add Wave Tools Layout Bookmarks Window Help
Wave - Default
Msgs
/tb/wb_clk_in
/tb/wb_rst_in
/tb/wb_we_in
/tb/wb_stb_in
/tb/wb_cyc_in
/tb/miso
/tb/wb_adr_in
/tb/wb_dat_in
/tb/wb_sel_in
/tb/wb_dat_o
/tb/wb_ack_out
/tb/wb_int_o
/tb/sclk_out
/tb/mosi
/tb/ss_pad_o
/tb/wb_err_in
/tb/MASTER/clk_in
/tb/MASTER/rst_in
/tb/MASTER/ack_in
/tb/MASTER/err_in
/tb/MASTER/dat_in
/tb/MASTER/adr_o
/tb/MASTER/cyc_o
/tb/MASTER/stb_o
/tb/MASTER/we_o
/tb/MASTER/dat_o
Now
2300 ps
Cursor 1
43 ps
Wave
tb.v



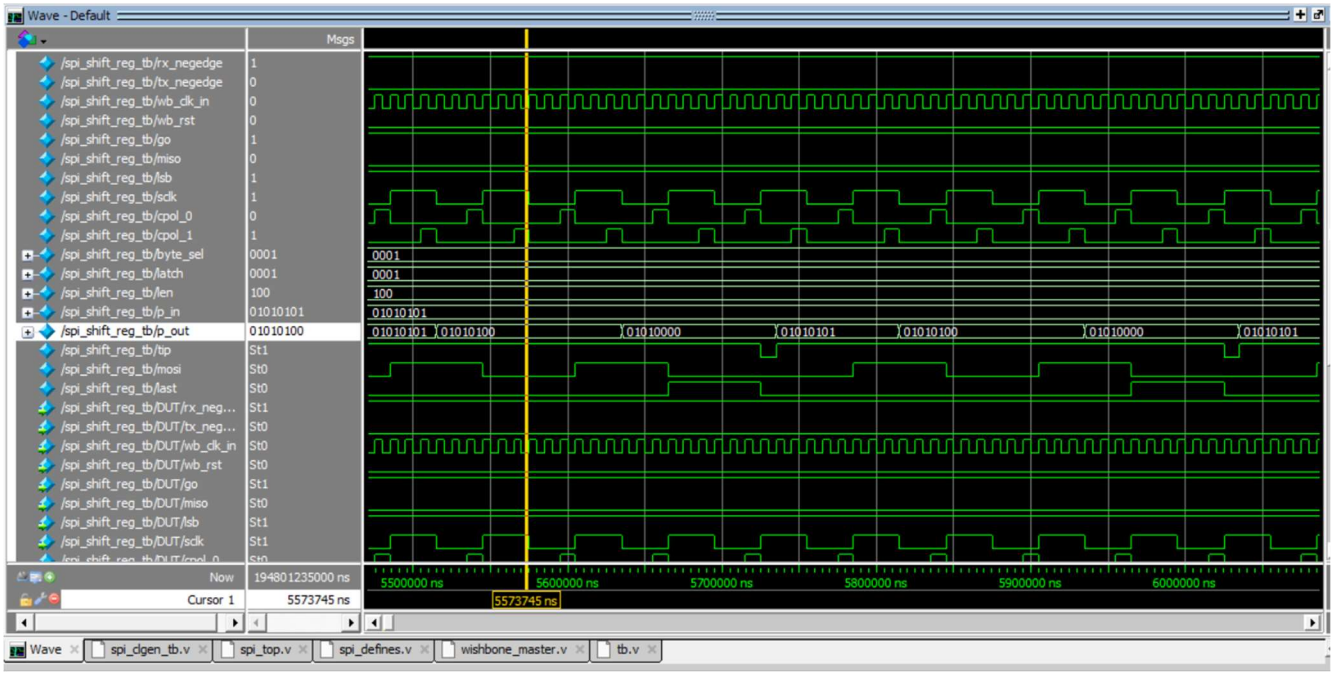
CLOCK GENERATION FOR DIVIDER LENGTH - 8:



CLOCK GENERATION FOR DIVIDER LENGTH - 16:



SHIFT REGISTER WAVEFORM:



VERILOG CODES:

SPI_DEFINES:

```
// Clock Generator Divider Length
//`define SPI_DIVIDER_LEN_8
`define SPI_DIVIDER_LEN_16
//`define SPI_DIVIDER_LEN_24
//`define SPI_DIVIDER_LEN_32

`ifdef SPI_DIVIDER_LEN_8
  `define SPI_DIVIDER_LEN      2   // Can be set from 1 to 8
  //`define SPI_DIVIDER_LEN      4   // Can be set from 1 to 8
`endif

`ifdef SPI_DIVIDER_LEN_16
  `define SPI_DIVIDER_LEN      16  // Can be set from 1 to 16
`endif

`ifdef SPI_DIVIDER_LEN_24
  `define SPI_DIVIDER_LEN      24  // Can be set from 1 to 24
`endif

`ifdef SPI_DIVIDER_LEN_32
  `define SPI_DIVIDER_LEN      32  // Can be set from 1 to 32
`endif
// Max Number Of Bits That Can Be Sent/Recieved At Once
`define SPI_MAX_CHAR_8
//`define SPI_MAX_CHAR_16
//`define SPI_MAX_CHAR_24
//`define SPI_MAX_CHAR_32
//`define SPI_MAX_CHAR_64
//`define SPI_MAX_CHAR_128

`ifdef SPI_MAX_CHAR_8
  `define SPI_MAX_CHAR 8
  `define SPI_CHAR_LEN_BITS 3
`endif

`ifdef SPI_MAX_CHAR_16
  `define SPI_MAX_CHAR 16
  `define SPI_CHAR_LEN_BITS 4
`endif

`ifdef SPI_MAX_CHAR_24
  `define SPI_MAX_CHAR 24
  `define SPI_CHAR_LEN_BITS 5
`endif

`ifdef SPI_MAX_CHAR_32
  `define SPI_MAX_CHAR 32
  `define SPI_CHAR_LEN_BITS 5
`endif

`ifdef SPI_MAX_CHAR_64
  `define SPI_MAX_CHAR 64
  `define SPI_CHAR_LEN_BITS 6
`endif

`ifdef SPI_MAX_CHAR_128
  `define SPI_MAX_CHAR 128
  `define SPI_CHAR_LEN_BITS 7
`endif

// Number of Devices Select Signals
```

```
//
`define SPI_SS_NB_8
//`define SPI_SS_NB_16
//`define SPI_SS_NB_24
//`define SPI_SS_NB_32

`ifdef SPI_SS_NB_8
  `define SPI_SS_NB 8
`endif
`ifdef SPI_SS_NB_16
  `define SPI_SS_NB 16
`endif
`ifdef SPI_SS_NB_24
  `define SPI_SS_NB 24
`endif
`ifdef SPI_SS_NB_32
  `define SPI_SS_NB 32
`endif
//
// Register Offsets
//
`define SPI_RX_0    5'b00000
`define SPI_RX_1    5'b00100
`define SPI_RX_2    5'b01000
`define SPI_RX_3    5'b01100
`define SPI_TX_0    5'b00000
`define SPI_TX_1    5'b00100
`define SPI_TX_2    5'b01000
`define SPI_TX_3    5'b01100
`define SPI_CTRL    5'b10000
`define SPI_DIVIDE  5'b10100
`define SPI_SS      5'b11000

// No. of Bits in CTRL Register

`define SPI_CTRL_BIT_NB 14
// Control Register Bit Position
`define SPI_CTRL_ASS         13
`define SPI_CTRL_IE          12
`define SPI_CTRL_LSB         11
`define SPI_CTRL_TX_NEGEDGE   10
`define SPI_CTRL_RX_NEGEDGE   9
`define SPI_CTRL_GO          8
`define SPI_CTRL_RES_1       7
`define SPI_CTRL_CHAR_LEN    6:0
```

SPI CLOCK GENERATOR:

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

 // Generation of the 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 <= 0;
        cpol_1 <= 0;
         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
```

SPI SHIFT REGISTER:

```
`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
        char_count <= {1'b0,len}; // This stores the character bits other than 128 bits
    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 rx_clk
  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}};
      // Recieving 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[15:8];
          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[15:8];
          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[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 bits from the serial line
  else
    begin
      if(rx_clk)
        master_data[rx_bit_pos[`SPI_CHAR_LEN_BITS-1:0]] <= miso;
      end
  end
endmodule
```

SPI SLAVE:

```
`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 recieving 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
```

SPI TOP:

```
`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
```

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;
    //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
```

TESTBENCH

SPI CLOCK GEN_TB:

```
`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
```

SPI SHIFT REG_TB:

```
`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
```

TOP TESTBENCH:

```
`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=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
endmodule
```

CONCLUSION:

The SPI Design project conducted during the VLSI Design Internship provided an in-depth understanding of the SPI protocol and its role in VLSI systems. The project involved designing and implementing essential components such as control registers, the SPI core, a clock generator, and a shift register, all integrated into a unified top block. Detailed RTL coding, along with the creation of test benches, simulation, and synthesis, validated the design's functionality and reliability. This hands-on experience solidified theoretical knowledge and showcased practical skills in digital system design and verification. Successfully completing the project demonstrates proficiency in designing, implementing, and testing intricate VLSI modules, establishing a solid foundation for future endeavors in the field.