Centre of Excellence in VLSI

SPI DESIGN PROJECT REPORT

REPORT SUBMITTED BY:

SHARVIKA S

22BEC0226

CONTENTS:

INTRODUCTION:

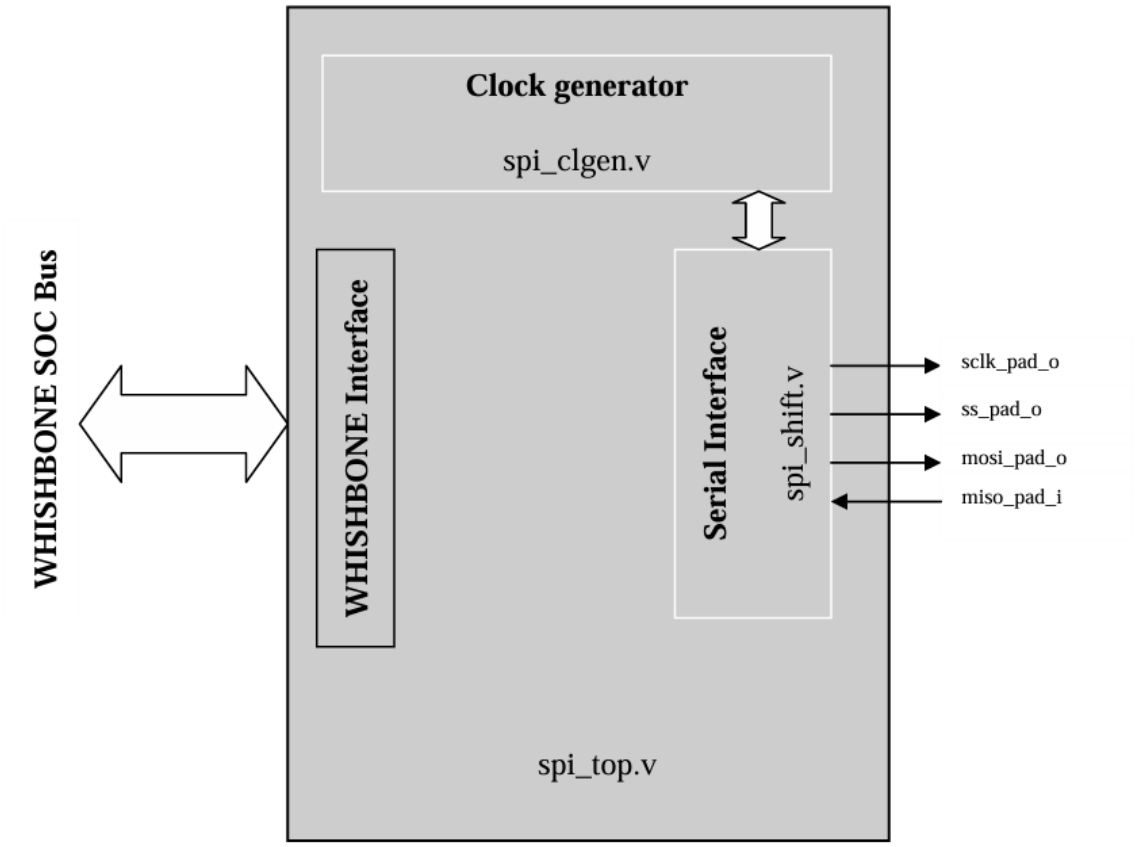
The design and implementation of the SPI (Serial Peripheral Interface) Master Core in Verilog are described in depth in this project report. The project's objectives were to deepen understanding of the SPI protocol and its applications and to have hands-on experience designing and constructing digital circuits using Verilog. The report offers a thorough explanation of the Verilog code, I/O ports, registers, and external connections used in the design, as well as a detailed description of the SPI Master Core design. It also covers the testing procedure and outcomes that were used to confirm the SPI Master Core's operation.

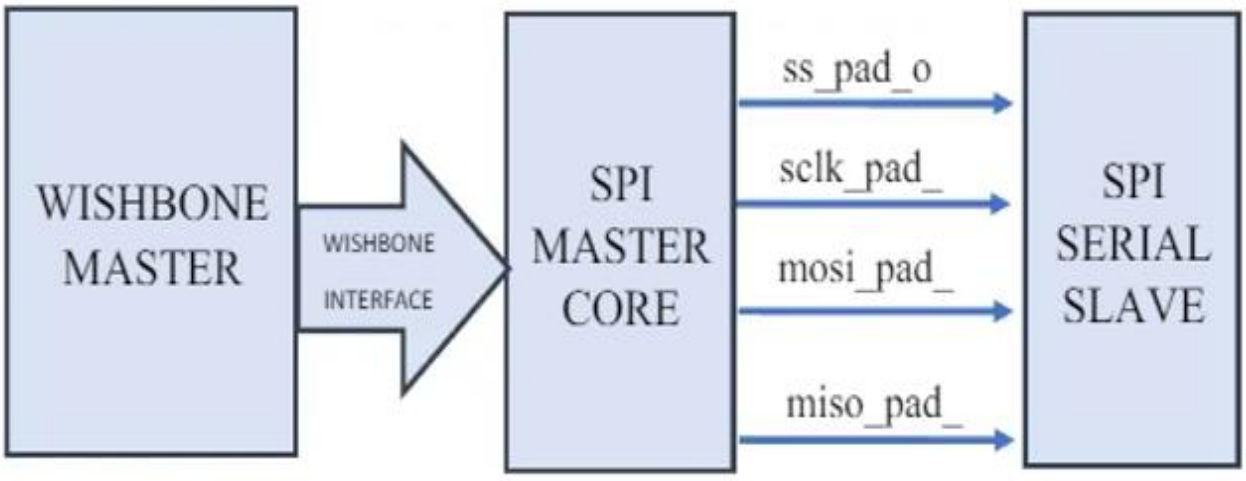
In addition, this document describes the specifications for the SPI Master Core, which is utilized for synchronous serial interfaces that enable cost-effective communication between devices like microcontrollers, DACs, ADCs, and others at the board level. While there isn't a single standard for asynchronous serial buses, industry guidelines rely on two popular implementations: SPI (a trademark of Motorola Semiconductor) and Microwire/Plus (a trademark of National Semiconductor). Numerous IC manufacturers create components that are compatible with SPI and Microwire/Plus. The SPI Master Core is capable of supporting both protocols in master mode and providing additional functionality. On the host side, the core acts as a WISHBONE-compliant slave device. It comes with key features such as:

- 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

A summary of the main conclusions of the project is included at the end of the report. All things considered, this project provided invaluable experience with the design and implementation of digital circuits using Verilog, along with a deeper comprehension of the SPI protocol and its uses.

CORE ARCHITECTURE:





SPI Master Core:

The Master Core in an SPI system initiates and controls all communication with slave devices. Its main functions include:

- Clock Generation:
  - ∞ Generates the clock signal (sclk) that synchronizes data transmission. The clock polarity (cpol) and phase can be configured to match the requirements of the slave device.
- Data Transmission:
  - ∞ Sends data to the slave device via the MOSI (Master Out Slave In) line.
  - ∞ Data is shifted out from the master to the slave in synchronization with the clock edges.
- Data Reception:
  - ∞ Receives data from the slave device via the MISO (Master In Slave Out) line.
  - ∞ Data is shifted into the master from the slave in synchronization with the clock edges.
- Control Signals:
  - ∞ Controls the chip select (CS) line to select and communicate with a specific slave device.
  - ∞ Manages start and stop conditions for communication.
- Transaction Management:
  - ∞ Initiates and terminates communication transactions.
  - ∞ Manages the transfer length and data frames

SPI Slave :

The Slave Core in an SPI system responds to communication initiated by the master device. Its main functions include:

- Clock Reception:
  - ∞ Receives the clock signal (sclk) from the master and synchronizes data transmission and reception accordingly.
- Data Reception:
  - ∞ Receives data from the master via the MOSI line.
  - ∞ Data is shifted into the slave from the master in synchronization with the clock edges.

- Data Transmission:
  - ∞ Sends data to the master via the MISO line.
  - ∞ Data is shifted out from the slave to the master in synchronization with the clock edges.
- Control Signals:
  - ∞ Monitors the chip select (CS) line to determine when it is being addressed by the master.
  - ∞ Responds to start and stop conditions for communication initiated by the master.
- Slave Logic:
  - ∞ Processes commands and data received from the master.
  - ∞ Prepares and transmits data back to the master as required.

Key Differences:

- Initiation: The master initiates communication and controls the clock, while the slave responds to the master's commands and clock.
- Control: The master controls the CS line and data flow, whereas the slave waits for the CS signal to know when to communicate.
- Clock: The master generates the clock signal, and the slave uses this clock to synchronize data exchange.

FEATURES:

- Full duplex synchronous serial data: Data can be transferred simultaneously in both directions.
- Variable length transfer words: Words can be up to 128 bits long.
- MSB or LSB first data transfer: The order of data transfer can be specified.
- Rx and Tx on both rising or falling edges: The receiver and transmitter can be independently controlled.
- Up to 32 slave select lines: The number of slave select lines can be configured.
- Auto Slave Select: Automatic slave select assertion can be enabled.

I/O PORTS:

WISHBONE INTERFACE:

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, while all input WISHBONE signals are captured on the rising edge of wb_clk_i.

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 LIST:

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

All registers are32-bit wide and accessible only with 32bits(all wb_sel_i signals must be active).

DATA RECEIVE REGISTERS[RXX]:

Bit #	31:0
Access	R
Name	Rx

The Data Receive registers (RxX) store the received data from the most recent transfer. The valid bits in these registers depend on the character length field in the CTRL register. For example, if CTRL[9:3] is set to 0x08, bits RxL[7:0] contain the received data. If the character length is 32 bits or less, Rx1, Rx2, and Rx3 are not used; if the character length is less than 64 bits, Rx2 and Rx3 are not used, and so on.

DATA TRANSMIT REGISTER[TXX]:

Bit #	31:0
Access	R/W
Name	Tx

The Data Receive registers store the data that will be sent over the next transfer. Valid bits are determined by the character length field in the CTRL register (i.e., if CTRL[9:3] is set to 0x08, the bit Tx0[7:0] will be sent in the subsequent transfer). Characters less than or equal to 32 bits do not use Tx1, Tx2, and Tx3. Characters fewer than 64 bits do not use Tx2 and Tx3, and so forth.

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

ASS :
Automatic generation of ss_pad_o signals occurs if this bit is set. This indicates that the SPI controller asserts the slave select signal, which is chosen in the SS register, when the transfer is initiated by setting CTRL[GO_BSY], then de-asserts it after the transfer is complete. Slave select signals are asserted and de-asserted by writing and clearing bits in the SS register if this bit is cleared.

IE :
When a transfer is complete, if this bit is set, the interrupt output is made active. Once a read or write is performed to any register, the interrupt signal is released.

LSB :
If this bit is set, the first bit received from the line will be placed in the LSB position in the Rx register (bit RxL[0]), and the LSB is sent first on the line (bit TxL[0]). If this portion is cleared, the MSB is sent or received first (the bit in the TxX/RxX register that corresponds to that depending on the CTRL register's CHAR_LEN field).

Tx_NEG:

This bit controls when the mosi_pad_o signal changes—either on the rising edge of a sclk_pad_o clock signal, if it is set, or on the falling edge of sclk_pad_o.

Rx_NEG:

The miso_pad_i signal is latching on the falling edge of a sclk_pad_o clock signal if this bit is set, and on the rising edge of a sclk_pad_o clock signal if it is not set.

GO_BSY:

The transfer is started by writing 1. This bit is automatically cleared once the transfer is complete, and it stays set during the transfer. This bit has no effect if you write 0.

CHAR_LEN:

This field specifies how many bits are transmitted in one transfer. Up to 64 bits can be transmitted.

CHAR_LEN = 0x01 … 1 bit

CHAR_LEN = 0x02 … 2 bits

…

CHAR_LEN = 0x7f … 127 bits

CHAR_LEN = 0x00 … 128 bits

DIVIDER REGISTER[DIVIDER]:

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

The value in this 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:

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

SLAVE SELECT REGISTER[SS]:

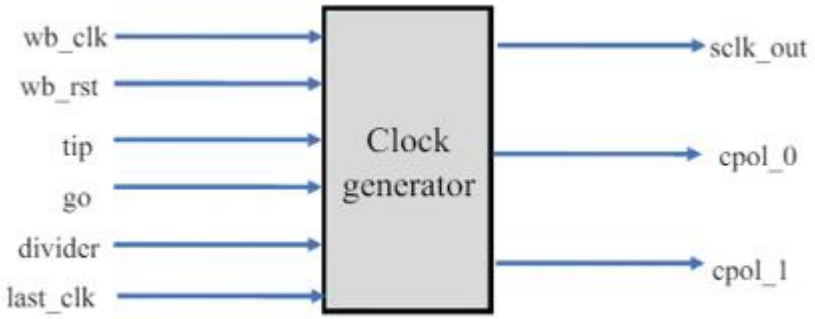
Bit #	31:8	7:0
Access	R	R/W

SS:

Writing 1 to any bit location in this field, if the CTRL[ASS] bit is cleared, puts the appropriate ss_pad_o line in an active state; writing 0 puts it back in an inactive state. The relevant ss_pad_o line will be automatically driven to the active state for the duration of the transfer if the CTRL[ASS] bit is set. If this is not the case, writing 1 to any bit location in this field will cause the line to remain in the inactive state.

CLOCK GENERATOR:

BLOCK DIAGRAM:



DESCRIPTION:

SPI (Serial Peripheral Interface) clock generator module is what the Verilog module "spi_clgen" seems to be. Based on the specified SPI configuration parameters, it generates the serial clock (SCLK) and provides control signals for clock polarity (CPOL). Here is a succinct summary of this code:

The system clock ("wb_clk_in"), the reset signal ("wb_rst"), the tip signal for transfer-in-progress indication, the "go" signal to start transfers, the last clock cycle tracking ("last_clk"), the divider signal to configure the clock divider, and the generated serial clock ("sclk_out"), "cpol_0," and "cpol_1" for clock polarity are among the input signals that the module receives.

The code makes use of a counter “cnt” to depend 1/2 of durations of the clock. It is up to date for the duration of every clock cycle and reset in case of a reset signal or while the switch is in progress (“tip” is asserted). The counter is used to decide while to toggle the SCLK signal.

The era of the serial clock “sclk_out” is synchronized with the machine clock and relies upon the counter value, the divider, and the “last_clk” sign. It guarantees that the SCLK transitions are generated atappropriate times.

The module additionally detects the positive and negative edges of the generated clock and units the “cpol_0” and n “cpol_1” alerts primarily based totally at the SPI mode's clock polarity requirements. These alerts are adjusted primarily based totally n at the SCLK signals edges and the divider value.

CODE:

```
`include "spi_define.v"

module spi_clockgen (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;

 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

  always@(posedge wb_clk_in or posedge wb_rst)
```

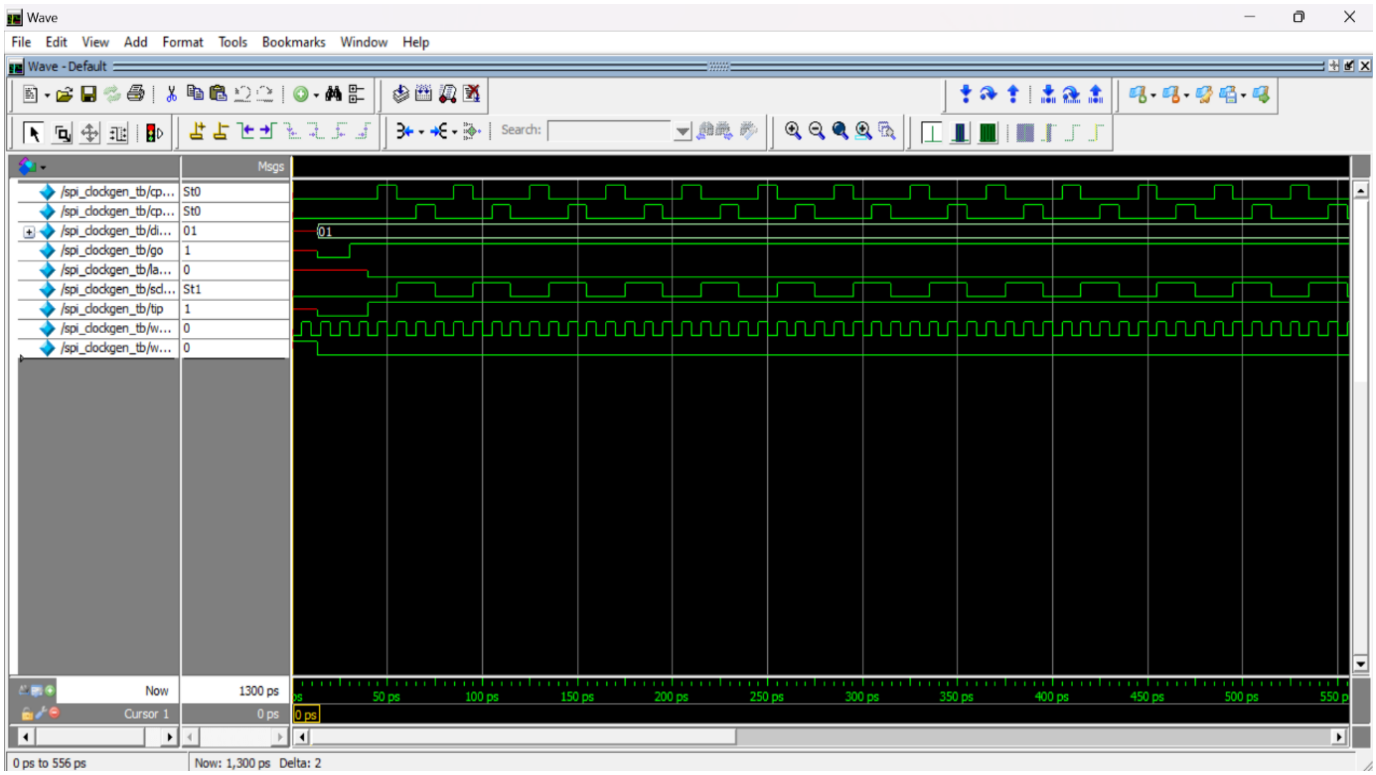
```
begin
 if(wb_rst)
  begin
   cpol_0 <= 1'b0;
   cpol_1 <= 1'b0;
  end
 else
  begin
   cpol_0 <= 0;
   cpol_1 <= 0;
    if(tip)
     begin
      if(~sclk_out)
       begin
        if(cnt == divider)
         begin
          cpol_0 <= 1;
         end
       end
     end
    if(tip)
     begin
      if(sclk_out)
       begin
        if(cnt == divider)
         begin
          cpol_1 <= 1;
         end
       end
     end
  end
 end



endmodule
```

TESTBENCH:

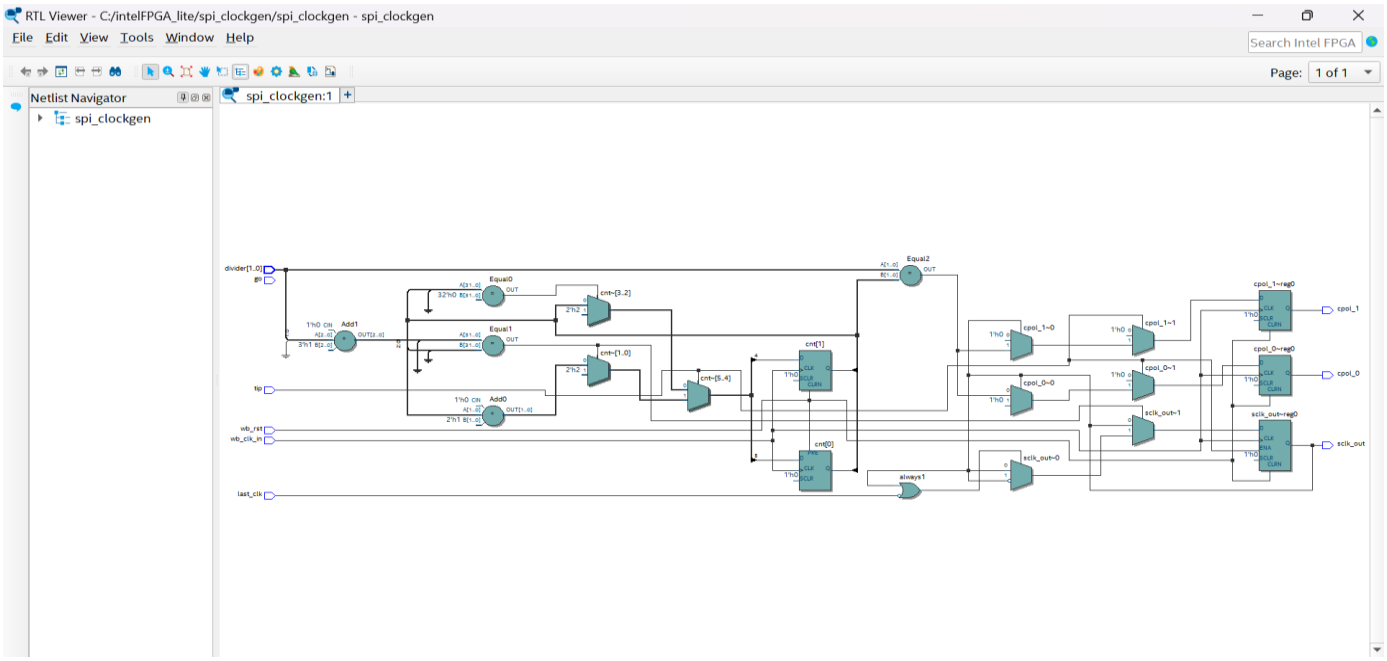
```
`include "spi_define.v"
module spi_clockgen_tb;
 reg wb_clk_in;
 reg wb_rst;
 reg go;
 reg tip;
 reg last_clk;
 wire sclk_out;
 wire cpol_0;
 wire cpol_1;
 reg [`SPI_DIVIDER_LEN-1:0] divider;
 spi_clockgen dut (
  .wb_clk_in(wb_clk_in),
  .wb_rst(wb_rst),
  .go(go),
  .tip(tip),
  .last_clk(last_clk),
  .divider(divider),
  .sclk_out(sclk_out),
  .cpol_0(cpol_0),
  .cpol_1(cpol_1)
 );

 always begin
  #5 wb_clk_in = ~wb_clk_in;
 end
 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
```

WAVEFORM:

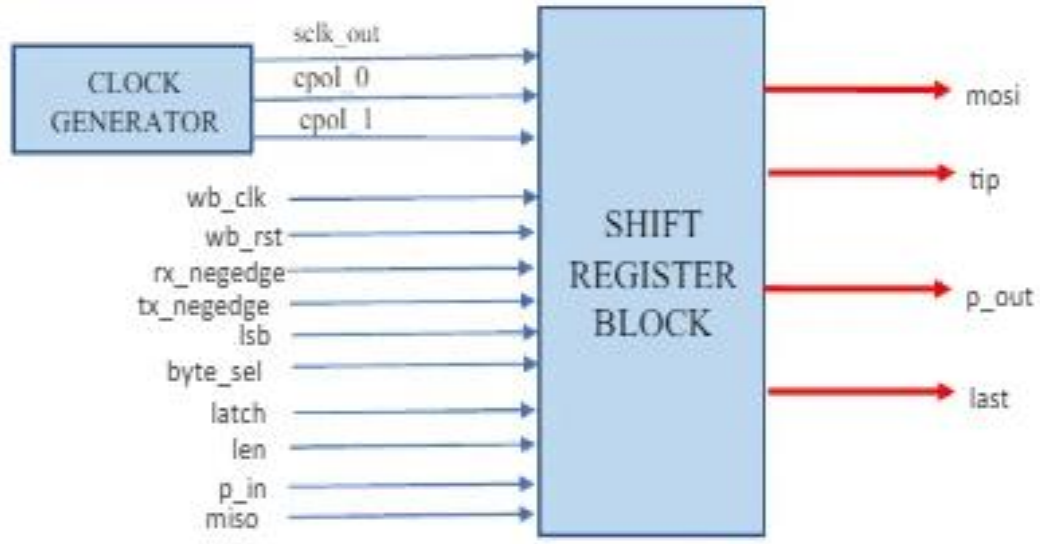


SYNTHESIS:


RTL Viewer - C:/intelFPGA_lite/spi_clockgen/spi_clockgen - spi_clockgen
File Edit View Tools Window Help
Search Intel FPGA
Page: 1 of 1
Netlist Navigator
spi_clockgen
spi_clockgen:1
divider[1..0]
tip
wb_rst
wb_clk_in
last_clk
Add1
Add0
Equal0
Equal1
Equal2
cnt[1]
cnt[0]
always1
cpol_1
cpol_0
sclk_out

SHIFT REGISTER:

BLOCK DIAGRAM:



DESCRIPTION:

- **Module Declaration:**
  Inputs and outputs are defined, including clock signals, reset, control signals, and data lines.
- **Registers and Wires:**
  Various internal registers and wires are declared to hold data, bit positions, and clock signals.
- **Character Counter:**
  Counts down the number of characters to be transmitted or received. Resets on wb_rst and updates based on cpol_0 when tip is active.
- **Transfer in Progress (TIP):**
  Indicates whether a transaction is ongoing. Activated on go signal and deactivated when the last bit is transmitted, conditioned by cpol_0.
- **Last Character Check:**
  Checks if all characters have been transmitted (char_count is zero).
- **Transmit (MOSI) Line:**
  Updates the MOSI line with the next bit to be transmitted based on the tx_bit_pos and tx_clk.

- **Clock Assignment:**
  Assigns tx_clk and rx_clk based on clock edge selection (tx_negedge and rx_negedge) and cpol values.
- **Bit Position Calculation:**
  Calculates the bit positions for transmission (tx_bit_pos) and reception (rx_bit_pos) based on lsb, len, and char_count.
- **Parallel Data Input:**
  Loads data from the parallel input (p_in) into master_data based on latch and byte_sel signals, depending on the defined maximum character length (SPI_MAX_CHAR).
- **Reception:**
  Updates master_data with incoming bits from the miso line when rx_clk is active.
  By defining these points, the module facilitates the shifting of data in and out over an SPI interface, controlled by various signals and parameters.

CODE:

```
`include "spi_define.v"


module spi_shift_register(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;
  reg [`SPI_CHAR_LEN_BITS:0] tx_bit_pos;
  reg [`SPI_CHAR_LEN_BITS:0] rx_bit_pos;
  wire rx_clk;
  wire tx_clk;


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


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


assign last = ~(|char_count);


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


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


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


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



assign p_out = master_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)
    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)
    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)
    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



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

```
      end
   end
endmodule
```

TESTBENCH:

```
`include "spi_define.v"
`timescale 1us/1ns
module spi_shift_register_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;


 spi_shift_register DUT (rx_negedge,
                tx_negedge,
```

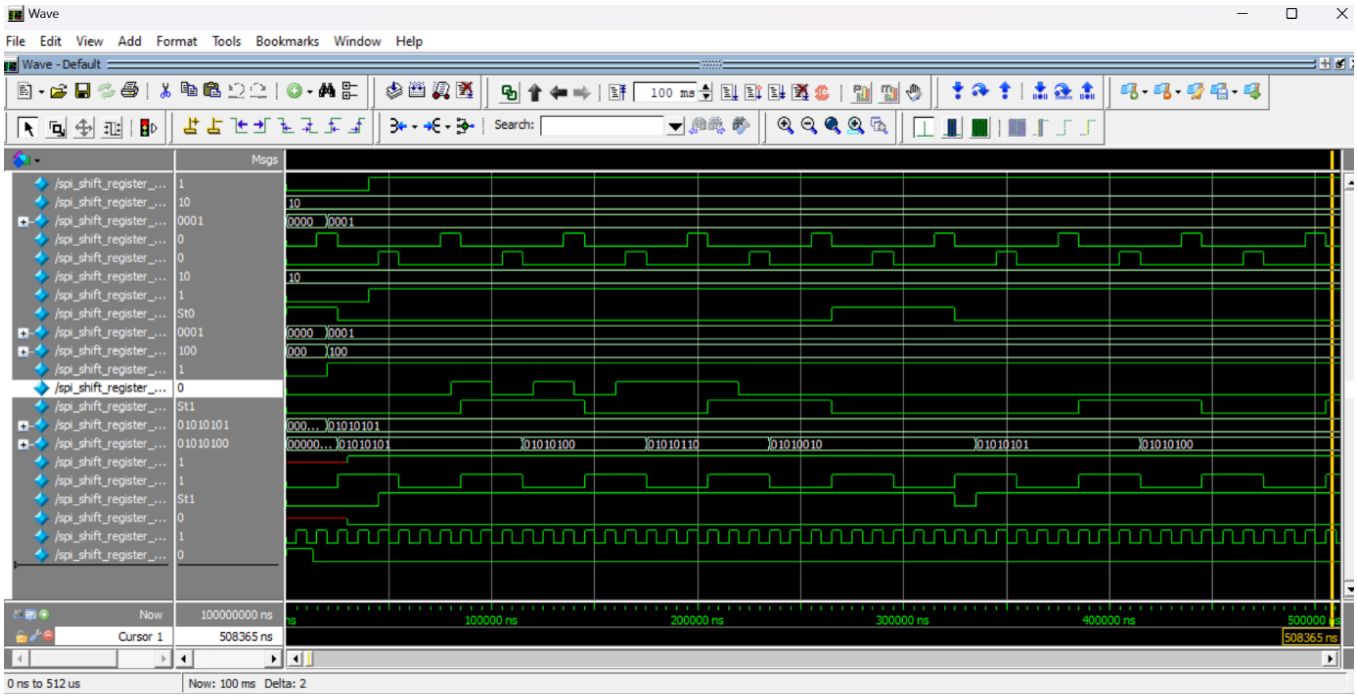
```
                byte_sel,
                latch,
                len,
                p_in,
                wb_clk_in,
                wb_rst,
                go,
                miso,
                lsb,
                sclk,
                cpol_0,
                cpol_1,
                p_out,
                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
```

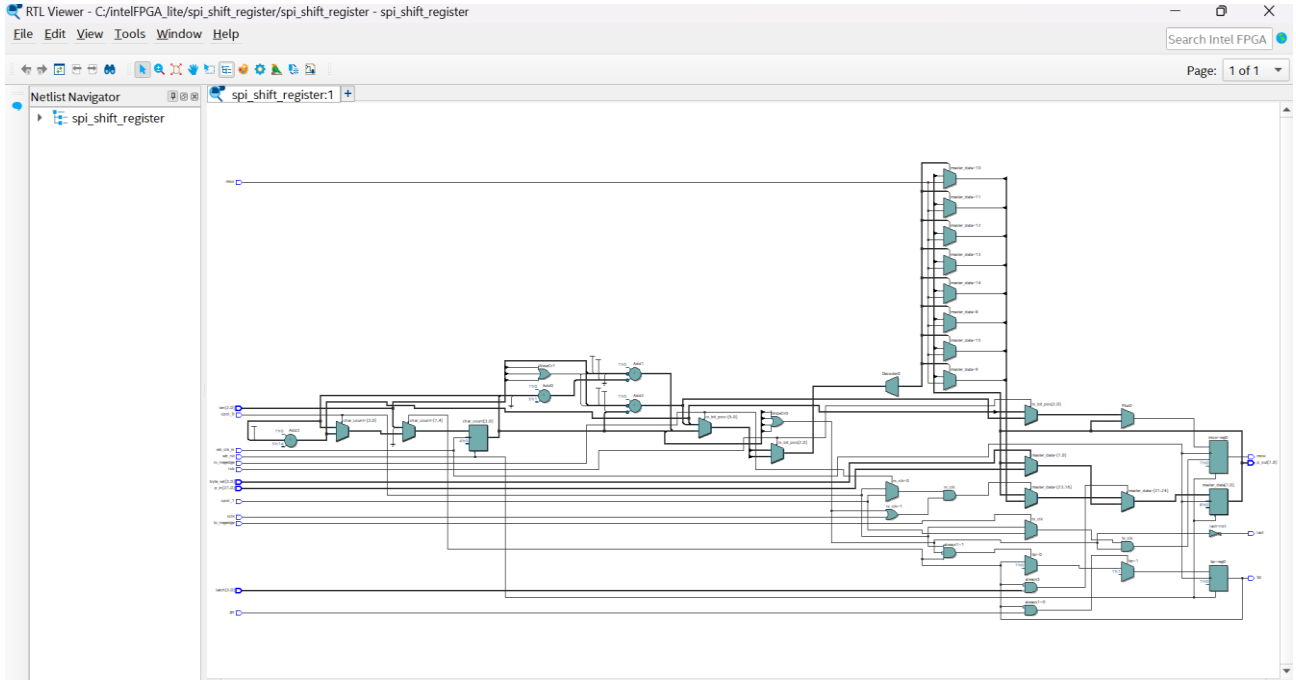
```
      begin
        repeat(divider_value*2 + 1)
          @(posedge wb_clk_in);
        cpol_0 = 1'b1;
        @(posedge wb_clk_in)
        cpol_0 = 1'b0;
      end
end

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
```

WAVEFORM:

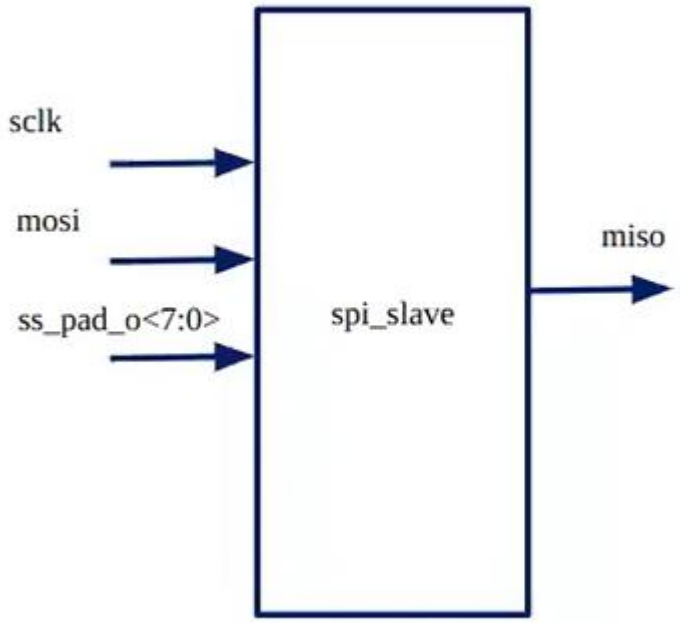


SYNTHESIS:



SPI SLAVE:

BLOCK DIAGRAM:



DESCRIPTION:

Signal Declarations

- **Inputs:**
  - ∞ sclk: Serial clock from the master.
  - ∞ mosi: Master Out Slave In data line.
  - ∞ ss_pad_o: Slave select signals, where a low state indicates the active slave.
- **Output:**
  - ∞ miso: Master In Slave Out data line, carrying data from the slave to the master.
- Data Registers
  - ∞ temp1 and temp2: 128-bit registers to store received and transmitted data, respectively.
- **Control Signals**
  - ∞ rx_slave: Indicates the slave is in receive mode.
  - ∞ tx_slave: Indicates the slave is in transmit mode.
- **Data Shifting**

Receive Mode (rx_slave):

- ∞ Data from mosi is shifted into temp1 on the rising edge of sclk when the slave is active and in receive mode.
- ∞ On the falling edge of sclk, the MSB of temp1 is assigned to miso1.

Transmit Mode (tx_slave):

- ∞ Data from mosi is shifted into temp2 on the falling edge of sclk when the slave is active and in transmit mode.
- ∞ On the falling edge of sclk, the MSB of temp2 is assigned to miso2.

- **MISO Generation**
  - ∞ The output miso is the logical OR of miso1 and miso2, representing the data being transmitted back to the master. The MSB of the relevant register (temp1 or temp2) is used based on the current mode (receive or transmit).

The spi_slave module performs the following key functions:

- **Receives Data**: Shifts incoming data from mosi into temp1 or temp2 based on the mode and clock edge.
- **Transmits Data**: Sets miso to the MSB of temp1 or temp2 to send data back to the master.
- **Mode Control**: Uses rx_slave and tx_slave signals to switch between receiving and transmitting modes.

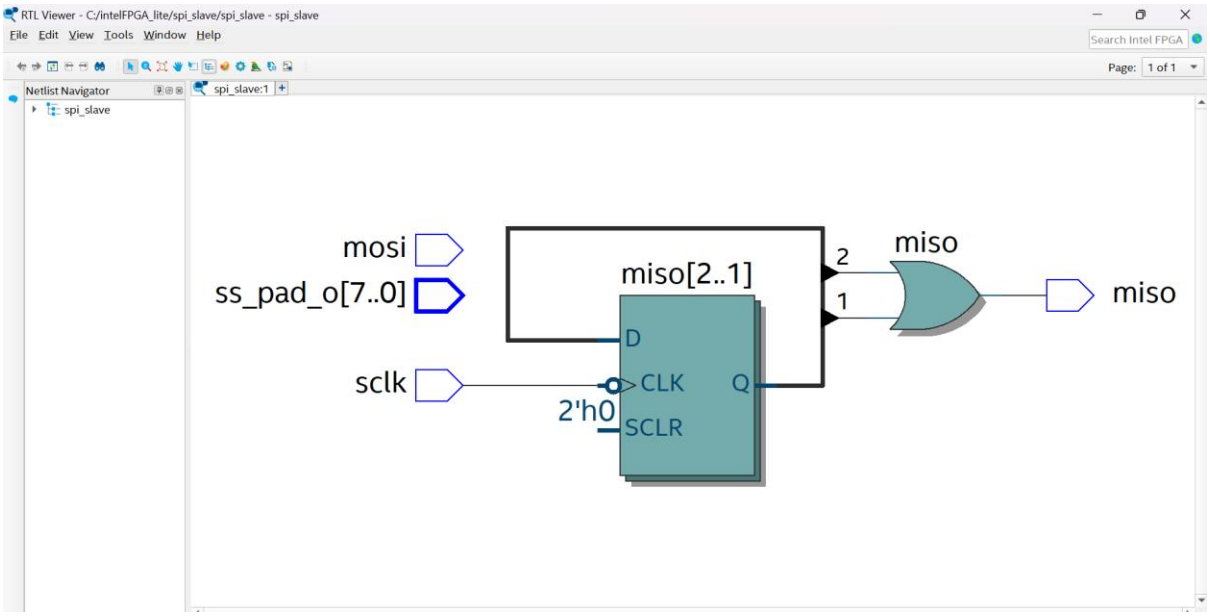
By managing these operations, the spi_slave module allows a device to communicate as a slave in an SPI system, receiving data from and sending data to the master device as dictated by the SPI protocol.

CODE:

```
`include "spi_define.v"


module spi_slave (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;
```

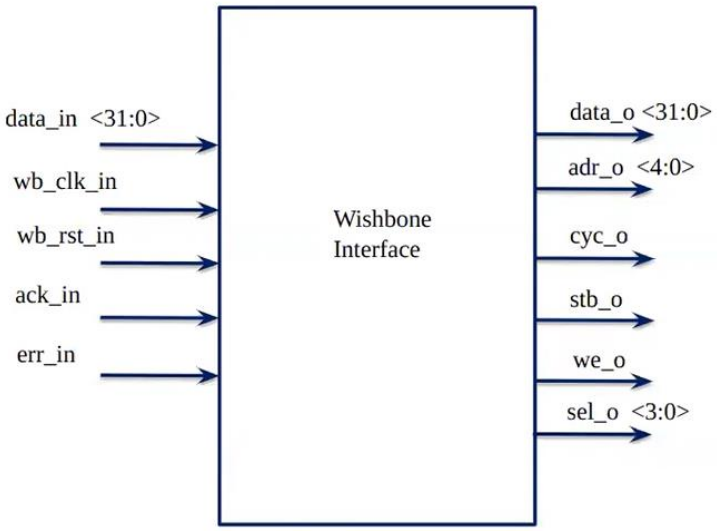
```
  always@(posedge sclk)
  begin
    if ((ss_pad_o != 8'b11111111) && ~rx_slave && tx_slave)
      begin
        temp1 <= {temp1[126:0],mosi};
      end
  end

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

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

SYNTHESIS:


RTL Viewer - C:/intelFPGA_lite/spi_slave/spi_slave - spi_slave
File Edit View Tools Window Help
Search Intel FPGA
Page: 1 of 1
Netlist Navigator
spi_slave
spi_slave:1
mosi
ss_pad_o[7..0]
sclk
miso[2..1]
D
CLK
Q
2'h0
SCLR
2
1
miso
miso

WISHBONE MASTER:

BLOCK DIAGRAM:



DESCRIPTION:

Signal Declarations

- **Inputs**:
    - ∞ clk_in: System clock.
    - ∞ rst_in: Reset signal.
    - ∞ ack_in: Acknowledge signal from the slave indicating the completion of a bus cycle.
    - ∞ dat_in: Data input from the slave.
- **Outputs**:
    - ∞ adr_o: Address output for the Wishbone bus.
    - ∞ cyc_o: Cycle indicator, active during a valid bus cycle.
    - ∞ stb_o: Strobe signal, indicating a valid data transfer cycle.
    - ∞ we_o: Write enable signal, indicating a write operation.
    - ∞ dat_o: Data output to be written to the slave.
    - ∞ sel_o: Byte select signals for the data bus.

Internal Signals

- adr_temp, sel_temp, dat_temp: Temporary storage for address, select lines, and data, respectively.

- we_temp, cyc_temp, stb_temp: Temporary control signals for write enable, cycle, and strobe

Tasks

- **Initialize Task**:
    - ∞ Resets all internal signals (adr_temp, cyc_temp, stb_temp, we_temp, dat_temp, sel_temp) to zero.
    - ∞ Used to set the module to a known starting state.
- **Single Write Task**:
    - ∞ Initiates a single write operation on the Wishbone bus.
    - ∞ Sets the address, data, select lines, and control signals to perform the write.
    - ∞ Waits for an acknowledgment (ack_in) from the slave before clearing the control signals to end the operation.

Signal Assignments

- Uses always blocks to assign values to the output signals based on the internal temporary signals:
    - ∞ adr_o, we_o, dat_o, sel_o are updated on the positive edge of clk_in from their respective temporary signals (adr_temp, we_temp, dat_temp, sel_temp).
    - ∞ cyc_o and stb_o are updated similarly but include reset handling to ensure they are de-asserted during a reset condition (rst_in).

Reset Handling

- Ensures that cyc_o and stb_o signals are de-asserted (set to 0) when the reset signal (rst_in) is active, ensuring proper reset behavior.

The wishbone_master module functions as a master on the Wishbone bus, initiating bus cycles and handling read and write operations. It:

- Controls and monitors the Wishbone bus operation using input and output signals.
- Manages temporary storage for address, data, and control signals.
- Initializes the module state using the initialize task.
- Performs single write operations using the single_write task.
- Uses always blocks to update output signals based on temporary signals, ensuring proper operation and reset handling.

By managing these operations, the wishbone_master module facilitates communication with slave devices on the Wishbone bus, allowing for single read and write operations to be performed efficiently.

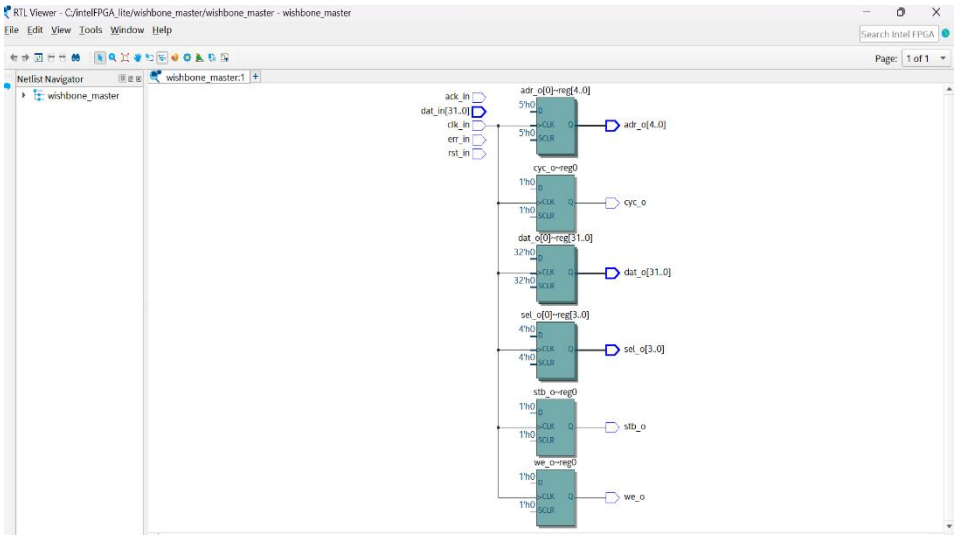
CODE:

```
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);



   integer adr_temp,sel_temp,dat_temp;
   reg we_temp,cyc_temp,stb_temp;
   task initialize;
      begin
         {adr_temp,cyc_temp,stb_temp,we_temp,dat_temp,sel_temp} = 0;
      end
   endtask
   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
```

SYNTHESIS:



SPI TOPMODULE:

DECRIPTION:

Purpose

The spi_topmodule serves as the central control unit for an SPI (Serial Peripheral Interface) Master. It manages clock generation, data transfer, and control for communication with multiple slave devices.

Signal Declarations

- **Inputs**:
    - **Clock and Reset**: wb_clk_in, wb_rst_in
    - **Control and Data Signals**:
        - ∞ wb_adr_in (address)
        - ∞ wb_dat_in (data input)
        - ∞ wb_sel_in (select lines)
        - ∞ wb_we_in (write enable)
        - ∞ wb_stb_in, wb_cyc_in (strobe and cycle signals)
        - ∞ miso (Master In Slave Out, data input from SPI slave)
- **Outputs**:
    - **Control and Data Signals**:
        - ∞ wb_dat_o (data output)
        - ∞ wb_ack_out (acknowledgment)
        - ∞ wb_int_o (interrupt)
    - **SPI Signals**:
        - ∞ ss_pad_o (slave select)
        - ∞ sclk_out (serial clock output)
        - ∞ mosi (Master Out Slave In, data output to SPI slave)

Internal Signals and Registers

- **Registers**:
    - ∞ wb_ack_out, wb_int_o (acknowledgment and interrupt registers)

- ∞ divider (for controlling the clock frequency)
- ∞ ctrl (for configuring various aspects of the SPI interface)
- ∞ ss (for selecting slave devices)
- ∞ wb_temp_dat (temporary data storage)

- **Wires**:

  - ∞ rx_negedge, tx_negedge (control signals for data shifting)
  - ∞ spi_tx_sel (select lines for data transfer)
  - ∞ char_len (character length)
  - ∞ go, ie, ass (control signals for SPI operations)
  - ∞ lsb, cpol_0, cpol_1, last, tip (various control signals)
  - ∞ rx (received data)
  - ∞ spi_divider_sel, spi_ctrl_sel, spi_ss_sel (select signals for different operations)

The spi_topmodule functions as an SPI Master controller, interfacing with a Wishbone bus to manage SPI communication. It handles:

- **Clock Generation**: Using spi_clockgen to generate necessary clock signals.
- **Data Transfer**: Using spi_shift_register to shift data for transmission and reception.
- **Register Access**: Providing read and write access to control, divider, and slave select registers.
- **Control Signals**: Generating control signals (sclk_out, mosi) for SPI communication.
- **Interrupt and Acknowledgment Handling**: Generating interrupts and acknowledgment signals based on bus transactions and SPI operations.
- **Slave Selection**: Managing the selection of up to 32 slave devices.

This module integrates all necessary components to function as a central control unit for SPI communication, ensuring synchronized data transfer, proper clock generation, and efficient slave device selection.

CODE:

```
`include "spi_define.v"

module spi_topmodule(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;



wire rx_negedge;
wire tx_negedge;
wire [3:0] spi_tx_sel;
wire [`SPI_CHAR_LEN_BITS-1:0] char_len;
wire go,ie,ass;
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;
reg  [31:0] wb_temp_dat;
reg  [`SPI_CTRL_BIT_NB-1:0] ctrl;
reg  [`SPI_SS_NB-1:0] ss;


spi_clockgen SC(wb_clk_in,
        wb_rst_in,
        go,
        tip,
        last,
            divider,
        sclk_out,
        cpol_0,
        cpol_1);


spi_shift_register 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);


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));


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


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


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


assign ss_pad_o = ~((ss & {`SPI_SS_NB{tip & ass}}) | (ss & {`SPI_SS_NB{!ass}}));


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



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
```

TESTBENCH:

```
`include "spi_define.v"

module spi_testbench;

  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_topmodule 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
```

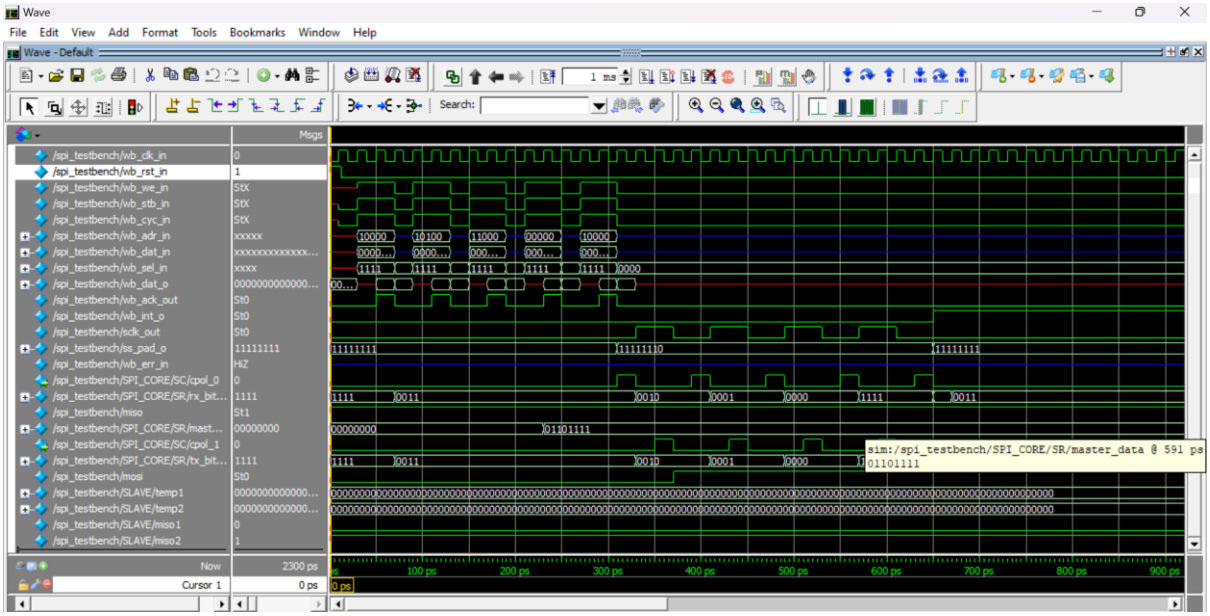
```
begin
   wb_clk_in = 1'b0;
   forever
      #(T/2) wb_clk_in = ~wb_clk_in;
end

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;
   MASTER.initialize;
   MASTER.single_write(5'h10,32'h0000_3c04,4'b1111);
   MASTER.single_write(5'h14,32'h0000_0004,4'b1111);
   MASTER.single_write(5'h18,32'h0000_0001,4'b1111);
   MASTER.single_write(5'h00,32'h0000_236f,4'b1111);
   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;
   MASTER.initialize;
```

```
    MASTER.single_write(5'h10,32'h0000_3404,4'b1111);
    MASTER.single_write(5'h14,32'h0000_0004,4'b1111);
    MASTER.single_write(5'h18,32'h0000_0001,4'b1111);
    MASTER.single_write(5'h00,32'h0000_236f,4'b1111);
    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;
    MASTER.initialize;
    MASTER.single_write(5'h10,32'h0000_3A04,4'b1111);
    MASTER.single_write(5'h14,32'h0000_0004,4'b1111);
    MASTER.single_write(5'h18,32'h0000_0001,4'b1111);
     MASTER.single_write(5'h00,32'h0000_236f,4'b1111);
    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=1, char_len=4
initial
begin
  rst;
  MASTER.initialize;
  MASTER.single_write(5'h10, 32'h0000_3A00, 4'b1111); // LSB=0
  MASTER.single_write(5'h14, 32'h0000_0004, 4'b1111); // char_len=4
  MASTER.single_write(5'h18, 32'h0000_0001, 4'b1111); // rx_neg=1
```

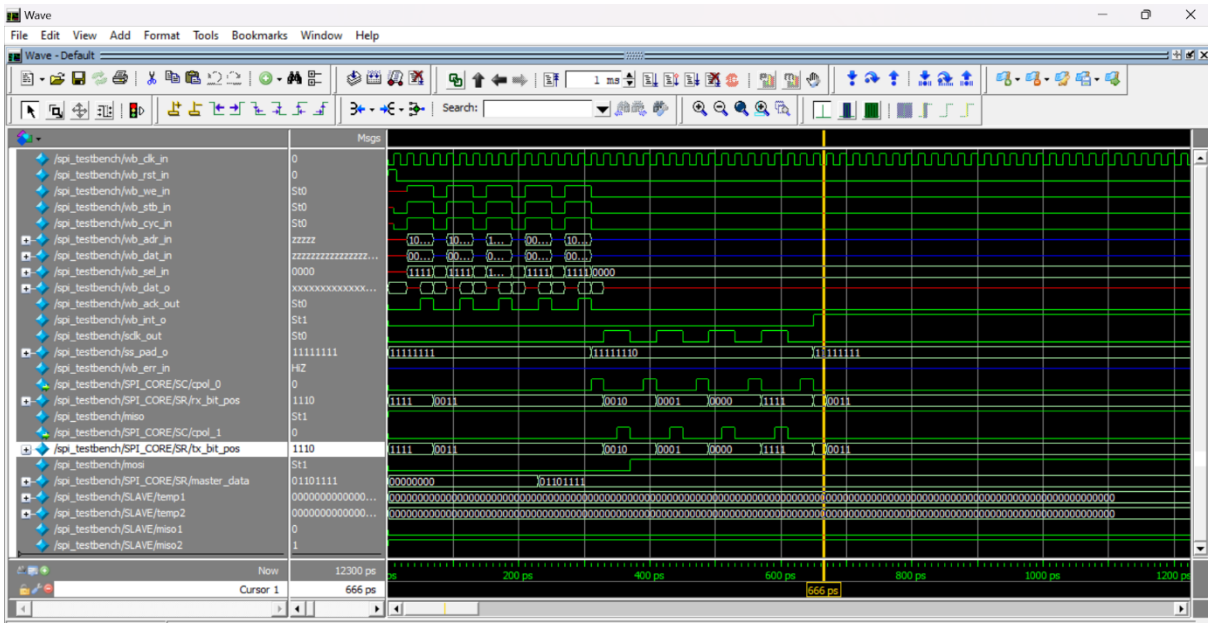
```
    MASTER.single_write(5'h00, 32'h0000_236f, 4'b1111); // tx_neg=0
    MASTER.single_write(5'h10, 32'h0000_3B00, 4'b1111); // LSB=0
    repeat(100)
        @(negedge wb_clk_in);
    $finish;
    #10000 $finish;
end

endmodule
```

WAVEFORM:

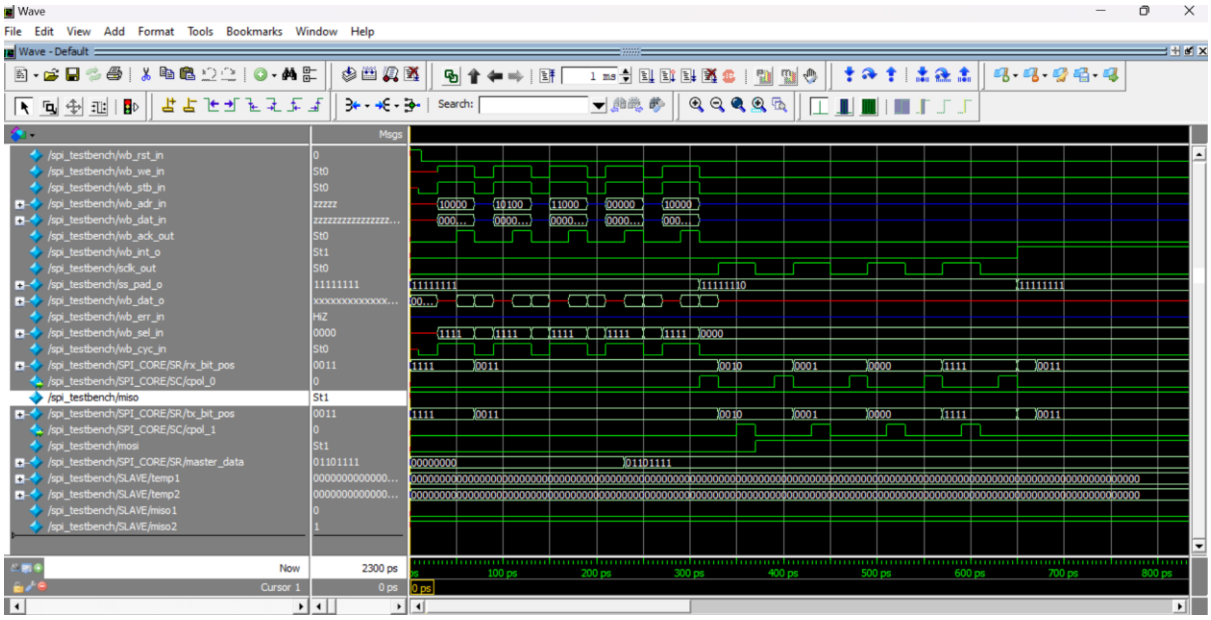
TX_NEG = 1, RX_NEG = 0, LSB = 1, CHAR_LEN = 4:



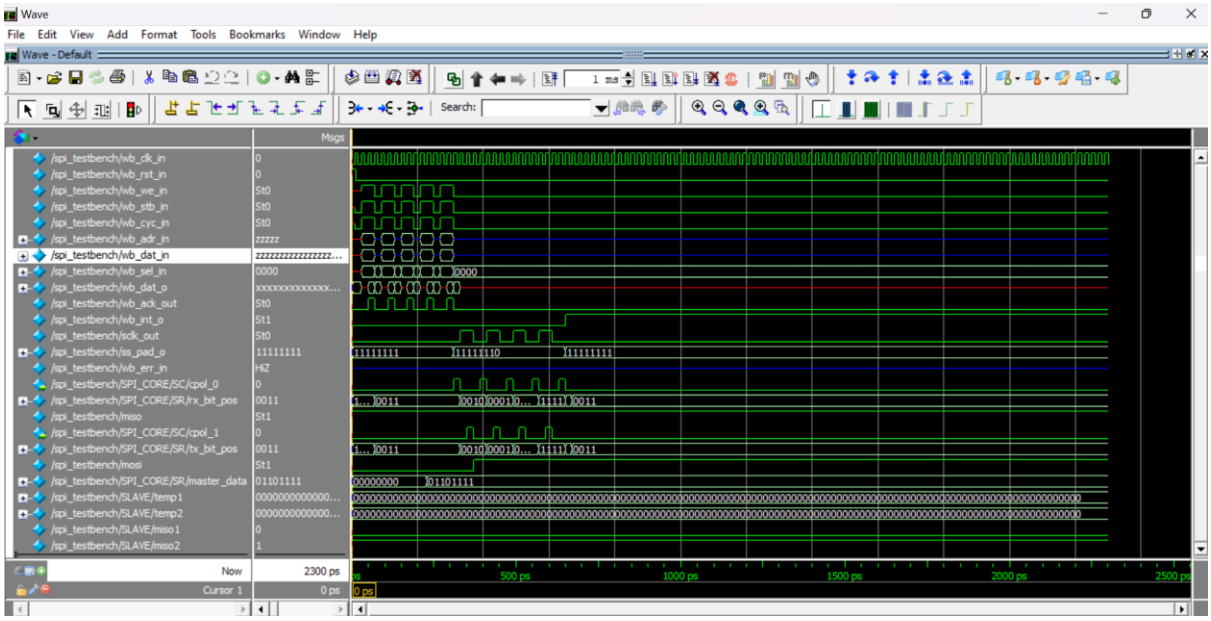
TX_NEG = 1, RX_NEG = 0, LSB = 0, CHAR_LEN = 4 :



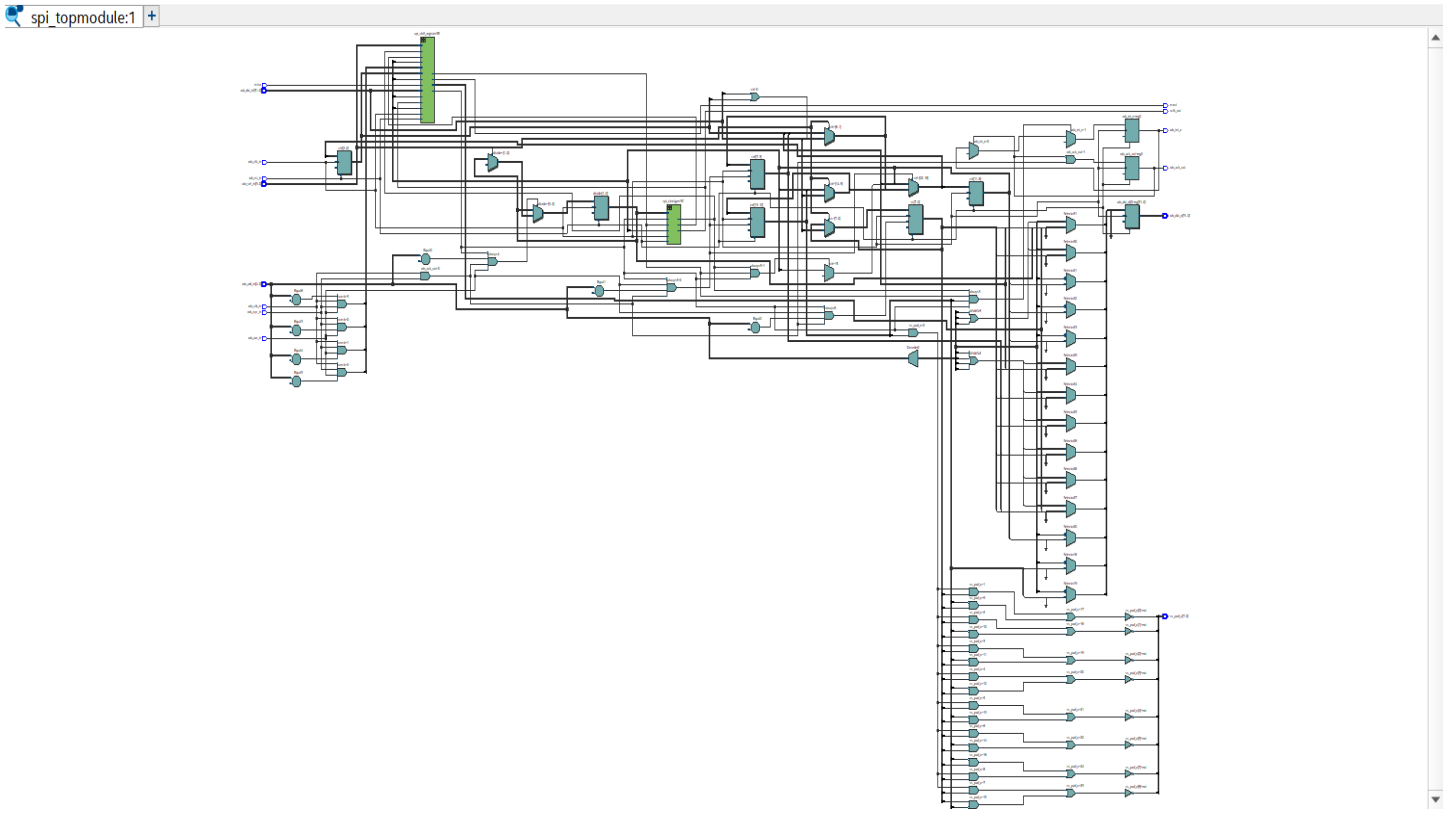
TX_NEG = 0, RX_NEG = 1, LSB = 1, CHAR_LEN = 4:



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

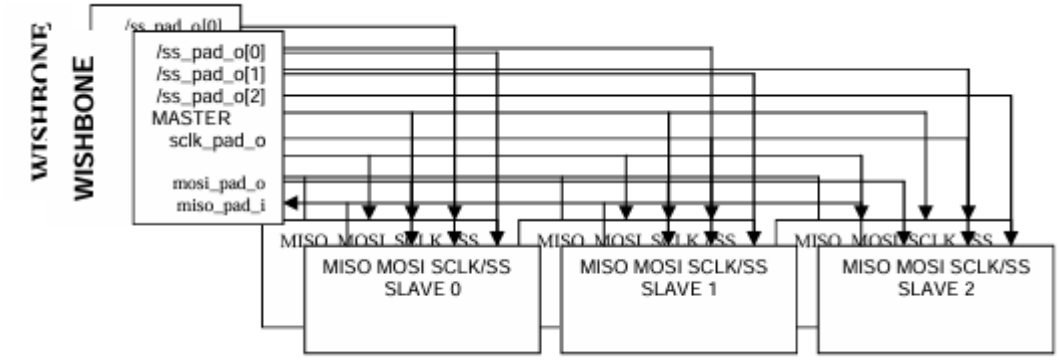


SYNTHESIS:



OPERATION:

This core is an SPI and Microwire/Plus compliant synchronous serial controller. At the host side, it is controlled via registers accessible through a WISHBONE rev. B1 interface.

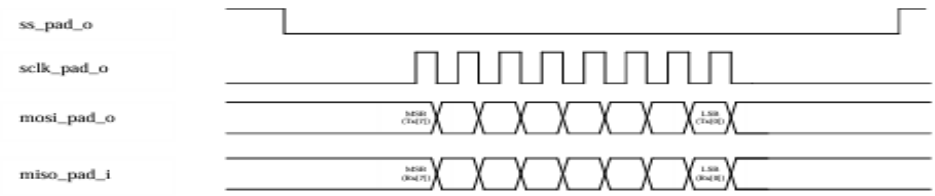


WISHBONE INTERFACE:

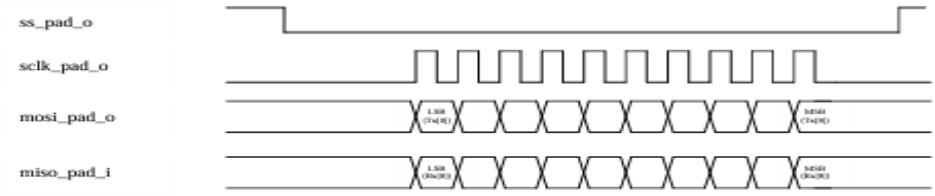
The SPI core has five 32-bit registers through the WISHBONE rev. B1 compatible interface. All accesses to SPI registers must be 32-bit (wb_sel[3:0] = 0xf).

SERIAL INTERFACE:

The serial interface consists of slave select lines, serial clock lines, as well as input and output data lines. All transfers are full duplex transfers of a programmable number of bits per transfer (up to 64 bits). Compared to the SPI/Microwire protocol, this core has some additional functionality. It can drive data to the output data line in respect to the falling (SPI/Microwire compliant) or rising edge of the serial clock, and it can latch data on an input data line on the rising (SPI/Microwire compliant) or falling edge of a serial clock line. It also can transmit (receive) the MSB first (SPI/Microwire compliant) or the LSB first. It is important to know that the RxX and TxX registers share the same flip-flops, which means that what is received from the input data line in one transfer will be transmitted on the output data line in the next transfer if no write access to the TxX register is executed between the transfers.



CTRL[LSB] = 0, CTRL[CHAR_LEN] = 0x08, CTRL[TX_NEG] = 1, CTRL[RX_NEG] = 0



CTRL[LSB] = 1, CTRL[CHAR_LEN] = 0x0a, CTRL[TX_NEG] = 0, CTRL[RX_NEG] = 1

CONCLUSION:

The design and implementation of the SPI Master Core in Verilog was successful, allowing us to incorporate the SPI protocol and develop a fully functional SPI Master Core for interfacing with various devices like microcontrollers, DACs, and ADCs.

Throughout the project, we gained valuable experience in creating and implementing digital circuits using Verilog, and furthered our understanding of the applications of the SPI protocol. We also explored synchronous serial protocols such as I2C and Microwire/Plus, comparing their features to SPI.

A notable discovery from the project was the simplicity and efficiency of the SPI protocol.

In comparison to other protocols, it has certain limitations, such as requiring a separate SS line for each slave device, but it remains a popular choice for many applications. It was also discovered that the I2C protocol serves as a dependable alternative to the SPI protocol, featuring built-in error detection and correction mechanisms.

Another important observation from the project was the significance of testing and verification in the design and implementation process. Through the use of simulation tools and testbenches, we were able to ensure that the SPI Master Core functioned correctly under various conditions and scenarios.

In conclusion, this project offered valuable exposure to the design and implementation of digital circuits using Verilog, as well as a deeper comprehension of the SPI protocol and its diverse applications.

THANK YOU