Serial Peripheral Interface (SPI)

Implementation using Verilog HDL

Report

Table of Contents

Table of Figures

Introduction

Serial Peripheral Interface (SPI) is a widely used synchronous serial communication protocol, commonly used in embedded systems to connect microcontrollers with peripheral devices such as sensors, memory chips and displays unlike asynchronous protocols SPI operates with a shared clock signal enabling faster and more reliable data transfer.

SPI is valued for its simplicity and high speed performance compared to other serial protocols while I2C supports multi master configurations and address based communication it trades off speed and complexity UART is asynchronous and good for long distance data exchange but lacks SPI s simultaneous full duplex transmission SPI s 4 wire design (MOSI, MISO, SCLK and CS) makes it efficient for short range and high speed data exchange where low latency is essential.

SPI play a significant role in system on chip architectures where integration of processors, memory, and peripherals requires fast and predictable communication interfaces. SPI is used to connect digital sensors, external flash memory, and configuration interfaces within SOCs especially where minimal overhead and timing are critical.

The objective is to design and simulate a verilog based SPI system consisting of both master and slave modules. through this implementation, we aim to reinforce understanding of digital design principles including finite state machines (FSM), timing control, and serial data exchange. by simulating the design in vivado the functionality of the system can be verified under realistic hardware conditions.

Technical Background (SPI)

The Serial Peripheral Interface (SPI) enables fast and full-duplex data exchange between a master device and one or more slave devices. Its widely adopted in digital systems where reliability, speed, and low communication are essential.

SPI communication is based on 4 signals :

MOSI (Master Out, Slave In) carries data from the master to the slave.

MISO (Master In, Slave Out) carries data from the slave to the master.

SCLK (Serial Clock) generated by the master to synchronize the data transmission.

CS (Chip Select) activated by the master to select which slave should respond.

the master starts communication by pulling the CS line low, activating the target slave. Then using the clock signal (SCLK) it controls the timing of data exchange sending bits via MOSI and simultaneously receiving bits via MISO. Each transfer is typically 8 bits long. (other sizes are possible depending on the implementation).

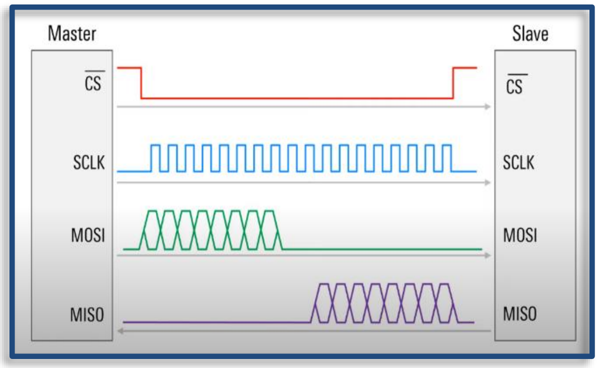


Figure 1

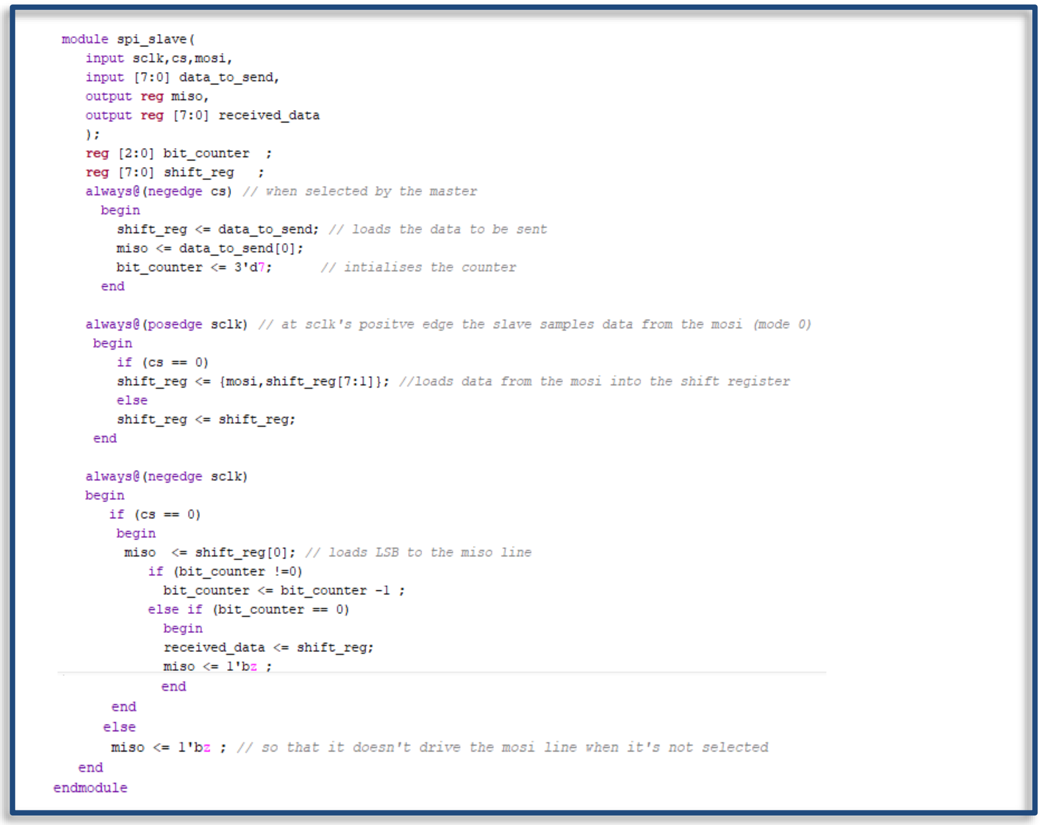
SPI operates using different clocking modes, determined by clock polarity (CPOL) and clock phase (CPHA). These modes define whether data is sampled or changed on rising or falling edges of SCLK. **Mode 0** is used In this project, the clock idles low, data changes on the falling edge, and is sampled on the rising edge.

Verilog Implementation

The design contains two main components the **SPI Master module** which controls the transmission process and the **SPI Slave module** which receives and processes incoming data. each module is designed to follow standard SPI protocol characteristics such as clock synchronization, chip select control, and bit data transfer.

1) SPI Slave Module

The slave module is responsible for capturing incoming bits on the mosi line and assembling them into a full byte. it uses a shift register and counts the number of bits received the slave also sends data back to the master on the miso line responding to clock edges following the SPI protocol.



```
module spi_slave(
   input sclk,cs,mosi,
   input [7:0] data_to_send,
   output reg miso,
   output reg [7:0] received_data
   );
   reg [2:0] bit_counter  ;
   reg [7:0] shift_reg   ;
   always@(negedge cs) // when selected by the master
     begin
       shift_reg <= data_to_send; // loads the data to be sent
       miso <= data_to_send[0];
       bit_counter <= 3'd7;      // intialises the counter
     end

   always@(posedge sclk) // at sclk's positve edge the slave samples data from the mosi (mode 0)
    begin
       if (cs == 0)
       shift_reg <= {mosi,shift_reg[7:1]}; //loads data from the mosi into the shift register
       else
       shift_reg <= shift_reg;
    end

   always@(negedge sclk)
   begin
      if (cs == 0)
       begin
        miso  <= shift_reg[0]; // loads LSB to the miso line
           if (bit_counter !=0)
             bit_counter <= bit_counter -1 ;
           else if (bit_counter == 0)
             begin
             received_data <= shift_reg;
             miso <= 1'bz ;
             end
      end
     else
      miso <= 1'bz ; // so that it doesn't drive the mosi line when it's not selected
  end
endmodule
```

Figure 2

2) SPI Master Module

The master module is a finite state machine that generates the serial clock (sclk) and activates the chip select signal (cs) then transmits data over the mosi line. It also reads incoming data from the miso line.

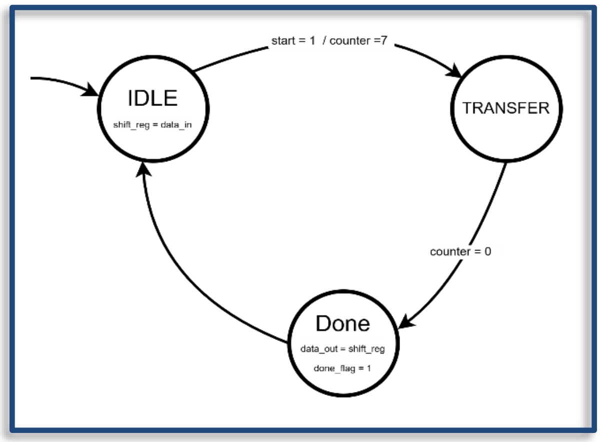
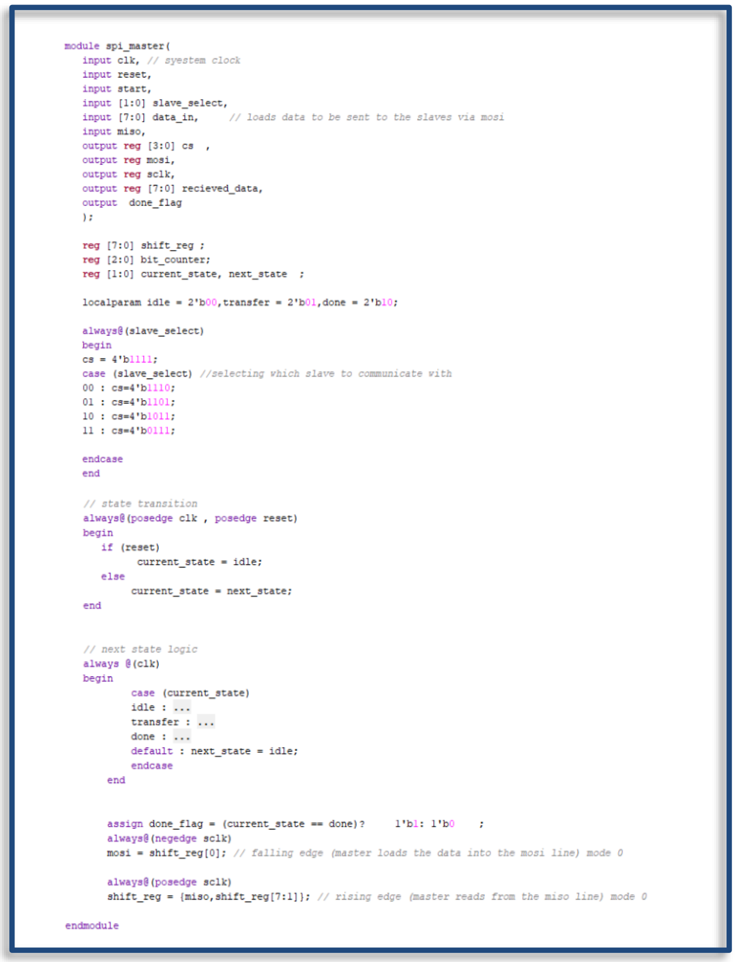


Figure 3



```
module spi_master(
   input clk, // syestem clock
   input reset,
   input start,
   input [1:0] slave_select,
   input [7:0] data_in,     // loads data to be sent to the slaves via mosi
   input miso,
   output reg [3:0] cs  ,
   output reg mosi,
   output reg sclk,
   output reg [7:0] recieved_data,
   output  done_flag
   );

   reg [7:0] shift_reg ;
   reg [2:0] bit_counter;
   reg [1:0] current_state, next_state  ;

   localparam idle = 2'b00,transfer = 2'b01,done = 2'b10;

   always@(slave_select)
   begin
   cs = 4'b1111;
   case (slave_select) //selecting which slave to communicate with
   00 : cs=4'b1110;
   01 : cs=4'b1101;
   10 : cs=4'b1011;
   11 : cs=4'b0111;

   endcase
   end

   // state transition
   always@(posedge clk , posedge reset)
   begin
      if (reset)
            current_state = idle;
      else
           current_state = next_state;
   end


   // next state logic
   always @(clk)
   begin
           case (current_state)
           idle : ...
           transfer : ...
           done : ...
           default : next_state = idle;
           endcase
       end


       assign done_flag = (current_state == done)?     1'b1: 1'b0     ;
       always@(negedge sclk)
       mosi = shift_reg[0]; // falling edge (master loads the data into the mosi line) mode 0

       always@(posedge sclk)
       shift_reg = {miso,shift_reg[7:1]}; // rising edge (master reads from the miso line) mode 0

endmodule
```

Figure 4

Simulation & Testbench

To verify the functionality of the SPI system a testbench module spi_tb was made. The simulation was targeting the verification of signal flow between the master and slave modules, the timing of transactionsand the data exchange.

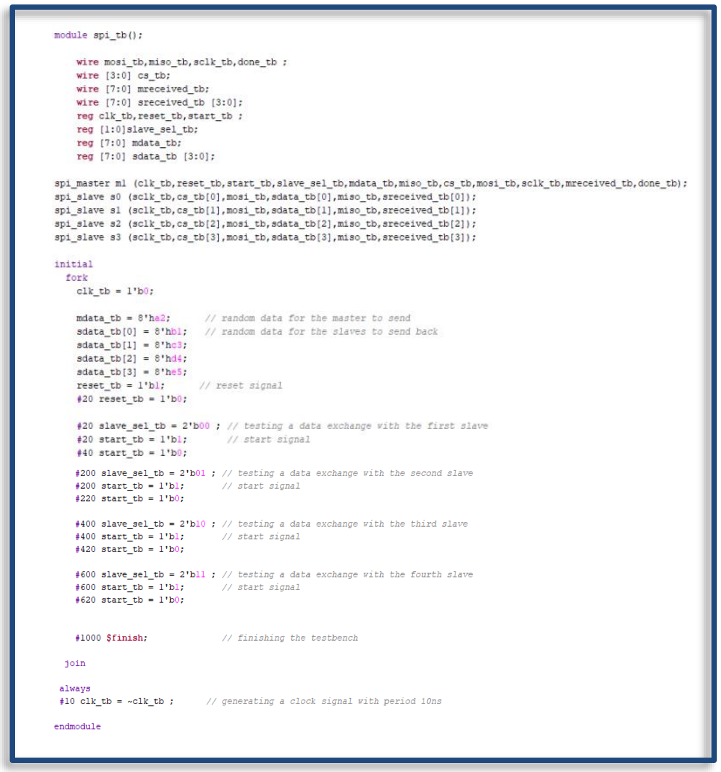
Testbench:

The testbench simulates a basic SPI data exchanges between the master and a selected slave.

Simulation Steps

1. **Reset and Initialization**
   - The system is reset for 20ns
   - clk_tb toggles every 10ns to create a 100MHz clock
2. **Starting exchange**
   - start_tb is used to start the SPI communication
   - Slave 0 is selected by the slave_sel_tb
   - Data bytes are loaded to mdata_tb and sdata_tb
3. **Data Transfer**
   - SPI master transmitss mdata_tb via mosi
   - Slave samples bits on rising/falling edges of sclk
   - Slave responds via miso with sdata_tb
4. **Output**
   - Master captures received byte in mreceived_tb
   - Slaves outputs received byte in sreceived_tb
   - done_tb indicates transaction completion
   - then the data exchange is repeated with another slave

Testbench Code



```
module spi_tb();

    wire mosi_tb,miso_tb,sclk_tb,done_tb ;
    wire [3:0] cs_tb;
    wire [7:0] mreceived_tb;
    wire [7:0] sreceived_tb [3:0];
    reg clk_tb,reset_tb,start_tb ;
    reg [1:0]slave_sel_tb;
    reg [7:0] mdata_tb;
    reg [7:0] sdata_tb [3:0];

spi_master m1 (clk_tb,reset_tb,start_tb,slave_sel_tb,mdata_tb,miso_tb,cs_tb,mosi_tb,sclk_tb,mreceived_tb,done_tb);
spi_slave s0 (sclk_tb,cs_tb[0],mosi_tb,sdata_tb[0],miso_tb,sreceived_tb[0]);
spi_slave s1 (sclk_tb,cs_tb[1],mosi_tb,sdata_tb[1],miso_tb,sreceived_tb[1]);
spi_slave s2 (sclk_tb,cs_tb[2],mosi_tb,sdata_tb[2],miso_tb,sreceived_tb[2]);
spi_slave s3 (sclk_tb,cs_tb[3],mosi_tb,sdata_tb[3],miso_tb,sreceived_tb[3]);

initial
  fork
    clk_tb = 1'b0;

    mdata_tb = 8'ha2;      // random data for the master to send
    sdata_tb[0] = 8'hb1;   // random data for the slaves to send back
    sdata_tb[1] = 8'hc3;
    sdata_tb[2] = 8'hd4;
    sdata_tb[3] = 8'he5;
    reset_tb = 1'b1;      // reset signal
    #20 reset_tb = 1'b0;

    #20 slave_sel_tb = 2'b00 ; // testing a data exchange with the first slave
    #20 start_tb = 1'b1;       // start signal
    #40 start_tb = 1'b0;

    #200 slave_sel_tb = 2'b01 ; // testing a data exchange with the second slave
    #200 start_tb = 1'b1;       // start signal
    #220 start_tb = 1'b0;

    #400 slave_sel_tb = 2'b10 ; // testing a data exchange with the third slave
    #400 start_tb = 1'b1;       // start signal
    #420 start_tb = 1'b0;

    #600 slave_sel_tb = 2'b11 ; // testing a data exchange with the fourth slave
    #600 start_tb = 1'b1;       // start signal
    #620 start_tb = 1'b0;


    #1000 $finish;             // finishing the testbench

  join

 always
 #10 clk_tb = ~clk_tb ;      // generating a clock signal with period 10ns

endmodule
```

Figure 5

Top Level Schematic

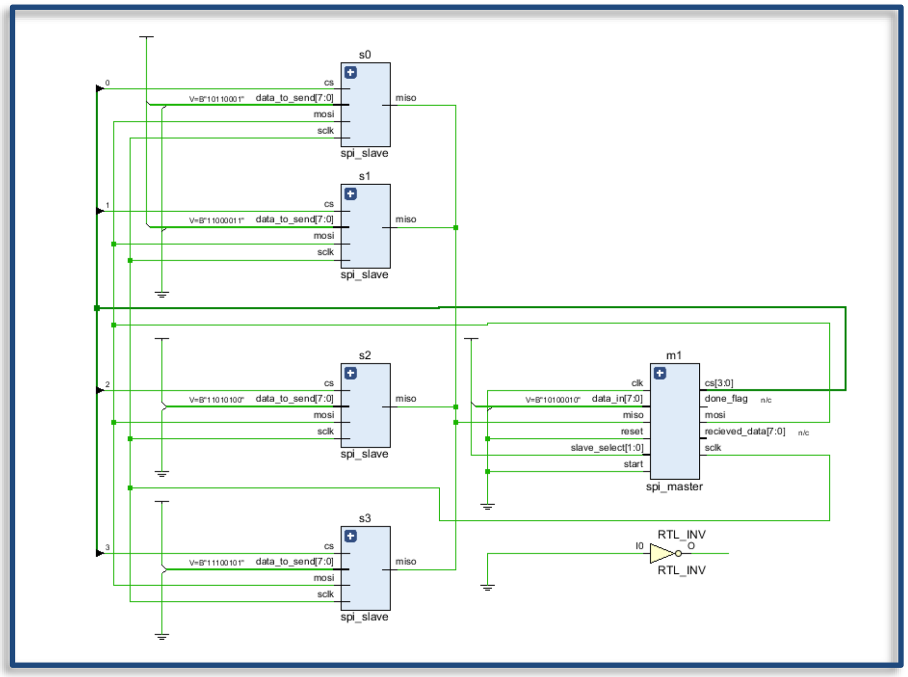


Figure 6

Results

The SPI communication system was simulated using the spi_tb testbench. the master and the slaves modules performed synchronized data exchange over the mosi and miso lines by the system clock (clk_tb) and serial clock (sclk_tb). The results waveform:

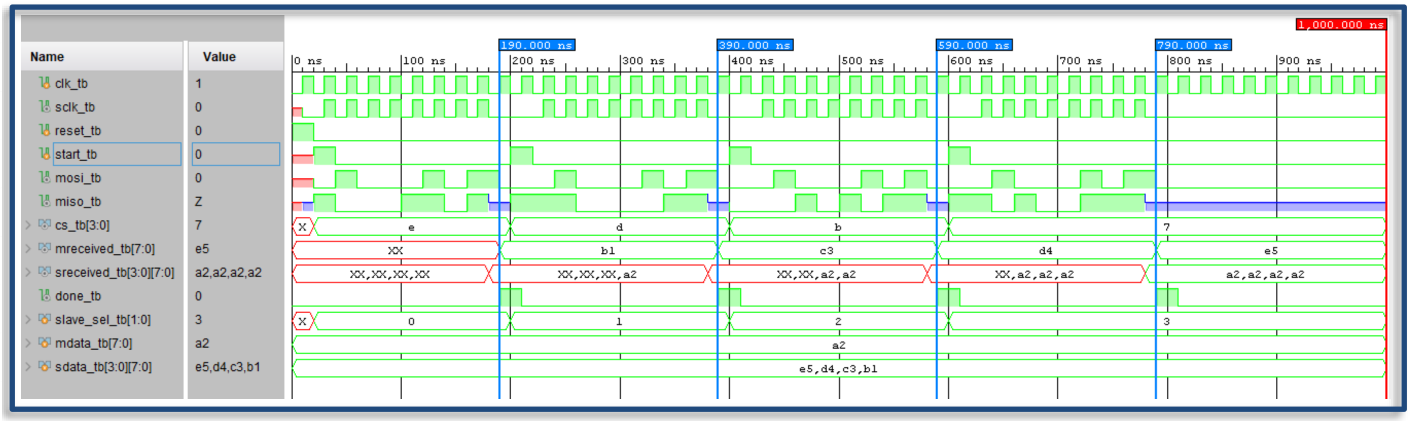


Figure 7

These results confirrm correct bit shifting and proper data capture by the master (mreceived_tb) and slaves (sreceived_tb) and module followed SPI mode 0 behavior.

RTL Schematics

Slave's schematic

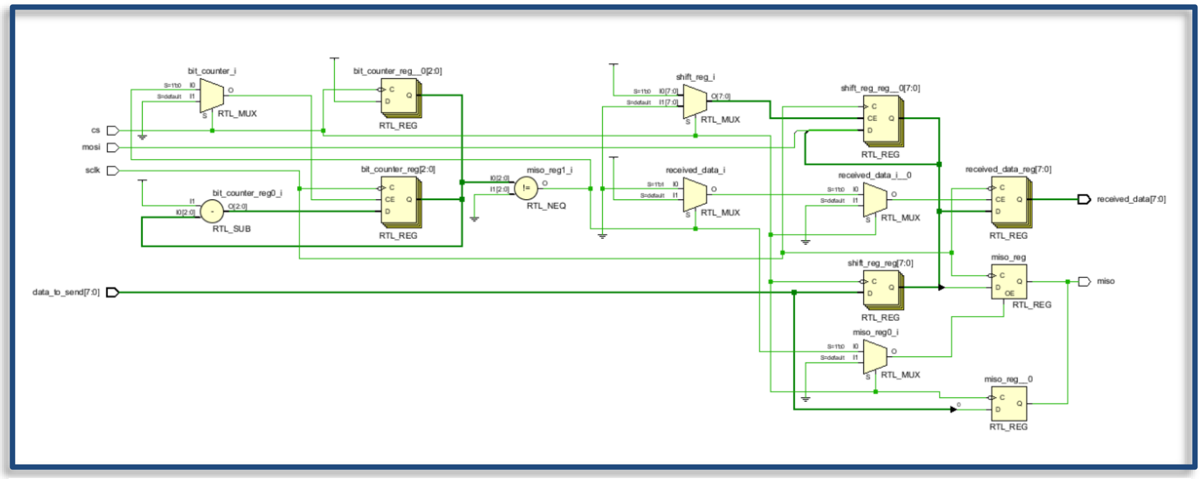


Figure 8

Master's schematic

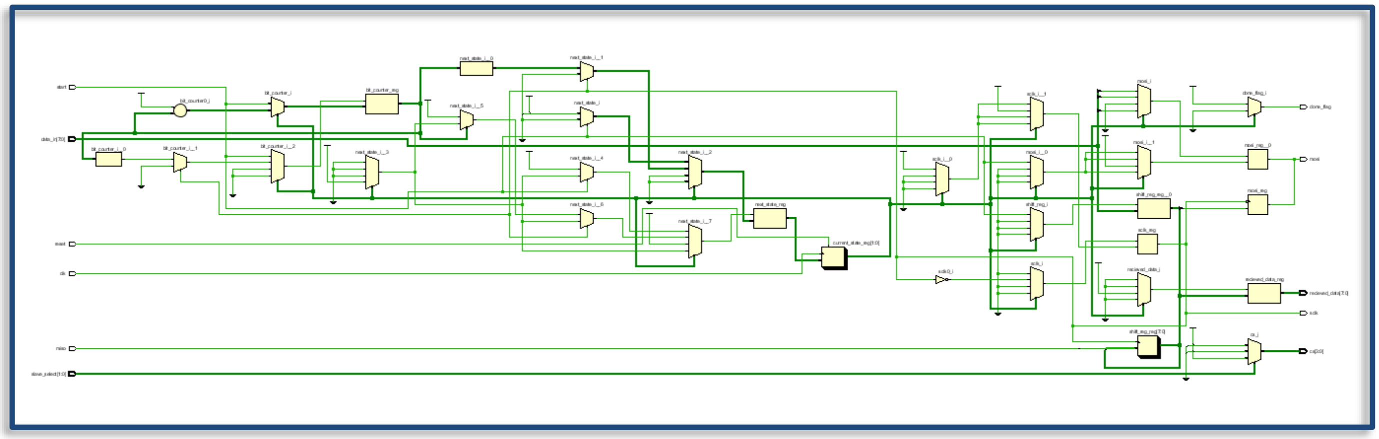


Figure 9

Conclusion

This project built and tested an SPI communication system using Verilog. the results showed that the master and slave modules worked well together sending and receiving data with accurate timing that followed standard SPI rules. The design used finite state machines and clear module connections, making it easy to follow and reliable.

The system can handle multiple slave devices thanks to the chip select feature. It can also be improved with error detection like checking for missing data or timing problems to make it even more dependable.

even though it follows standard SPI behaviorr the code was written from scratch. This gave full control over how it works and makes it flexible for upgrades or use in bigger systems.

References

- All About Circuits (SPI Protocol Explained) Link
- Brock J. Lameres (Quick Start Guide to Verilog) Link