EE 214 Project

Task 1 REPORT

VHDL Code Development for SPI Master and SPI Slave

Team Members :

1. Name : Revanth Manepalli Roll No. : 23B1223
2. Name : Manasvi Kadam Roll No. : 23B1300

Introduction

This report presents the design and implementation of a complete Serial Peripheral Interface (SPI) communication system consisting of an SPI Master and an SPI Slave. SPI is a synchronous serial communication protocol widely used for short-distance communication between devices. This implementation aims to facilitate effective data transmission and reception between the Master and Slave components.

Design Objectives

The primary objectives of the SPI Master and Slave design include:

- **Data Transmission and Reception**: The Master sends data bit-wise to the Slave via MOSI and simultaneously receives data from it via MISO.
- **Synchronous Operation**: The system utilizes a clock signal to synchronize data transfer between the Master and Slave.
- **Control Signals**: Manage Chip Select (CS), Master Out Slave In (MOSI), Master In Slave Out (MISO), and Serial Clock (SCLK) signals.
- **Reset Capability**: Provide a mechanism to reset the system to a known state.

VHDL Implementation:

SPI Master

1) Entity Declaration

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity spi_master is
   port(clk,reset,start,miso: in std_logic;
         sclk,mosi,cs: out std_logic;
         data_out: out std_logic_vector(7 downto 0));
end entity;
```

The SPI Master entity defines the input and output ports:

- **Inputs**:
  - clk: System clock.
  - reset: Asynchronous reset signal.
  - start: Signal to initiate the SPI transaction.
  - miso: Master In Slave Out signal from the slave device.
- **Outputs**:
  - sclk: Serial clock signal to synchronize data transmission.
  - mosi: Master Out Slave In signal to send data to the slave.
  - cs: Chip Select signal to enable the slave device.
  - data_out: Output register to hold received data from the slave.

2) Architecture :

```
architecture behaviour of spi_master is
    signal master_data: std_logic_vector(7 downto 0) := "00000101";
    signal bit_count: integer;

    signal busy     : std_logic := '0';              -- state of transmission
    signal temp_data : std_logic_vector(7 downto 0);
begin
    process(clk, reset)
    begin
        if reset = '1' then                          -- Master goes back to initial state
            busy <= '0';
            cs <= '1';
            mosi <= '0';
            bit_count <= 0;
            temp_data <= (others => '0');

        elsif rising_edge(clk) then
            if start = '1' and busy = '0' then
                busy <= '1';
                cs <= '0';
                bit_count <= 7;
            elsif busy = '1' then

                mosi <= master_data(7);                    -- master send MSB through MOSI
                master_data <= master_data(6 downto 0) & '0';   -- data is shifted so that next bit is sent in following cycle

                temp_data <= temp_data(6 downto 0) & miso;      -- Master stores the data received from slave via miso

                if bit_count = 0 then                      -- transmission stopped after 8 bits
                    busy <= '0';
                    cs <= '1';
                else
                    bit_count <= bit_count - 1;            -- Bit count decreases with each bit transmitted and received
                end if;

            end if;
        end if;
    end process;
    sclk <= clk when (busy = '1') else '0';
    data_out <= temp_data;
end architecture;
```

The behavior of the SPI Master is controlled by a clocked process triggered on the rising edge of the clock (clk) signal and managed by a reset signal (reset). The process handles the following actions:

- **Reset State**: If the reset signal is active, all signals return to their initial state. Communication is disabled, the CS signal is set high (disabling the Slave), and the bit_count is set to 0.
- **Start Condition**: When the start signal is asserted, the Master initiates comunication, setting the busy signal high and lowering the cs signal to enable the Slave. The bit_count is initialized to 7, representing the 8 bits of data to be transmitted(from 7 to 0).
- **Data Transmission**: While busy is high, the Master sends the most significant bit (MSB) of master_data via the mosi line. Simultaneously, the Master shifts in the incoming bit from the Slave on the miso line, storing it in temp_data.
- **Transmission Completion**: When all bits have been transmitted (bit_count = 0), the Master stops communication by setting busy to '0', raising cs to '1', and completing the transaction.

SPI SLAVE

1) Entity Declaration

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity spi_slave is
   port(clk,reset,sclk,cs,mosi: in std_logic;
         miso: out std_logic;
         data_out: out std_logic_vector(7 downto 0));
end entity;
```

The spi_slave entity defines the necessary I/O ports for SPI communication:

- **Inputs**:
  - clk: System clock.
  - reset: Asynchronous reset signal.
  - start: Signal to initiate the SPI transaction.
  - mosi: Master out Slave in signal from the master device.
- **Outputs**:
  - miso: Master in Slave Out signal to send data to the master.
  - data_out: Output register to hold received data from the master

2) Architecture

```vhdl
architecture behaviour of spi_slave is
    signal slave_data : std_logic_vector(7 downto 0) := "00000111";  -- Data to send to master(7)
    signal bit_count : integer;
    signal temp_data : std_logic_vector(7 downto 0);  |
begin
    process(sclk, reset)
    begin
        if reset = '1' then                              -- Slave goes to initial state
            temp_data <= (others => '0');
            bit_count <= 0;
            miso <= '0';
        elsif rising_edge(sclk) then

                temp_data <= temp_data(6 downto 0) & mosi;   -- Temporarily stores data from master
                bit_count <= bit_count;

        elsif falling_edge(sclk) then
            if cs = '0' then
                miso <= slave_data(7);                   -- slave sends data to master through MISO
                slave_data <= slave_data(6 downto 0) & '0';
            end if;
        end if;
    end process;
    data_out <= temp_data;
end architecture;
```

The spi_slave module uses a process triggered on both the rising and falling edges of the SPI clock (sclk). The rising edge handles data reception, while the falling edge manages data transmission.

1. **Reset State**: When reset is '1', the Slave returns to its initial state, clearing internal data and outputs.
2. **Data Reception** (Rising Edge of sclk):
   - The Slave captures the data sent from the Master on the mosi line and shifts it into temp_data.
3. **Data Transmission** (Falling Edge of sclk):
   - The Slave transmits the most significant bit (MSB) of slave_data via the miso line and shifts the remaining data bits left.
4. **Completion of Data Exchange**: The Slave continues to shift and send/receive data on each clock cycle until all 8 bits are exchanged. The received data is output as data_out.

Top-level Entity:

Connects the Master and Slave, managing data flow and control signals

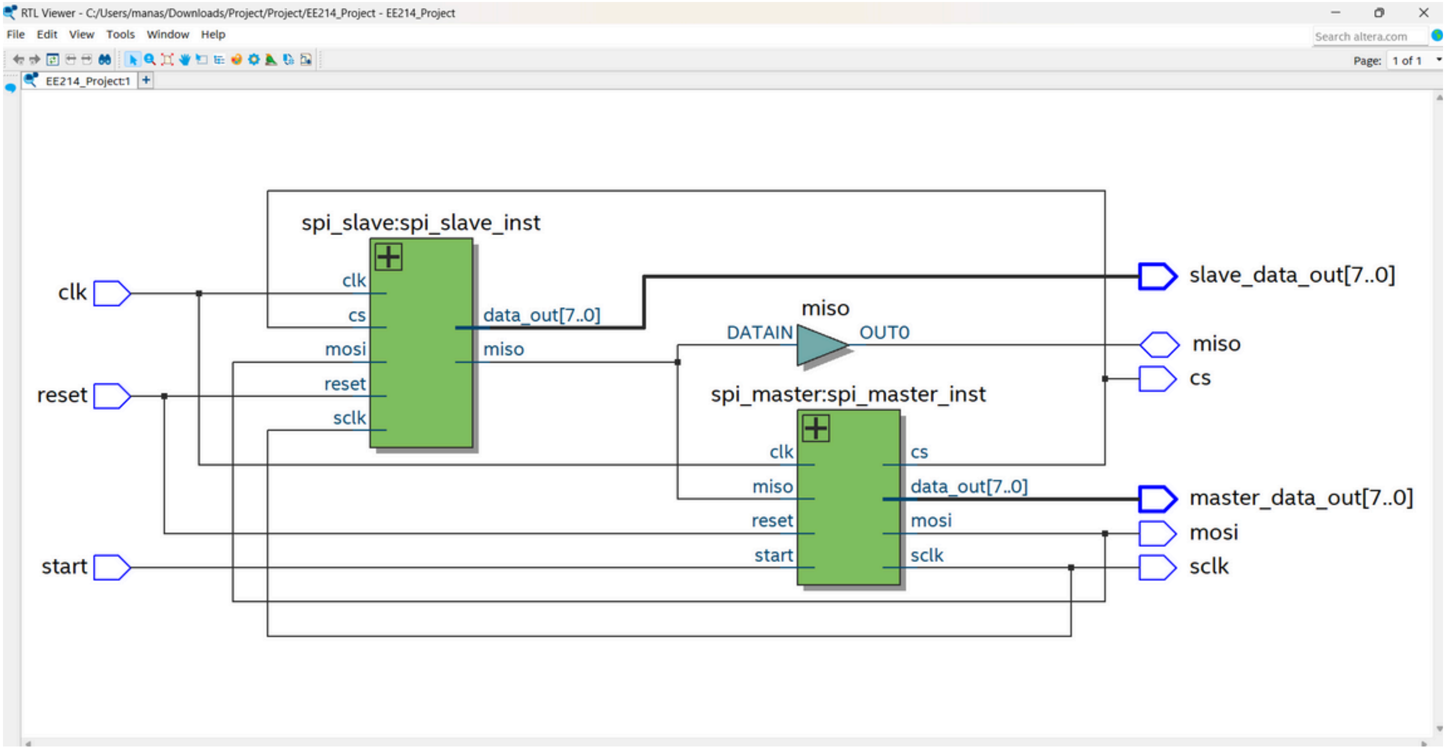
Testbench :

Generates a clock signal and provides stimulus to the Master to initiate data transmission.

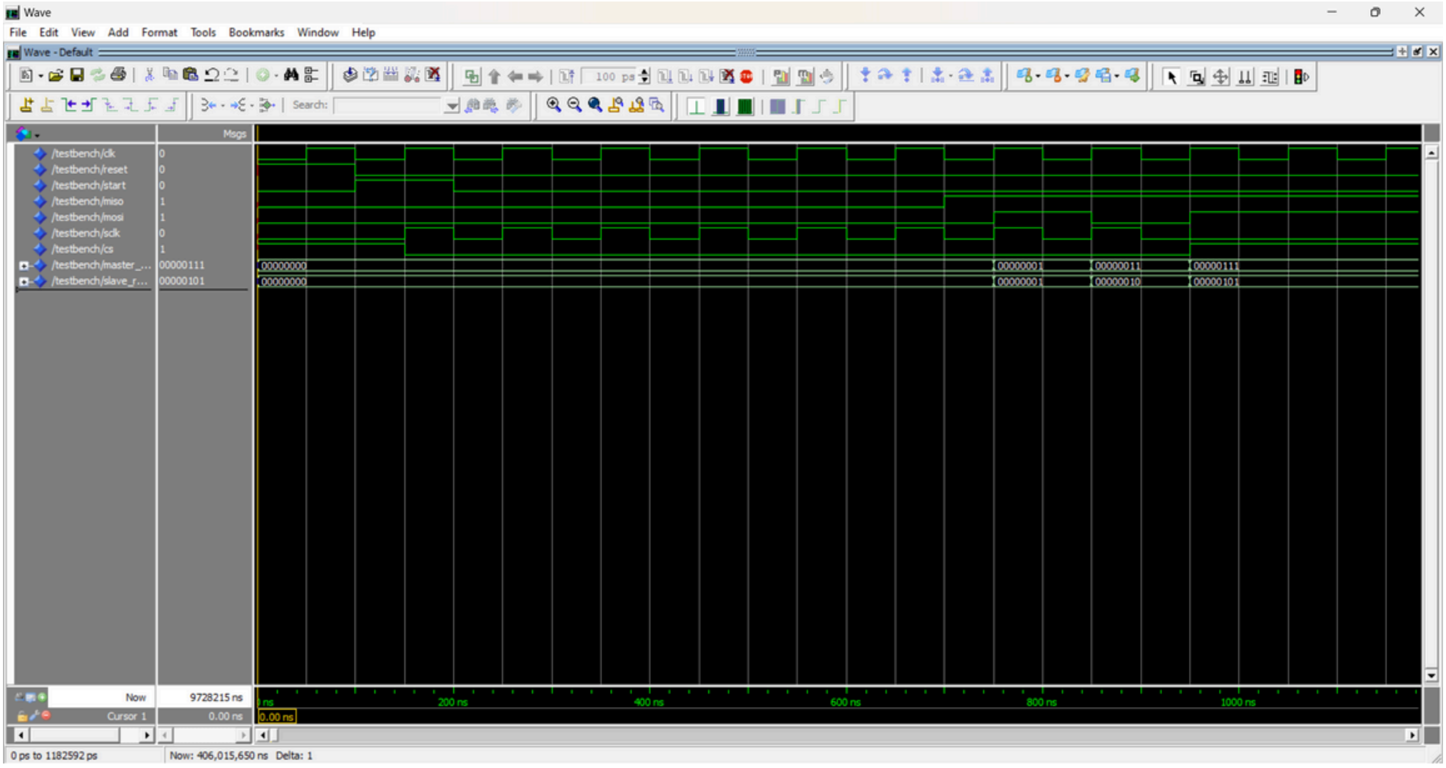
Monitors outputs to validate the communication between the Master and Slave.

Simulation

<u>RTL Netlist View:</u>



Waveform :



Breakdown of work done by each member:

After discussion about the problem statement and the approach to solve it, the work was divided in the following way:

1. Revanth :
   - Designed SPI Master and the Top-Level Entity
2. Manasvi :
   - Designed SPI Slave and Testbench

Simulation and Debugging was done by both the members.

This project was a collaborative effort with both members discussing and designing their respective components to create a functional SPI communication system.