PROJECT REPORT

SIMULATION AND VERFICATION OF SPI PROTOCOL USING VERILOG

Submitted to

AP IIIT RAJIV GANDHI UNIVERSITY OF KNOWLEDGE TECHNOLOGIES RK VALLEY, KADAPA

Partial fulfillment of the requirements for the award of the Degree of

BACHELOR OF TECHNOLOGY IN ELECTRONICS AND COMMUNICATION ENGINEERING

Submitted by:

Ch.Pranai T.ObulSai K.SreeBadrinath

Under the Guidance of:
P.Janardhana
Assistant Professor of ECE

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

RAJIV GANDHI UNIVERSITY OF KNOWLEDGE TECHNOLOGIES

RKVALLEY,Vempalli(M),Kadapa(D),AndhraPradesh(S),516330

2024-2025

RAJIV GANDHI UNIVERSITY OF KNOWLEDGE TECHNOLOGIES
RK VALLEY ,KADAPA 516330

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

DECLARATION

We hereby declare that the project report entitled “**A SIMULATION AND VERFICATION OF SPI PROTOCOL USING VERILOG** ” submitted to the Department of **ELECTRONICS AND COMMUNICATION ENGINEERING** in partial fulfillment of requirements for the award of the degree of **BACHELOR OF TECHNOLOGY**. This project is result of our own effort and that it has not been Submitted to any other University or Institution for the award of any degree or diploma other than specified above.

TABLE OF CONTENTS

ABSTRACT

SIMULATION AND VERFICATION OF SPI PROTOCOL USING VERILOG

The **Serial Peripheral Interface (SPI)** protocol is a high-speed, full-duplex, synchronous serial communication interface used for data exchange between a microcontroller (master) and one or more peripheral devices (slaves). Developed by Motorola in the 1980s, SPI has since become a de facto standard for interfacing low- to medium-speed devices such as sensors, memory chips, SD cards, and display controllers.

SPI operates on a master-slave architecture and utilizes four main lines: **MOSI (Master Out Slave In)**, **MISO (Master In Slave Out)**, **SCLK (Serial Clock)**, and **SS/CS (Slave Select or Chip Select)**. The master device generates the clock signal and initiates communication by asserting the appropriate slave select line. Data is simultaneously transmitted and received on the MOSI and MISO lines, respectively, with synchronization provided by the clock signal on SCLK.

The protocol is characterized by its **simplicity, low pin count, and high data transfer rates**, typically ranging from hundreds of kHz to tens of MHz. SPI supports multiple slaves via individual chip select lines or through daisy-chaining configurations. However, it lacks formal standards for device addressing and error checking, which must be implemented at the application level if required.

SPI supports different **clock polarity (CPOL)** and **clock phase (CPHA)** configurations, allowing compatibility with a wide range of devices. These parameters define the relationship between data and clock signals, enabling flexible timing options suited for various hardware requirements.

Despite its advantages in speed and simplicity, SPI is generally used for short-range communication due to the lack of built-in flow control or error correction. It is less suitable than protocols like I²C or UART for multi-device communication in complex systems, but its performance and efficiency make it ideal for fast, point-to-point data exchange.

In summary, the SPI protocol is a robust and efficient method for high-speed serial communication between digital devices, offering simplicity and performance for applications where data integrity and timing can be managed at the system level.

CHAPTER 1

INTRODUCTION

The **Serial Peripheral Interface (SPI)** is a widely used synchronous serial communication protocol that enables high-speed data exchange between a **master** device and one or more **slave** devices. Originally developed by **Motorola** in the 1980s, SPI has since become a standard interface for embedded systems and microcontroller applications due to its simplicity, speed, and versatility.

SPI operates in **full-duplex mode**, meaning that data can be transmitted and received simultaneously. Unlike asynchronous communication protocols like UART, SPI relies on a **shared clock signal** generated by the master device, ensuring synchronized communication. This makes it suitable for time-critical applications and devices requiring fast and reliable data transfers.

The SPI bus typically consists of four lines:

- **MOSI (Master Out Slave In)**: Transmits data from the master to the slave.
- **MISO (Master In Slave Out)**: Transmits data from the slave to the master.
- **SCLK (Serial Clock)**: Clock signal generated by the master to synchronize data transmission.
- **SS or CS (Slave Select or Chip Select)**: Used by the master to select which slave device to communicate with. This line is active low.

SPI supports multiple slaves through either **individual chip select lines** or **daisy-chaining**, though it does not include built-in addressing or arbitration features like I^{2}C. Communication is initiated and controlled by the master, which selects the slave and controls the timing of the data exchange. SPI can be configured in four modes, determined by the **clock polarity (CPOL)** and **clock phase (CPHA)** settings, providing flexibility in data sampling and clock edge alignment.

While SPI offers significant advantages in terms of speed (often operating at frequencies of 10 MHz or higher) and simplicity, it does have limitations. It requires more physical wiring compared to some other protocols and does not support multi-master configurations or long-distance communication. Furthermore, it lacks built-in error detection or correction, which must be implemented separately if needed.

Despite these trade-offs, SPI remains a **popular and efficient solution** for interfacing microcontrollers with a wide range of peripherals such as sensors, memory devices, display controllers, and communication modules.

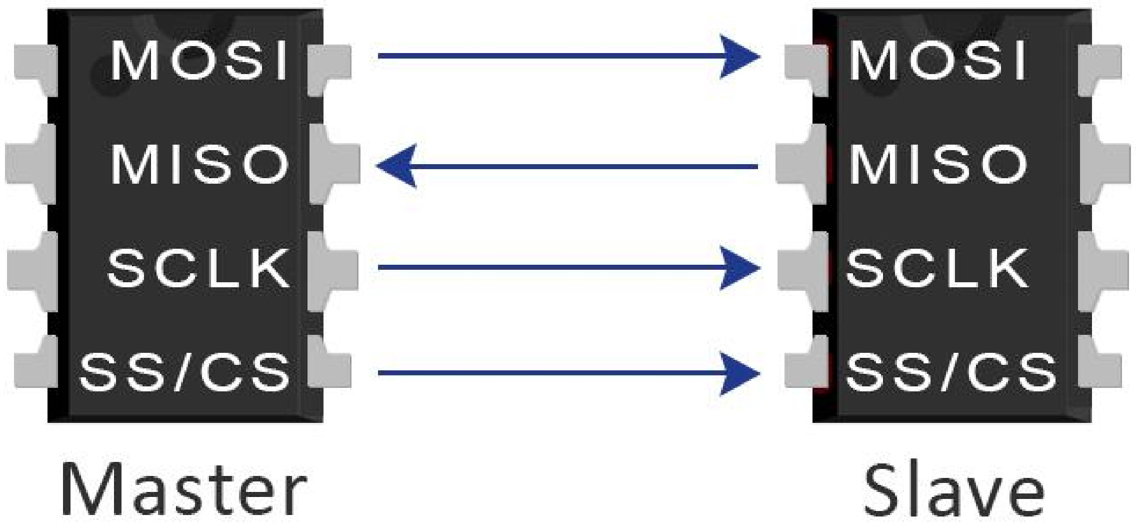
CHAPTER 2

INFORMATION ABOUT SPI

1. What is SPI?

The **Serial Peripheral Interface (SPI)** is a **synchronous, full-duplex, master-slave communication protocol** used for short-range communication between microcontrollers and peripheral devices. It is commonly used in embedded systems to connect components such as sensors, displays, memory chips, and ADC/DAC converters.

Developed by **Motorola** in the mid-1980s, SPI has become a standard due to its **high speed, ease of use, and flexibility**.



2. SPI Bus Architecture:

The SPI bus typically consists of **four signal lines**:

Signal	Full Form	Direction	Description
MOSI	Master Out Slave In	Master → Slave	Transmits data from master to slave
MISO	Master In Slave Out	Slave → Master	Transmits data from slave to master
SCLK	Serial Clock	Master → Slave	Clock signal generated by master
SS/CS	Slave Select / Chip Select	Master → Slave	Used to select which slave to communicate with

- The **master** initiates all communication.
- Multiple slaves can be connected, each having its own **CS (Chip Select)** line.
- Data is shifted simultaneously from master to slave (MOSI) and from slave to master (MISO).

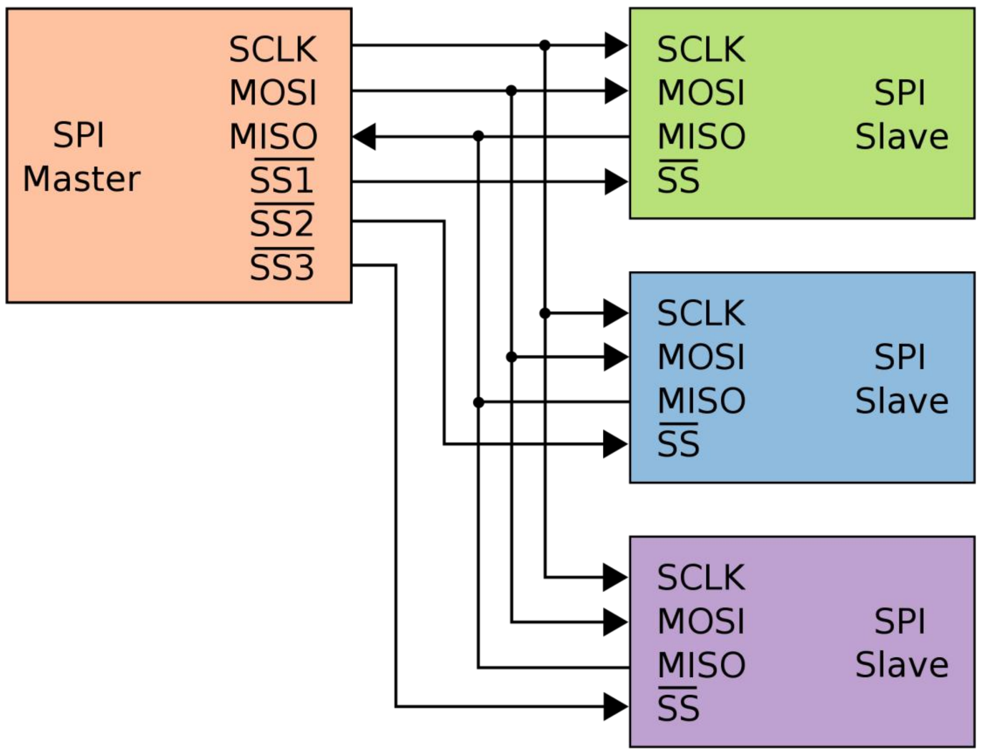
3. SPI Operation:

Full-Duplex Data Transfer:

- SPI transfers **one bit per clock cycle**.
- While the master sends data on the MOSI line, the slave simultaneously sends data back on the MISO line.
- This makes SPI **full-duplex**, unlike I²C and UART which are half-duplex or simplex.

Clock Synchronization:

- The master controls the **SCLK** signal.
- Data is sampled and shifted according to the clock signal based on two configuration parameters:
-
    - **CPOL (Clock Polarity)**: Determines the idle state of the clock (0 = low, 1 = high).
    -



    -
    - **CPHA (Clock Phase)**: Determines when data is sampled (first or second clock edge).

4. SPI Modes:

SPI has **four communication modes** based on CPOL and CPHA values:

Mode	CPOL	CPHA	Clock Idle	Data Sampled
0	0	0	Low	Rising Edge
1	0	1	Low	Falling Edge
2	1	0	High	Falling Edge
3	1	1	High	Rising Edge

The master and slave **must be configured in the same mode** to ensure correct data exchange.

5. Multi-Slave Communication:

SPI supports multiple slaves in two configurations:

1. **Independent Chip Select Lines**:
   - Each slave has a separate **CS pin** connected to the master.
   - The master selects one slave at a time by pulling its CS pin **low**.
2. **Daisy-Chaining**:
   - Slaves are connected in series.
   - Data from the master is shifted through each slave and passed on.
   - Used less commonly due to complexity.

6. Advantages of SPI:

- **High Speed**: Can operate at frequencies >10 MHz (much faster than I²C or UART).
- **Full-Duplex Communication**: Simultaneous data transmission and reception.
- **Simple Protocol**: Easy to implement in hardware and software.
- **Flexible Configuration**: Adjustable clock polarity and phase.
- **No Arbitration Needed**: No bus conflicts like in I²C.

7. Limitations of SPI:

- **No Standard Device Addressing**: Each slave requires a dedicated CS line.
- **No Acknowledgment or Error Checking**: No built-in mechanism to detect or correct errors.
- **Wiring Overhead**: Requires 4 wires (plus one CS per slave), which increases with more devices.
- **Single Master Only**: SPI doesn't support multi-master configurations easily.

- **Short Distance**: Not suitable for long-distance data transmission.

8. Applications of SPI:

SPI is widely used in embedded systems for connecting:

- **Flash memory chips** (EEPROM, NOR/NAND)
- **LCDs and OLED displays**
- **Real-time clocks (RTC)**
- **Analog-to-Digital Converters (ADC)**
- **Digital-to-Analog Converters (DAC)**
- **SD and micro SD cards**
- **Sensors (temperature, accelerometer, etc.)**

9. SPI vs Other Protocols:

Feature	SPI	I²C	UART
Communication	Full-Duplex	Half-Duplex	Full/Half Duplex
Speed	High (10+ MHz)	Medium (up to 3.4 MHz)	Medium (up to 1 Mbps)
Pins Required	4 + n (for CS)	2	2
Addressing	No	Yes	No
Multi-Master	Difficult	Supported	Not Supported
Error Checking	No	Yes (ACK/NACK)	Basic (Parity)

10. Summary:

The **SPI protocol** is a powerful and efficient communication method for high-speed, short-range device communication. It is best suited for applications where **speed, simplicity**, and **low latency** are more important than scalability and error checking. Despite its limitations in multi-device setups, SPI remains a backbone protocol in modern embedded design due to its performance and flexibility.

CHAPTER 3

INFOEMATION ON MOSI AND MISO

MASTER OUT SLAVE IN:

What is MOSI?

MOSI (Master Out Slave In) is a **unidirectional** data line used in the **SPI (Serial Peripheral Interface)** protocol. It **transmits data from the master device to the slave device**. In Verilog, this signal is typically declared as an **output** in the SPI master module and as an **input** in the SPI slave module.

Key Functions of MOSI in SPI:

1. **Data Line from Master to Slave**
   - Sends serial data bit-by-bit from the master.
   - Typically used to send commands, addresses, or data to peripherals (e.g., sensors, memory, displays).
2. **Synchronized by Clock (SCLK)**
   - The data on MOSI is **synchronized with the SPI clock (SCLK)**.
   - Depending on the SPI mode (CPOL and CPHA), the data is:
     - **Launched** by the master on one clock edge.
     - **Captured** by the slave on the opposite edge.
3. **Uses a Shift Register Mechanism**
   - The master usually holds the data in a shift register.
   - On every valid clock edge, one bit is shifted out to MOSI.
   - Most implementations send the **Most Significant Bit (MSB) first**, though LSB-first modes also exist.

Design Considerations for MOSI in Verilog:

- **Signal Direction**: In the SPI master, MOSI is an `output`; in the slave, it is an `input`.
- **Clock Dependency**: The correct value must appear on MOSI **before the relevant clock edge** based on CPHA/CPOL.
- **Data Stability**: The data must be **stable during the clock edge** when the slave reads it.
- **SPI Mode Awareness**: The logic that drives MOSI must match the SPI mode to ensure proper timing (e.g., Mode 0: CPOL=0, CPHA=0).
- **Shift Timing**: The master shifts out the next data bit on MOSI either:
  - On the leading edge (CPHA = 0), or

- On the trailing edge (CPHA = 1) of the SPI clock.

Multi-Slave Systems:

In systems with multiple slaves:

- The **MOSI line is shared** among all slaves.
- Only the **selected slave (via CS/SS line)** listens to the MOSI line.
- Unselected slaves **ignore** the data being transmitted.

MASTER IN SLAVE OUT:

What is MISO?

MISO (Master In Slave Out) is a **unidirectional data line** in the **SPI protocol** used to **transmit data from the slave device to the master**. In a Verilog SPI design:

- MISO is typically declared as an **input** in the **master module**.
- It is declared as an **output** in the **slave module**.

Key Functions of MISO in SPI:

1. **Slave-to-Master Data Transfer**
   - MISO carries data **from the slave back to the master**, such as sensor readings, memory contents, or status bytes.
   - It operates **synchronously with the SPI clock (SCLK)** generated by the master.
2. **Used During Full-Duplex Communication**
   - While the master sends data on **MOSI**, it **simultaneously receives** data on **MISO**.
   - The data on MISO is shifted out bit-by-bit in sync with the SPI clock.
3. **Shift Register Based**
   - The slave uses a **shift register** to prepare and output its response data.
   - On every valid clock edge, a bit from the slave is placed onto the MISO line.
   - Like MOSI, the **MSB is usually sent first**, unless LSB-first mode is used.

Design Considerations for MISO in Verilog:

- **Signal Direction**:
  - MISO is an **input** in the master Verilog module.

  - MISO is an **output** in the slave Verilog module.
- **Timing and Synchronization**:
  - The slave must **output the correct bit on MISO before the master samples it**.
  - The timing depends on the SPI mode (CPOL and CPHA):
    - CPHA = 0: Data must be valid before the clock rising edge.
    - CPHA = 1: Data is output on the first clock edge and captured on the next.
- **Tri-state or High-Z Handling (for multiple slaves)**:
  - Only the **active slave** (selected by its CS line) should drive the MISO line.
  - **Unselected slaves must place MISO in a high-impedance (Z) state** to avoid bus contention.
  - This is typically implemented using **conditional assignments or tri-state buffers** in Verilog.
- **Data Preparation**:
  - The slave must **load its response data** into a shift register before or during the transmission initiated by the master.
  - This ensures the correct data is available on MISO at the right time.

<u>Multi-Slave Considerations:</u>

- In a system with multiple SPI slaves:
  - The **MISO line is shared** among all slaves.
  - Only the **selected slave** (via its active-low CS) should drive the MISO line.
  - **Unselected slaves must leave MISO disconnected or in high-Z** to avoid electrical conflicts.

CHAPTER 4

VERILOG CODE AND WAVEFORMS

1. MASTER-OUT-SLAVE-IN:

```
module spi_master(
   input wire clk,
   input wire reset,
   input wire tx_enable,
   input wire miso,
   output reg mosi,
   output reg sclk,
   output reg chip_select,
   output reg [7:0] rx_data  // received from slave
);

// FSM states
parameter idle = 2'b00, tx_start = 2'b01, tx_data = 2'b10, tx_end =
2'b11;

reg [1:0] state, next_state;
reg [2:0] count = 0;
reg [3:0] bit_count = 0;
reg [7:0] send_data = 8'd181;

// state update
always @(posedge clk) begin
   if (reset)
      state <= idle;
   else
      state <= next_state;
end

// count and bit_count
always @(posedge clk) begin
   case (state)
      idle: begin
```

```
            count <= 0;
            bit_count <= 0;
        end

        tx_start: begin
            count <= count + 1;
            bit_count <= 0;
        end

        tx_data: begin
            if (bit_count < 8) begin
                if (count == 7) begin
                    bit_count <= bit_count + 1;
                    count <= 0;
                end else
                    count <= count + 1;
            end
        end

        tx_end: begin
            bit_count <= 0;
            count <= count + 1;
        end
    endcase
end

// generate sclk
always @(posedge clk) begin
    case (next_state)
        idle: sclk <= 0;
        tx_start, tx_data: sclk <= (count < 3 || count == 7) ? 1 : 0;
        tx_end: sclk <= (count < 3) ? 1 : 0;
    endcase
end

// FSM transitions and output logic
always @(*) begin
    case (state)
```

```
        idle: begin
            chip_select <= 1;
            mosi <= 0;
            next_state <= tx_enable ? tx_start : idle;
        end

        tx_start: begin
            mosi <= 0;
            chip_select <= 0;
            next_state <= (count == 7) ? tx_data : tx_start;
        end

        tx_data: begin
            mosi <= send_data[7 - bit_count];
            if (bit_count < 8)
                next_state <= tx_data;
            else begin
                next_state <= tx_end;
                mosi <= 0;
            end
        end

        tx_end: begin
            mosi <= 0;
            chip_select <= 1;
            next_state <= (count == 7) ? idle : tx_end;
        end
    endcase
end

// sample miso (receive data)
always @(posedge clk) begin
    if (state == tx_data && count == 3) begin
        rx_data <= {rx_data[6:0], miso};  // MSB first
    end
end

endmodule
```

2. MASTER-IN-SLAVE-OUT:

```
 module spi_slave(
    input wire mosi,
    input wire sclk,
    input wire chip_select,
    input wire [7:0] fixed_data,
    output wire [7:0] data_out,
    output reg done,
    output reg miso
);

reg [7:0] data = 0;
reg [3:0] bit_count = 0;
reg [7:0] shift_out;
parameter idle = 1'b0, sample = 1'b1;
reg state;

always @(negedge sclk) begin
    case (state)
        idle: begin
            done = 0;
            if (chip_select)
                state <= idle;
            else begin
                state <= sample;
                shift_out <= fixed_data;  // preload outgoing data
            end
        end

        sample: begin
            if (bit_count < 8) begin
                data <= {data[6:0], mosi};  // shift in mosi
                miso <= shift_out[7 - bit_count];  // shift out miso
                bit_count <= bit_count + 1;
                state <= sample;
            end else begin
```

```
                state <= idle;
                done <= 1;
                bit_count <= 0;
                data <= 0;
            end
        end

        default: state <= idle;
    endcase
end

assign data_out = data;

endmodule
    endmodule
```

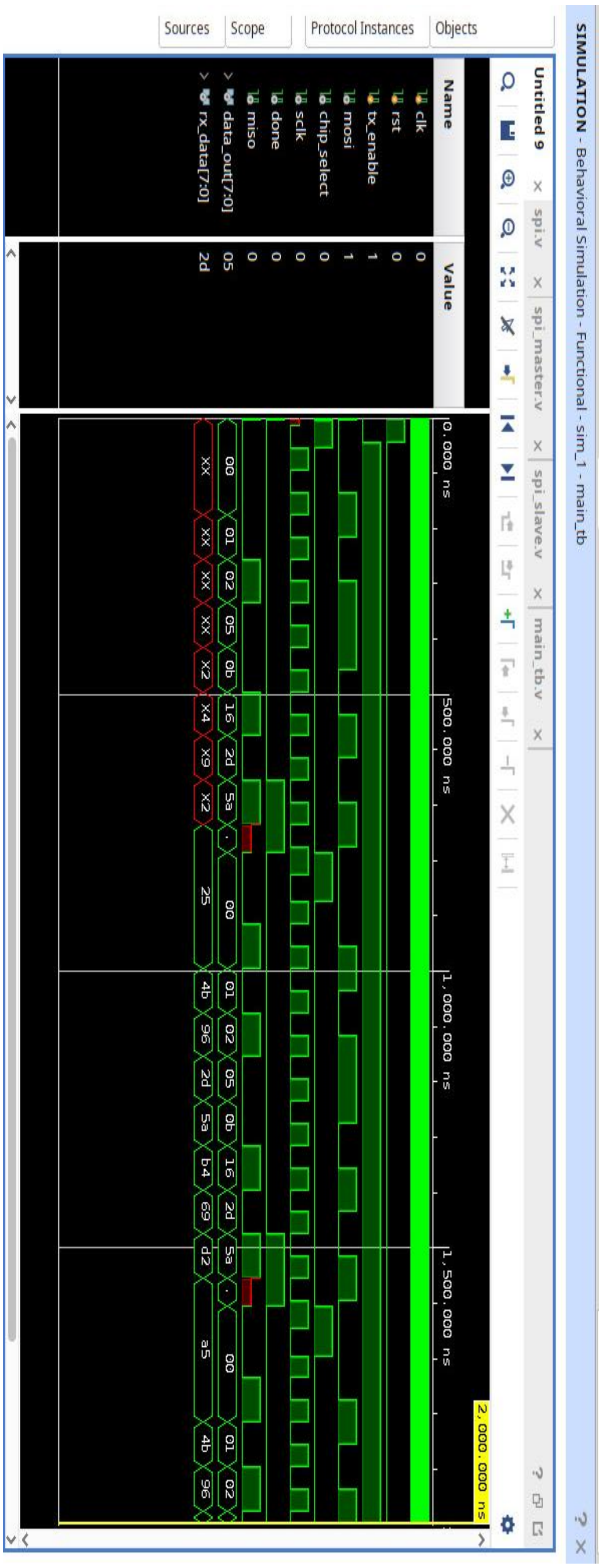
MAIN CODE:

```
module spi (
    input clk,
    input reset,
    input tx_enable,
    output done,
    output [7:0] data_out,
    output [7:0] rx_data,
    output sclk,
    output chip_select,
    output mosi,
    output miso
);

spi_master spi_m (
    .clk(clk), .reset(reset), .tx_enable(tx_enable),
    .miso(miso), .mosi(mosi), .sclk(sclk),
    .chip_select(chip_select), .rx_data(rx_data)
);

spi_slave spi_s (
    .sclk(sclk), .chip_select(chip_select), .mosi(mosi),
    .done(done), .data_out(data_out), .fixed_data(8'hA5),
    .miso(miso)
);

Endmodule
```

TEST BENCH CODE:

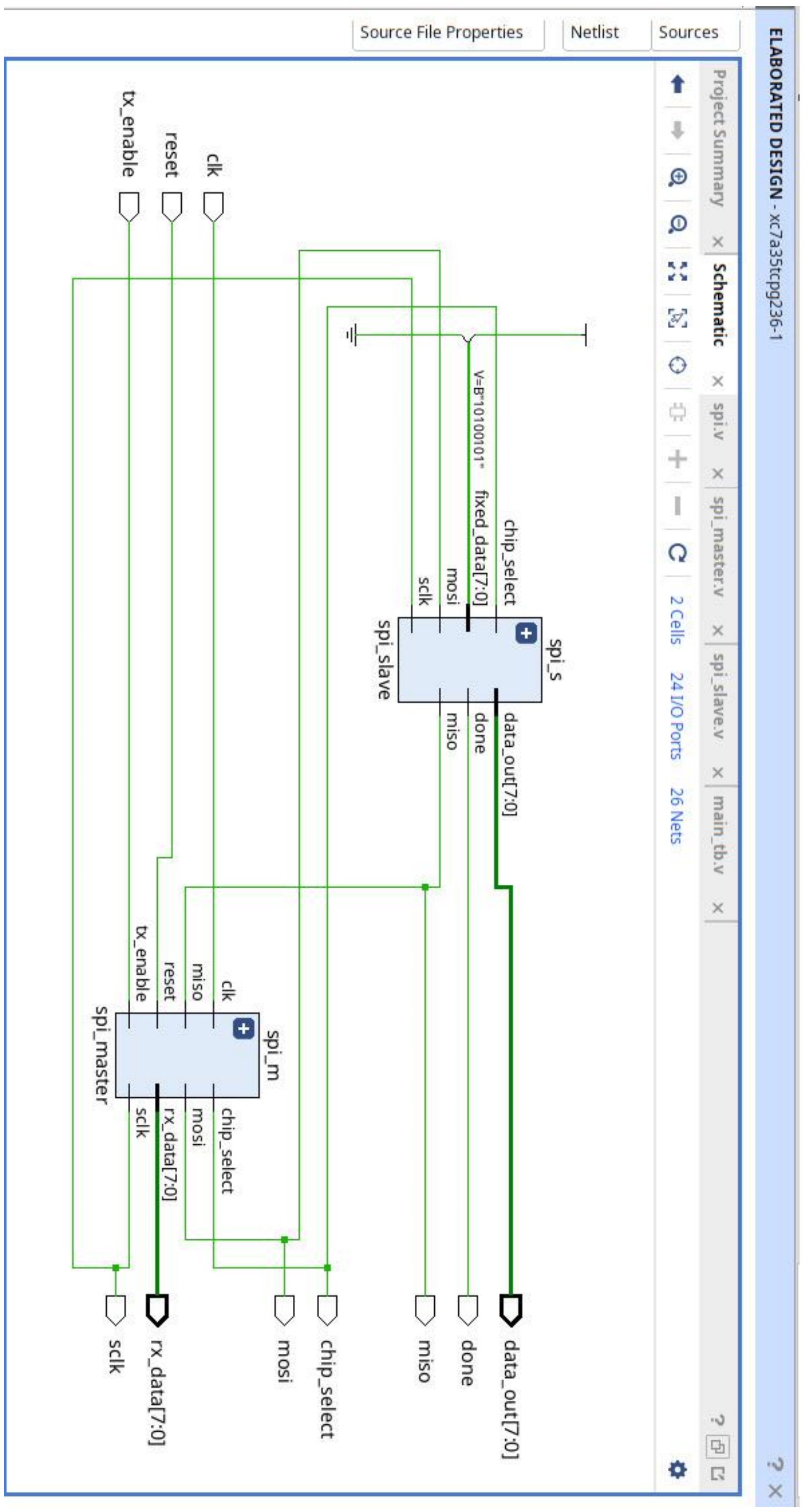
```
module main_tb;

   reg clk = 0;
   reg rst = 0;
   reg tx_enable = 0;
   wire mosi, chip_select, sclk, done, miso;
   wire [7:0] data_out;
   wire [7:0] rx_data;

   always #5 clk = ~clk;

   initial begin
      rst = 1;
      repeat(5) @(posedge clk);
      rst = 0;
   end

   initial begin
      tx_enable = 0;
      repeat(5) @(posedge clk);
      tx_enable = 1;
   end

    spi dut (
      .clk(clk), .reset(rst), .tx_enable(tx_enable),
      .mosi(mosi), .chip_select(chip_select),
      .sclk(sclk), .done(done), .data_out(data_out),
      .rx_data(rx_data), .miso(miso)
   );

Endmodule
```

WAVEFORMS :



TIMING DIAGRAM

RTL DESIGN :


ELABORATED DESIGN - xc7a35tcpg236-1
Sources
Netlist
Source File Properties
Project Summary
Schematic
spi.v
spi_master.v
spi_slave.v
main_tb.v
2 Cells
24 I/O Ports
26 Nets
clk
reset
tx_enable
V=B"10100101"
fixed_data[7:0]
chip_select
mosi
sclk
spi_s
spi_slave
data_out[7:0]
done
miso
spi_m
spi_master
clk
miso
reset
tx_enable
chip_select
mosi
rx_data[7:0]
sclk
data_out[7:0]
done
miso
chip_select
mosi
rx_data[7:0]
sclk

CHAPTER 5

CONCLUSION AND REFERENCES

CONCLUSION:

The **SPI (Serial Peripheral Interface) protocol**, when implemented in **Verilog**, provides a robust and efficient way to enable **synchronous, full-duplex communication** between digital systems, especially between microcontrollers (masters) and peripheral devices (slaves). It is widely used due to its **simplicity, speed, and deterministic timing**.

In Verilog, implementing SPI involves:

- Designing **shift registers** for serial data transmission and reception.
- Managing **synchronization** of data with the **SPI clock (SCLK)**.
- Handling **MOSI** (Master Out Slave In) and **MISO** (Master In Slave Out) data lines with accurate timing.
- Implementing **control logic** for **chip select (CS)** and **SPI mode** handling (CPOL/CPHA).
- Ensuring **tri-state logic** on the MISO line in multi-slave systems to prevent bus contention.

A well-designed SPI module in Verilog must respect protocol timing requirements, handle data shifting precisely, and be flexible enough to support different modes and configurations. This not only ensures reliable communication but also enhances the reusability of the SPI module across various digital systems and applications.

In conclusion, **Verilog provides a powerful platform for implementing SPI**, enabling designers to create **scalable, reusable, and high-performance communication interfaces** suitable for modern embedded and FPGA-based systems.

REFERNECES:

Here are some **reliable references and resources** you can cite or use to further study the **SPI protocol implementation in Verilog**:

Technical References and Documentation:

1. **Motorola (Freescale/NXP) SPI Protocol Specification:**
   - The original SPI specification was developed by Motorola. While not an official IEEE standard, it is widely accepted.
   - ✅ Reference: *Motorola SPI Block Guide (SPIGuide.pdf)*
   - NXP SPI Overview

2. **Xilinx and Intel (Altera) FPGA Documentation:**
   - FPGA manufacturers provide application notes and IP core documentation for SPI implementations in Verilog.
   - ✅ Reference: *Xilinx Application Note XAPP353: "Implementing SPI in Spartan-3 FPGAs"*
   - Xilinx App Note XAPP353
   - ✅ Reference: *Intel FPGA IP User Guide – Serial Peripheral Interface (SPI)*
   - Intel SPI IP Guide

3. **OpenCores.org – Open Source Verilog Implementations:**
   - A large repository of open-source hardware designs including SPI masters and slaves in Verilog.
   - ✅ Reference: *OpenCores SPI Master Core*
   - OpenCores SPI Master

4. **IEEE and ACM Digital Library Papers:**
   - Search for "SPI Verilog implementation" or "Serial Peripheral Interface HDL design" for peer-reviewed papers on optimized SPI implementations.
   - IEEE Xplore
   - ACM Digital Library

Textbooks and Learning Resources:

5. **Digital Design and Computer Architecture** by David Harris & Sarah Harris
   - A solid foundation for understanding digital communication interfaces like SPI in the context of Verilog design.
6. **FPGA Prototyping by Verilog Examples** by Pong P. Chu
   - This book includes several SPI communication examples implemented in Verilog, ideal for students and professionals.
7. **The Designer's Guide to Verilog** by Douglas Smith
   - Offers a practical overview of Verilog syntax and real-world use cases like serial communication interfaces.

Online Tutorials and Learning Platforms:

8. **ASIC World – SPI Protocol and Verilog Example**
   - ✅ SPI Protocol Tutorial
9. **GitHub – Community Verilog Projects**
   - Explore GitHub repositories for open-source Verilog SPI implementations.
10. **All About Circuits – SPI Interface Overview**
    - ✅ All About Circuits SPI