SPI Controller Design Documentation

Chris Zambrana

CPP Digital, Cal Poly Pomona

August 11, 2025

Contents

1 Project Overview

1.1 Purpose and Scope

This project delivers a robust, configurable, and synthesizable **Serial Peripheral Interface (SPI) controller**, implemented in Verilog for FPGA targets and verified via simulation in Vivado. The SPI controller adheres to standard SPI protocol behavior with support for **all four SPI modes (Mode 0–3)** and **both MSB-first and LSB-first bit ordering**.

The purpose of this module is to provide a reliable and reusable component for **full-duplex, high-speed serial communication** between a host (acting as SPI master) and one or more slave peripherals. The controller is designed with parameterizable datapath widths and flexible mode configuration to facilitate integration into a wide range of embedded systems, SoCs, or high-level IP designs.

This documentation provides a deep dive into the RTL structure, simulation testbench, configuration capabilities, and practical integration considerations for engineering teams that need to deploy or extend this IP.

1.2 Design Objectives

As part of the SPI IP development effort, the following design goals were prioritized:

- **Multi-mode SPI Support**: Full support for SPI Modes 0 through 3 (CPOL and CPHA configurable).
- **Flexible Bit Ordering**: Parameterized logic for MSB-first and LSB-first data transfers.
- **Multi-Slave Architecture**: Support for multiple slave devices via dynamic chip-select logic.
- **Synthesis-Ready**: Synthesizable RTL conforming to Vivado timing and resource constraints.
- **Protocol Accuracy**: Timing-aligned sampling and driving logic to ensure protocol-compliant communication.
- **Modular Hierarchy**: Clean architectural separation between the controller logic, top-level integration, and testbench.
- **Verification-Centric**: Self-contained testbench that exercises all supported modes and verifies bit-accurate data transfer in loopback and multi-slave scenarios.

1.3 Target Platform and Toolchain

The SPI controller is designed and verified using the following toolchain and environment:

- **HDL Language**: Verilog-2001
- **Development Tools**: Xilinx Vivado Design Suite (2023.x or later)
- **Simulation Environment**: Vivado Simulator (xsim)
- **Target Device**: Zynq-7000
- **Clock Domains**: Assumes synchronous logic driven by a system clock (`clk`) with optional support for divided SPI clock generation
- **FPGA Constraints**: Customizable `.xdc` constraints for I/O mapping to physical SPI pins

1.4 Features Summary

Feature	**Description**
SPI Modes	Mode 0 (CPOL=0, CPHA=0), Mode 1, Mode 2, Mode 3
Bit Ordering	MSB-first and LSB-first selectable
Configurable Frame Size	Parameterized number of bits per transfer (`NBITS`) and number of slaves being communicated with (`NSLAVE`)
Multi-Slave Support	Dynamic CS handling for multiple slave devices
Clock Divider	Integer divider for generating SPI clock from system clock
FSM Transfer Logic	Robust state machine to manage full-duplex transfers
Simulation Testbench	Validates protocol logic, edge timing, data correctness
Synthesis Ready	Verified for Vivado implementation and timing closure

Table 1: Summary of SPI Controller Features

2 Background on SPI Interface

2.1 Overview of the SPI Protocol

The **Serial Peripheral Interface (SPI)** is a high-speed, full-duplex, synchronous serial communication protocol widely used in embedded systems for short-distance device-to-device communication. It enables a single **master** to communicate with one or more **slaves** over a four-wire interface. SPI is favored for its simplicity, low latency, and high throughput compared to other serial buses such as I^{2}C or UART.

2.2 Signal Lines and Roles

SPI uses the following core signal lines:

- **SCLK (Serial Clock)**: Clock signal generated by the master to synchronize data transfer.
- **MOSI (Master Out, Slave In)**: Data line for master-to-slave transmission.
- **MISO (Master In, Slave Out)**: Data line for slave-to-master transmission. *In multi-slave systems, each slave typically has its own dedicated MISO line connected to the master. Only the selected slave's MISO is read by the master to avoid contention.*
- **CS / SS (Chip Select / Slave Select)**: Active-low signal driven by the master to select a particular slave device. One CS line is usually dedicated per slave.

2.3 Data Transmission Model

SPI operates in **full-duplex** mode, where data is simultaneously transmitted and received on each clock cycle. Both master and slave use shift registers to move data in and out bit-by-bit. The transfer process follows this sequence:

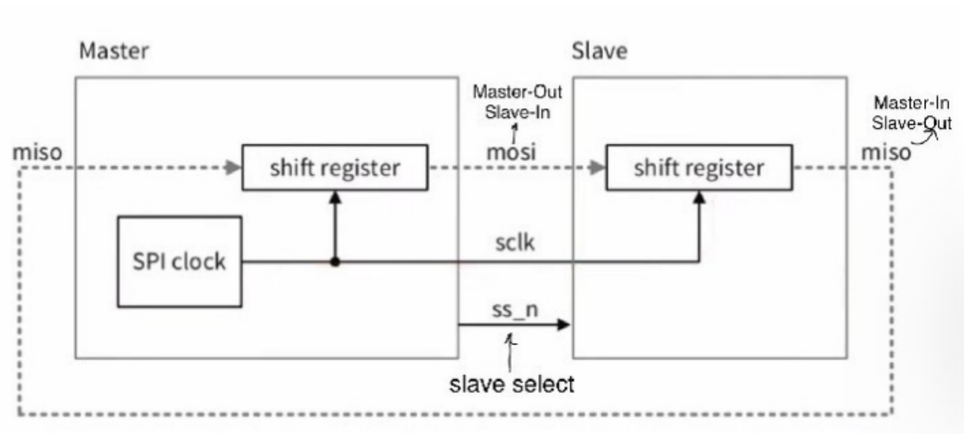

Figure 1: Basic SPI signal lines and single-master/single-slave connection

1. Master asserts the CS line for the target slave (active-low).
2. Master generates clock pulses on the SCLK line.
3. On each clock edge, both master and slave:
   - Shift out the next data bit onto their respective output lines.
   - Sample the data on the input line.

The process continues for a predefined number of bits (often 8, 16, or 32), depending on the application.

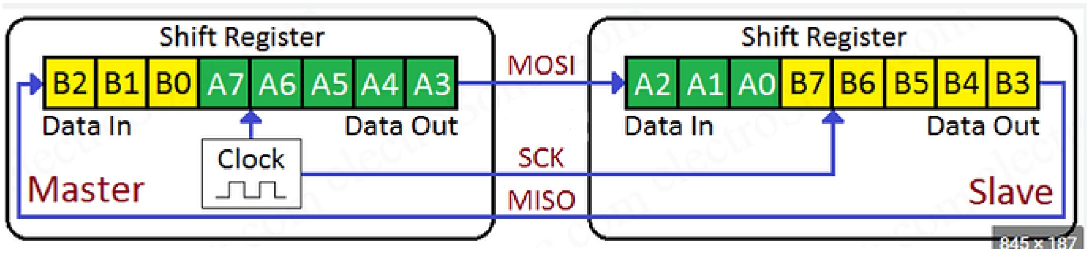

Figure 2: Simultaneous data shifting between master and slave. ”A” = master data and ”B” = slave data, and data is transferred starting from LSB. The third sampling has occurred already at the time of this image

2.4 SPI Modes: CPOL and CPHA

SPI supports four operating modes defined by two parameters:

- **CPOL (Clock Polarity)**: Sets the idle level of the clock signal.
  - CPOL = 0: Clock idles low
  - CPOL = 1: Clock idles high
- **CPHA (Clock Phase)**: Determines which edge the data is sampled on.
  - CPHA = 0: Data sampled on the first edge (after CS is asserted)
  - CPHA = 1: Data sampled on the second edge

Mode	CPOL	CPHA	Clock Idle	Sample Edge	Shift Edge
0	0	0	Low	Rising	Falling
1	0	1	Low	Falling	Rising
2	1	0	High	Falling	Rising
3	1	1	High	Rising	Falling

Table 2: SPI Modes Defined by CPOL and CPHA

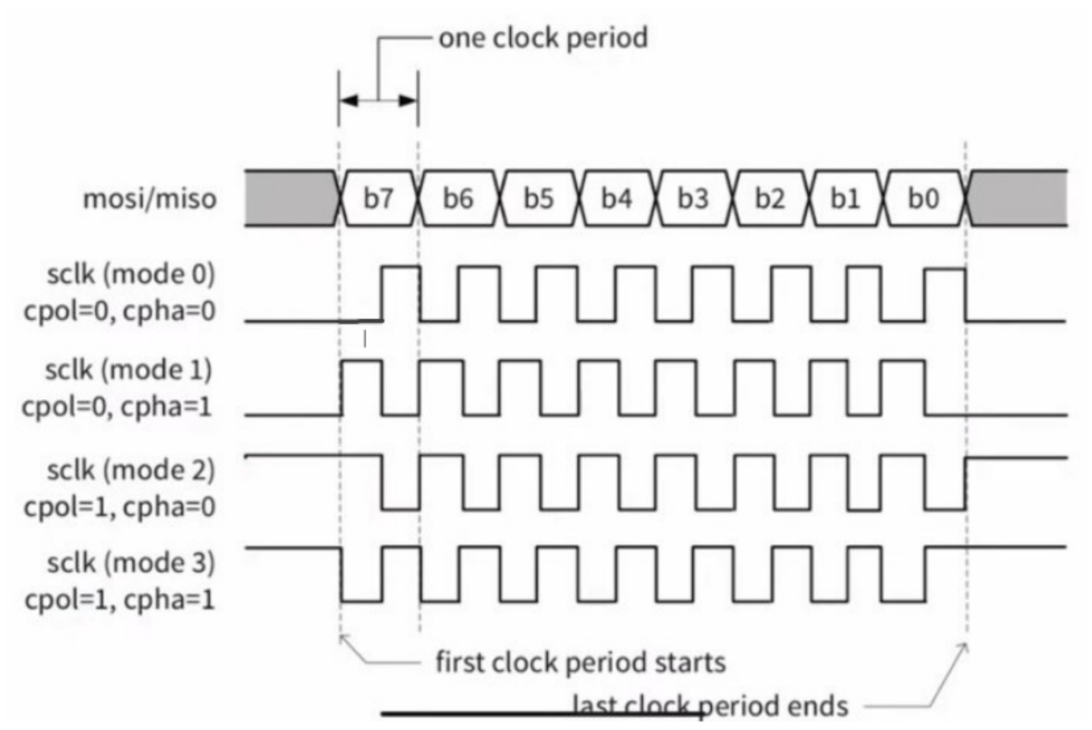

Figure 3: Timing diagrams for SPI Modes 0–3

2.5 Bit Order: MSB-first vs. LSB-first

SPI typically transmits the **most significant bit (MSB)** first, but some devices require **least significant bit (LSB)** first. The SPI controller must support both configurations.

- **MSB-first**: Bits are transmitted left to right (most significant bit first).
- **LSB-first**: Bits are transmitted right to left (least significant bit first).

This is managed by the controller's shift register, which shifts data in the appropriate direction based on configuration.

2.6 Multi-Slave Configuration

In multi-slave systems, the master:

- Drives only one CS line low at a time.
- Shares MOSI and SCLK lines among all slaves.
- Connects to **a unique MISO line for each slave**.

Only the selected slave should actively drive its MISO line. Others must disable or tri-state their output drivers to prevent contention. Internally, the master typically uses multiplexing logic to select and read the correct MISO line based on which CS is active.

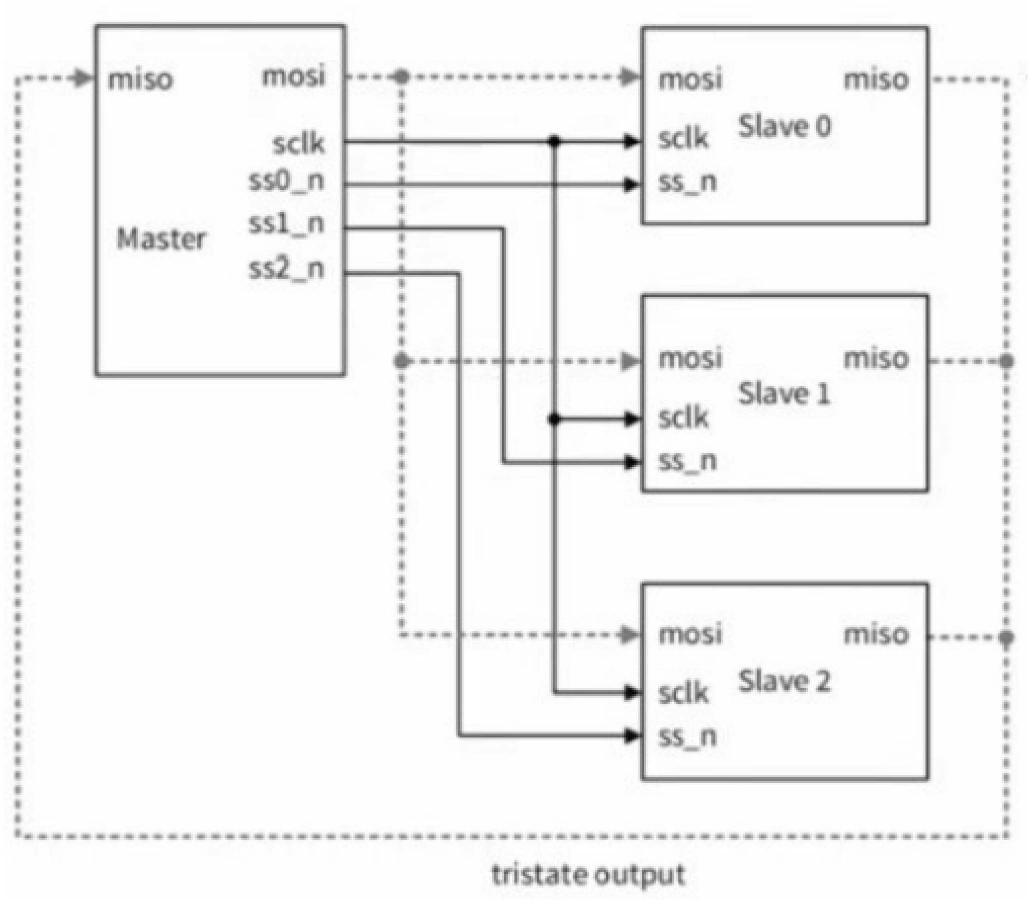

Figure 4: Multi-slave configuration with dedicated CS and MISO lines

2.7 Clocking and Throughput Considerations

SPI is often faster than other serial protocols due to its synchronous nature. Throughput depends on:

- **SCLK Frequency**: Determined by a divider from the system clock. Must be balanced for timing margin and peripheral compatibility.
- **Frame Size**: Larger frames reduce control overhead.
- **Slave Setup Time**: Some slaves require delay between CS assertion and valid data output.

Careful alignment of sampling and shifting edges is essential for reliable communication, particularly when implementing all four SPI modes.

3 Module Hierarchy and System Architecture

3.1 Overview

This section describes the structural architecture of the SPI controller design, as implemented in RTL and represented in the Vivado block diagram. The design consists of a modular and reusable SPI master controller that communicates with multiple independently configurable SPI slave modules. The system is fully parameterized to support different SPI modes (CPOL, CPHA), bit ordering, and frame size.

The top-level module (`top_spi_master`) integrates the SPI controller core and slave selection logic, acting as the interface layer between user inputs and the underlying SPI bus protocol. The architecture emphasizes clean separation of concerns, allowing each component to focus on a single responsibility (e.g., data shifting, chip select decoding, or signal tri-state control).

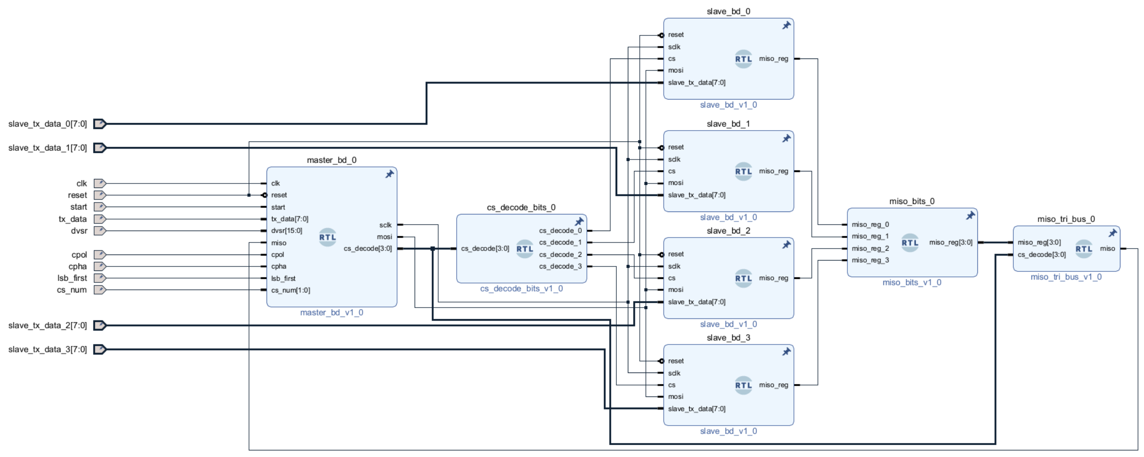

Figure 5: Vivado block diagram showing the high-level composition of `top_spi_master`, SPI controller, and simulated slave interface

3.2 Module Hierarchy Tree

- `top_spi_master`
  - `spi_controller` : Core FSM and shift-register logic.
  - `miso_tri_bus` : *Primarily utilized in simulation with slaves, NOT USED WITHIN CORE CONTROLLER.* Tri-state multiplexer for active MISO selection.
  - `master_bd` : *Not used in RTL logic; used for block diagram visualization only.* Represents the top-level wrapper in the Vivado BD environment. Some parameters are discarded in this variant.
  - `cs_decode_bits` : *Not used in RTL logic; used for block diagram visualization only.* Converts binary slave select to one-hot CS line activation.
  - `slave_bd_i` : *Not used in RTL logic; used for block diagram visualization only.* Simulated SPI slaves with shift register behavior.
  - `miso_bits` : *Not used in RTL logic; used for block diagram visualization only.* Collects MISO register outputs from all slave instances.

3.3 Top-Level Design Functionality

The `top_spi_master` module acts as the integration wrapper. It receives control and data inputs (such as start, tx_data, clock divisor, CPOL/CPHA/bit order), and drives the SPI bus signals (SCLK, MOSI, CS). Internally, it selects the correct slave device, forwards transmit data, and receives MISO input through a multiplexing logic.

Key top-level responsibilities include:

- Forwarding input parameters to the controller core.
- Converting binary slave select to one-hot signals for chip select lines.
- Routing SPI signals (CS, SCLK, MOSI) to the selected slave device.
- Accepting incoming MISO data from the active slave.
- Emitting status signals (`ready`, `spi_done_tick`) for handshake and simulation observability.
- Encapsulating the core controller with external I/O interface abstraction.

Refer to Tables 3 and 4 for a breakdown of all interface signals and their module scope.

3.4 SPI Controller Core (`spi_controller`)

This module implements the core logic responsible for serial communication. It features:

- A finite state machine for handling all SPI transaction stages.
- Configurable CPOL and CPHA logic to support all four SPI modes.
- Bit-ordering support for MSB-first or LSB-first transfer formats.
- SCLK signal generation via a programmable divider driven by the system clock.
- Shift registers for full-duplex TX and RX data exchange.
- Active-low chip select output logic based on FSM state.
- busy/done tracking for transaction status.

Refer to Tables 3 and 4 for a breakdown of all interface signals and their module scope.

3.5 Parameterization Summary

The architecture is controlled via key parameters, all propagated through the top module:

- `NBITS` – Width of each SPI frame
- `NSLAVE` – Number of slaves to communicate with

These parameters enable flexible adaptation of the design to different SPI protocols, frame sizes, or peripheral configurations.

Unified Interface Signal Reference Table

Symbol Legend:

⋆ Present only in `top_spi_master`

◦ Present only in `spi_controller`

Present in both modules

† Used primarily for simulation/debug purposes

Input Signal (Symbol)	Description
`clk`	System clock input
`reset`	Synchronous system reset
`start` ⋆	Start signal to initiate SPI transaction
`tx_data[NBITS-1:0]`	Parallel data input to be transmitted over SPI
`dvsr[15:0]`	Clock divider to generate SCLK from system clock
`cpol`	Clock polarity configuration bit
`cpha`	Clock phase configuration bit
`lsb_first`	Bit-ordering configuration (1 = LSB-first)
`miso`	Serial input from slave
`cs_num[1:0]` ⋆	Binary slave address selection

Table 3: Unified Input Signals for `top_spi_master` and `spi_controller`

Output Signal (Symbol)	Description
`rx_data[NBITS-1:0]`	Parallel data received via MISO
`mosi`	Serial output to slave
`sclk`	SPI clock output to slave
`cs_decode[NSLAVE-1:0]` ⋆	One-hot chip select output for slaves
`cs` ∘	Active-low chip select from SPI controller FSM
`ready` †	Controller ready for new transaction
`spi_done_tick` †	1-cycle done pulse when SPI transaction finishes

Table 4: Unified Output Signals for `top_spi_master` and `spi_controller`

4 SPI Protocol Support

This section details how the SPI controller design conforms to the SPI protocol standards and supports the full range of configurable features found in real-world SPI communication. The design enables protocol-level flexibility through runtime and compile-time parameters, making it suitable for integration with a wide variety of SPI-compatible devices.

4.1 Supported SPI Modes (Mode 0–3)

The controller fully supports all four standard SPI modes:

- Mode 0: CPOL = 0, CPHA = 0
- Mode 1: CPOL = 0, CPHA = 1
- Mode 2: CPOL = 1, CPHA = 0
- Mode 3: CPOL = 1, CPHA = 1

The internal state machine accounts for CPHA when determining when to sample MISO or shift out MOSI. The output clock (sclk) waveform is dynamically generated based on the CPOL input to ensure alignment with protocol expectations.

4.2 Bit Order Configurability

The SPI controller supports both Most Significant Bit first (MSB-first) and Least Significant Bit first (LSB-first) transmission.

This flexibility allows compatibility with peripherals that require non-standard bit ordering. Internally, the controller adjusts the bit index and shift direction during both transmission and reception to ensure correctness.

Figure 6: Comparison of MSB-first and LSB-first transmission

4.3 Frame Size (Bit-Width) Support

The controller is parameterized by `NBITS`, which defines the number of bits per SPI transfer frame.

- Default value in the modules is 8 bits, but can be adjusted to what is necessary for operation

This value determines the depth of the internal shift registers and the transfer loop count. The FSM is dynamically responsive to the frame size, making the controller adaptable to devices with custom word lengths (16, 24, or 32-bit transfers).

4.4 Multi-Slave Support

The controller supports communication with multiple SPI slave devices through:

- `cs_num[1:0]` input – selects the target slave via binary encoding.
- `cs_decode[NSLAVE-1:0]` output – one-hot decoded chip select vector.

This allows the master to address any of the `NSLAVE` connected devices individually. Only the selected slave's chip-select line is asserted during the transaction.

Note: The actual logic to decode `cs_num` into `cs_decode` is implemented within the top level module after instantiations.

4.5 Clock Divider and Timing Control

The SPI clock (`sclk`) is derived from the system clock (`clk`) using a programmable clock divider input:

- `dvsr[15:0]` sets the division ratio.
- Internally implemented using a counter to produce a toggling signal.

The divider allows fine-grained control over the SPI frequency to meet the timing requirements of slower slave devices. Additionally, the timing logic ensures correct alignment of sampling and shifting edges, taking into account CPHA.

5 Testbench Design

This section describes the simulation environment and validation strategy used to verify the SPI controller functionality. The testbench is fully self-contained, supporting functional correctness across all supported SPI modes, bit orderings, and slave selection logic.

5.1 Testbench Overview

The testbench instantiates the top-level SPI master (`top_spi_master`) and connects it to four simulated slave devices. Each slave is modeled using register-based loopback logic. The environment tests the controller continuously across all SPI protocol modes and bit orders.

Key features of the testbench include:

- Parameterized support for variable frame width (`NBITS`) and number of slaves (`NSLAVE`)
- Simulation of all four SPI modes (Mode 0–3)
- Both MSB-first and LSB-first transmission formats
- Rotating slave selection for each iteration
- Internal loopback to validate full-duplex data exchange

5.2 Clock and Reset Generation

The testbench generates a periodic system clock using a parameterized timing interval defined by `CLK_PER`. The reset is asserted at the beginning of the simulation and deasserted after a fixed delay.

```
always #(CLK_PER/2) clk = ~clk;

initial begin
  reset = 0;
  ...
  repeat(2) @(posedge clk);
  reset = 1;
end
```

Listing 1: Clock and Reset Logic in Testbench

This logic ensures that:

- `clk` toggles every half period, simulating a 100 MHz or user-defined clock
- `reset` stays asserted for 2 clock cycles, ensuring proper module initialization

5.3 MISO Bus Simulation

In a real SPI system, only the selected slave device is allowed to drive the MISO line, while all others must remain electrically disconnected (tri-stated) to avoid bus contention. The testbench models this behavior using a simulation-only module called `miso_tri_bus`.

This module receives:

- A vector of slave outputs: `miso_reg[NSLAVE-1:0]`
- A one-hot active-low chip select vector: `cs_decode[NSLAVE-1:0]`

Only one chip-select line is active-low at a time (e.g., `cs_decode = 4'b1110` for Slave 0). Based on the active chip-select, the module forwards the corresponding slave's MISO value to the shared output `miso`. If no slave is selected, the output is set to high-impedance (`1'bz`), mimicking open-drain behavior.

```
module miso_tri_bus
  #(parameter NSLAVE = 4)
(
  input   [NSLAVE-1:0] miso_reg,      // MISO outputs from all slaves
  input   [NSLAVE-1:0] cs_decode,     // Active-low CS decode signal
  output reg miso                     // Shared MISO line to master
);

  always @(*) begin
    casez (cs_decode)
      4'b1110: miso = miso_reg[0];    // Slave 0 selected
      4'b1101: miso = miso_reg[1];    // Slave 1 selected
      4'b1011: miso = miso_reg[2];    // Slave 2 selected
      4'b0111: miso = miso_reg[3];    // Slave 3 selected
      default: miso = 1'bz;           // No slave selected ? tri-state
    endcase
  end

endmodule
```

Listing 2: Tri-State MISO Bus Multiplexer

This logic allows simulation of realistic shared-bus behavior without contention. It assumes exactly one slave is selected at any time and works effectively in multi-slave testing scenarios.

Note: This module is intended for simulation only and is not used in synthesis.

5.4 Shift Register Behavior and Loopback Modeling

The testbench simulates each slave using a shift register that responds to SPI clock and data signals. These slaves are instantiated using a `generate-for` loop, ensuring scalable and independent behavior per slave.

Each simulated slave has:

- An internal shift register (`slave_shift_reg[i]`) to output bits via MISO.
- A control condition that activates logic only if the slave's chip-select (`cs_decode[i]`) is asserted (active-low).

The simulation runs in parallel across all slave instances. Only the currently selected slave (the one with an active `cs_decode[i] = 0`) is allowed to shift data and influence the MISO bus.

```
genvar i;
generate
  for (i = 0; i < NSLAVE; i = i + 1) begin
    always @(posedge sclk or negedge reset) begin
      if (!reset) begin
        slave_shift_reg[i] <= 8'h00;
        miso_reg[i] <= 1'b0;
      end else if (cs_decode[i] == 1'b0) begin
        // Only selected slave is active
        if ((!cpha & !cpol) | (cpha & cpol))
          slave_shift_reg[i] <= {slave_shift_reg[i][NBITS-2:0], mosi};
        else
        // Shift MISO output for next bit
        miso_reg[i] <= lsb_first
                    ? slave_shift_reg[i][1]
                    : slave_shift_reg[i][NBITS-2];
      end
    end
  end
endgenerate
```

Listing 3: Slave behavior on the posedge using generate-for loop.

Note: This code represents the behavior of the **positive edge** of the SPI clock. An identical block exists in the testbench that uses `negedge sclk`, and is conditionally enabled for SPI Modes 1 and 2. The only difference is the sensitivity list of the `always` block.

Explanation of key behaviors:

- **Edge Selection Based on CPOL/CPHA:** The condition `(!cpha & !cpol) | (cpha & cpol)` evaluates to true for SPI Modes 0 and 3, where sampling occurs on the rising edge. Conversely, the same condition is used on the `negedge sclk` block to simulate Modes 1 and 2, which shift on the falling edge.
- **CS-Based Slave Activation:** Each slave block checks `cs_decode[i] == 1'b0` before acting. This ensures only the selected slave updates its shift register and MISO output.
- **Bit Alignment for MISO:** If the conditional statement proves to be false, the next bit to be sent is assigned to `miso_reg[i]`. The bit index is selected dynamically:
  - If `lsb_first == 1`, output the next least significant bit.
  - If `lsb_first == 0`, output the next most significant bit.
- **Simulation Parallelism:** All slave shift registers are instantiated using the `generate` loop, and operate in parallel. However, only the active slave modifies state during any transaction cycle.

The MISO line itself is tri-stated through the `miso_tri_bus` module, which ensures only the selected slave's output is visible to the master during transmission (see Section 5.3).

5.5 Simulation Configuration and Protocol Execution

The task `run_tx` encapsulates the initialization and execution of a single SPI transaction. This task is called repeatedly during simulation to test various protocol configurations and slave indices.

It performs the following responsibilities:

- Waits until the controller is ready
- Applies mode parameters (`cpol`, `cpha`, `lsb_first`)
- Selects the target slave via `cs_num`
- Loads transmit data to `tx_data`
- Initializes the slave shift register with expected response
- Sets up the initial MISO value based on bit order
- Pulses the `start` signal to initiate transfer
- Waits for `spi_done_tick` indicating completion

```
task run_tx(
    input [NBITS-1:0] master_tx_data,
    input [NBITS-1:0] slave_tx_data,
    input mode_cpol,
    input mode_cpha,
    input mode_lsb,
    input [$clog2(NSLAVE)-1:0] slave_num
  );
  begin
    wait(ready);
    cpol        = mode_cpol;
    cpha        = mode_cpha;
    lsb_first   = mode_lsb;
    cs_num      = slave_num;
    dvsr        = DVSR;
    tx_data     = master_tx_data;

    // Preload slave register and initial MISO bit
    slave_shift_reg[slave_num] = slave_tx_data;
    miso_reg[slave_num] = mode_lsb
                        ? slave_tx_data[0]
                        : slave_tx_data[NBITS-1];

    @(posedge clk);
    start <= 1'b1;
    @(posedge clk);
    start <= 1'b0;

    wait(spi_done_tick);
  end
endtask
```

Listing 4: SPI Transaction Execution Task

By abstracting this sequence into a task, the testbench maintains clean and readable structure while ensuring consistent protocol behavior across hundreds of test iterations.

5.6 Waveform Observation and Debugging

All key SPI signals and control flags are observable via simulation. These waveforms serve to visually validate protocol behavior and confirm bit alignment across different SPI modes.

SPI Bus Logic	`sclk`, `mosi`, `miso`, `cs_decode`, `cpol`, `cpha`,
Control Logic	`start`, `ready`, `spi_done_tick`, `tx_data`, `rx_data`, `cs_num`, `lsb_first`, `slave_shift_reg`, `reset`,
FSM Behavior	`cs_n`, `state_reg`, `toggle`, `clk`,

Waveform analysis can be performed using Vivado's built-in simulator (XSIM).