EE496
Undergraduate Project
Lokesh Mehra
220591

1. Introduction

1.1 Background

In modern digital system design, efficient inter-module communication is critical for the performance of System-on-Chip (SoC) architectures and embedded systems. The Serial Peripheral Interface (SPI) remains one of the most ubiquitous synchronous serial communication protocols due to its simplicity, full-duplex capability, and high-speed data transfer rates. While standard SPI implementations are common, complex applications often require customized addressing schemes and scalable multi-slave architectures to manage distributed resources effectively.

This project focuses on the design, simulation, and hardware implementation of a robust, addressable SPI communication system. The primary motivation is to develop a scalable register control interface capable of managing multiple distinct slave devices through a unified master controller, a requirement frequently encountered in sensor networks, industrial automation, and complex ASIC control paths.

1.2 Project Objectives

The primary goal of this project was to design a custom SPI protocol wrapper to control a 16-bit register bank system. The specific technical objectives were as follows:

- **Register Bank Implementation: To design a 16-bit wide register bank with an address space ranging from `0x00` to `0xFF`, allowing for extensive data storage and retrieval.**
- **Custom Instruction Protocol: To implement an SPI frame structure that embeds the target register address within the read/write instruction, ensuring precise data manipulation.**
- **Scalable Multi-Slave Architecture: To incorporate a 3-bit Chip ID mechanism, enabling a single SPI Master to uniquely address and control up to 16 individual SPI Slaves sharing the same bus.**

- **Global Write Capability: To implement a broadcasting feature (Global Write) that allows the master to simultaneously write the same data to a specific register address across all connected slaves, optimizing configuration time.**

1.3 Scope of Implementation

This report documents the complete design flow, starting from Register Transfer Level (RTL) coding to hardware validation and physical synthesis. The scope includes:

1. **Functional Verification: Simulation of the SPI Master and Slave logic to verify adherence to the custom protocol and read/write timing.**
2. **FPGA Prototyping: Hardware implementation using two separate FPGA boards. One board functions as the Master controller, and the second board acts as a single representative Slave (mimicking one of the 16 potential slaves) to validate physical signal integrity and communication logic.**
3. **ASIC Synthesis: Synthesis of the verified Master and Slave RTL code targeting the TSMC 65nm technology node to analyze area, power, and timing characteristics for potential ASIC integration.**

2. System Architecture

The architecture is divided into two main subsystems: the Master Subsystem and the Slave Subsystem.

A. Master Subsystem (FPGA 1)

The Master subsystem is responsible for initiating transactions, generating the clock, and managing the control flow for single and global operations.

- **Top Level (`master_top.v`): This module integrates the control logic. It drives the physical SPI interface (`spi_sclk_out`, `spi_mosi_out`, `spi_cs_n_out`) and receives `spi_miso_in`.**
- **System Control FSM (`system_control_fsm.v`): This is the "brain" of the master. It handles:**
    - **User Inputs: Accepts commands via physical buttons or VIO to latch addresses, data, and trigger Read/Write events.**
    - **Packet Construction: It assembles the 16-bit command packet containing the Chip ID, Address, and Read/Write flags.**

    - **Global Write Logic: It implements the "Global Write" feature by iterating through Chip IDs (0 to 7) sequentially, sending the same write command to every slave when the global flag is set.**
- **SPI Master FSM (`spi_master_fsm.v`): This acts as the Physical Layer (PHY). It takes the high-level 16-bit command and data from the control FSM and serializes them onto the MOSI line. It generates the serial clock (`sclk_out`) derived from the system clock using a configurable divider (`SCLK_DIV`). It also handles Chip Select decoding, activating the specific `cs_n_out` line based on the Chip ID.**

B. Slave Subsystem (FPGA 2)

The Slave subsystem responds to the Master, executes register reads/writes, and drives local peripherals (LEDs).

- **Top Level (`slave_top.v`): Connects the physical SPI lines to the internal slave logic. It includes a latch mechanism to display written data on LEDs to visually verify successful transactions.**
- **SPI Slave FSM (`spi_slave_fsm.v`): This module synchronizes the asynchronous SPI signals (`sclk`, `cs_n`) into the slave FPGA's clock domain. It deserializes the incoming command and data. It parses the instruction to determine if a Read or Write is requested.**
- **Register Bank (`slave_ram.v`): A 256-depth x 16-bit wide RAM that serves as the register bank. It supports single-cycle read and write operations initiated by the Slave FSM.**

3. Working of the SPI Protocol

The system utilizes a standard 4-wire Serial Peripheral Interface (SPI) protocol, enhanced with a custom packet structure to handle addressing and read/write commands over a single data line.

A. User Input & Command Construction

The Master FPGA abstracts the low-level protocol generation from the user through a "System Control FSM." The command generation process occurs in three stages:

1. **Parameter Selection (Switches): The user defines the target implementation using physical switches on the Master board.**

- **Target ID: Switches `[10:8]` select the 3-bit Chip ID (0-7).**
- **Target Address: Switches `[7:0]` select the 8-bit Register Address (0x00 - 0xFF).**
- ***Note:* Upon pressing the "Latch" button, these values are stored in the internal registers `latched_id` and `latched_addr`.**

2. **Operation Selection (Buttons): The specific operation is triggered by pushbuttons, which determine the Read/Write bit logic:**
   - **Write Button: Sets the internal R/W flag to `0`.**
   - **Read Button: Sets the internal R/W flag to `1`.**
3. **Packet Assembly: The Control FSM constructs a 16-bit command packet by shifting the user inputs to accommodate protocol flags. The 8-bit address from the switches is left-shifted by 3 positions to bits `[10:3]`, ensuring it does not overlap with the R/W flag at bit `[1]`.**
   The final 16-bit packet sent to the SPI Master PHY is structured as follows:
   - **Bits: Padding (00)**
   - **Bits: Chip ID (Derived from Switches [10:8])**
   - **Bits [10:3]: Register Address (Derived from Switches [7:0])**
   - **Bit [2]: Global Write Flag (Internal Logic)**
   - **Bit [1]: Read/Write Bit (0 = Write, 1 = Read)**
   - **Bit [0]: Padding (0)**

B. Physical Layer Transmission

The transmission adheres to SPI Mode 0 (CPOL=0, CPHA=0). The Master generates the serial clock (`sclk`) and manages the Chip Select lines (`cs_n`).

1. **CS_N Activation: Upon receiving the start signal, the Master decodes the 3-bit Chip ID and pulls the corresponding line on the 8-bit `cs_n_out` bus Low to activate the specific slave.**
2. **Command Phase (First 16 Clock Cycles):**
   - **The Master shifts out the 16-bit command packet on the MOSI line.**
   - **The Slave samples MOSI on the rising edge of `sclk`.**
   - **At the end of this phase (Bit count = 1), the Slave captures the packet and extracts the Address (`[10:3]`) and the R/W bit (`[1]`) to prepare the RAM for the next phase.**
3. **Data Phase (Next 16 Clock Cycles):**
   - **If Write (R/W = 0): The Master continues driving MOSI with the 16-bit data payload derived from the switches (during the write command trigger). The Slave shifts this data in and writes it to the RAM at the decoded address.**

    - If Read (R/W = 1): The Slave takes control of the MISO line. It retrieves data from the RAM address specified in the command phase and shifts it out on MISO. The Master samples this data on the rising edge of `sclk`.
4. Termination: After 32 total clock cycles (16 Command + 16 Data), the Master de-asserts the `cs_n` line (returns it to High), completing the transaction.

C. Global Write Operation (Sequential Broadcast)

The system implements a "Global Write" feature designed to update the register banks of all connected slaves (IDs 0 through 7) with identical data at the same address. Unlike a hardware broadcast, this is implemented as an automated sequential loop within the Master's Control FSM.

1. Initiation and Configuration The Global Write mode is armed by the user during the parameter selection phase:

- Global Flag: The user must set the Most Significant Bit (MSB) of the input switches (Switch `[15]`) to High.
- Latching: When the "Latch" button is pressed, this bit is stored in the internal `latched_global` register.
- Trigger: The process officially begins when the user presses the "Write" button. The FSM detects the `latched_global` flag and transitions to the `S_GW_BUILD` state instead of the standard single transmission state.

2. Automated Execution Loop Once triggered, the System Control FSM takes over control of the SPI Master interface, ignoring the "Chip ID" switches set by the user. It executes a loop using an internal 3-bit counter, `gw_idx`.

The loop operates through the following state sequence for `gw_idx = 0` to `7`:

- State: S_GW_BUILD (Packet Construction): The FSM constructs a new command packet for the current iteration. Crucially, it replaces the static user ID with the dynamic loop counter `gw_idx`:
    - Packet (ID): Derived dynamically from `gw_idx[2:0]` (iterating 000 to 111).
    - Packet [10:3] (Address): Derived from the static `latched_addr`.
    - Packet [15:0] (Data): The data captured when the write button was first pressed is used for every transaction.
- State: S_GW_START (Transmission): The FSM signals `master_start_tx` to the SPI PHY, initiating the physical 32-cycle SPI transaction for the current Slave ID.
- State: S_GW_WAIT (Synchronization): The Control FSM waits for the `master_tx_done` signal, ensuring the SPI bus is free before proceeding.

- **State: S_GW_NEXT (Iteration): The FSM checks the value of `gw_idx`:**
    - **If `gw_idx < 7`: The counter is incremented (`gw_idx + 1`), and the FSM returns to `S_GW_BUILD` to target the next slave.**
    - **If `gw_idx == 7`: The loop is complete, and the FSM transitions to `S_GW_DONE`.**

3. Completion After successfully completing 8 separate SPI transactions (covering IDs 0-7), the Master illuminates the "Write Done" LED and starts the 3-second timer to indicate the global operation is finished.

4. Simulation and Verification

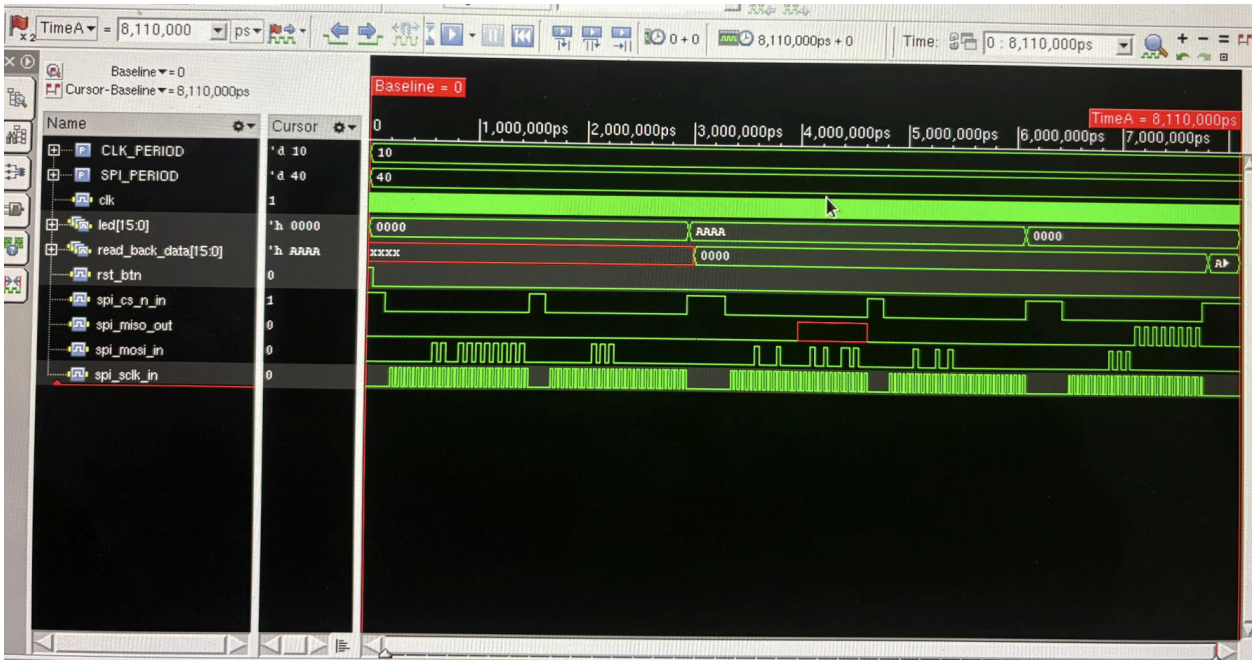
To validate the functional correctness of the RTL design before synthesis, two distinct simulation environments were created: one for the Slave Subsystem to verify protocol compliance, and one for the Master Subsystem to verify the User Interface and Global Write logic.

A. Slave Subsystem Verification

The slave testbench was designed to simulate a Master communicating with the Device Under Test (DUT). The testbench manually drives the SPI signals (`spi_sclk`, `spi_mosi`, `spi_cs_n`) to emulate the timing of SPI Mode 0.

Test Sequence: The simulation performs a "Self-Checking" sequence to verify the RAM works as expected:A

1. **Reset: The system is reset to ensure the FSM starts in the IDLE state.**
2. **Single Write: The testbench writes the value `0xAAAA` to register address `0x05`.**
3. **Single Read: The testbench initiates a read from address `0x05`. It captures the serial data on `spi_miso_out` and compares it against the expected value (`0xAAAA`).**



Analysis of Slave Simulation Waveform

1. **Successful Write Transaction (First Pulse):**
    - **Action: The Master asserts `spi_cs_n_in` (goes low) and clocks in data on `spi_mosi_in`.**
    - **Result: Look at the `led[15:0]` signal. Initially, it is `0000`. After the first CS_N pulse completes, the `led` bus updates to `AAAA`. This confirms that the write command (writing `0xAAAA` to address `0x05`) was successfully decoded, the data was stored in RAM, and the "write pulse" latched this data to the LEDs.**
2. **Successful Read Transaction (Second Pulse):**
    - **Action: The Master asserts `spi_cs_n_in` again for a read command.**
    - **Result: During this pulse, look at the `spi_miso_out` line. It becomes active (toggling high and low) while `spi_cs_n_in` is low. This indicates the Slave is actively driving data back to the Master.**
    - **Verification: The `read_back_data` signal (which samples MISO in the testbench) updates to `AAAA` at the end of the transaction. This matches the data written in the previous step, proving the read-after-write integrity.**

Simulation Results: The simulation confirms that the Slave FSM correctly samples MOSI on rising edges and drives MISO on falling edges. The self-check passed, confirming the RAM successfully stores and retrieves data.

B. Master Subsystem Verification

Testbench: `tb_master_top.v`

The Master testbench validates the `system_control_fsm` and the `spi_master_fsm`. To accurately test the Master, a Behavioral Slave Model was implemented within the testbench. This model acts as a "perfect" slave that responds to the Master's signals, allowing us to verify that the Master is sending the correct commands.

Behavioral Slave Model: The testbench includes a simplified slave logic that:

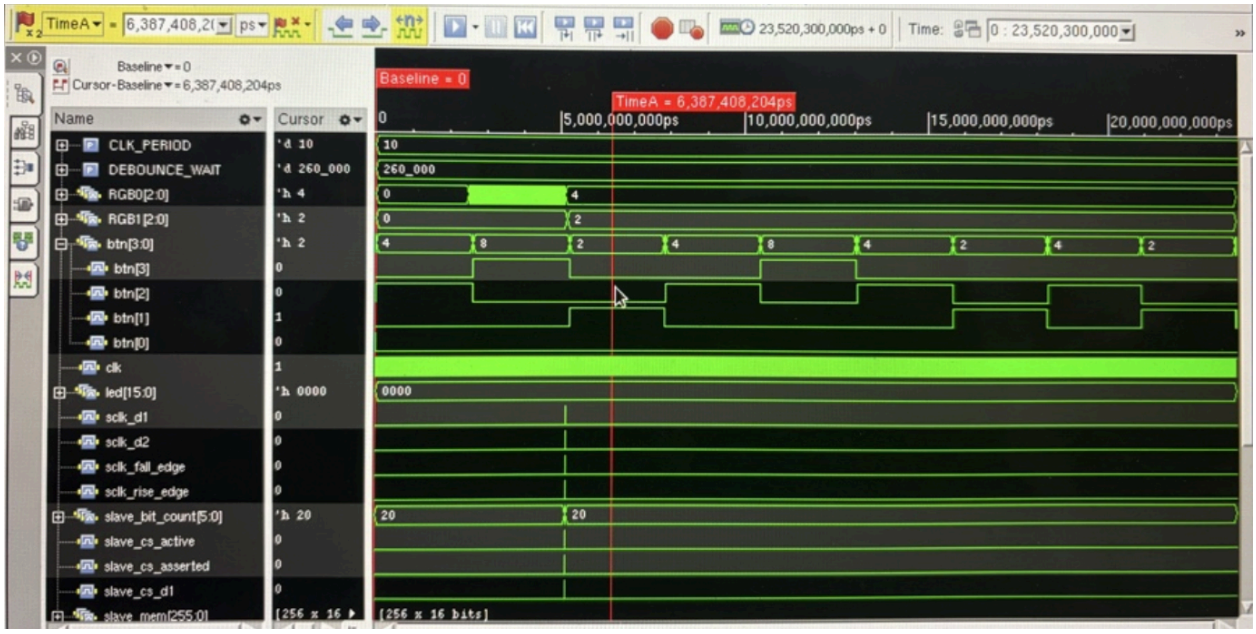
- **Detects the 8-bit Address and 1-bit R/W flag from the Master's MOSI stream.**
- **Maintains an internal simulated memory array `slave_mem [255:0]`.**
- **Dynamically updates memory on Write commands and drives MISO on Read commands.**

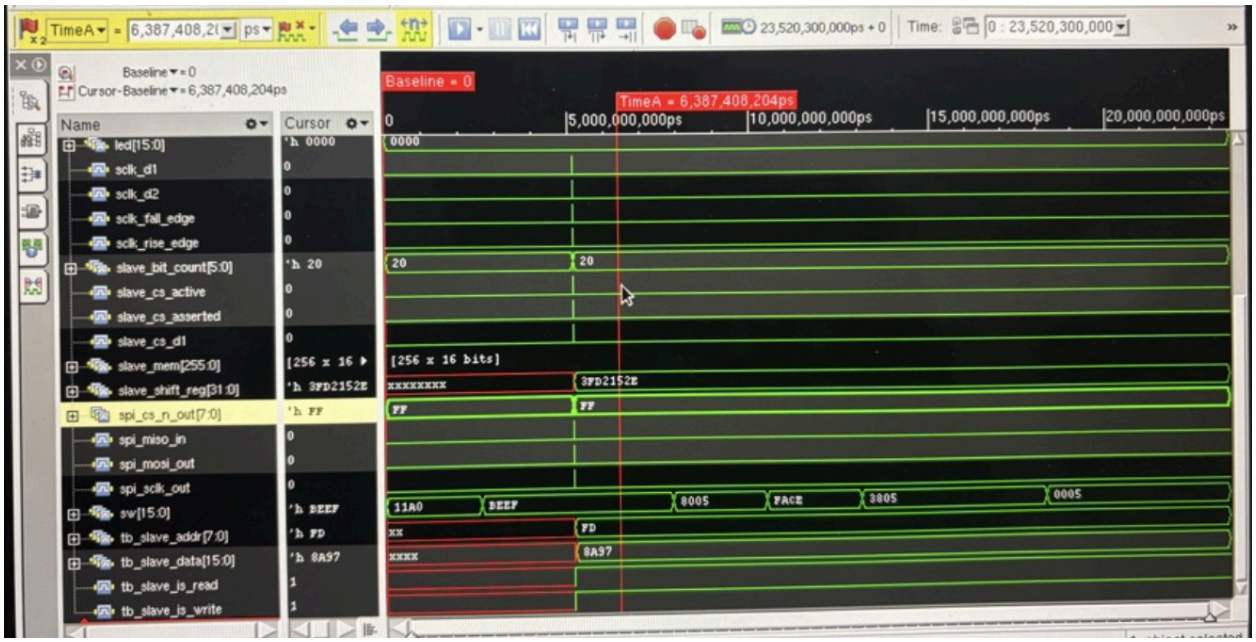
Test Sequence:

1. **Single Write (User Input): The testbench simulates button presses (`pulse_btn`) to latch Chip ID `1` and Address `0x1A`. It then inputs data `0xBEEF` and triggers a write. This verifies the input shifting logic.**
2. **Single Read: The testbench triggers a read operation. The simulation verifies that the Master waits for the transaction to complete and displays the data received from the behavioral slave on the LEDs.**
3. **Global Write Verification:**
    - **Trigger: The testbench sets the Global Flag (`sw[15]=1`) and triggers a write with data `0xFACE`.**

- **Observation: The simulation waveform shows the Master automatically generating 8 consecutive SPI transactions, iterating through Chip IDs 0 to 7.**

4. **Global Write Validation: To prove the Global Write worked, the testbench performs individual reads from separate IDs (ID 7 and ID 0). Both read operations successfully retrieve the data `0xFACE`, confirming the broadcast was successful.**





Analysis of Master Simulation

1. Test Sequence Verification (Top Waveform Image) The `btn[3:0]` signals clearly show the stimulus applied by the testbench, matching the operational procedures:

- **Time ~2.6 ms (Test 1): `btn[2]` (Latch) toggles, followed quickly by `btn[3]` (Write). This triggers the Single Write operation.**
- **Time ~6.5 ms (Test 2): `btn[1]` (Read) toggles, triggering the Single Read operation.**
- **Time ~10.5 ms (Test 3): `btn[2]` (Latch) toggles again (loading the Global settings), followed by `btn[3]` (Write).**
- **Time ~14.0 ms & 18.0 ms (Test 4): Verification reads are triggered to check if the global write persisted.**

2. SPI Bus & Global Write Validation (Bottom Waveform Image) The correspondence between the User Interface (Switches) and the SPI Bus (`spi_cs_n_out`) confirms the architectural logic:

- **Single Write (Left Side): When `sw` is set to `0xBEEF`, we see a single, narrow period of activity on `spi_cs_n_out`. This represents one discrete SPI transaction (32 clock cycles) targeted at a single ID.**
- **Global Write (Middle - The "Smoking Gun"):**
  - **Look at the timeline around 11.0 ms, where `sw` is set to `0xFACE`.**
  - **Unlike the single write, the activity on `spi_cs_n_out` is a wide, continuous block.**
  - **This extended duration confirms that the Master FSM entered the `S_GW` loop and is autonomously executing 8 consecutive SPI transactions (one for each Slave ID 0-7) without requiring 8 separate button presses.**
  - **This visually proves the "Global Write" feature successfully iterates through the slave address space.**

Simulation Results: The waveforms confirm that the Master FSM correctly constructs the packet (shifting the address bits) and that the "Global Write" loop correctly iterates through all slave IDs without user intervention.

4. FPGA Hardware Demonstration

This section details the procedure to verify the SPI protocol on the physical hardware. The setup consists of two FPGAs:

1. **Master Board: Controls transactions and inputs.**
2. **Slave Board: Acts as the SPI peripheral (Slave) and displays received data.**

Hardware User Interface Map:

- **Switch [15]: Global Write Enable (1 = Global, 0 = Single)**
- **Switch [10:8]: Chip ID Selection (3-bit)**
- **Switch [7:0]: Register Address (8-bit)**
- **Switch [15:0]: Data Input (Used only during the Write Data phase)**
- **Button [2]: LATCH (Stores ID, Address, and Global settings)**
- **Button [3]: WRITE (Triggers a Write transaction with current switch data)**
- **Button [1]: READ (Triggers a Read transaction from the latched address)**

Experiment A: Single Write Operation

Objective: Write the value `0xABCD` to Register Address `0x10` on Slave ID `2`.

1. **Parameter Setup:**
    - **Set Switch [15] to `0` (Disable Global Mode).**
    - **Set Switches [10:8] to `010` (Binary for ID 2).**
    - **Set Switches [7:0] to `00010000` (Binary for Address `0x10`).**
2. **Latch Configuration:**
    - **Press Button [2] (Latch).**
    - ***Observation:* The Master's Red LED flashes briefly, indicating the FSM has stored the ID and Address.**
3. **Data Input:**
    - **Change Switches [15:0] to `1010_1011_1100_1101` (Hex `0xABCD`).**
4. **Execute Write:**
    - **Press Button [3] (Write).**
    - ***Observation (Master):* The Blue LED lights up, indicating the SPI bus is busy transmitting.**
    - ***Observation (Slave):* The LEDs on the Slave FPGA update to show `0xABCD`. This confirms the Slave received the data, stored it in RAM, and output it to the debug LEDs.**

Experiment B: Single Read Operation

Objective: Verify the previous write by reading back from Address `0x10` on Slave ID `2`.

1. **Setup:**
    - **The ID (`2`) and Address (`0x10`) are already stored in the Master's internal registers from Experiment A. (If not, repeat the "Parameter Setup" and "Latch Configuration" steps from Exp A).**
    - ***Note:* The position of the switches currently does not matter for a Read operation, as the Master uses the internally latched values.**
2. **Execute Read:**
    - **Press Button [1] (Read).**
3. **Verification:**
    - **The Master sends the Read command (`R/W` bit = 1).**
    - **The Slave retrieves `0xABCD` from its RAM at address `0x10` and sends it back on the MISO line.**
    - ***Observation (Master):* The Master FPGA LEDs update to display `0xABCD`. The Green LED lights up, indicating a successful read of non-zero data**

Experiment C: Global Write (Broadcast)

Objective: Write the value `0xFFFF` to Register Address `0x55` on ALL Slaves (IDs 0-7).

1. **Parameter Setup:**
    - **Set Switch [15] to `1` (Enable Global Mode).**
    - **Set Switches [7:0] to `01010101` (Binary for Address `0x55`).**
    - ***Note:* The Chip ID switches [10:8] are ignored in this mode.**
2. **Latch Configuration:**
    - **Press Button [2] (Latch).**
    - ***Observation:* Internal flag `latched_global` is set to 1.**
3. **Data Input:**
    - **Set Switches [15:0] to `1111_1111_1111_1111` (Hex `0xFFFF`).**
4. **Execute Global Write:**
    - **Press Button [3] (Write).**
    - ***Internal Process:* The Master FSM enters the `S_GW_BUILD` state. It automatically iterates `gw_idx` from 0 to 7, sending eight separate SPI write commands sequentially.**
    - ***Observation:* The Blue LED on the Master stays illuminated noticeably longer than in a single write, reflecting the time taken to complete 8 consecutive transactions.**
5. **Verification:**
    - **Since the Slave FPGA is hardcoded (via parameter or DIP switch) to a specific ID (e.g., ID 0), it will accept the write when the Master loops to ID 0.**
    - ***Result:* The Slave FPGA LEDs turn on all 16 bits (`0xFFFF`), confirming it accepted the global broadcast.**

6. ASIC Synthesis (TSMC 65nm)

6.1 Synthesis Environment & Library

The synthesis was performed using the `tcbn65gpluswc_ccs.lib` standard cell library.

- **Technology: TSMC 65nm**
- **Library Variant: `wc_ccs` (Worst-Case, Composite Current Source).**
- **Operating Conditions: WCCOM (Worst Case Commercial).**
    - ***Significance:*** **Synthesizing at the "Worst Case" corner ensures the design meets timing requirements even under worst-case Process, Voltage, and Temperature (PVT) variations (e.g., low voltage, high temperature, slow silicon).**

6.2 Design Constraints (SDC)

A specific set of constraints was applied via the TCL script to ensure the generated netlist meets timing and electrical requirements:

1. **Clock Definitions:**
    - **A main system clock was defined with a period of 100 ns (10 MHz frequency) using the command: `create_clock -name clk -period 100 -waveform {0 50} [get_ports clk]`**
    - **Clock Uncertainty: A skew/jitter margin of 2 ns was applied (`set_clock_uncertainty`) to model real-world clock distribution variations.**
    - **Clock Latency: A source latency of 0.25 ns was assumed.**
2. **I/O Constraints:**
    - **Load: An external capacitive load of 0.1 pF was applied to all output ports (`set_load 0.1`) to simulate the capacitance of wire traces or downstream logic.**
    - **Max Capacitance: A limit of 0.1 pF was set on inputs to ensure the previous stage is not overloaded.**
3. **Design Rule Checks (DRC):**
    - **Max Fanout: Limited to 20. This forces the tool to insert buffers if a single signal (like a Reset or Enable) tries to drive more than 20 logic gates, preventing signal degradation.**
    - **Max Transition: Maximum signal rise/fall time restricted to 0.6 ns to prevent slow transitions that cause short-circuit power consumption.**

6.3 Synthesis Flow

The synthesis followed a standard three-step optimization flow:

1. **Elaboration (`elaborate`): The Verilog RTL (`spi_top_master.v` and `spi_top_slave.v`) was read and converted into a generic, technology-independent logic structure (GTECH). This step checks for syntax errors and infers registers (flip-flops) from the `always @(posedge clk)` blocks.**
2. **Generic Synthesis (`syn_gen`): The generic logic was optimized (Boolean optimization, constant propagation, and dead-code removal) independent of the target library.**
3. **Mapping and Optimization (`syn_map` & `syn_opt`): The generic gates were mapped to specific physical cells in the TSMC 65nm library (e.g., `AN2B1CHD`, `DFQD1`). The tool iteratively optimized the design to meet the setup time (100ns period) and design rules (Fanout < 20).**

6.4 Output Products

Upon successful completion, the synthesis tool generated the following files for the Back-End (Physical Design) phase:

- **Netlist (`.v`): The gate-level Verilog file containing standard cells connectivity.**
- **SDC (`.sdc`): Synopsys Design Constraints file capturing the timing requirements.**
- **SDF (`.sdf`): Standard Delay Format file containing precise cell and interconnect delays for back-annotated simulation.**
- **Reports: Timing reports (Setup/Hold slack) and Area/Power reports.**