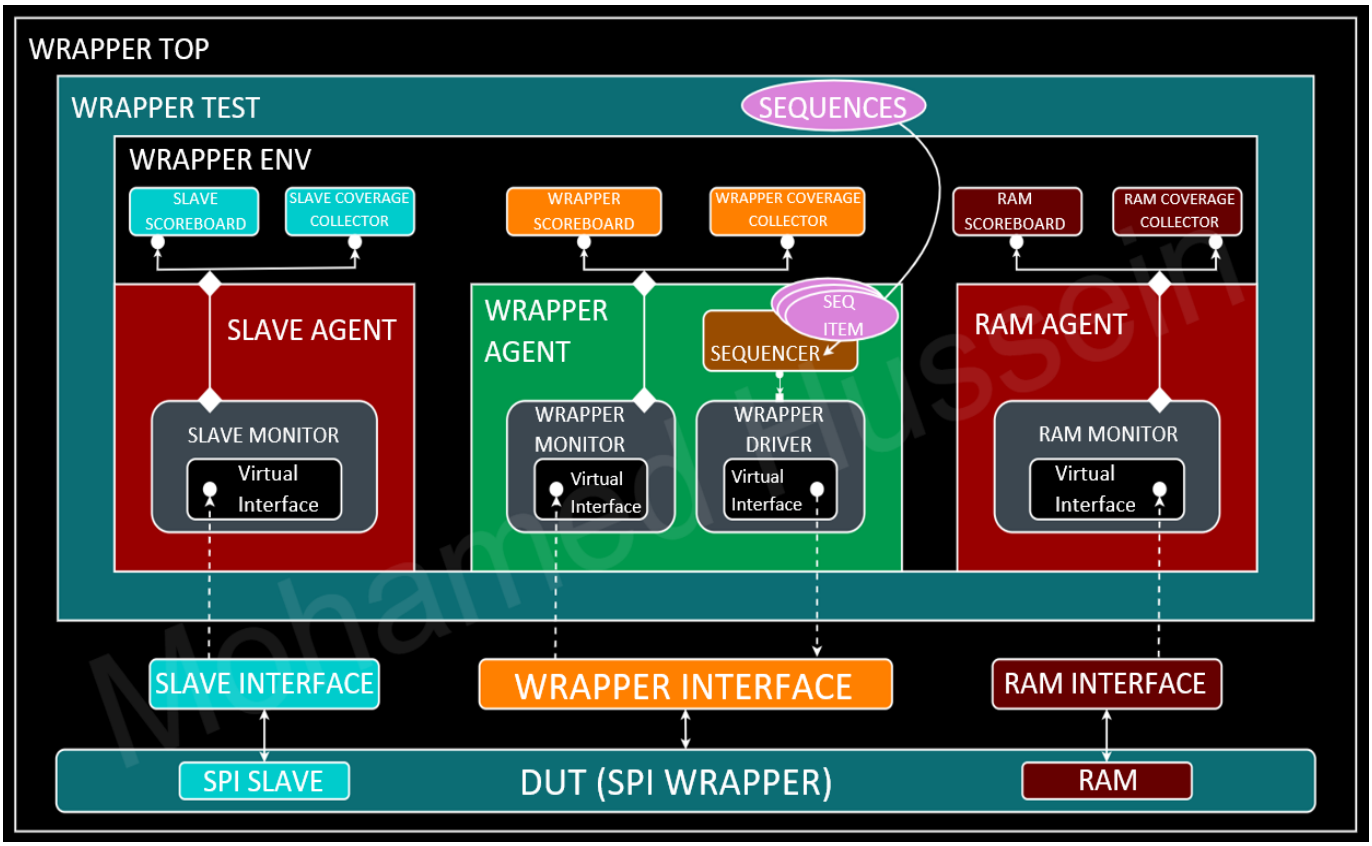
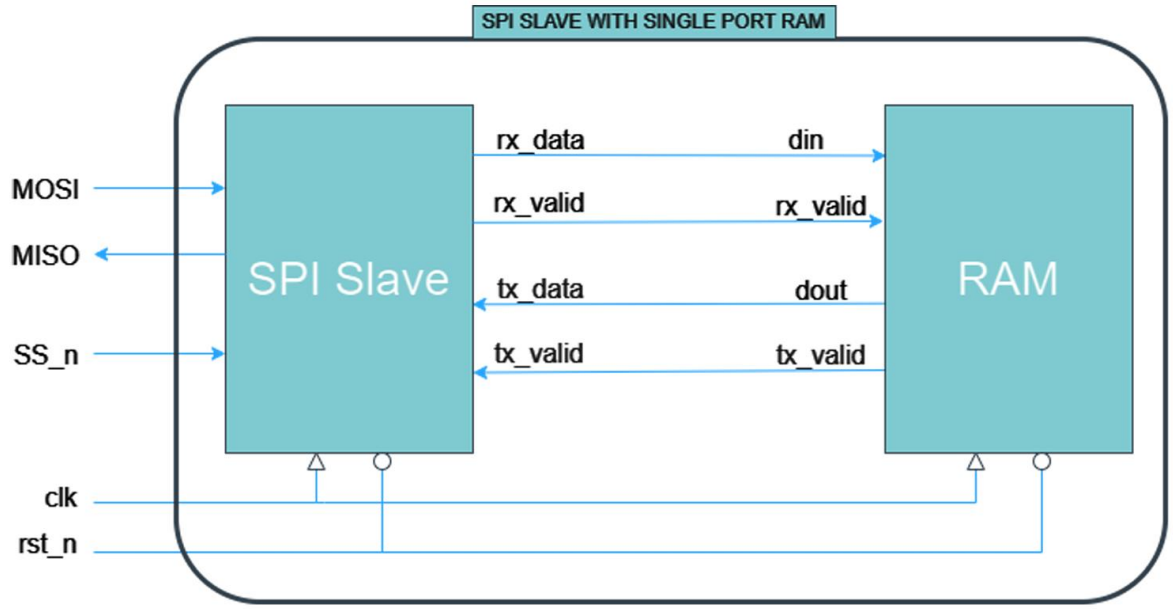
SPI Slave with RAM using UVM



Created by:

Mohamed Ahmed Mohamed Hussein **23/4/25**

Overview:



This project focuses on verifying an SPI Slave module that interfaces with internal RAM using the Universal Verification Methodology (UVM). The SPI Slave receives serial data from an SPI Master, decodes the command, and performs memory operations such as read and write based on the received instructions.

To ensure a thorough and modular verification, a multiple-agent UVM environment has been developed. The testbench is structured to verify both the SPI communication protocol and the internal RAM behavior of the SPI Slave.

Key Highlights:

- **Multiple Agents Environment**:
  The UVM environment includes separate agents for the SPI interface and the memory interface. The SPI agent actively drives and monitors the serial protocol behavior, while the RAM agent passively observes memory operations or can be made active for advanced memory interactions.
- **Active vs Passive Agents**:
  - **Active Agent**: Drives stimulus and monitors responses. In this project, the SPI agent is active as it initiates communication with the DUT (Device Under Test).

    - **Passive Agent**: Monitors signals without driving them. A passive RAM agent can be used to observe and verify memory interactions, ensuring correctness without interfering with DUT behavior.
- **UVM Components:**
  The testbench is built using the standard UVM components, including:
    - `Interface modules`: Define the physical connections to the DUT.
    - `Configuration objects`: Provide test parameters and control flags.
    - `Sequence items & sequences`: Generate transactions and test scenarios for SPI transfers and memory access.
    - `Driver & monitor`: Translate transactions into pin-level signals and observe DUT behavior.
    - `Scoreboard`: Validates expected data vs actual outputs from the SPI Slave and RAM.
    - `Coverage collector`: Ensures protocol coverage, memory operation coverage, and corner cases.
    - `Assertions`: Used to validate protocol rules and detect violations (e.g., incorrect chip select timing or memory access faults).
- **Shared Package for Reusability:**
  A shared SystemVerilog package has been implemented to store signals and common parameters used across multiple components. This approach improves reusability and avoids redundant configurations in different files. Unlike putting all parameters in the configuration object, this package allows easy sharing of constants, types, and common logic between agents and environment components.
- **Test Scenarios:**
  The project includes multiple UVM tests, each targeting a specific aspect of the SPI + RAM behavior:
    - Basic SPI read/write operations
    - Edge cases like simultaneous operations
    - Stress tests with randomized delays and back-to-back transfers

This UVM-based verification environment ensures the SPI Slave with RAM module is **robust**, functionally correct, and protocol-compliant.

Introduction to SPI Communication Protocol

The Serial Peripheral Interface (SPI) is a synchronous serial communication protocol developed by Motorola, widely used for short-distance communication in embedded systems. SPI allows for high-speed data transfer between a master device and one or more peripheral devices. It is particularly valued for its simplicity, efficiency, and low pin count, making it an essential protocol in digital design and embedded systems.

Definition of SPI Communication Protocol

SPI operates in a master-slave configuration, where the master device initiates communication and controls the clock signal. The key signals in SPI communication are:

- **MOSI (Master Out Slave In):** Data line for transmission from master to slave.
- **MISO (Master In Slave Out):** Data line for transmission from slave to master.
- **SCLK (Serial Clock):** Clock signal generated by the master to synchronize data transfer.
- **SS (Slave Select):** Control line for selecting the slave device.

Data transfer in SPI is full-duplex, meaning data can be sent and received simultaneously. The protocol supports different modes of operation, determined by the clock polarity (CPOL) and clock phase (CPHA) settings, allowing flexibility in communication.

Importance of SPI Communication Protocol

SPI is crucial in embedded systems and digital design for several reasons:

1. **Speed:** SPI supports high-speed data transfers, making it suitable for applications requiring quick data exchanges, such as sensors, memory devices, and display controllers.
2. **Simplicity:** The protocol's straightforward design and minimal overhead make it easy to implement and debug, reducing development time and complexity.
3. **Efficiency:** SPI's full-duplex communication and streamlined signal lines contribute to efficient data transmission, minimizing latency and maximizing throughput.
4. **Scalability:** SPI can connect multiple slave devices to a single master, using individual slave select lines, allowing for scalable and expandable system designs.
5. **Versatility:** SPI is compatible with a wide range of devices, from simple sensors to complex microcontrollers, making it a versatile choice for various applications.

Specifications:

- In **write address** state: first received bit by MOSI is **“0”** followed by two other zeros **“2’b00”.**
- In **write data** state: first received bit by MOSI is **“0”** followed by zero then one **“2’b01”.**
- In **read address** state: first received bit by MOSI is **“1”** followed by one then zero **“2’b10”.**
- In **read data** state: first received bit by MOSI is **“1”** followed by two other ones **“2’b11”.**
- If you want to know more about Specs try checking the Design Repository: SPI Slave with single port RAM

Verification Plan:

File	Usage
SPI COVERAGE COLLECTOR	The SPI coverage collector tracks all combinations of SPI inputs (MOSI, SCLK, SS), capturing whether all transfer types (write address, write data, read address, read data) and protocol edges have been exercised. It ensures coverage for command decoding, timing, and correct SPI cycle sequences.

Cover Point Name	Description	Stimulus Generation	Functional Coverage	Functionality Check
cp_tx_valid	Check if the tx_valid signal is exercised	Directed during the simulation	Covers when tx_valid is high.	Observe tx_valid is asserted during a correct SPI transmission
cp_tx_data	Verify full range of tx_data values, split into halves.	Directed during the simulation	Bins: [0:127] (random_1st_half), [128:255] (random_2nd_half).	Ensure tx_data matches expected transmit data
cp_rx_valid	Check that rx_valid goes high when slave receives data	Directed during the simulation	Covers when rx_valid is high	Ensure rx_valid is asserted when a byte is correctly received
cp_ss_n	Verify assertion of ss_n during transaction.	Directed during the simulation	Covers both high and low states.	Monitor that SPI activity only occurs when ss_n is low.
cp_rx_data9	Sample the 9th bit of received data	Directed during the simulation	Covers bit 9 of rx_data	Confirm that bit 9 is correctly interpreted as part of address/data field
cp_rx_data8	Sample the 8th bit of received data.	Directed during the simulation	Covers bit 8 of rx_data	Confirm that bit 8 is correctly interpreted
wr_addr_C	Write Address Transaction Coverage when (rx_data[9:8]=2'b00, rx_valid=1)	Directed during the simulation	Bins when bit 9=0, bit 8=0, and rx_valid=1	Memory write address transaction is captured
rd_addr_C	Read Address Transaction Coverage (rx_data[9:8]=2'b10, rx_valid=1).	Directed during the simulation	Bins when bit 9=1, bit 8=0, and rx_valid=1	Memory read address transaction is captured
wr_data_C	Write Data Transaction Coverage (rx_data[9:8]=2'b01, rx_valid=1)	Directed during the simulation	Bins when bit 9=0, bit 8=1, and rx_valid=1	Memory write data transaction is captured
rd_data_C	Read Data Transaction Coverage (rx_data[9:8]=2'b11, rx_valid=1)	Directed during the simulation	Bins when bit 9=1, bit 8=1, and rx_valid=1	Memory read data transaction is captured
receive_C	Cross coverage between tx_valid and tx_data for receive	Directed during the simulation	Bins where tx_valid=1 and any tx_data value	Data received from SPI slave when tx_valid is asserted

File	Usage
RAM COVERAGE COLLECTORR	The RAM coverage collector ensures that all memory addresses are accessed, all operation types (addr_wr, addr_re, data write, data read) are triggered, and edge cases (e.g., back-to-back reads/writes) are covered. It confirms that each memory access path is thoroughly tested.

Cover Point Name	Description	Stimulus Generation	Functional Coverage	Functionality Check
cp_rx_valid	Sample the rx_valid signal to ensure the receiver side is asserting the valid signal as expected.	Directed during the simulation	Cover both asserted and deasserted cases of rx_valid.	Check that rx_valid accurately reflects valid data reception readiness.
cp_din9	Sample bit 9 of the input data (din[9]) to ensure it toggles correctly and is not stuck.	Directed during the simulation	Cover both 0 and 1 values for din[9].	Verify `din[9]` is handled properly by the RAM during both read and write cycles.
cp_din8	Sample bit 8 of the input data (din[8]) to ensure it's driven correctly during memory transactions.	Directed during the simulation	Cover both 0 and 1 values for din[8].	Verify din[8] is written and read accurately from the RAM.
wr_addr_C	Cross: din[9]=0, din[8]=0, rx_valid=1 → indicates a valid write address.	Directed during the simulation	Cover wr_addr_C bin with specific write address pattern to verify the write enable sequence.	Confirm data is written to correct address when rx_valid is high and control bits match.
rd_addr_C	Cross: din[9]=1, din[8]=0, rx_valid=1 → indicates a valid read address.	Directed during the simulation	Cover rd_addr_C bin with specific read address pattern.	Ensure the RAM is accessed at the intended read address when conditions match.
wr_data_C	Cross: din[9]=0, din[8]=1, rx_valid=1 → indicates a valid write data pattern.	Directed during the simulation	Cover wr_data_C bin to ensure data write condition is exercised.	Check data is correctly written to memory under write data pattern conditions.
rd_data_C	Cross: din[9]=1, din[8]=1, rx_valid=1, tx_valid=1 → indicates valid read data available for transmit.	Directed during the simulation	Cover rd_data_C bin to ensure read data condition is met and data is driven correctly to the output.	Confirm that correct data is transmitted from RAM when all control and data bits match this condition.

File	Usage
SPI WRAPPER SEQUENCE ITEM	This file has the constraints and defines the transaction object (sequence item) that carries the data and control signals for write and read operations to/from the Wrapper . It abstracts the data transfer operations.

Constraint Name	Description	Stimulus Generation	Functional Coverage	Functionality Check
reset_con	A constraint used to control the distribution of rst_n, making reset occur less frequently.	Applied randomly during or between SPI transactions		Ensures that rst_n is mostly high with rare assertions, simulating rare reset conditions during test runs.
ss_n_con	A constraint used to make ss_n high only at the end of each transaction (state_finished).	Triggered when state finishes		Ensures that ss_n is asserted only at the end of each SPI state, respecting chip select timing protocol.
MOSI_write_con	A constraint to make MOSI set the first bit to 0 in WRITE state.	When ns == WRITE and write_constraint is set		Validates write command framing by ensuring MOSI starts with 0 in write operations.
MOSI_read_add_con	A constraint to make MOSI set the first two bits to 2'b10 in READ_ADD state.	When ns == READ_ADD and read_add_constraint is set		Ensures correct framing of read address by enforcing MOSI = 2'b10 during read address operations.
MOSI_read_data_con	A constraint to make MOSI set the first two bits to 2'b11 in READ_DATA state.	When ns == READ_DATA and read_data_constraint is set		Verifies correct read data phase initiation with MOSI = 1 in read data state.
MOSI_con	A constraint that makes MOSI more likely to be 0 than 1, prioritizing writes over reads.	Applied across all SPI transactions		Verifies that write operations (MOSI = 0) occur more frequently than reads (MOSI = 1) as intended.

File	Usage
SPI SCOREBOARD	The scoreboard is responsible for checking the correctness of the SPI Slave behavior. It compares the expected serial-to-parallel converted data and memory transactions with the actual outputs observed during simulation. It validates whether the SPI protocol is respected and all bytes are accurately written to memory.

File	Usage
RAM SCOREBOARD	The RAM scoreboard verifies memory behavior, ensuring correct address decoding, read/write operations, and data retention. It compares the expected values (tracked internally during testbench operation) against actual dout values and checks correct updates during rx_valid operations.

File	Usage
SPI SVA	The SPI assertions file (SPI_SVA.sv) contains checks to ensure the protocol is followed—such as correct SS edge triggering, clock alignment, and valid data shift timing. It ensures that SPI decoding is valid and flags violations of protocol timing or signal integrity during transactions..

Next State Assertions:

Assertion Name	Description	Stimulus Generation	Functional Coverage	Functionality Check
rst_n_ns_a	After reset (rst_n = 0), FSM should go to IDLE state.	Directed during the simulation		DUT.cs must be DUT.IDLE one cycle after rst_n is low.
idle_to_chk_cmd_a	FSM should move from IDLE to CHK_CMD when ss_n = 0.	Directed during the simulation		DUT.cs must be DUT.CHK_CMD in the next cycle.
chk_cmd_to_write_a	FSM moves from CHK_CMD to WRITE when MOSI = 0.	Directed during the simulation		DUT.cs must become DUT.WRITE.
chk_cmd_to_read_add_a	FSM moves from CHK_CMD to READ_ADD when MOSI = 1 and ADD_DATA_checker = 1.	Directed during the simulation		DUT.cs must become DUT.READ_ADD.
chk_cmd_to_read_data_a	FSM moves from CHK_CMD to READ_DATA when MOSI = 1 and ADD_DATA_checker = 0.	Directed during the simulation		DUT.cs must become DUT.READ_DATA.
write_hold_a	FSM remains in WRITE when ss_n = 0.	Directed during the simulation		No state transition occurs; DUT.cs remains DUT.WRITE.
write_to_idle_a	FSM moves from WRITE to IDLE when ss_n = 1.	Directed during the simulation		DUT.cs must become DUT.IDLE.
read_add_hold_a	FSM remains in READ_ADD when ss_n = 0.	Directed during the simulation		DUT.cs must stay DUT.READ_ADD.
read_add_to_idle_a	FSM moves from READ_ADD to IDLE when ss_n = 1.	Directed during the simulation		DUT.cs must become DUT.IDLE.
read_data_hold_a	FSM remains in READ_DATA when ss_n = 0.	Directed during the simulation		DUT.cs must remain DUT.READ_DATA.
read_data_to_idle_a	FSM moves from READ_DATA to IDLE when ss_n = 1.	Directed during the simulation		DUT.cs must become DUT.IDLE.

Output Logic Assertions:

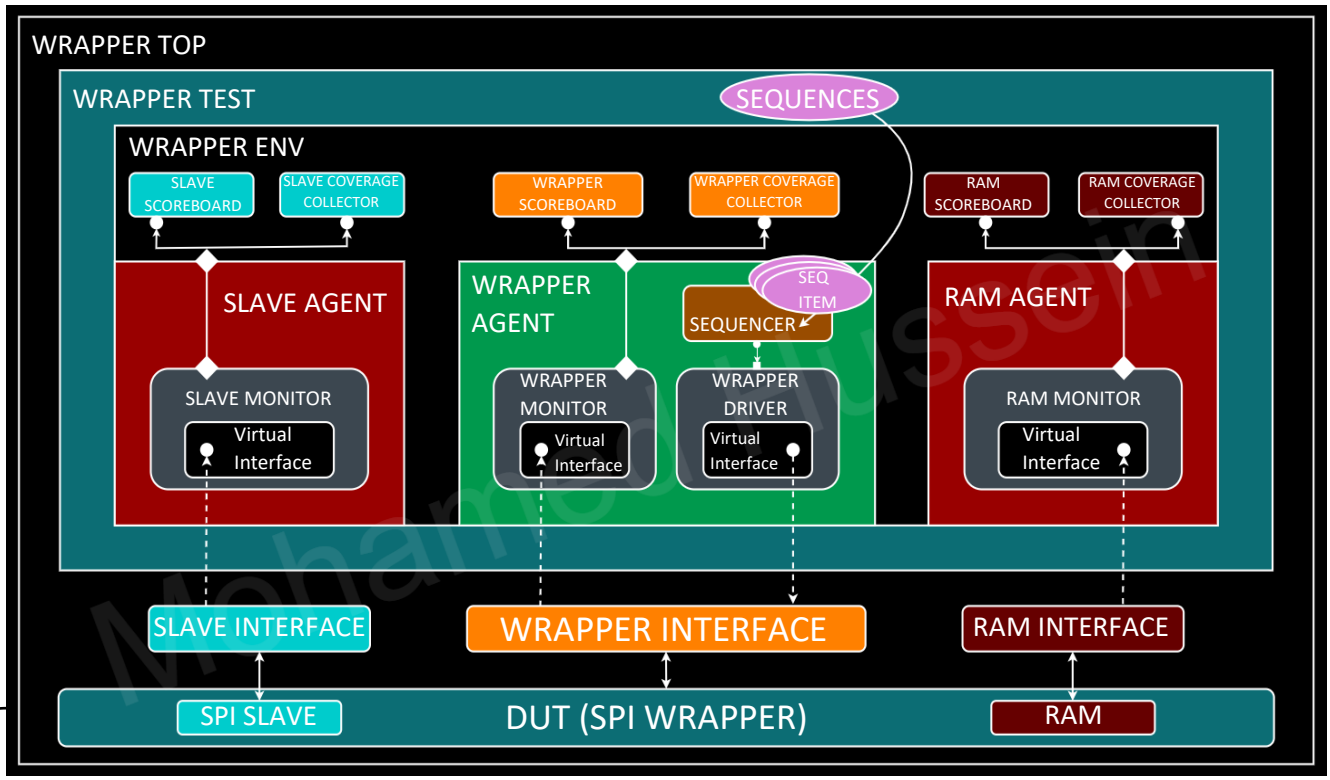
Assertion Name	Description	Stimulus Generation	Functional Coverage	Functionality Check
rst_n_o_a	Checks that rx_data == 0, rx_valid == 0, and MISO == 0 when rst_n is deasserted (active low reset).	Directed during the simulation		Ensure all output signals reset to default values during reset.
idle_o_a	Ensures rx_valid stays low when SPI slave is in IDLE state.	Directed during the simulation		Confirm that rx_valid remains 0 during the entire IDLE period.
write_o_a	On DUT.cs = WRITE and counter1 = 4'hF, asserts rx_valid rises, rx_data == bus, then rx_valid falls.	Directed during the simulation		Observe correct timing of rx_valid (1 cycle high), and rx_data equal to bus.
read_add_o_a	Similar to write, but in READ_ADD state: asserts rx_valid rises, rx_data == bus, ADD_DATA_checker falls, and rx_valid falls after that.	Directed during the simulation		Validate the sequence of signals: rx_valid rise, rx_data == bus, ADD_DATA_checker fall, rx_valid fall.
read_data_o_a_1	In READ_DATA with counter1 = 4'hF, asserts rx_valid rises and rx_data == bus, then rx_valid falls.	Directed during the simulation		Confirm proper one-cycle rx_valid and correct rx_data output.
read_data_o_a_2	When DUT.cs = READ_DATA and tx_valid = 1, MISO must equal correct bit from tx_data indexed by counter2.	Directed during the simulation		Check that MISO equals tx_data[counter2] during this state.
read_data_o_a_3	When in READ_DATA state and counter2 = 3'b111, asserts ADD_DATA_checker is high.	Directed during the simulation		Ensure that ADD_DATA_checker becomes high exactly at counter2 == 3'b111.

File	Usage
RAM SVA	The RAM assertions (RAM_SVA.sv) enforce correct behavior on the RAM module. This includes address updates, correct data writes and reads, and that tx_valid asserts only during read operations. These assertions ensure internal control signals are consistent with the expected operation type..

Assertion Name	Description	Stimulus Generation	Functional Coverage	Functionality Check
rst_n_a	Checks that outputs (dout, tx_valid) and internal addresses (addr_wr, addr_re) reset correctly when rst_n is low.	Directed during the simulation		Verify dout == 8'h00, tx_valid == 0, addr_wr == 0, and addr_re == 0.
addr_wr_a	Validates that addr_wr updates correctly when rx_valid is high and MSBs of din are 2'b00 (write address operation).	Directed during the simulation		Confirm addr_wr == addr in the next cycle.
addr_re_a	Validates that addr_re updates correctly on read address operation (din[9:8] = 2'b10).	Directed during the simulation		Confirm addr_re == addr in the next cycle.
w_data_a	Ensures that write data goes into the correct memory address when rx_valid is high and operation type is 01 (write data).	Directed during the simulation		Verify DUT.memory[addr_wr] == data.
r_data_a	Verifies dout outputs correct memory content when read data operation (din[9:8] = 2'b11) occurs.	Directed during the simulation		Ensure dout == memory[addr_re] from previous cycle.
tx_valid_a	Ensures tx_valid goes high during a read data operation (din[9:8] = 2'b11).	Directed during the simulation		Confirm tx_valid == 1 in the next cycle.
not_tx_valid_a	Ensures tx_valid stays low for all operations except read data (din[9:8] != 2'b11).	Directed during the simulation		Confirm tx_valid == 0 in the next cycle.
rx_valid_a	Confirms dout holds its previous value when rx_valid is low, even if din[9:8] = 2'b11.	Directed during the simulation		Ensure dout remains unchanged from the previous cycle.

SPI Wrapper:

1. UVM Structure



Note: You can click on any component or object in this UVM structure and it will get you directly to its code!

How It Works?

1. Top Module (e.g., TOP.sv)

This is the highest level of the verification environment. It instantiates the **Design Under Test (DUT)** and connects it to the UVM testbench. The top module also instantiates the **interface**, linking it with the DUT to ensure that the signal connections between the testbench and DUT are properly established.

2. Interface (e.g., INTERFACE.sv)

The interface groups all signals that communicate with the DUT, such as clk, rst, control signals, and data ports. It provides a structured way to access these signals within different UVM components like the driver, monitor, and scoreboard, facilitating cleaner and more maintainable code.

3. UVM Testbench Flow

The UVM testbench automates the verification flow by generating stimulus, applying it to the DUT, monitoring DUT behavior, and analyzing the results.

3.1 Configuration (e.g., CONFIG.sv)

The configuration object holds **environment parameters** (e.g., data width, timeout values, protocol-specific settings) and distributes them across components to maintain consistency.

Note: In this design, **shared signals used across multiple components are included in a dedicated** `shared_package` rather than in the configuration object. This promotes **modularity and reusability**, allowing multiple components to easily import and use common definitions without duplicating code.

3.2 Sequences and Sequence Items

- **Sequence Item (`SEQUENCE_ITEM.sv`)**
  Defines the transaction object which carries data and control signals. It abstracts protocol-level behavior in a reusable way.
- **Sequences (`RESET_SEQUENCE.sv`, `WRITE_SEQUENCE.sv`, `READ_SEQUENCE.sv`, `MAIN_SEQUENCE.sv`, etc.)**
  Collections of sequence items that model higher-level operations or protocol flows. Each sequence represents a scenario (e.g., reset, write, read, or mixed operations). The **main sequence** typically orchestrates how and when these sub-sequences are executed.

3.3 Sequencer (e.g., SEQUENCER.sv)

The sequencer acts as a **command center** that provides sequence items to the driver. It manages timing and order of operations, ensuring a controlled flow of stimulus from sequences to the DUT.

3.4 Driver (e.g., DRIVER.sv)

The driver translates sequence items into pin-level activity by driving DUT signals via the interface. It handles low-level protocol details and ensures accurate, cycle-by-cycle control of stimulus.

3.5 Monitor (e.g., MONITOR.sv)

The monitor **passively observes** signal activity through the interface without affecting DUT behavior. It captures transactions, checks protocol compliance, and forwards information to components such as the scoreboard and coverage collector.

4. Analysis Components

4.1 Scoreboard (e.g., SCOREBOARD.sv)

The scoreboard checks the **functional correctness** of the DUT by comparing actual outputs (observed by the monitor) with expected values (from a reference model or predicted logic). It verifies correctness of behavior under all applied stimuli.

4.2 Coverage Collector (e.g., COVERAGE_COLLECTOR.sv)

This component gathers **functional coverage** metrics to assess how thoroughly the DUT has been exercised. It tracks conditions like combinations of inputs, state transitions, corner cases, and protocol-specific behaviors.

5. Environment (e.g., ENV.sv)

The environment instantiates and connects all core UVM components such as agents, scoreboard, and coverage collector. It forms the **testbench backbone**, ensuring all components are initialized and communicate correctly.

- ## Agent (e.g., AGENT.sv)
  The agent is a container for the sequencer, driver, and monitor. It manages their instantiation and interaction.

  There are **two types of agents**:

  - **Active Agent**: Contains both **sequencer and driver**, and generates stimulus to drive the DUT, **Wrapper Agent** in our case!
  - **Passive Agent**: Contains only a **monitor**, and is used to observe DUT activity without driving it—useful when working with externally driven DUT interfaces or for verification at system level, both **Slave** & **RAM** Agents in our case!

6. Assertions (e.g., SVA.sv)

The assertions module defines **SystemVerilog Assertions (SVA)** to check for protocol compliance and design properties during simulation. Assertions are runtime checks for issues like:

- Invalid transitions
- Protocol violations
- Timing errors
- Data integrity

These provide an **immediate indication of violations**, improving debug efficiency.

7. Test (e.g., TEST.sv)

The test layer defines the overall **test strategy** by extending the UVM test class. It creates the environment, configures it, and launches the appropriate sequences. Multiple test classes can be created to verify:

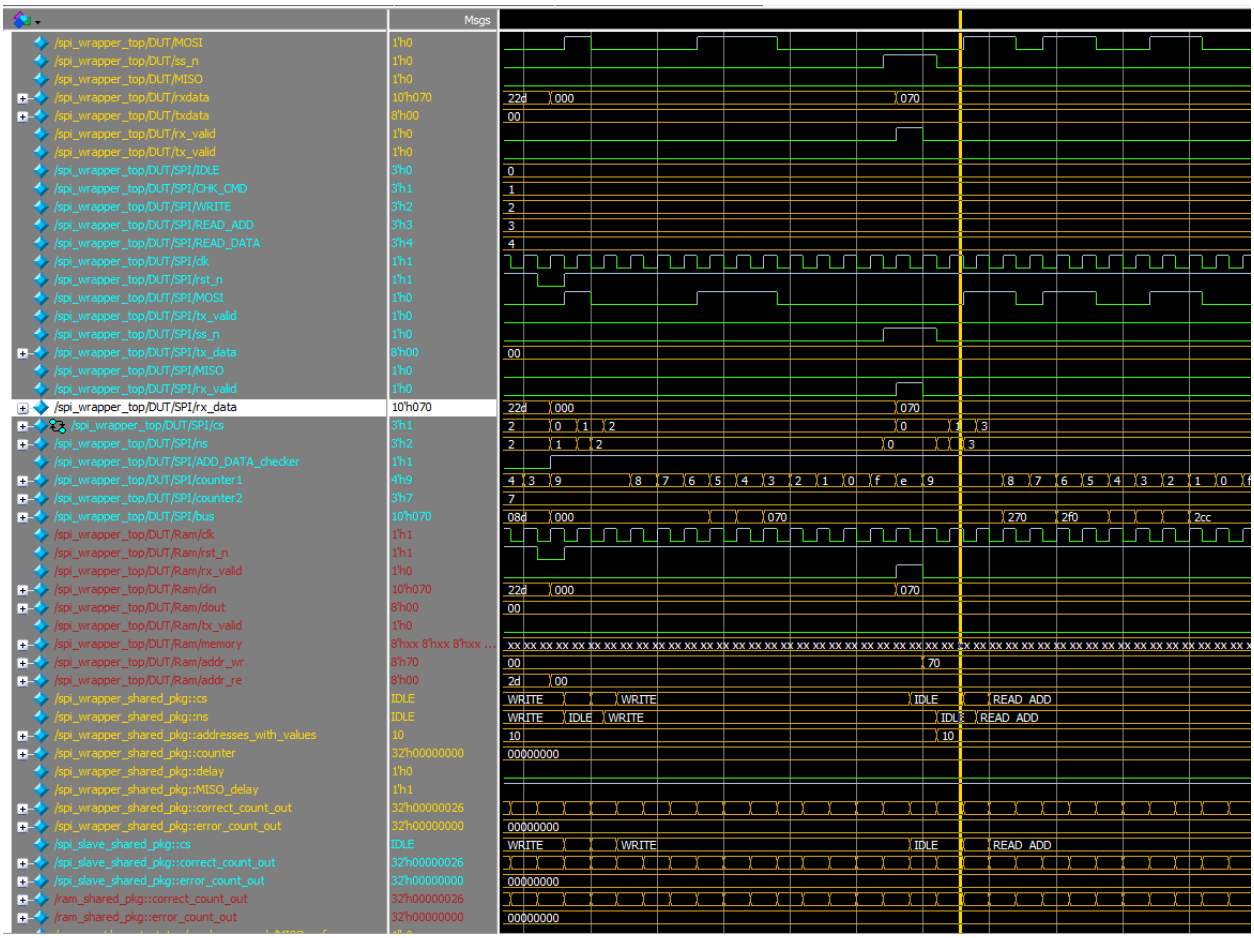
- Basic functionality
- Corner cases
- Stress conditions
- Error handling

2. Verifying functionality

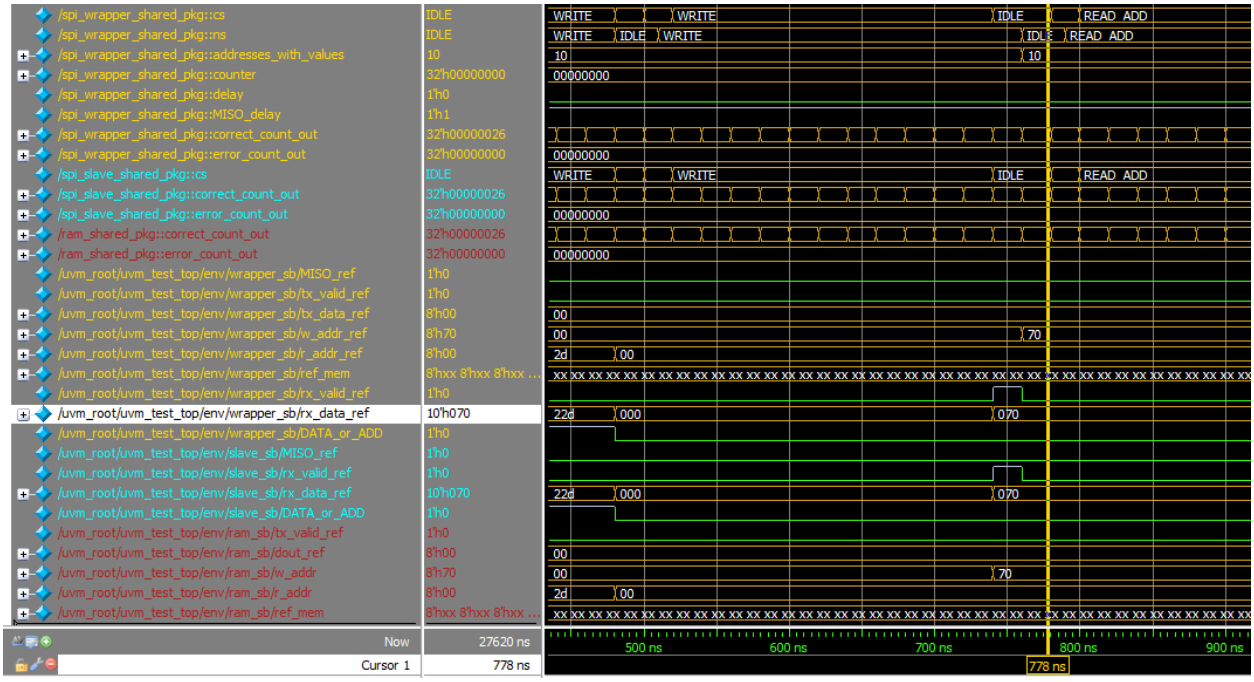
In this section we will compare a snapshot from the **design** waveform to another from the **scoreboard** waveform to make sure our verification went correctly

- **Write Address State:**

Design snapshot:



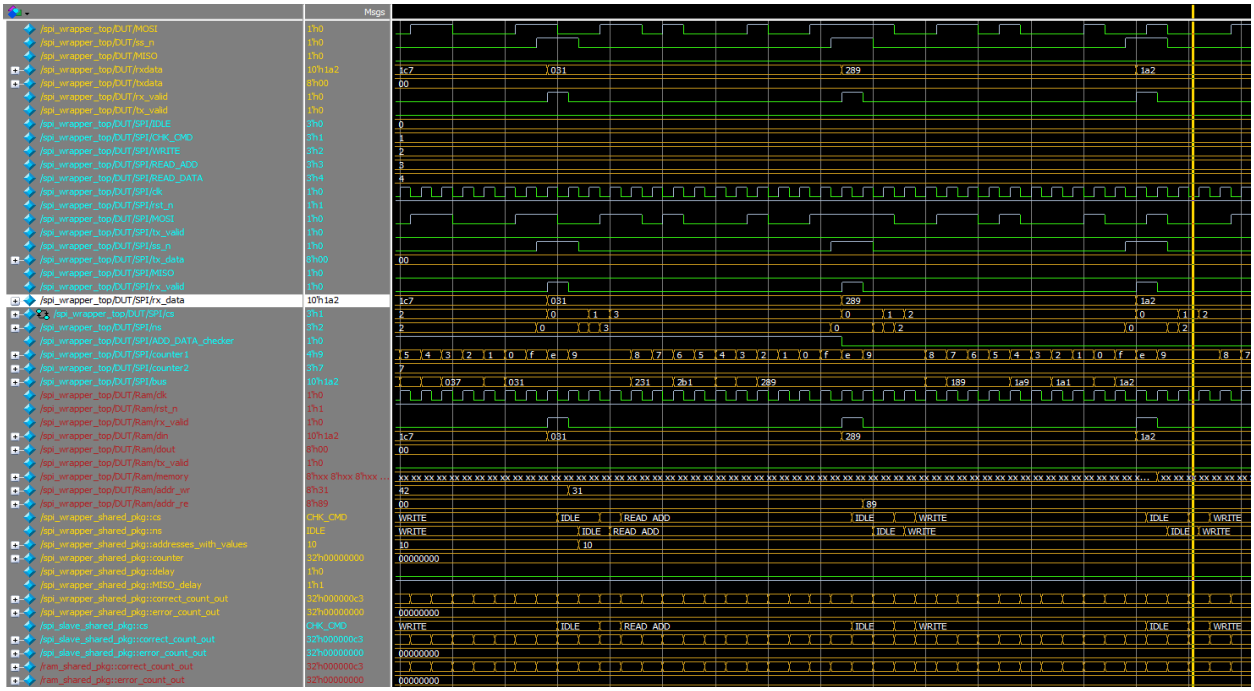
Scoreboard snapshot:



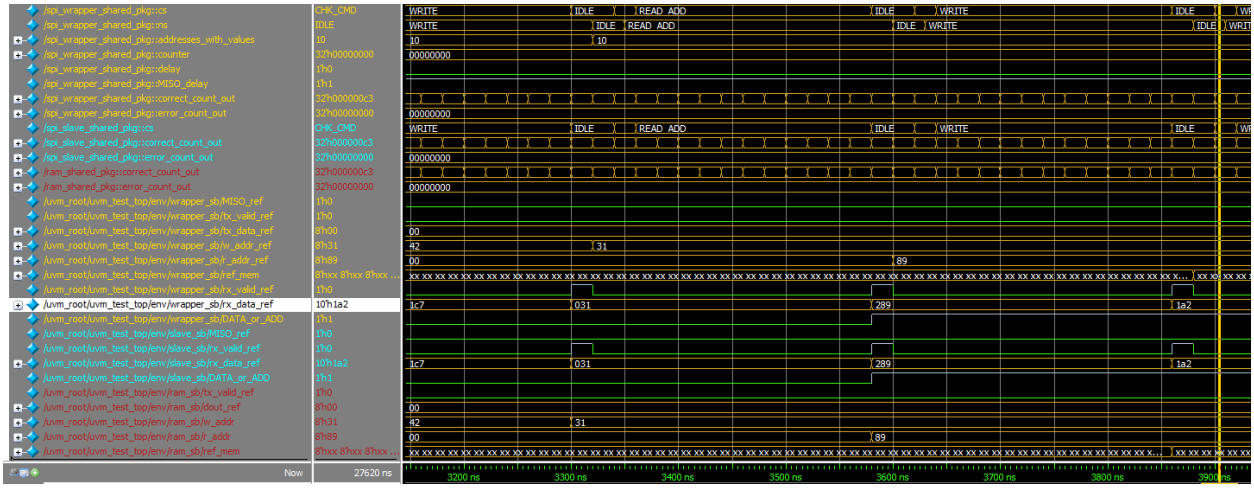
From design and scoreboards we deduce: Rx_data in Design (DUT & SPI) is the same as rx_data_ref in scoreboard (both Wrapper & Slave) and also rx_valid is the same as rx_valid_ref, and the states are the same, notice that the Most Significant two Bits of rx_data[9:8] = **2'b00** as the state is write **address** (from Specs), so **addr_wr** has the value 0x70 in both design and scoreboard of RAM

- **Write Data State:**

Design snapshot:



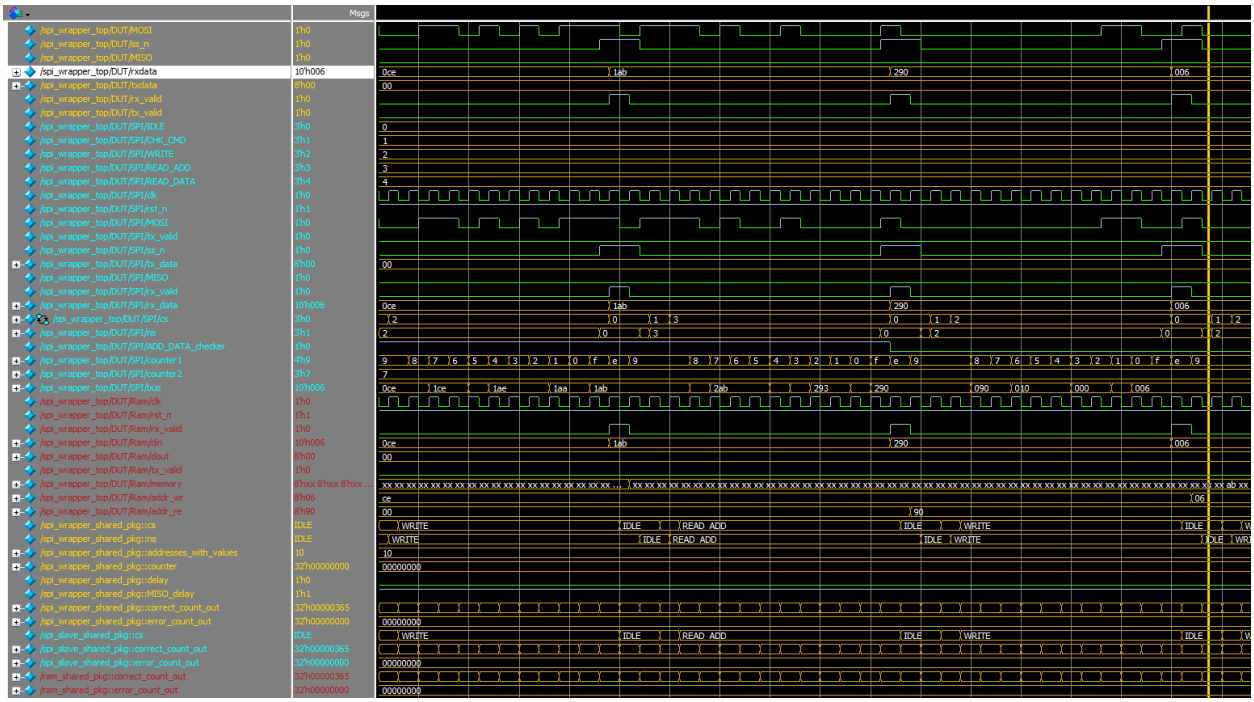
Scoreboard snapshot:



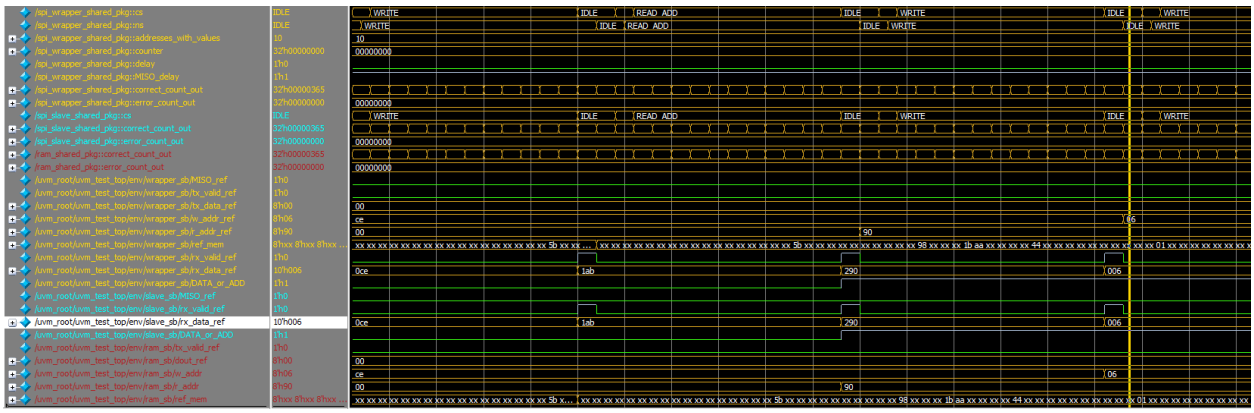
From design and scoreboards we deduce: Rx_data in Design (DUT & SPI) is the same as rx_data_ref in scoreboard (both Wrapper & Slave) and also rx_valid is the same as rx_valid_ref, and the states are the same, notice that the Most Significant two Bits of rx_data[9:8] = **2'b01** as the state is write **data** (from Specs), so we will notice that some value (0xa2) has been written in memory (in address 0x31)

- **Random snapshot for Write State:**

Design snapshot:

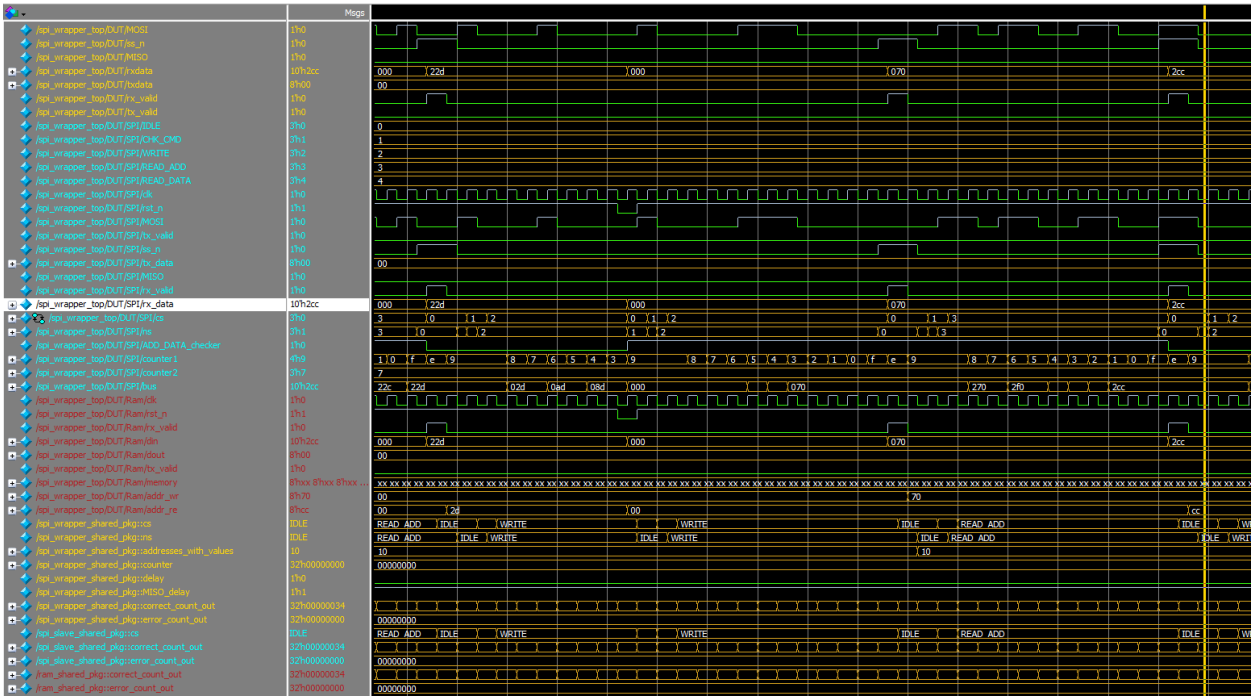


Scoreboard snapshot:

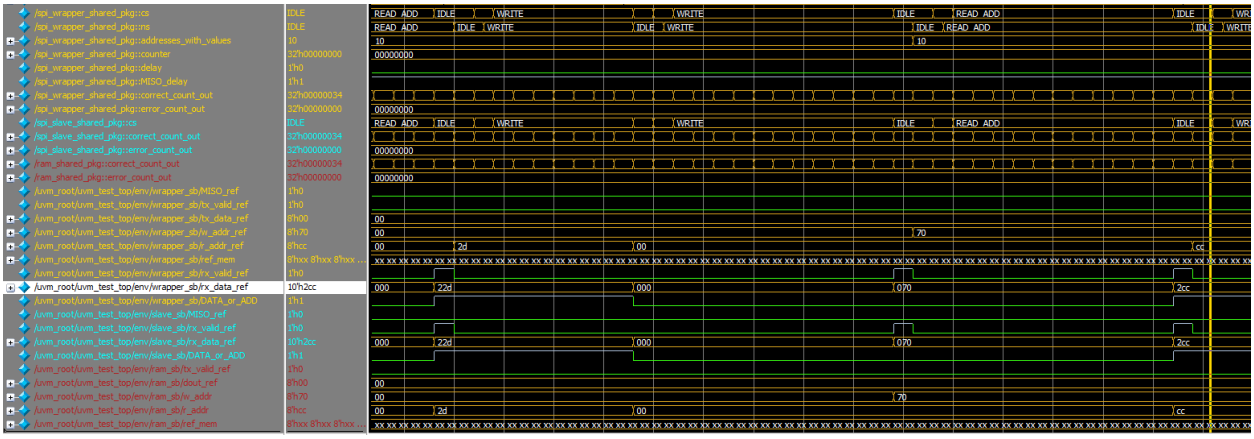


- **Read Address state:**

Design snapshot:



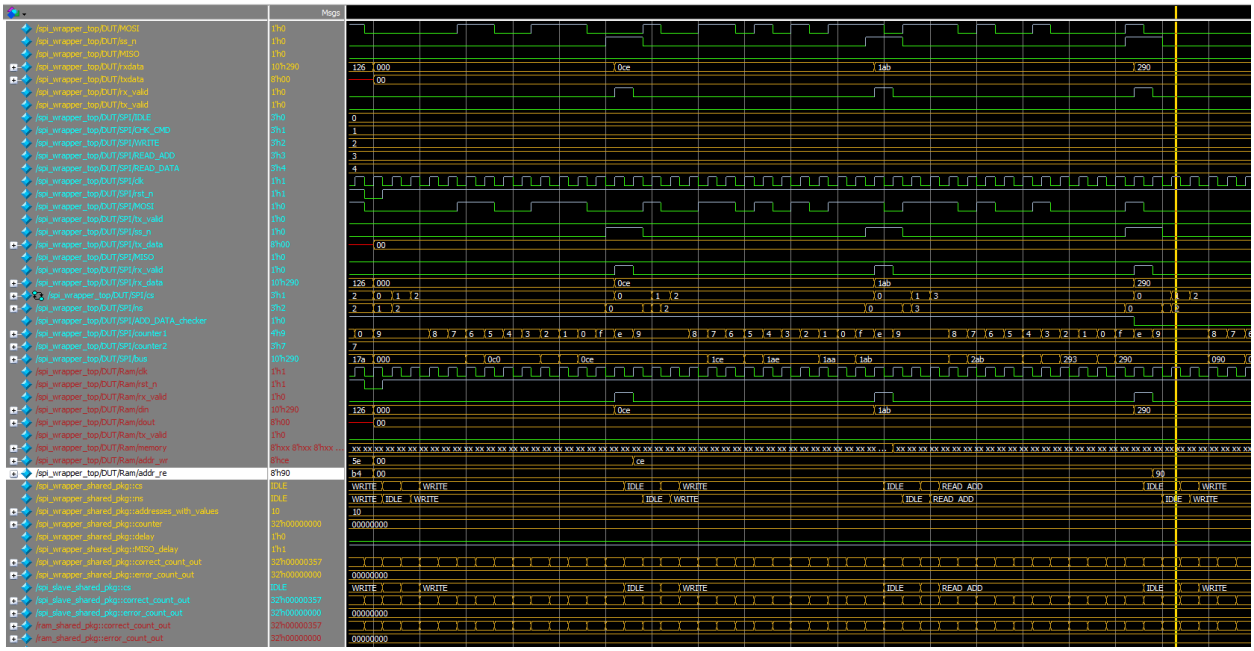
Scoreboard snapshot:



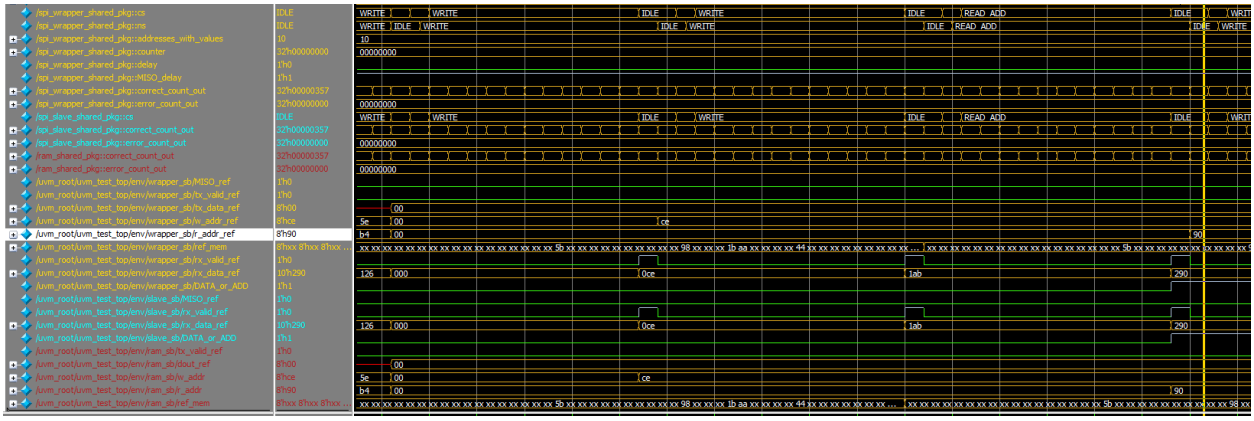
From design and scoreboards we deduce: Rx_data in Design (DUT & SPI) is the same as rx_data_ref in scoreboard (both Wrapper & Slave) and also rx_valid is the same as rx_valid_ref, and the states are the same, notice that the Most Significant two Bits of rx_data[9:8] = **2'b10** as the state is Read **address** (from Specs), also **addr_rd** has the value 0xcc in both design and scoreboard of RAM

- **Random snapshot for Read Address state:**

Design snapshot:



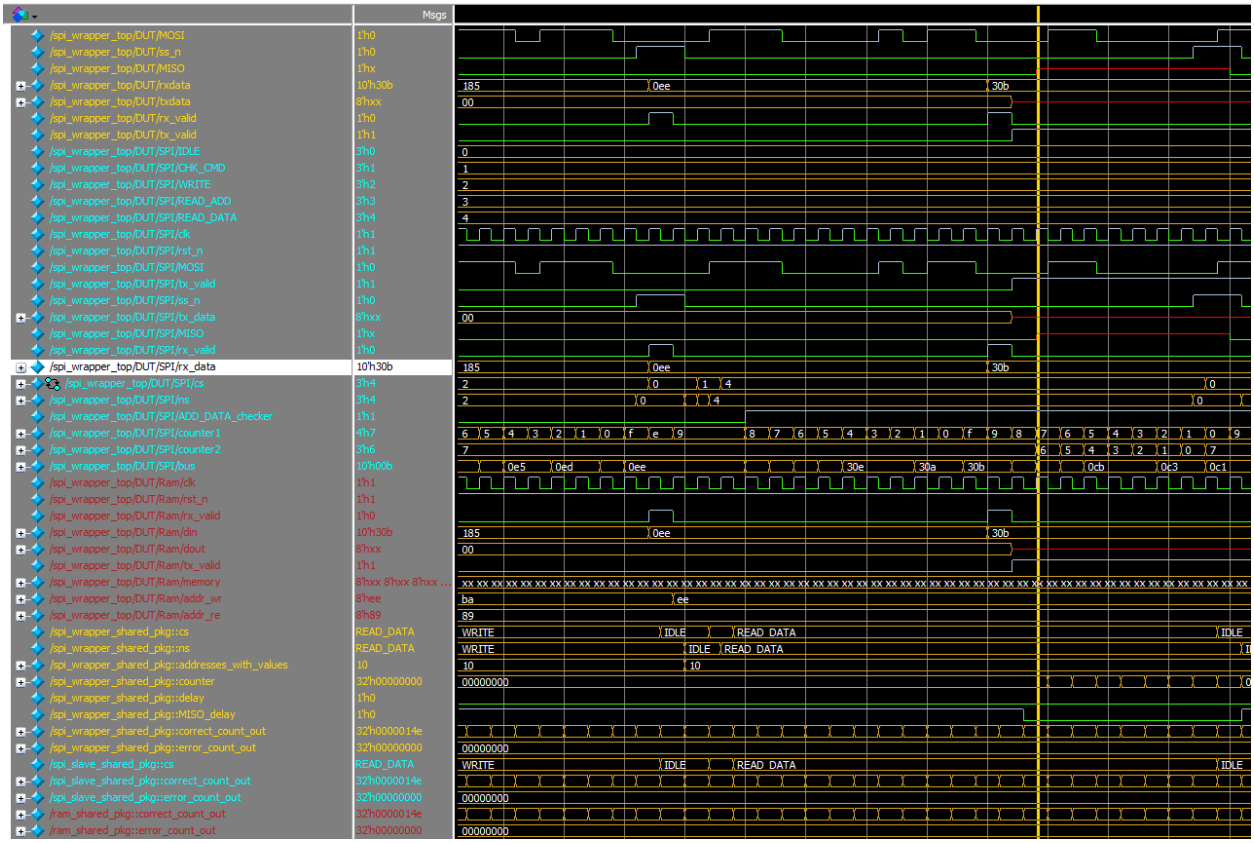
Scoreboard snapshot:



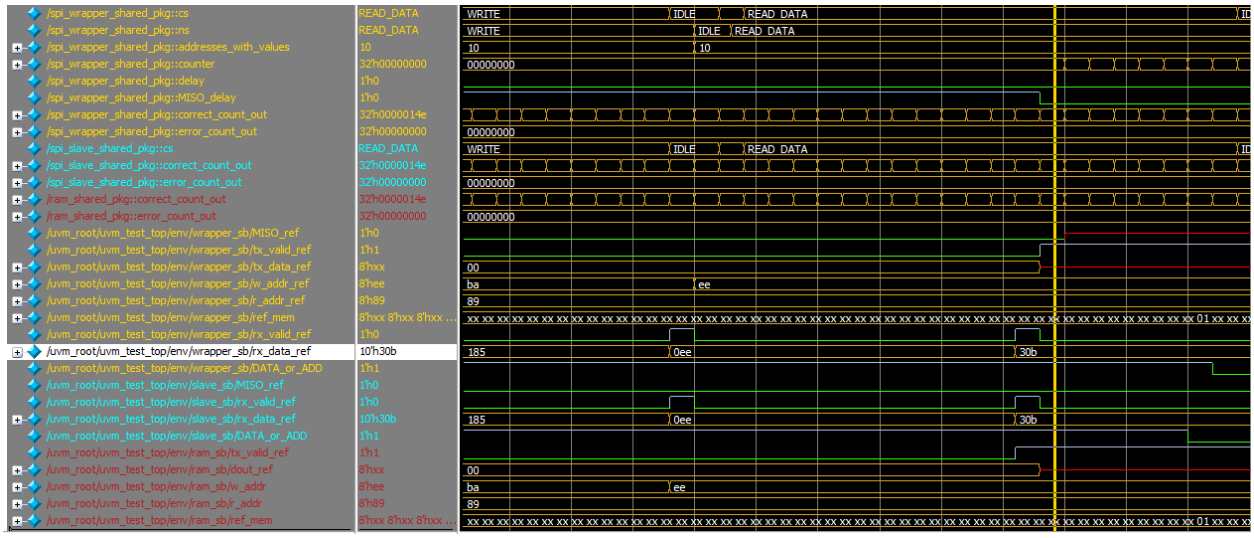
you will find that addr_re as well as r_addr_ref (Wrapper & RAM) having the same value (0x90)

- **Read Data state (Randomized Sequence):**

Design snapshot:

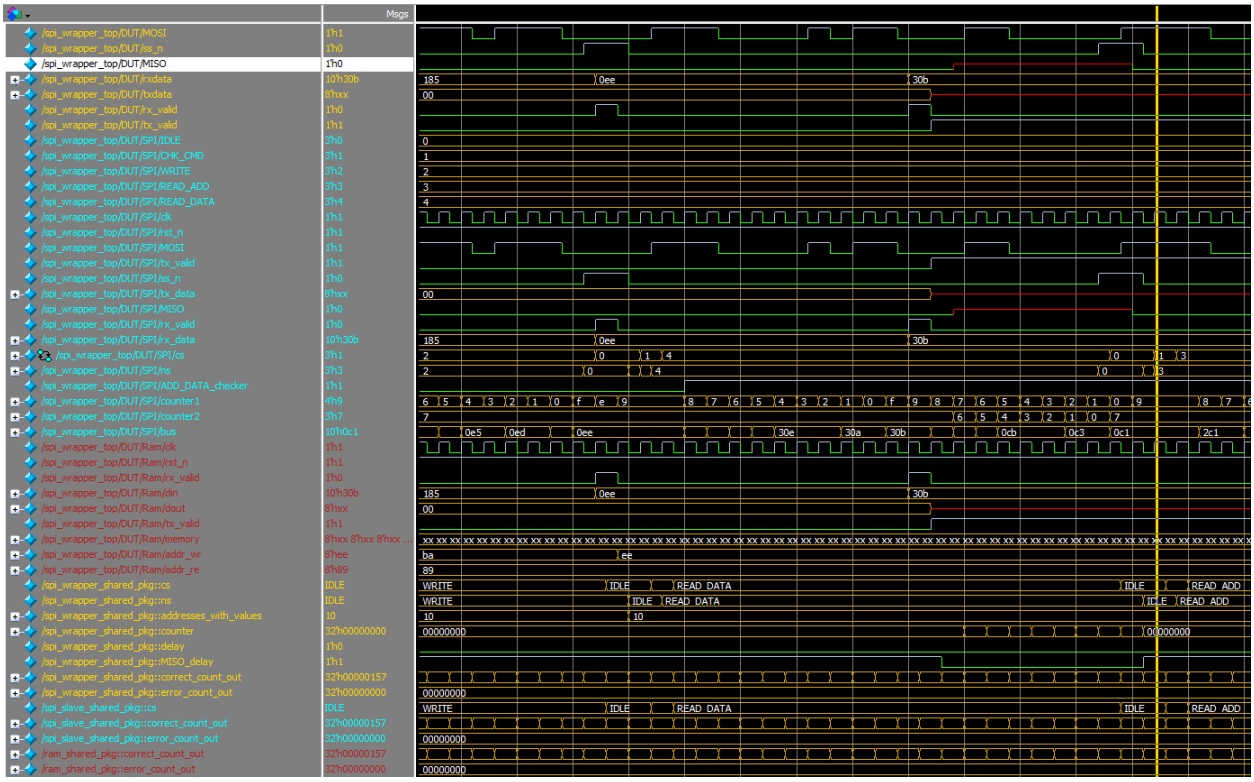


Scoreboard snapshot:

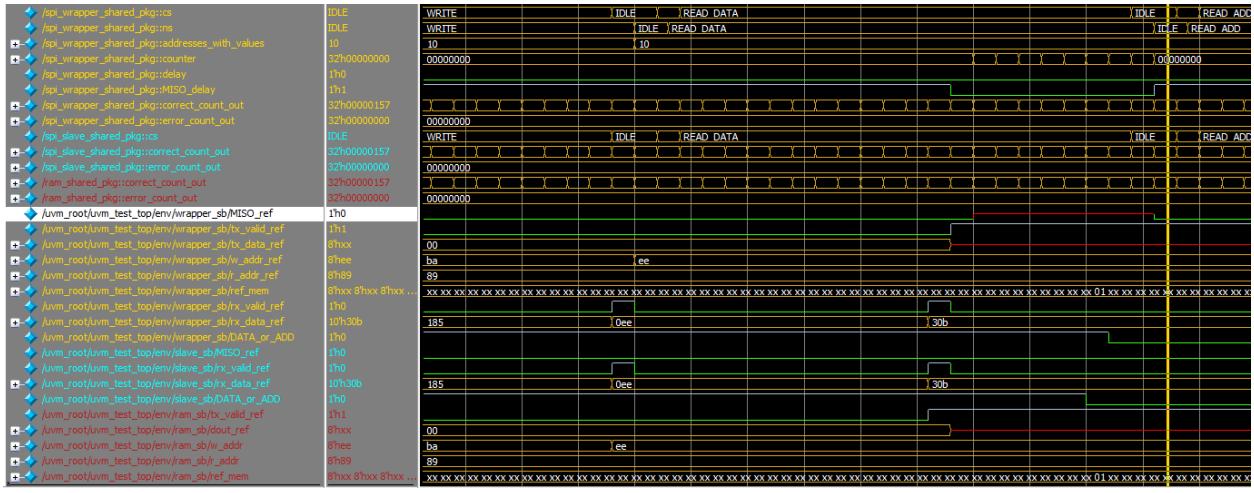


From design and scoreboards we deduce: In Read Data state, rx_data is sent correctly but we care about the **MISO** signal in this state (see the next comparison), Most Significant two Bits of rx_data[9:8] = **2'b11** as the state is Read **Data** (from Specs), since the write address is **not the same** as read address as it is **randomized** test so the data read from the memory will often be **unknown**

Design snapshot:



Scoreboard snapshot:

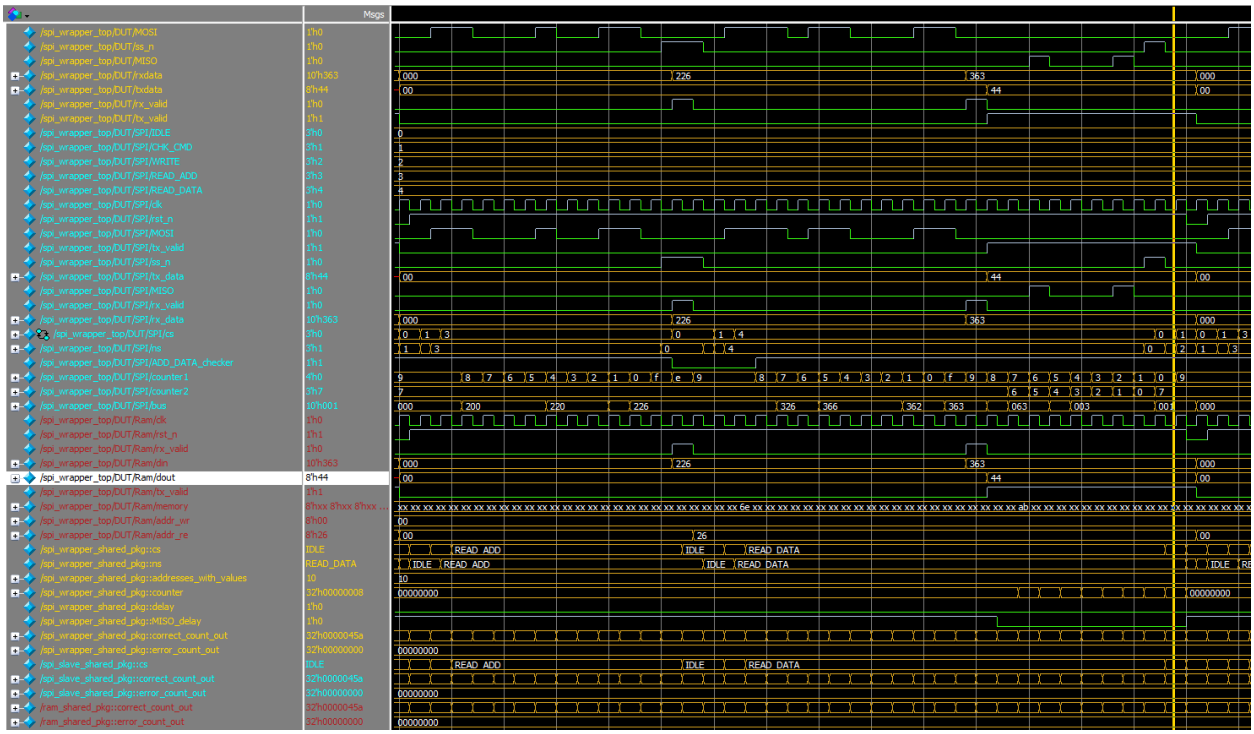


After rx_data is sent, we will notice the same behavior for MISO signal and MISO_ref (Wrapper) signal which indicates the correctness of the design, but why the MISO_ref (Slave) is not the same as MISO_ref(Wrapper)?

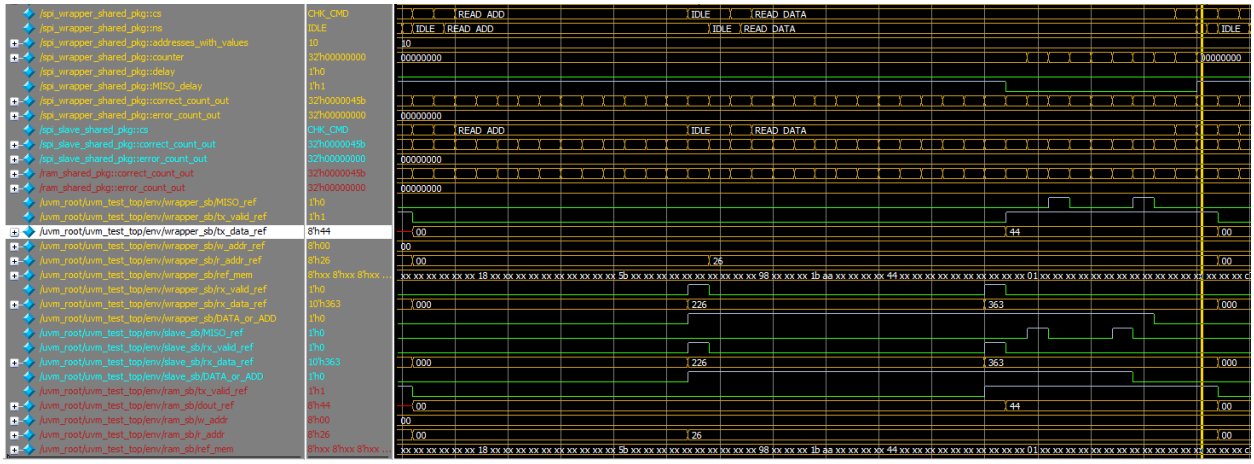
As in SPI_Slave scoreboard we **don't include a memory** as we are verifying the logic of SPI Slave alone so we will count on MISO_ref signal which is in the Wrapper scoreboard as **it has a memory**

- **Read Data state (Directed Sequences):**

Design snapshot:

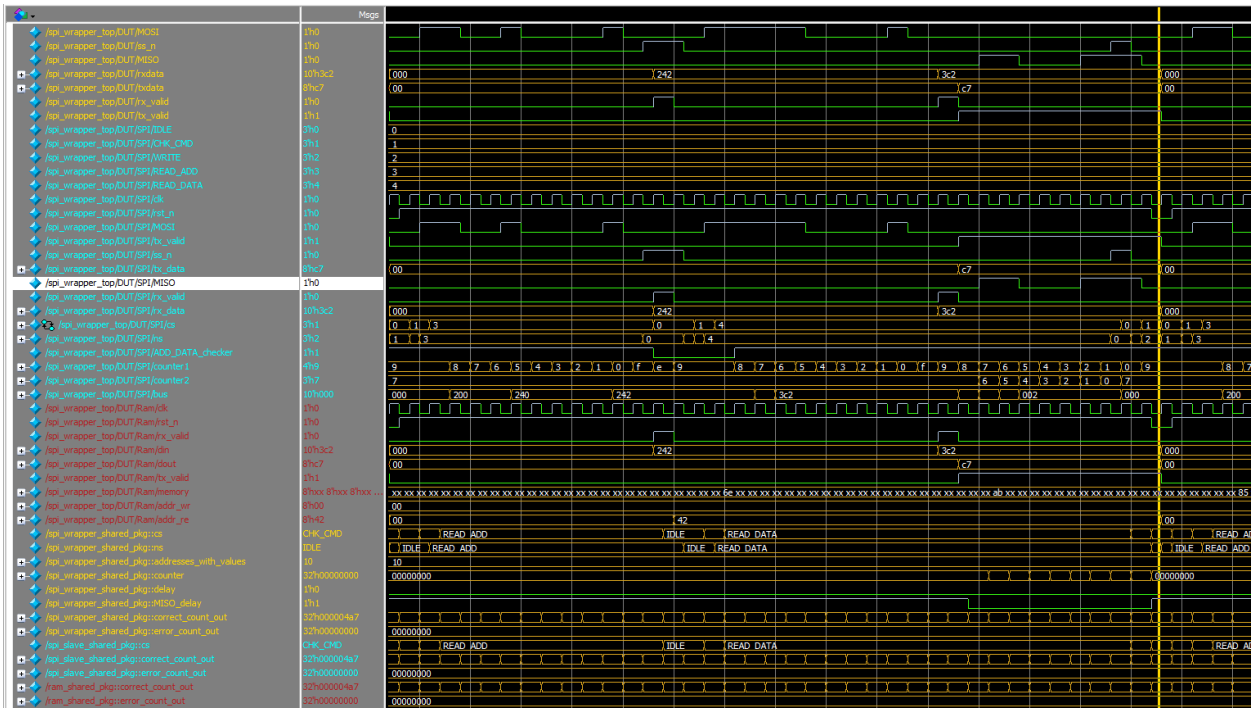


Scoreboard snapshot:

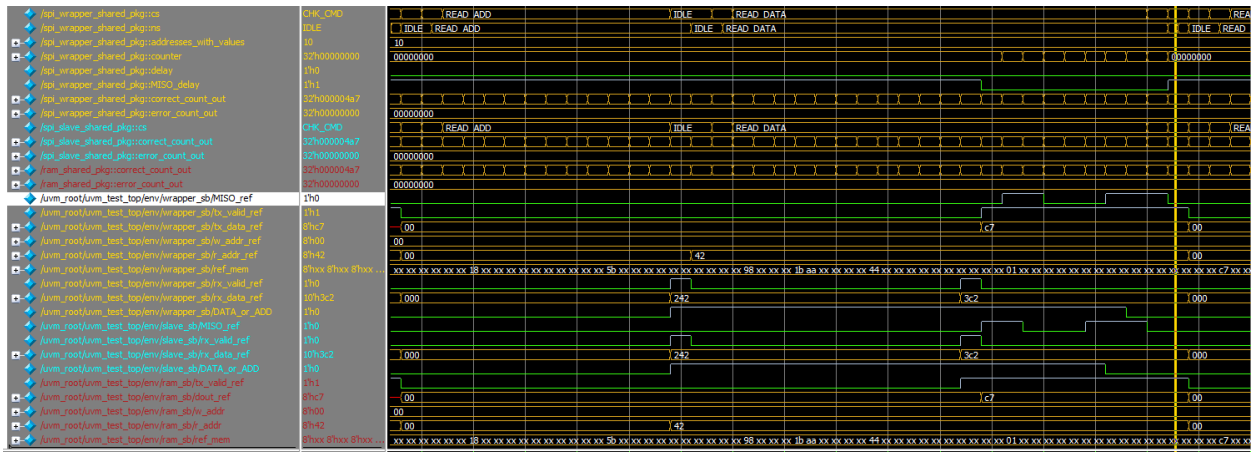


Since the the read address **is not the same** as write address so we usually expect an **unknown behavior** In the MISO signal (output signal), so I created a nearly directed tests to read a real value from the memory (made read address **the same** as any write address **has a value in the memory**), from comparison we deduce: dout has the same value (0x44) as Tx_data (Wrapper) & dout_ref (RAM) signal which will be out on MISO signal (see the next comparison)

Design snapshot:



Scoreboard snapshot:

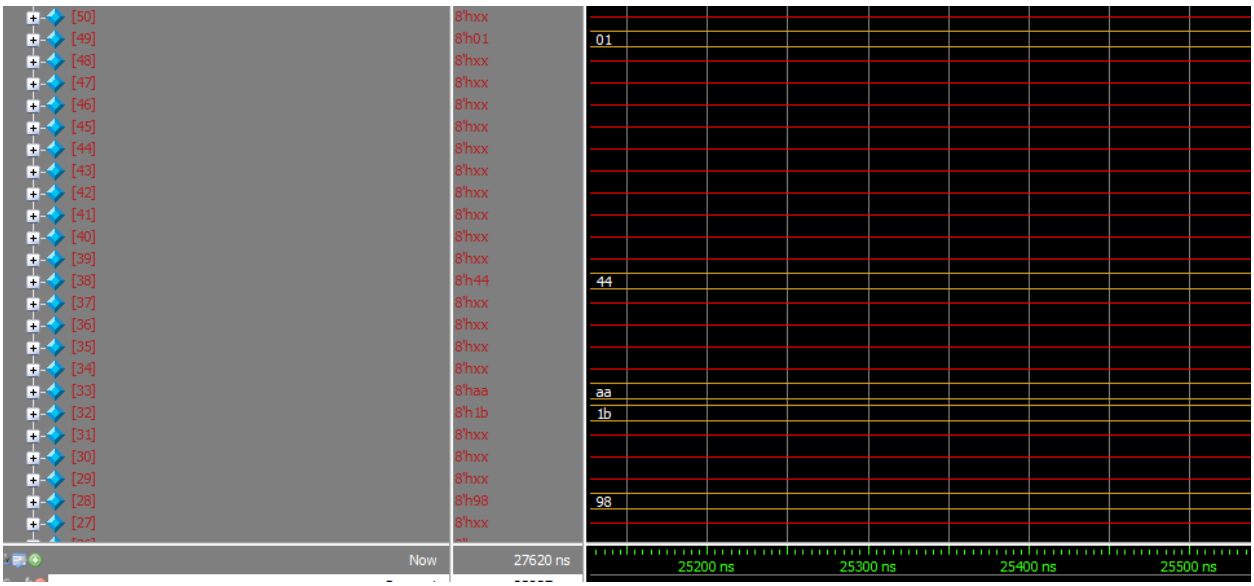


We experience the **same behavior** for MISO signal **in both Design and Scoreboard** in another case of directed test which has the value 0xc7 to be out on MISO signal serially

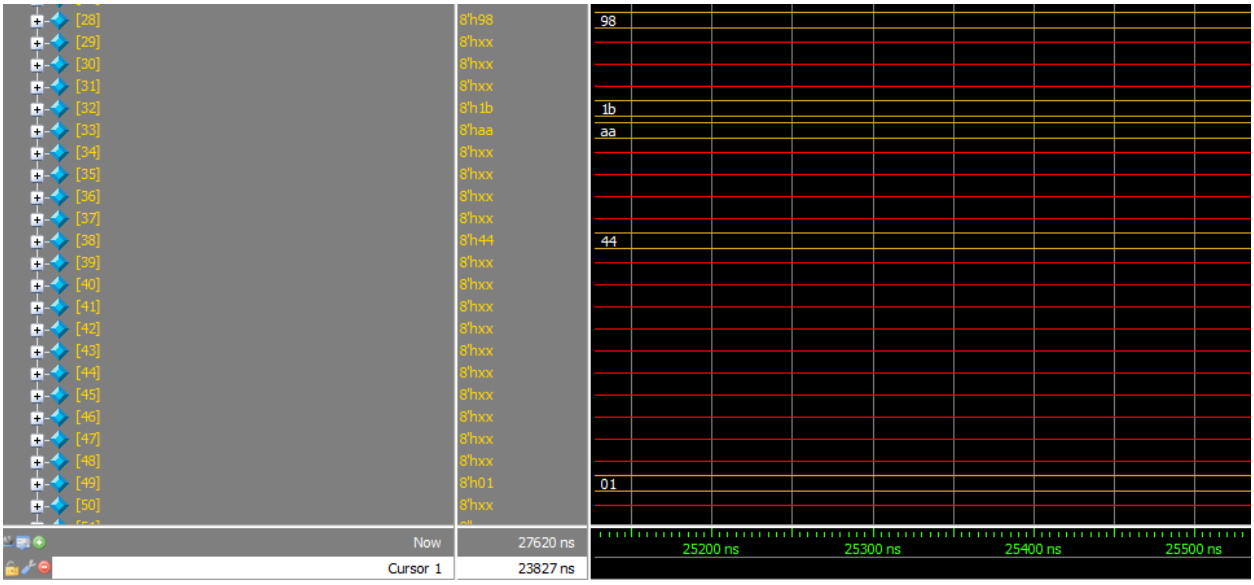
- **Memories Comparison:**

We will compare the three memories (Design memory & Wrapper scoreboard memory & RAM scoreboard memory) values at specific addresses to see that **they are matched**

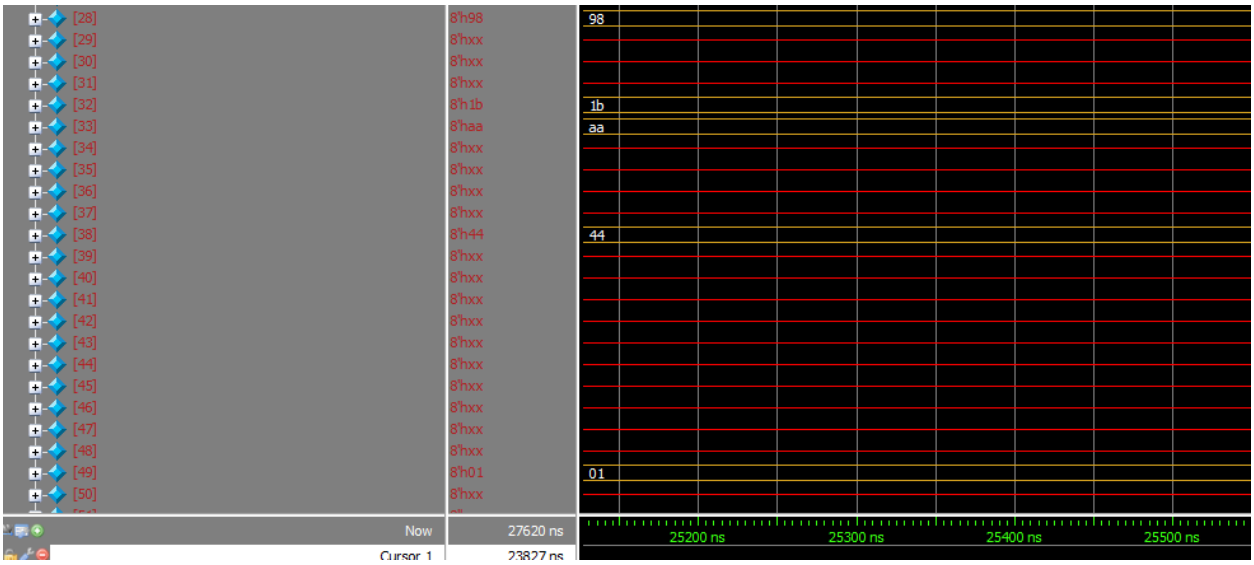
Design memory:



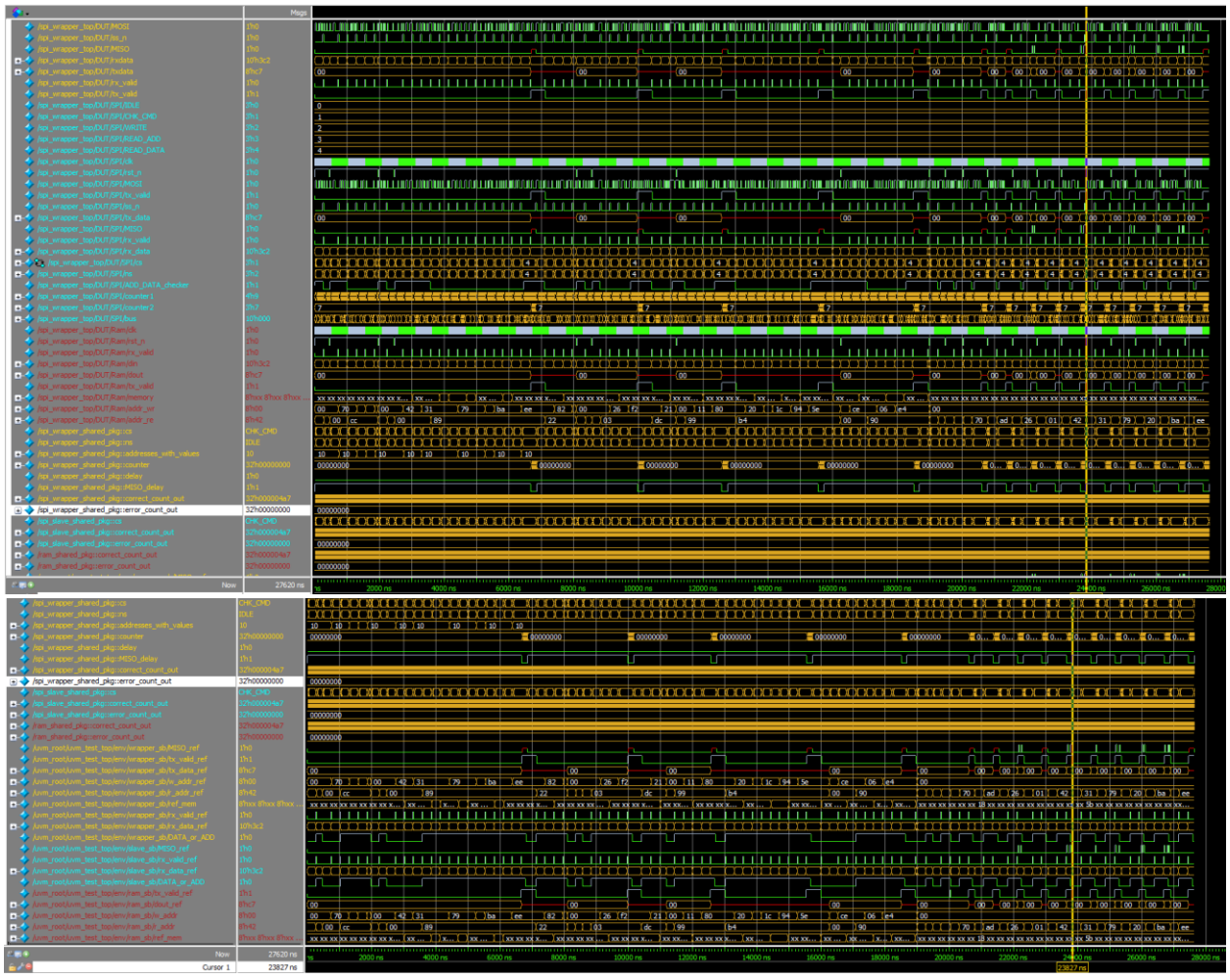
Wrapper scoreboard memory:



RAM scoreboard memory:



- **Full waveform:**

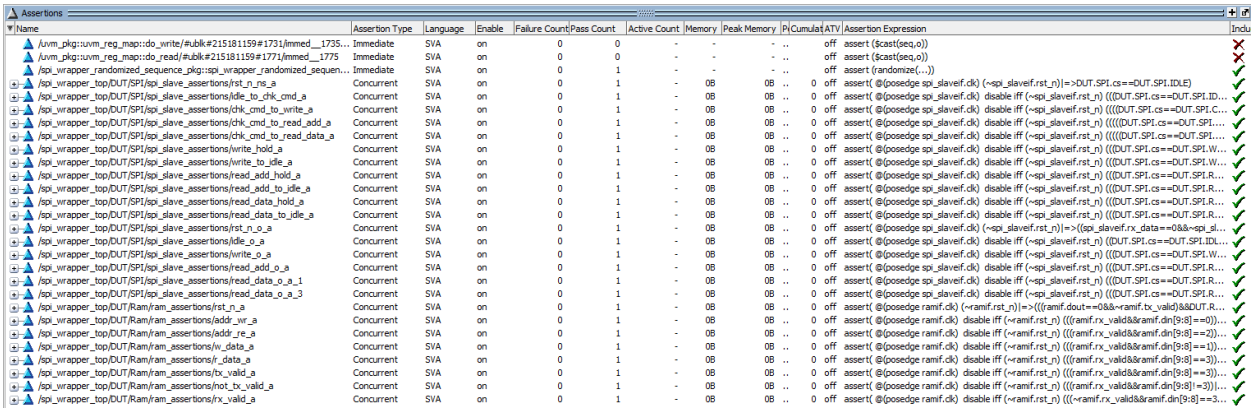


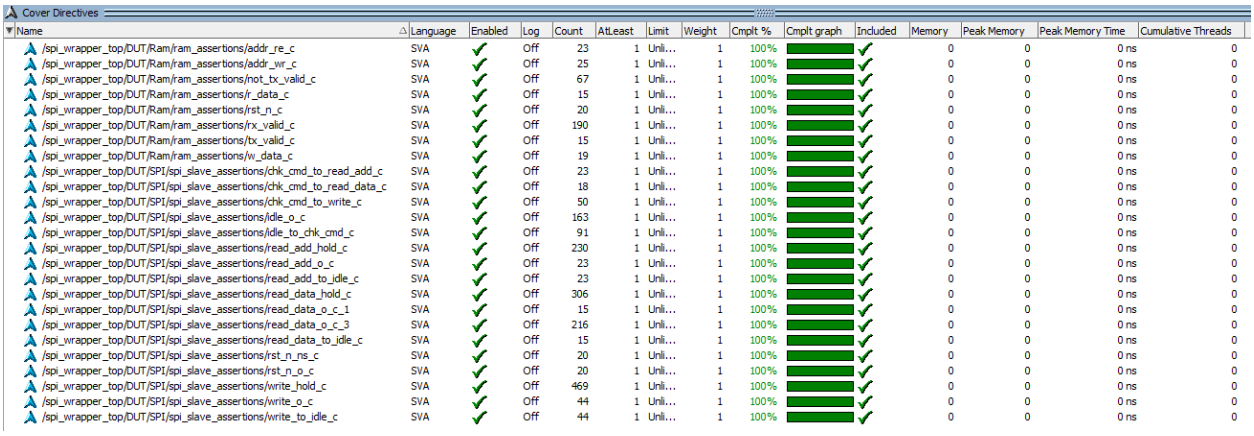
Full waveform proves **all error counters** were **never executed**, so we have **no output mismatch** between the original design and the reference models in all scoreboard files (Wrapper, Slave and RAM

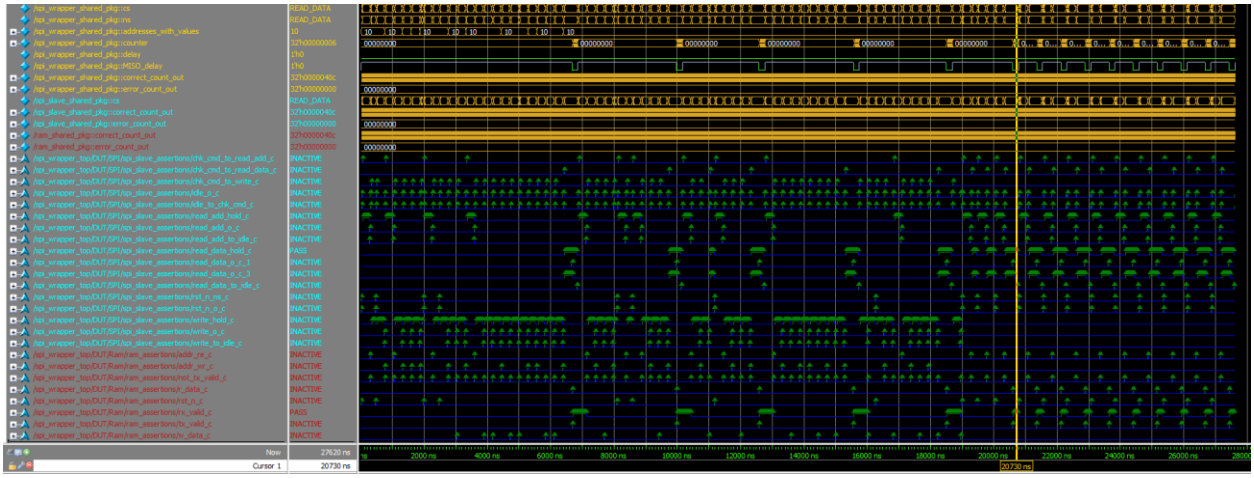
3. Cover groups:

Name	Coverage	Goal	% of Goal	Status	Included	Merge_instances
/spi_wrapper_coverage_collector_pkg/spi_wrapper_coverage_...	100.00%					
TYPE SPI_WRAPPER_Cross_Group	100.00%	100	100.00%	✓		auto(0)
INST \/spi_wrapper_coverage_collector_pkg::spi_wrap...	100.00%	100	100.00%	✓		
CVP cp_ss_n	100.00%	100	100.00%	✓		
bin state_start	82	1	100.00%	✓		
bin state_end	82	1	100.00%	✓		
CVP SPI_WRAPPER_Cross_Group::cp_ss_n	100.00%	100	100.00%	✓		
/spi_slave_coverage_collector_pkg/spi_slave_coverage_collect...	100.00%					
TYPE SPI_SLAVE_Cross_Group	100.00%	100	100.00%	✓		auto(1)
CVP SPI_SLAVE_Cross_Group::cp_tx_valid	100.00%	100	100.00%	✓		
bin auto[1]	429	1	100.00%	✓		
bin auto[0]	2333	1	100.00%	✓		
CVP SPI_SLAVE_Cross_Group::cp_tx_data	100.00%	100	100.00%	✓		
bin random_2nd_half	60	1	100.00%	✓		
bin random_1st_half	2702	1	100.00%	✓		
CVP SPI_SLAVE_Cross_Group::cp_ss_n	100.00%	100	100.00%	✓		
bin auto[1]	308	1	100.00%	✓		
bin auto[0]	2454	1	100.00%	✓		
CVP SPI_SLAVE_Cross_Group::cp_rx_valid	100.00%	100	100.00%	✓		
bin auto[1]	166	1	100.00%	✓		
bin auto[0]	2595	1	100.00%	✓		
CVP SPI_SLAVE_Cross_Group::cp_rx_data9	100.00%	100	100.00%	✓		
bin auto[1]	1071	1	100.00%	✓		
bin auto[0]	1690	1	100.00%	✓		
CVP SPI_SLAVE_Cross_Group::cp_rx_data8	100.00%	100	100.00%	✓		
bin auto[1]	951	1	100.00%	✓		
bin auto[0]	1810	1	100.00%	✓		
CROSS SPI_SLAVE_Cross_Group::wr_data_C	100.00%	100	100.00%	✓		
bin wr_data	38	1	100.00%	✓		
CROSS SPI_SLAVE_Cross_Group::wr_addr_C	100.00%	100	100.00%	✓		
bin wr_addr	50	1	100.00%	✓		
CROSS SPI_SLAVE_Cross_Group::receive_C	100.00%	100	100.00%	✓		
bin receive	429	1	100.00%	✓		
CROSS SPI_SLAVE_Cross_Group::rd_data_C	100.00%	100	100.00%	✓		
bin rd_data	32	1	100.00%	✓		
CROSS SPI_SLAVE_Cross_Group::rd_addr_C	100.00%	100	100.00%	✓		
bin rd_addr	46	1	100.00%	✓		
/ram_coverage_collector_pkg/ram_coverage_collector	100.00%					
TYPE RAM_Cross_Group	100.00%	100	100.00%	✓		auto(1)
CVP RAM_Cross_Group::cp_tx_valid	100.00%	100	100.00%	✓		
bin auto[1]	215	1	100.00%	✓		
bin auto[0]	1166	1	100.00%	✓		
CVP RAM_Cross_Group::cp_rx_valid	100.00%	100	100.00%	✓		
bin auto[1]	83	1	100.00%	✓		
bin auto[0]	1298	1	100.00%	✓		
CVP RAM_Cross_Group::cp_din9	100.00%	100	100.00%	✓		
bin auto[1]	536	1	100.00%	✓		
bin auto[0]	845	1	100.00%	✓		
CVP RAM_Cross_Group::cp_din8	100.00%	100	100.00%	✓		
bin auto[1]	476	1	100.00%	✓		
bin auto[0]	905	1	100.00%	✓		
CROSS RAM_Cross_Group::wr_data_C	100.00%	100	100.00%	✓		
bin wr_data	19	1	100.00%	✓		
CROSS RAM_Cross_Group::wr_addr_C	100.00%	100	100.00%	✓		
bin wr_addr	25	1	100.00%	✓		
CROSS RAM_Cross_Group::rd_data_C	100.00%	100	100.00%	✓		
bin rd_data	210	1	100.00%	✓		
CROSS RAM_Cross_Group::rd_addr_C	100.00%	100	100.00%	✓		
bin rd_addr	23	1	100.00%	✓		

4. Assertions:



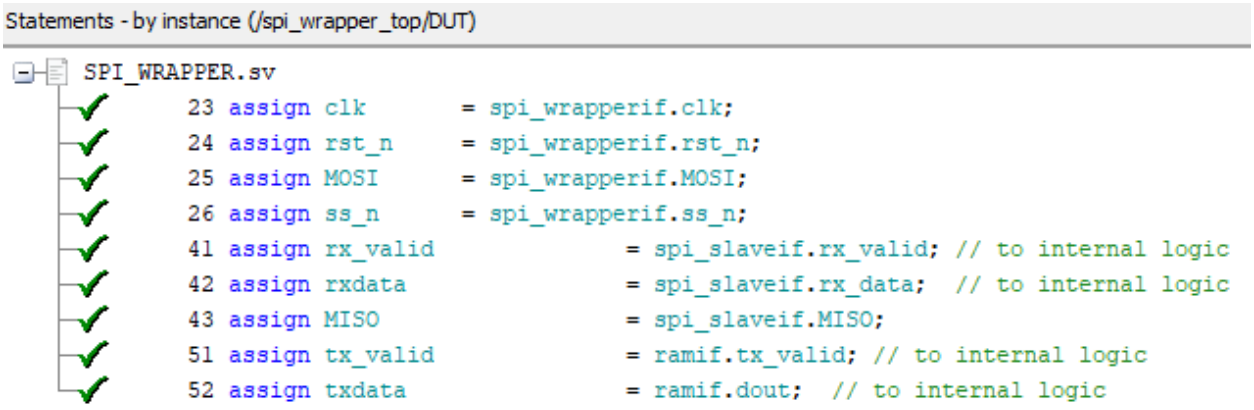




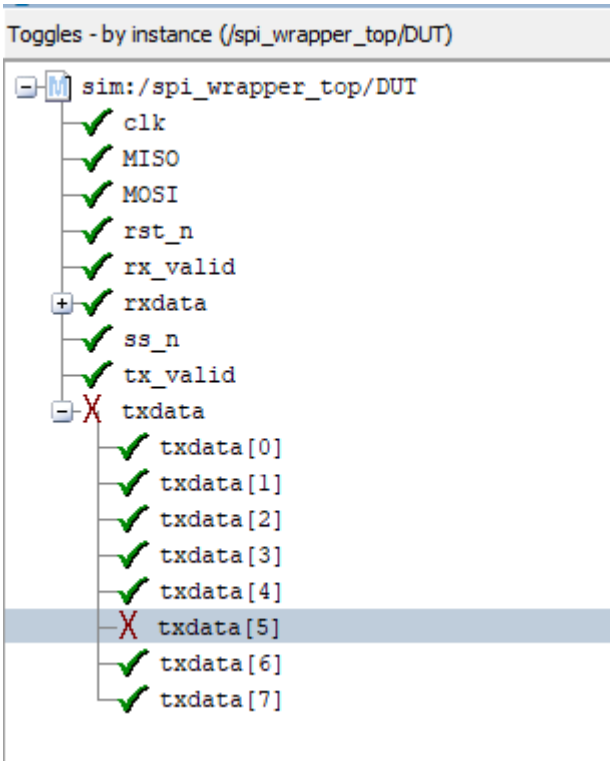
All assertions are **passed**!

5. Code Coverage:

- Statement:



- Toggle



Tx_data is randomly generated so if we increased the tests it is expected to get 100% toggle coverage

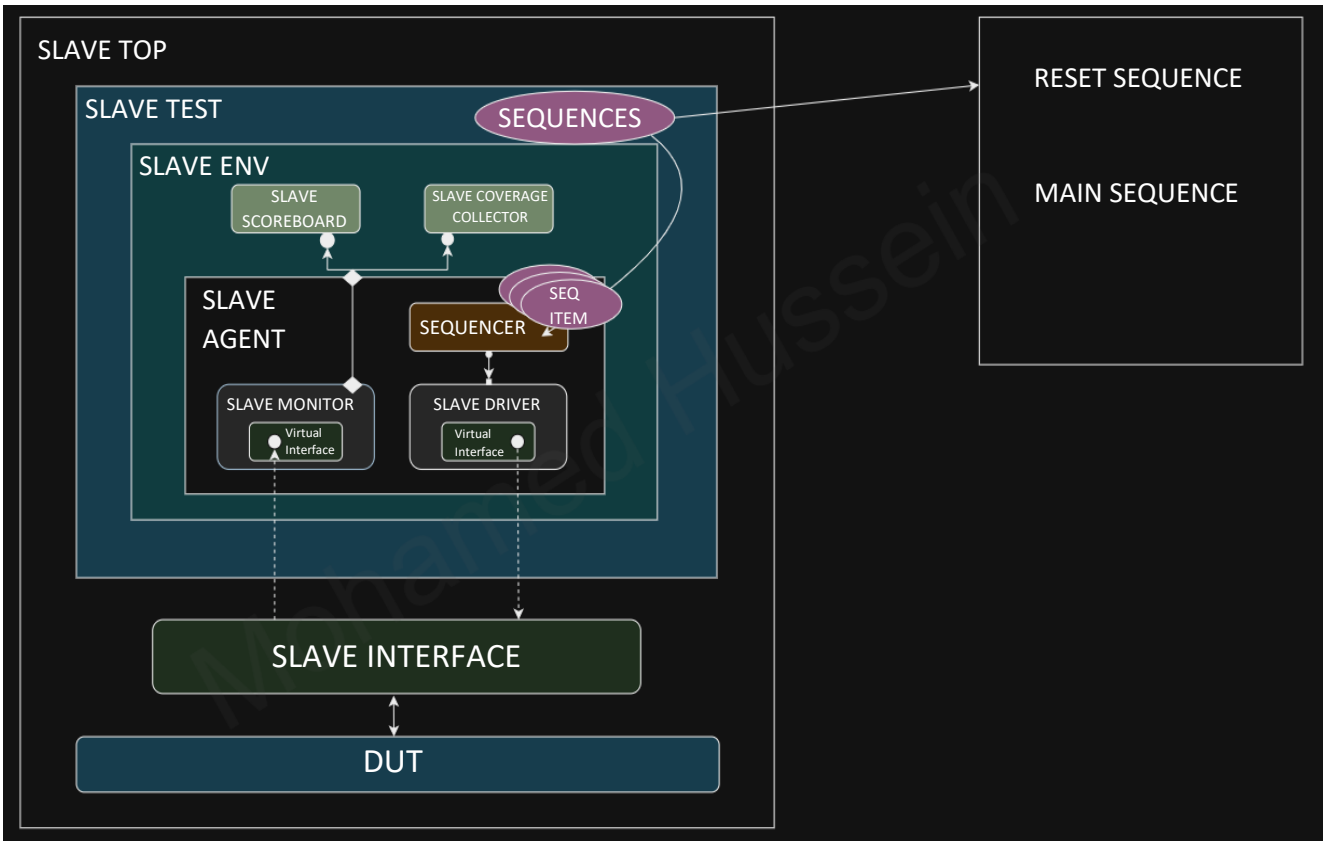
6. UVM Report:

```
# UVM_INFO SPI_WRAPPER_TEST.sv(81) @ 20: uvm_test_top [run_phase] reset dasserted
# UVM_INFO SPI_WRAPPER_TEST.sv(84) @ 20: uvm_test_top [run_phase] randomized asserted
# UVM_INFO SPI_WRAPPER_TEST.sv(86) @ 20020: uvm_test_top [run_phase] randomized dasserted
# UVM_INFO SPI_WRAPPER_TEST.sv(89) @ 20020: uvm_test_top [run_phase] reset asserted
# UVM_INFO SPI_WRAPPER_TEST.sv(91) @ 20040: uvm_test_top [run_phase] reset dasserted
# UVM_INFO SPI_WRAPPER_TEST.sv(94) @ 20040: uvm_test_top [run_phase] read_address asserted
# UVM_INFO SPI_WRAPPER_TEST.sv(96) @ 20340: uvm_test_top [run_phase] read_address dasserted
# UVM_INFO SPI_WRAPPER_TEST.sv(99) @ 20340: uvm_test_top [run_phase] read_data asserted
# UVM_INFO SPI_WRAPPER_TEST.sv(101) @ 20780: uvm_test_top [run_phase] read_data dasserted
# UVM_INFO SPI_WRAPPER_TEST.sv(89) @ 20780: uvm_test_top [run_phase] reset asserted
# UVM_INFO SPI_WRAPPER_TEST.sv(91) @ 20800: uvm_test_top [run_phase] reset dasserted
# UVM_INFO SPI_WRAPPER_TEST.sv(94) @ 20800: uvm_test_top [run_phase] read_address asserted
# UVM_INFO SPI_WRAPPER_TEST.sv(96) @ 21100: uvm_test_top [run_phase] read_address dasserted
# UVM_INFO SPI_WRAPPER_TEST.sv(99) @ 21100: uvm_test_top [run_phase] read_data asserted
# UVM_INFO SPI_WRAPPER_TEST.sv(101) @ 21540: uvm_test_top [run_phase] read_data dasserted
# UVM_INFO SPI_WRAPPER_TEST.sv(89) @ 21540: uvm_test_top [run_phase] reset asserted
# UVM_INFO SPI_WRAPPER_TEST.sv(91) @ 21560: uvm_test_top [run_phase] reset dasserted
# UVM_INFO SPI_WRAPPER_TEST.sv(94) @ 21560: uvm_test_top [run_phase] read_address asserted
# UVM_INFO SPI_WRAPPER_TEST.sv(96) @ 21860: uvm_test_top [run_phase] read_address dasserted
# UVM_INFO SPI_WRAPPER_TEST.sv(99) @ 21860: uvm_test_top [run_phase] read_data asserted
# UVM_INFO SPI_WRAPPER_TEST.sv(101) @ 22300: uvm_test_top [run_phase] read_data dasserted
# UVM_INFO SPI_WRAPPER_TEST.sv(89) @ 22300: uvm_test_top [run_phase] reset asserted
# UVM_INFO SPI_WRAPPER_TEST.sv(91) @ 22320: uvm_test_top [run_phase] reset dasserted
# UVM_INFO SPI_WRAPPER_TEST.sv(94) @ 22320: uvm_test_top [run_phase] read_address asserted
# UVM_INFO SPI_WRAPPER_TEST.sv(96) @ 22620: uvm_test_top [run_phase] read_address dasserted
# UVM_INFO SPI_WRAPPER_TEST.sv(99) @ 22620: uvm_test_top [run_phase] read_data asserted
# UVM_INFO SPI_WRAPPER_TEST.sv(101) @ 23060: uvm_test_top [run_phase] read_data dasserted
# UVM_INFO SPI_WRAPPER_TEST.sv(89) @ 23060: uvm_test_top [run_phase] reset asserted
# UVM_INFO SPI_WRAPPER_TEST.sv(91) @ 23080: uvm_test_top [run_phase] reset dasserted
# UVM_INFO SPI_WRAPPER_TEST.sv(94) @ 23080: uvm_test_top [run_phase] read_address asserted
# UVM_INFO SPI_WRAPPER_TEST.sv(96) @ 23380: uvm_test_top [run_phase] read_address dasserted
# UVM_INFO SPI_WRAPPER_TEST.sv(99) @ 23380: uvm_test_top [run_phase] read_data asserted
# UVM_INFO SPI_WRAPPER_TEST.sv(101) @ 23820: uvm_test_top [run_phase] read_data dasserted
# UVM_INFO SPI_WRAPPER_TEST.sv(89) @ 23820: uvm_test_top [run_phase] reset asserted
# UVM_INFO SPI_WRAPPER_TEST.sv(91) @ 23840: uvm_test_top [run_phase] reset dasserted
# UVM_INFO SPI_WRAPPER_TEST.sv(94) @ 23840: uvm_test_top [run_phase] read_address asserted
# UVM_INFO SPI_WRAPPER_TEST.sv(96) @ 24140: uvm_test_top [run_phase] read_address dasserted
# UVM_INFO SPI_WRAPPER_TEST.sv(99) @ 24140: uvm_test_top [run_phase] read_data asserted
# UVM_INFO SPI_WRAPPER_TEST.sv(101) @ 24580: uvm_test_top [run_phase] read_data dasserted
# UVM_INFO SPI_WRAPPER_TEST.sv(89) @ 24580: uvm_test_top [run_phase] reset asserted
# UVM_INFO SPI_WRAPPER_TEST.sv(91) @ 24600: uvm_test_top [run_phase] reset dasserted
# UVM_INFO SPI_WRAPPER_TEST.sv(94) @ 24600: uvm_test_top [run_phase] read_address asserted
# UVM_INFO SPI_WRAPPER_TEST.sv(96) @ 24900: uvm_test_top [run_phase] read_address dasserted
# UVM_INFO SPI_WRAPPER_TEST.sv(99) @ 24900: uvm_test_top [run_phase] read_data asserted
# UVM_INFO SPI_WRAPPER_TEST.sv(101) @ 25340: uvm_test_top [run_phase] read_data dasserted
# UVM_INFO SPI_WRAPPER_TEST.sv(89) @ 25340: uvm_test_top [run_phase] reset asserted
# UVM_INFO SPI_WRAPPER_TEST.sv(91) @ 25360: uvm_test_top [run_phase] reset dasserted
# UVM_INFO SPI_WRAPPER_TEST.sv(94) @ 25360: uvm_test_top [run_phase] read_address asserted
# UVM_INFO SPI_WRAPPER_TEST.sv(96) @ 25660: uvm_test_top [run_phase] read_address dasserted
# UVM_INFO SPI_WRAPPER_TEST.sv(99) @ 25660: uvm_test_top [run_phase] read_data asserted
# UVM_INFO SPI_WRAPPER_TEST.sv(101) @ 26100: uvm_test_top [run_phase] read_data dasserted
# UVM_INFO SPI_WRAPPER_TEST.sv(89) @ 26100: uvm_test_top [run_phase] reset asserted
# UVM_INFO SPI_WRAPPER_TEST.sv(91) @ 26120: uvm_test_top [run_phase] reset dasserted
# UVM_INFO SPI_WRAPPER_TEST.sv(94) @ 26120: uvm_test_top [run_phase] read_address asserted
# UVM_INFO SPI_WRAPPER_TEST.sv(96) @ 26420: uvm_test_top [run_phase] read_address dasserted
# UVM_INFO SPI_WRAPPER_TEST.sv(99) @ 26420: uvm_test_top [run_phase] read_data asserted
# UVM_INFO SPI_WRAPPER_TEST.sv(101) @ 26860: uvm_test_top [run_phase] read_data dasserted
# UVM_INFO SPI_WRAPPER_TEST.sv(89) @ 26860: uvm_test_top [run_phase] reset asserted
# UVM_INFO SPI_WRAPPER_TEST.sv(91) @ 26880: uvm_test_top [run_phase] reset dasserted
# UVM_INFO SPI_WRAPPER_TEST.sv(94) @ 26880: uvm_test_top [run_phase] read_address asserted
# UVM_INFO SPI_WRAPPER_TEST.sv(96) @ 27180: uvm_test_top [run_phase] read_address dasserted
# UVM_INFO SPI_WRAPPER_TEST.sv(99) @ 27180: uvm_test_top [run_phase] read_data asserted
# UVM_INFO SPI_WRAPPER_TEST.sv(101) @ 27620: uvm_test_top [run_phase] read_data dasserted
```

```
# UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1267) @ 27620: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
# UVM_INFO RAM_SCOREBOARD.sv(104) @ 27620: uvm_test_top.env.ram_sb [report_phase] total successful transactions in RAM: 1381
# UVM_INFO RAM_SCOREBOARD.sv(105) @ 27620: uvm_test_top.env.ram_sb [report_phase] total failed transactions in RAM: 0
#
#
# UVM_INFO SPI_SLAVE_SCOREBOARD.sv(177) @ 27620: uvm_test_top.env.slave_sb [report_phase] total successful transactions in SPI Slave: 1381
# UVM_INFO SPI_SLAVE_SCOREBOARD.sv(178) @ 27620: uvm_test_top.env.slave_sb [report_phase] total failed transactionsin in SPI Slave: 0
#
#
# UVM_INFO SPI_WRAPPER_SCOREBOARD.sv(216) @ 27620: uvm_test_top.env.wrapper_sb [report_phase] total successful transactions in Wrapper: 1381
# UVM_INFO SPI_WRAPPER_SCOREBOARD.sv(217) @ 27620: uvm_test_top.env.wrapper_sb [report_phase] total failed transactions in Wrapper: 0
#
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :   74
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [Questa UVM]     2
# [RNTST]     1
# [TEST_DONE]     1
# [report_phase]     6
# [run_phase]    64
# ** Note: $finish    : C:/Users/DELL/AppData/Local/Programs/win64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
#    Time: 27620 ns  Iteration: 60  Instance: /spi_wrapper_top
# Break in Task uvm_pkg/uvm_root::run_test at C:/Users/DELL/AppData/Local/Programs/win64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh line 430
```

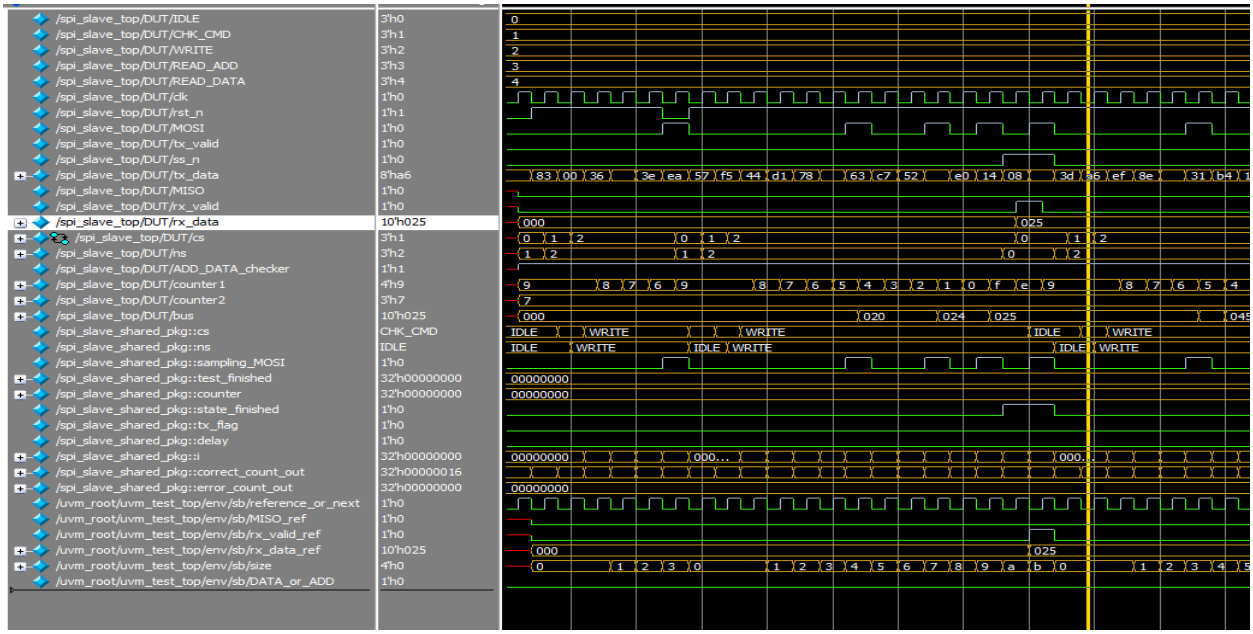
SPI SLAVE:

1. UVM Structure

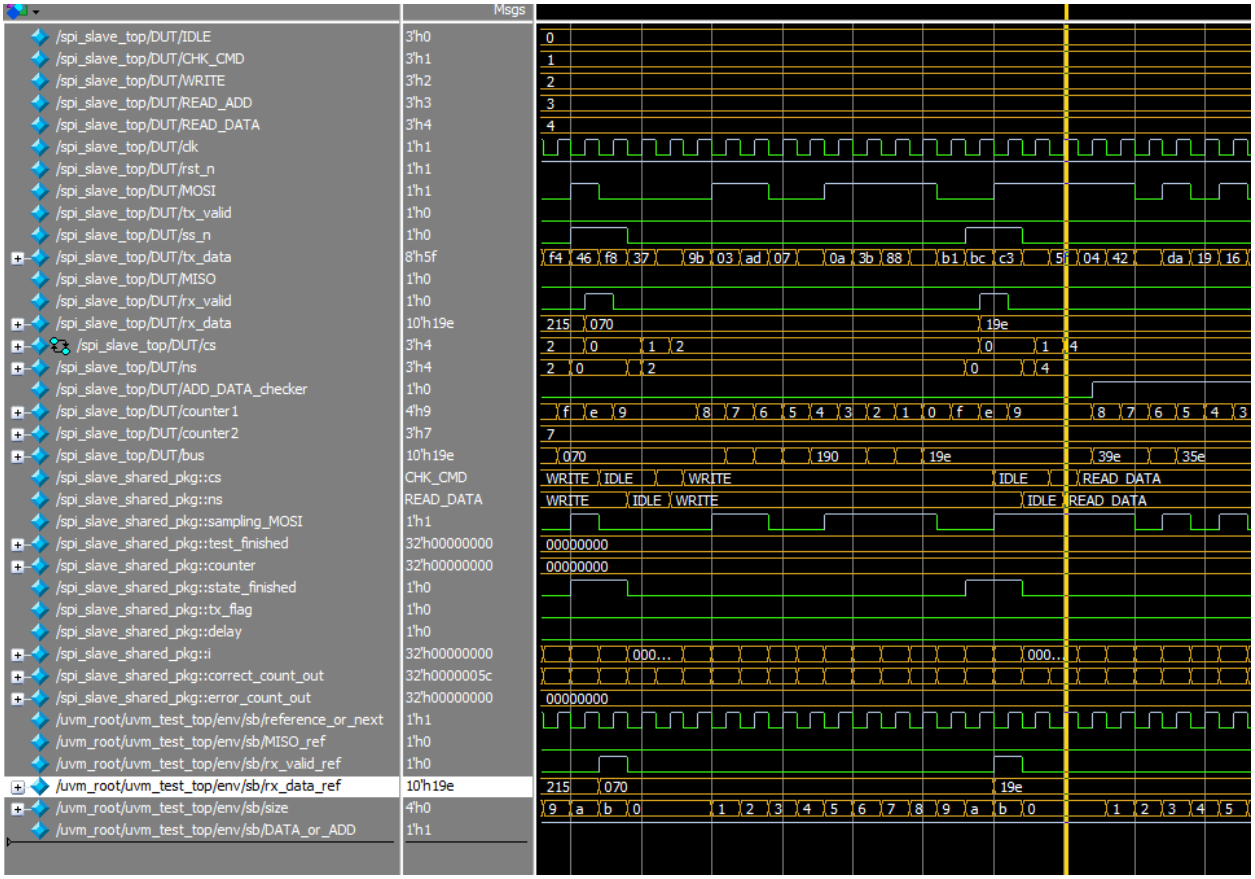


2. Verifying Functionality

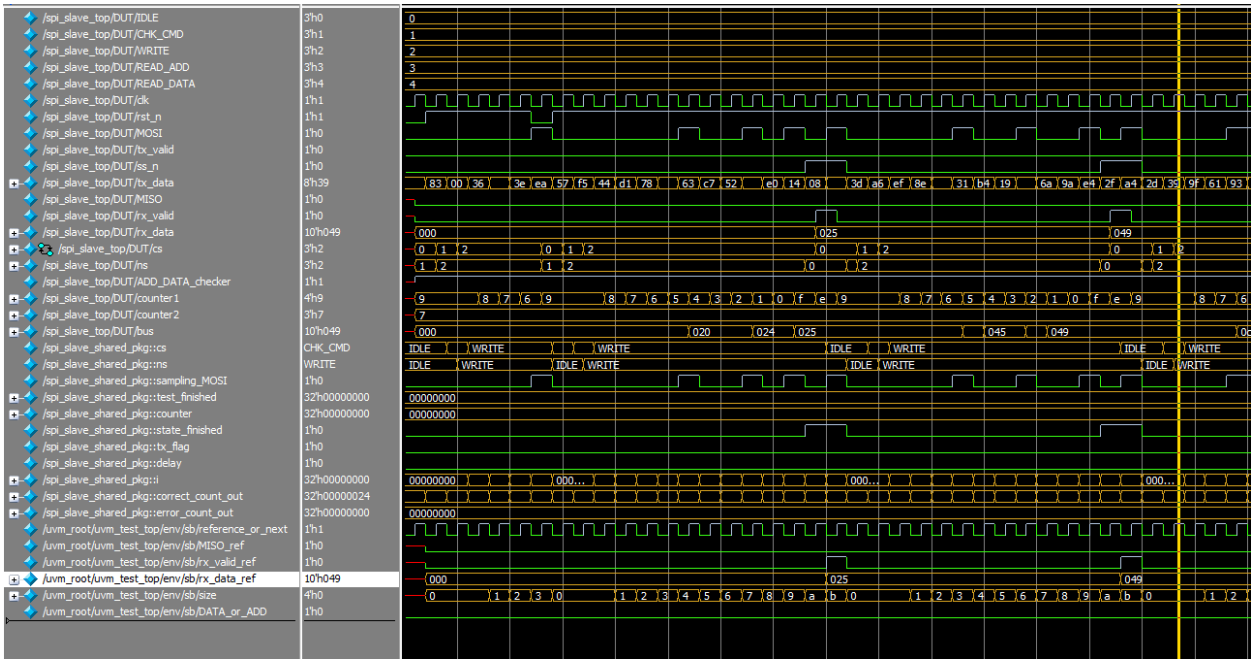
- **Write state:**



Rx_data is the same as rx_data_ref and also rx_valid is the same as rx_valid_ref, and the states are the same, notice that the Most Significant two Bits of **rx_data[9:8] = 2'b00** as the state is write **address** (from Specs)

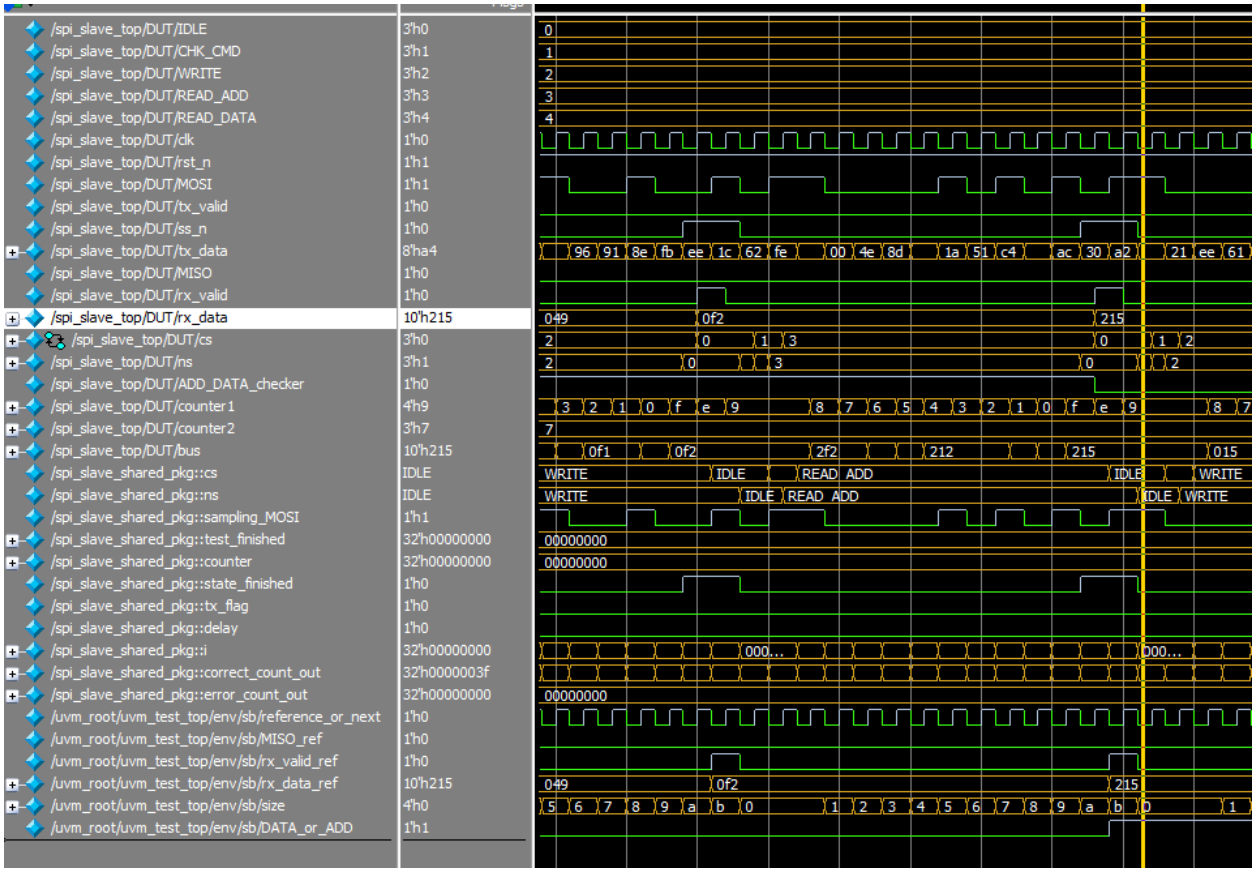


Rx_data is the same as rx_data_ref and also rx_valid is the same as rx_valid_ref, and the states are the same, notice that the Most Significant two Bits of **rx_data [9:8] = 2'b01** as the state is write **Data** (from Specs)

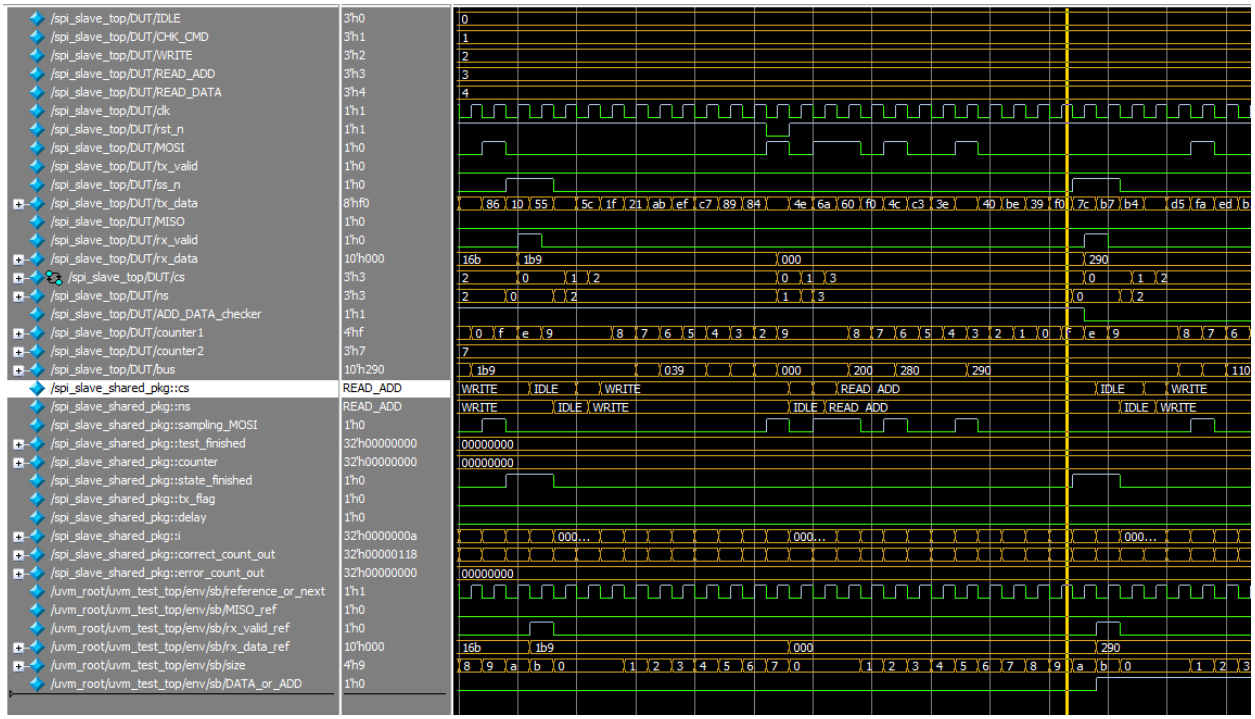


Another proof

- **READ_ADD state:**

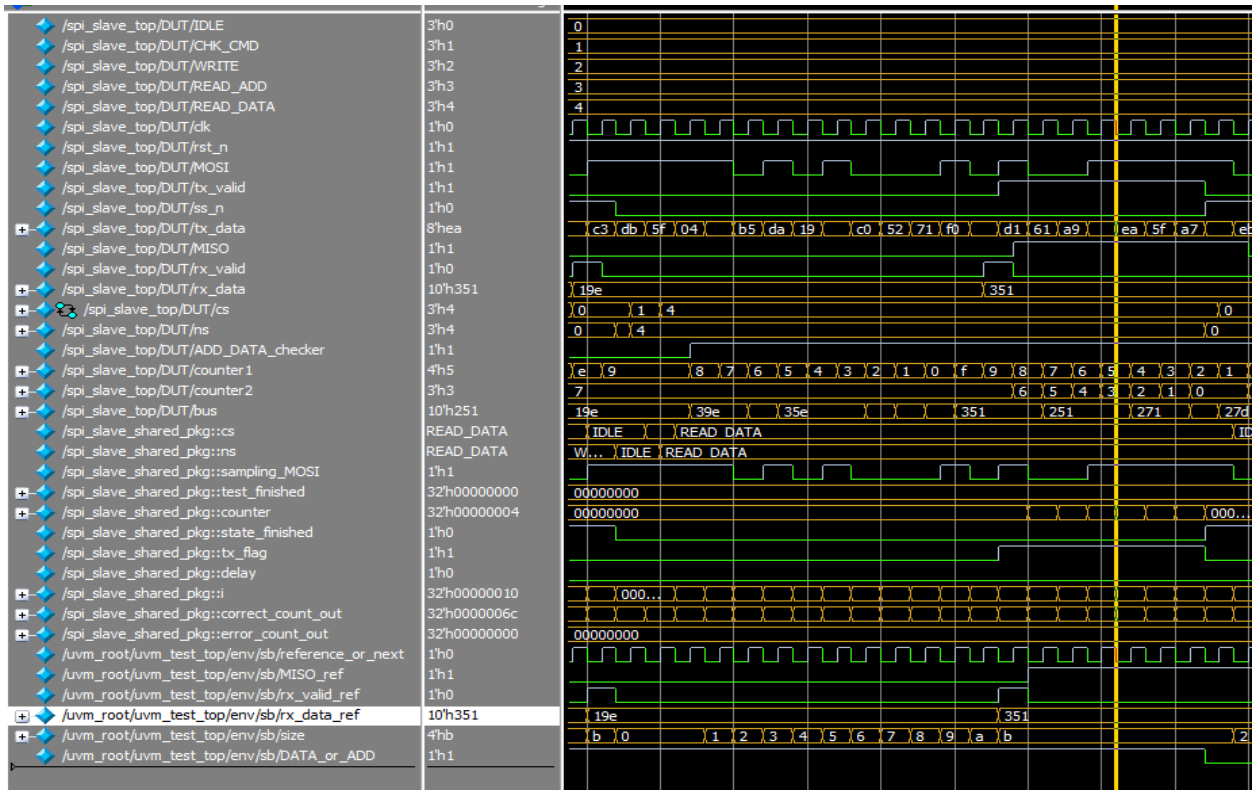


You will notice that rx_data and rx_data_ref have the same data (0x261), Most Significant two Bits of **rx_data [9:8] = 2'b10** as the state is read **address** (from Specs)

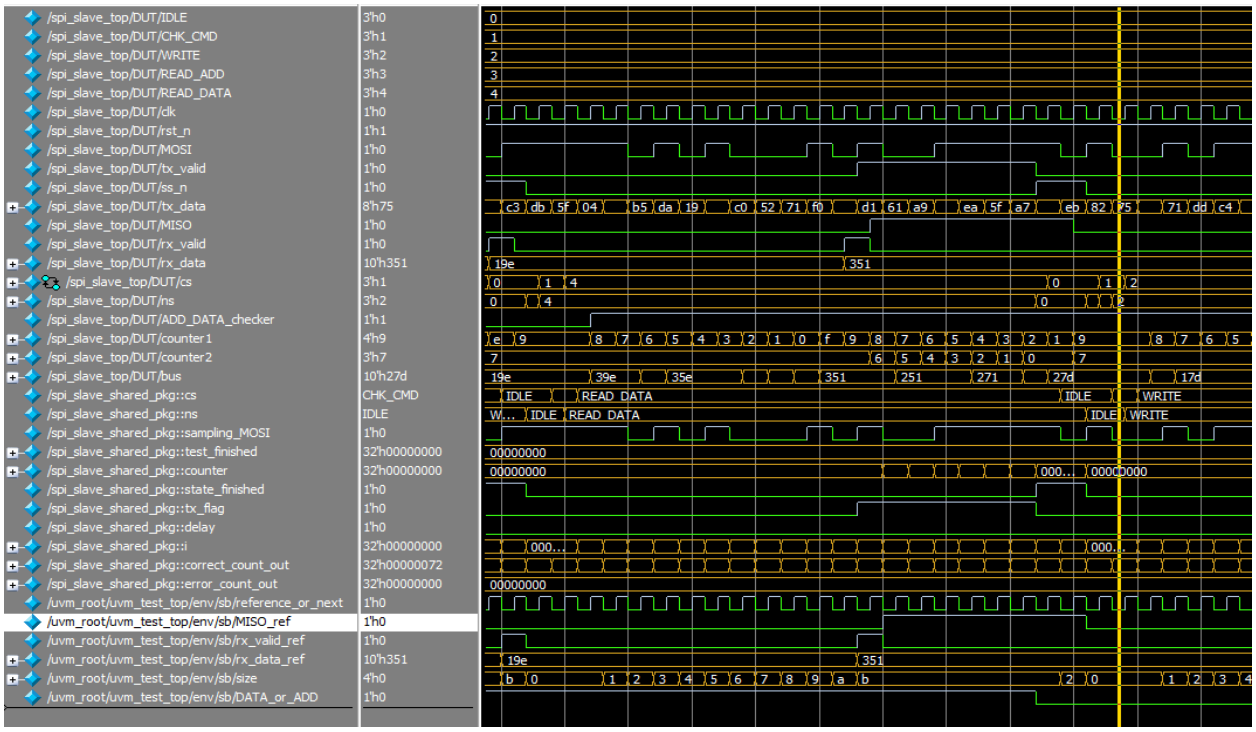


Another proof

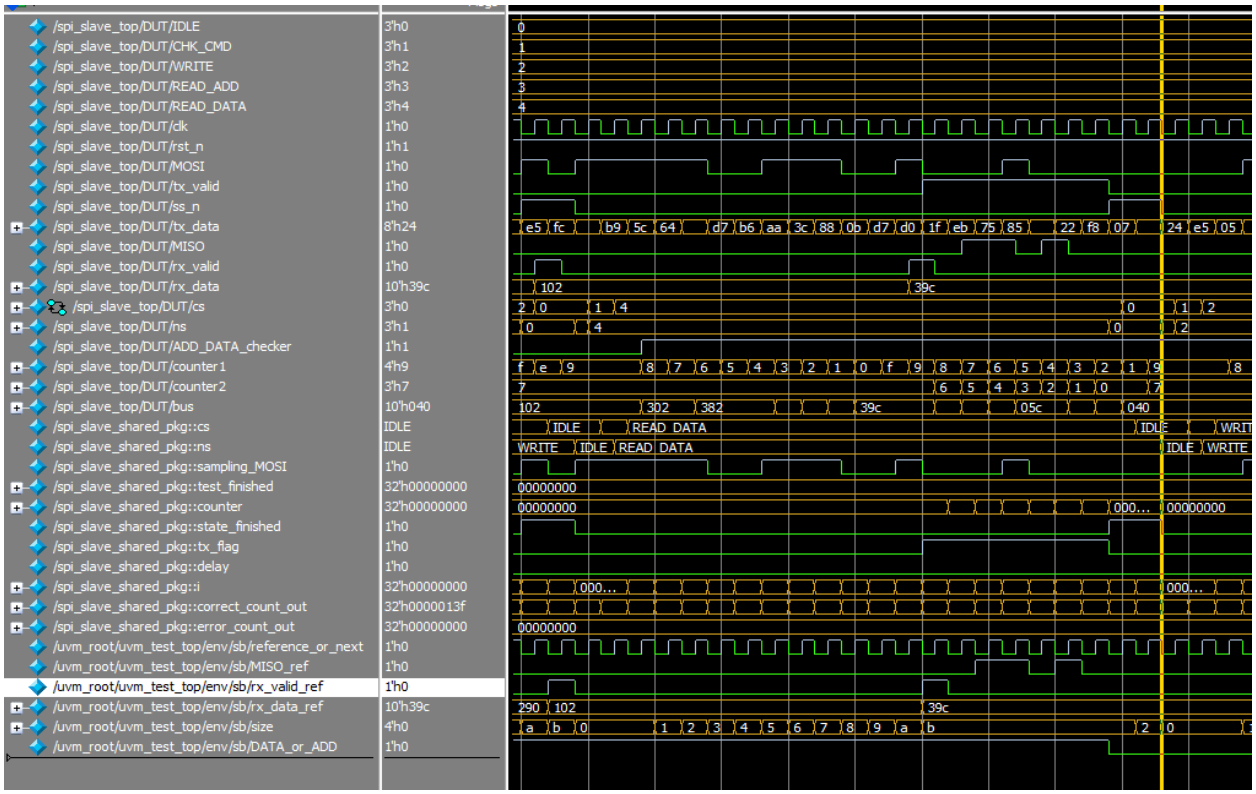
- **READ_DATA state:**



In read data state, rx_data is sent correctly but we care about the MISO signal in this state, Most Significant two Bits of **rx_data [9:8] = 2'b11** as the state is read **Data** (from Specs)

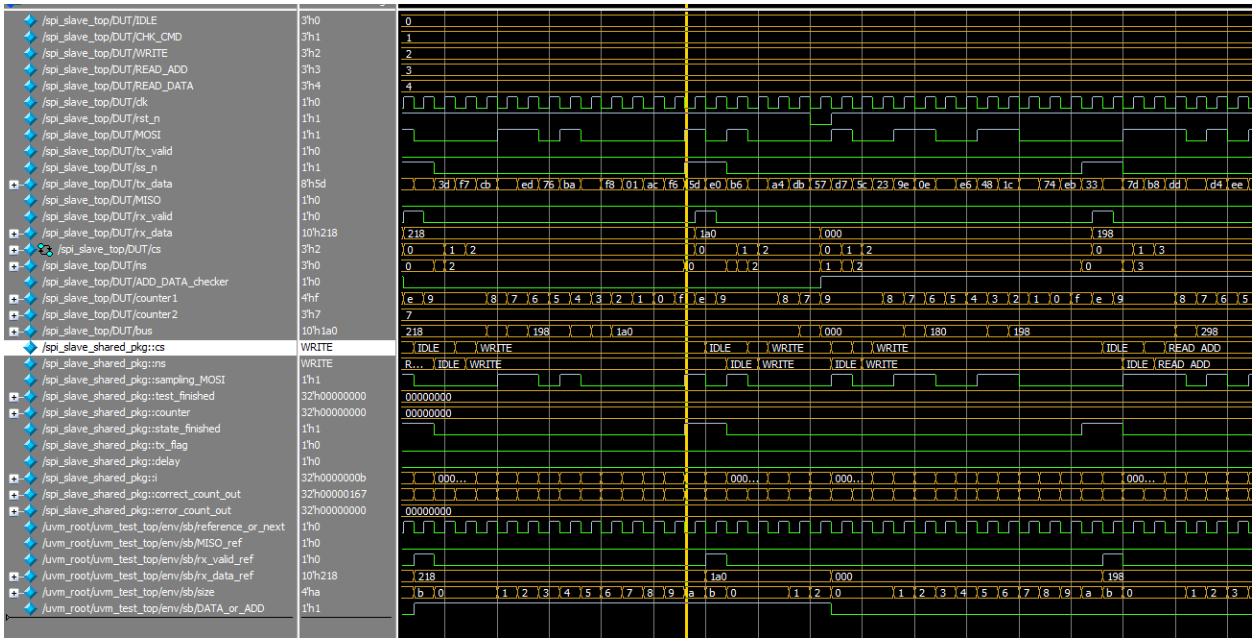


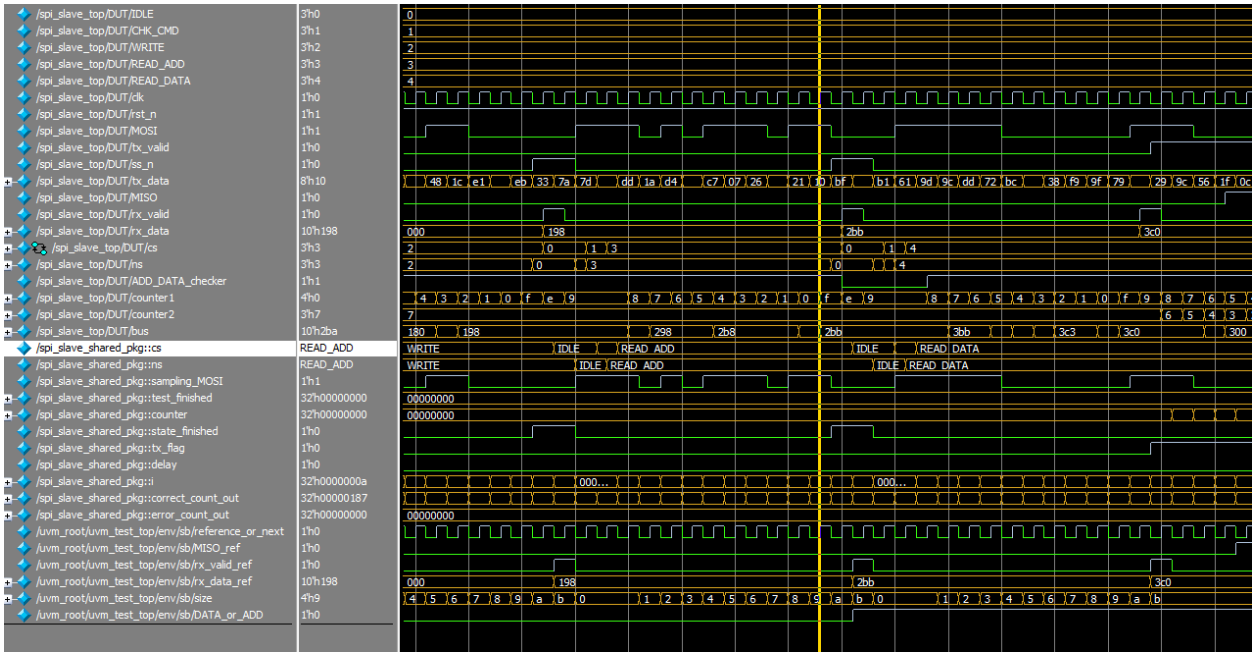
After rx_data is sent, we will notice the same behavior for MISO signal and MISO_ref signal which indicates the correctness of the design

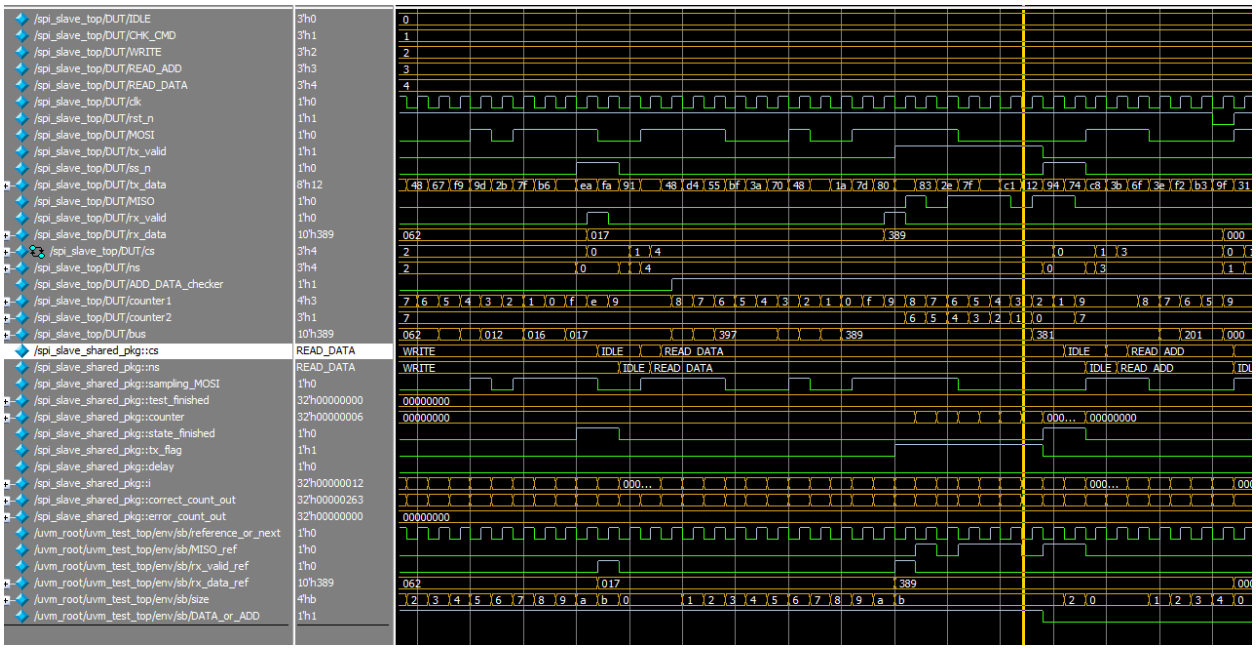


Another proof

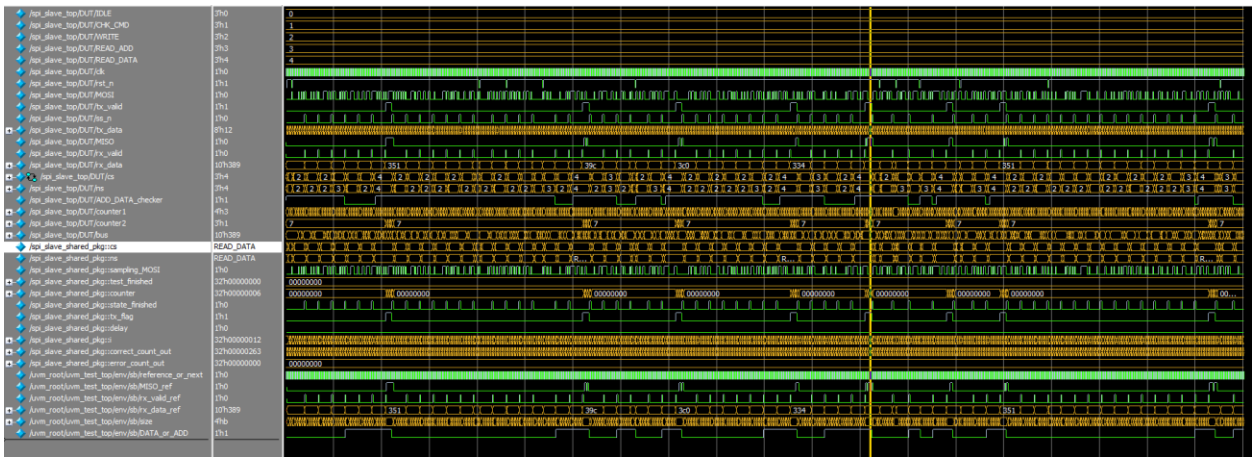
- **Random snapshots:**





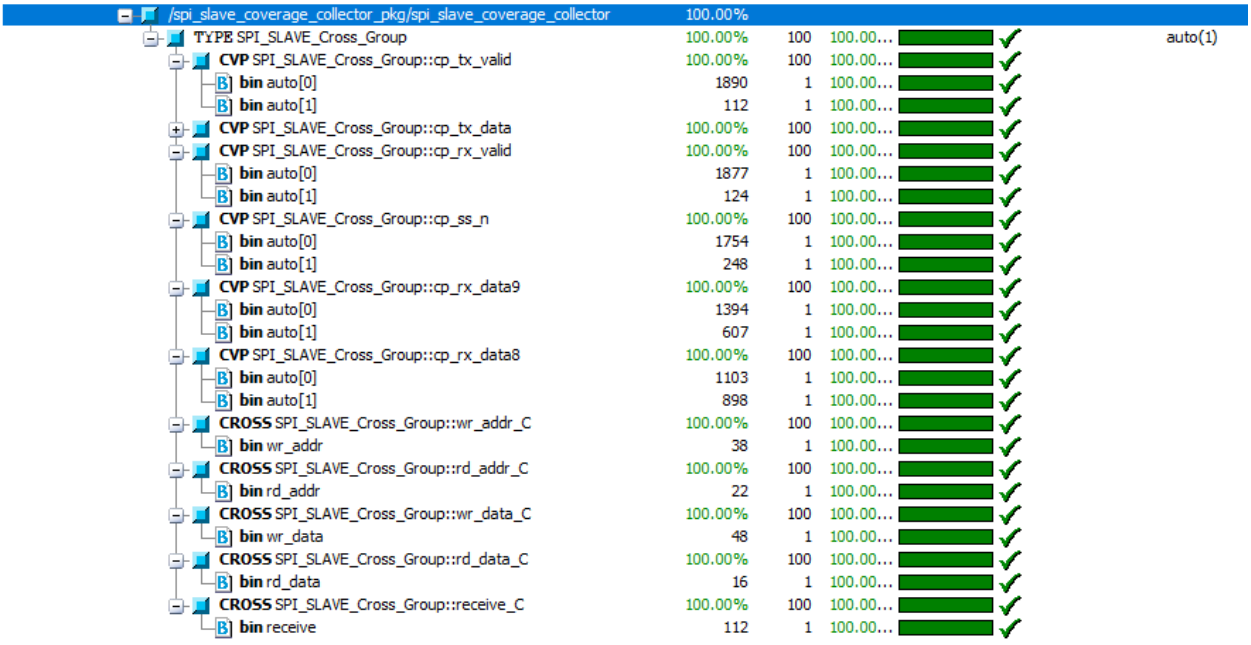


- **Full Waveform:**



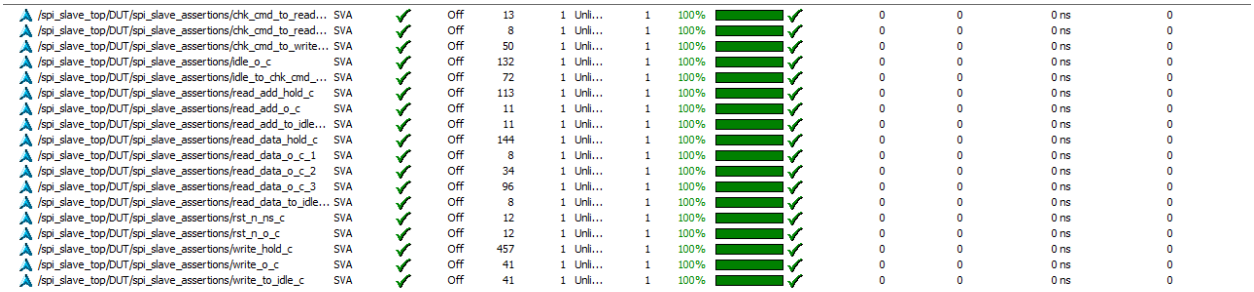
Full waveform proves error counter was **never executed**, so we have **no output mismatch** between the original design and the reference model in scoreboard

3. Cover groups:

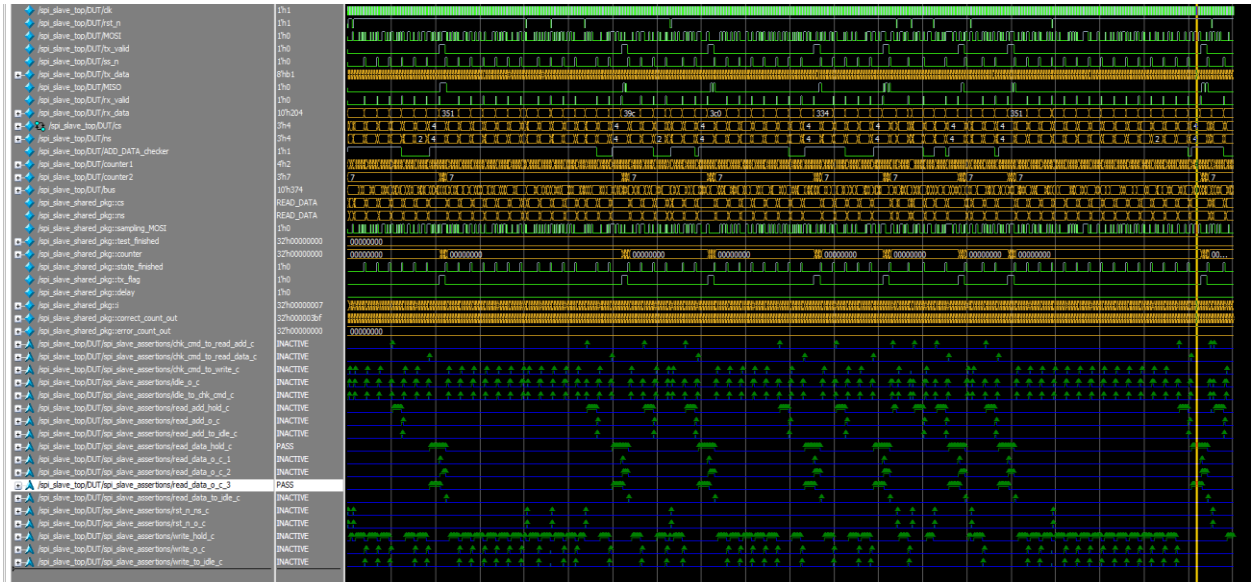


/spi_slave_coverage_collector_pkg/spi_slave_coverage_collector	100.00%			
TYPE SPI_SLAVE_Cross_Group	100.00%	100	100.00...	auto(1)
CVP SPI_SLAVE_Cross_Group::cp_tx_valid	100.00%	100	100.00...	
bin auto[0]	1890	1	100.00...	
bin auto[1]	112	1	100.00...	
CVP SPI_SLAVE_Cross_Group::cp_tx_data	100.00%	100	100.00...	
CVP SPI_SLAVE_Cross_Group::cp_rx_valid	100.00%	100	100.00...	
bin auto[0]	1877	1	100.00...	
bin auto[1]	124	1	100.00...	
CVP SPI_SLAVE_Cross_Group::cp_ss_n	100.00%	100	100.00...	
bin auto[0]	1754	1	100.00...	
bin auto[1]	248	1	100.00...	
CVP SPI_SLAVE_Cross_Group::cp_rx_data9	100.00%	100	100.00...	
bin auto[0]	1394	1	100.00...	
bin auto[1]	607	1	100.00...	
CVP SPI_SLAVE_Cross_Group::cp_rx_data8	100.00%	100	100.00...	
bin auto[0]	1103	1	100.00...	
bin auto[1]	898	1	100.00...	
CROSS SPI_SLAVE_Cross_Group::wr_addr_C	100.00%	100	100.00...	
bin wr_addr	38	1	100.00...	
CROSS SPI_SLAVE_Cross_Group::rd_addr_C	100.00%	100	100.00...	
bin rd_addr	22	1	100.00...	
CROSS SPI_SLAVE_Cross_Group::wr_data_C	100.00%	100	100.00...	
bin wr_data	48	1	100.00...	
CROSS SPI_SLAVE_Cross_Group::rd_data_C	100.00%	100	100.00...	
bin rd_data	16	1	100.00...	
CROSS SPI_SLAVE_Cross_Group::receive_C	100.00%	100	100.00...	
bin receive	112	1	100.00...	

4. Assertions:



/spi_slave_top/DUT/spi_slave_assertions/chk_cmd_to_read...	SVA	✓	Off	13	1	Unli...	1	100%	✓	0	0	0 ns	0
/spi_slave_top/DUT/spi_slave_assertions/chk_cmd_to_read...	SVA	✓	Off	8	1	Unli...	1	100%	✓	0	0	0 ns	0
/spi_slave_top/DUT/spi_slave_assertions/chk_cmd_to_write...	SVA	✓	Off	50	1	Unli...	1	100%	✓	0	0	0 ns	0
/spi_slave_top/DUT/spi_slave_assertions/idle_o_c	SVA	✓	Off	132	1	Unli...	1	100%	✓	0	0	0 ns	0
/spi_slave_top/DUT/spi_slave_assertions/idle_to_chk_cmd_...	SVA	✓	Off	72	1	Unli...	1	100%	✓	0	0	0 ns	0
/spi_slave_top/DUT/spi_slave_assertions/read_add_hold_c	SVA	✓	Off	113	1	Unli...	1	100%	✓	0	0	0 ns	0
/spi_slave_top/DUT/spi_slave_assertions/read_add_o_c	SVA	✓	Off	11	1	Unli...	1	100%	✓	0	0	0 ns	0
/spi_slave_top/DUT/spi_slave_assertions/read_add_to_idle...	SVA	✓	Off	11	1	Unli...	1	100%	✓	0	0	0 ns	0
/spi_slave_top/DUT/spi_slave_assertions/read_data_hold_c	SVA	✓	Off	144	1	Unli...	1	100%	✓	0	0	0 ns	0
/spi_slave_top/DUT/spi_slave_assertions/read_data_o_c_1	SVA	✓	Off	8	1	Unli...	1	100%	✓	0	0	0 ns	0
/spi_slave_top/DUT/spi_slave_assertions/read_data_o_c_2	SVA	✓	Off	34	1	Unli...	1	100%	✓	0	0	0 ns	0
/spi_slave_top/DUT/spi_slave_assertions/read_data_o_c_3	SVA	✓	Off	96	1	Unli...	1	100%	✓	0	0	0 ns	0
/spi_slave_top/DUT/spi_slave_assertions/read_data_to_idle...	SVA	✓	Off	8	1	Unli...	1	100%	✓	0	0	0 ns	0
/spi_slave_top/DUT/spi_slave_assertions/rst_n_ns_c	SVA	✓	Off	12	1	Unli...	1	100%	✓	0	0	0 ns	0
/spi_slave_top/DUT/spi_slave_assertions/rst_n_o_c	SVA	✓	Off	12	1	Unli...	1	100%	✓	0	0	0 ns	0
/spi_slave_top/DUT/spi_slave_assertions/write_hold_c	SVA	✓	Off	457	1	Unli...	1	100%	✓	0	0	0 ns	0
/spi_slave_top/DUT/spi_slave_assertions/write_o_c	SVA	✓	Off	41	1	Unli...	1	100%	✓	0	0	0 ns	0
/spi_slave_top/DUT/spi_slave_assertions/write_to_idle_c	SVA	✓	Off	41	1	Unli...	1	100%	✓	0	0	0 ns	0



5. Code Coverage:

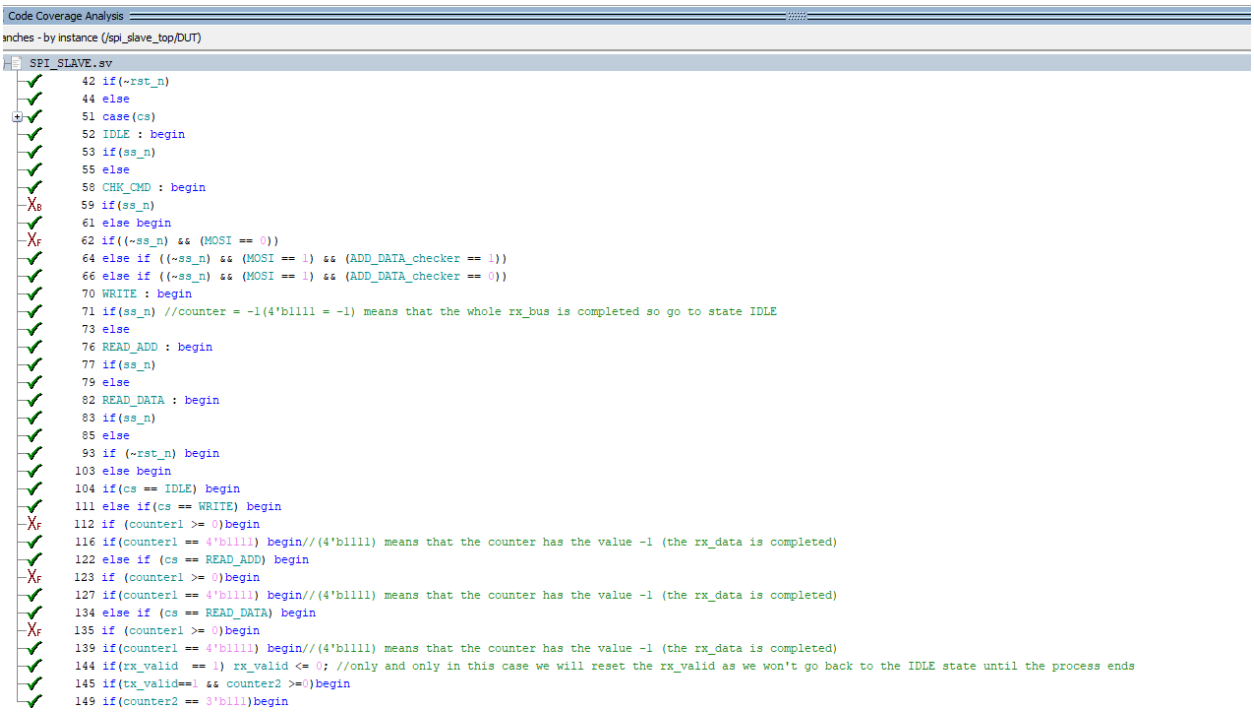
- Statement:

SPI_SLAVE.sv

```
14 assign clk = spi_slaveif.clk;
15 assign rst_n = spi_slaveif.rst_n;
16 assign MOSI = spi_slaveif.MOSI;
17 assign tx_valid = spi_slaveif.tx_valid;
18 assign ss_n = spi_slaveif.ss_n;
19 assign tx_data = spi_slaveif.tx_data;
40 always @(posedge clk)
43 cs <= IDLE;
45 cs <= ns ;
49 always @(*) begin
50 ns = cs ;
54 ns = IDLE;
56 ns = CHK_CMD;
E  60 ns = IDLE;
63 ns = WRITE;
65 ns = READ_ADD;
67 ns = READ_DATA;
72 ns = IDLE;
74 ns = WRITE;
78 ns = IDLE;
80 ns = READ_ADD;
84 ns = IDLE;
86 ns = READ_DATA;
92 always @(posedge clk) begin
94 counter1 <= 9; //as the first bit entered will be the MSB
95 counter2 <= 7; // as the first bit outted will be the MSB
96 ADD_DATA_checker <= 1; // as reading address first is the default
97 bus <= 0;
98 rx_data <= 0;
99 rx_valid <= 0;
100 MISO  <= 0; // making the default output is zero
105 rx_valid <= 0;
106 counter1 <= 9 ; //to start the same proccess in other states without resetting
107 counter2 <= 7 ;
108 MISO <= 0;
113 bus[counter1] <= MOSI;
114 counter1 <= counter1 - 1;   //decrement the counter to fill the whole output rx_data
117 rx_valid <= 1;
118 rx_data <= bus ; //sending the parallel data to the spi_slave
124 bus[counter1] <= MOSI;
125 counter1 <= counter1 - 1;   //decrement the counter to fill the whole output rx_data
128 rx_valid <= 1;
129 rx_data <= bus ; //sending the parallel data to the spi_slave
130 ADD_DATA_checker <= 0; //(means that the read address is recieved) as when this state ends we will go to the READ_DATA state
```

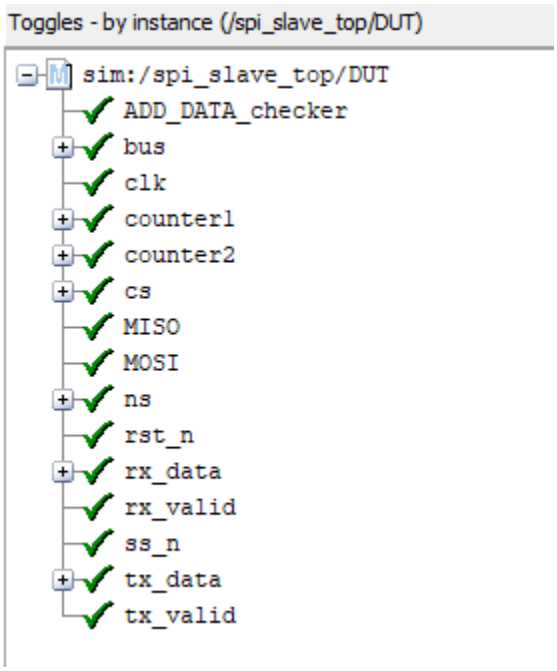
```
130 ADD_DATA_checker <= 0; //(means that the read address is recieved) as when this state ends we will go to the READ_DATA state
136 bus[counter1] <= MOSI;
137 counter1 <= counter1 - 1;   //decrement the counter to fill the whole output rx_data
140 rx_valid <= 1;
141 rx_data <= bus ; //sending the parallel data to the spi_slave
142 counter1 <= 9 ; //only and only in this case we will reset the counter as we won't go back to the IDLE state until the process ends
144 if(rx_valid  == 1) rx_valid <= 0; //only and only in this case we will reset the rx_valid as we won't go back to the IDLE state until the process ends
146 MISO <= tx_data[counter2] ; //counter-2 as it it's an 8 bit bus not 10 bit bus
147 counter2 <= counter2 - 1 ;
150 ADD_DATA_checker <= 1; //(means that we should send another address for reading in the next time) so we will go to READ_ADD state
```

- Branch:

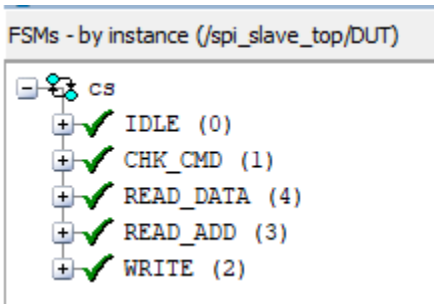


Some branches never fail, but it is acceptable as they are designed not to fail

- Toggle



- FSM:

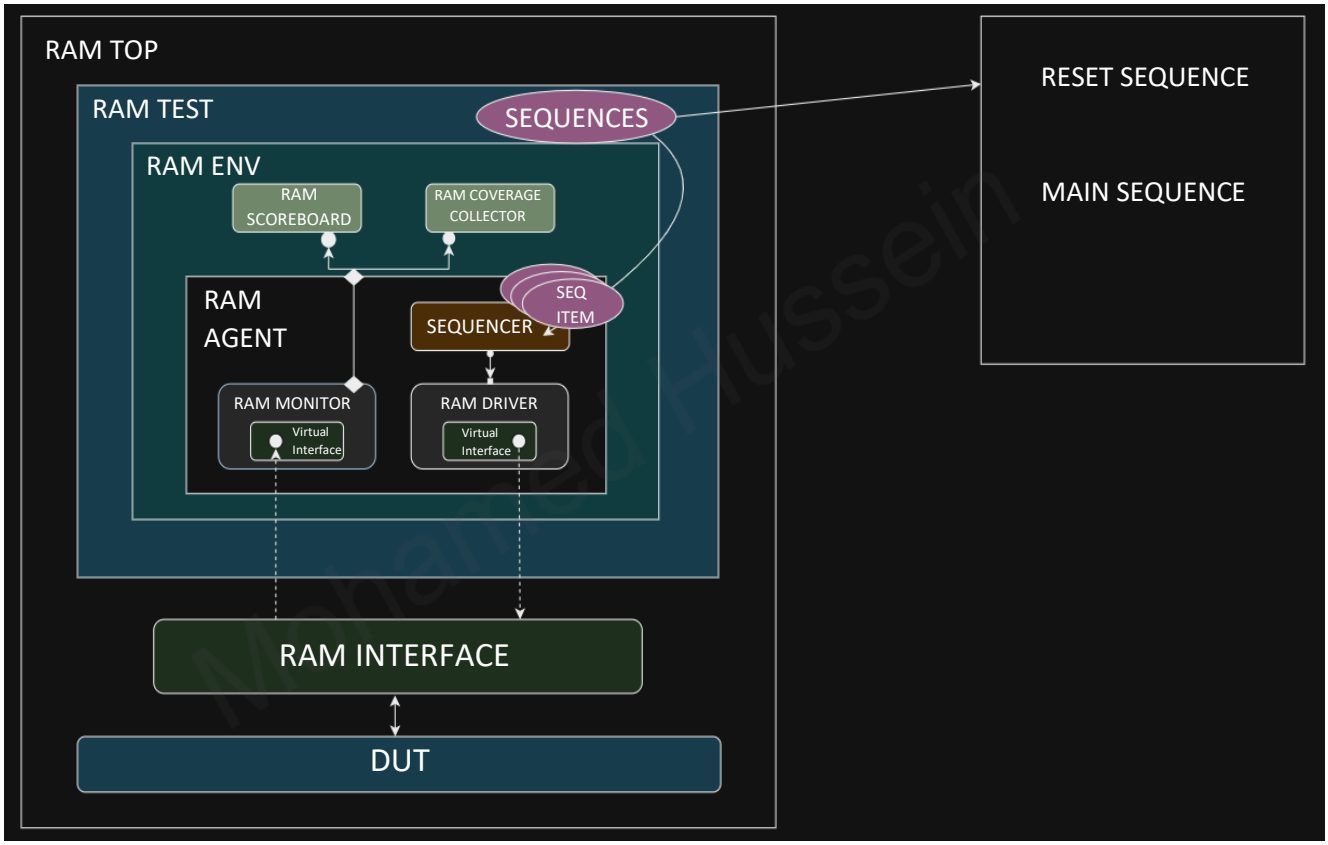


6. UVM Report:

```
#
#   ***********       IMPORTANT RELEASE NOTES         ************
#
#   You are using a version of the UVM library that has been compiled
#   with `UVM_NO_DEPRECATED undefined.
#   See http://www.eda.org/svdb/view.php?id=3313 for more details.
#
#   You are using a version of the UVM library that has been compiled
#   with `UVM_OBJECT_MUST_HAVE_CONSTRUCTOR undefined.
#   See http://www.eda.org/svdb/view.php?id=3770 for more details.
#
#       (Specify +UVM_NO_RELNOTES to turn off this notice)
#
# UVM_INFO verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(277) @ 0: reporter [Questa UVM] QUESTA_UVM-1.2.3
# UVM_INFO verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(278) @ 0: reporter [Questa UVM]  questa_uvm::init(all)
# UVM_INFO @ 0: reporter [RNTST] Running test spi_slave_test...
# UVM_INFO SPI_SLAVE_TEST.sv(59) @ 0: uvm_test_top [run_phase] reset asserted
# ****************************************************************************
# * Questa UVM Transaction Recording Turned ON.                              *
# * recording_detail has been set.                                           *
# *  To turn off, set 'recording_detail' to off:                             *
# * uvm_config_db#(int)            ::set(null, "", "recording_detail", 0); *
# * uvm_config_db#(uvm_bitstream_t)::set(null, "", "recording_detail", 0); *
# ****************************************************************************
# UVM_INFO SPI_SLAVE_TEST.sv(61) @ 20: uvm_test_top [run_phase] reset dasserted
# UVM_INFO SPI_SLAVE_TEST.sv(67) @ 20: uvm_test_top [run_phase] main asserted
# UVM_INFO SPI_SLAVE_TEST.sv(69) @ 20020: uvm_test_top [run_phase] main dasserted
# UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1267) @ 20020: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
# UVM_INFO SPI_SLAVE_SCOREBOARD.sv(177) @ 20020: uvm_test_top.env.sb [report_phase] total successful transactions: 1001
# UVM_INFO SPI_SLAVE_SCOREBOARD.sv(178) @ 20020: uvm_test_top.env.sb [report_phase] total failed transactions: 0
#
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :   10
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [Questa UVM]     2
# [RNTST]     1
# [TEST_DONE]     1
# [report_phase]     2
# [run_phase]     4
# ** Note: $finish    : C:/Users/DELL/AppData/Local/Programs/win64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
#    Time: 20020 ns  Iteration: 60  Instance: /spi_slave_top
```

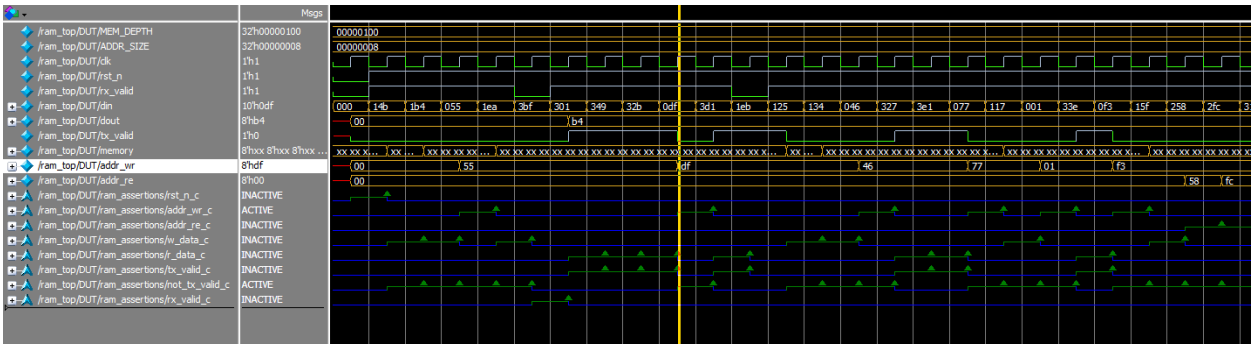
RAM:

1. UVM Structure



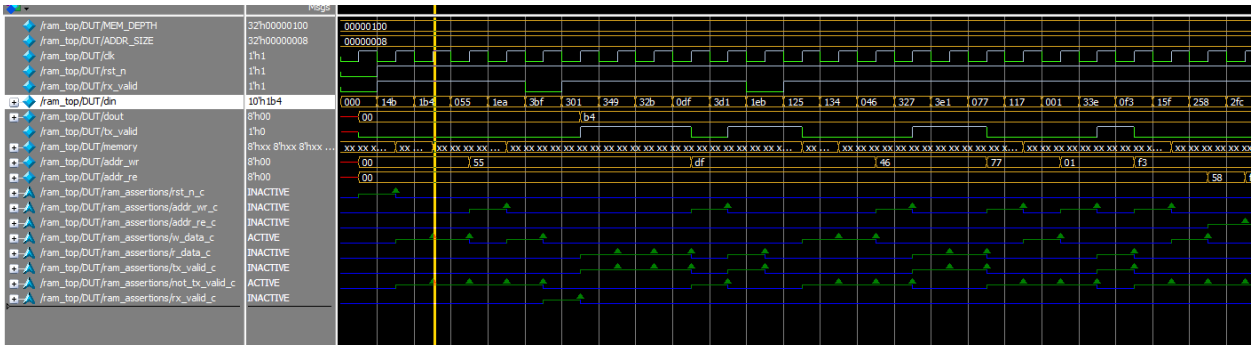
2. Verifying Functionality

- **Write Address:**



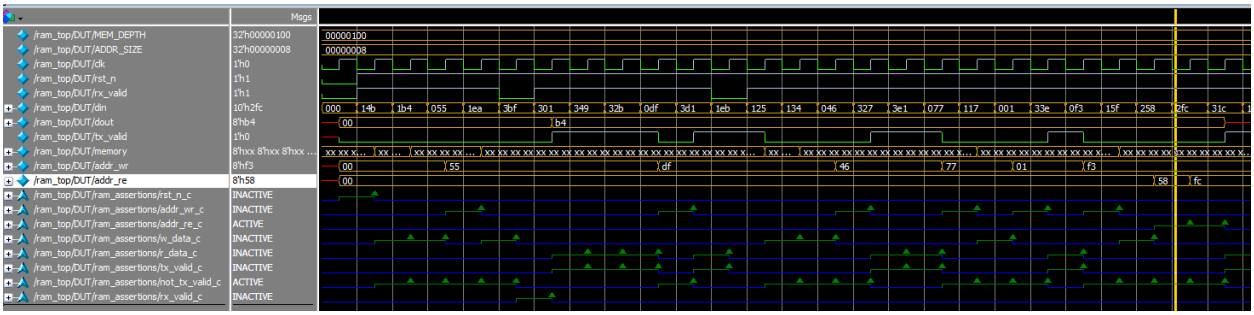
When din [9:8] = 0 it indicates a **write Address** process

- **Write Data:**



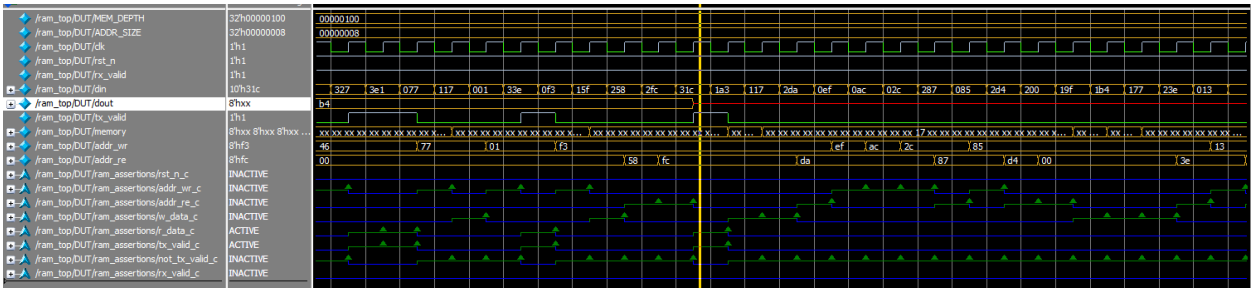
When din[9:8] = 1 it indicates a **write data** process

- **Read Address:**



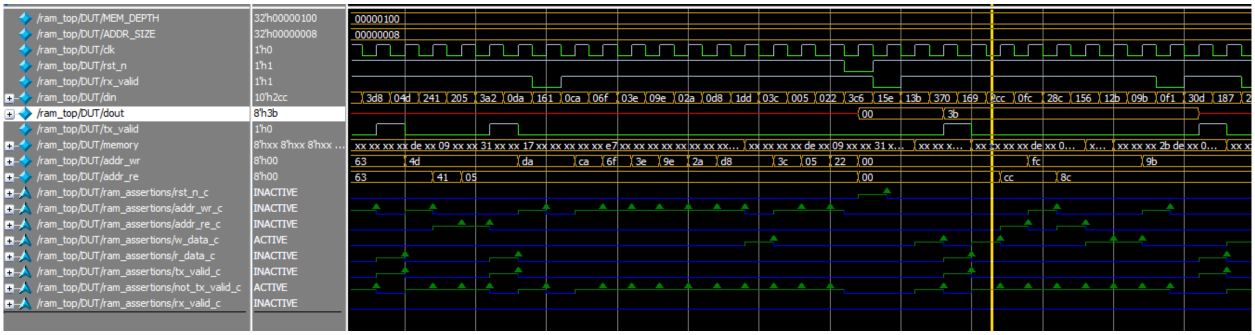
Din [9:8] = 2 it indicates a **Read Address** process

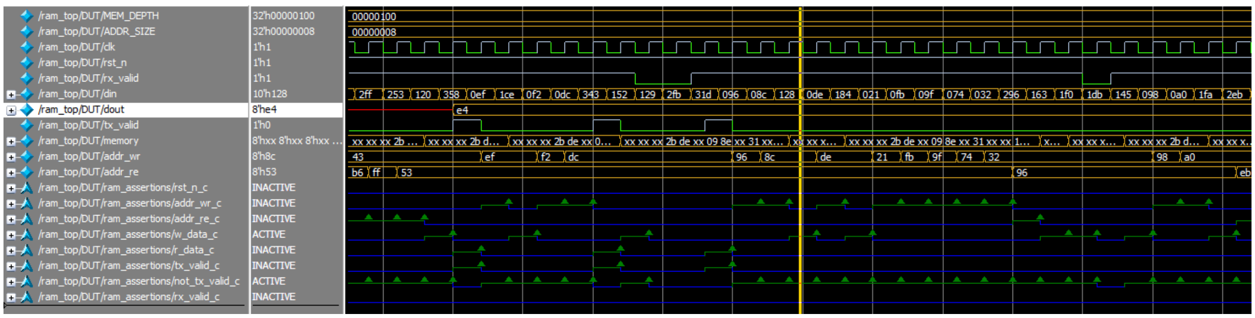
- **Read Data:**

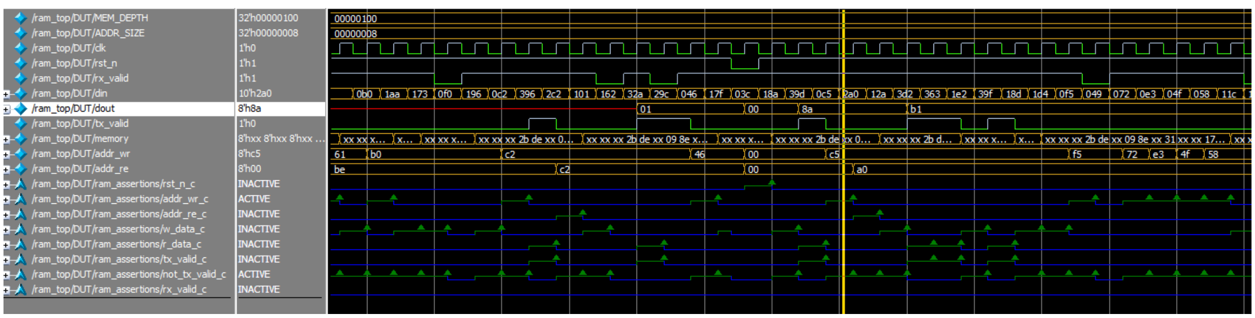


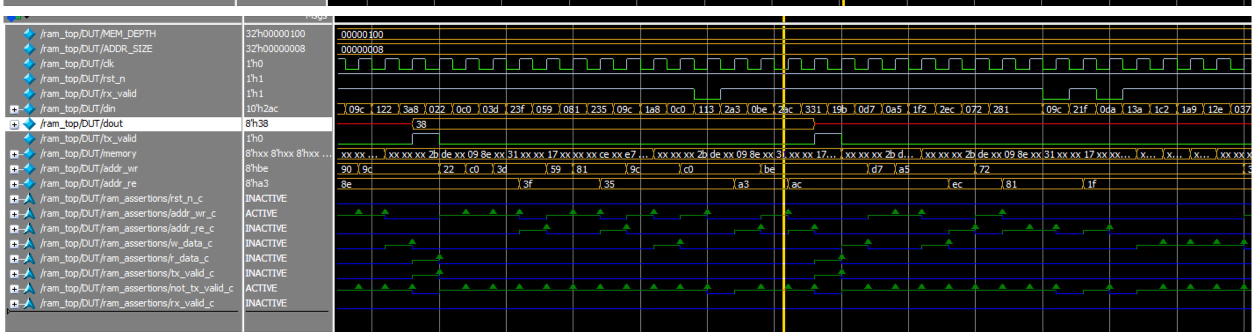
Din [9:8] = 3 it indicates a **Read Data** process

- **Random Snapshots:**

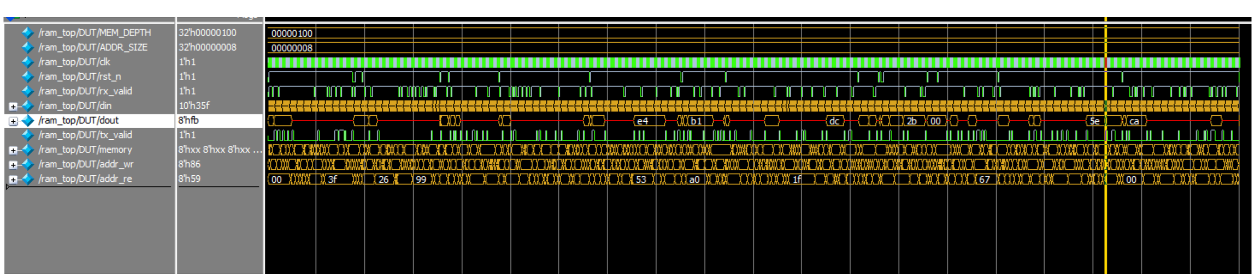








- **Full Waveform:**

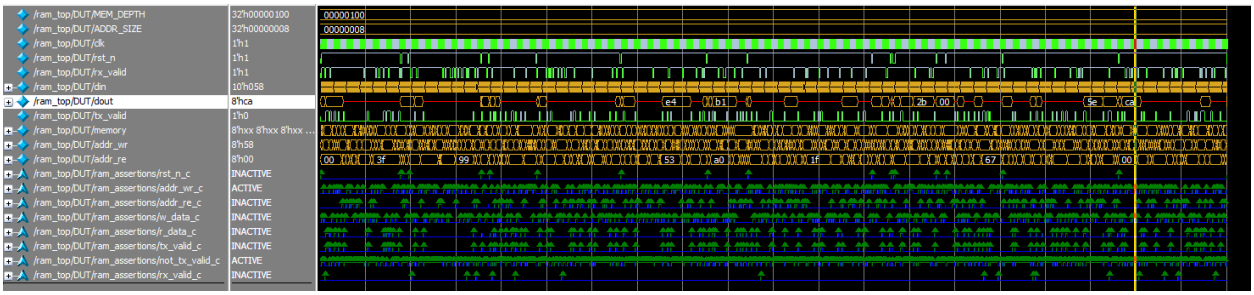


3. Functional Coverage

/ram_coverage_collector_pkg/ram_coverage_collector	100.00%				
TYPE RAM_Cross_Group	100.00%	100	100.00...		auto(1)
CVP RAM_Cross_Group::cp_tx_valid	100.00%	100	100.00...		
bin auto[0]	859	1	100.00...		
bin auto[1]	142	1	100.00...		
CVP RAM_Cross_Group::cp_rx_valid	100.00%	100	100.00...		
bin auto[0]	106	1	100.00...		
bin auto[1]	895	1	100.00...		
CVP RAM_Cross_Group::cp_din9	100.00%	100	100.00...		
bin auto[0]	669	1	100.00...		
bin auto[1]	332	1	100.00...		
CVP RAM_Cross_Group::cp_din8	100.00%	100	100.00...		
bin auto[0]	544	1	100.00...		
bin auto[1]	457	1	100.00...		
CROSS RAM_Cross_Group::wr_addr_C	100.00%	100	100.00...		
bin wr_addr	328	1	100.00...		
CROSS RAM_Cross_Group::rd_addr_C	100.00%	100	100.00...		
bin rd_addr	161	1	100.00...		
CROSS RAM_Cross_Group::wr_data_C	100.00%	100	100.00...		
bin wr_data	277	1	100.00...		
CROSS RAM_Cross_Group::rd_data_C	100.00%	100	100.00...		
bin rd_data	127	1	100.00...		

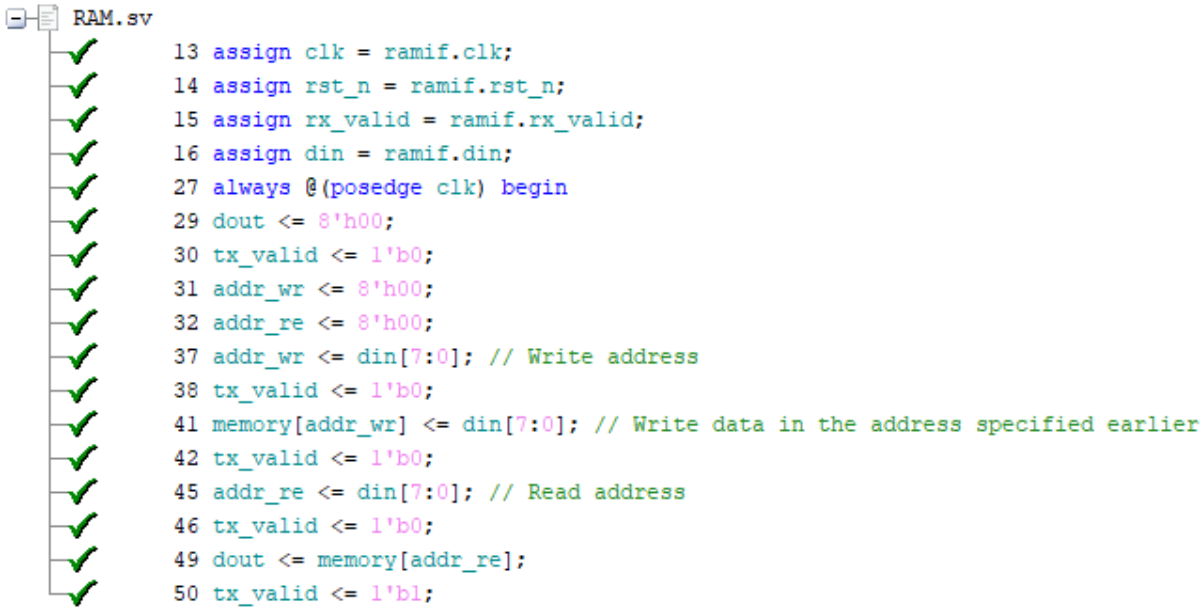
4. Assertions:

/ram_top/DUT/ram_assertions/addr_re_c	SVA		Off	156	1	Unli...	1	100%		0	0	0 ns	0
/ram_top/DUT/ram_assertions/addr_wr_c	SVA		Off	313	1	Unli...	1	100%		0	0	0 ns	0
/ram_top/DUT/ram_assertions/not_tx_valid_c	SVA		Off	738	1	Unli...	1	100%		0	0	0 ns	0
/ram_top/DUT/ram_assertions/r_data_c	SVA		Off	116	1	Unli...	1	100%		0	0	0 ns	0
/ram_top/DUT/ram_assertions/rst_n_c	SVA		Off	18	1	Unli...	1	100%		0	0	0 ns	0
/ram_top/DUT/ram_assertions/rx_valid_c	SVA		Off	17	1	Unli...	1	100%		0	0	0 ns	0
/ram_top/DUT/ram_assertions/tx_valid_c	SVA		Off	125	1	Unli...	1	100%		0	0	0 ns	0
/ram_top/DUT/ram_assertions/w_data_c	SVA		Off	269	1	Unli...	1	100%		0	0	0 ns	0

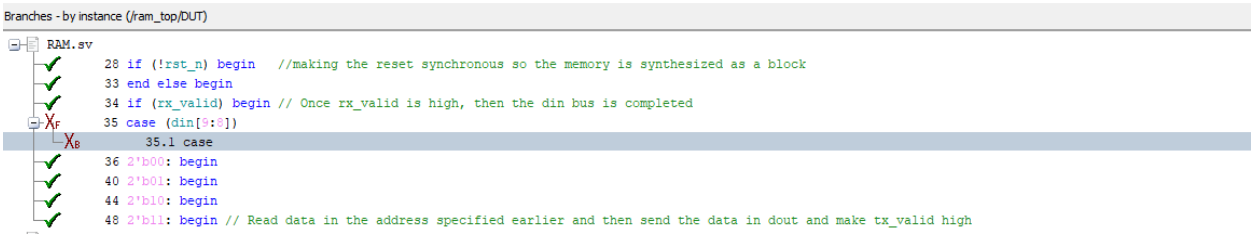


5. Code Coverage:

- Statement:

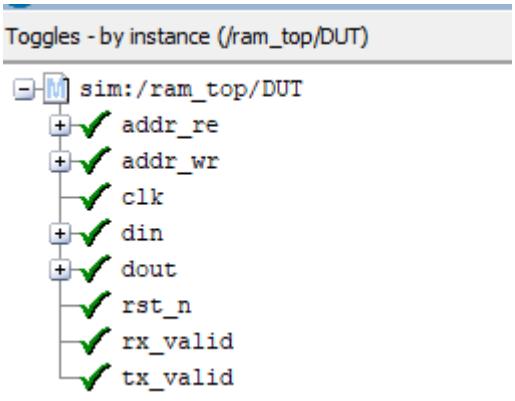


- Branch:



We got all cases except for unknown din[9:8] and as we also did not include **default** in the case statement but I did this to make a RAM block when synthesizing the design, check the design Repository for deeper understanding

- Toggle



6. UVM Report:

```
# ----------------------------------------------------------------
#
#   ***********       IMPORTANT RELEASE NOTES         ************
#
#   You are using a version of the UVM library that has been compiled
#   with `UVM_NO_DEPRECATED undefined.
#   See http://www.eda.org/svdb/view.php?id=3313 for more details.
#
#   You are using a version of the UVM library that has been compiled
#   with `UVM_OBJECT_MUST_HAVE_CONSTRUCTOR undefined.
#   See http://www.eda.org/svdb/view.php?id=3770 for more details.
#
#       (Specify +UVM_NO_RELNOTES to turn off this notice)
#
# UVM_INFO verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(277) @ 0: reporter [Questa UVM] QUESTA_UVM-1.2.3
# UVM_INFO verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(278) @ 0: reporter [Questa UVM]  questa_uvm::init(all)
# UVM_INFO @ 0: reporter [RNTST] Running test ram_test...
# UVM_INFO RAM_TEST.sv(45) @ 0: uvm_test_top [run_phase] reset asserted
# ****************************************************************************
# * Questa UVM Transaction Recording Turned ON.                              *
# * recording_detail has been set.                                           *
# *  To turn off, set 'recording_detail' to off:                             *
# * uvm_config_db#(int)            ::set(null, "", "recording_detail", 0); *
# * uvm_config_db#(uvm_bitstream_t)::set(null, "", "recording_detail", 0); *
# ****************************************************************************
# UVM_INFO RAM_TEST.sv(47) @ 20: uvm_test_top [run_phase] reset dasserted
# UVM_INFO RAM_TEST.sv(50) @ 20: uvm_test_top [run_phase] main asserted
# UVM_INFO RAM_TEST.sv(52) @ 20020: uvm_test_top [run_phase] main dasserted
# UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1267) @ 20020: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
# UVM_INFO RAM_SCOREBOARD.sv(93) @ 20020: uvm_test_top.env.sb [report_phase] total successful transactions: 1001
# UVM_INFO RAM_SCOREBOARD.sv(94) @ 20020: uvm_test_top.env.sb [report_phase] total failed transactions: 0
#
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :   10
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [Questa UVM]     2
# [RNTST]     1
# [TEST_DONE]     1
# [report_phase]     2
# [run_phase]     4
# ** Note: $finish    : C:/Users/DELL/AppData/Local/Programs/win64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
#    Time: 20020 ns  Iteration: 60  Instance: /ram_top
```