REPORT

Project (2)

SPI

TEAM #21

PREPARED BY
Nouran Hamdy

SUBMITTED TO
Eng Kareem Waseem

TABLE OF CONTENTS

Verification Plan

GENERAL PLAN:

• White-Box Verification

- (mem, write_addr, read_addr) internal signals are imported from the design to be used in Functional Coverage and Assertions.

• Checking Results

- Outputs' values are checked against golden model.
- Assertions are used for specific test cases.

• Test Cases to Consider

- Reset Signal (rst_n) Functionality across any combination of inputs.
- Write operations in RAM – Filling up RAM with initial data.
- Writing into then reading from different RAM addresses.
- Completely randomized inputs to check for any unexpected behavior.

• General Constraints (For Good Usage of RAM)

- If previous signal = 2'b00 -> Next signal has to be 2'b01 (Write data in the address given previously).
- If previous signal = 2'b10 -> Next signal has to be 2'b11 (Read data from the address given previously).
- If signal = 2'b10 -> The following bits must contain an address that have data written to it in previous cycles.

• Functional Coverage Main Cover points and Cross Coverage

- First 2bits of din (Signal) has taken, and transitioned between, all values.
- Last 8bits of din (Data) has taken corner values.
- All RAM addresses have been exercised.
- Data has taken corner values with each state and transition of Signal.
- All states and transitions occurred when rx_valid was active.
- All RAM addresses have gone through all states of Signal.

• Assertions (Specific Test Cases)

- Stable cases (IDLE behavior) through the effect of control signals (rst_n and rx_valid)
- Normal Operation of RAM through different Signals (1st 2bits of din).
- tx_valid behavior, when it must (and must not) be active.
- Writing into a specific address.
- Reading from a specific address.
- Writing into a specific address, then reading from the same location.

Verification Requirements

Test Item	Description and Expected Output Behavior	Stimulus Generation
Reset Signal (rst_n)	RAM outputs (dout and tx_valid) should reset to zero on reset assertion.	rst_n = 0. All other inputs are randomized under no constraints.
Control Signal (rx_valid)	RAM should accept din if rx_valid is active. RAM contents should be stable if rx_valid is inactive.	Included in all tasks of stimulus generation where rx_valid is either active or randomized to be active most of the time.
RAM Data Initialization (Write Operations)	rx_valid must be high for receiving an address or for a write operation to occur. Signal must be 2'b00 for saving a write address. Signal must be 2'b01 for writing data into the pre-given address. Signals (2'b00 and 2'b01) should be consecutive for a correct write operation.	rx_valid = 1. rst_n is constrained to be mostly inactive. First 2 bits of din (Signal) are randomized to be always 2'b00 or 2'b01 for writing operations only. Signal is constrained to follow any 2'b00 by 2'b01 (Signal should equal 2'b01 if the previous signal was 2'b00). Last 8 bits of din (Data) are constrained to mostly take values of corners (All Ones, Zero, Walking Ones).
Normal RAM Operation (Writing and Reading from various addresses)	Same as (RAM Data Initialization) Description for correct write operations, and: Signal must be 2'b10 for saving a read address. Signal must be 2'b11 for reading data from the pre-given address. Signals (2'b10 and 2'b11) should be consecutive for a correct read operation.	rst_n is randomized and constrained to be mostly inactive. rx_valid is randomized and constrained to be mostly active. Signal is constrained to follow any 2'b10 by 2'b11 (Signal should equal 2'b11 if the previous signal was 2'b10). Last 8 bits of din (Data) are constrained to mostly take values of corners (All Ones, Zero, Walking Ones). If signal = 2'b10, the following bits are constrained to contain an address that have data written to it in previous cycles.

Verification Requirements

Test Item	Functional Coverage	Functionality Check	Assertions
Reset Signal (rst_n)	Included in Cross Coverage: cross_rst_addr: All RAM addresses heve experienced the effect of rst_n. cross_rst_signal: All States and Transitions heve experienced the effect of rst_n.	Outputs must equal zero. Checked in check_reset task.	Property: reset_asserted
Control Signal (rx_valid)	Included in Cross Coverage: cross_rx_addr: All RAM addresses heve experienced the effect of rx_valid. cross_signal_rx: All states and transitions experienced the effect of rx when active and inactive.	Output is checked against golden model.	Property: rx_valid_inactive
RAM Data Initialization (Write Operations)	Included in Coverpoints: Signal_cp.WR_states: Signal has taken values needed for writing operation. Data_cp: Data has taken corner values. WR_addr_cp: All RAM addresses have been covered. Included in Cross Coverage: cross_signal_data.WR_data: All corners of data came along Write States. cross_signal_rx.WR_valid: WR states are entered at active rx_valid. cross_signal_addr.WR_OP: The same addresses passed accross both Write States.	Output is checked against golden model.	Property: save_wr_addr: Checks Saving Write Address into write_addr (internal signal). wr_data: Checks if data is written in RAM at the required address. normal_write_op: Checks full write operation (Sending address followed by data).
Normal RAM Operation (Writing and Reading from various addresses)	Included in Coverpoints: Signal_cp: Signal has taken, and transitioned between, all values. Data_cp: Data has taken corner values. WR_addr_cp: All RAM addresses have been covered. Included in Cross Coverage: cross_signal_data: Data has taken corner values with each state and transition of Signal. cross_signal_rx: All states and transitions occurred when rx_valid was active. cross_signal_addr: The same addresses passed accross all states and transitions.	Output is checked against golden model.	Property: save_rd_addr: Checks Saving Read Address into read_addr (internal signal). rd_data: Checks if the data read is the same as the data in RAM at the required address. tx_valid_active: Checks if tx_valid is active when data is ready. tx_valid_inactive: Checks tx_valid is not active when data is not ready. normal_read_op: Checks full read operation (Sending address followed by data). main_op: Checks writing then reading from the same address.

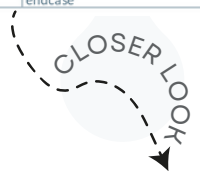
Bug Report

Bug	Original Code	Fix	Lines (Original Version)	Lines (Edited Version)
(Not a Bug) The design is converted to an sv file to access reg-defined internal signals from the sva file (mem, write_addr, read_addr) datatypes' are converted to logic instead of reg	reg [ADDR_SIZE-1:0] write_addr, read_addr; reg [ADDR_SIZE-1:0] mem [MEM_DEPTH-1:0];	logic [ADDR_SIZE-1:0] write_addr, read_addr; logic [ADDR_SIZE-1:0] mem [MEM_DEPTH-1:0];	[13:14]	[13:14]

Parameter (ADDR_SIZE) should be instead of the actual size to respond correctly to changes in parameters	case (din[9:8])
	write_addr <= din[7:0];
	mem [write_addr] <= din[7:0];
	read_addr <= din[7:0];
		case (din[ADDR_SIZE+1:ADDR_SIZE])
		write_addr <= din[ADDR_SIZE-1:0];
		mem [write_addr] <= din[ADDR_SIZE-1:0];
		read_addr <= din[ADDR_SIZE-1:0];
			29
			31
			35
			39
				65
				67
				71
				75

Memory elements are not supposed to reset to zero on each reset assertion, instead outputs and internal signals reset to zero	if(~rst_n) begin
	for (i=0; i < MEM_DEPTH; i=i+1) begin
	mem[i] = 0;
	end
	end
		if(~rst_n) begin
		dout <= 0;
		tx_valid <= 0;
		write_addr <= 0;
		read_addr <= 0;
		end
			[23:27]	[58:63]

(Not a Bug)
No default case
	case (din[9:8])
	2'b00: begin
	write_addr <= din[7:0];
	tx_valid <=0;
	end
	2'b01: begin
	mem [write_addr] <= din[7:0];
	tx_valid <=0;
	end
	2'b10: begin
	read_addr <= din[7:0];
	tx_valid <=0;
	end
	2'b11: begin
	dout <= mem[read_addr];
	tx_valid <=1;
	end
	endcase
		case (din[ADDR_SIZE+1:ADDR_SIZE])
		2'b00: begin
		write_addr <= din[ADDR_SIZE-1:0];
		tx_valid <=0;
		end
		2'b01: begin
		mem [write_addr] <= din[ADDR_SIZE-1:0];
		tx_valid <=0;
		end
		2'b10: begin
		read_addr <= din[ADDR_SIZE-1:0];
		tx_valid <=0;
		end
		default: begin
		dout <= mem[read_addr];
		tx_valid <=1;
		end
		endcase
			[29:46]	[65:82]



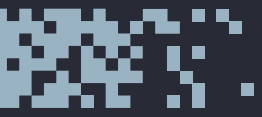
Bug	Original Code	Fix
(Not a Bug) The design is converted to an sv file to access reg-defined internal signals from the sva file (mem, write_addr, read_addr) datatypes' are converted to logic instead of reg	reg [ADDR_SIZE-1:0] write_addr, read_addr; reg [ADDR_SIZE-1:0] mem [MEM_DEPTH-1:0];	logic [ADDR_SIZE-1:0] write_addr, read_addr; logic [ADDR_SIZE-1:0] mem [MEM_DEPTH-1:0];

Parameter (ADDR_SIZE) should be instead of the actual size to respond correctly to changes in parameters	case (din[9:8])
	write_addr <= din[7:0];
	mem [write_addr] <= din[7:0];
	read_addr <= din[7:0];
		case (din[ADDR_SIZE+1:ADDR_SIZE])
		write_addr <= din[ADDR_SIZE-1:0];
		mem [write_addr] <= din[ADDR_SIZE-1:0];
		read_addr <= din[ADDR_SIZE-1:0];

Memory elements are not supposed to reset to zero on each reset assertion, instead outputs and internal signals reset to zero	if(~rst_n) begin
	for (i=0; i < MEM_DEPTH; i=i+1) begin
	mem[i] = 0;
	end
	end
		if(~rst_n) begin
		dout <= 0;
		tx_valid <= 0;
		write_addr <= 0;
		read_addr <= 0;
		end

(Not a Bug)
No default case
	case (din[9:8])
	2'b00: begin
	write_addr <= din[7:0];
	tx_valid <=0;
	end
	2'b01: begin
	mem [write_addr] <= din[7:0];
	tx_valid <=0;
	end
	2'b10: begin
	read_addr <= din[7:0];
	tx_valid <=0;
	end
	2'b11: begin
	dout <= mem[read_addr];
	tx_valid <=1;
	end
	endcase
		case (din[ADDR_SIZE+1:ADDR_SIZE])
		2'b00: begin
		write_addr <= din[ADDR_SIZE-1:0];
		tx_valid <=0;
		end
		2'b01: begin
		mem [write_addr] <= din[ADDR_SIZE-1:0];
		tx_valid <=0;
		end
		2'b10: begin
		read_addr <= din[ADDR_SIZE-1:0];
		tx_valid <=0;
		end
		default: begin
		dout <= mem[read_addr];
		tx_valid <=1;
		end
		endcase

Code Snippets



DESIGN

Before (Original)

```
17
18  ////////////////////////////////Original Code////////////////////////////////
19
20  /*
21      integer i=0;
22      always @(posedge clk,negedge rst_n) begin
23          if(~rst_n) begin
24              for (i=0; i < MEM_DEPTH; i=i+1) begin       //memory values should not equal zero following each reset, only Module outputs (and wr/rd addresses).
25                  mem[i] = 0;
26              end
27          end
28          else if(rx_valid) begin
29              case (din[9:8])                             //Should use parameter (ADDR_SIZE) instead of actual size
30                  2'b00: begin
31                      write_addr <= din[7:0];
32                      tx_valid <=0;
33                  end
34                  2'b01: begin
35                      mem [write_addr] <= din[7:0];
36                      tx_valid <=0;
37                  end
38                  2'b10: begin
39                      read_addr <= din[7:0];
40                      tx_valid <=0;
41                  end
42                  2'b11: begin                            //For Code Coverage = 100% -> This branch is moved to default
43                      dout <= mem[read_addr];
44                      tx_valid <=1;
45                  end
46              endcase
47          end
48          else
49              tx_valid =0;
50      end
51  */
```

After (Edited)

```
55  ////////////////////////////////Edited Code////////////////////////////////
56
57      always @(posedge clk,negedge rst_n) begin
58          if(~rst_n) begin
59              dout <= 0;
60              tx_valid <= 0;
61              write_addr <= 0;
62              read_addr <= 0;
63          end
64          else if(rx_valid) begin
65              case (din[ADDR_SIZE+1:ADDR_SIZE])
66                  2'b00: begin
67                      write_addr <= din[ADDR_SIZE-1:0];
68                      tx_valid <=0;
69                  end
70                  2'b01: begin
71                      mem [write_addr] <= din[ADDR_SIZE-1:0];
72                      tx_valid <=0;
73                  end
74                  2'b10: begin
75                      read_addr <= din[ADDR_SIZE-1:0];
76                      tx_valid <=0;
77                  end
78                  default: begin
79                      dout <= mem[read_addr];
80                      tx_valid <=1;
81                  end
82              endcase
83          end
84      end
```

Code Snippets

PACKAGE

Class Properties and Pre-defined Variables

```
1   package RAM_pkg;
2
3       typedef enum logic [1:0] {STORE_WR_ADDR, WRITE_DATA, STORE_RD_ADDR, READ_DATA} signal_e;
4       typedef bit [1:0] two_bit;
5
6       class RAM_class #(int ADDR_SIZE = 8);
7
8           bit clk;
9           rand bit rst_n, rx_valid;
10          rand logic [ADDR_SIZE-1:0] data;
11          rand signal_e signal;
12
13          //For post randomization
14          logic [ADDR_SIZE-1:0] data_old;
15          signal_e signal_old;
16
17          //For specifying valid addresses (that have been written previously)
18          logic [ADDR_SIZE-1:0] valid_addr_q[$];
19          bit [ADDR_SIZE-1:0] WR_addr;
20
21
22          //Parameters for data Constraints
23          parameter ONES = (2**ADDR_SIZE) - 1;
24          parameter ZERO = 0;
25
26          rand bit [1:0] selector_data;
27
```

Constraints

```
28          constraint c {
29              rst_n           dist {0:=5,     1:=95};                     //rst_n is inactive most of the time
30              rx_valid        dist {0:=30,    1:=70};                     //rx_valid is active most of the time
31
32              (signal_old == STORE_WR_ADDR) -> signal == WRITE_DATA;      //Whenever a "SAVE WR ADDRESS" signal is sent, it's followed by a "WRITE DATA" signal
33
34              (signal_old == STORE_RD_ADDR) -> signal == READ_DATA;       //Whenever a "SAVE RD ADDRESS" signal is sent, it's followed by a "READ DATA" signal
35
36              selector_data   dist {[0:1]:/40, 2:=40, 3:=20};            //Data is at its corner cases most of the time
37
38              if(selector_data == 0)          data == {ONES};
39              else if(selector_data == 1)     data == {ZERO};
40              else if(selector_data == 2)     $countones(data) == 1;
41          }
42
43          constraint valid_addr {
44              (signal == STORE_RD_ADDR) -> data inside {valid_addr_q};    //"SAVE RD ADDRESS" has to be followed by an address that is readable
45          }
46
47          function void post_randomize;
48              data_old    = data;
49              signal_old  = signal;
50
51              if(rst_n && rx_valid && (signal == WRITE_DATA)) valid_addr_q.push_front(WR_addr);   //Whenever a successful WRITE OP is done in a specific address,
52          endfunction                                                                             //that address is pushed to the valid_addr queue
53
```

Code Snippets

Covergroup

```
covergroup RAM_cg;
    signal_cp:      coverpoint signal{
        bins WR_states  = {STORE_WR_ADDR, WRITE_DATA};
        bins RD_states  = {STORE_RD_ADDR, READ_DATA};
        bins WR_to_RD   = (STORE_WR_ADDR => WRITE_DATA => STORE_RD_ADDR, READ_DATA);    //Normal Operation: Read after Write
        bins RD_to_WR   = (STORE_RD_ADDR => READ_DATA => STORE_WR_ADDR, WRITE_DATA);    //Proposed Scenario: Write after Read
    }

    data_cp:        coverpoint data{                    //Ensuring access to all bits of data and memory
        bins ALL_ones       = {ONES};
        bins ZERO           = {0};
        bins Walking_ones   = {2**(ADDR_SIZE-1), 2**(ADDR_SIZE-2), 2**(ADDR_SIZE-3), 2**(ADDR_SIZE-4), 2**(ADDR_SIZE-5), 2**(ADDR_SIZE-6), 2**(ADDR_SIZE-7), 2**(ADDR_SIZE-8)};
    }
    rx_valid_cp:    coverpoint rx_valid;
    WR_addr_cp:     coverpoint WR_addr;                                             //All RAM addresses have been covered

    cross_rst_addr:         cross rst_n, WR_addr_cp;                                //All RAM addresses heve experienced the effect of rst_n
    cross_rx_addr:          cross rx_valid_cp, WR_addr_cp;                          //All RAM addresses heve experienced the effect of rx_valid

    cross_rst_signal:       cross rst_n, signal_cp;                                 //All States and Transitions heve experienced the effect of rst_n

    cross_signal_data:      cross signal_cp, data_cp{
        bins WR_data        = binsof(signal_cp.WR_states) && binsof(data_cp);        //All corners of data came along Write States
        bins RD_data        = binsof(signal_cp.RD_states) && binsof(data_cp);        //All corners of data came along Read States
        bins data_WR_trans  = binsof(signal_cp.WR_to_RD) && binsof(data_cp);         //All corners of data came along the transition from Write States to Read States
        bins data_RW_trans  = binsof(signal_cp.RD_to_WR) && binsof(data_cp);         //All corners of data came along the transition from Read States to Write States
    }

    cross_signal_rx:        cross signal_cp, rx_valid_cp{
        bins WR_valid       = binsof(signal_cp.WR_states) && binsof(rx_valid_cp) intersect {1};     //WR states are entered at active rx_valid
        bins RD_valid       = binsof(signal_cp.RD_states) && binsof(rx_valid_cp) intersect {1};     //RD states are entered at active rx_valid
        bins WR_to_RD_valid = binsof(signal_cp.WR_to_RD) && binsof(rx_valid_cp) intersect {1};      //Transition from WR to RD states is done at active rx_valid
        bins RD_to_WR_valid = binsof(signal_cp.RD_to_WR) && binsof(rx_valid_cp) intersect {1};      //Transition from RD to WR states is done at active rx_valid
        bins states_IDLE    = binsof(signal_cp) && binsof(rx_valid_cp) intersect {0};            //All states and transitions passed IDLE state (inactive rx_valid)
    }

    cross_signal_addr:      cross signal_cp, WR_addr_cp{
        bins WR_OP          = binsof(signal_cp.WR_states) && binsof(WR_addr_cp);      //The same addresses passed accross both Write States
        bins RD_OP          = binsof(signal_cp.RD_states) && binsof(WR_addr_cp);      //The same addresses passed accross both Read States
        bins WR_to_RD_addr  = binsof(signal_cp.WR_to_RD) && binsof(WR_addr_cp);      //The same addresses passed accross WR to RD Transition
        bins RD_to_WR_addr  = binsof(signal_cp.RD_to_WR) && binsof(WR_addr_cp);      //The same addresses passed accross WR to RD Transition
    }
endgroup
```

Code Snippets

TESTBENCH

Stimulus Generation

```
initial begin

    //Assert Reset - Initial State
    assert_reset;



    /*TEST 0:    - Checks (rst_n) Functionality
                 - RAM outputs (dout, tx_valid) reset to zero @(rst_n = 0)
                 - Randomization Under No Constraints*/
    stimulus_gen_reset;


    //Deassert Reset
    deassert_reset;



    /*Writing data  - Filling up RAM with initial values
                    - rx_valid = 1
                    - 1st 2-bits of din (signal) = 2'b00: Save WR Address
                    - THEN:              signal  = 2'b01: WRITE Operation*/
    ram_initial;



    /*TEST 1:    - Checks Normal Operation of RAM
                 - RAM should use din                    if(rx_valid = 1)
                 - Saves WR address                      if(signal = 2'b00)
                 - Writes data in pre-given address      if(signal = 2'b01)
                 - Saves RD address                      if(signal = 2'b10)
                 - Reads data from pre-given address     if(signal = 2'b11)
    */
    stimulus_gen1;


    //TEST 2: Complete Randomization
    stimulus_gen2;



    //Correct Count and Error Count Display
    $display("At End of Simulation: Correct Count = %0d, Error Count = %0d", correct_count, error_count);
    @(negedge clk); $stop;

end
```

Code Snippets

Tasks

```
//Assert Reset
task assert_reset;
    rst_n = 0;
    ram_c.c.constraint_mode(0);

    @(negedge clk);
endtask

//TEST 0: Reset Asserted
task stimulus_gen_reset;
    for(int i=0; i<TESTS; i++) begin
        assert(ram_c.randomize());
        rx_valid    = ram_c.rx_valid;
        din         = {two_bit'(ram_c.signal), ram_c.data};

        signal          = ram_c.signal;
        data            = ram_c.data;

        check_reset;
    end
endtask

//CHECKER: Reset Asserted
task check_reset;
    @(negedge clk);

    if(dout !== 0) begin
        $display("ERROR: (Reset Asserted) -> Output -dout- equals %0h, but should equal 0 \t\t--time: %0t", dout, $time);
        error_count++;
    end
    else correct_count++;

    if(tx_valid !== 0) begin
        $display("ERROR: (Reset Asserted) -> Output -tx_valid- equals %0h, but should equal 0 \t\t--time: %0t", tx_valid, $time);
        error_count++;
    end
    else correct_count++;

    @(negedge clk);
endtask

//Deassert Reset
task deassert_reset;
    rst_n = 1;
    din[ADDR_SIZE+1 : ADDR_SIZE] = WRITE_DATA;      //Setting a starting point after reset is deasserted
    ram_c.c.constraint_mode(1);

    @(negedge clk);
endtask
```

Code Snippets

Tasks

```
//RAM data initialization
task ram_initial;
    for(int i=0; i<TESTS; i++) begin
        rx_valid = 1;

        //Data randomized to only include write-related signals
        assert(ram_c.randomize() with {signal inside {STORE_WR_ADDR, WRITE_DATA};});
        rst_n    = ram_c.rst_n;
        din      = {two_bit'(ram_c.signal), ram_c.data};

        signal       = ram_c.signal;
        data         = ram_c.data;

        //To access RAM_assoc
        golden_model;
        @(negedge clk);
    end
endtask



//TEST 1: Normal Operation
task stimulus_gen1;
    for(int i=0; i<TESTS; i++) begin
        assert(ram_c.randomize());
        rst_n        = ram_c.rst_n;
        rx_valid     = ram_c.rx_valid;
        din          = {two_bit'(ram_c.signal), ram_c.data};

        signal           = ram_c.signal;
        data             = ram_c.data;

        check_result;
    end
endtask


//TEST 2: Complete Randomization
task stimulus_gen2;
    ram_c.c.constraint_mode(0);

    rst_n = 1;
    for(int i=0; i<TESTS; i++) begin
        assert(ram_c.randomize());
        rx_valid     = ram_c.rx_valid;
        din          = {two_bit'(ram_c.signal), ram_c.data};

        signal           = ram_c.signal;
        data             = ram_c.data;

        check_result;
    end
endtask
```

Code Snippets

Checker

```
//CHECKER: General
task check_result;
    @(posedge clk);

    golden_model;

    if(!rst_n) check_reset;
    else begin
        @(negedge clk);

        if(dout !== dout_exp) begin
            $display("ERROR: Output -dout- equals %0h, but should equal %0h \t\t--time: %0t", dout, dout_exp, $time);
            error_count++;
        end
        else correct_count++;

        if(tx_valid !== tx_valid_exp) begin
            $display("ERROR: Output -tx_valid- equals %0h, but should equal %0h \t\t--time: %0t", tx_valid, tx_valid_exp, $time);
            error_count++;
        end
        else correct_count++;
    end

    //To prepare available addresses for read operations -> passed to class
    ram_c.WR_addr = WR_addr;
    @(negedge clk);
endtask

//Golden Model
task golden_model;
    if(!rst_n) begin
        tx_valid_exp = 0;
        dout_exp = 0;
        WR_addr = 0;
        RD_addr = 0;
    end
    else begin
        if(rx_valid) begin
            case(signal)
            STORE_WR_ADDR:  WR_addr = data;

            WRITE_DATA:     RAM_assoc[WR_addr] = data;

            STORE_RD_ADDR:  RD_addr = data;

            READ_DATA:      dout_exp = RAM_assoc[RD_addr];
            endcase

            if(signal == READ_DATA) tx_valid_exp = 1;
            else tx_valid_exp = 0;
        end
    end
endtask
```

CODE SNIPPETS

ASSERTIONS

IDLE Behavior

```
23  ////////////////////////////////////////////Checking Stable Cases (IDLE)////////////////////////////////////////////
24
25      //Checks Reset Functionality -on internal signals and outputs
26      property reset_asserted;
27          @(posedge clk)
28
29          $fell(rst_n) |=> !(dout || tx_valid || wr_addr || rd_addr);
30      endproperty
31
32      //Ensures No memory manipulation when rx_valid is inactive
33      property rx_valid_inactive;
34          @(posedge clk) disable iff(!rst_n)
35
36          $fell(rx_valid) |=> $stable(mem) && $stable(wr_addr) && $stable(rd_addr);
37      endproperty
```

Original Functionality

```
40
41  ////////////////////////////////////////////Checking Original Functionality////////////////////////////////////////////
42
43      //Checks Saving Write Address
44      property save_wr_addr;
45          logic [ADDR_SIZE-1:0] data_old;
46          @(posedge clk) disable iff(!rst_n || !rx_valid)
47
48          (signal == 0, data_old = data) |=> (wr_addr == data_old);
49      endproperty
50
51      //Checks Writing Data
52      property wr_data;
53          logic [ADDR_SIZE-1:0] data_old;
54          @(posedge clk) disable iff(!rst_n || !rx_valid)
55
56          (signal == 1, data_old = data) |=> (mem[wr_addr] == data_old);
57      endproperty
58
59      //Checks Saving Read Address
60      property save_rd_addr;
61          logic [ADDR_SIZE-1:0] data_old;
62          @(posedge clk) disable iff(!rst_n || !rx_valid)
63
64          (signal == 2, data_old = data) |=> (rd_addr == data_old);
65      endproperty
66
67      //Checks Reading Data
68      property rd_data;
69          logic [ADDR_SIZE-1:0] rd_addr_old;
70          @(posedge clk) disable iff(!rst_n || !rx_valid)
71
72          (signal == 3, rd_addr_old = rd_addr) |=> (dout == mem[rd_addr_old]);
73      endproperty
75      //Checks tx_valid is active if data is ready
76      property tx_valid_active;
77          @(posedge clk) disable iff(!rst_n || !rx_valid)
78
79          (signal == 3) |=> tx_valid;
80      endproperty
81
82      //Checks tx_valid is not active if data is not ready (signal = 3)
83      property tx_valid_inactive;
84          @(posedge clk) disable iff(!rst_n || !rx_valid)
85
86          !(signal == 3) |=> !tx_valid;
87      endproperty
```

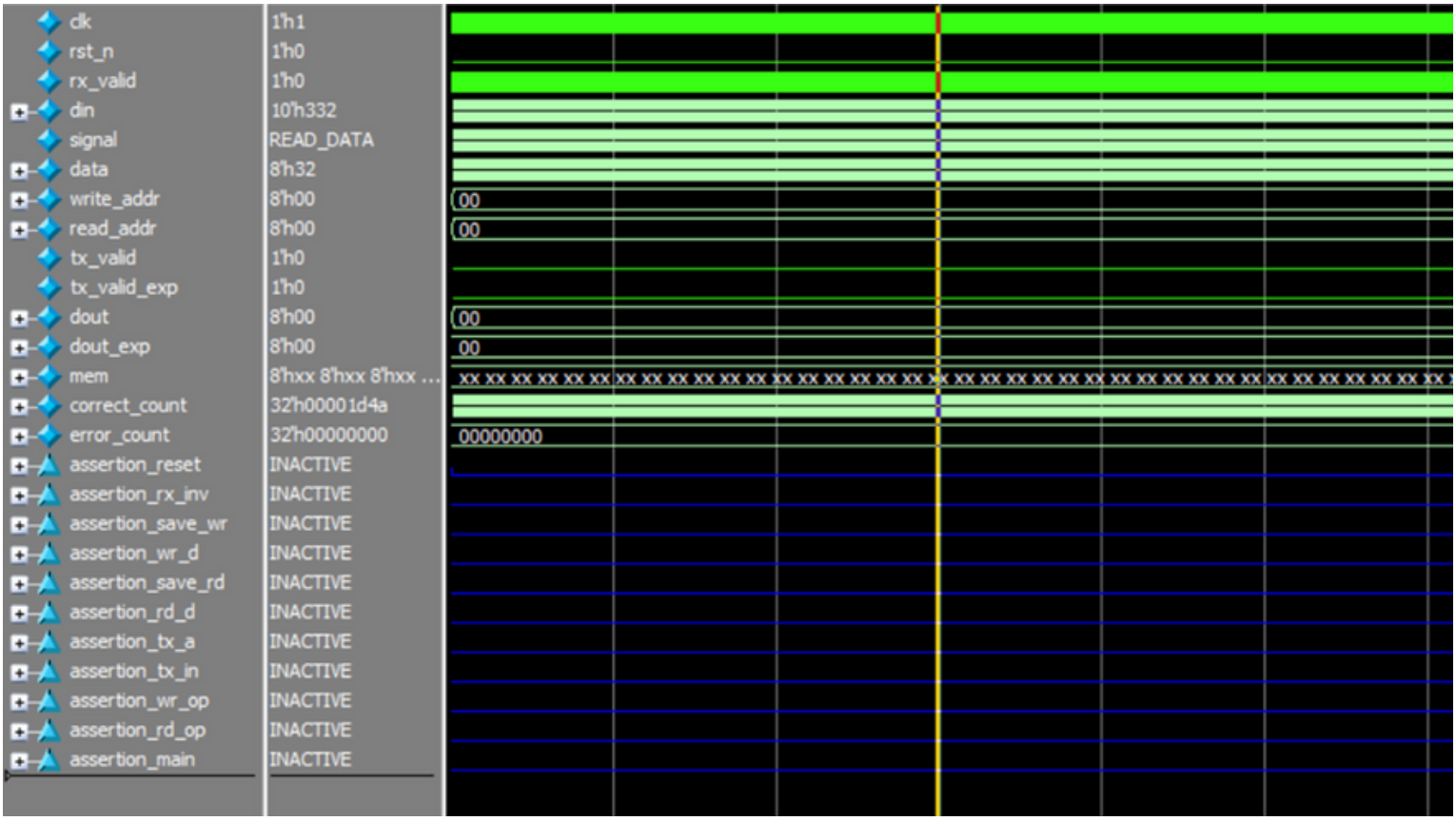
Code Snippets

Different Scenarios

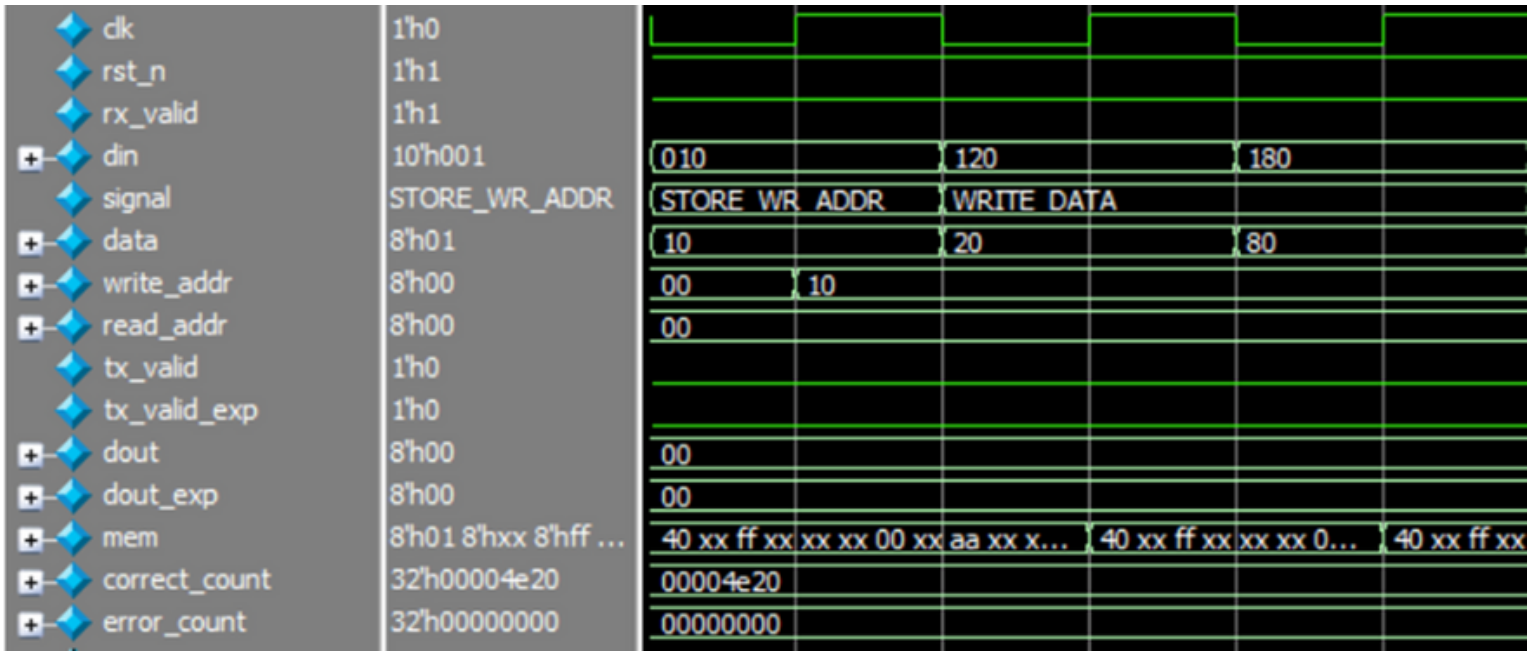
```
89
90      ////////////////////////////////////////////Checking Different Scenarios//////////////////////////////////////////
91
92      //Checks Normal Write Operation
93      property normal_write_op;
94          logic [ADDR_SIZE-1:0] address, content;
95          @(posedge clk) disable iff(!rst_n || !rx_valid)
96
97          //Signal = 0: Saving Address -> Signal = 1: Writing data in that address -> Check: The written data is actually in the location provided in the internal memory
98          (signal == 0, address = data) ##[1:2] (signal == 1, content = data) |=> (mem[address] == content);
99      endproperty
100
101     //Checks Read Operation
102     property normal_read_op;
103         logic [ADDR_SIZE-1:0] address, content;
104         @(posedge clk) disable iff(!rst_n || !rx_valid)
105
106         //Signal = 2: Saving Address -> Signal = 3: Reading data from that address -> Check: The read data is actually the data found at the location provided in the internal memory
107         (signal == 2, address = data) ##[1:2] (signal == 3) |=> (dout == mem[address]);
108     endproperty
109
110     //Checks Writing then Reading from the same location
111     property main_op;
112         logic [ADDR_SIZE-1:0] address, content;
113         @(posedge clk) disable iff(!rst_n || !rx_valid)
114
115         //Signal = 1: Data (content) is written into a specific location (address) -> signal = 1 would override its content (ignored) -> Signal = 2 and rd_addr = previous location (address):
116         ((signal == 1), content = data, address = wr_addr) ##1 (signal !== 1) throughout ( (signal == 2) && (rd_addr == address) )[->1] ##1 (signal == 3) |=> (dout == content);
117     endproperty
118
```

Waveform Snippets

Reset Active: Outputs equal Zero



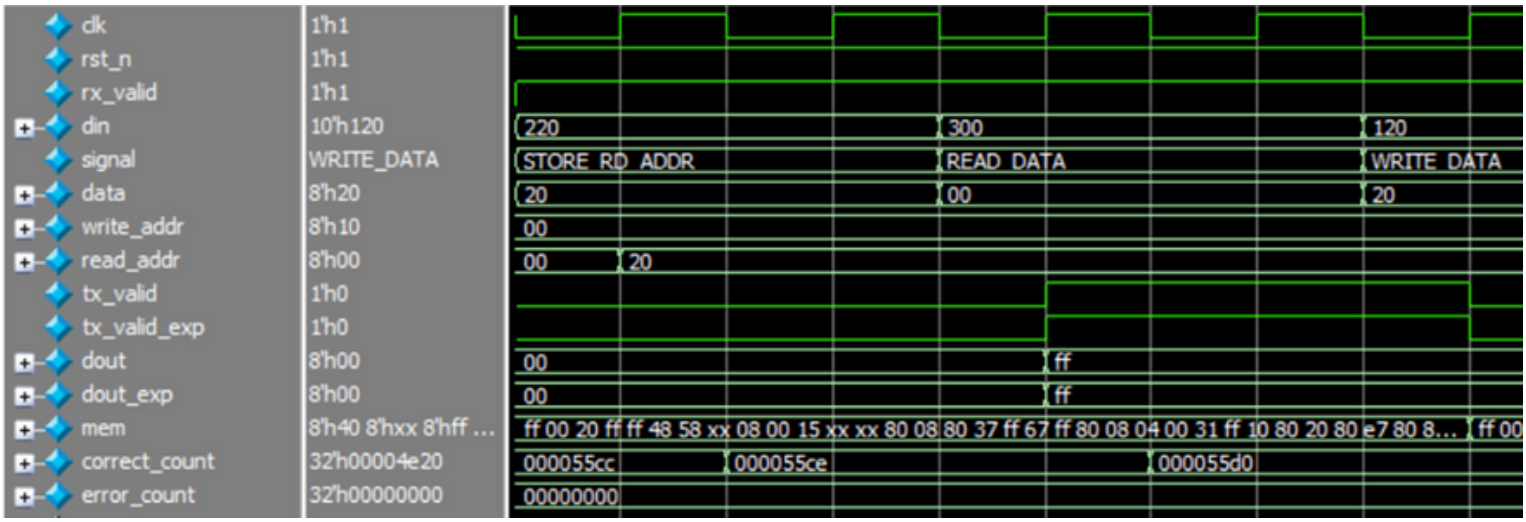
Write Operation:



Address = 8'h10, Data = 8'h20 then 8'h80

QUESTASIM SNIPPETS

Read Operation

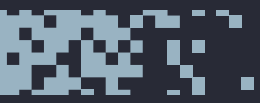


Address = 8'h20, dout = 8'hff and tx_valid is high

Correct Count and Error Count

```
# At End of Simulation: Correct Count = 60000, Error Count = 0
```

Coverage Report



```
7
8     Assertion Coverage:
9         Assertions                   11        11         0   100.00%
10    ----------------------------------------------------------------

36
37    Directive Coverage:
38        Directives                   11        11         0   100.00%

141   Branch Coverage:
142       Enabled Coverage            Bins      Hits    Misses  Coverage
143       ----------------            ----      ----    ------  --------
144       Branches                       7         7         0   100.00%
145

175   Statement Coverage:
176       Enabled Coverage            Bins      Hits    Misses  Coverage
177       ----------------            ----      ----    ------  --------
178       Statements                    13        13         0   100.00%

348   Toggle Coverage:
349       Enabled Coverage            Bins      Hits    Misses  Coverage
350       ----------------            ----      ----    ------  --------
351       Toggles                       76        76         0   100.00%
352

1255
1256  Covergroup Coverage:
1257      Covergroups                    1        na        na   100.00%
1258          Coverpoints/Crosses       11        na        na        na
1259              Covergroup Bins      352       352         0   100.00%
```

Verification Plan

GENERAL PLAN:

·White-Box Verification

- (cs and ns) internal signals are imported from the design to be used in Functional Coverage and Assertions.

· Checking Results

- Outputs' values are checked against golden model.
- Assertions are used for specific test cases.

· Test Cases to Consider

- Reset Signal (rst_n) Functionality across any combination of inputs.
- Normal Write Operation, 1st 2bits of MOSI = 2'b00, then 2'b01 after the next SS_n falling edge.
- Normal Read Operation, 1st 2bits of MOSI = 2'b10, then 2'b11 after the next SS_n falling edge.
- Slave's behavior on transitioning from one sequence to the other.
- Slave's behavior when SS_n stays active after transmission or goes inactive mid-transmission.

· General Considerations (For Good Usage of Slave)

- To simulate RAM's behavior: tx_valid is raised after 8 cycles (while receiving dummy data) in READ_DATA state, stays high until slave leaves that state.
- To simulate Master's behavior: SS_n and MOSI bits are changed every clock cycle.

· Functional Coverage Main Cover points and Cross Coverage

- tx_data has taken corner values.
- cs has gone through all states and important transitions (e.g. wr_to_rd / rd_to_wr).
- Illegal bins to cover illegal transitions (e.g. Idle to Read/Write).
- tx_data has taken corner values at reading states.
- tx_data has taken corner values when tx_valid was high.
- All combinations of control signals (rst_n, SS_n, MOSI) are covered.
- All states have experienced the effect of reset signal (rst_n).
- All states have experienced the effect of control signal (SS_n).

· Assertions (Specific Test Cases)

- Stable cases (IDLE behavior) through the effect of control signals (rst_n and SS_n)
- Necessary and Illegal Transitions of Slave.
- Transitions based on 1st MOSI bit following SS_n falling edge.
- Read Transitions based on 1st 3 MOSI bits following SS_n falling edge.
- No two consecutive READ_ADD or READ_DATA states.
- rx_data and rx_valid values following complete data transmission.
- 1st two bits of rx_data has to indicate the current state of the slave.
- rx_valid and tx_valid cannot be high at the same time.
- rx_valid cannot be high in any checking state.
- MISO's value following tx_valid rising edge.
- MISO cannot be high at any state other than READ_DATA.

Verification Requirements

Test Item	Description and Expected Output Behavior	Stimulus Generation	Functional Coverage	Functionality Check	Assertions
Reset Signal (rst_n)	Slave outputs (rx_data, rx_valid, MISO) should reset to zero on reset assertion.	rst_n = 0. All other inputs are randomized under no constraints.	Included in Cross Coverage: cross_MOSI_SS_rst: All combinations of control signals are covered. cross_rst_states: All States have experienced the effect of rst_n.	Outputs must equal zero. Checked in check_reset task.	Property: reset_asserted
Control Signal (SS_n)	Slave should stay Idle if SS_n is inactive or transition to other states (based on MOSI bits) if SS_n is active. Slave should stay stable if SS_n remained active long after data transmission. Slave should go idle if SS_n is deactivated mid data transmission.	Included in all tasks of stimulus generation where SS_n is either active or randomized to be active most of the time.	Included in Cross Coverage: cross_MOSI_SS_rst: All combinations of control signals are covered. cross_SS_states: All States have experienced the effect of SS_n.	Output is checked against golden model.	Property: SS_inactive
Normal Write Operation of Slave	SS_n should equal 0 throughout the cycle -> 1 after data is collected. Cycle(1): MOSI should equal 0 (WRITE) -> 00 (Send Address) rx_data should be ready and rx_valid should be high after 8 cycles. Cycle(2): MOSI should equal 0 (WRITE) -> 01 (Send Data) rx_data should be ready and rx_valid should be high after 8 cycles.	rst_n = 1. SS_n is directed to be activated at the beginning of each sequence and deactivated following data transmission. MOSI bits are randomized under no constraints.	Included in Coverpoints: cs_cp.States: The slave has gone through write states. cs_cp.wr_add_data: The slave has transitioned from write state to another write (address then data).	Output is checked against golden model.	Property (States): CHK_WR: Checks going from IDLE to CHK_WR state where it waits for the 1st MOSI bit if it equals 0. MOSI_wr: Checks going to WRITE state on receiving 2 MOSI bits equal 0 following SS_n falling edge. Property (Outputs): rx: Checks the value of rx_data and rx_valid after complete data transmission. rx_wr: Checks that 1st 2 bits of rx_data equal 00 or 01 when the slave is in WRITE state.
Slave transitioning from one sequence to the other.	SS_n should equal 0 throughout the cycle -> 1 after data is collected. Sequences: (1): MOSI should equal 0 (WRITE) -> 00 (Send Address) rx_data should be ready and rx_valid should be high after 8 cycles. (2): MOSI should equal 0 (WRITE) -> 01 (Send Data) rx_data should be ready and rx_valid should be high after 8 cycles. (3): MOSI should equal 1 (READ) -> 10 (Send Address) rx_data should be ready and rx_valid should be high after 8 cycles. (4): MOSI should equal 1 (READ) -> 11 (Read Data) Dummy data are sent and rx_valid should be high after 8 cycles, then tx_valid becomes high and MISO reads tx_data in 8 cycles. Slave's behavior should not be affected by the order of different sequences. The slave can't go through two consecutive (READ_ADD) sequences or (READ_DATA) sequences.	rst_n = 1. SS_n is directed to be activated at the beginning of each sequence and deactivated following data transmission. MOSI bits are randomized under no constraints.	Included in Coverpoints: cs_cp.States: The slave has gone through write states. cs_cp.rd_add_data: The slave has transitioned from READ_ADD state to READ_DATA state. cs_cp.wr_add_data: The slave has transitioned from write state to another write (address then data). cs_cp.(rd_to_wr/wr_to_rd/rd_wr_rd/wr_rd_wr/wr_rd2_wr) checking for random order of sequences.	Output is checked against golden model.	Property (States): CHK_WR: Checks going from IDLE to CHK_WR state where it waits for the 1st MOSI bit if it equals 0. CHK_RD: Checks going from IDLE to CHK_RD state where it waits for the 1st MOSI bit if it equals 1. MOSI_wr: Checks going to WRITE state on receiving 2 MOSI bits equal 0 following SS_n falling edge. MOSI_rd: Checks going to WAIT_RD2 state on receiving 2 MOSI bits equal 1 following SS_n falling edge. MOSI_rd_add: Checks going to READ_ADD state on receiving 2 MOSI bits equal 1 then 1 MOSI bit equal 0 following SS_n falling edge. MOSI_rd_data: Checks going to READ_DATA state on receiving 3 MOSI bits equal 1 following SS_n falling edge. no_add_add/no_data_data: Checks that slave does not go through two consecutive (READ_ADD) sequences or (READ_DATA) sequences. Property (Outputs): rx: Checks the value of rx_data and rx_valid after complete data transmission. rx_wr: Checks that 1st 2 bits of rx_data equal 00 or 01 when the slave is in WRITE state. rx_rd_add: Checks that 1st 2 bits of rx_data equal 10 when the slave is in READ_ADD state. rx_rd_data: Checks that 1st 2 bits of rx_data equal 11 when the slave is in READ_DATA state. rx_tx: rx_valid cannot be high if tx_valid is high. rx_chk: rx_valid cannot be high at any checking state. tx: Checks MISO output following tx_valid rising edge. MISO_rd_only: MISO cannot be high at any state other than READ_DATA.
Normal Operation of Slave (Constrained)	rst_n activated: Outputs reset to zero. SS_n deactivated mid-transmission: slave goes IDLE. SS_n deactivated after-transmission: slave should stay stable. MOSI bits control the flow of the slave. rx_data is ready and rx_valid is high 8 cycles into Write/Read state. MISO bits reads tx_data across 8 cycles when tx_valid is high (which starts after 8 cycles into READ_DATA state).	Randomization of all variables under constraints: - rst_n is inactive most of the time - SS_n is active most of the time - tx_valid is inactive most of the time - tx_valid is high if cs = READ_DATA and rx_valid was high the previous cycle (to mimic RAM behavior)	Included in Coverpoints: cs_cp: The slave has gone through all states and transitions. tx_data_cp: tx_data has taken corner values. Included in Cross Coverage: cross_tx_read: All corners of read data came along read states. cross_tx_valid_data: All corners of read data came along active tx_valid. cross_MOSI_SS_rst: All combinations of Control Signals occured.	Output is checked against golden model.	

Verification Requirements

Test Item	Description and Expected Output Behavior	Stimulus Generation
Reset Signal (rst_n)	Slave outputs (rx_data, rx_valid, MISO) should reset to zero on reset assertion.	rst_n = 0. All other inputs are randomized under no constraints.
Control Signal (SS_n)	Slave should stay Idle if SS_n is inactive or transition to other states (based on MOSI bits) if SS_n is active. Slave should stay stable if SS_n remained active long after data transmission. Slave should go idle if SS_n is deactivated mid data transmission.	Included in all tasks of stimulus generation where SS_n is either active or randomized to be active most of the time.
Normal Write Operation of Slave	SS_n should equal 0 throughout the cycle -> 1 after data is collected. Cycle(1): MOSI should equal 0 (WRITE) -> 00 (Send Address) rx_data should be ready and rx_valid should be high after 8 cycles. Cycle(2): MOSI should equal 0 (WRITE) -> 01 (Send Data) rx_data should be ready and rx_valid should be high after 8 cycles.	rst_n = 1. SS_n is directed to be activated at the beginning of each sequence and deactivated following data transmission. MOSI bits are randomized under no constraints.
Slave transitioning from one sequence to the other.	SS_n should equal 0 throughout the cycle -> 1 after data is collected. Sequences: (1): MOSI should equal 0 (WRITE) -> 00 (Send Address) rx_data should be ready and rx_valid should be high after 8 cycles. (2): MOSI should equal 0 (WRITE) -> 01 (Send Data) rx_data should be ready and rx_valid should be high after 8 cycles. (3): MOSI should equal 1 (READ) -> 10 (Send Address) rx_data should be ready and rx_valid should be high after 8 cycles. (4): MOSI should equal 1 (READ) -> 11 (Read Data) Dummy data are sent and rx_valid should be high after 8 cycles, then tx_valid becomes high and MISO reads tx_data in 8 cycles. Slave's behavior should not be affected by the order of different sequences. The slave can't go through two consecutive (READ_ADD) sequences or (READ_DATA) sequences.	rst_n = 1. SS_n is directed to be activated at the beginning of each sequence and deactivated following data transmission. MOSI bits are randomized under no constraints.
Normal Operation of Slave (Constrained)	rst_n activated: Outputs reset to zero. SS_n deactivated mid-transmission: slave goes IDLE. SS_n deactivated after-transmission: slave should stay stable. MOSI bits control the flow of the slave. rx_data is ready and rx_valid is high 8 cycles into Write/Read state. MISO bits reads tx_data across 8 cycles when tx_valid is high (which starts after 8 cycles into READ_DATA state).	Randomization of all variables under constraints: - rst_n is inactive most of the time - SS_n is active most of the time - tx_valid is inactive most of the time - tx_valid is high if cs = READ_DATA and rx_valid was high the previous cycle (to mimic RAM behavior)

Verification Requirements

Test Item	Functional Coverage	Functionality Check	Assertions
Reset Signal (rst_n)	Included in Cross Coverage: cross_MOSI_SS_rst: All combinations of control signals are covered. cross_rst_states: All States have experienced the effect of rst_n.	Outputs must equal zero. Checked in check_reset task.	Property: reset_asserted
Control Signal (SS_n)	Included in Cross Coverage: cross_MOSI_SS_rst: All combinations of control signals are covered. cross_SS_states: All States have experienced the effect of SS_n.	Output is checked against golden model.	Property: SS_inactive
Normal Write Operation of Slave	Included in Coverpoints: cs_cp.States: The slave has gone through write states. cs_cp.wr_add_data: The slave has transitioned from write state to another write (address then data).	Output is checked against golden model.	Property (States): CHK_WR: Checks going from IDLE to CHK_WR state where it waits for the 1st MOSI bit if it equals 0. MOSI_wr: Checks going to WRITE state on receiving 2 MOSI bits equal 0 following SS_n falling edge. Property (Outputs): rx: Checks the value of rx_data and rx_valid after complete data transmission. rx_wr: Checks that 1st 2 bits of rx_data equal 00 or 01 when the slave is in WRITE state.
Slave transitioning from one sequence to the other.	Included in Coverpoints: cs_cp.States: The slave has gone through write states. cs_cp.rd_add_data: The slave has transitioned from READ_ADD state to READ_DATA state. cs_cp.wr_add_data: The slave has transitioned from write state to another write (address then data). cs_cp.(rd_to_wr/wr_to_rd/rd_wr_rd/wr_rd_wr/wr_rd2_wr) checking for random order of sequences.	Output is checked against golden model.	Property (States): CHK_WR: Checks going from IDLE to CHK_WR state where it waits for the 1st MOSI bit if it equals 0. CHK_RD: Checks going from IDLE to CHK_RD state where it waits for the 1st MOSI bit if it equals 1. MOSI_wr: Checks going to WRITE state on receiving 2 MOSI bits equal 0 following SS_n falling edge. MOSI_rd: Checks going to WAIT_RD2 state on receiving 2 MOSI bits equal 1 following SS_n falling edge. MOSI_rd_add: Checks going to READ_ADD state on receiving 2 MOSI bits equal 1 then 1 MOSI bit equal 0 following SS_n falling edge. MOSI_rd_data: Checks going to READ_DATA state on receiving 3 MOSI bits equal 1 following SS_n falling edge. no_add_add/no_data_data: Checks that slave does not go through two consecutive (READ_ADD) sequences or (READ_DATA) sequences. Property (Outputs): rx: Checks the value of rx_data and rx_valid after complete data transmission. rx_wr: Checks that 1st 2 bits of rx_data equal 00 or 01 when the slave is in WRITE state. rx_rd_add: Checks that 1st 2 bits of rx_data equal 10 when the slave is in READ_ADD state. rx_rd_data: Checks that 1st 2 bits of rx_data equal 11 when the slave is in READ_DATA state. rx_tx: rx_valid cannot be high if tx_valid is high. rx_chk: rx_valid cannot be high at any checking state. tx: Checks MISO output following tx_valid rising edge. MISO_rd_only: MISO cannot be high at any state other than READ_DATA.
Normal Operation of Slave (Constrained)	Included in Coverpoints: cs_cp: The slave has gone through all states and transitions. tx_data_cp: tx_data has taken corner values. Included in Cross Coverage: cross_tx_read: All corners of read data came along read states. cross_tx_valid_data: All corners of read data came along active tx_valid. cross_MOSI_SS_rst: All combinations of Control Signals occured.	Output is checked against golden model.	

Bug Report

Bug	Original Code	Fix	Lines (Original Version)	Lines (Edited Version)
Parameter (ADDR_SIZE) should be instead of the actual size to respond correctly to changes in parameters				

Next State Logic
 	
When First MOSI bit = 0, followed by 1, slave enters WRITE state and sends a read signal to RAM.
 	
When First MOSI bit = 1, followed by 0, slave enters READ_ADD/READ_DATA states and sends a write signal to RAM.
 	
When previous first 3 MOSI bits equaled 110 directing slave to READ_ADD state. After next SS_n falling edge MOSI bits equaled 110 again, entering READ_DATA state but sends read address signal to RAM.
	READ_DATA:
	if(~SS_n &&(start_to_take || start_to_give)) begin
	ns <= READ_DATA ;
	end
	else begin
	ns <= IDLE ;
	end
	CHK_CMD:
	if((~SS_n) && (MOSI == 1) && rd_addr_received) begin
	ns <= READ_DATA ;
	end
	else if((~SS_n) && (MOSI == 1)) begin
	ns <= READ_ADD ;
	end
	else if ((~SS_n) && (MOSI == 0)) begin
	ns <= WRITE ;
	end
	else if (SS_n) begin
	ns <= IDLE ;
	end
	WRITE:
	if(~SS_n && start_to_give) begin
	ns <= WRITE ;
	end
	else begin
	ns <= IDLE ;
	end
		READ_DATA:
		if(SS_n) begin
		ns <= IDLE;
		end
		else if(start_to_take) begin
		ns <= READ_DATA ;
		end
		CHK_CMD:
		if(SS_n) ns <= IDLE;
		else begin
		if(MOSI) ns <= WAIT_RD;
		else ns <= WAIT_WR;
		end
		WAIT_WR:
		if(SS_n || MOSI) ns <= IDLE;
		else ns <= WRITE;
		WAIT_RD:
		if(SS_n || ~MOSI) ns <= IDLE;
		else ns <= WAIT_RD2;
		WAIT_RD2:
		if(SS_n) ns <= IDLE;
		else begin
		if(rd_addr_received && MOSI) ns <= READ_DATA;
		else if(~rd_addr_received && ~MOSI) ns <= READ_ADD;
		else ns <= IDLE;
		end
			[42:88]	[152:204]

rx_data (after editing next state logic) does not have access to the first 2 MOSI bits, as it starts reading when in WRITE/READ_ADD states	always@(posedge clk) begin
	if (start_to_give ==1 && ~SS_n) begin
	rx_data <= {rx_data[ADDR_SIZE:0],MOSI};
	if (i==ADDR_SIZE+1) begin
	i<=0;
	rx_valid =1;
	start_to_give <= 0;
	end
	else begin
	i <= i + 1 ;
	rx_valid <= 0;
	end
	end
	else begin
	rx_valid <=0;
	if((cs == CHK_CMD) && (SS_n == 0))
	start_to_give <= 1;
	end
	end
		if (start_to_give) begin
		rx_temp <= {rx_temp[ADDR_SIZE-1:0],MOSI};
		if (i==ADDR_SIZE) begin
		i<=0;
		start_to_give <= 0;
		rx_valid <= 1;
		end
		else begin
		i <= i + 1 ;
		rx_valid <= 0;
		end
		end
		else begin
		rx_valid <= 0;
		if(((cs == WAIT_WR) && ~MOSI) || ((cs == WAIT_RD) && MOSI)) begin
		start_to_give <= 1;
		i <= 0;
		end
		end
			[97:115]	27
				[231:249]
				293

Bug Report

Bug	Original Code	Fix	Lines (Original Version)	Lines (Edited Version)
Parameter (ADDR_SIZE) should be instead of the actual size to respond correctly to changes in parameters				

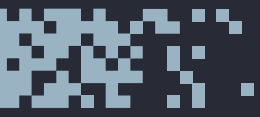
Outputs do not reset to zero on reset assertion		always @(posedge clk or negedge rst_n)
		begin
		if(cs == IDLE) begin
		rx_temp <= 0;
		rx_valid <= 0;
		MISO <= 0;
		start_to_give <= 0;
		start_to_take <= 0;
		temp <= 0;
		end
		end
		
		always@(posedge clk or negedge rst_n)
		begin
		if(~rst_n || SS_n) begin
		rx_temp <= 0;
		rx_valid <= 0;
		i <= 0;
		end
		
		always@ (posedge tx_valid or negedge rst_n)begin
		if(~rst_n) begin
		temp <= 0;
		start_to_take <=0;
		rd_addr_received <= 0;
		end
				[206:215]
				[224:229]
				[253:258]
				[266:271]

MISO bits are reversed when reading tx_data
Last bit in tx_data is not read
	always@(start_to_take,posedge clk) begin
	if (start_to_take==1 && ~SS_n) begin
	MISO <= temp[0];
	temp <= {1'b0,temp[ADDR_SIZE-1:1]};
	if (j == ADDR_SIZE-1) begin
	start_to_take <= 0 ;
	j <= 0;
	end
	else begin
	j <= j + 1 ;
	end
	end
	end
		if (tx_valid && start_to_take) begin
		MISO <= temp[ADDR_SIZE-1];
		temp <= {temp[ADDR_SIZE-2:0], 1'b0};
		if (j == ADDR_SIZE) begin
		start_to_take <= 0 ;
		j <= 0;
		end
		else begin
		j <= j + 1 ;
		end
		end
		else begin
		MISO <= 0;
		start_to_take <= 0;
		j <= 0;
		end
			[124:136]	[273:288]

closer look

```
always @(posedge clk or negedge rst_n) begin
 if(cs == IDLE) begin
  rx_temp <= 0;
  rx_valid <= 0;
  MISO <= 0;
  start_to_give <= 0;
  start_to_take <= 0;
  temp <= 0;
 end
end

always@(posedge clk or negedge rst_n) begin
 if(~rst_n || SS_n) begin
  rx_temp <= 0;
  rx_valid <= 0;
  i <= 0;
 end

always@ (posedge tx_valid or negedge rst_n)begin
 if(~rst_n) begin
  temp <= 0;
  start_to_take <=0;
  rd_addr_received <= 0;
 end

always@(posedge clk or negedge rst_n) begin
 if(~rst_n || SS_n) begin
  MISO <= 0;
  start_to_take <= 0;
  j <= 0;
 end
```

CODE SNIPPETS



DESIGN

Before (Original)

```
30  ////////////////////////////////Original Code//////////////////////////////////////
31  /*
32
33      always@(posedge clk or negedge rst_n)   begin
34          if(~rst_n)  begin
35              cs <= IDLE ;
36          end
37          else    begin
38              cs <= ns ;
39          end
40      end
41      //Next state logic
42      always@(cs,SS_n,MOSI)   begin
43          case(cs)
44              IDLE:
45                  if(SS_n)        begin
46                      ns <= IDLE ;
47                  end
48                  else    begin
49                      ns <= CHK_CMD ;
50                  end
51              READ_DATA:
52                  if(~SS_n &&( start_to_take || start_to_give ))  begin
53                      ns <= READ_DATA ;
54                  end
55                  else    begin
56                      ns <= IDLE ;
57                  end
58              READ_ADD:
59                  if(~SS_n && start_to_give)  begin
60                      ns <= READ_ADD ;
61                  end
62                  else    begin
63                      ns <= IDLE ;
64                  end
65              CHK_CMD:
66                  if( (~SS_n) && (MOSI == 1) && rd_addr_received )    begin
67                      ns <= READ_DATA ;
68                  end
69                  else if( (~SS_n) && (MOSI == 1) )   begin
70                      ns <= READ_ADD ;
71                  end
72                  else if ( (~SS_n) && (MOSI == 0) )  begin
73                      ns <= WRITE ;
74                  end
75                  else if (SS_n)  begin
76                      ns <= IDLE ;
77                  end
```

Code Snippets

DESIGN

Before (Original)

```
78              WRITE:
79                  if(~SS_n && start_to_give)  begin
80                      ns <= WRITE ;
81                  end
82                  else    begin
83                      ns <= IDLE ;
84                  end
85
86              default: ns <= IDLE;
87          endcase
88      end
89
90      always @(posedge clk) begin
91          if (cs == READ_ADD)
92              rd_addr_received=1;
93          else if (cs == READ_DATA)
94              rd_addr_received=0;
95      end
96
97      always@(posedge clk)    begin
98          if (start_to_give ==1 && ~SS_n) begin
99              rx_data <= {rx_data[ADDR_SIZE:0],MOSI};     //param
100             if (i==ADDR_SIZE+1) begin                   //param
101                 i<=0;
102                 rx_valid =1;
103                 start_to_give <= 0;
104             end
105             else  begin
106                 i <= i + 1 ;
107                 rx_valid <= 0;
108             end
109         end
110         else begin
111             rx_valid <=0;
112             if((cs == CHK_CMD) && (SS_n == 0))
113                 start_to_give <= 1;
114         end
115     end
```

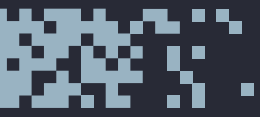
Code Snippets

DESIGN

Before (Original)

```
117
118
119         always@ (posedge tx_valid)begin
120             start_to_take <=1;
121             temp <= tx_data;
122         end
123
124         always@(start_to_take,posedge clk)  begin
125             if (start_to_take==1 && ~SS_n) begin
126                 MISO <= temp[0];
127                 temp <= {1'b0,temp[ADDR_SIZE-1:1]};         //param
128                 if (j == ADDR_SIZE-1)   begin               //param
129                     start_to_take <= 0 ;
130                     j <= 0;
131                 end
132                 else begin
133                 j <= j + 1 ;
134                 end
135             end
136         end
137     */
138     ////////////////////////////////////////////////////////////////////////////////
139
```

CODE SNIPPETS

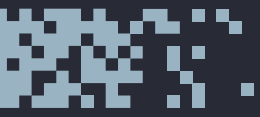


DESIGN

After (Edited)

```
141  ////////////////////////////////Edited Code//////////////////////////////////////
142
143      always@(posedge clk or negedge rst_n)   begin
144          if(~rst_n)  begin
145              cs <= IDLE ;
146          end
147          else    begin
148              cs <= ns ;
149          end
150      end
151      //Next state logic
152      always@(cs,SS_n,MOSI)   begin
153          case(cs)
154              IDLE:
155                  if(SS_n)        begin
156                      ns <= IDLE ;
157                  end
158                  else    begin
159                      ns <= CHK_CMD ;
160                  end
161              READ_DATA:
162                  if(SS_n)    begin
163                      ns <= IDLE;
164                  end
165                  else if( start_to_take )    begin
166                      ns <= READ_DATA ;
167                  end
168              READ_ADD:
169                  if(~SS_n && start_to_give)  begin
170                      ns <= READ_ADD ;
171                  end
172                  else if(SS_n)   begin
173                      ns <= IDLE ;
174                  end
175              CHK_CMD:
176                  if(SS_n) ns <= IDLE;
177                  else begin
178                      if(MOSI) ns <= WAIT_RD;
179                      else ns <= WAIT_WR;
180                  end
181              WAIT_WR:
182                  if(SS_n || MOSI) ns <= IDLE;
183                  else ns <= WRITE;
184              WAIT_RD:
185                  if(SS_n || ~MOSI) ns <= IDLE;
186                  else ns <= WAIT_RD2;
```

CODE SNIPPETS



DESIGN

After (Edited)

```
187                 WAIT_RD2:
188                     if(SS_n) ns <= IDLE;
189                     else begin
190                         if(rd_addr_received && MOSI) ns <= READ_DATA;
191                         else if(~rd_addr_received && ~MOSI) ns <= READ_ADD;
192                         else ns <= IDLE;
193                     end
194                 WRITE:
195                     if(~SS_n && start_to_give)  begin
196                         ns <= WRITE ;
197                     end
198                     else if(SS_n)   begin
199                         ns <= IDLE ;
200                     end
201 
202                 default: ns <= IDLE;
203             endcase
204         end
205 
206         always @(posedge clk or negedge rst_n) begin
207             if(cs == IDLE) begin
208                 rx_temp <= 0;
209                 rx_valid <= 0;
210                 MISO <= 0;
211                 start_to_give <= 0;
212                 start_to_take <= 0;
213                 temp <= 0;
214             end
215         end
216 
217         always @(posedge clk) begin
218             if (cs == READ_ADD)
219                 rd_addr_received=1;
220             else if (cs == READ_DATA)
221                 rd_addr_received=0;
222         end
```

CODE SNIPPETS

DESIGN

After (Edited)

```
224     always@(posedge clk or negedge rst_n) begin
225         if(~rst_n || SS_n) begin
226             rx_temp <= 0;
227             rx_valid <= 0;
228             i <= 0;
229         end
230         else begin
231             if (start_to_give)  begin
232                 rx_temp <= {rx_temp[ADDR_SIZE-1:0],MOSI};   //param
233                 if (i==ADDR_SIZE) begin                     //param
234                     i<=0;
235                     start_to_give <= 0;
236                     rx_valid <= 1;
237                 end
238                 else  begin
239                     i <= i + 1 ;
240                     rx_valid <= 0;
241                 end
242             end
243             else begin
244                 rx_valid <= 0;
245                 if( ((cs == WAIT_WR) && ~MOSI) || ((cs == WAIT_RD) && MOSI) ) begin
246                     start_to_give <= 1;
247                     i <= 0;
248                 end
249             end
250         end
251     end
252
253     always@ (posedge tx_valid or negedge rst_n)begin
254         if(~rst_n) begin
255             temp <= 0;
256             start_to_take <=0;
257             rd_addr_received <= 0;
258         end
259         else if(cs == READ_DATA) begin
260             start_to_take <=1;
261             temp <= tx_data;
262         end
263         else start_to_take <= 0;
264     end
```

CODE SNIPPETS

DESIGN

After (Edited)

```
266     always@(posedge clk or negedge rst_n)   begin
267         if(~rst_n || SS_n) begin
268             MISO <= 0;
269             start_to_take <= 0;
270             j <= 0;
271         end
272         else begin
273             if (tx_valid && start_to_take) begin
274                 MISO <= temp[ADDR_SIZE-1];
275                 temp <= {temp[ADDR_SIZE-2:0], 1'b0};     //param
276                 if (j == ADDR_SIZE) begin                //param
277                     start_to_take <= 0 ;
278                     j <= 0;
279                 end
280                 else begin
281                     j <= j + 1 ;
282                 end
283             end
284             else begin
285                 MISO <= 0;
286                 start_to_take <= 0;
287                 j <= 0;
288             end
289         end
290     end
291
292
293     assign rx_data = ( (cs == READ_ADD) || (cs == READ_DATA) ) ? {1'b1, rx_temp} : {1'b0, rx_temp};
294
295 ////////////////////////////////////////////////////////////////////////////////
```

CODE SNIPPETS

PACKAGE

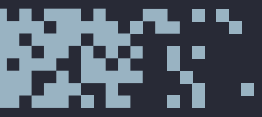
Class Properties and Pre-defined Variables

```
package slave_pkg;

    typedef enum logic [2:0] {IDLE, READ_DATA, READ_ADD, CHK_CMD, WRITE, WAIT_WR, WAIT_RD, WAIT_RD2} state_dut_e;
    typedef enum logic [2:0] {IDLE_, CHK_CMD_, WRITE_, READ_ADD_, READ_DATA_, WAIT_WR_, WAIT_RD_, WAIT_RD2_} state_gm_e;


    class slave_class #(int ADDR_SIZE = 8);

        //Parameters for Constraints and Covergroup
        parameter ALL_ONES = 2*(ADDR_SIZE) - 1;
        parameter ZERO = 0;

        //Inputs to DUT
        bit clk;
        rand bit MOSI, SS_n, rst_n, tx_valid;
        rand logic [ADDR_SIZE-1:0] tx_data;

        //Signals Imported from Design
        bit rx_valid;
        logic [ADDR_SIZE+1:0] rx_data, rx_data_old;
        state_dut_e cs;
        //int rx_high_since = 0;

        //For tx_data Constraint
        rand bit [1:0] selector_tx;

        //For Randomizing Sequences
        rand bit [1:0] selector_seq;
```

Constraints

```
        constraint rst_c {
            rst_n           dist {0:=5,     1:=95};                     //rst_n is inactive most of the time
        }

        constraint SS_c {
            SS_n            dist {0:=90,    1:=10};                     //SS_n is active most of the time
        }

        constraint tx {
            tx_valid        dist {0:=70,    1:=30};                     //tx_valid is inactive most of the time
            selector_tx     dist {0:=30,    1:=30,      [2:3]:=40};

            if(selector_tx == 0) tx_data inside {ALL_ONES, ZERO};
            if(selector_tx == 1) $countones(tx_data) == 1;
        }

```

CODE SNIPPETS

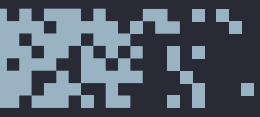


PACKAGE

Covergroup

```
63      covergroup slave_cg;
64
65          ////////////////////////////////////////////////////////////////Coverpoints//////////////////////////////////////////////////////////////////
66
67          tx_data_cp:     coverpoint tx_data{                     //Rapid changes in tx_data might not be caught by MISO
68              bins Corners    = {ZERO, ALL_ONES, 8'haa, 8'h55};   //Needs to be edited if a different ADDER_SIZE is used
69              bins low_high   = (ZERO => ALL_ONES);
70              bins high_low   = (ALL_ONES => ZERO);
71          }
72
73          cs_cp:          coverpoint cs{                          //All states, and transitions must be covered
74              //States
75              bins States[]       = {IDLE, READ_DATA, READ_ADD, CHK_CMD, WRITE, WAIT_WR, WAIT_RD, WAIT_RD2};
76              /*
77              bins Ctrl_States    = {IDLE, CHK_CMD};
78              bins Write_States   = {WRITE};
79              bins Read_States    = {READ_DATA, READ_ADD};
80              bins hold_States    = {WAIT_WR, WAIT_RD, WAIT_RD2};
81              */
82
83              //Normal Transitions
84              bins chk_to_all         = (CHK_CMD => IDLE, WAIT_WR, WAIT_RD);
85              bins rd_add_data        = (READ_ADD => IDLE => CHK_CMD => WAIT_RD => WAIT_RD2 => READ_DATA);
86              bins wr_add_data        = (WRITE => IDLE => CHK_CMD => WAIT_WR => WRITE);
87
88              //Illegal Transitions
89              illegal_bins IDLE_to_other      = (IDLE => READ_DATA, READ_ADD, WRITE, WAIT_WR, WAIT_RD, WAIT_RD2);
90              illegal_bins CHK_to_other       = (CHK_CMD => WAIT_RD2, READ_DATA, READ_ADD, WRITE);
91              illegal_bins WAIT_RD_to_other   = (WAIT_RD => READ_DATA, READ_ADD, CHK_CMD, WRITE, WAIT_WR);
92
93              //Different Scenarios
94              bins rd_to_wr[]     = (READ_ADD, READ_DATA => IDLE => CHK_CMD => WAIT_WR => WRITE);
95              bins wr_to_rd[]     = (WRITE => IDLE => CHK_CMD => WAIT_RD => WAIT_RD2 => READ_ADD, READ_DATA);
96              bins rd_wr_rd       = (READ_ADD => IDLE => CHK_CMD => WAIT_WR => WRITE[*10] => IDLE => CHK_CMD => WAIT_RD => WAIT_RD2 => READ_DATA);
97              bins wr_rd_wr[]     = (WRITE => IDLE => CHK_CMD => WAIT_RD => WAIT_RD2 => READ_ADD[*9] => IDLE => CHK_CMD => WAIT_WR => WRITE);
98              bins wr_rd2_wr[]    = (WRITE => IDLE => CHK_CMD => WAIT_RD => WAIT_RD2 => READ_DATA[*18] => IDLE => CHK_CMD => WAIT_WR => WRITE);
99          }
100
101         rst_n_cp:       coverpoint rst_n;
102         SS_n_cp:        coverpoint SS_n;
103         MOSI_cp:        coverpoint MOSI;
104         tx_valid_cp:    coverpoint tx_valid;


106
107         ////////////////////////////////////////////////////////////////Cross Coverage//////////////////////////////////////////////////////////////////
108
109         cross_tx_read:          cross tx_data_cp, cs_cp{
110             bins tx_read            = binsof(tx_data_cp) && binsof(cs_cp.States) intersect {READ_ADD, READ_DATA};   //All corners of read data came along Read States
111             ignore_bins tx_no_read1 = binsof(tx_data_cp) && !binsof(cs_cp.States);
112             ignore_bins tx_no_read2 = binsof(tx_data_cp) && binsof(cs_cp.States) intersect {IDLE, CHK_CMD, WRITE, WAIT_WR, WAIT_RD, WAIT_RD2};
113         }
114
115         cross_tx_valid_data:    cross tx_data_cp, tx_valid_cp{                                                  //All corners of read data came along active tx_valid
116             bins tx_val_data        = binsof(tx_data_cp) && binsof(tx_valid_cp) intersect {1};
117             ignore_bins tx_inv      = binsof(tx_data_cp) && binsof(tx_valid_cp) intersect {0};
118         }
119
120         cross_MOSI_SS_rst:          cross rst_n_cp, SS_n_cp, MOSI_cp;                                           //All combinations of Control Signals occured
121
122         cross_rst_states:       cross cs_cp, rst_n_cp{
123             bins rst_all            = binsof(cs_cp.States) && binsof(rst_n_cp);                                 //All states experienced rst_n effect (low/high)
124             ignore_bins rst_trans   = !binsof(cs_cp.States) && binsof(rst_n_cp);
125         }
126
127         cross_SS_states:        cross cs_cp, SS_n_cp{
128             bins SS_all             = binsof(cs_cp.States) && binsof(SS_n_cp);                                  //All states experienced SS_n effect (low/high)
129             ignore_bins SS_trans    = !binsof(cs_cp.States) && binsof(SS_n_cp);
130         }
131     endgroup
```

CODE SNIPPETS



TESTBENCH

Stimulus Generation

```
61
62      initial begin
63
64          //Assert Reset - Initial State
65          assert_reset;
66
67
68
69
70          /*TEST 0:    - Checks (rst_n) Functionality
71                       - slave outputs (rx_data, rx_valid, MISO) reset to zero @(rst_n = 0)
72                       - Randomization (of all variables) is done Under No Constraints
73          */
74          stimulus_gen_reset;
75
76
77
78
79          //Deassert Reset
80          deassert_reset;
81
82
83
84
85
86
87
88
89          /*TEST 1:    - Checks Normal Write Operation of Slave
90                       - SS_n should equal 0 throughout the cycle -> 1 after data is collected
91                       - Cycle(1): MOSI should equal 0 (WRITE) -> 00 (Send Address)
92                       - rx_data should be ready and rx_valid should be high after 8 cycles
93                       - Cycle(2): MOSI should equal 0 (WRITE) -> 01 (Send Data)
94                       - rx_data should be ready and rx_valid should be high after 8 cycles
95                       - Randomization (of MOSI) is done Under No Constraints
96          */
97          stimulus_gen1;
98
99
100
101
102
103         /*TEST 2:    - Checks Normal Read Operation of Slave
104                      - SS_n should equal 0 throughout the 1st cycle -> 1 after data is collected
105                      - Cycle(1): MOSI should equal 1 (READ) -> 10 (Send Address)
106                      - Cycle(2): MOSI should equal 1 (READ) -> 11 (Receive Data)
107                      - MISO should translate tx_data to serial output when tx_valid is high
108                      - Randomization (of MOSI) is done Under No Constraints
109         */
110         stimulus_gen2;
111
```

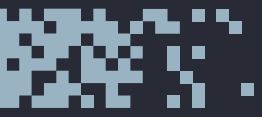
CODE SNIPPETS

TESTBENCH

Stimulus Generation

```
116         /*TEST 3:    - Checks slave's behavior going from one sequence to another
117                       - Same functionality as Normal Operation
118                       - The slave can't go through two consecutive (READ_ADD) sequences or (READ_DATA) sequences
119                       - Randomization (of MOSI) is done Under No Constraints
120         */
121         stimulus_gen3;
122
123
124
125
126
127         /*TEST 4:    - Randomization of all variables under constraints:
128                         - rst_n is inactive most of the time
129                         - SS_n is active most of the time
130                         - tx_valid is inactive most of the time
131                         - tx_valid is high if cs = READ_DATA and rx_valid was high the previous cycle (to mimic RAM behavior)
132                       - SS_n deactivated mid-transmission -> slave goes IDLE, outputs reset to zero
133                       - SS_n deactivated after-transmission -> slave should stay stable
134         */
135         stimulus_gen4;
136
137
138
139
140         /*TEST 5:    - Randomization of all variables under constraints
141                       - Output is checked against golden model
142         */
143         stimulus_gen5;
144
145
146
147
148         //Correct Count and Error Count Display
149         $display("At End of Simulation: Correct Count = %0d, Error Count = %0d", correct_count, error_count);
150         @(negedge clk); $stop;
151
152     end
```

CODE SNIPPETS



TESTBENCH

Tasks

```
162     ////////////////////////////Reset-Related//////////////////////////////
163
164     //Assert Reset
165     task assert_reset;
166         rst_n = 0;
167         sc.constraint_mode(0);
168
169         @(negedge clk);
170     endtask
171
172     //TEST 0: Reset Asserted
173     task stimulus_gen_reset;
174         for(int i=0; i<TESTS; i++) begin
175             assert(sc.randomize());
176             SS_n        = sc.SS_n;
177             MOSI        = sc.MOSI;
178             tx_valid    = sc.tx_valid;
179             tx_data     = sc.tx_data;
180
181             check_reset;
182             @(negedge clk);
183         end
184     endtask
185
186     //CHECKER: Reset Asserted
187     task check_reset;
188         @(posedge clk);
189
190         if(rx_data !== 0) begin
191             $display("ERROR: (Reset Asserted) -> Output -rx_data- equals %0h, but should equal 0 \t\t--time: %0t", rx_data, $time);
192             error_count++;
193         end
194         else correct_count++;
195
196         if(rx_valid !== 0) begin
197             $display("ERROR: (Reset Asserted) -> Output -rx_valid- equals %0h, but should equal 0 \t\t--time: %0t", rx_valid, $time);
198             error_count++;
199         end
200         else correct_count++;
201
202         if(MISO !== 0) begin
203             $display("ERROR: (Reset Asserted) -> Output -MISO- equals %0h, but should equal 0 \t\t--time: %0t", MISO, $time);
204             error_count++;
205         end
206         else correct_count++;
207     endtask
208
209     //Deassert Reset
210     task deassert_reset;
211         rst_n = 1;
212         SS_n = 1;
213         sc.constraint_mode(1);
214
215         @(negedge clk);
216     endtask
```

CODE SNIPPETS

TESTBENCH

Tasks

```
224     //////////////////////Preparing Possible Sequences for Slave (IDLE to IDLE)//////////////////////
225
226
227     /*
228     To simulate RAM's behavior  - Inputs are driven @(posedge clk)
229                                 - tx_valid and tx_data do not change until rx_valid is high
230     */
231     int rx_high_since = 0;
232     always @(posedge clk) begin
233         assert(sc.randomize());
234         if(cs_DUT == READ_DATA) begin
235             rx_high_since++;
236             if(rx_high_since > ADDR_SIZE - 1) tx_valid <= 1;
237             else begin
238                 tx_valid <= 0;
239                 tx_data <= sc.tx_data;
240             end
241         end
242         else begin
243             if(ns_DUT == READ_DATA) tx_valid <= 0;
244             else tx_valid <= sc.tx_valid;
245             rx_high_since <= 0;
246             tx_data <= sc.tx_data;
247         end
248     end
249
250     //Marks the beginning and ending of all sequences
251     task start_seq;
252         SS_n = 0;
253         @(negedge clk);
254     endtask
255
256     task end_seq;
257         @(negedge clk);
258         SS_n = 1;
259         @(negedge clk);
260     endtask
261
262
263
264
265     //WRITE_ADD -> MOSI: 000
266     task write_add_seq;
267         start_seq;
268         MOSI = 0;
269         repeat(2) @(negedge clk);
270         for(int i=0; i<ADDR_SIZE; i++) begin
271             @(negedge clk);
272             assert(sc.randomize());
273             MOSI = sc.MOSI;
274         end
275         end_seq;
276     endtask
```

Code Snippets

TESTBENCH

Tasks

```
278     //WRITE_DATA -> MOSI: 001
279     task write_data_seq;
280         start_seq;
281         MOSI = 0;
282         repeat(2) @(negedge clk);
283         MOSI = 1;
284         for(int i=0; i<ADDR_SIZE; i++) begin
285             @(negedge clk);
286             assert(sc.randomize());
287             MOSI = sc.MOSI;
288         end
289         end_seq;
290     endtask
291
292     //READ_ADD -> MOSI: 110
293     task read_add_seq;
294         start_seq;
295         MOSI = 1;
296         repeat(2) @(negedge clk);
297         MOSI = 0;
298         for(int i=0; i<ADDR_SIZE; i++) begin
299             @(negedge clk);
300             assert(sc.randomize());
301             MOSI = sc.MOSI;
302         end
303         end_seq;
304     endtask
305
306     //READ_DATA -> MOSI: 111
307     task read_data_seq;
308         start_seq;
309         MOSI = 1;
310         repeat(2) @(negedge clk);
311         for(int i=0; i<ADDR_SIZE; i++) begin
312             @(negedge clk);
313             assert(sc.randomize());
314             MOSI = sc.MOSI;
315         end
316         @(negedge clk);
317
318         repeat (9) @(negedge clk);
319
320         SS_n = 1;
321         @(negedge clk);
322     endtask
```

CODE SNIPPETS

TESTBENCH

Tasks

```
327     ////////////////////////////////Stimulus Generation////////////////////////////////
328
329
330     //TEST 1: Normal Write Operation
331     task stimulus_gen1;
332         sc.constraint_mode(0);
333         for(int i=0; i<TESTS; i++) begin
334             write_add_seq;
335             write_data_seq;
336         end
337     endtask
338
339
340
341     //TEST 2: Normal Read Operation
342     task stimulus_gen2;
343         sc.constraint_mode(0);
344         for(int i=0; i<TESTS; i++) begin
345             read_add_seq;
346             read_data_seq;
347         end
348     endtask
349
350
351
352     //TEST 3: Randomization of Different Scenarios
353     task stimulus_gen3;
354         sc.constraint_mode(0);
355         for(int i=0; i<TESTS; i++) begin
356             assert(sc.randomize());
357             case(sc.selector_seq)
358                 0: write_add_seq;
359                 1: write_data_seq;
360                 2: read_add_seq;
361                 3: read_data_seq;
362             endcase
363         end
364     endtask
365
```

Code Snippets

TESTBENCH

Tasks

```
386     //Test 4: Randomization of all variables - with Constraints
387     task stimulus_gen4;
388         sc.constraint_mode(1);
389         for(int i=0; i<TESTS; i++) begin
390             assert(sc.randomize());
391             rst_n    = sc.rst_n;
392             SS_n     = sc.SS_n;
393             MOSI     = sc.MOSI;
394 
395             @(negedge clk);
396         end
397     endtask
398 
399 
400     //Test 5: Randomization of all variables - No Constraints
401     task stimulus_gen5;
402         sc.constraint_mode(0);
403         for(int i=0; i<TESTS; i++) begin
404             assert(sc.randomize());
405             rst_n    = sc.rst_n;
406             SS_n     = sc.SS_n;
407             MOSI     = sc.MOSI;
408 
409             @(negedge clk);
410         end
411     endtask
```

CODE SNIPPETS

TESTBENCH

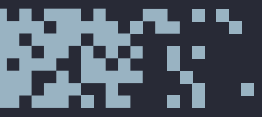
Checker

```
416     ////////////////////////////Checking Results////////////////////////////
417
418     //CHECKER:
419
420     always @(negedge clk or negedge rst_n) begin
421         if(~rst_n) check_reset;
422         else begin
423             check_rx_valid;
424             check_MISO;
425             if(rx_valid) check_rx_data;
426         end
427     end
428
429     task check_rx_valid;
430         if(rx_valid !== rx_valid_exp) begin
431             $display("ERROR: Output -rx_valid- equals %0h, but should equal %0h \t\t--time: %0t", rx_valid, rx_valid_exp, $time);
432             error_count++;
433         end
434         else correct_count++;
435     endtask
436
437     task check_rx_data;
438         if(rx_data !== rx_data_exp) begin
439             $display("ERROR: Output -rx_data- equals %0h, but should equal %0h \t\t--time: %0t", rx_data, rx_data_exp, $time);
440             error_count++;
441         end
442         else correct_count++;
443     endtask
444
445     task check_MISO;
446         if(MISO !== MISO_exp) begin
447             $display("ERROR: Output -MISO- equals %0h, but should equal %0h \t\t--time: %0t", MISO, MISO_exp, $time);
448             error_count++;
449         end
450         else correct_count++;
451     endtask
```

Assertions

```
22  ////////////////////////////////////////////Checking Stable Cases (IDLE)////////////////////////////////////////////
23
24      //Checks rst_n Functionality - Output Values and State Transitions
25      property reset_asserted;
26          @(posedge clk)
27
28          !rst_n |-> !(rx_valid || rx_data || MISO) && (cs_sva == IDLE);
29      endproperty
30
31      //Checks SS_n Functionality - State Transitions
32      property SS_inactive;
33          @(posedge clk) disable iff(!rst_n)
34
35          $rose(SS_n) |=> (cs_sva == IDLE);
36      endproperty
```

CODE SNIPPETS



TESTBENCH

Assertions

```
40  ////////////////////////////////////////////Checking Original Functionality////////////////////////////////////////////
41
42      ////////////////////////////////////////////Checking State Transitions
43
44      //Slave must always go from IDLE to CHK_CMD
45      property CHK_first;
46          @(posedge clk) disable iff(!rst_n)
47
48          $fell(SS_n) |=> (cs_sva == CHK_CMD);
49      endproperty
50
51      //Slave must always go from CHK_CMD to WAIT_WR if 1st MOSI bit = 0
52      property CHK_WR;
53          @(posedge clk) disable iff(!rst_n)
54
55          $fell(SS_n) ##1 (~MOSI && ~SS_n) |=> (cs_sva == WAIT_WR);
56      endproperty
57
58      //Slave must always go from CHK_CMD to WAIT_RD if 1st MOSI bit = 1
59      property CHK_RD;
60          @(posedge clk) disable iff(!rst_n)
61
62          $fell(SS_n) ##1 (MOSI && ~SS_n) |=> (cs_sva == WAIT_RD);
63      endproperty
64
65
66
67      //MOSI = 0x
68      property MOSI_wr;
69          @(posedge clk) disable iff(!rst_n)
70
71          $fell(SS_n) ##1 (~MOSI && ~SS_n)[*2] |=> (cs_sva == WRITE);
72      endproperty
73
74      //MOSI = 1x
75      property MOSI_rd;
76          @(posedge clk) disable iff(!rst_n)
77
78          $fell(SS_n) ##1 (MOSI && ~SS_n)[*2] |=> (cs_sva == WAIT_RD2);
79      endproperty
80
81      //MOSI = 10
82      property MOSI_rd_add;
83          @(posedge clk) disable iff(!rst_n)
84
85          ( (cs_sva == READ_DATA) throughout SS_n[->1] ) ##[1:5] $fell(SS_n) ##1 (MOSI && ~SS_n)[*2] ##1 (~MOSI && ~SS_n) |=> (cs_sva == READ_ADD);
86      endproperty
87
88      //MOSI = 11
89      property MOSI_rd_data;
90          @(posedge clk) disable iff(!rst_n)
91
92          ( (cs_sva == READ_ADD) throughout SS_n[->1] ) ##[1:5] $fell(SS_n) ##1 (MOSI && ~SS_n)[*3] |=> (cs_sva == READ_DATA);
93      endproperty



97      //(READ_ADD -> READ_ADD) or (READ_DATA -> READ_DATA) must not occur
98      property no_add_add;
99          @(posedge clk) disable iff(!rst_n)
100
101         $fell(SS_n) ##1 (!SS_n)[*3] ##1 (cs_sva == READ_ADD)[*1:$] ##1 (SS_n) |=> ( (cs_sva != READ_ADD) throughout (cs_sva == READ_DATA)[->1] );
102     endproperty
103
104     property no_data_data;
105         @(posedge clk) disable iff(!rst_n)
106
107         $fell(SS_n) ##1 (!SS_n)[*3] ##1 (cs_sva == READ_DATA)[*1:$] ##1 (SS_n) |=> ( (cs_sva != READ_DATA) throughout (cs_sva == READ_ADD)[->1] );
108     endproperty
```

CODE SNIPPETS

TESTBENCH

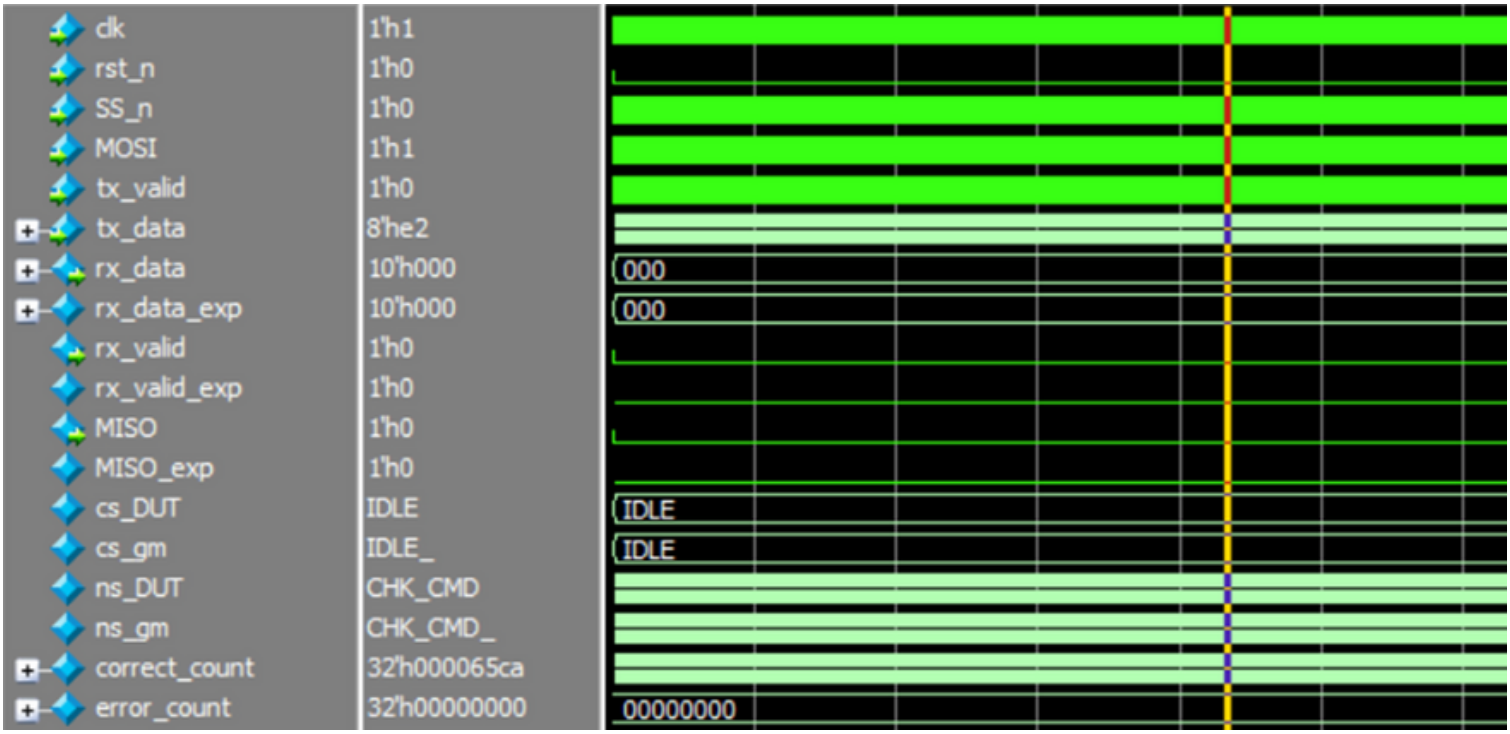
Assertions

```
113     ////////////////////////////////////////Checking Outputs
114
115     ////////////////rx_data & rx_valid
116     property rx;
117         logic [ADDR_SIZE+1:0] data_sva = 0;
118         @(posedge clk) disable iff(!rst_n)
119
120         $fell(SS_n) ##1 (!SS_n) ##1 ( ((cs != IDLE) && !SS_n), data_sva = {data_sva[ADDR_SIZE:0], MOSI} )[*(ADDR_SIZE+2)] |=> ( (rx_data == data_sva) && rx_valid);
121     endproperty
122
123     //rx_data @(rx_valid == 1) in WRITE state must start with 00 or 01
124     property rx_wr;
125         @(posedge clk) disable iff(!rst_n)
126
127         (rx_valid && (cs_sva == WRITE)) |-> ( (rx_data[ADDR_SIZE+1:ADDR_SIZE] == 2'b00) || (rx_data[ADDR_SIZE+1:ADDR_SIZE] == 2'b01) );
128     endproperty
129
130     //rx_data @(rx_valid == 1) in READ_ADD state must start with 10
131     property rx_rd_add;
132         @(posedge clk) disable iff(!rst_n)
133
134         (rx_valid && (cs_sva == READ_ADD)) |-> (rx_data[ADDR_SIZE+1:ADDR_SIZE] == 2'b10);
135     endproperty
136
137     //rx_data @(rx_valid == 1) in READ_DATA state must start with 11
138     property rx_rd_data;
139         @(posedge clk) disable iff(!rst_n)
140
141         (rx_valid && (cs_sva == READ_DATA)) |-> (rx_data[ADDR_SIZE+1:ADDR_SIZE] == 2'b11);
142     endproperty
143
144     //rx_valid cannot be high if tx_valid is high
145     property rx_tx;
146         @(posedge clk) disable iff(!rst_n)
147
148         ($rose(tx_valid) && (cs == READ_DATA)) |=> (~rx_valid) throughout (~tx_valid || (cs != READ_DATA))[->1];
149     endproperty
150
151     //rx_valid cannot be high at any checking state
152     property rx_chk;
153         @(posedge clk) disable iff(!rst_n)
154
155         ~( (cs == WRITE) || (cs == READ_ADD) || (cs == READ_DATA) ) |-> (~rx_valid);
156     endproperty
```

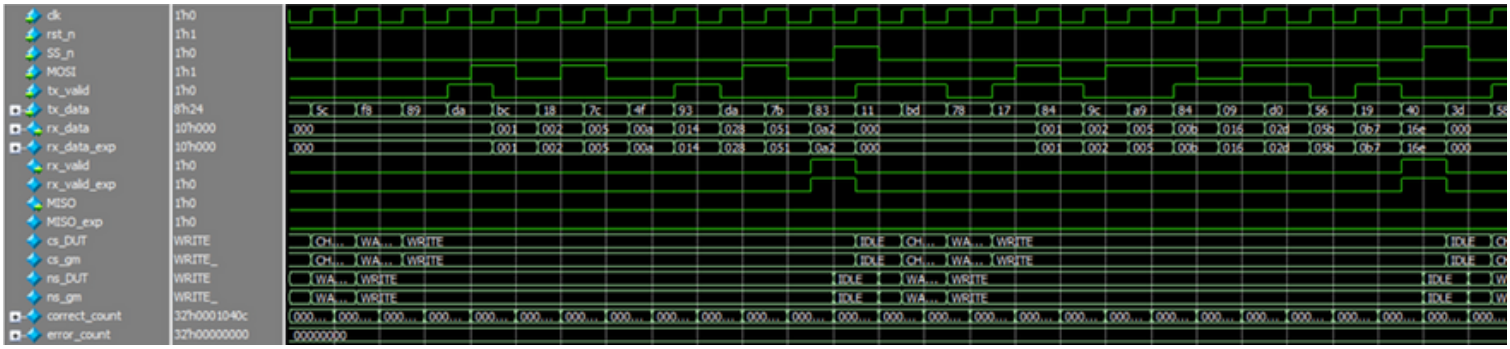
```
159     ////////////////MISO
160     property tx;
161         logic [ADDR_SIZE-1:0] data_sva = 0;
162         int i = 0;
163         @(posedge clk) disable iff(!rst_n || SS_n || $fell(tx_valid, @(negedge clk)) )
164
165         ( ( $rose(tx_valid) && (cs_sva == READ_DATA) ), data_sva = tx_data) |=> ((MISO == data_sva[ADDR_SIZE-1-i]), i++)[*(ADDR_SIZE)];
166     endproperty
167
168
169     //MISO cannot be high at any state other than READ_DATA
170     property MISO_rd_only;
171         @(posedge clk) disable iff(!rst_n || SS_n)
172
173         ~(cs_sva == READ_DATA) |-> (~MISO);
174     endproperty
```

Waveform Snippets

Reset Active: Outputs equal Zero

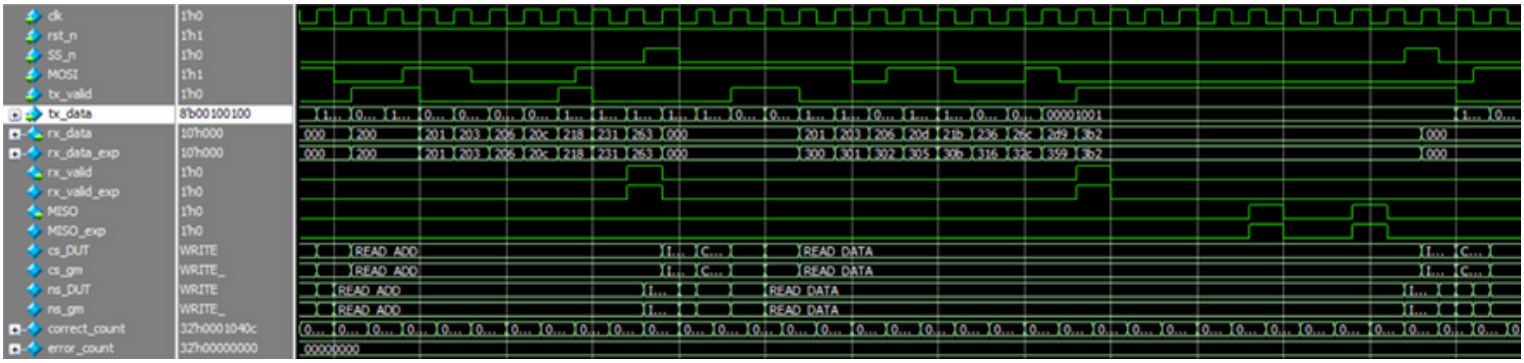


Write Operation

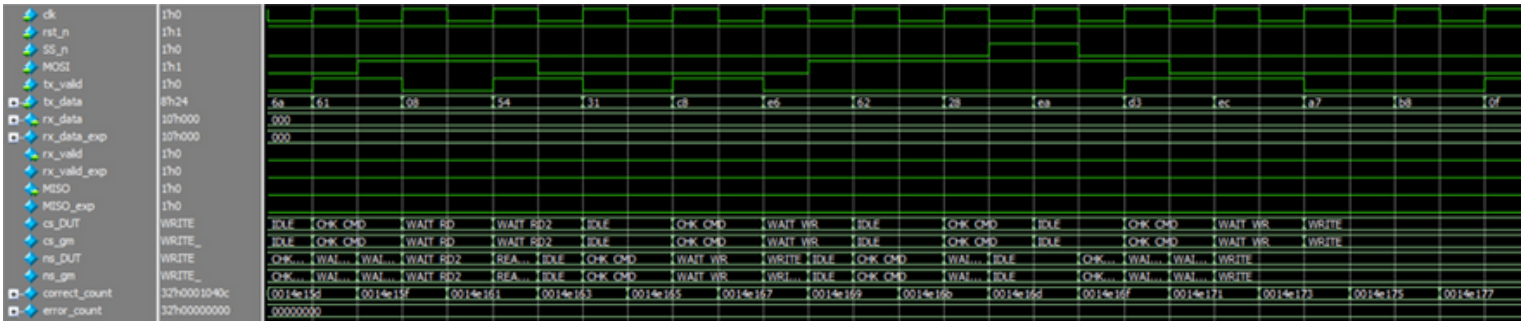


Waveform Snippets

Read Operation

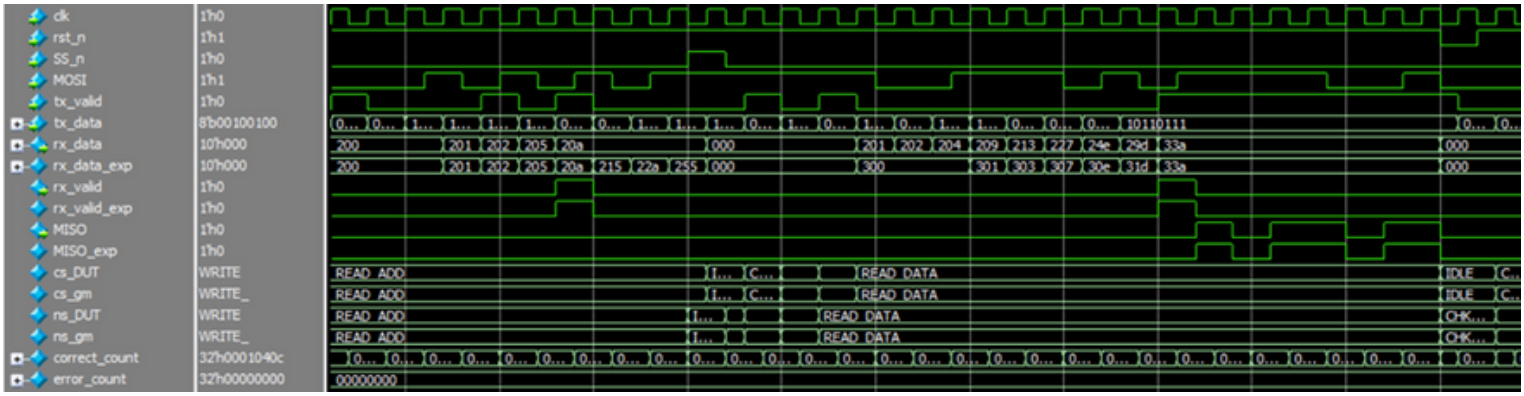


Checking States Transitions on entering Invalid MOSI Inputs



SS_n deactivated mid-transmission

rst_n activated mid-transmission



Correct Count and Error Count

```
# At End of Simulation: Correct Count = 1684243, Error Count = 0
```

Coverage Report

```
 8    Assertion Coverage:
 9        Assertions                      19        19         0   100.00%
10    ----------------------------------------------------------------------

53    Directive Coverage:
54        Directives                      19        19         0   100.00%
55

214   Branch Coverage:
215       Enabled Coverage              Bins      Hits    Misses  Coverage
216       ----------------              ----      ----    ------  --------
217       Branches                        59        59         0   100.00%

431   Condition Coverage:
432       Enabled Coverage              Bins   Covered    Misses  Coverage
433       ----------------              ----      ----    ------  --------
434       Conditions                      29        29         0   100.00%

700   FSM Coverage:
701       Enabled Coverage              Bins      Hits    Misses  Coverage
702       ----------------              ----      ----    ------  --------
703       FSM States                       8         8         0   100.00%
704       FSM Transitions                 14        14         0   100.00%

761   Statement Coverage:
762       Enabled Coverage              Bins      Hits    Misses  Coverage
763       ----------------              ----      ----    ------  --------
764       Statements                      69        69         0   100.00%
765

1378  Toggle Coverage:
1379      Enabled Coverage              Bins      Hits    Misses  Coverage
1380      ----------------              ----      ----    ------  --------
1381      Toggles                        118       118         0   100.00%

3649  Covergroup Coverage:
3650      Covergroups                      1        na        na   100.00%
3651          Coverpoints/Crosses         11        na        na        na
3652              Covergroup Bins         41        41         0   100.00%
```

Verification Plan

GENERAL PLAN:

• **White-Box Verification:**
- (mem, cs, tx_valid, tx_data, rx_valid, rx_data) internal signals are imported from the design to be used in Functional Coverage and Assertions.

• **Checking Results**
- Output value is checked against golden model.
- Output of read after write operation is checked in a separate task.
- Assertions are used for specific test cases.

• **Test Cases to Consider**
- Reset Signal (rst_n) Functionality across any combination of inputs.
- Writing then Reading from the same location.
- Normal Operation of SPI.
- SPI's behavior when SS_n stays active after transmission or goes inactive mid-transmission.
- Randomizing inputs across the same address (Exhausting all RAM addresses).

• **Functional Coverage Main Cover points and Cross Coverage**
- All RAM addresses are exercised.
- All Combinations of Control Signals are experienced.
- All addresses/data experienced the effect of rst_n signal.
- All addresses/data experienced the effect of SS_n signal.

• **Assertions**
- Critical Conditions (Memory Data must not be corrupted in these cases):
- Following rst_n assertion.
- At states other than (WRITE, READ_ADD).
- If rx_valid is not high.

Verification Requirements

Test Item	Description and Expected Output Behavior	Stimulus Generation	Functional Coverage	Functionality Check	Assertions
Reset Signal (rst_n)	SPI output (MISO) should reset to zero on reset assertion.	rst_n = 0. All other inputs are randomized under no constraints.	Included in Cross Coverage: cross_rst_SS: All combinations of control signals are covered. cross_rst_n_data: All addresses/data experienced the effect of rst_n signal.	Output must equal zero. Checked in check_reset task.	Property: reset_asserted
Control Signal (SS_n)	Slave should stay stable if SS_n remained active long after data transmission. Slave should go idle if SS_n is deactivated mid data transmission.	Included in all tasks of stimulus generation where SS_n is either active or randomized to be active most of the time.	Included in Cross Coverage: cross_rst_SS: All combinations of control signals are covered. cross_SS_n_data: All addresses/data experienced the effect of SS_n signal.	Output is checked against golden model.	Property: SS_inactive
Normal Operation of SPI	SS_n is deacitvated not directly after the supposed ending of a state to Check on IDLE behavior. MISO should always equal zero unless MOSI's first 3 bits, following SS_n falling edge, equal 3'b111.	rst_n = 1. SS_n = 1 deactivates each 30 clock cycles. Randomization of MOSI bits.	Included in Coverpoints: addresses_cp: All RAM Addresses are Exercised. Included in Cross Coverage: cross_SS_n_data: All addresses/data experienced the effect of SS_n signal.	Output is checked against golden model.	Property: no_wr_op: RAM contents must not change unless the slave is in a WRITE or READ_ADD state. MISO_no_rd: MISO must equal zero if slave is not in READ_DATA state. MISO_tx_inactive: MISO must equal zero if tx_valid is not high.
Performing all kinds of data manipulation across all addresses	Control Signal's access to all RAM's addresses is ensured. MISO should always equal zero unless MOSI's first 3 bits, following SS_n falling edge, equal 3'b111.	All inputs are randomized under Constraints ((rst_n inactive, SS_n active) most of the time).	Included in Coverpoints: addresses_cp: All RAM Addresses are Exercised. Included in Cross Coverage: cross_rst_SS: All Combinations of Control Signals are experienced. cross_rst_n_data: All addresses/data experienced the effect of rst_n signal. cross_SS_n_data: All addresses/data experienced the effect of SS_n signal.	Output is checked against golden model.	

Verification Requirements

Test Item	Description and Expected Output Behavior	Stimulus Generation
Reset Signal (rst_n)	SPI output (MISO) should reset to zero on reset assertion.	rst_n = 0. All other inputs are randomized under no constraints.
Control Signal (SS_n)	Slave should stay stable if SS_n remained active long after data transmission. Slave should go idle if SS_n is deactivated mid data transmission.	Included in all tasks of stimulus generation where SS_n is either active or randomized to be active most of the time.
Normal Operation of SPI	SS_n is deacitvated not directly after the supposed ending of a state to Check on IDLE behavior. MISO should always equal zero unless MOSI's first 3 bits, following SS_n falling edge, equal 3'b111.	rst_n = 1. SS_n = 1 deactivates each 30 clock cycles. Randomization of MOSI bits.
Performing all kinds of data manipulation across all addresses	Control Signal's access to all RAM's addresses is ensured. MISO should always equal zero unless MOSI's first 3 bits, following SS_n falling edge, equal 3'b111.	All inputs are randomized under Constraints ((rst_n inactive, SS_n active) most of the time).

Verification Requirements

Test Item	Functional Coverage	Functionality Check	Assertions
Reset Signal (rst_n)	Included in Cross Coverage: cross_rst_SS: All combinations of control signals are covered. cross_rst_n_data: All addresses/data experienced the effect of rst_n signal.	Output must equal zero. Checked in check_reset task.	Property: reset_asserted
Control Signal (SS_n)	Included in Cross Coverage: cross_rst_SS: All combinations of control signals are covered. cross_SS_n_data: All addresses/data experienced the effect of SS_n signal.	Output is checked against golden model.	Property: SS_inactive
Normal Operation of SPI	Included in Coverpoints: addresses_cp: All RAM Addresses are Exercised. Included in Cross Coverage: cross_SS_n_data: All addresses/data experienced the effect of SS_n signal.	Output is checked against golden model.	Property: no_wr_op: RAM contents must not change unless the slave is in a WRITE or READ_ADD state. MISO_no_rd: MISO must equal zero if slave is not in READ_DATA state. MISO_tx_inactive: MISO must equal zero if tx_valid Is not high.
Performing all kinds of data manipulation across all addresses	Included in Coverpoints: addresses_cp: All RAM Addresses are Exercised. Included in Cross Coverage: cross_rst_SS: All Combinations of Control Signals are experienced. cross_rst_n_data: All addresses/data experienced the effect of rst_n signal. cross_SS_n_data: All addresses/data experienced the effect of SS_n signal.	Output is checked against golden model.	

Bug Report

Bug	Original Code	Fix	Lines (Original Version)	Lines (Edited Version)

Parameter (ADDR_SIZE) should be used instead of the actual size to respond correctly to changes in parameters	wire [7:0] tx_data1;
	wire [9:0] rx_data1;
		parameter ADDR_SIZE = 8;
		parameter MEM_DEPTH = 256;
		wire [ADDR_SIZE-1:0] tx_data1;
		wire [ADDR_SIZE+1:0] rx_data1;
			8
			10
				[18:19]
				24
				29

Code Snippets

DESIGN

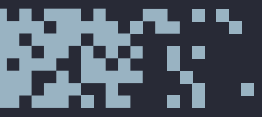
Before (Original)

```
    ////////////////////////////////////////////////////////////////Original Code//////////////////////////////////////////////////////////////////
/*
    input MOSI,clk,rst_n,SS_n;
    output MISO;

    wire [7:0] tx_data1;
    wire tx_valid1,rx_valid1;
    wire [9:0] rx_data1;
    slave #(ADDR_SIZE) s1(.MOSI(MOSI),.SS_n(SS_n),.clk(clk),.rst_n(rst_n),.tx_valid(tx_valid1),.tx_data(tx_data1),.rx_data(rx_data1),.rx_valid(rx_valid1),.MISO(MISO));
    ram #(.ADDR_SIZE(ADDR_SIZE), .MEM_DEPTH(MEM_DEPTH)) r1(.din(rx_data1),.clk(clk),.rst_n(rst_n),.rx_valid(rx_valid1),.dout(tx_data1),.tx_valid(tx_valid1));
*/
    ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

After (Edited)

```
    ////////////////////////////////////////////////////////////////Edited Code//////////////////////////////////////////////////////////////////
    //Parameters
    parameter ADDR_SIZE = 8;
    parameter MEM_DEPTH = 256;

    input MOSI,clk,rst_n,SS_n;
    output MISO;

    wire [ADDR_SIZE-1:0] tx_data1;
    wire tx_valid1,rx_valid1;
    wire [ADDR_SIZE+1:0] rx_data1;
    slave #(ADDR_SIZE) s1(.MOSI(MOSI),.SS_n(SS_n),.clk(clk),.rst_n(rst_n),.tx_valid(tx_valid1),.tx_data(tx_data1),.rx_data(rx_data1),.rx_valid(rx_valid1),.MISO(MISO));
    ram #(.ADDR_SIZE(ADDR_SIZE), .MEM_DEPTH(MEM_DEPTH)) r1(.din(rx_data1),.clk(clk),.rst_n(rst_n),.rx_valid(rx_valid1),.dout(tx_data1),.tx_valid(tx_valid1));

    ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

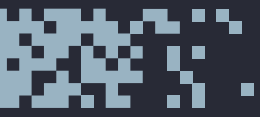
CODE SNIPPETS



PACKAGE

```
1   package wrapper_pkg;
2
3       class wrapper_class #(int ADDR_SIZE = 8, MEM_DEPTH = 256);
4
5           //Inputs to DUT
6           bit clk;
7           rand bit rst_n, SS_n, MOSI;
8
9           //Target Address (Used in Stimulus Generation)
10          rand bit [ADDR_SIZE-1:0] temp_address;
11          rand bit [ADDR_SIZE+1:0] temp_data;
12
13
14          constraint rst_c {
15              rst_n           dist {0:=2,     1:=98};             //rst_n is inactive most of the time
16          }
17
18          constraint SS_c {
19              SS_n            dist {0:=95,    1:=5};              //SS_n is active most of the time
20          }
21
22
23          covergroup wrapper_cg;
24
25              addresses_cp:       coverpoint temp_address;    //All Addresses are Exercised
26
27              cross_rst_SS:       cross rst_n, SS_n, MOSI;    //All Combinations of Control Signals are experienced
28
29              cross_rst_n_data:   cross rst_n, temp_data;     //All addresses/data experienced the effect of rst_n signal
30
31              cross_SS_n_data:    cross SS_n, temp_data;      //All addresses/data experienced the effect of SS_n signal
32
33          endgroup
34
35          function new(bit rst_n = 0, SS_n = 1, MOSI = 0);
36              this.rst_n      = rst_n;
37              this.SS_n       = SS_n;
38              this.MOSI       = MOSI;
39
40              wrapper_cg = new();
41          endfunction
42
43      endclass
44
45  endpackage
```

CODE SNIPPETS



TESTBENCH

Stimulus Generation

```
53      initial begin
54
55          //Assert Reset - Initial State
56          assert_reset;
57
58
59
60
61          /*TEST 0:   - Checks (rst_n) Functionality
62                      - Output (MISO) resets to zero @(rst_n = 0)
63                      - Randomization Under No Constraints
64          */
65          stimulus_gen_reset;
66
67
68
69
70          //Deassert Reset
71          deassert_reset;
72
73
74
75
76
77          /*TEST 1:   - Write Data to a specific address
78                      - All Addresses are Exercised
79                      - Read from the Same address
80                      - Semi Directed Test Cases
81          */
82          stimulus_gen1;
83
84
85
86
87
88          /*TEST 2:   - Normal Operation of SPI
89                      - Randomization of MOSI bit (rst_n inactive, SS_n deactivates each 30 clock cycles)
90                      - SS_n is deacitvated not directly after the supposed ending of a state -> Checking for IDLE behavior
91                      - MISO should always equal zero unless MOSI's first 3 bits, following SS_n falling edge, equal 3'b111
92          */
93          stimulus_gen2;
94
95
96
97
98          /*TEST 3:   - Randomization under Constraints ((rst_n inactive, SS_n active) most of the time)
99                      - MOSI bits (representing data/address) experience the effect of rst_n and SS_n
100                     - Ensuring Control Signal's access to all RAM's addresses
101                     - MISO should always equal zero unless MOSI's first 3 bits, following SS_n falling edge, equal 3'b111
102         */
103         stimulus_gen3;
104
109         /*TEST 4:   - Randomization under Constraints ((rst_n inactive, SS_n active) most of the time)
110                     - rst_n active -> Output resets to zero
111                     - SS_n inactive -> Output resets to zero
112                     - Otherwise, MOSI bits control the flow
113                     - MISO should always equal zero unless MOSI's first 3 bits, following SS_n falling edge, equal 3'b111
114         */
115         stimulus_gen4;
116
117
118
119
120
121
122
123         //Correct Count and Error Count Display
124         $display("At End of Simulation: Correct Count = %0d, Error Count = %0d", correct_count, error_count);
125         @(negedge clk); $stop;
126
127     end
```

CODE SNIPPETS

TESTBENCH

Tasks

```
136
137     ////////////////////////////Reset-Related/////////////////////////////
138
139     //Assert Reset
140     task assert_reset;
141         rst_n = 0;
142         wrap.constraint_mode(0);
143
144         @(negedge clk);
145     endtask
146
147     //TEST 0: Reset Asserted
148     task stimulus_gen_reset;
149         for(int i=0; i<TESTS; i++) begin
150             assert(wrap.randomize());
151             SS_n        = wrap.SS_n;
152             MOSI        = wrap.MOSI;
153
154             check_reset;
155             @(negedge clk);
156         end
157     endtask
158
159     //CHECKER: Reset Asserted
160     task check_reset;
161         @(posedge clk);
162
163         if(MISO !== 0) begin
164             $display("ERROR: (Reset Asserted) -> Output -MISO- equals %0h, but should equal 0 \t\t--time: %0t", MISO, $time);
165             error_count++;
166         end
167         else correct_count++;
168     endtask
169
170     //Deassert Reset
171     task deassert_reset;
172         rst_n = 1;
173         SS_n = 1;                       //Setting a Starting Point following rst_n deassertion
174         wrap.constraint_mode(1);
175
176         @(negedge clk);
177     endtask
178
179
```

Code Snippets

TESTBENCH

```
187     //Marks the beginning of every possible sequence
188     task start_seq;
189         SS_n = 0;
190         @(negedge clk);
191     endtask
192
193     //TEST 1: Same Address (Write then Read)
194     task stimulus_gen1;
195         for(int i=0; i<TESTS; i++) begin
196             assert(wrap.randomize());
197             temp_address = wrap.temp_address;
198
199             //Send Write Address
200             start_seq;
201
202             MOSI = 0;
203             repeat (3) @(negedge clk);
204
205             for(int i=0; i<ADDR_SIZE; i++) begin
206                 MOSI = temp_address[ADDR_SIZE-1-i];
207                 @(negedge clk);
208             end
209
210             SS_n = 1;
211             @(negedge clk);
212
213             //Send Write Data
214             start_seq;
215
216             MOSI = 0;
217             repeat (2) @(negedge clk);
218             MOSI = 1;
219             @(negedge clk);
220
221             for(int i=0; i<ADDR_SIZE; i++) begin
222                 assert(wrap.randomize());
223                 MOSI = wrap.MOSI;
224                 saved_MISO[ADDR_SIZE-1-i] = wrap.MOSI;
225                 @(negedge clk);
226             end
227
228             SS_n = 1;
229             @(negedge clk);
```

CODE SNIPPETS

TESTBENCH

```
231                 //Send Read Address (= Write Address)
232                 start_seq;
233
234                 MOSI = 1;
235                 repeat (2) @(negedge clk);
236                 MOSI = 0;
237                 @(negedge clk);
238
239                 for(int i=0; i<ADDR_SIZE; i++) begin
240                     MOSI = temp_address[ADDR_SIZE-1-i];
241                     @(negedge clk);
242                 end
243
244                 SS_n = 1;
245                 @(negedge clk);
246
247                 //Wait for Read Data
248                 start_seq;
249
250                 MOSI = 1;
251                 repeat (3) @(negedge clk);
252
253                 for(int i=0; i<ADDR_SIZE; i++) begin
254                     assert(wrap.randomize());
255                     MOSI = wrap.MOSI;
256                     @(negedge clk);
257                 end
258
259                 repeat (2) @(negedge clk);
260
261                 for(int i=0; i<ADDR_SIZE; i++) begin
262                     check_rd_after_wr(i);
263                     @(negedge clk);
264                 end
265
266                 SS_n = 1;
267                 @(negedge clk);
268             end
269
270     endtask
```

CODE SNIPPETS

TESTBENCH

```
274     //TEST 2: Normal Operation
275     task stimulus_gen2;
276         for(int i=0; i<TESTS; i++) begin
277             SS_n = 0;
278             for(int i=0; i<30; i++) begin
279                 assert(wrap.randomize());
280                 MOSI = wrap.MOSI;
281                 @(negedge clk);
282             end
283             SS_n = 1;
284             @(negedge clk);
285         end
286     endtask
287
288
289
290     //TEST 3: SS_n and rst_n randomized accross all data/addresses (MISO bits)
291     task stimulus_gen3;
292         wrap.constraint_mode(1);
293         for(int i=0; i<TESTS; i++) begin
294             assert(wrap.randomize());
295             temp_data   = wrap.temp_data;
296             for(int i=0; i<ADDR_SIZE+2; i++) begin
297                 assert(wrap.randomize());
298                 rst_n   = wrap.rst_n;
299                 SS_n    = wrap.SS_n;
300                 MOSI    = temp_data[i];
301                 @(negedge clk);
302             end
303         end
304     endtask
305
306
307
308     //TEST 3: Randomization
309     task stimulus_gen4;
310         wrap.constraint_mode(1);
311         for(int i=0; i<TESTS; i++) begin
312             assert(wrap.randomize());
313             rst_n   = wrap.rst_n;
314             SS_n    = wrap.SS_n;
315             MOSI    = wrap.MOSI;
316
317             @(negedge clk);
318         end
319     endtask
```

CODE SNIPPETS

TESTBENCH

Checker

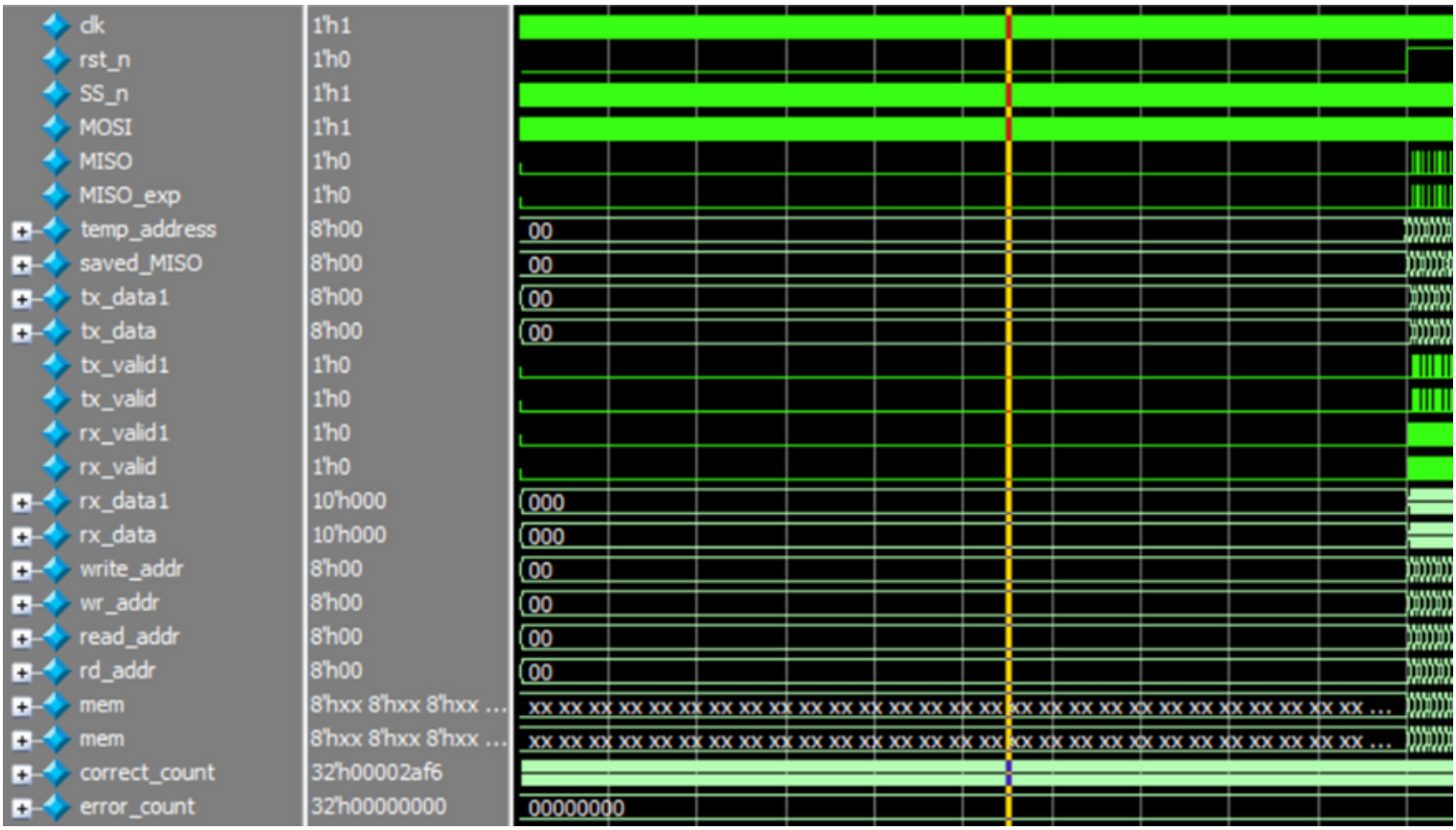
```
330     ////////////////////////////Checking Results////////////////////////////
331
332     //CHECKER:
333
334     //General Checking
335     always @(negedge clk or negedge rst_n) begin
336         if(~rst_n) check_reset;
337         else check_result;
338     end
339
340     task check_result;
341         if(MISO !== MISO_exp) begin
342             $display("ERROR: Output -MISO- equals %0h, but should equal %0h \t\t--time: %0t", MISO, MISO_exp, $time);
343             error_count++;
344         end
345         else correct_count++;
346     endtask
347
348
349     //Specific to Consecutive Write and Read Operations
350     task check_rd_after_wr(int i);
351         if(MISO !== saved_MISO[ADDR_SIZE-1-i]) begin
352             $display("ERROR (Corrupted RAM Data): Output -MISO- equals %0h, but should equal %0h \t\t--time: %0t", MISO, saved_MISO[ADDR_SIZE-1-i], $time);
353             error_count++;
354         end
355         else correct_count++;
356     endtask
```

ASSERTIONS

```
35      property reset_asserted;
36          @(posedge clk)
37
38          !rst_n |=> (~MISO && $stable(mem));
39      endproperty
40
41      property no_wr_op;
42          @(posedge clk) disable iff(!rst_n)
43
44          !((cs_sva == WRITE) || (cs_sva == READ_ADD)) |=> $stable(mem);
45      endproperty
46
47      property SS_inactive;
48          @(posedge clk) disable iff(!rst_n)
49
50          (SS_n) |=> ##1 $stable(mem);
51      endproperty
52
53
54
55
56      //////////////Checking Output Behavior
57
58      property MISO_no_rd;
59          @(posedge clk) disable iff(!rst_n)
60
61          !(cs_sva == READ_DATA) |=> ~MISO;
62      endproperty
63
64      property MISO_tx_inactive;
65          @(posedge clk) disable iff(!rst_n)
66
67          !(tx_valid) |=> ~MISO;
68      endproperty
```

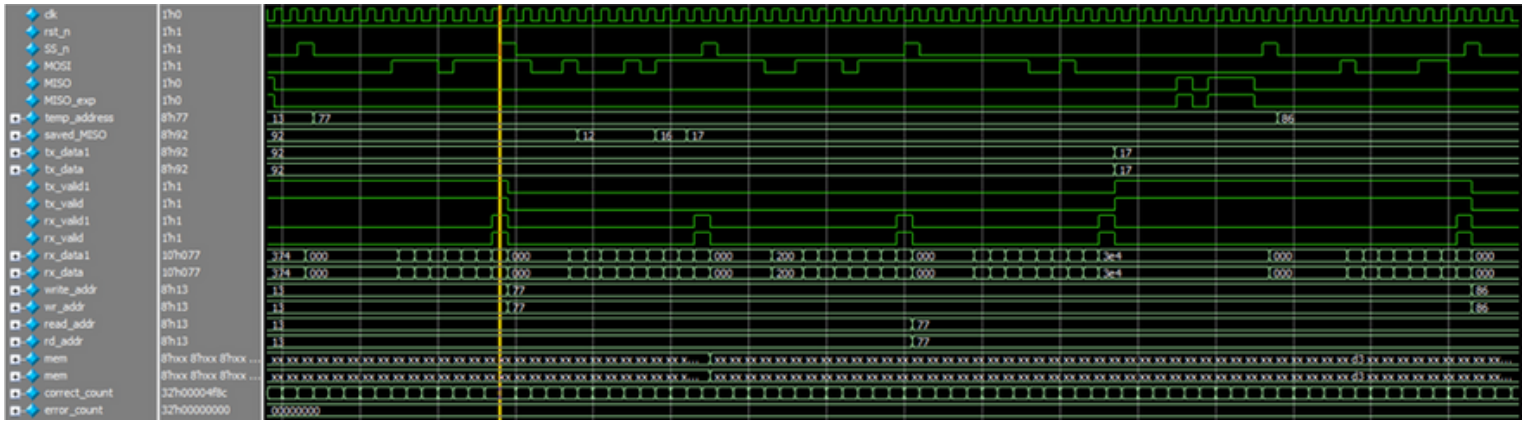
Waveform Snippets

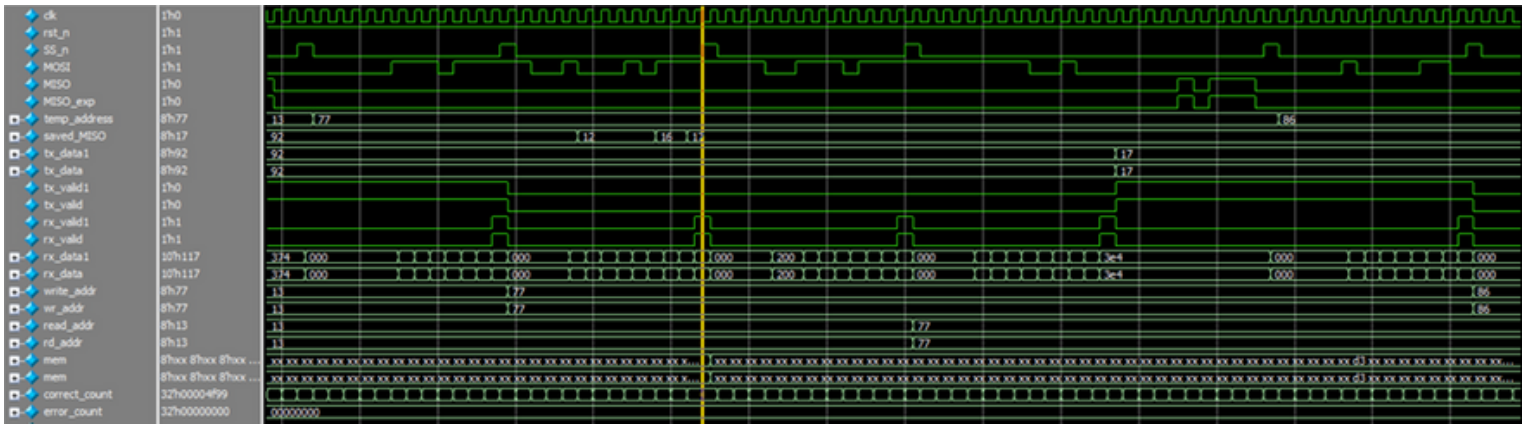
Reset Active: Outputs equal Zero



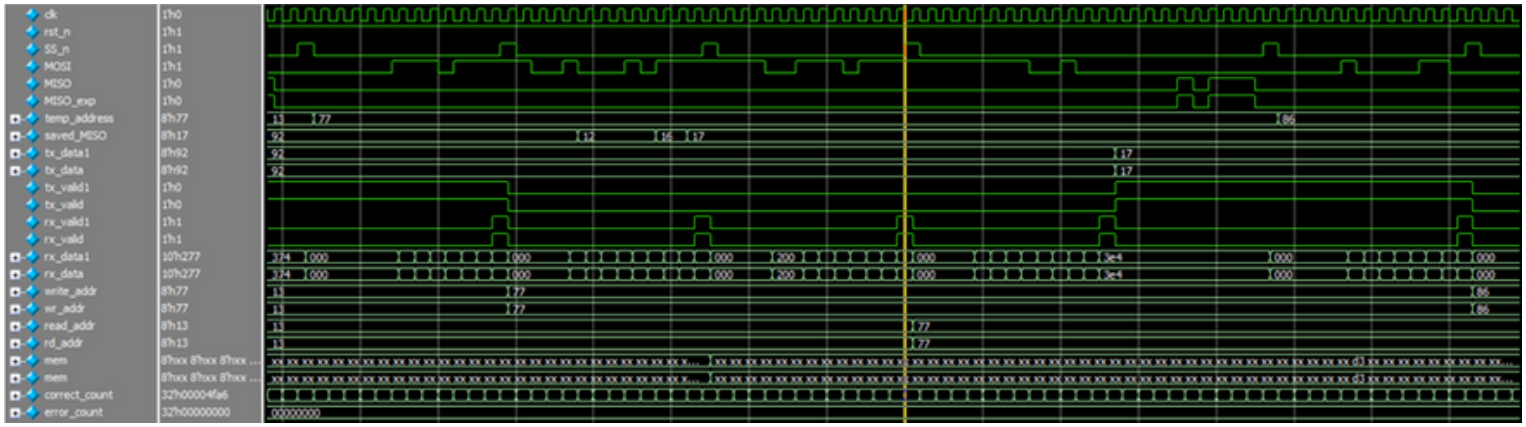
Waveform Snippets

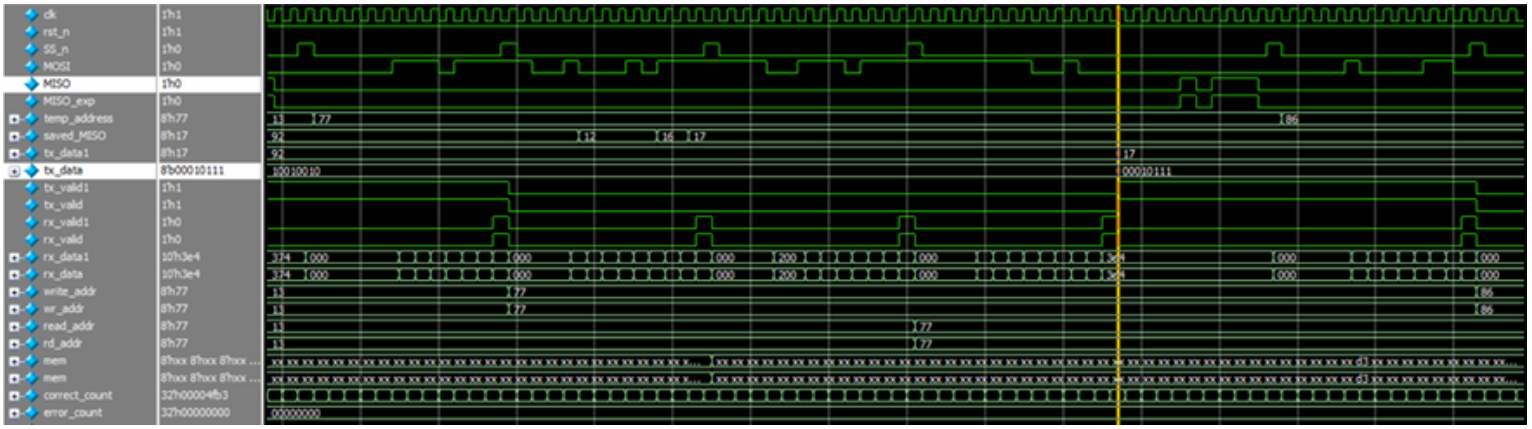
Write then Read from the same address:





Waveform Snippets





Correct Count and Error Count

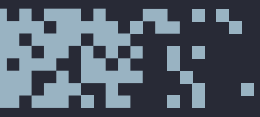
```
# At End of Simulation: Correct Count = 1141110, Error Count = 0
```

Coverage Report

SLAVE

```
3   ================================================================================
4   === Instance: /SPI_Wrapper_tb/DUT/s1
5   === Design Unit: work.slave
6   ================================================================================
7   Branch Coverage:
8       Enabled Coverage              Bins      Hits    Misses  Coverage
9       ----------------              ----      ----    ------  --------
10      Branches                        59        59         0   100.00%
11
---
224 Condition Coverage:
225     Enabled Coverage              Bins   Covered    Misses  Coverage
226     ----------------              ----      ----    ------  --------
227     Conditions                      29        29         0   100.00%
228
493 FSM Coverage:
494     Enabled Coverage              Bins      Hits    Misses  Coverage
495     ----------------              ----      ----    ------  --------
496     FSM States                       8         8         0   100.00%
497     FSM Transitions                 14        14         0   100.00%
554 Statement Coverage:
555     Enabled Coverage              Bins      Hits    Misses  Coverage
556     ----------------              ----      ----    ------  --------
557     Statements                      69        69         0   100.00%
558
---
1171 Toggle Coverage:
1172    Enabled Coverage              Bins      Hits    Misses  Coverage
1173    ----------------              ----      ----    ------  --------
1174    Toggles                        118       118         0   100.00%
1175
```

Coverage Report



RAM

```
1207    ================================================================================
1208    === Instance: /SPI_Wrapper_tb/DUT/r1
1209    === Design Unit: work.ram
1210    ================================================================================
1211    Branch Coverage:
1212        Enabled Coverage              Bins      Hits    Misses  Coverage
1213        ----------------              ----      ----    ------  --------
1214        Branches                         7         7         0   100.00%
1215
1245    Statement Coverage:
1246        Enabled Coverage              Bins      Hits    Misses  Coverage
1247        ----------------              ----      ----    ------  --------
1248        Statements                      13        13         0   100.00%

1416    Toggle Coverage:
1417        Enabled Coverage              Bins      Hits    Misses  Coverage
1418        ----------------              ----      ----    ------  --------
1419        Toggles                         76        76         0   100.00%
```

Coverage Report

SPI WRAPPER

```
1577  ================================================================================
1578  === Instance: /SPI_Wrapper_tb/DUT
1579  === Design Unit: work.SPI_Wrapper
1580  ================================================================================
1581  Toggle Coverage:
1582      Enabled Coverage              Bins      Hits    Misses  Coverage
1583      ----------------              ----      ----    ------  --------
1584      Toggles                         50        50         0   100.00%
1585
```

```
3644  Covergroup Coverage:
3645      Covergroups                     1        na         na   100.00%
3646          Coverpoints/Crosses         8        na         na        na
3647              Covergroup Bins       398       398          0   100.00%
```

```
1447  Assertion Coverage:
1448      Assertions                      5         5         0   100.00%
1449  --------------------------------------------------------------------
```

```
1464  Directive Coverage:
1465      Directives                      5         5         0   100.00%
1466
```