CENTER OF NANOELECTRONICS & DEVICES

CND 212: Digital Testing and Verification

Final Project [2]

Verification of SPI Slave IP with System Verilog

Submitted by:

Amr Sami V23010343

Hagar Yasser Raslan V23010615

Fatma Hazem Abd El-salam V23010086

Omnia Alaa Farouk V23010574

Fayza Ahmed Sayed Ahmed V23010517

Submitted to TA: Eng. Rana

Date: 17th April 2024

Contents

Table of Figures

1 SPI Slave Module Overview

The SPI slave module is designed to facilitate communication with a master device using the Serial Peripheral Interface (SPI) protocol. This module is responsible for managing data transmission and reception between the master and slave devices.

1.1 Key Features:

1. **State-Based Operation:**
   - The module operates in multiple states, including idle, command checking, writing address/data, and reading address/data. These states are controlled by a state machine that transitions based on input signals and internal logic as seen in figure [1].
2. **Input Handling:**
   - Input signals from the SPI interface are monitored to detect various events such as command reception, data transmission, and clock cycles. These inputs trigger state transitions and data processing within the module.
3. **Output Handling:**
   - The module manages data transmission to the master device and reception of data from the master. Output signals are controlled based on the current state and data availability.

1.2 Operation:

1. **Idle State:**
   - In the idle state, the module waits for the slave select signal (io.ss_n) to indicate communication from the master device. Upon detection of the slave select signal, the module transitions to the command checking state.
2. **Command Checking State:**
   - Upon receiving the slave select signal, the module checks for the command sent by the master device. Depending on the command received (cmd), the module transitions to the corresponding state for address/data writing or reading.
3. **Write Address/Data State:**
   - In this state, the module receives address or data from the master device and stores it internally (internal_register). It increments a counter to track the number of bits received and transitions to the idle state once the expected number of bits is received.

4. **Read Address/Data State:**

   - Similar to the write state, this state involves receiving address or data from the master device and transmitting it back (io.rx_data) after processing. Once the transmission is complete, the module transitions back to the idle state.

1.3 Conclusion:

The SPI slave module provides essential functionality for interfacing with a master device using the SPI protocol. Its state-based operation and input/output handling mechanisms make it suitable for integration into larger digital systems requiring SPI communication capabilities.

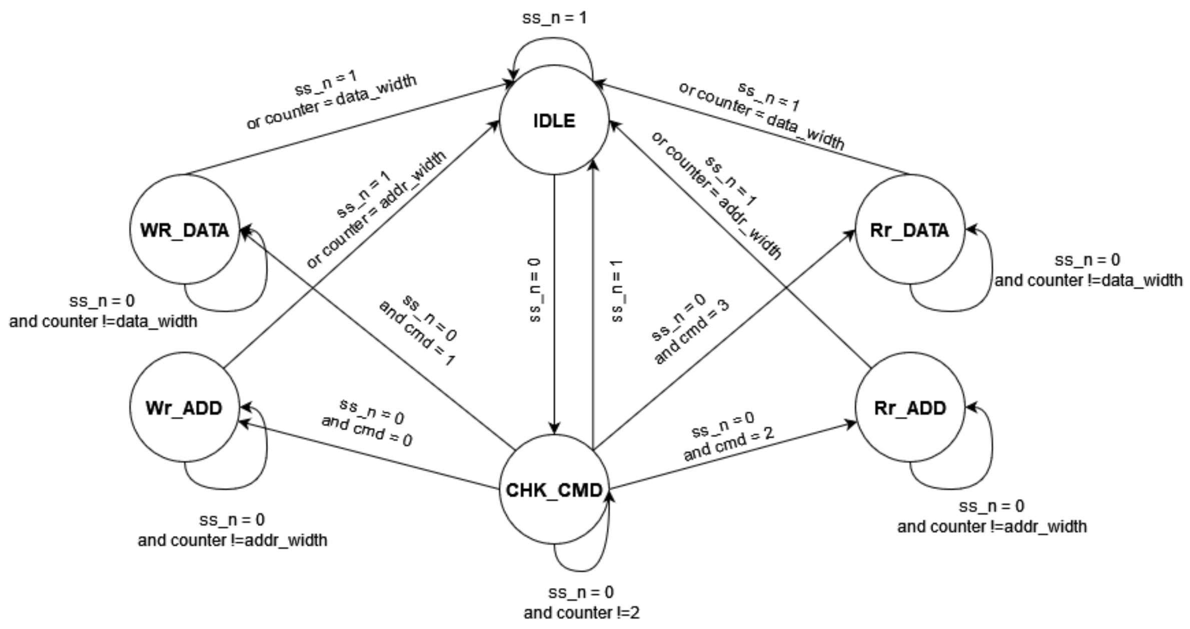

ss_n = 1
IDLE
ss_n = 1 or counter = data_width
ss_n = 1 or counter = data_width
ss_n = 1 or counter = addr_width
ss_n = 1 or counter = addr_width
WR_DATA
Rr_DATA
ss_n = 0 and counter !=data_width
ss_n = 0 and counter !=data_width
ss_n = 0
ss_n = 1
ss_n = 0 and cmd = 1
ss_n = 0 and cmd = 3
Wr_ADD
Rr_ADD
ss_n = 0 and cmd = 0
ss_n = 0 and cmd = 2
CHK_CMD
ss_n = 0 and counter !=addr_width
ss_n = 0 and counter !=addr_width
ss_n = 0 and counter !=2

Figure 1: State Machine Diagram

1.4 SPI Slave Verilog Code

```
module spi_slave #(parameter ADDR_width = 8) (spi_slave_io io);

  localparam idle = 0, chk_cmd = 1, write_address = 2, write_data = 3,
read_address = 4, read_dummy = 5;

  reg [2:0]                  current_state, next_state;
  reg [ADDR_width-1:0]       internal_register;
  reg [3:0]                  counter_receiver = 0;
  reg [3:0]                  counter_transmitter = 0;
  reg [1:0]                  cmd;
  reg [7:0]               temp_reg;
  reg                     tx_valid_flag;

  //next state always block
  always @(posedge io.clk or negedge io.rst_n)
    begin
      if(!io.rst_n)
        begin
          current_state <= idle;
        end
      else
        begin
          current_state <= next_state;
        end
    end

  //current state always block
  always @(*)
    begin
      case(current_state)
      idle : begin
              if (!io.ss_n)
                begin
                  next_state = chk_cmd;
                end
              else
                begin
                  next_state = idle;
                end
             end

      chk_cmd : begin
                  if ( !io.ss_n)
                    begin
                      if(counter_receiver == 2'b00)
                        begin
                          case(cmd)
                            2'b00: next_state = write_address;
                            2'b01: next_state = write_data;
                            2'b10: next_state = read_address;
                            2'b11: next_state = read_dummy;
                            default : next_state = idle;
                          endcase
                        end
                      else
                        begin
 next_state = chk_cmd;
                        end

                    end
                  else
                    begin
                      next_state = idle;
                    end
                end


     write_address : begin
                       if(!io.ss_n)
                         begin
                           if(counter_receiver == (ADDR_width+1 ) )
                             begin
                               next_state = idle;
                             end
                           else
                             begin
                               next_state = write_address;
                             end
                         end
                       else
                         begin
                           next_state = idle;
                         end
                     end

     write_data : begin
                       if(!io.ss_n)
                         begin
                           if(counter_receiver == (ADDR_width+1 ) )
                             begin
                               next_state = idle;
                             end
                           else
                             begin
                               next_state = write_data;
                             end
                         end
                       else
                         begin
                           next_state = idle;
                         end
                     end


     read_address : begin
                       if(!io.ss_n)
                         begin
                           if(counter_receiver == (ADDR_width+1 ) )
                             begin
                               next_state = idle;
                             end
                           else
                             begin
                               next_state = read_address;
                             end
                         end
 else
                         begin
                           next_state = idle;
                         end
                     end
     read_dummy : begin
                       if(!io.ss_n)
                         begin
                           if(counter_receiver == (ADDR_width +1) )
                             begin
                               next_state = idle;
                             end
                           else
                             begin
                               next_state = read_dummy;
                             end
                         end
                       else
                         begin
                           next_state = idle;
                         end
                     end
     default : begin
                  next_state = idle;
     end
     endcase
    end


  always @(posedge io.clk)
    begin
      case(current_state)
      idle : begin
               io.rx_valid <= 1'b0;
               io.rx_data <= 'b0;
               internal_register <= 'b0;
               counter_receiver <= 'b0;
               cmd <= {io.mosi,cmd[1]};
      end

      chk_cmd : begin
        if ( !io.ss_n)
                    begin
                      counter_receiver <= counter_receiver+1;
                      cmd <= {io.mosi,cmd[1]};
                    end
                  else
                    begin
                      counter_receiver <= 'b0;
                      cmd <= 2'b0;
                    end
                end


     write_address : begin
       if (!io.ss_n)
                         begin
                           if(counter_receiver == (ADDR_width+1) )
                               begin
                                 io.rx_valid <= 1'b1;
   io.rx_data <= {internal_register,cmd};
                               end
                           else
                             begin
                               counter_receiver <= counter_receiver+1;
                               internal_register       <=       {io.mosi,
  internal_register[ADDR_width-1:1]};
                             end
                         end
                       else
                         begin
                           counter_receiver <= 'b0;
                           io.rx_valid <= 'b0;
                           io.rx_data <= 'b0;
                         end
                     end

     write_data :begin
       if (!io.ss_n)
                         begin
                           if(counter_receiver == (ADDR_width+1) )
                               begin
                                 io.rx_valid <= 1'b1;
                                 io.rx_data <= {internal_register,cmd};
                               end
                           else
                             begin
                               counter_receiver <= counter_receiver+1;
                               internal_register       <=       {io.mosi,
  internal_register[ADDR_width-1:1]};
                             end
                         end
                       else
                         begin
                           counter_receiver <= 'b0;
                           io.rx_valid <= 'b0;
                           io.rx_data <= 'b0;
                         end
                     end



     read_address : begin
       if (!io.ss_n)
                         begin
                           if(counter_receiver == (ADDR_width +1) )
                               begin
                                 io.rx_valid <= 1'b1;
                                 io.rx_data <= {internal_register,cmd};
                               end
                           else
                             begin
                               counter_receiver <= counter_receiver+1;
                               internal_register       <=       {io.mosi,
  internal_register[ADDR_width-1:1]};
                             end
                         end
                       else
                         begin
                           counter_receiver <= 'b0;
 io.rx_valid <= 'b0;
                           io.rx_data <= 'b0;
                         end
                     end

     read_dummy : begin
                       if (!io.ss_n)
                         begin
                           if(counter_receiver == (ADDR_width+1) )
                               begin
                                 io.rx_valid <= 1'b1;
                                 io.rx_data <= {internal_register,cmd};
                               end
                           else
                             begin
                               counter_receiver <= counter_receiver+1;
                               internal_register       <=       {io.mosi,
  internal_register[ADDR_width-1:1]};
                             end
                         end
                       else
                         begin
                           counter_receiver <= 'b0;
                           io.rx_valid <= 'b0;
                           io.rx_data <= 'b0;
                         end
                     end

     default : begin
                   io.rx_data <= 'b0;
                   io.rx_valid <= 1'b0;
                   counter_receiver <= 'b0;
                   internal_register <= 'b0;
     end

     endcase
    end


   // output always block
   always@(posedge io.clk or negedge io.rst_n)
     begin

       if(!io.rst_n)
         begin
           counter_transmitter <= 'b0;
           io.miso <= 1'b0;
           temp_reg <= 8'b0;
           tx_valid_flag <= 1'b0;
         end

     else
       begin
       if (io.tx_valid)
           begin
               temp_reg <= io.tx_data;
               counter_transmitter <= 'b0;
               tx_valid_flag <= 1'b1;
           end
         else
 begin
             if(!io.ss_n & tx_valid_flag)
               begin
                 if (counter_transmitter != ADDR_width)
                   begin
                     io.miso <= io.tx_data[counter_transmitter];
                     counter_transmitter <= counter_transmitter+1;
                   end
                 else
                   begin
                     counter_transmitter <= 'b0;
                     tx_valid_flag <= 1'b0;
                   end
                end
             else
               begin
                 counter_transmitter <= 'b0;
                 tx_valid_flag <= 1'b0;
               end
         end
    end
    end

  endmodule
```

2 Interface

2.1 Overview

The SPI slave interface defines the communication signals and clocking mechanism for interfacing with the SPI master device. It provides a standardized interface for data exchange and synchronization between the master and slave devices.

2.1.1 Interface Signals:

1. **Clock Signal (clk):**

   - Synchronous clock signal generated by the master device and sent to the slave to synchronize data transmission.

2. **Reset Signal (rst_n):**

   - Negative-edge reset signal used to reset the internal state of the SPI slave module.

3. **Master Output Serial Input (mosi):**
   - Signal representing the data transmitted from the master device to the slave device over the SPI bus.
4. **Master Input Serial Output (miso):**

   - Signal representing the data transmitted from the slave device to the master device over the SPI bus
   -

.

5. **Slave Select Signal (ss_n):**
   - Signal indicating the selection of the slave device for communication. It is active low and controlled by the master device.

6. **Receive Valid Signal (rx_valid):**
   - Signal indicating the validity of the received data by the slave device.

7. **Transmit Valid Signal (tx_valid):**
   - Signal indicating the validity of the transmitted data by the RAM Memory.

8. **Receive Data (rx_data):**
   - Data received by the slave device from the master device. The width of the data bus is determined by the ADDR_width parameter.

9. **Transmit Data (tx_data):**
   - Data transmitted by the slave device to the master device. The width of the data bus is determined by the ADDR_width parameter.

2.1.2 Clocking Block:

The clocking block (cb) defines the clocking mechanism for the interface. It provides timing controls for input and output signals to ensure proper synchronization between the master and slave devices.

2.1.3 Modports:

1. **Testbench Modport (TB):**
   - Specifies the clocking and reset signals for use in the testbench environment. It exposes the clocking block and reset signal for simulation purposes.

2. **Device Under Test Modport (DUT):**
   - Defines the interface signals required for the operation of the SPI slave module within the device under test. It includes clock, reset, data input, data output, and control signals.

2.2 Interface System Verilog Code

```
interface spi_slave_io #(parameter ADDR_width = 8)(input bit clk) ; //
Synchronus clock sent from the master

  bit rst_n; //Negative Edge reset
  bit mosi;  // master output serial input
  bit miso;  // master input serial output
  bit ss_n;
  bit rx_valid;
  bit tx_valid;
  bit [ADDR_width+1:0] rx_data;
  bit [ADDR_width-1:0] tx_data;

  clocking cb @(posedge clk);

    inout mosi;
    output ss_n;
    inout tx_valid;
    inout tx_data;
    input miso;
    input rx_valid;
    input rx_data;

  endclocking

  modport TB (clocking cb , output rst_n);
  modport DUT (input clk, rst_n, mosi, ss_n, tx_valid, tx_data,output
rx_valid, rx_data, miso);

endinterface
```

3 Program

3.1 Overview

The testbench is designed to verify the functionality of the SPI slave module. It consists of various tasks, assertions, and coverage groups to thoroughly test the behavior of the SPI communication protocol.

3.1.1 Class rand_congf_data:

- This class defines random data members data_sent and data_received used for generating random test stimuli.
- It includes a constraint sending_cmd to ensure that the command bits of data_sent are set to '11' for read data command transmission.

3.1.2 Assertions:

1. **Data Reception Assertion:**

   - Asserts that the received data (intf.cb.rx_data) matches the previously transmitted data (data_sent_temp) after a delay of 3 clock cycles.

2. **Data Transmission Assertion:**

   - Asserts that the transmitted data (intf.cb.tx_data) matches the received data (data_received_temp) after a delay of 11 clock cycles.

3. **Reset Assertions (Intermediate):**

   - validate_reset **Task:**
     This intermediate assertion task verifies the behavior of the SPI slave module after a reset operation.

     1. **MISO Reset Verification:**
        Asserts that the miso signal is reset to '0' after the reset operation.

     2. **RX Valid Reset Verification:**
        Asserts that the rx_valid signal is reset to '0' after the reset operation.

     3. **RX Data Reset Verification:**
        Asserts that the rx_data signal is reset to '0' after the reset operation.

If any of these assertions fail, it indicates a potential issue with the reset functionality.

3.1.3 Coverage Groups:

- Four coverage groups (rx_data_cg, tx_data_cg, tx_valid_cg, rx_valid_cg) are defined to track the coverage of various signals (rx_data, tx_data, tx_valid, rx_valid) during simulation.

3.1.4 Tasks:

1. **validate_reset Task:**

   - Drives a reset signal (intf.rst_n) to reset the SPI slave module and verifies that the reset is successful by checking the states of miso, rx_valid, and rx_data.

2. **validate_spi_transfer Task:**

   - Simulates the SPI data transfer process by driving data on mosi, receiving data on miso, and sampling rx_valid to track data reception.

3. **validate_spi_sending Task:**

- Simulates the SPI data sending process by driving data on miso, transmitting data on mosi, and sampling tx_valid to track data transmission.

3.1.5 Initialization:

- Initializes coverage groups, randomization class, and interface signals.
- Performs random data generation and verifies the SPI data transfer and sending processes.

3.2 Program System Verilog Code

```
program spi_slave_tb (spi_slave_io.TB intf, input bit clk);

   class rand_congf_data;
         rand bit [9:0] data_sent;
         rand bit [7:0] data_received;

      constraint sending_cmd { data_sent[1:0] == 2'b11;
      }

   endclass

   rand_congf_data crand;
    bit [9:0] data_sent_temp;
    bit [7:0] data_received_temp;


    assert property (@(intf.cb) intf.cb.rx_valid |-> (intf.cb.rx_data ==
$past(data_sent_temp,3)));

    assert    property    (@(intf.cb)    intf.cb.tx_valid    |->    ##11
(data_received_temp == intf.cb.tx_data));


    covergroup rx_data_cg ;

      coverpoint intf.cb.rx_data;

    endgroup


    covergroup tx_data_cg ;

      coverpoint intf.cb.tx_data;

    endgroup


    covergroup tx_valid_cg ;

      coverpoint intf.cb.tx_valid;

    endgroup
```

```
covergroup rx_valid_cg ;

    coverpoint intf.cb.rx_valid;

  endgroup

    rx_data_cg cg1 ;
    tx_data_cg cg2 ;
    tx_valid_cg cg3 ;
    rx_valid_cg cg4 ;


task validate_reset ;
  intf.rst_n <= 1;
  #5
  intf.rst_n <= 0;
  #5

  assert (intf.cb.miso == 1'b0)
    $display("MISO is succesfully reseted");
  else
    $error("MISO is not successfully reseted");
  assert (intf.cb.rx_valid == 1'b0)
    $display("rx_valid is succesfully reseted");
  else
    $error("rx_valid is not successfully reseted");
  assert (intf.cb.rx_data == 'b0)
    $display("rx_data is succesfully reseted");
  else
    $error("rx_data is not successfully reseted");

  #5
  intf.rst_n <= 1;

endtask


task validate_spi_transfer(inout bit [9:0] data_sent);

  bit [3:0] counter;
  // Drive chip select low
  intf.cb.ss_n <= 0;

  repeat (10) begin
    @(intf.cb);

    if(intf.cb.rx_valid)
      begin
        cg1.sample();
      end
    cg4.sample();
    intf.cb.mosi <= data_sent[0];
    data_sent[9:0] <= {1'b0,data_sent[9:1]};
    //$display("temp:%h",data_sent_temp);
  end
endtask

 task validate_spi_sending(input bit [7:0] data_received,
                           inout bit [9:0] data_sent,
                              inout bit [7:0] data_out);
  cg3.sample();
  intf.cb.ss_n <= 1'b0;
  intf.cb.tx_valid <= 1'b0;

  // Drive chip select low


  intf.cb.tx_data <= data_received;
  cg2.sample();


   repeat(2) begin
    if(intf.cb.rx_valid)
      begin
        cg1.sample();
      end
    cg4.sample();
    @(intf.cb);
    intf.cb.mosi <= data_sent[0];
    data_out [7:0] <= {intf.cb.miso, data_out[7:1]};
    data_sent[9:0] <= {1'b0,data_sent[9:1]};
    $display("temp:%h",data_sent_temp);

    cg3.sample();
   end


   repeat (8) begin
    @(intf.cb);
    if(intf.cb.rx_valid)
      begin
        cg1.sample();
      end
    cg4.sample();
    intf.cb.mosi <= data_sent[0];
    data_sent[9:0] <= {1'b0,data_sent[9:1]};
    data_out [7:0] <= {intf.cb.miso, data_out[7:1]};
    cg3.sample();
    $display("%h",data_out);
    //$display("temp:%h",data_sent_temp);
  end

endtask


initial
  begin

    cg1 = new();
    cg2 = new();
    cg3 = new();
    cg4 = new();
    crand = new();

    intf.rst_n <= 1'b1;
    intf.cb.ss_n <= 1'b1;
    validate_reset;
    #15

    crand.constraint_mode(0);
    for (int i = 0 ; i <= 130; i++)
      begin
        assert(crand.randomize())
          $display("Randomizatin Successeded: %h",crand.data_sent);
        else
          $error("Randomization Failed");
        data_sent_temp <= crand.data_sent;
        intf.cb.ss_n <= 1'b0;
        validate_spi_transfer(crand.data_sent);
        @(intf.cb);


      end



    crand.constraint_mode(1);
    repeat(2) begin
      @(intf.cb);
    end
    intf.rst_n <= 1'b1;
    intf.cb.ss_n <= 1'b1;
    validate_reset;
    #15

    for (int i = 0 ; i <= 160; i++)
      begin
        assert(crand.randomize())
          $display("Randomizatin  Successeded  sent:  %h,  receive  :
%h",crand.data_sent,crand.data_received);
        else
          $error("Randomization Failed");
        data_sent_temp <= crand.data_sent;
        intf.cb.ss_n <= 1'b1;
        intf.cb.tx_valid <= 1'b1;
        cg3.sample();
        @(intf.cb);

validate_spi_sending(crand.data_received,crand.data_sent,data_received_t
emp);



      end
  end

endprogram
```

4 Top Module

4.1 Overview

This top module serves as the top-level module for the simulation environment. It instantiates the clock generator, SPI slave interface (spi_slave_io), testbench (spi_slave_tb), and the design under test (spi_slave). Additionally, it sets up VCD dumping for waveform analysis during simulation.

4.2 Top Module Verilog Code

```
module top;
  parameter ADDR_width = 8;
  bit clk ;
  always #5 clk = ~clk;

  spi_slave_io i1 (clk);
  spi_slave_tb t1 (i1.TB,clk);
  spi_slave d1 (i1.DUT);


  initial
    begin
      $dumpfile("test.vcd");
      $dumpvars;
      #1000000 $finish;
    end

endmodule
```

5 Waveforms

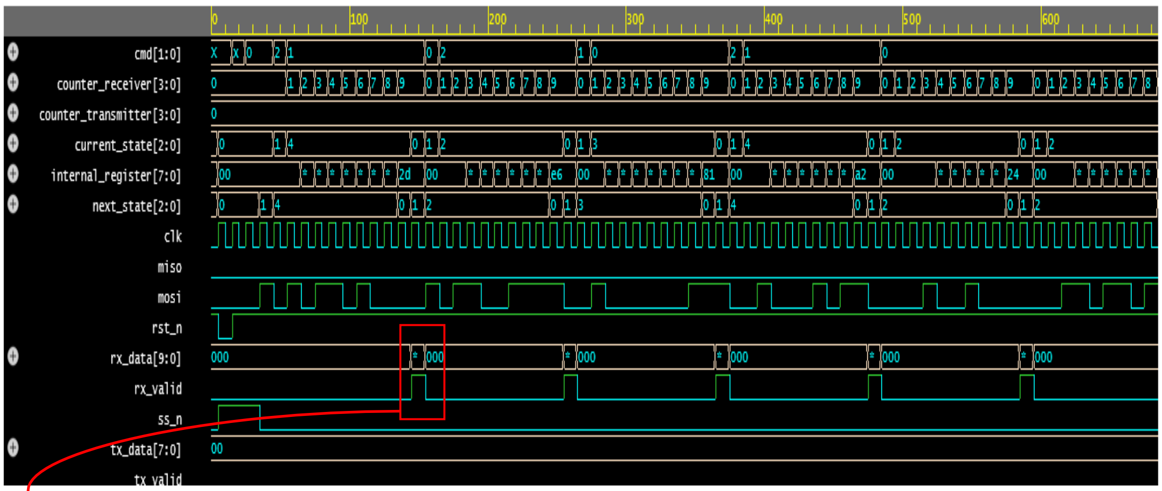


Figure 2: Waveform for 5 succesive receiving runs

Note:

The Highlighted part is the first transmission operation received by the slave where rx_valid is set to high and the received data is ready on the output wires.

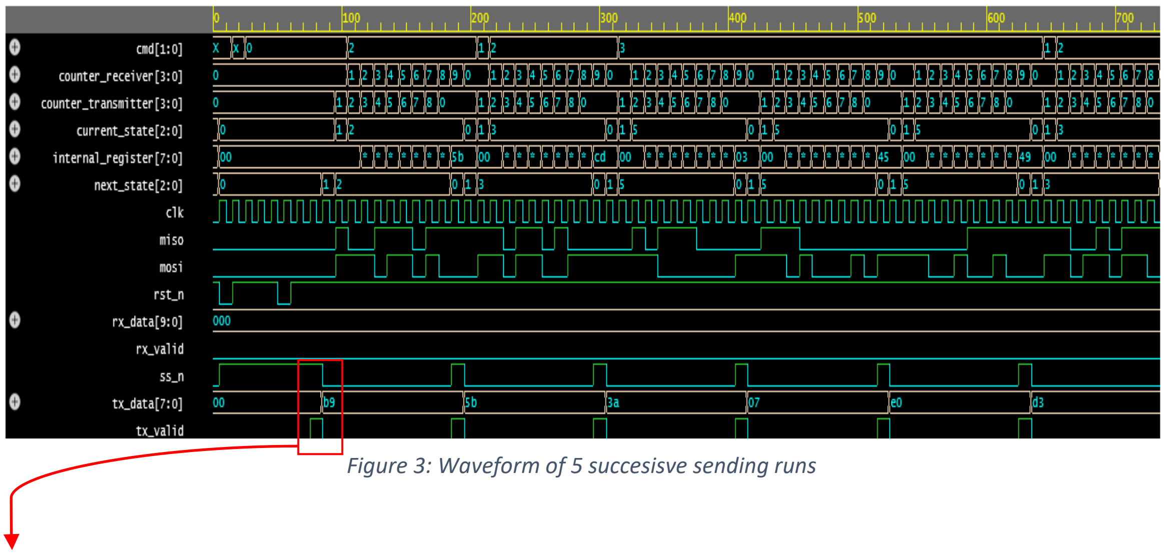


Figure 3: Waveform of 5 succesisve sending runs

Note:

The Highlighted part is the first transmission operation by the slave where tx_valid is set to high for only one cycle and the received data is sent on the MISO wire.

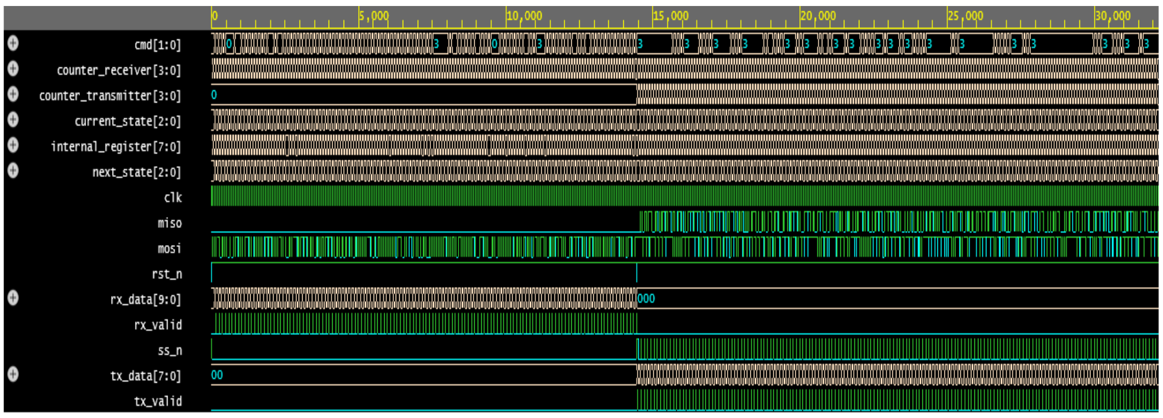
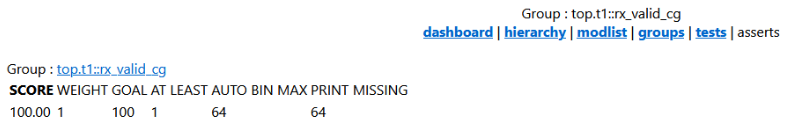


Figure 4: Waveform for all the runs required to reach our coverage goals

6 Coverage Reports

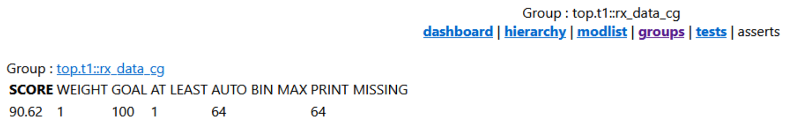
6.1 Coverage Groups



```
                                   Group : top.t1::rx_valid_cg
                        dashboard | hierarchy | modlist | groups | tests | asserts

Group : top.t1::rx_valid_cg
SCORE WEIGHT GOAL AT LEAST AUTO BIN MAX PRINT MISSING
100.00 1       100    1         64            64
```

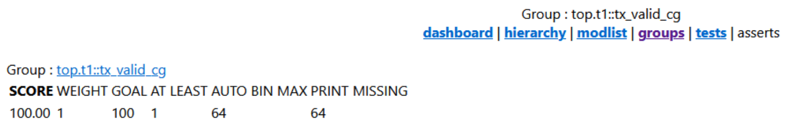
Figure 5: rx_valid coverage group



```
                                   Group : top.t1::rx_data_cg
                        dashboard | hierarchy | modlist | groups | tests | asserts

Group : top.t1::rx_data_cg
SCORE WEIGHT GOAL AT LEAST AUTO BIN MAX PRINT MISSING
90.62   1      100    1         64            64
```

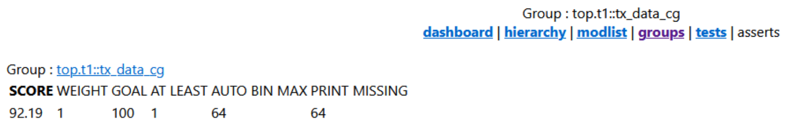
Figure 6: rx_data coverage group



```
                                   Group : top.t1::tx_valid_cg
                        dashboard | hierarchy | modlist | groups | tests | asserts

Group : top.t1::tx_valid_cg
SCORE WEIGHT GOAL AT LEAST AUTO BIN MAX PRINT MISSING
100.00 1       100    1         64            64
```

Figure 7: tx_valid coverage group

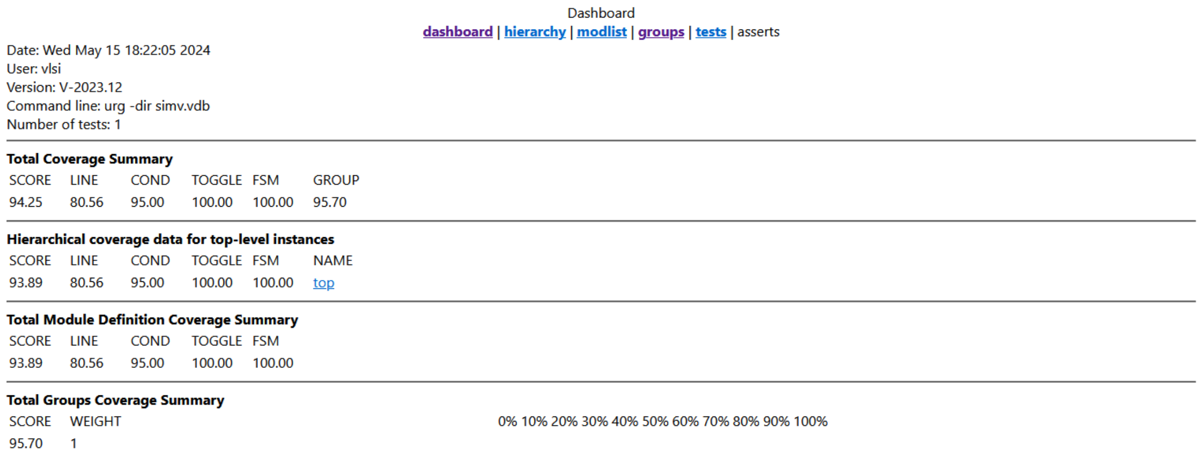


```
                                   Group : top.t1::tx_data_cg
                        dashboard | hierarchy | modlist | groups | tests | asserts

Group : top.t1::tx_data_cg
SCORE WEIGHT GOAL AT LEAST AUTO BIN MAX PRINT MISSING
92.19   1      100    1         64            64
```

Figure 8: tx_data coverage group

6.2 Coverage Dashboard



```
                                        Dashboard
                        dashboard | hierarchy | modlist | groups | tests | asserts
Date: Wed May 15 18:22:05 2024
User: vlsi
Version: V-2023.12
Command line: urg -dir simv.vdb
Number of tests: 1

Total Coverage Summary
SCORE  LINE   COND   TOGGLE  FSM     GROUP
94.25  80.56  95.00  100.00  100.00  95.70

Hierarchical coverage data for top-level instances
SCORE  LINE   COND   TOGGLE  FSM     NAME
93.89  80.56  95.00  100.00  100.00  top

Total Module Definition Coverage Summary
SCORE  LINE   COND   TOGGLE  FSM
93.89  80.56  95.00  100.00  100.00

Total Groups Coverage Summary
SCORE  WEIGHT                                  0% 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
95.70  1
```

Figure 9: Coverage Dashboard