2021

SPI-AVIP

Contents

List of Tables

List of Figures

List of Abbreviations

Abbreviation	Description
uvm	universal verification methodology
spi	serial peripheral interface
avip	accelerated verification intellectual property
hdl	hardware descriptive language
hvl	hardware verification language
bfm	bus functional model
tlm	transaction level modeling
pclk	
sclk	serial clock
cs	chip select
mosi	master out slave in
miso	master in slave out
cpol	clock polarity
cpha	clock phase
lsb	least significant bit
msb	most significant bit
fifo	first in first out
intf	interface
tx	transaction
c2t	
t2c	
wdelay	
tb	testbench

env	environment
cfg	config
mon	monitor
drv	driver
seq	sequence
seqr	sequencer

Chapter 1

Introduction

1.1 Key Features

1. SPI is a Serial Communication Interface.
2. Full Duplex System
3. Configurable shift register, basically of multiple of 2 bits (depends on character length).
4. Programmable SPI clock frequency range
5. Programmable character length (multiples of 8 bits)
6. Programmable clock phase (delay or no delay)
7. Programmable clock polarity (high or low)
8. Single Master - Multiple Slaves.
9. Simple SPI.
10. Single SPI :Single SPI has 4 pins i.e., SCLK,SS/CS,MOSI,MISO.

a.One bit transferred at a single clock cycle.
b.Full-Duplex System.
c.Serial Transfer

11. Configurable shift directions.
12. Delay of chip select low to posedge of sclk, last edge of sclk to raising edge of cs and chip select assert to deassert.
13. Continuous and discontinuous transfer

TODO:

1. Dual SPI
2. Quad SPI
3. Multiples slaves

Chapter 2

Architecture

2.1 SPI AVIP Testbench Architecture

The accelerated VIP has divided into the two top modules as HVL and HDL top as shown in the fig 2.1. The whole idea of using Accelerated VIP is to push the synthesizable part of the testbench into the separate top module along with the interface and it is named as HDL TOP. and the unsynthesizable part is pushed into the HVL TOP it provides the ability to run the longer tests quickly. This particular testbench can be used for the simulation as well as the emulation based on the mode of operation.

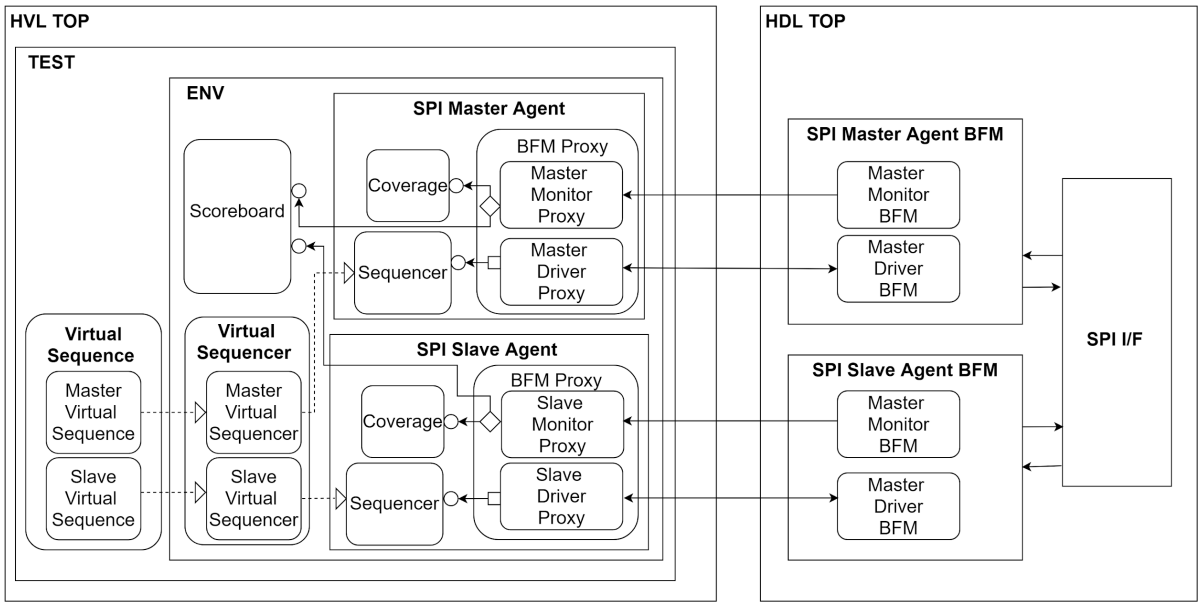


Fig 2.1 SPI_AVIP Architecture

HVL TOP has the design which is untimed and the transactions flow from both master virtual sequence and slave virtual sequence onto the SPI I/F through the BFM Proxy and BFM and gets the data from monitor BFM and uses the data to do checks using scoreboard and coverage.

HDL TOP consists of the design part which is timed and synthesizable, Clock and reset signals are generated in the HDL TOP. Bus Functional Models (BFMs) i.e synthesizable part of drivers and monitors are present in HDL TOP, BFMs also have the back pointers to it's proxy to call non-blocking methods which are defined in the proxy.

We have the tasks and functions within the drivers and monitors which are called by the driver and monitor proxy inside the HVL. This is how the data is transferred between the HVL TOP and HDL TOP.

HDL and HVL uses the transaction based communication to enable the information rich transactions and since clock is generated within the HDL TOP inside the emulator it allows the emulator to run at full speed.

Chapter 3

Implementation

3.1 Pin Interface

Table 3.1 shows the SPI pins used to interface to external devices.

Signal	Master Direction	Slave Direction	Width of the signal	Description
pclk	input	input	1 bit	System generated clock
areset_n	input	input	1 bit	It is an active low reset generated by system
cs_n	input	input	Based on NO_OF_SLAVES parameter	Active low chip select signal is used to select the slave(s). Each bit is for one slave selection.
sclk	input	input	1 bit	Clock signal to synchronize the transfer of data
mosi0	output	input	1 bit	Data which is output of master
mosi1	output	input	1 bit	Data which is output of master
mosi2	output	input	1 bit	Data which is output of master
mosi3	output	input	1 bit	Data which is output of master
miso0	input	output	1 bit	Data which is output of slave 0
miso1	input	output	1 bit	Data which is output of slave 1
miso2	input	output	1 bit	Data which is output of slave 2
miso3	input	output	1 bit	Data which is output of slave 3

For more details refer - spi_avip_verification_plan.xlsx

3.2 Testbench Components

In this section, testbench components of the spi-avip are discussed

3.2.1 Spi Hdl Top

Hdl top is synthesizable, where generation of the clock and reset is done. Instantiation of the spi interface handle, master agent bfm handle and slave agent bfm handle is done as shown in Figure 3.1.

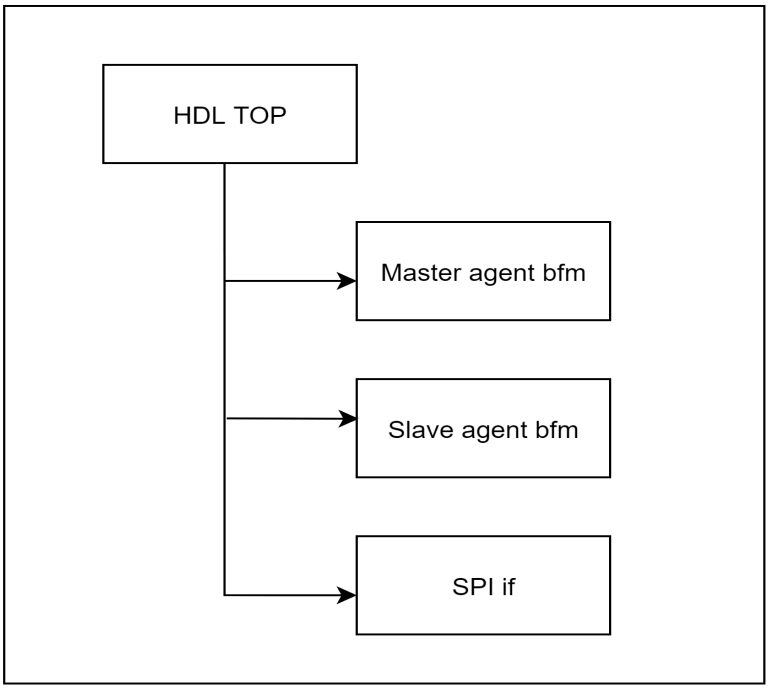


Fig. 3.1 HDL Top

3.2.2 SPI Interface

Importing the global packages
Passing Signals: pclk, areset
Declaration of signals: sclk, cs[], mois0, miso0, mosi1, miso1, mosi2, miso2, mosi3, miso3 are declared as logic type.

3.2.3 SPI Master Agent BFM Module

Instantiates the below two interfaces here

a) spi master driver bfm and
b) spi master monitor bfm.

Instantiates the spi master assertions and binds it with the spi master monitor bfm handle and maps the signals of spi master assertions with the spi interface signals. The spi interface signals are passed to the spi master driver and monitor bfm in instantiations.

```
master_driver_bfm master_drv_bfm_h (.pclk(intf.pclk),
                                    .areset(intf.areset),
                                    .sclk(intf.sclk),
                                    .cs(intf.cs),
                                    .mosi0(intf.mosi0),
                                    .mosi1(intf.mosi1),
                                    .mosi2(intf.mosi2),
                                    .mosi3(intf.mosi3),
                                    .miso0(intf.miso0),
                                    .miso1(intf.miso1),
                                    .miso2(intf.miso2),
                                    .miso3(intf.miso3)
                                   );
```

Fig. 3.2 Spi driver bfm instantiation in spi master agent bfm code snippet

Fig. 3.2 and 3.3 are the code snippets of instantiations of spi master driver and monitor bfm

```
master_monitor_bfm master_mon_bfm_h (.pclk(intf.pclk),
                                     .areset(intf.areset),
                                     .sclk(intf.sclk),
                                     .cs(intf.cs),
                                     .mosi0(intf.mosi0),
                                     .mosi1(intf.mosi1),
                                     .mosi2(intf.mosi2),
                                     .mosi3(intf.mosi3),
                                     .miso0(intf.miso0),
                                     .miso1(intf.miso1),
                                     .miso2(intf.miso2),
                                     .miso3(intf.miso3)
                                    );
```

Fig. 3.3 Spi monitor bfm instantiation in spi master agent bfm code snippet

3.2.4 SPI Master Driver BFM Interface

Spi master driver bfm is an interface where it will get the signals from the spi interface. It has a method drive_to_bfm(data_packet, configuration packet) which will be called by the spi master driver proxy which drives the mosi data to the spi interface. fig.3.2 gives the reference of the instantiation of spi master driver bfm.

3.2.5 SPI Master Monitor BFM Interface

Spi master monitor bfm is an interface where it will get the signals from the spi interface. It has a method sample_data(data_packet, configuration packet) which will be called by the spi master monitor proxy which samples the mosi and miso data from the spi interface. After sampling the data, the spi master monitor bfm interface sends the data to the spi master monitor proxy using the output port of sample_data task. fig.3.3 gives the reference of the instantiation of spi master monitor bfm.

3.2.6 SPI Slave Agent BFM Module

Instantiates the below two interfaces here

1. spi slave driver bfm and
2. spi slave monitor bfm.

Instantiates the spi slave assertions and binds it with the spi slave monitor bfm handle and maps the signals of spi slave assertions with the spi interface signals. The spi interface signals are passed to the spi slave driver and monitor bfm in instantiations

```
slave_driver_bfm slave_drv_bfm_h (.pclk(intf.pclk),
                                  .areset(intf.areset),
                                  .sclk(intf.sclk),
                                  .cs(intf.cs[NO_OF_SLAVES_1]),
                                  .mosi0(intf.mosi0),
                                  .mosi1(intf.mosi1),
                                  .mosi2(intf.mosi2),
                                  .mosi3(intf.mosi3),
                                  .miso0(intf.miso0),
                                  .miso1(intf.miso1),
                                  .miso2(intf.miso2),
                                  .miso3(intf.miso3)
                                 );
```

Fig 3.4 Spi slave driver bfm instantiation in spi slave agent bfm code snippet

```
slave_monitor_bfm slave_mon_bfm_h (.pclk(intf.pclk),
                                   .areset(intf.areset),
                                   .sclk(intf.sclk),
                                   .cs(intf.cs[NO_OF_SLAVES-1]),
                                   .mosi0(intf.mosi0),
                                   .mosi1(intf.mosi1),
                                   .mosi2(intf.mosi2),
                                   .mosi3(intf.mosi3),
                                   .miso0(intf.miso0),
                                   .miso1(intf.miso1),
                                   .miso2(intf.miso2),
                                   .miso3(intf.miso3)
                                  );
```

Fig 3.5 Spi slave monitor bfm instantiation in spi slave agent bfm code snippet

3.2.7 SPI Slave Driver BFM Interface

Spi slave driver bfm is an interface where it will get the signals from the spi interface. It has a method drive_to_bfm(data_packet, configuration packet) which will be called by the spi slave driver proxy which drives the mosi data to the spi interface. fig.3.4 gives the reference for the instantiation of spi slave driver bfm.

3.2.8 SPI Slave Monitor BFM Interface

Spi slave monitor bfm is an interface where it will get the signals from the spi interface. It has a method sample_data(data_packet, configuration packet) which will be called by the spi slave monitor proxy which samples the mosi and miso data from the spi interface. After sampling the data, the spi slave monitor bfm interface sends the data to the spi slave monitor proxy using the output port of sample_data task. fig.3.5 gives the reference for the instantiation of spi slave monitor bfm.

3.2.9 SPI HVL_TOP

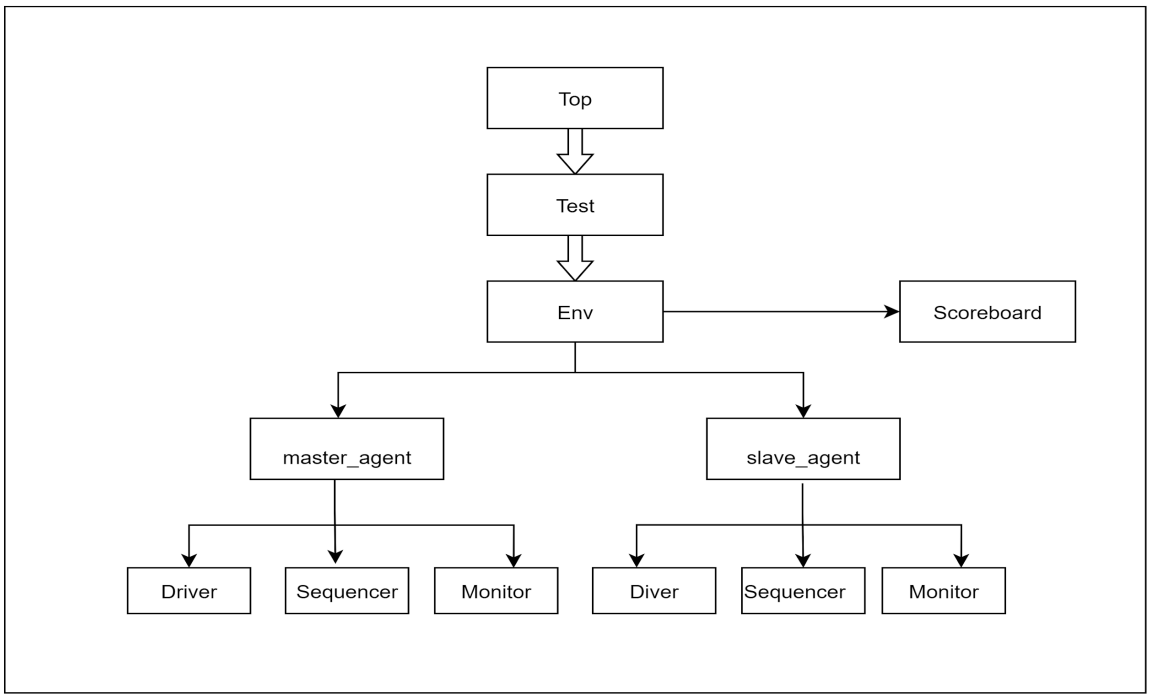


Fig 3.6 HVL Top

In top we are running the test by using the **run_test("test_name")** method , which will start the whole tb components.

3.2.10 SPI Environment

Environment has the below components

a. spi_scoreboard
b. spi_virtual_sequencer
c. spi_master_agent
d. spi_slave_agent

In the build phase, env_cfg handle will be called and create the memory for the above declared components.

In the connect phase, the spi_master_monitor_proxy is connected to spi_scoreboard and spi_slave_monitor_proxy to spi_scoreboard using analysis port and analysis fifo as shown in fig 3.7.

3.2.11 SPI Scoreboard

A scoreboard is a verification component that contains checkers and verifies the functionality of a design. The scoreboard is implemented by extending uvm_ scoreboard.

The purpose of the scoreboard in the SPI-AVIP project is to

1. Compare the MOSI and MISO data from the slave and master
2. Keep track of pass and failure rates identified in the comparison process
3. Report comparison success/failures result at the end of the simulation

The scoreboard consists of two analysis fifo's which receive the packets from the analysis port of the monitor class. fig. 3.7 shows the connection between the analysis port and analysis fifo.

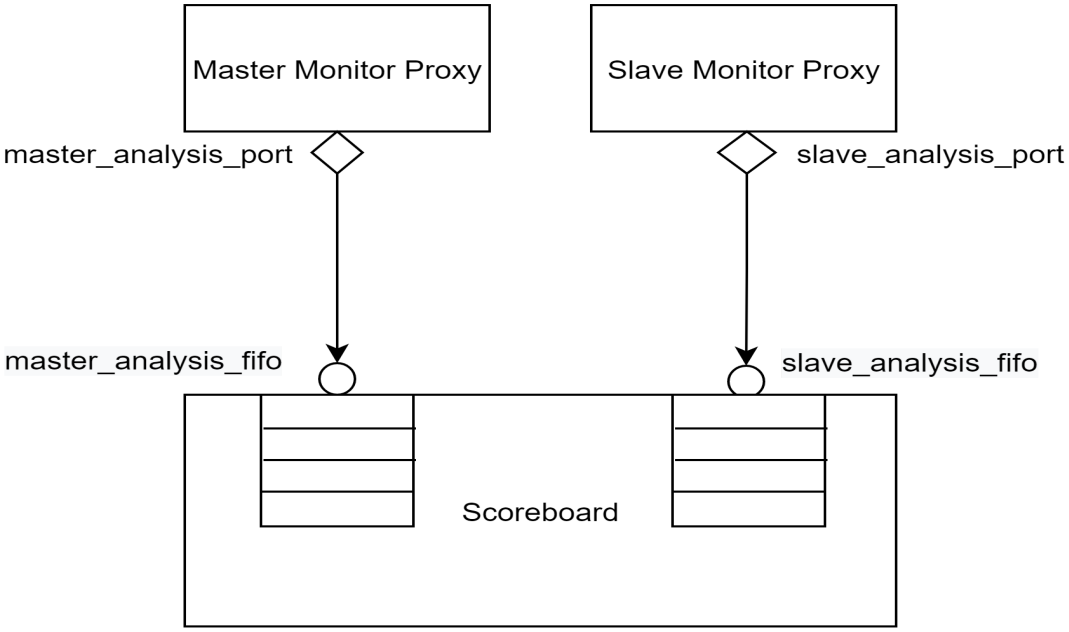


Fig 3.7 connection of the analysis ports of the monitor to the scoreboard analysis fifo

In the monitor proxy class of master and slave, two analysis ports are declared. Fig 3.8 shows the declaration of master analysis port and slave analysis port in the master monitor proxy and slave monitor proxy.

```
uvm_analysis_port #(master_tx) master_analysis_port;
uvm_analysis_port #(slave_tx) slave_analysis_port;
```

Fig 3.8 shows the declaration of slave and master analysis port in the slave and master monitor proxy

In the scoreboard, two analysis fifo's are declared. Fig 3.9 shows the declaration of master analysis fiifo and slave analysis fifo in the scoreboard.

```
uvm_tlm_analysis_fifo#(master_tx)master_analysis_fifo;

uvm_tlm_analysis_fifo#(slave_tx)slave_analysis_fifo;
```

Fig 3.9 shows the declaration of master and slave analysis fifo in the scoreboard

In the constructor, create objects for the two declared analysis fifo's. Fig 3.10 shows the creation of the master and slave analysis port.

```
function spi_scoreboard::new(string name = "spi_scoreboard", uvm_component parent = null);
  super.new(name, parent);
  master_analysis_fifo = new("master_analysis_fifo",this);
  slave_analysis_fifo = new("slave_analysis_fifo",this);
endfunction : new
```

Fig 3.10 shows the creation of the master and slave analysis port

In connect phase of the environment class, the analysis port of both master and slave monitor proxy class is connected to the analysis export of the master and slave fifo in the scoreboard. Fig 3.11 shows the connection made between the monitor analysis port and the scoreboard fifo's in the connect phase of the env class.

```
foreach(slave_agent_h[i]) begin
slave_agent_h[i].slave_mon_proxy_h.slave_analysis_port.connect
                (scoreboard_h.slave_analysis_fifo.analysis_export);
end
master_agent_h.master_mon_proxy_h.master_analysis_port.connect
                (scoreboard_h.master_analysis_fifo.analysis_export);
```

Fig 3.11 Connection done between the analysis port and analysis fifo exportin the env class

In the run phase of the scoreboard, the get() method is used to get the data packet from the monitor write() method. Fig 3.12 shows the use of the get() method to get the transaction from the monitor analysis port.

```
task spi_scoreboard::run_phase(uvm_phase phase);
  super.run_phase(phase);
  forever begin
     master_analysis_fifo.get(master_tx_h);
     master_tx_count++;

     slave_analysis_fifo.get(slave_tx_h);
     slave_tx_count++;
     ...
  end
```

Fig 3.12 Use of get method to get the packet from monitor analysis port

The Comparison of the mosi data and miso data from the master monitor and slave monitor is done in the run phase. Fig 3.13 shows the comparison of the master mosi data with slave mosi data.

```
foreach(slave_tx_h.master_in_slave_out[i]) begin
    if(slave_tx_h.master_in_slave_out[i] != master_tx_h.master_in_slave_out[i]) begin
      `uvm_error("ERROR_SC_MISO_DATA_MISMATCH",
                $sformatf("Slave MISO[%0d] = 'h%0x and Master MISO[%0d] = 'h%0x",
                          i, slave_tx_h.master_in_slave_out[i],
                          i, master_tx_h.master_in_slave_out[i]) );
      byte_data_cmp_failed_miso_count++;
    end
    else begin
      `uvm_info("SB_MISO_DATA_MATCH",
                $sformatf("Slave MISO[%0d] = 'h%0x and Master MISO[%0d] = 'h%0x",
                          i, slave_tx_h.master_in_slave_out[i],
                          i, master_tx_h.master_in_slave_out[i]), UVM_HIGH);
      byte_data_cmp_verified_miso_count++;
    end
  end
```

Fig 3.13 The comparison of the master mosi data with slave mosi data

Similarly, the comparison is done for the miso data as well.

Fig 3.14 explains the flow chart of the run phase in the scoreboard.

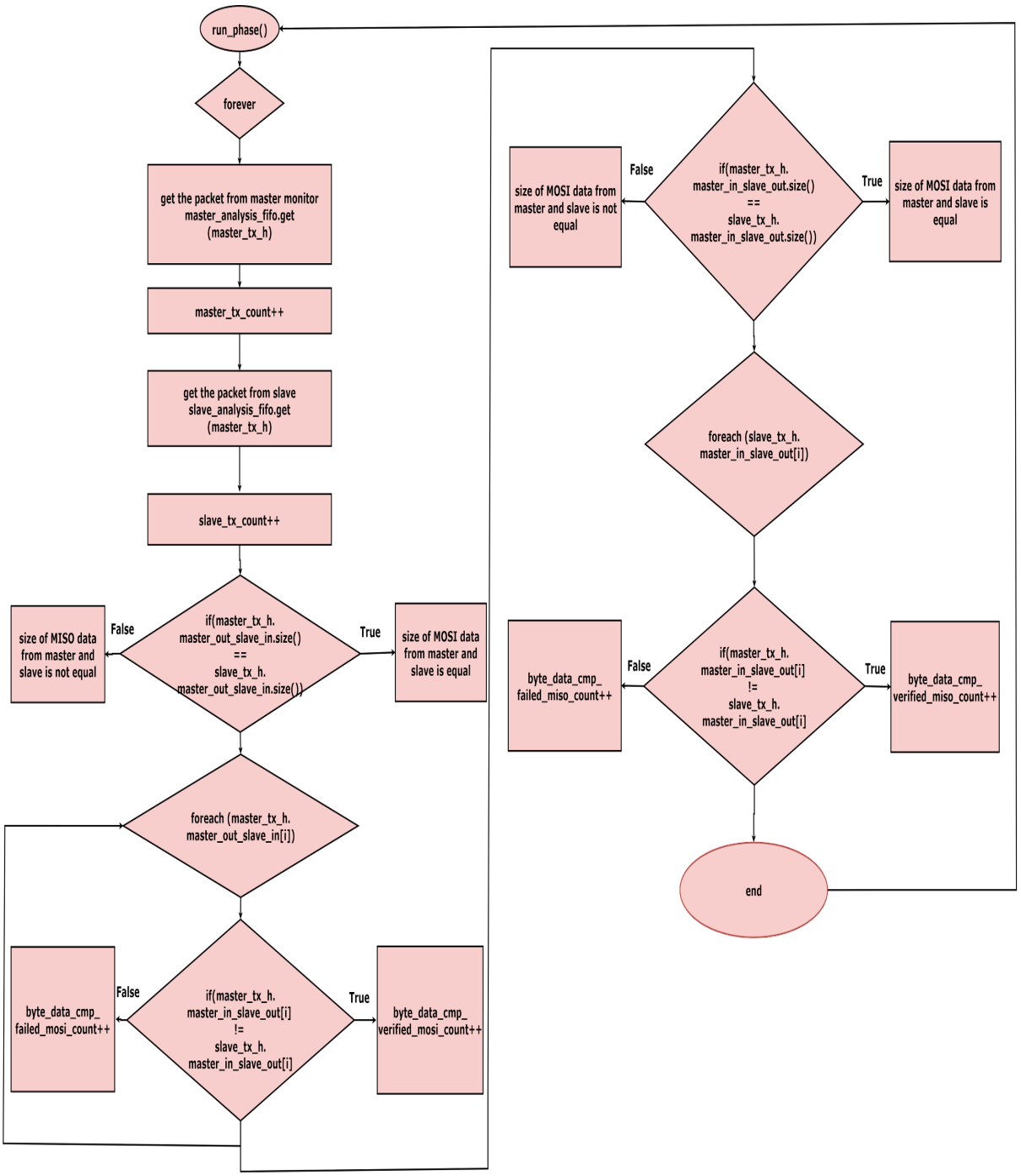


Fig 3.14 Flow chart of the scoreboard run phase

In the run phase, inside the forever loop, the scoreboard master analysis fifo gets the transaction from the master monitor analysis port using the get() method. Whenever the packet is received the transaction counter i.e, master_tx_count will be incremented. Here the master_tx_count will be incremented whenever the fifo gets the packets. The array length of the transaction from master and slave is compared to verify whether the size of both the mosi data is equal or not. For each of the mosi data received by the master analysis fifo will be compared with the mosi data received by the analysis fifo of slaves. Whenever the mosi comparison is true, byte_data_cmp_verified_mosi_count will be incremented and if the

comparison fails, byte_data_cmp_failed_mosi_count will be incremented. Similarly, the slave analysis fifo gets the data, whenever fifo receives the packets, slave_tx_count counter is incremented to indicate number of transaction and then compares the size of the miso data from both master and slave monitor and display whether the miso data size is equal or not. The miso data from the master analysis port and from slave analysis port are compared, if the comparisios is true the counter byte_data_cmp_verified_miso_count will be incremented if the comparison fails byte_data_cmp_failed_count will be incremented.

After the run phase, the next is the check phase.In check Phase of the scoreboard, with the help of counter variables we verify the following

a. Whether all mosi and miso comparisons are successful
b. Whether all transactions from master and slave monitors are equal
c. Whether both FIFO's are empty or not

Fig 3.15 is the flow chart of the scoreboard report phase, which explains checks made to identify the success/failure rates.

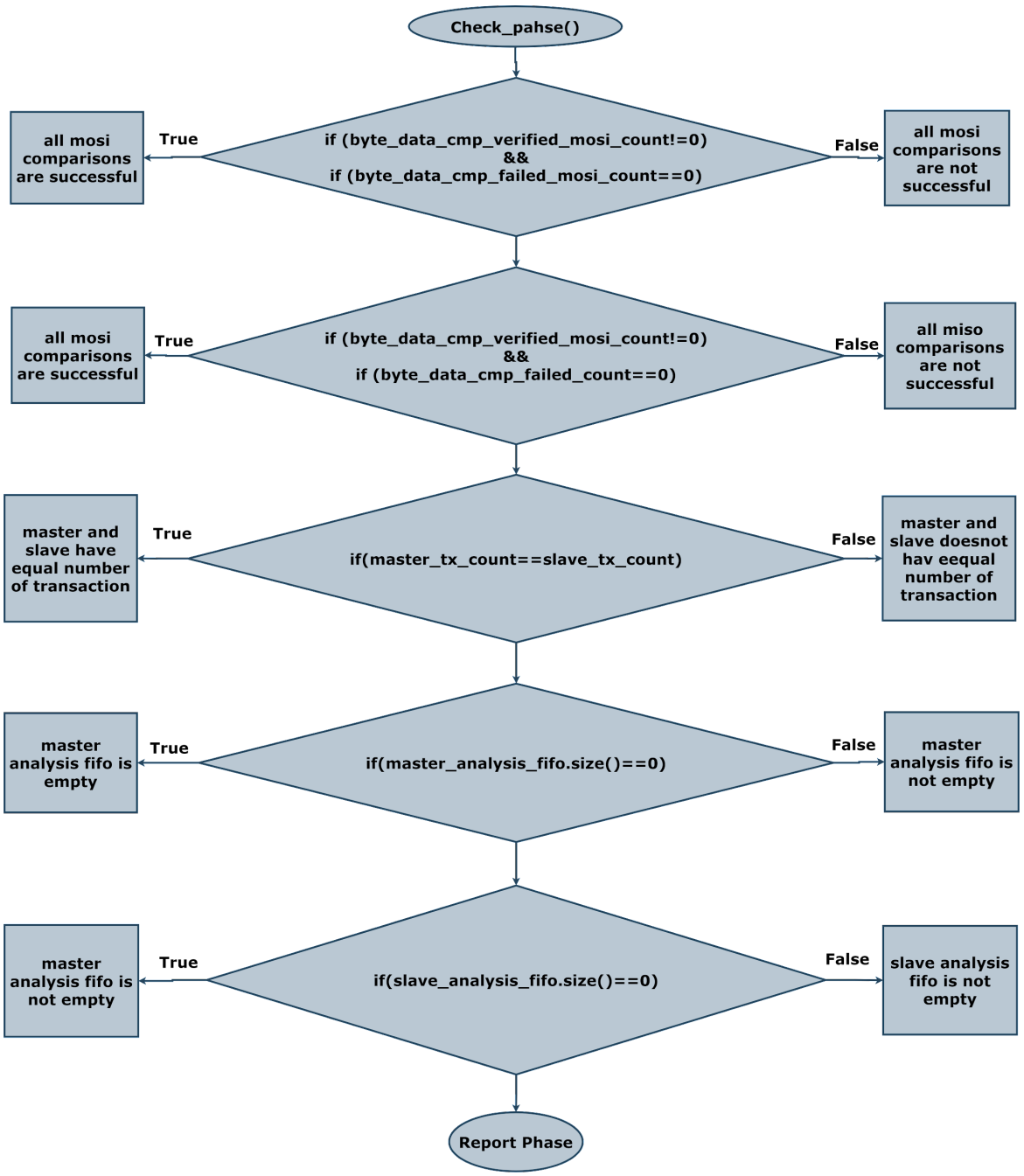


Fig 3.15 Flow chart of the scoreboard report phase

After the completion of the run phase, the check phase is executed, in the check phase we check if all the mosi data verified count is non-zero and the mosi data failed count is zero. If this condition is passed then we say, all comparisons are successful even if one comparison fails we display all comparisons are not successful. Next, master transaction counts and slave transaction counts are compared to verify whether the number of packets are equal or not. After all comparisons done the fifo's should be empty, so the master analysis fifo size and the slave analysis fifo is compared to zero, if equal then the fifo are empty or else fifo are not empty. In the report phase, all the counter variables along with the information of the checks are displayed.

3.2.12 SPI Virtual Sequencer

In virtual sequencer we are declaring the handles for environment_configuration, master_sequencer and slave_sequencer. In the build phase we are getting environment_configuration and creating the memory for master_sequencer and slave_sequencer.

3.2.13 SPI Master Agent

Spi master agent component is a class extending from uvm_agent. It gets the spi master agent config handle and based on that we will create and connect the components. It creates the spi master sequencer and spi master driver only if the spi master agent is active which will depend on the value of is_active variable declared in the spi master agent configuration file. The spi master coverage is created in build_phase if has_covergae variable is 1 which is declared in the spi master agent configuration file. Please refer to figure 3.16 for the spi master agent build_phase code snippet. For more information about the spi master agent configuration, please refer to Chapter 5.2

Spi master agent build phase has creation of,

a. spi master sequencer
b. spi master driver proxy
c. spi master monitor proxy
d. spi master coverage components.

```
function void master_agent::build_phase(uvm_phase phase);
  super.build_phase(phase);

  if(!uvm_config_db #(master_agent_config)::get(this,"","master_agent_config",master_agent_cfg_h)) begin
    `uvm_fatal("FATAL_MA_CANNOT_GET_MASTER_AGENT_CONFIG","cannot get master_agent_cfg_h from uvm_config_db");
  end

    if(master_agent_cfg_h.is_active == UVM_ACTIVE) begin
    master_drv_proxy_h=master_driver_proxy::type_id::create("master_drv_proxy_h",this);
    master_seqr_h=master_sequencer::type_id::create("master_seqr_h",this);
  end

  master_mon_proxy_h=master_monitor_proxy::type_id::create("master_mon_proxy_h",this);

  if(master_agent_cfg_h.has_coverage) begin
    master_cov_h = master_coverage::type_id::create("master_cov_h",this);
  end
endfunction : build_phase
```

Fig 3.16 Spi master agent build phase code snippet

Spi master agent configuration handles declared in the above created components will be mapped here in the connect phase. The spi master driver proxy and spi master sequencer are connected using TLM ports if the spi master agent is active. The spi master coverage's analysis_export will be connected to spi master monitor proxy's master_analysis_port in connect_phase.

```
function void master_agent::connect_phase(uvm_phase phase);
  if(master_agent_cfg_h.is_active == UVM_ACTIVE) begin
    master_drv_proxy_h.master_agent_cfg_h = master_agent_cfg_h;
    master_seqr_h.master_agent_cfg_h = master_agent_cfg_h;

    //Connecting the ports
    master_drv_proxy_h.seq_item_port.connect(master_seqr_h.seq_item_export);
  end

  if(master_agent_cfg_h.has_coverage) begin
        master_cov_h.master_agent_cfg_h = master_agent_cfg_h;

    master_mon_proxy_h.master_analysis_port.connect(master_cov_h.analysis_export);
  end

  master_mon_proxy_h.master_agent_cfg_h = master_agent_cfg_h;

endfunction: connect_phase
```

Fig 3.17 Spi master agent connect phase code snippet

3.2.14 SPI Master Sequencer

Spi master sequencer component is a parameterised class of type spi master transaction, extending uvm_sequencer. Spi sequencer sends the data from the spi master sequences to the spi driver proxy.

3.2.15 SPI Master Driver Proxy

Spi master driver proxy component is a parameterised class of type spi master transaction, extending uvm_driver. It gets the spi master agent config handle and based on the configurations we will drive and sample the mosi and miso signals respectively. It gets the master transaction into the spi driver proxy using get_next_item() method.

As the spi driver bfm interface cannot access the class based spi master transaction data, so we have to convert that into struct data type. Similarly, it converts the spi master configuration values into struct data type. Spi master driver proxy will call the converter class to convert the master transaction packet and master configuration packet into struct data packet and struct configuration packet respectively(declared in spi global package) and then will pass it to spi master driver bfm using drive to bfm method declared in spi master driver bfm. The drive to bfm method starts the drive_msb_first_pos_edge(data_packet, configuration packet) which is declared in spi driver bfm.

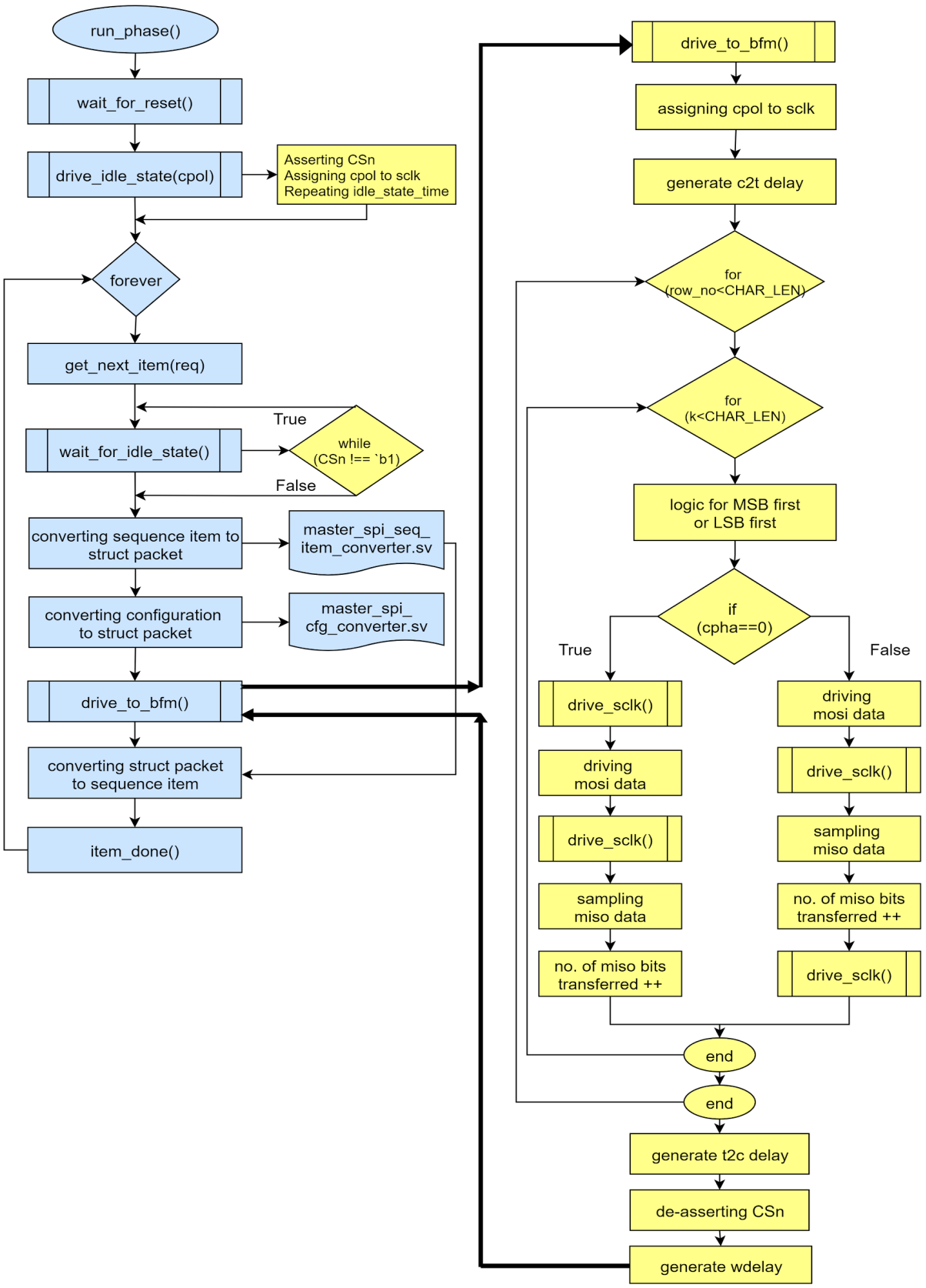


Fig 3.18 Flowchart of communication between master driver proxy and master driver bfm

```
task master_driver_proxy::run_phase(uvm_phase phase);
  bit cpol, cpha;
  super.run_phase(phase);
    {cpol, cpha} = operation_modes_e'(master_agent_cfg_h.spi_mode);
   master_drv_bfm_h.wait_for_reset();
   master_drv_bfm_h.drive_idle_state(cpol);
   forever begin
    spi_transfer_char_s struc_packet;
    spi_transfer_cfg_s struct_cfg;
    seq_item_port.get_next_item(req);
    master_drv_bfm_h.wait_for_idle_state();
    master_spi_seq_item_converter::from_class(req, struc_packet);
    master_spi_cfg_converter::from_class(master_agent_cfg_h, struct_cfg);
    `uvm_info(get_type_name(),$sformatf("STRUCT PACKET : , \n %p",
    struc_packet.master_out_slave_in[0]),UVM_LOW);
    `uvm_info(get_type_name(),$sformatf("STRUCT PACKET : , \n %p",
    struc_packet.master_out_slave_in[1]),UVM_LOW);
    `uvm_info(get_type_name(),$sformatf("STRUCT CONFIGURATION : , \n %p",struct_cfg),UVM_LOW);
    drive_to_bfm(struc_packet, struct_cfg);
    master_spi_seq_item_converter::to_class(struc_packet, req);
    `uvm_info(get_type_name(),$sformatf("Received packet from BFM : , \n %s",     req.sprint()),UVM_LOW)
    seq_item_port.item_done();
  end
endtask : run_phase
```

Fig 3.19 run phase of spi master driver proxy code snippet

3.2.16 SPI Master Monitor Proxy

Spi master monitor proxy component is a class extending uvm_monitor. It gets the spi master agent config handle and based on the configurations we will sample the mosi and miso signals. It declares and creates the spi master analysis port to send the sampled data.

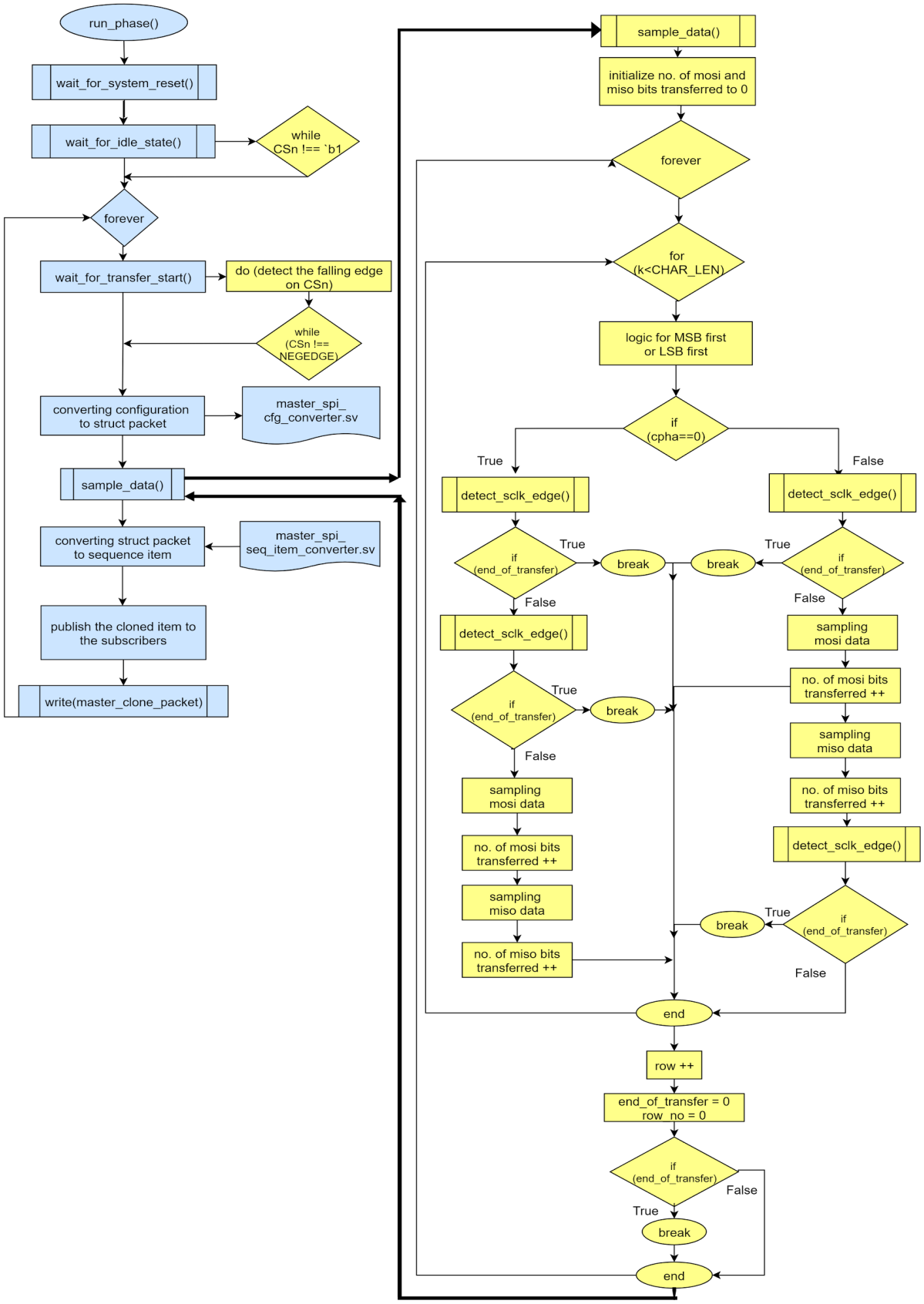


Fig 3.20 Flowchart of spi master monitor proxy and spi master monitor bfm communication

```
task master_monitor_proxy::run_phase(uvm_phase phase);
  master_tx master_packet;
  `uvm_info(get_type_name(), $sformatf("Inside the master_monitor_proxy"), UVM_LOW);
  master_packet = master_tx::type_id::create("master_packet");
  master_mon_bfm_h.wait_for_system_reset();
  master_mon_bfm_h.wait_for_idle_state();

  forever begin
    spi_transfer_char_s struct_packet;
    spi_transfer_cfg_s struct_cfg;
    master_tx master_clone_packet;
    master_mon_bfm_h.wait_for_transfer_start();
    master_spi_cfg_converter::from_class(master_agent_cfg_h, struct_cfg);
    master_mon_bfm_h.sample_data(struct_packet, struct_cfg);
    master_spi_seq_item_converter::to_class(struct_packet, master_packet);
    `uvm_info(get_type_name(),$sformatf("Received packet from BFM : , \n %s",
                                        master_packet.sprint()),UVM_HIGH)
    $cast(master_clone_packet, master_packet.clone());
    `uvm_info(get_type_name(),$sformatf("Sending packet via analysis_port : , \n %s",
                                        master_clone_packet.sprint()),UVM_HIGH)
    master_analysis_port.write(master_clone_packet);
  end
endtask : run_phase
```

Fig 3.21 Spi master monitor proxy run phase code snippet

3.2.17 SPI Slave Agent

Spi slave agent component is a class extending uvm_agent. It gets the spi slave agent configuration and based on that we will create and connect the components. It creates the spi slave sequencer and spi slave driver only if the spi slave agent is active which will depend on the value of is_active variable declared in the spi slave agent configuration file. The spi slave coverage is created in build_phase if has_covergae variable is 1 which is declared in the spi slave agent configuration file. For more information about spi master agent configuration, please refer to Chapter 5.3

Spi slave agent build phase has creation of,

a. spi slave sequencer
b. spi slave driver proxy
c. spi slave monitor proxy
d. spi slave coverage components.

```
function void slave_agent::build_phase(uvm_phase phase);
  super.build_phase(phase);

  if(!uvm_config_db #(slave_agent_config)::get(this,"","slave_agent_config",slave_agent_cfg_h)) begin
   `uvm_fatal("FATAL_SA_AGENT_CONFIG", $sformatf("Couldn't get the slave_agent_config from config_db"))
  end

   if(slave_agent_cfg_h.is_active == UVM_ACTIVE) begin
     slave_drv_proxy_h = slave_driver_proxy::type_id::create("slave_drv_proxy_h",this);
     slave_seqr_h=slave_sequencer::type_id::create("slave_seqr_h",this);
   end

   slave_mon_proxy_h = slave_monitor_proxy::type_id::create("slave_mon_proxy_h",this);

   if(slave_agent_cfg_h.has_coverage) begin
    slave_cov_h = slave_coverage::type_id::create("slave_cov_h",this);
   end


endfunction : build_phase
```

Fig 3.22 Spi slave agent build phase code snippet

Spi slave agent configuration handles declared in the above created components will be mapped here in the connect phase. The spi slave driver proxy and spi slave sequencer is connected using tlm ports if the spi slave agent is active. The spi slave coverage's analysis_export will be connected to spi slave monitor proxy's slave_analysis_port in connect_phase.

```
function void slave_agent::connect_phase(uvm_phase phase);
  super.connect_phase(phase);

  if(slave_agent_cfg_h.is_active == UVM_ACTIVE) begin
    slave_drv_proxy_h.slave_agent_cfg_h = slave_agent_cfg_h;
    slave_seqr_h.slave_agent_cfg_h = slave_agent_cfg_h;
    slave_cov_h.slave_agent_cfg_h = slave_agent_cfg_h;

    // Connecting the ports
    slave_drv_proxy_h.seq_item_port.connect(slave_seqr_h.seq_item_export);
  end

    if(slave_agent_cfg_h.has_coverage)begin
      slave_cov_h.slave_agent_cfg_h=slave_agent_cfg_h;
      slave_mon_proxy_h.slave_analysis_port.connect(slave_cov_h.analysis_export);
   end

  slave_mon_proxy_h.slave_agent_cfg_h = slave_agent_cfg_h;


endfunction: connect_phase
```

Fig 3.23 Spi slave agent connect phase code snippet

3.2.18 SPI Slave Sequencer

Spi slave sequencer component is a parameterised class of type spi slave transaction, extending uvm_sequencer. Spi sequencer sends the data from the spi slave sequences to the spi driver proxy.

3.2.19 SPI Slave Driver Proxy

Spi slave driver proxy component is a parameterised class of type spi slave transaction, extending uvm_driver. It gets the spi slave agent config handle and based on the configurations we will drive and sample the mosi and miso signals respectively. It gets the slave transaction into the spi driver proxy using get_next_item() method.

As the spi driver bfm interface cannot access the class based spi slave transaction data, so we have to convert that into struct data type. Similarly, it converts the spi slave configuration values into struct data type. Spi slave driver proxy will call the converter class to convert the slave transaction packet and slave configuration packet into struct data packet and struct configuration packet respectively(declared in spi global package) and then will pass it to spi slave driver bfm using drive to bfm method declared in spi slave driver bfm. The drive to bfm method starts the drive_to_bfm (data_packet, configuration packet) which is declared in spi driver bfm.

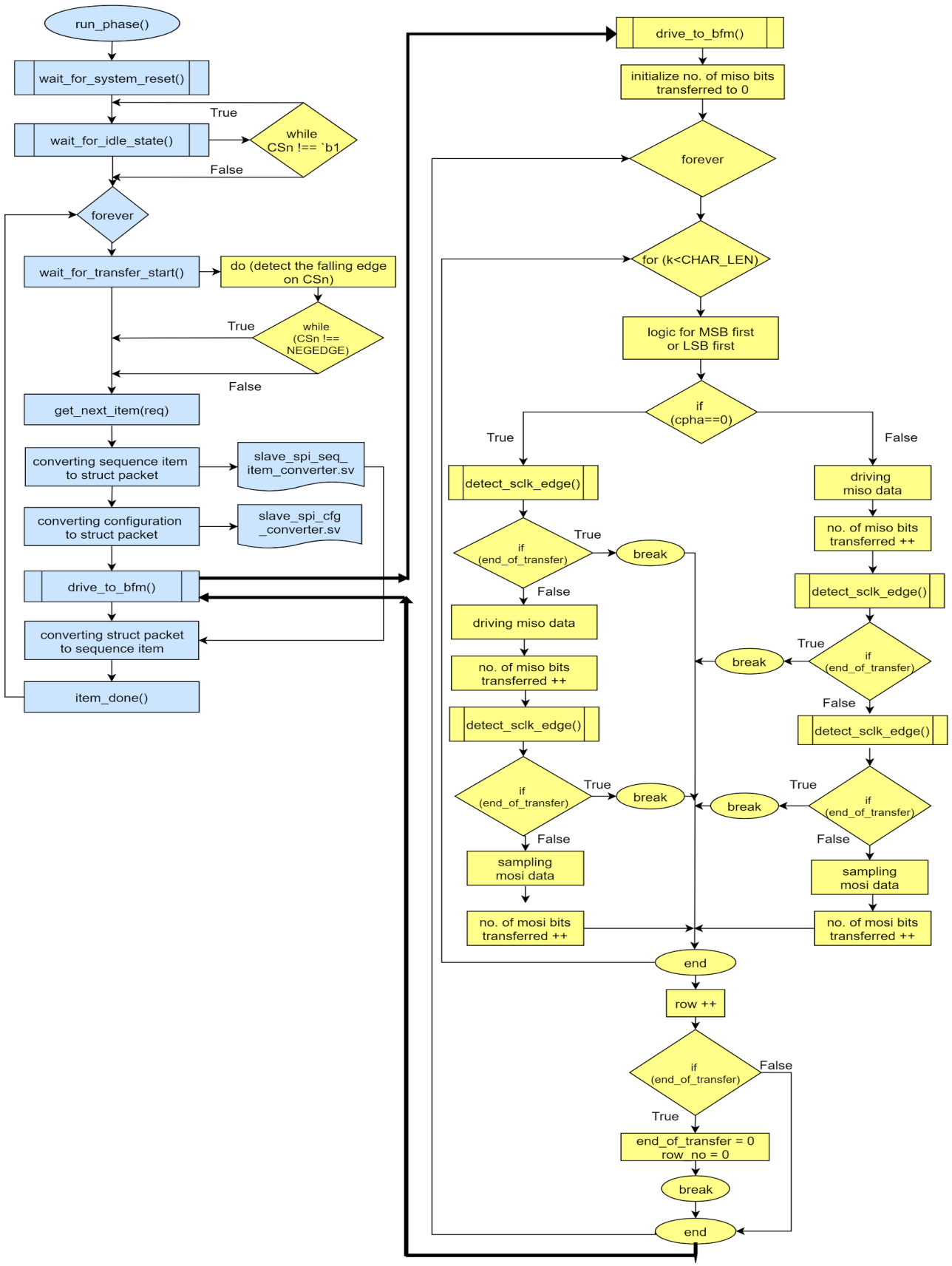


Fig 3.24 Flowchart of spi slave driver bfm and slave driver proxy communication

```
task slave_driver_proxy::run_phase(uvm_phase phase);

  super.run_phase(phase);
  slave_drv_bfm_h.wait_for_system_reset();
  slave_drv_bfm_h.wait_for_idle_state();
    spi_transfer_char_s struct_packet;
    spi_transfer_cfg_s struct_cfg;

    slave_drv_bfm_h.wait_for_transfer_start();

    seq_item_port.get_next_item(req);
    `uvm_info(get_type_name(),$sformatf("Received packet from slave seqeuncer : , \n %s",
                                        req.sprint()),UVM_LOW)

    slave_spi_seq_item_converter::from_class(req, struct_packet);
    slave_spi_cfg_converter::from_class(slave_agent_cfg_h, struct_cfg);
    drive_to_bfm(struct_packet, struct_cfg);
    slave_spi_seq_item_converter::to_class(struct_packet, req);
    `uvm_info(get_type_name(),$sformatf("Received packet from BFM : , \n %s",
                                        req.sprint()),UVM_LOW)
    seq_item_port.item_done();
  end
endtask : run_phase
```

Fig 3.25 Spi slave driver proxy build phase code snippet

3.2.20 SPI Slave Monitor Proxy

Spi slave monitor proxy component is a class extending uvm_monitor. It gets the spi slave agent config handle and based on the configurations we will sample the mosi and miso signals. It declares and creates the spi slave analysis port to send the sampled data. The spi slave monitor proxy will get the sampled data from spi master monitor bfm as shown in figure 3.26.

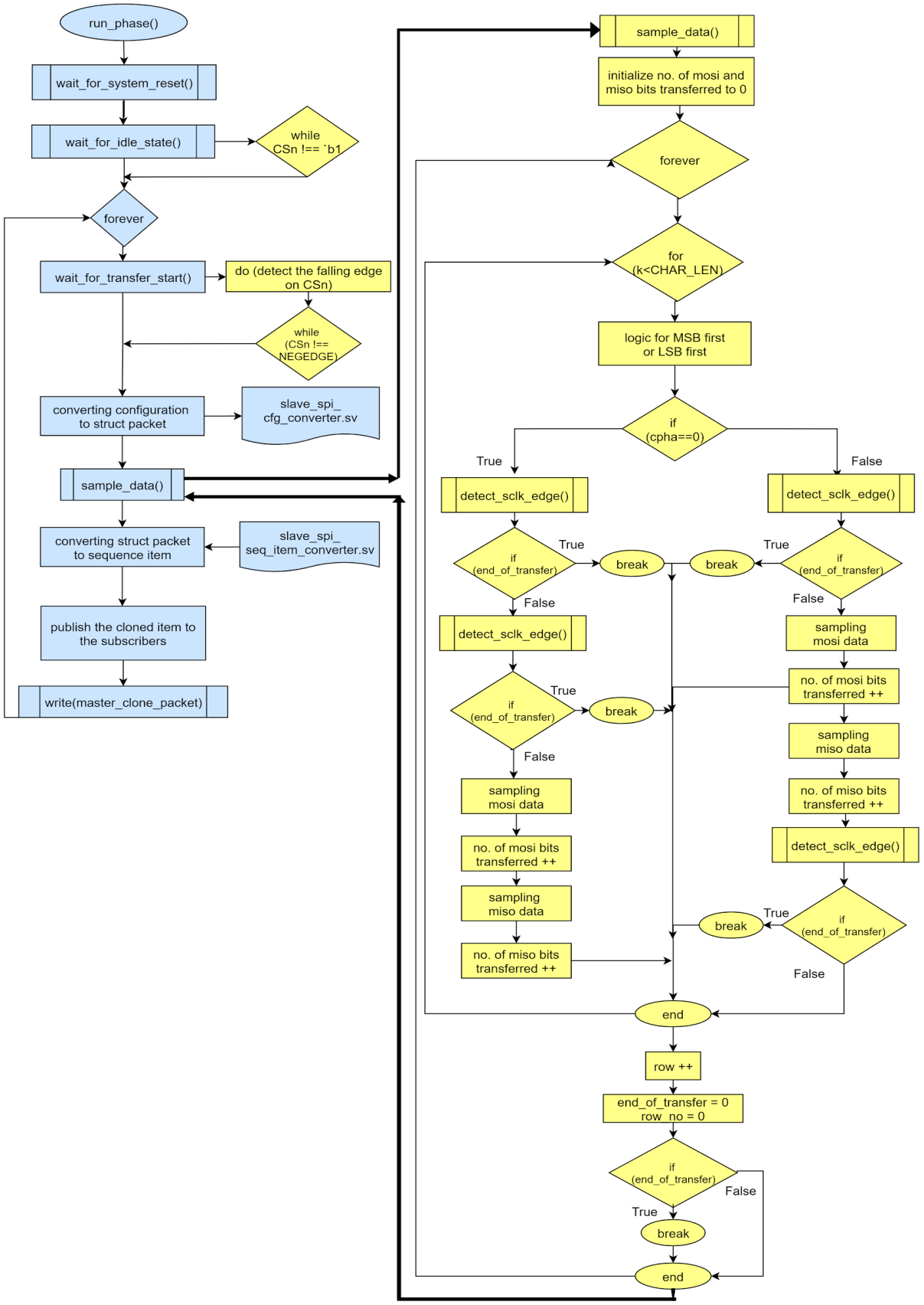


Fig 3.26 Flowchart of spi slave monitor bfm and slave monitor proxy communication

```
task slave_monitor_proxy::run_phase(uvm_phase phase);
  slave_tx slave_packet;
  `uvm_info(get_type_name(), $sformatf("Inside the slave_monitor_proxy"), UVM_LOW);
  slave_packet = slave_tx::type_id::create("slave_packet");
  slave_mon_bfm_h.wait_for_system_reset();
  slave_mon_bfm_h.wait_for_idle_state();
  forever begin
    spi_transfer_char_s struct_packet;
    spi_transfer_cfg_s struct_cfg;
    slave_tx slave_clone_packet;
    slave_mon_bfm_h.wait_for_transfer_start();
    slave_spi_cfg_converter::from_class(slave_agent_cfg_h, struct_cfg);
    slave_mon_bfm_h.sample_data(struct_packet, struct_cfg);

    slave_spi_seq_item_converter::to_class(struct_packet, slave_packet);

    `uvm_info(get_type_name(),$sformatf("Received packet from BFM : , \n %s",
                                        slave_packet.sprint()),UVM_HIGH)
    $cast(slave_clone_packet, slave_packet.clone());
    `uvm_info(get_type_name(),$sformatf("Sending packet via analysis_port : , \n %s",
                                        slave_clone_packet.sprint()),UVM_HIGH)
    slave_analysis_port.write(slave_clone_packet);
  end
endtask : run_phase
```

Fig 3.27 run phase of spi slave monitor proxy code snippet

3.2.21 UVM Verbosity

There are predefined UVM verbosity settings built into UVM (and OVM). These settings are included in the UVM src/uvm_object_globals.svh file and the settings are part of the enumerated uvm_verbosity type definition. The settings actually have integer values that increment by 100 as shown below table

Table 3.2 UVM verbosity Priorities

Verbosity	Default Value
UVM_NONE	0(Highest Priority)
UVM_LOW	100
UVM_MEDIUM	200
UVM_HIGH	300
UVM_FULL	400
UVM_DEBUG	500(Lowest Priority)

By default, when running a UVM simulation, all messages with verbosity settings of UVM_MEDIUM or lower (UVM_MEDIUM, UVM_LOW and UVM_NONE) will print. Table 3.3 shows the Verbosity levels that have used in this particular project

Table 3.3 Descriptions of each Verbosity level

Verbosity	Description
UVM_NONE	UVM_NONE is level 0 and should be used to reduce report verbosity to a bare minimum of vital simulation regression suite messages.
UVM_LOW	UVM_LOW is level 100 and should be used to reduce report verbosity and only shows important messages
UVM_MEDIUM	UVM_MEDIUM is level 200 and should be used as the default $display command. If the verbosity isn't selected then, these messages will print by default as UVM_MEDIUM. This verbosity setting should not be used for any debugging messages or for standard test-passing messages.
UVM_HIGH	UVM_HIGH is level 300 and should be used to increase report verbosity by showing both failing and passing transaction information, but does not show annoying UVM phase status information after it has been established that the UVM phases are working properly
UVM_FULL	UVM_FULL is level 400 and should be used to increase report verbosity by showing UVM phase status information as well as both failing and passing transaction information.

```
// Asserting CS and driving sclk with initial value
`uvm_info("MASTER_DRIVER_BFM", $sformatf("Transfer start is detected"), UVM_NONE);
@(posedge pclk);
cs <= data_packet.cs;
sclk <= cfg_pkt.cpol;
```

Fig 3.28 Verbosity with UVM_NONE

```
drive_to_bfm(struc_packet, struct_cfg);

master_spi_seq_item_converter::to_class(struc_packet,    );

`uvm_info(get_type_name(),$sformatf("Received packet from MASTER DRIVER BFM : , \n %s",
                                    .sprint()),UVM_LOW)
            .item_done();
end
```

Fig 3.29 Verbosity with UVM_LOW

```
slave_drv_bfm_h.drive_the_miso_data(packet,struct_cfg);
`uvm_info(get_type_name(),$sformatf("BFM STRUCT DATA : , \n %p",
                                    packet),UVM_MEDIUM)
if(packet.no_of_miso_bits_transfer == packet.no_of_mosi_bits_transfer) begin
  `uvm_info("DEBUG_SLAVE_DRIVER_PROXY","MOSI AND MISO TRANSFER BITS SIZE IS SAME",UVM_HIGH)
end
else begin
  `uvm_error("DEBUG_SLAVE_DRIVER_PROXY","MOSI AND MISO TRANSFER SIZE IS DIFFERENT")
end
```

Fig 3.29 Verbosity with UVM_MEDIUM

```
//-------------------------------------------------------
// Task: wait_for_reset
// Waiting for system reset to be active
//-------------------------------------------------------
task wait_for_reset();
  @(negedge areset);
  `uvm_info("MASTER_DRIVER_BFM", $sformatf("System reset detected"), UVM_HIGH);
  @(posedge areset);
  `uvm_info("MASTER_DRIVER_BFM", $sformatf("System reset deactivated"), UVM_HIGH);
endtask: wait_for_reset
```

Fig 3.30 Verbosity with UVM_HIGH

```
  end while(! ((sclk_local == POSEDGE) || (sclk_local == NEGEDGE)) );

  sclk_edge_value = edge_detect_e'(sclk_local);
  `uvm_info("MASTER_MONITOR_BFM", $sformatf("SCLK %s detected", sclk_edge_value.name()),UVM_FULL)

endtask: detect_sclk
```

Fig 3.31 Verbosity with UVM_FULL

Fig from 3.28 to 3.31 shows all the verbosity levels used in this project at various components

Chapter 4

Directory Structure

The package structure diagram navigates users to find out the file locations , where it is located and which folder.

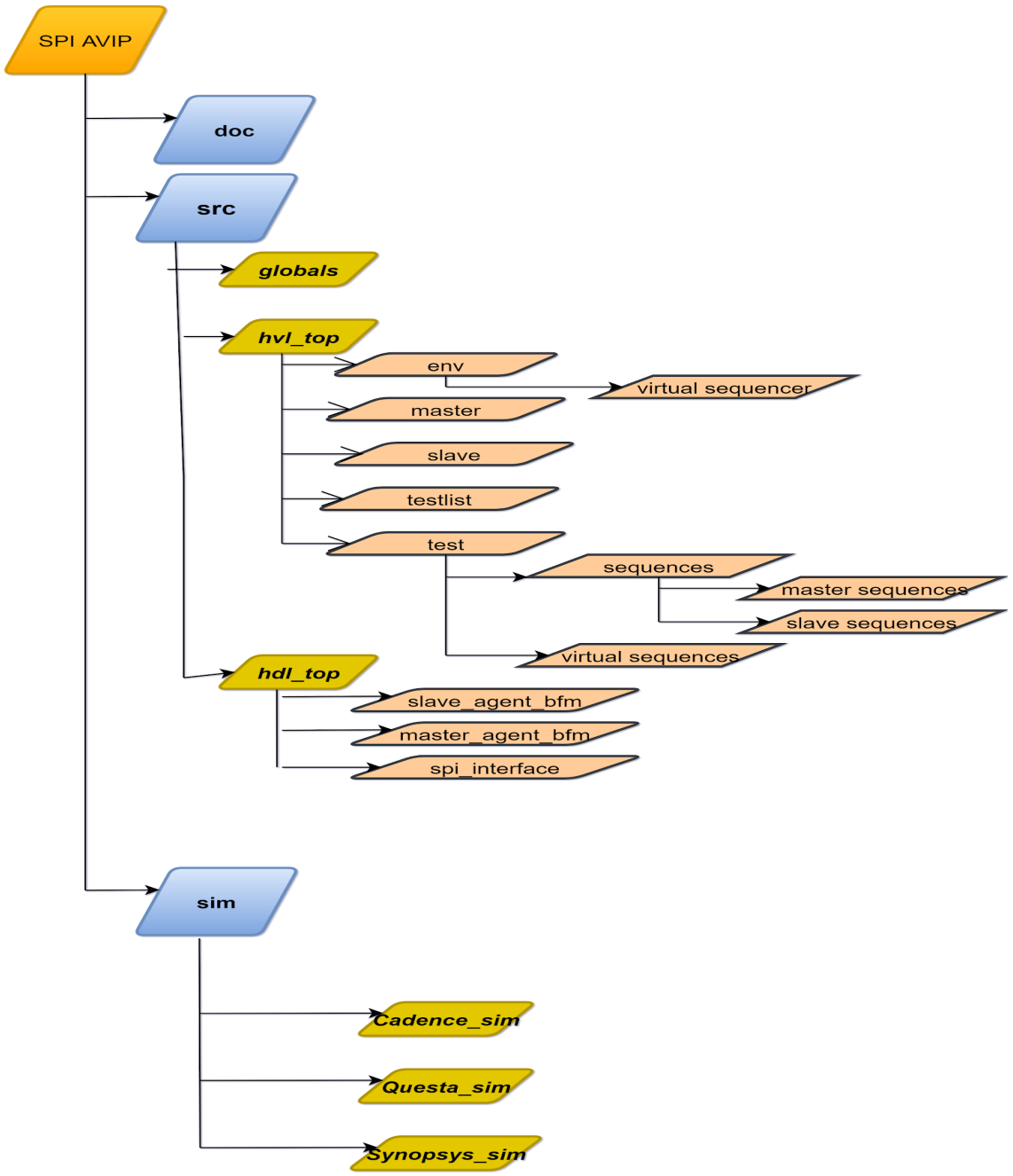


Figure 29. Package Structure of SPI_AVIP

Table 4.1 Directory Path

Directory	Description
spi_avip/doc	contains test bench architecture and components description and verification plan and assertion plan
spi_avip/sim	Contains all simulating tools and spi_compile.f file which contain all directories and compiling files
spi_avip/src/globals	Contains global package parameters(names,modes)
spi_avip/src/hvl_top	Contain all tb component folder(master,env,slave,test)

spi_avip/src/hdl_top	Contain all bfm files and assertions files
spi_avip/src/hdl_top/master_agent_bfm	Contain master agent, driver and monitor bfm files
spi_avip/src/hdl_top/slave_agent_bfm	Contains slave agent, driver and monitor bfm files
spi_avip/src/hdl_top/spi_interface	Contain spi interface file
spi_avip/src/hvl_top/test	Contains all test cases files
spi_avip/src/hvl_top/test/sequences/master_sequences	Contain all master sequence test files
spi_avip/src/hvl_top/test/sequences/slave_sequences	Contain all slave sequence test files
spi_avip/src/hvl_top/test/sequences/virtual_sequences	Contain all virtual sequence test files
spi_avip/src/hvl_top/env	Contain env config files and score board file
spi_avip/src/hvl_top/env/virtual_sequencer	Contain virtual sequencer file
spi_avip/src/hvl_top/master	Contain master agent files , coverage file
spi_avip/src/hvl_top/slave	Contain slave agent files , coverage file

Chapter 5

Configurations

5.1 Global package variables

Name	Type	Description
NO_OF_SLAVES	integer	specifies no of slaves connected to the spi interface
CHAR_LENGTH	integer	specifies the length of the transfer
MAXIMUM_BITS	integer	It's maximum bits supported per transfer
NO_OF_ROWS	integer	specifies total no of 8 bits data packets
edge_detect_e	enum	**POSEDGE =2'b01** : posedge on the signal, the transition from 0->1 **NEGEDGE =2'b10** : negedge on the signal, the transition from 1->0
shift_direction_e	enum	**LSB_FIRST =1'b0** : lsb will be shifted first **MSB_FIRST =1'b1** : msb will be shifted first.
operation_modes_e	enum	Modes of operation : **CPOL0_CPHA0 = 2'b00** **CPOL0_CPHA1 = 2'b01** **CPOL1_CPHA0 = 2'b10** **CPOL1_CPHA1 = 2'b11**
spi_type_e	enum	Types of spi like **SIMPLE_SPI = 3'd1** **DUAL_SPI = 3'd2** **QUAD_SPI = 3'd4**
spi_transfer_char_s	struct	Structure to hold the packet data.
spi_transfer_cfg_s	struct	Structure to hold the configuration data.

Configuration used

1. Env configuration
2. Master Agent configuration
3. Slave Agent configuration

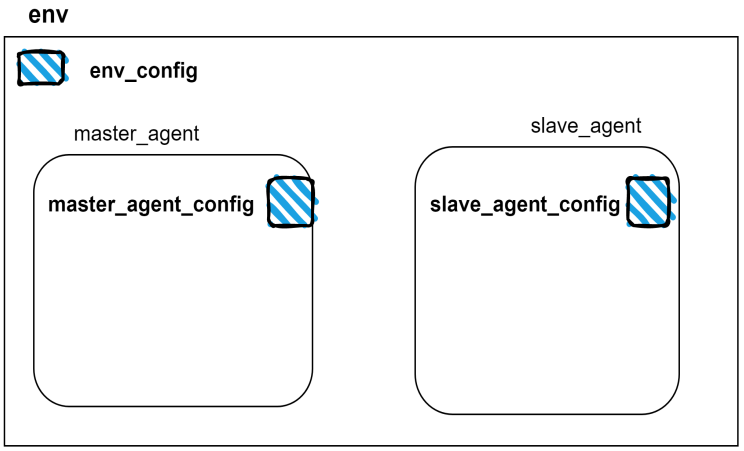


Fig 5.1 Shows various configurations

5.2 Master agent configuration

Name	Type	Default value	Description
is_active	enum	UVM_ACTIVE	It will be used for configuring an agent as an active agent means it has sequencer,driver and monitor or passive agent which has monitor only.
no_of_slaves	integer	‘d1	Used for specifying the number of slaves connected to the master
spi_modes	enum	CPOL0_CPHA0	Used for setting the operation mode.There are 4 modes in spi.
shift_dir	enum	LSB_FIRST	Shifts the data, LSB first or MSB first
c2tdelay	integer	‘d1	Chip select to the first rising edge of spi clock(sclk)
t2cdelay	integer	‘d1	It’s the duration of time from the last edge of the spi clock(sclk) untill the chip select gets deasserted.
wdelay	integer	‘d1	It occurs between t2cdelay and c2tdelay. Then duration of wdelay will be equal to the time duration for which chip select is high(or deasserted) between the two transfers.
primary_prescalar	integer	‘d0	Used for setting the primary_prescalar value for baudrate_divisor,it’s a protected bit so the user can change it’s value by calling set_baudrate_divisior function. **Refer 5.2.6**
secondary_prescalar	integer	‘d0	Used for setting the secondary_prescalar value for baudrate_divisor,,it’s a protected bit so the user can change it’s value by calling set_baudrate_divisor function. **Refer 5.2.6**
baudrate_divisor	integer	‘d2	Defines the data rate,if baudrate_divisor is 2 then period of one sclk equal to the period of 2 pclk.
has_coverage	integer	‘d1	Used for enabling the master agent coverage

5.2.1 spi_modes

The SPI provides the flexibility to program four different clock modes that SPICLK may operate, enabling a choice of the clock phase (delay or no delay) and the clock polarity (rising edge or falling edge)
There are basically four modes of operation in spi

Table 5.3 Spi_modes

POLARITY	PHASE	Action
0	0	Data is output on the rising edge of SPICLK. Input data is latched on the falling edge
0	1	Data is output one half-cycle before the first rising edge of SPICLK and on subsequent falling edges. Input data is latched on the rising edge of SPICLK
1	0	Data is output on the falling edge of SPICLK. Input data is latched on the rising edge
1	1	Data is output one half-cycle before the first falling edge of SPICLK and on subsequent rising edges. Input data is latched on the falling edge of SPICLK.

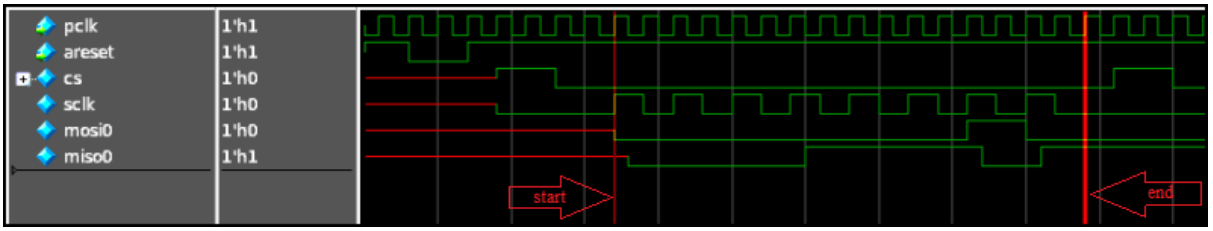


Figure 5.2 cpol0_cpha0

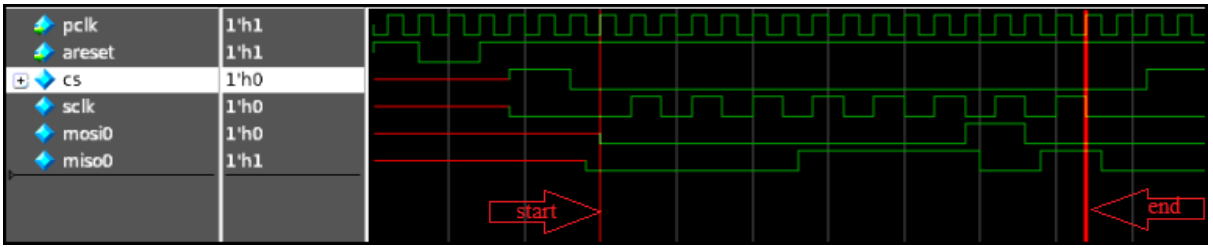


Figure 5.3 cpol0_cpha1

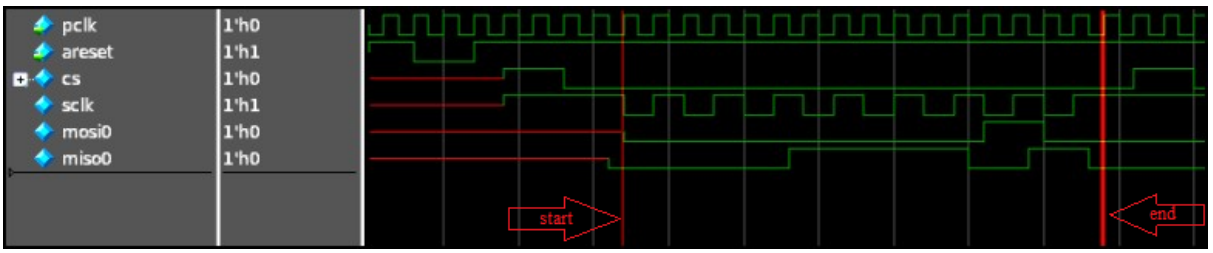


Figure 5.4 cpol1_cpha0

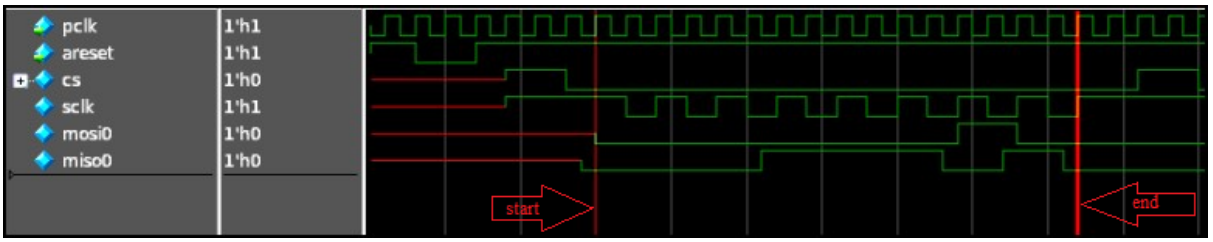


Figure 5.5 cpol1_cpha1

5.2.2 shift_dir

shift the data MSB_FIRST OR LSB_FIRST based on configuration.

LSB_FIRST 32 bits data transfer

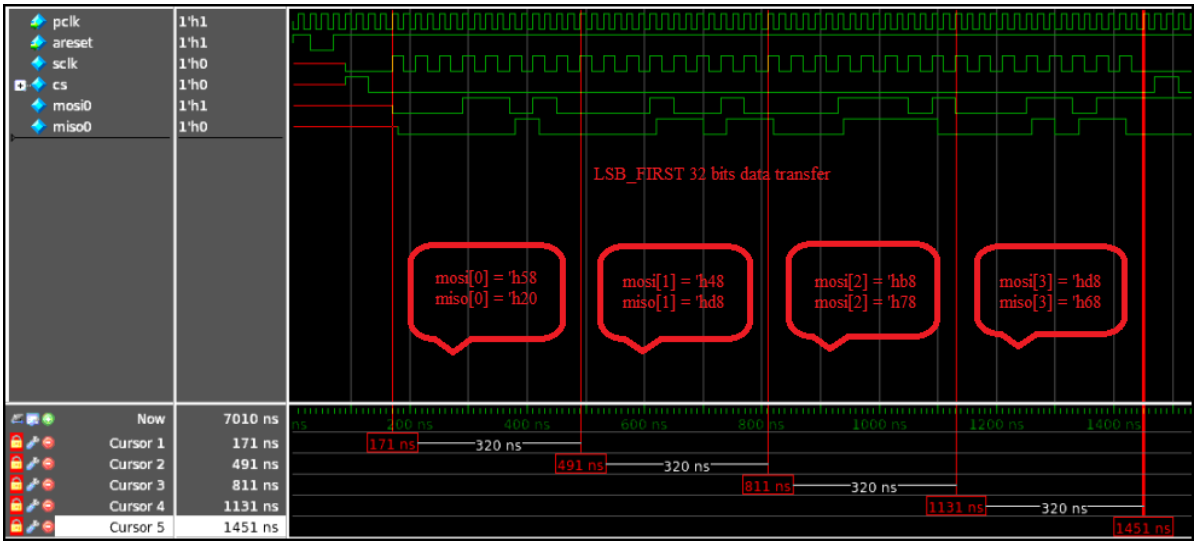


Fig 5.6 LSB_FIRST

MSB_FIRST 32 bits data transfer

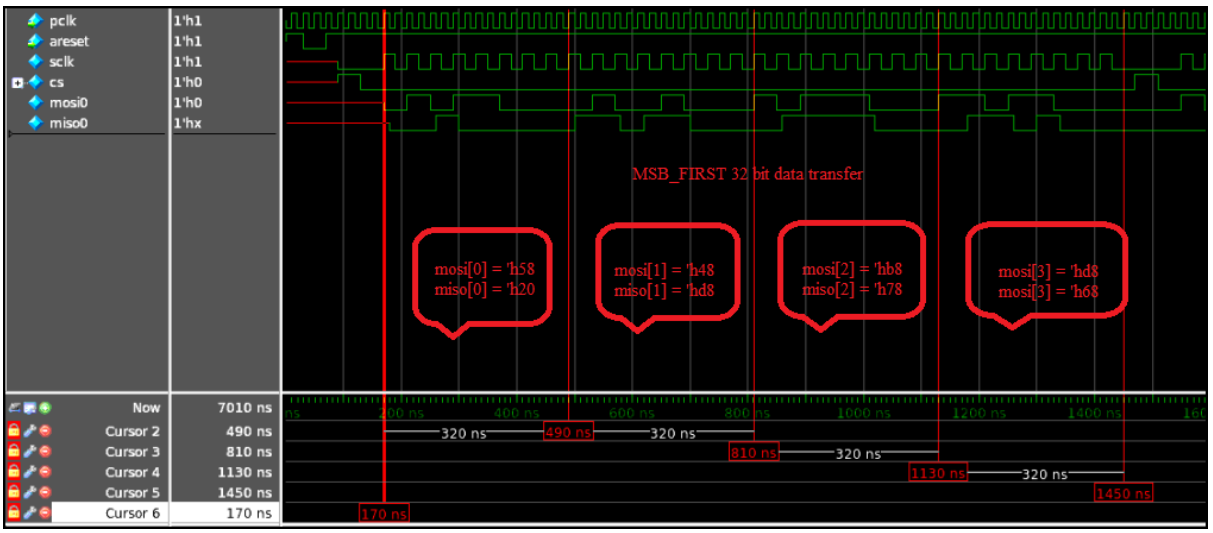


Figure 5.6 MSB_FIRST

5.2.3 c2tdelay

chip select active to transmit start delay .c2t delay is used in master mode only. It defines a setup time for the slave device that delays the data transmission from the chip select active edge by a multiple of SPI module clock cycles

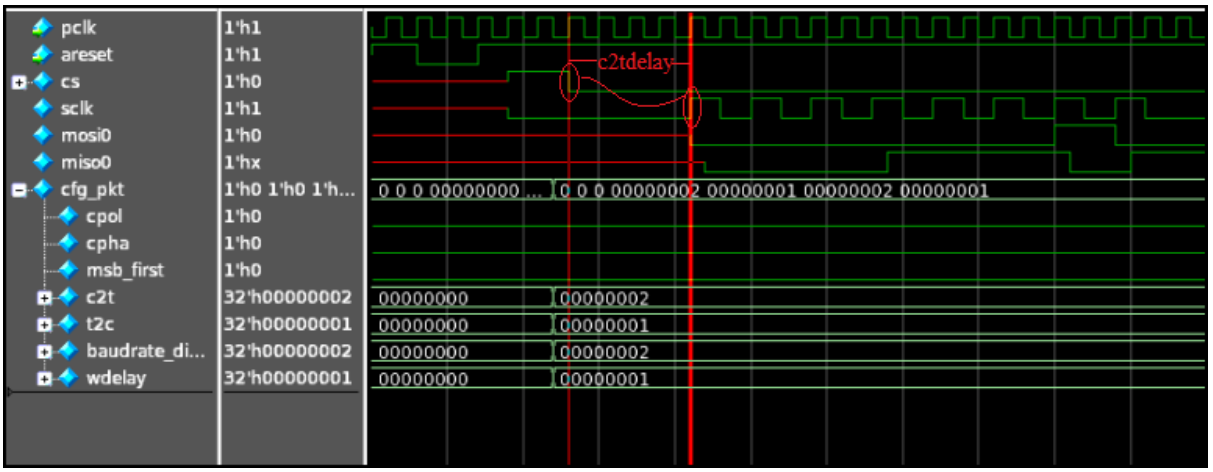


Figure 5.7 c2tdelay

5.2.4 t2cdelay

Transmit-end-to-chip-select-inactive delay,t2cdelay is used in master mode only. It defines a hold time for the slave device that delays the chip select deactivation by a multiple of spi module clock (pclk) cycles after the last bit is transferred.

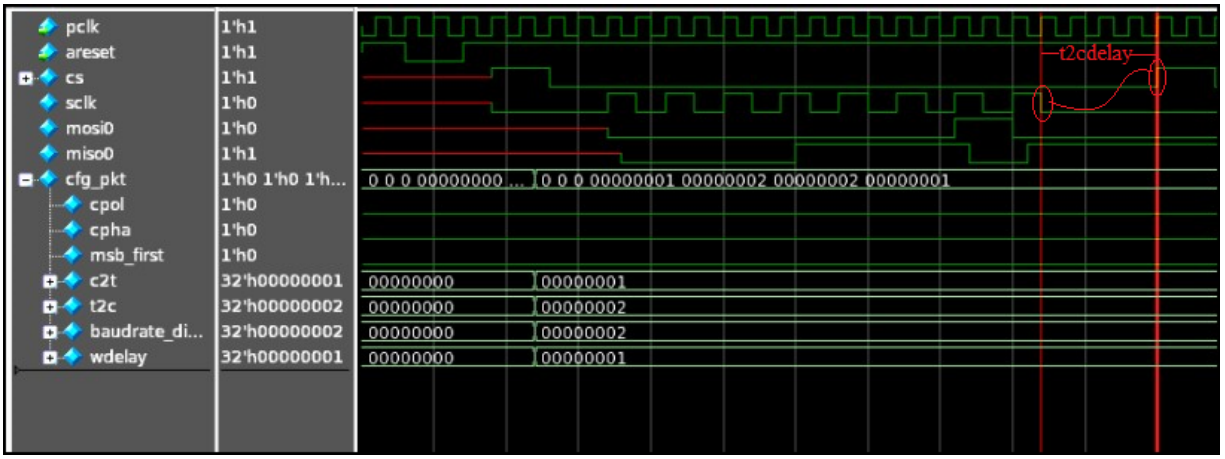


Figure 5.8 shows t2cdelay

5.2.5 wdelay

Enable the delay at the end of the current transaction. The wdelay bit is supported in master mode only. In slave mode, this bit is ignored.

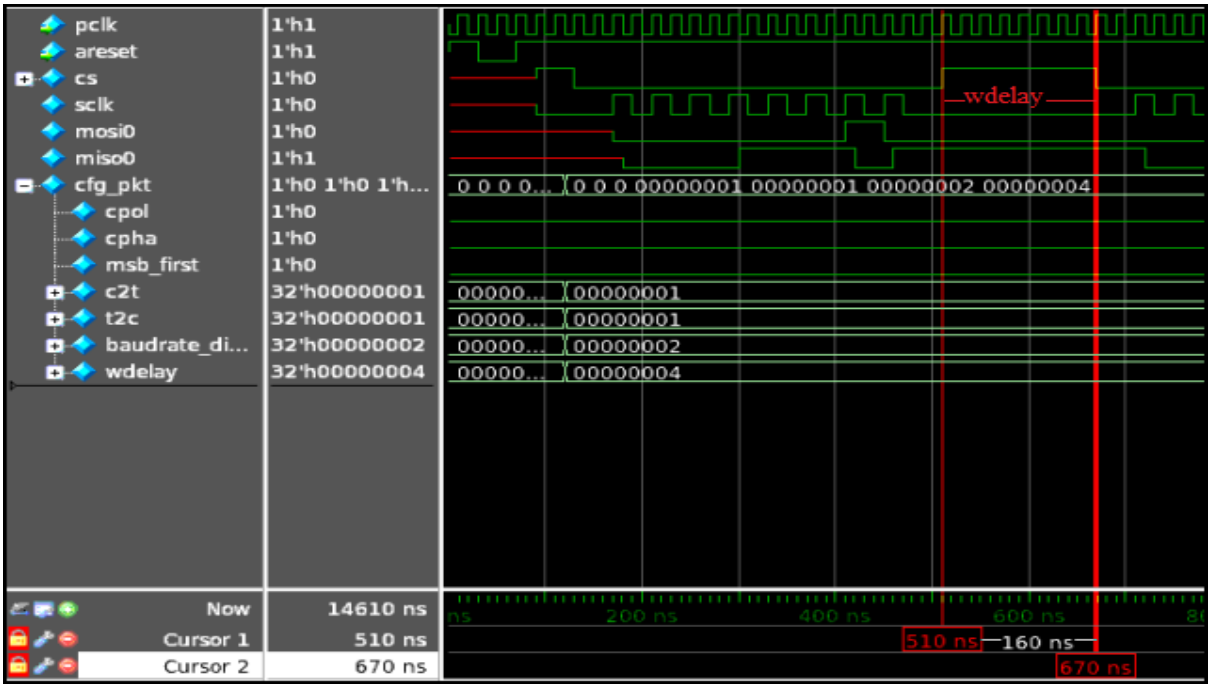


Fig 5.9 wdelay

5.2.6 secondary_prescalar and primary_prescalar

How to set the value of secondary_prescalar and primary_prescalar

```
function void spi_simple_fd_baudrate_test::setup_master_agent_cfg();
  super.setup_master_agent_cfg();
  env_cfg_h.master_agent_cfg_h.set_baudrate_divisor(.primary_prescalar(1),.secondary_prescalar(2));
endfunction : setup_master_agent_cfg
```

Figure 5.10 set values of prescalar

Baudrate_divisor = (secondary_prescalar+1) * (2 ** (primary_prescalar+1))

Table 5.4 Prescaler values

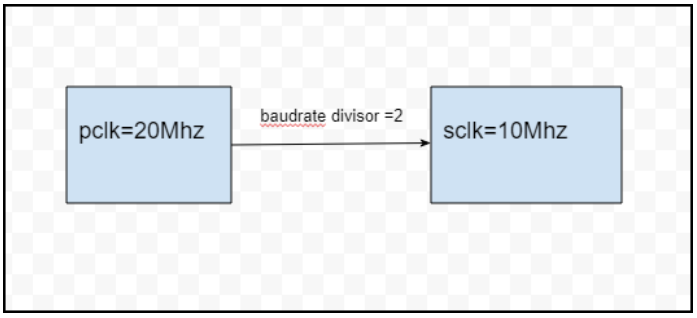
secondary_prescalar	primary_prescalar	**baudrate_divisor**
0	0	2
0	1	4
0	2	8
0	3	16
0	4	32
0	5	64
0	6	128
0	7	256
1	0	4
1	1	8
1	2	16
1	3	32
1	4	64
1	5	128
1	6	256
1	7	512
2	0	6
2	1	12
2	2	24
2	3	48
2	4	96
2	5	192

2	6	384
2	7	768
3	0	8
3	1	16
3	2	32
3	3	64
3	4	128
3	5	256
3	6	512
3	7	1024
4	0	10
4	1	20
4	2	40
4	3	80
4	4	160
4	5	320
4	6	640
4	7	1280
5	0	12
5	1	24
5	2	48
5	3	96
5	4	192
5	5	384
5	6	768
5	7	1536
6	0	14
6	1	28
6	2	56
6	3	112
6	4	224
6	5	448

6	6	896
6	7	1792
7	0	16
7	1	32
7	2	64
7	3	128
7	4	256
7	5	512
7	6	1024
7	7	2048

5.2.7 Baudrate_divisor

baudrate_divisor= (secondary_prescalar+1) * (2 ** (primary_prescalar+1))



Assume secondary_prescalar and primary prescaler to be 0 then baud rate divisor = 2

So if pclk=20Mhz then sclk=10Mhz.

T =1/f =>1/10 => 100ns
so duration of 1 bit is equal to 100ns
1 bit = 100ns
so 1sec = 1/100n bits

Baud rate = 10Mbps
so the baud rate will be 10 Megabits per second

5.3 Slave agent configuration

Table 5.5 Slave_agent_config

Name	Type	Default value	Description
is_active	enum	UVM_ACTIVE	It will be used for configuring agent as an active agent means it has sequencer,driver and monitor and if it's a passive agent then it will have only monitor
slave _id	integer	'd0	Used for indicating the ID of this slave e.g. slave 0 is selected.
spi_modes	enum	CPOL0_CPHA0	Used for setting the operation modes. (refer Table 5.3 for explanation)
shift_dir	enum	LSB_FIRST	Shifts the data, LSB_FIRST or MSB_FIRST.
has_coverage	integer	'd1	Used for enabling the slave agent coverage.

5.4 Environment configuration

Table 5.6 Env_config

Name	Type	Default value	Description
has_scoreboard	integer	1	Enables the scoreboard,it usually receives the transaction level objects via TLM ANALYSIS PORT.
has_virtual_sqr	integer	1	Enables the virtual sequencer which has master and slave sequencer
no_of_slaves	integer	'h1	Number of slaves connected to the SPI interface

Chapter 6

Verification Plan

6.1 Verification plan

Verification plan is an important step in Verification flow; it defines the plan of an entire project and verifies the different scenarios to achieve the test plan.

A Verification plan defines what needs to be verified in Design under test(DUT) and then drives the verification strategy. As an example, the verification plan may define the features that a system has and these may get translated into the coverage metrics that are set.

Refer the below link for SPI Specifications:

Serial Peripheral Interface (SPI) for KeyStone Devices User's Guide (Rev. A)

6.2 Specifications for SPI

Bits of transfers (8, 16, 32, 64, maximum bits (128))

Configurations

Four modes of Configurations

- CPOL0 and CPHASE0
- CPOL0 and CPHASE 1
- CPOL1 and CPHASE0
- CPOL1 and CPHASE 1

Delays

- C2T delay
- T2C Delay
- Wdelay

Shift direction

- MSB
- LSB
- Rate of transfer

Baud rate depends on Primary and Secondary Prescaler

- Baud Rate divisor=(Secondary Prescaler+1)*(2 ** (Primary Prescaler+1)

6.3 Template of Verification Plan

For more information of Verification Plan click below link
spi_avip_verification_plan.xlsx

In the below Figure
Section A shows the S.No
Section B shows the Directed Test Cases
Section C shows the Features

	A	B	C
1	S.NO		Features
2	A	Directed Testcases	
3			
4	1.1	Data transfer	8bits
5	1.2		16bits
6	1.3		32bits
7	1.4		64bits

Fig 6.3.1 Verification plan Template

Section D represents the Description

A	D
S.NO	Description
A	
1.1	Transfer of 8 bits
1.2	Number of bits to be transfered in conjuction Multiples of 8bits (16bits)
1.3	Number of bits to be transfered in conjuction Multiples of 8bits (32bits)
1.4	Number of bits to be transfered in conjuction Multiples of 8bits (64bits)

Fig 6.3.2 Verification plan Section D is Description for tests

Section E and F shows the Test Cases names and Status for heading(Directed Test cases)

	E	F
	Testcases Names	Status
1.1	spi_simple_fd_8b_test	PASS
1.2	spi_simple_fd_16b_test	PASS
1.3	spi_simple_fd_32b_test	PASS
1.4	spi_simple_fd_64b_test	PASS

Fig 6.3.3 Verification plan Section E and F is for Test Names and Status

6.4 Sections for different test Scenarios

Creating the different Sections for different test cases to be developed in the Point of implementing the test scenarios

6.4.1 Directed test

These directed tests provide explicit stimulus to the design inputs, run the design in simulation, and check the behavior of the design against expected results.

Directed test names

This tests describes the different combinations of number of bits transfer

S.NO	No of bit transfers	Test names	Description
1	8bit	spi_simple_fd_8b_test	Checking the 8 bit transfer
2	16bit	spi_simple_fd_16b_test	Checking the 16bit transfer
3	24bit	spi_simple_fd_24b_test	Checking the 24bit transfer
4	32bit	spi_simple_fd_32b_test	Checking the 32 bit transfers
5	Maximum bits	spi_simple_fd_maximum_bits_transfers	Checking the maximum bits transfers (128)
6	Continuous Transfers	spi_simple_fd_ct_test	Checking continuous transfers
7	Discontinuous Transfers	spi_simple_fd_dct_test	Checking discontinuous transfers

. No of bits transfers

Configurations:
This tests describes the Configurations (CPOL and CPHA) and how the SPI protocol follows four modes of Configurations

Table no: 6.2 Configurations

S.No	Configurations	Test names	Description
1	CPOL0_CPHA0	spi_simple_fd_cpol0_cpha0_test	When cpol is low sclk should starts from idle state to active high clock pulse, when cphase is low miso should be happen on trailing edge and mosi should happen on leading edge
2	CPOL0_CPHA1	spi_simple_fd_cpol0_cpha1_test	When cpol is low sclk should starts from idle state to active high clock pulse, when cphase is high miso should be happen on leading edge and mosi should happen on trailing edge
3	CPOL1_CPHA0	spi_simple_fd_cpol1_cpha0_test	When cpol is high sclk should starts from active high clock pulse to idle state, when cphase is low

			miso should be happen on trailing edge and mosi should happen on leading edge
4	CPOL1_CPHA1	spi_simple_fd_cpol1_cpha1_test	When cpol is high sclk should starts from active high clock pulse to idle state, when cphase is high miso should be happen on leading edge and mosi should happen on trailing edge

13.

Delays:
This tests describes the different Delays

S.No	Delays	Test names	Description
1	C2tdelay	spi_c2t_delay_test	Checking the Delay between active low cs and sclk there should be 1sclk delay(c2t=1)
2	T2cdelay	spi_t2c_delay_test	Delay between last edge of sclk and active high cs there should be 1sclk delay(t2c=1)
3	wdelay	spi_simple_fd_dct_test	Delay happens in between t2c and c2t after the first transaction (after first transfer cs should be high)

Table 14. no: 6.3 Delays

Shift direction
This tests describes the direction of number of bits transfers(LSB and MSB)

S.No	Shift direction	Test names	Description
1	LSB	spi_simple_fd_lsb_test	Checking whether lsb first or msb first in bit transfer
2	MSB	spi_simple_fd_msb_test	Checking whether msb first or lsb first in bit transfer

Table 15. Shift direction

Baud Rate
This tests describes the Baud rate which depends on Primary Prescaler and Secondary Prescaler

S.No	Feature	Test names	Description

1	Baud rate	spi_baudrate_test	Checking baud rate with respect to primary and secondary prescaler It describes rate of each transfer(when baudrate=2, 1sclk=2 pclks)

Table 16. Baud Rate

6.4.2 Random test

Though a random test case is powerful in terms of finding bugs faster than the directed one, usually we prefer random test cases for the module and sub-system level verification and mostly we prefer directed test cases for the SoC level verification. .

Purely random test generations are not very useful because of the following two reasons-

a) Generated scenarios may violate the assumptions, under which the design was constructed

b) Many of the scenarios may not be interesting, thus wasting valuable simulation time, hence random stimulus with the constraints..

Random test name

This test describes the randomization of Configurations,Delays and Shift direction

S.No	Name	Test	Description
1	Configurations(Cpol, cpha) Delays(t2c,c2t,w) and shift directions(lsb, msb)	spi_simple_fd_rand_test	Randomizing configurations, shift directions, delays in one test_case, to check randomized values and to increase coverage

Table 17. Randomize the Configurations,Delays,Shift direction

6.4.3 Cross test

The Cross test describes the creation of specific test cases required to hit the crosses and cover points defined in the functional coverage and running them with multiple seeds with functional coverage.

Cross test name:

This test describes the Cross test for Configurations(CPOL and CPHA), Delays and Shift

direction(LSB and MSB)

S.No	Name	Test	Description
1	Configurations(Cpol0,Cpha1) Delays(t2c,c2t,wdelay) shift directions(msb) and baud rate	spi_simple_fd_cross_test	Checking cross coverage for configurations (Cpol0,Cpha1), shift direction, and delays for fixed values(c2t=2, t2c=2, wdelay=2, baud rate)

Table 18. Cross test for Configurations,Delays,Shift direction

4. Implement the negative scenarios in order to verify whether protocol is verifying the different tests according to the protocol

6.4.4 Negative test

Negative testing commonly referred to as error path testing or failure testing is generally done to ensure the stability of the application.

Negative testing is the process of applying as much creativity as possible and validating the application against invalid data. This means its intended purpose is to check if the errors are being shown to the user where it's supposed to, or handling a bad value more gracefully.

Negative test names

This test describes the negative scenarios for Configurations(CPOL and CPHA),Delays and Shift direction (LSB and MSB)

S.No	Name	Test	Description
1	Randomizing the different configurations in master and slave configuration class	Spi_ simple_fd_negative_scenarios_test	Checking the negative scenarios for different configurations(cpol, cpha), delays(c2t,t2c,w) and shift direction(lsb, msb)

Table 19. Negative test for Configurations,Delays,Shift direction

6.4.5 Coverage Closure

Include Directed test,Random test and Cross test and run regression to get Coverage closure

a) Directed test

b) Random test

c) Cross test

Directed test

No of bits transfers -

This tests describes the different combinations of number of bits transfer

S.NO	No of bit transfers	Test names	Description
1	8bit	spi_simple_fd_8b_test	Checking the 8 bit transfer
2	16bit	spi_simple_fd_16b_test	Checking the 16bit transfer
3	24bit	spi_simple_fd_24b_test	Checking the 24bit transfer
4	32bit	spi_simple_fd_32b_test	Checking the 32 bit transfers
5	Maximum bits	spi_simple_fd_maximum_bits_transfers	Checking the maximum bits transfers(128)
6	Continuous Transfers	spi_simple_fd_ct_test	Checking continuous transfers
7	Discontinuous Transfers	spi_simple_fd_dct_test	Checking discontinuous transfers

Table 20. Checking coverage closure for No of bits transfers

Configurations:

This tests describes the Configurations (CPOL and CPHA) and how the SPI protocol follows four modes of Configurations

Table 6.10 Checking coverage closure for Configurations

S.No	Configurations	Test names	Description
1	CPOL0_ CPHA0	spi_simple_fd_cpol0_cpha0_test	When cpol is low sclk should starts from idle state to active high clock pulse, when cphase is low miso should be happen on trailing edge and mosi should happen on leading edge
2	CPOL0_ CPHA1	spi_simple_fd_cpol0_cpha1_test	When cpol is low sclk should starts from idle state to active high clock pulse, when cphase is high miso should be happen on leading edge and mosi should happen on trailing edge
3	CPOL1_ CPHA0	spi_simple_fd_cpol1_cpha0_test	When cpol is high sclk should starts from active high clock pulse to idle state, when cphase is low miso should be happen on trailing edge and mosi should happen on leading edge
4	CPOL1_ CPHA1	spi_simple_fd_cpol1_cpha1_test	When cpol is high sclk should starts from active high clock pulse to idle state, when cphase is high miso should be happen on leading edge and mosi should happen on trailing edge

Table 21.

Delays -

This tests describes the different Delays

Table 6.11 Checking coverage closure for Delays

S.No	Delays	Test names	Description
1	C2tdelay	spi_c2t_delay_test	Checking the Delay between active low cs and sclk there should be 1sclk delay(c2t=1)
2	T2cdelay	spi_t2c_delay_test	Delay between last edge of sclk and active high cs there should be 1sclk delay(t2c=1)
3	wdelay	spi_simple_fd_dct_test	Delay happens in between t2c and c2t after the first transaction (after first transfer cs should be high)

Table 22.

Shift direction

This tests describes the direction of number of bits transfers(LSB and MSB)

S.No	Shift direction	Test names	Description
1	LSB	spi_simple_fd_lsb_test	Checking whether lsb first or msb first in bit transfer
2	MSB	spi_simple_fd_msb_test	Checking whether msb first or lsb first in bit transfer

Table 23. Checking coverage closure for Shift direction

Baud Rate

This tests describes the Baud rate which depends on Primary Prescaler and Secondary Prescaler

S.No	Features	Test names	Description
1	Baud rate	spi_baudrate_test	Checking baud rate with respect to primary and secondary prescaler. It describes rate of each transfer (when baudrate=2,1sclk=2 pclks)

Table 24. Checking coverage closure for Baud Rate

Random test name

This test describes the randomization of Configurations,Delays and Shift direction

S.No	Name	Test	Description
1	Configurations(Cpol,cpha) Delays(t2c,c2t,w) and shift directions(lsb, msb)	spi_simple_fd_rand_test	Randomizing configurations, shift directions, delays in one test_case, to check randomized values and to increase coverage

Table 25. Checking coverage closure for Randomize the Configurations,Delays,Shift direction

Cross test name

This test describes the Cross test for Configurations(CPOL and CPHA), Delays and Shift

direction(LSB and MSB)

S.No	Name	Test	Description
1	Configurations(Cpol0,Cpha1) Delays(t2c,c2t,wdelay) shift directions(msb) and baud rate	spi_simple_fd_cross_test	Checking cross coverage for configurations (Cpol0,Cpha1), shift direction, and delays for fixed values(c2t=2, t2c=2, wdelay=2, baud rate)

Table 26. Checking coverage closure for Cross test for Configurations,Delays,Shift direction

For more information about Verification plan refer below link

spi_avip_verification_plan.xlsx

Chapter 7

Assertion Plan

7.1 Assertion Plan overview

Assertion plan is an important step in verification flow, which validates the behaviour of design at every instance.

7.1.1 What are assertions?

- An assertion specifies the behavior of the system.

- Piece of verification code that monitors a design implementation for compliance with the specifications
- Directive to a verification tool that the tool should attempt to prove/assume/count a given property using formal methods

7.1.2 Why do we use it?

- Assertions are primarily used to validate the behavior of a design.
- Assertions can be used to provide functional coverage and to flag that input stimulus, which is used for validation, does not conform to assumed requirements.
- Assertions are used to find more bugs and source the bugs faster.

7.1.3 Benefits of Assertions

- Improves observability of the design.
- Improves debugging of the design.
- Improves documentation of the design.

7.2 Template of Assertion Plan

Template for Assertion plan is done in an excel sheet and refer to link below: assertions_plan.xlsx

7.3 Assertion Condition

Table 7.1 Assertion Table

Assertion Label	Description
IF_SIGNALS_ARE_STABLE_SIMPLE_SPI	This assert property Check for stable sclk,miso and mosi when cs is high at posedge pclk
CS_LOW_CHECK_SIMPLE_SPI	This assert property Check for sclk ,miso and mosi data is valid (not unknown values) when cs is low at posedge pclk
CPOL_IDEL_STATE_CHECK	This assert property will check for idle state of chip select.
CPOL_0_CPHA_0_SIMPLE_SPI	This assert property will check for the data should be stable between the two posedge clocks
CPOL_0_CPHA_1_SIMPLE_SPI	This assert property will check for the data should be stable between the two negedge clocks
CPOL_1_CPHA_0_SIMPLE_SPI	This assert property will check for the data should be stable between the two negedge clocks
CPOL_1_CPHA_1_SIMPLE_SPI	This assert property will check for the data should be stable between the two posedge clocks

Table 27.

7.3.1 Stable condition check

When cs is high, the signals sclk, mosi, miso should be stable.

```
  property if_signals_are_stable(logic mosi_local, logic miso_local);
    @(posedge pclk) disable iff(!areset)
    cs=='1  |=> $stable(sclk) && $stable(mosi_local) && $stable(miso_local);
  endproperty : if_signals_are_stable
IF_SIGNALS_ARE_STABLE_SIMPLE_SPI: assert property (if_signals_are_stable(mosi0,miso0));
```

Figure 43.

Property if_signal_are_stable is evaluated as follows:

- Initially it will check for posedge of pclk and areset should be high.
- When cs is high ,in the next cycle it will check for sclk,mosi and miso signal should be stable. Then property is true.
- Otherwise the property will fail.

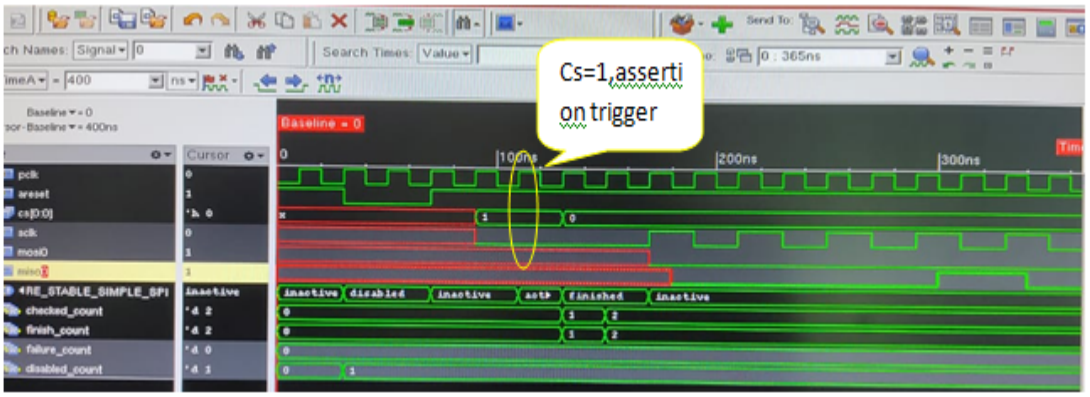


Figure 44.

Fig. 7.3.1: Stable condition Check

7.3.2. Data Valid Check

Mosi and miso data should never be unknown or high impedance during data sampling and driving.

```
property mosi_miso_valid_p(logic mosi_local, logic miso_local);
  @(posedge pclk) disable iff(!areset)
  cs=='0 |=> !$isunknown(sclk) && !$isunknown(mosi_local) |-> !$isunknown(miso_local);
endproperty : mosi_miso_valid_p
CS_LOW_CHECK_SIMPLE_SPI: assert property (mosi_miso_valid_p(mosi0,miso0));
```

Figure 45.

Property master_mosi0_valid_p is evaluated as follows:

- Initially it will check for posedge of pclk and areset should be high.
- When cs is low,it triggers the condition and in the next cycle of pclk it will check for sclk and mosi should not be unknown or high impedance. At the same clock cycle it

checks for miso signals that should not be unknown and high impedance. Then property is true.

- Otherwise the property will fail.

Figure 46. Data valid check

7.3.3. Clock polarity check

When cpol is low, idle state should be logic low/When cpol is high, idle state should be logic high.

```
property cpol_idle_state_check_p;
  @(posedge pclk) disable iff(!areset)
  cs=='1 |-> sclk == cpol;
endproperty : cpol_idle_state_check_p

CPOL_IDLE_STATE_CHECK: assert property(cpol_idle_state_check_p);
```

Figure 47.

Property master_mosi0_valid_p is evaluated as follows:

- Initially it will check for posedge of pclk and areset should be high.
- When cs = 1, in the next clock cycle of pclk, sclk is equal to cpol value.

7.3.4. Configuration check

Here mosi and miso data sampling happen on posedge or negedge of clock depending on the mode of cpol and cpha configuration.

7.3.4.1. CPOL_0_CPHA_0

```
property mode_of_cfg_cpol_0_cpha_0(logic mosi_local,logic miso_local);
  @(negedge sclk) disable iff(!areset)
  cpol==0 && cpha==0 |-> $stable(mosi_local) && $stable(miso_local);
endproperty: mode_of_cfg_cpol_0_cpha_0

CPOL_0_CPHA_0_SIMPLE_SPI: assert property (mode_of_cfg_cpol_0_cpha_0(mosi0,miso0));
```

Figure 48.

This property will check for the mosi and miso data should be stable at negedge of sclk during sampling. If the data is toggled at the negedge of sclk the assertion failure happens.

7.3.4.2. CPOL_0_CPHA_1

```
property mode_of_cfg_cpol_0_cpha_1(logic mosi_local, logic miso_local);
  @(posedge sclk) disable iff(!areset)
 cpol==0 && cpha==1 |-> $stable(mosi_local) && $stable(miso_local);
endproperty: mode_of_cfg_cpol_0_cpha_1

CPOL_0_CPHA_1_SIMPLE_SPI: assert property (mode_of_cfg_cpol_0_cpha_1(mosi0,miso0));
```

Figure 49.

This property will check for the mosi and miso data should be stable at posedge of sclk during sampling. If the data is toggled at the negedge of sclk the assertion failure happens.

7.3.4.3. CPOL_1_CPHA_0

```
property mode_of_cfg_cpol_1_cpha_0(logic mosi_local,logic miso_local);
  @(posedge sclk) disable iff(!areset)
  cpol==1 && cpha==0 |-> $stable(mosi_local) && $stable(miso_local);
endproperty: mode_of_cfg_cpol_1_cpha_0

CPOL_1_CPHA_0_SIMPLE_SPI: assert property (mode_of_cfg_cpol_1_cpha_0(mosi0,miso0));
```

Figure 50.

This property will check for the mosi and miso data should be stable at posedge of sclk during sampling. If the data is toggled at the negedge of sclk the assertion failure happens.

7.3.4.4. CPOL_1_CPHA_1

```
property mode_of_cfg_cpol_1_cpha_1(logic mosi_local,logic miso_local);
  @(negedge sclk) disable iff(!areset)
  cpol==1 && cpha==1 |-> $stable(mosi_local) && $stable(miso_local);
endproperty : mode_of_cfg_cpol_1_cpha_1

CPOL_1_CPHA_1_SIMPLE_SPI: assert property (mode_of_cfg_cpol_1_cpha_1(mosi0,miso0));
```

Figure 51.

This property will check for the mosi and miso data should be stable at negedge of sclk during sampling. If the data is toggled at the negedge of sclk the assertion failure happens.

7.4 Binding the assertions

At first declare the keyword **bind,** followed by the place where binding is done, assertion module name, handle declared for the assertion module as shown in fig.5.2

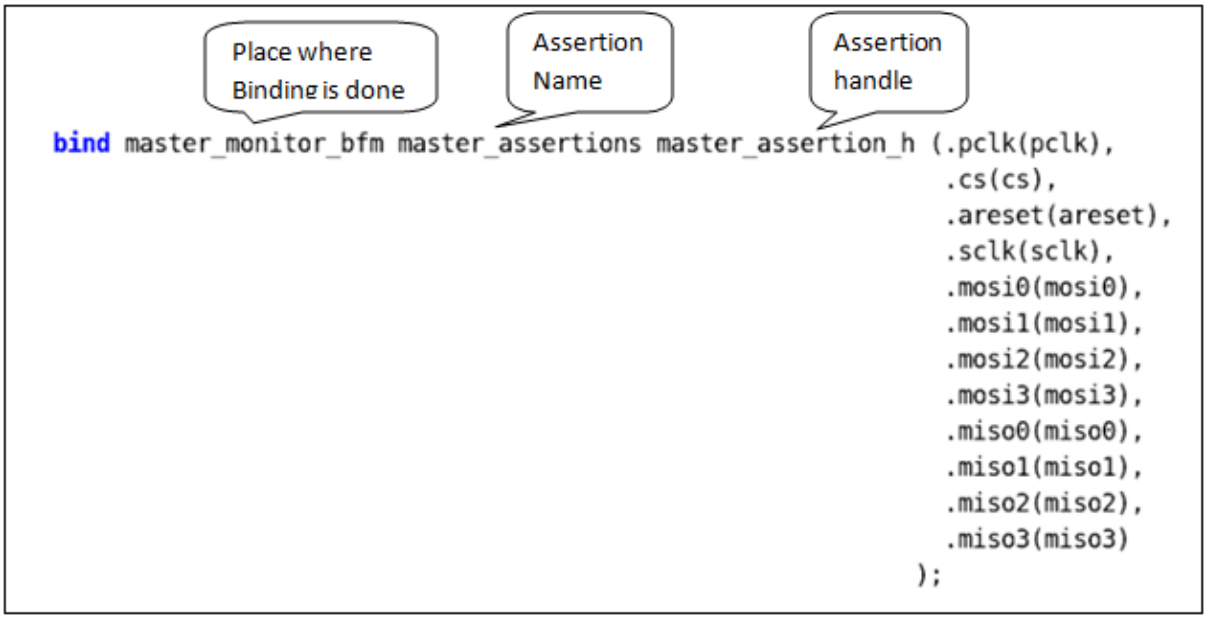


Figure 52.

Chapter 8

Coverage

8.1 Functional Coverage

- Functional coverage is the coverage data generated from the user defined functional coverage model and assertions usually written in System Verilog During simulation,

the simulator generates functional coverage based on the stimulus. Looking at the functional coverage data, one can identify the portions of the DUT [Features] verified. Also, it helps us to target the DUT features that are unverified.

- The reason for switching to the functional coverage is that we can create the bins manually as per our requirement while in the code coverage it is generated by the system by itself.

8.2 Uvm_Subscriber

- This class provides an analysis export for receiving transactions from a connected analysis export. Making such a connection "subscribes" this component to any transactions emitted by the connected analysis port. Subtypes of this class must define the write method to process the incoming transactions. This class is particularly useful when designing a coverage collector that attaches to a monitor.

```
virtual class  uvm_subscriber #(type T=int) extends uvm_component;
typedef uvm_subscriber #(T) this_type;

// Port: analysis_export
//
// This export provides access to the write method, which derived subscribers
// must implement.

uvm_analysis_imp #(T, this_type) analysis_export;

// Function: new
//
// Creates and initializes an instance of this class using the normal
// constructor arguments for <uvm_component>: ~name~ is the name of the
// instance, and ~parent~ is the handle to the hierarchical parent, if any.

function new (string name, uvm_component parent);
  super.new(name, parent);
  analysis_export = new("analysis_imp", this);
endfunction

// Function: write
//
// A pure virtual method that must be defined in each subclass. Access
// to this method by outside components should be done via the
// analysis_export.

 pure virtual function void write(T t);

 endclass
```

Figure 53. Uvm_subscriber

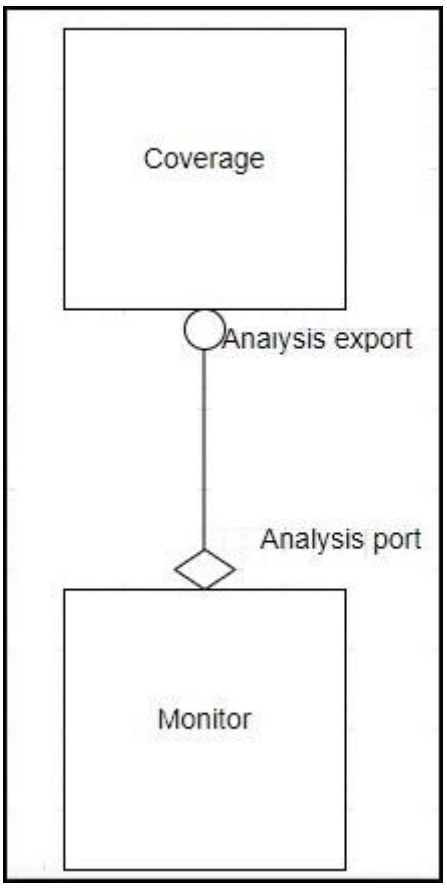


Figure 54. Monitor and coverage connection

8.2.1 Analysis export

This export provides access to the write method, which derived subscribers must implement.

8.2.2 Write function

The write function is to process the incoming transactions.

```
function void master_coverage::write(master_tx t);


  `uvm_info("MUNEEB_DEBUG", $sformatf("Config values = %0s", master_agent_cfg_h.sprint()), UVM_HIGH);
//sampling for the covergroup is started over here
   master_covergroup.sample(master_agent_cfg_h,t);

endfunction: write
```

Figure 55. Write function

8.3 Covergroup

```
//------------------------------------------------------------------------------------------
class master_coverage extends uvm_subscriber#(master_tx);
  `uvm_component_utils(master_coverage)

  // Variable: master_agent_cfg_h
  // Declaring handle for master agent configuration class
    master_agent_config master_agent_cfg_h;

  //-----------------------------------------------------
  // Covergroup
  // // TODO(mshariff): Add comments
  // Covergroup consists of the various coverpoints based on the no. of the variables used to impl
  //-----------------------------------------------------
  covergroup master_covergroup with function sample (master_agent_config cfg, master_tx packet);
    option.per_instance = 1;
    // Mode of the operation

    // {cpol,cpha} = operation_modes_e'(cfg.spi_mode);
    OPERATION_MODE_CP : coverpoint operation_modes_e'(cfg.spi_mode) {
      option.comment = "Operation mode SPI. CPOL and CPHA";
      // TODO(mshariff):
        bins MODE[] = {[0:3]};
    }

    // Chip-selcet to first SCLK-edge delay
    C2T_DELAY_CP : coverpoint cfg.c2tdelay {
      option.comment = "Delay betwen CS assertion to first SCLK edge";
      // TODO(mshariff):
       bins DELAY_1 = {1};
       bins DELAY_2 = {2};
       bins DELAY_3 = {3};
```

Figure 56. Covergroup

The above red mark points in Figure 8.3 is explained below :-

1. **With function sample**: - It is used to pass a variable to covergroup.
2. Parameter based on which the coverpoint is generated.
3. **Per Instance Coverage - *'option.per_instance'***

In your test bench, you might have instantiated coverage_group multiple times. By default, System Verilog collects all the coverage data from all the instances. You might have more than one generator and they might generate different streams of transaction. In this case you may want to see separate reports. Using this option, you can keep track of coverage for each instance.

3.1. ***option.per_instance=1*** Each instance contributes to the overall coverage information for the covergroup type. When true, coverage information for this covergroup instance shall be saved in the coverage database and included in the coverage report.

```
covergroup master_covergroup with function sample (master_agent_config cfg, master_tx packet);
  option.per_instance = 1;
```

Figure 57. option.per_instance

4. Cover Group Comment - *'option.comment'*

You can add a comment in to coverage report to make them easier while analyzing:

Comment:
Operation mode SPI. CPOL and CPHA

Search:

Bin Name	At Least	Hits
MODE[CPOL0_CPHA0]	1	5
MODE[CPOL0_CPHA1]	1	0
MODE[CPOL1_CPHA0]	1	0
MODE[CPOL1_CPHA1]	1	0

Figure 8.3.2 option.comment

For example, you could see the usage of 'option.comment' feature. This way you can make the coverage group easier for the analysis.

Figure 58.

8.4 Bucket

In this we create the single bin for the multiple values i.e.

```
bins MODE[] = {[0:3]};   //4 bins will be created one-one for the each values
bins W_DELAY_MAX = {[4:MAXIMUM_BITS]}; // one bin is created for 4 to max_bits
```

Figure 59. Bucket

- In the above 2 points we can see that the Mode[] have the bins for each value.
- In the second W_Delay_Max there is only one bin created for the many values

8.5 Coverpoints - MOSI(MASTER_OUT_SLAVE_IN)

There we are created the bins based on the mosi.size and the character length

```
MOSI_DATA_TRANSFER_CP : coverpoint packet.master_out_slave_in.size()*CHAR_LENGTH {
  option.comment = "Data size of the packet transfer";
  bins TRANSFER_8BIT = {8};
  bins TRANSFER_16BIT = {16};
  bins TRANSFER_24BIT = {24};
  bins TRANSFER_32BIT = {32};
  bins TRANSFER_64BIT = {64};
  bins TRANSFER_MANY_BITS = {[72:MAXIMUM_BITS]};
}
```

Figure 60. Coverpoint

8.6 Cross coverpoints

Cross allows keeping track of information which is received simultaneous on more than one cover point. Cross coverage is specified using the cross construct.

-> CS (CHIP_SELECT) with direction, operation mode, delays and mosi

```
//cross of the cs_cp with mosi_data_transfer_cp, shift_direction, operation_modeand the delays
CS_CP_X_MOSI_DATA_TRANSFER_CP : cross CS_CP,MOSI_DATA_TRANSFER_CP;
CS_CP_X_SHIFT_DIRECTION_CP : cross CS_CP,SHIFT_DIRECTION_CP;
CS_CP_X_OPERATION_MODE_CP : cross CS_CP,OPERATION_MODE_CP;
CS_CP_X_C2T_DELAY_CP_X_T2C_DELAY_CP_X_W_DELAY_CP : cross CS_CP,C2T_DELAY_CP,T2C_DELAY_CP,W_DELAY_CP;
```

Figure 61. Cross Coverpoints

8.6.1 Illegal bins

illegal_bins illegal_bin = {0};

Illegal bins are used when we don't want to have the particular value eg - we don't want to have the baud_rate_divisior to be zero so we create the illegal bin for it.

For the more coverpoints and coverage refer this link - Coverpoints

8.7 Creation of the covergroup

```
function master_coverage::new(string name = "master_coverage", uvm_component parent = null);
  super.new(name, parent);

     master_covergroup = new();
endfunction : new
```

Figure 62. Creation of covergroup

In this function the creation of the covergroup is done with the new as shown in the figure above.

8.8 Sampling of the covergroup

In this the sampling of the covergroup is done in the write function as shown below

```
function void master_coverage::write(master_tx t);


  `uvm_info("MUNEEB_DEBUG", $sformatf("Config values = %0s", master_agent_cfg_h.sprint()), UVM_HIGH);
//sampling for the covergroup is started over here
   master_covergroup.sample(master_agent_cfg_h,t);

endfunction: write
```

Figure 63. Sampling of the covergroup

8.9 Checking for the coverage

1. Make Compile
2. Make simulate
3. Open the log file

```
Log file path: spi_simple_fd_8b_test/spi_simple_fd_8b_test.log
```

Figure 64.

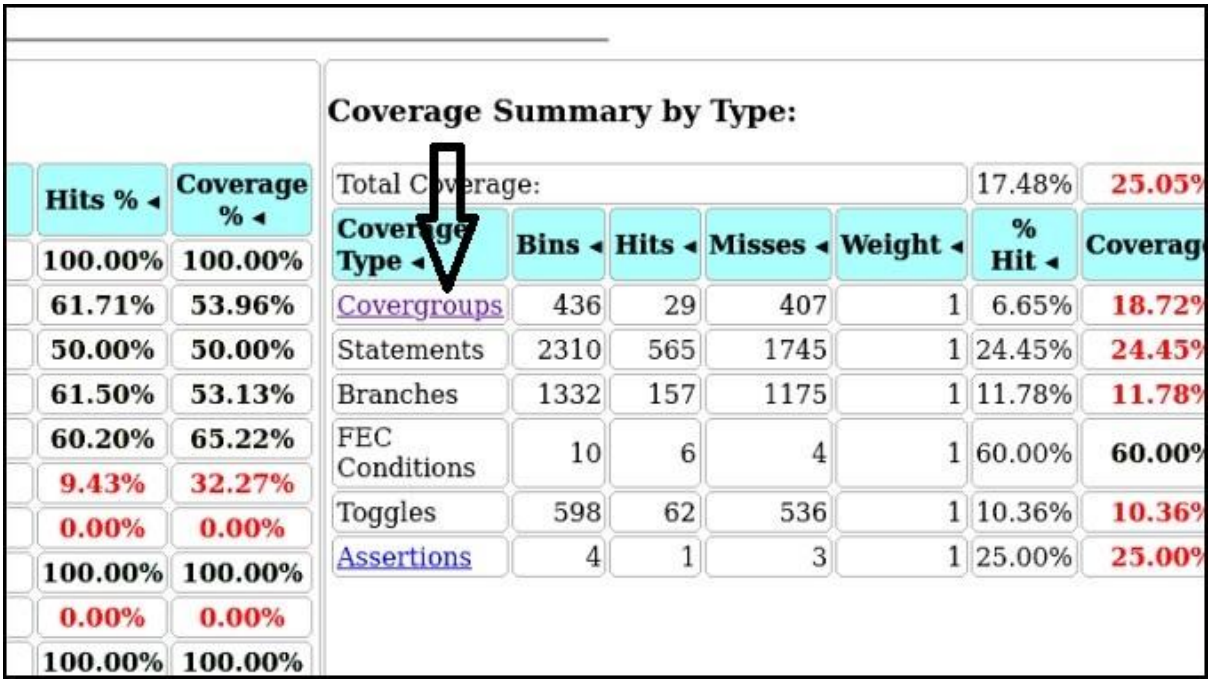
4. Search for the coverage (There it will be the full coverage) in the log file.

5. To check the individual coverage bins hit open the coverage report as shown :-

```
[mukultomar@HweServer questa_sim]$ firefox spi_simple_fd_8b_test/html_cov_report/index.html &
[1] 25492
```

Figure 65.

Then new html window will open



Coverage Summary by Type:

Hits %	Coverage %	Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverag
		Total Coverage:					17.48%	25.05%
100.00%	100.00%							
61.71%	53.96%	Covergroups	436	29	407	1	6.65%	18.72%
50.00%	50.00%	Statements	2310	565	1745	1	24.45%	24.45%
61.50%	53.13%	Branches	1332	157	1175	1	11.78%	11.78%
60.20%	65.22%	FEC Conditions	10	6	4	1	60.00%	60.00%
9.43%	32.27%	Toggles	598	62	536	1	10.36%	10.36%
0.00%	0.00%	Assertions	4	1	3	1	25.00%	25.00%
100.00%	100.00%							
0.00%	0.00%							
100.00%	100.00%							

Figure 66. HTML window showing all coverage

Here click on the covergroup there we can see the per instance created and inside that each coverpoint with bins is present there.

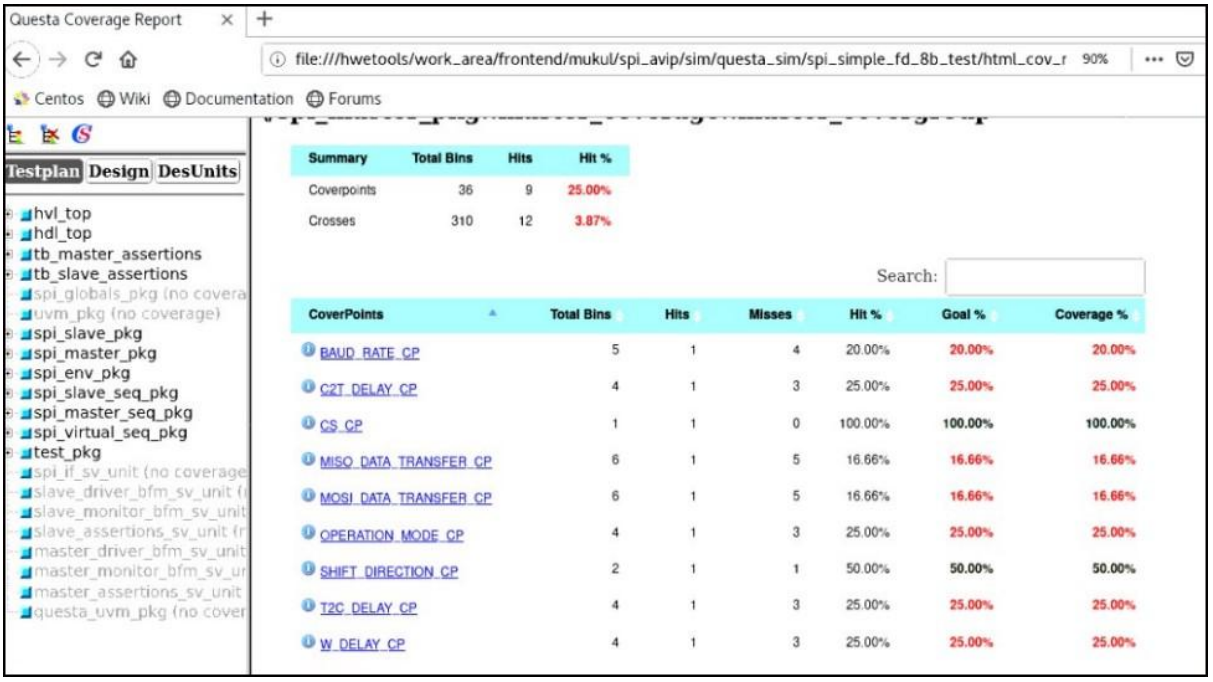


Figure 67. All coverpoints present in the Covergroup

Figure 68. Individual Coverpoint Hit

8.10 Errors

- The first error was with uvm subscriber was with superclass variable declaration i.e. we need to declare the variable handle as (T).
  This error is resolved when replacing the variable with the T which is declared in the super class i.e. uvm_subscriber.
- The 2nd error was the declaration of the handle which is declared to access the sample.

  This error we need to be carefull with the pure virtual function for the write which is in the uvm_subsciber so we cannot create the memory for it i.e. memory can be created for the extended class to use that function.

Chapter 9

Test Cases

9.1 Test Flow

In the test, there is virtual sequence and in virtual sequence, sequences are there, sequence_item get started in sequences, sequences will start in virtual sequence and virtual sequence will start in Test

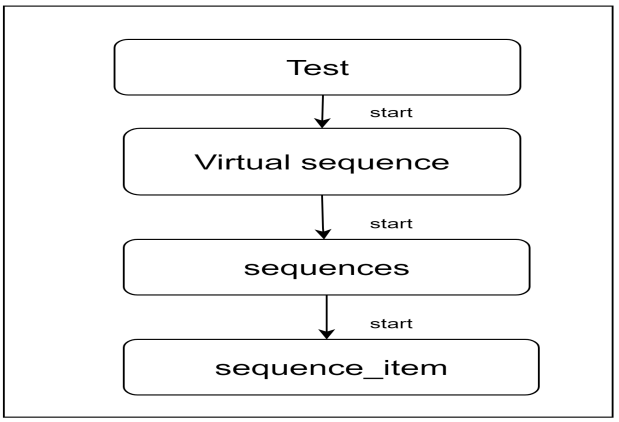


Figure 69.

Fig 9.1 Test flow

9.2 SPI Test Cases Flow Chart

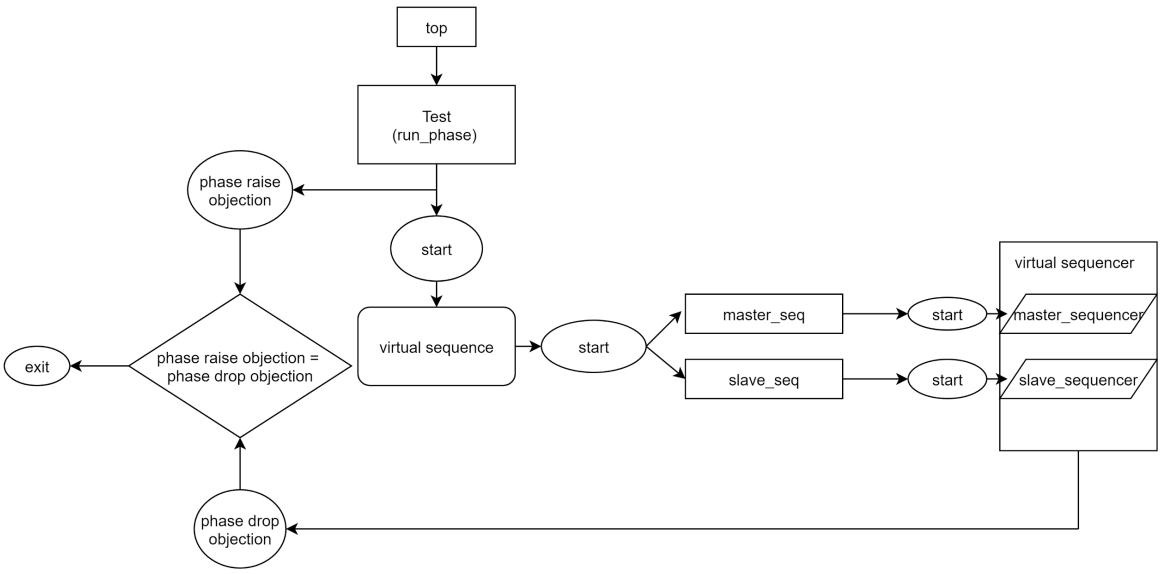


Figure 70.

Fig 9.2 SPI test cases flow chart

9.3 Transaction

Variables	Type	Description
cs	bit	Master asserts the chip select to select the slave device
master_out_slave_in	bit	Master Out ⇒ Slave In. Data leaves the master device and enters the slave device. MOSI lines on chip A are connected to MOSI lines on chip B.
master_in_slave_out	bit	Master In ⇐ Slave Out. Data leaves the slave device and enters the master device (or another slave. MISO lines on chip A are connected to MISO lines on chip B.

Table 28. Transaction variables

9.3.1 Master_tx

- ➢ Master_tx class is extended from the uvm_sequence_item holds the data items required to drive stimulus to dut
- ➢ Declared all the variables(cs, MOSI, MISO)

- ➢ Constraint declared for MOSI tx data between 0 to maximum upto 128 bytes.

```
constraint mosi_c { master_out_slave_in.size() > 0 ;
                    master_out_slave_in.size() < MAXIMUM_BITS/CHAR_LENGTH;}
```

Figure 71.

Fig 9.3 Constraint for mosi

Table 9.3.2 Describing constraint for mosi

Constraint	Description
mosi_c	Declaring master_out_slave_in value in between 0 and maximum upto 128 bytes(maximum_bits=1024 bits and character_length=8 bits).

Table 29.

- …...TODO….(dual_spi)
- Written functions for do_copy, do_compare, do_print methods, $casting is used to copy the data member values and compare the data member values and by using a printer , printing the MOSI and MISO values.

```
function void master_tx::do_copy (uvm_object rhs);
  master_tx rhs_;

  if(!$cast(rhs_,rhs)) begin
    `uvm_fatal("do_copy","cast of the rhs object failed")
  end
  super.do_copy(rhs);

  cs = rhs_.cs;
  master_in_slave_out = rhs_.master_in_slave_out;
  master_out_slave_in = rhs_.master_out_slave_in;

endfunction : do_copy
```

Figure 72.

Fig 9.4 do_copy method

```
function bit  master_tx::do_compare (uvm_object rhs,uvm_comparer comparer);
  master_tx rhs_;

  if(!$cast(rhs_,rhs)) begin
  `uvm_fatal("FATAL_MASTER_TX_DO_COMPARE_FAILED","cast of the rhs object failed")
  return 0;
  end

  return super.do_compare(rhs,comparer) &&
  master_out_slave_in== rhs_.master_out_slave_in &&
  master_in_slave_out== rhs_.master_in_slave_out;
endfunction : do_compare
```

Figure 73.

Fig 9.5 do_compare method

```
function void master_tx::do_print(uvm_printer printer);
  super.do_print(printer);
  printer.print_field( "cs", cs , 2,UVM_BIN);
  foreach(master_out_slave_in[i]) begin
    printer.print_field($sformatf("master_out_slave_in[%0d]",i),this.master_out_slave_in[i],
                                  8,UVM_HEX);
  end
  foreach(master_in_slave_out[i]) begin
    printer.print_field($sformatf("master_in_slave_out[%0d]",i),this.master_in_slave_out[i],
                                  8,UVM_HEX);
  end

endfunction : do_print
```

Figure 74.

Fig 9.6 do_print method

9.3.2 Slave_tx

- ➢ Declared all the variables(cs, MOSI, MISO)
- ➢ Constraint declared for MISO tx data between 0 maximum upto 128 bytes

```
constraint miso_c { master_in_slave_out.size() > 0 ;
                    master_in_slave_out.size() < MAXIMUM_BITS/CHAR_LENGTH;}
```

Figure 75.

Fig 9.7 Constraint for miso

Table 9.3.3 Describing constraint for miso

Constraint	Description
miso_c	Declaring master_in_slave_out value in between 0 and maximum upto 128 bytes(maximum_bits=1024 bits and character_length=8 bits).

Table 30.

- …..TO DO……(dual_spi)
- Written functions for do_copy, do_compare, do_print methods, $casting is used to copy the data member values and compare the data member values and by using a printer , printing the MOSI and MISO values.

9.4 Sequences

A UVM Sequence is an object that contains a behavior for generating stimulus. A sequence generates a series of sequence_item's and sends it to the driver via sequencer, Sequence is written by extending the uvm_sequence.

9.4.1 Methods

Method	Description
new	Creates and initializes a new sequence object
start_item	This method will send the request item to the sequencer, which will forward it to the driver
req.randomize()	Generate the transaction(seq_item).
finish_item	Wait for acknowledgement or response

Table 31. Sequence methods

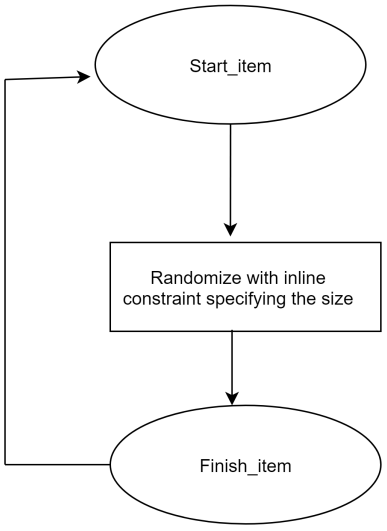


Figure 76.

Fig 9.8 Flow chart for sequence methods

Data transfers	spi_fd_8b_master_seq	spi_fd_8b_slave_seq spi_fd_64b_slave_seq	Extended from base sequence. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomizing the req with the miso size.(size is 1 for 8b) and selecting number of slaves
	spi_fd_16b_master_seq	spi_fd_16b_slave_seq	Extended from base sequence. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomizing the req with the miso size.(size 2 for 16b) and selecting number of slaves
	spi_fd_24b_master_seq	spi_fd_24b_slave_seq	Extended from base sequence. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomizing the req with the miso size.(size 3 for 24b) and selecting number of slaves
	spi_fd_32b_master_seq	spi_fd_32b_slave_seq	Extended from base sequence. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomizing the req with the miso size.(size 4 for 32b) and selecting number of slaves
	spi_fd_64b_master_seq	spi_fd_64b_slave_seq	Extended from base sequence. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomizing the req with the miso size.(size 8 for 64 b) and selecting number of slaves
Continuous transfer	spi_fd_8b_ct_master_seq	spi_fd_8b_ct_slave_seq	Extend the class from master_ base sequence and create and randomize the req with size 1 (checking for 8 bits) selecting one slave and chip select should be low for continuous transfer

Discontinuous transfer	spi_fd_dct_master_seq	spi_fd_dct_slave_seq	Extend the class from master_ base sequence and create and randomize the req with size 1 (checking for 8 bits) selecting one slave and chip select may differ for dis -continuoustransfer
Maximum bits	spi_fd_maximum_bits_master_seq	spi_fd_maximum_bits_slave_seq	Extended from the base class and created and randomized the req with maximum bits divided to character length then printing the req.
Configurations	spi_fd_cpol0_cpha0_master_seq	spi_fd_cpol0_cpha0_slave_seq	Extend the class from master_ base sequence and create and randomize the req with size 1 (checking for 8 bits) selecting one slave and chip select. Then print req
	spi_fd_cpol0_cpha1_master_seq	spi_fd_cpol0_cpha1_slave_seq	Extend the class from master_ base sequence and create and randomize the req with size 1 (checking for 8 bits) selecting one slave and chip select. Then print req
	spi_fd_cpol1_cpha0_master_seq	spi_fd_cpol1_cpha0_slave_seq	Extend the class from master_ base sequence and create and randomize the req with size 1 (checking for 8 bits) selecting one slave and chip select. Then print req
	spi_fd_cpol1_cpha1_master_seq	spi_fd_cpol1_cpha1_slave_seq	Extend the class from master_ base sequence and create and randomize the req with size 1 (checking for 8 bits) selecting one slave and chip select. Then print req
Delays	spi_fd_ct2_master_seq	spi_fd_ct2_slave_seq	Extend the class from master_ base sequence and create and randomize the req with size 1, and selecting one slave and chip select. Then print req
	spi_fd_t2c_master_seq	spi_fd_t2c_slave_seq	Extend the class from master_ base sequence and create and randomize the req with size 1, and selecting one slave and chip select. Then print req
Shift directions	spi_fd_msb_master_seq	spi_fd_msb_slave_seq	Extend the class from master_ base sequence and create and randomize the req with size 1, and selecting one slave and chip select. Then print req
	spi_fd_lsb_master_seq	spi_fd_lsb_slave_seq	Extend the class from master_ base sequence and create and randomize the req with size 1, and selecting one slave and chip select. Then print req

Cross testing	spi_fd_cross_master_seq	spi_fd_cross_slave_seq	Extend the class from master_ base sequence and create and randomize the req with size 1, and selecting one slave and chip select. Then print req

Table 32. Describing master and slave sequences

In master_seq body creating req and start item will start seq and randomizing the req with inline constraint and selecting slave then print req followed by finish item

```
task spi_fd_8b_master_seq::body();
  req = master_tx::type_id::create("req");

  start_item(req);

  if(!req.randomize() with {req.master_out_slave_in.size() == 1;
                            // Selecting only one slave
                            $countones(req.cs) == NO_OF_SLAVES - 1;
                            // Selecting slave 0
                            req.cs[0] == 0;
                           }) begin
    `uvm_fatal(get_type_name(),"Randomization failed")
  end

  req.print();
  finish_item(req);

endtask:body
```

Figure 77.

Fig 9.9 Master seq body method

In slave_seq body creating req and start item will start seq and randomizing the req with inline constraint and print req followed by finish item

```
task spi_fd_8b_slave_seq::body();
  req=slave_tx::type_id::create("req");
  repeat(5) begin
    start_item(req);
    if(!req.randomize() with { req.master_in_slave_out.size()==1;})
      `uvm_fatal(get_type_name(),"Randomization FAILED")
      //req.print();
      finish_item(req);
    end

endtask : body
```

Figure 78.

Fig 9.10 Slave seq body method

9.5 Virtual sequences

A virtual sequence is a container to start multiple sequences on different sequencers in the environment. This virtual sequence is usually executed by a virtual sequencer which has handles to real sequencers. This need for a virtual sequence arises when you require different sequences to be run on different environments.

Virtual sequence base class

Virtual sequence base class is extended from uvm_sequence and parameterized with uvm_transaction. Declaring p_sequencer as macro , handles virtual sequencer and master, slave sequencer and environment config.

```
class spi_fd_virtual_seq_base extends uvm_sequence#(uvm_sequence_item);
  `uvm_object_utils(spi_fd_virtual_seq_base)

   //p sequencer macro declaration
   `uvm_declare_p_sequencer(virtual_sequencer)


  //------------------------------------------------------------------------------
  // declaring handles for master and slave sequencer and environment config
  //------------------------------------------------------------------------------
  master_sequencer  master_seqr_h;
  slave_sequencer   slave_seqr_h;
  env_config env_cfg_h;

  //------------------------------------------------------------------------------
  // Externally defined tasks and functions
  //------------------------------------------------------------------------------
  extern function new(string name="spi_fd_virtual_seq_base");
  extern task body();

endclass:spi_fd_virtual_seq_base
```

Figure 79.

Fig 9.11 Virtual base sequence

In virtual sequence body method,Getting the env configurations and Dynamic casting of p_sequencer and m_sequencer .Connect the master sequencer and slave sequencer in p_sequencer with local master sequencer and slave sequencer.

```
task spi_fd_virtual_seq_base::body();
  if(!uvm_config_db#(env_config) ::get(null,get_full_name(),"env_config",env_cfg_h)) begin
    `uvm_fatal("CONFIG","cannot get() env_cfg from uvm_config_db.Have you set() it?")
  end


  //dynamic casting of p_sequncer and m_sequencer
  if(!$cast(p_sequencer,m_sequencer))begin
    `uvm_error(get_full_name(),"Virtual sequencer pointer cast failed")
  end

  //connecting master sequencer and slave sequencer present in p_sequencer to
  // local master sequencer and slave sequencer
  master_seqr_h=p_sequencer.master_seqr_h;
  slave_seqr_h=p_sequencer.slave_seqr_h;

endtask:body
```

Figure 80.

Fig 9.12 Virtual base sequence body

In the virtual sequence body method, creating master and slave sequence handles and starts the slave sequence within fork join_none and master sequence within repeat statement.

```
task spi_fd_8b_virtual_seq::body();
 super.body(); //Sets up the sub-sequencer pointer

   //crearions master and slave sequence handles here
   spi_fd_8b_master_seq_h=spi_fd_8b_master_seq::type_id::create("spi_fd_8b_master_seq_h");
   spi_fd_8b_slave_seq_h=spi_fd_8b_slave_seq::type_id::create("spi_fd_8b_slave_seq_h");



  fork

     //starting slave sequencer with respective to p_sequencer declared in virtual seq base
     forever begin : SLAVE_SEQ_START
       spi_fd_8b_slave_seq_h.start(p_sequencer.slave_seqr_h);
     end
    join_none

   //starting master sequencer with respective to p_sequencer declared in virtual seq base
   repeat(5) begin : MASTER_SEQ_START
     spi_fd_8b_master_seq_h.start(p_sequencer.master_seqr_h);
   end

endtask: body
```

Figure 81.

Fig 9.13 Virtual 8bit sequence body

Sections	Virtual sequences	Description
Data transfer	spi_fd_8b_virtual_seq	Inside the 8bit virtual sequence, extending from base class. Declaring handles of sequences and inside body method constructing handles of sequence. Configuring the master and slave sequencers.
	spi_fd_16b_virtual_seq	Extended from virtual base sequence and declaring handles of master and slave sequencers . In the sequence body method start the slave and master sequences with p_sequencer,,master can be repeated for five times (five data packets) slave can be started within fork join_none.
	spi_fd_32b_virtual_seq	Extended from virtual base sequence and declaring handles of master and slave sequencers . start the slave and master sequences with p_sequencer ,master can be repeated for five times (five data packets) slave can be started within fork join_none
	spi_fd_64b_virtual_seq	Extended from virtual base sequence and declaring handles of master and slave sequencers . start the slave and master sequences with p_sequencer ,master can be repeated for five times (five data packets) slave can be started within fork join_none
	spi_fd_maximum_bits_virtual_seq	Extended from virtual base sequence and declaring handles of master and slave sequencers . start the slave and master sequences with p_sequencer ,master can be repeated for five times (five data packets) slave can be started within fork join_none.
Continuous transfer	spi_fd_8b_ct_virtual_seq	Extended from virtual base sequence and declaring handles of master and slave sequencers .start the slave and master sequences with p_sequencer ,master can be repeated for five times (five data packets) slave can be started within fork join_none. Data can transfer continuously without cs becoming high in between data transfer.
Discontinuous transfers	spi_fd_dct_virtual_seq	Extended from virtual base sequence and declaring handles of master and slave sequencers . start the slave and master sequences with p_sequencer ,master can be repeated for five times (five data packets) slave can be started within fork join_none. Data can transfer discontinuously, cs becomes high in between data transfer.

Delays	spi_fd_c2t_virtual_seq	Extended from virtual base sequence and declaring handles of master and slave sequencers . start the slave and master sequences with p_sequencer. c2t delay is from cs high to first posedge of sclk there should be 2sclk delay(when c2t=2).
	spi_fd_t2c_virtual_seq	Extended from virtual base sequence and declaring handles of master and slave sequencers . start the slave and master sequences with p_sequencer. t2c delay is in between the last edge of sclk and active high cs there should be 2sclk delay(when t2c=2).
	spi_fd_wdelay_virtual_seq	Extended from virtual base sequence and declaring handles of master and slave sequencers . start the slave and master sequences with p_sequencer.Duration of wdelay will be equal to the time for which chip select is high between two transfers.
	spi_fd_baudrate_virtual_seq	Extended from virtual base sequence and declaring handles of master and slave sequencers . start the slave and master sequences with p_sequencer Baud Rate is the rate of each transfer(when baud rate is 2, 1sclk will have 2 pclk)
Shift directions	spi_fd_msb_virtual_seq	Extended from virtual base sequence and declaring handles of master and slave sequencers . start the slave and master sequences with p_sequencer Checking for msb first in data transfer(Ex:'h98-9 is msb)
	spi_fd_lsb_virtual_seq	Extended from virtual base sequence and declaring handles of master and slave sequencers . start the slave and master sequences with p_sequencer Checking for lsb bit first(Ex:'h98- 8 is lsb)
Configurations	spi_fd_cpol0_cpha0_virtual_seq	Extended from virtual base sequence and declaring handles of master and slave sequencers . start the slave and master sequences with p_sequencer. When cpol =0.cpha=0 miso should happen on the trailing edge and mosi should happen on the leading edge.
	spi_fd_cpol0_cpha1_virtual_seq	Extended from virtual base sequence and declaring handles of master and slave sequencers . start the slave and master sequences with p_sequencer. When cpol=0.cpha=1 miso should happen on the leading edge and mosi should happen on the trailing edge.

	spi_fd_cpol1_cpha0_virtual_seq	Extended from virtual base sequence and declaring handles of master and slave sequencers . start the slave and master sequences with p_sequencer. When cpol=1.cpha=0 miso should happen on the trailing edge and mosi should happen on the leading edge.
	spi_fd_cpol1_cpha1_virtual_seq	Extended from virtual base sequence and declaring handles of master and slave sequencers . start the slave and master sequences with p_sequencer. When cpol=1.cpha=1 miso should happen on the leading edge and mosi should be on the trailing edge
Cross test	spi_fd_cross_virtual_seq	Extended from virtual base sequence and declaring handles of master and slave sequencers . start the slave and master sequences with p_sequencer. cross testing is checking for cpol0, cpha1, msb bit first with c2t ,t2c,wdelay and baud rate and number of slaves.(where c2t=2,t2c=2,baud rate=4,wdelay=2)

Table 33. Describing virtual sequences

9.6 Test Cases

The uvm_test class defines the test scenario and verification goals.

A) In base test, declaring the handles for environment config and environment class.

```
class base_test extends uvm_test;
  `uvm_component_utils(base_test)
  // Variable: env_cfg_h
  // Declaring environment config handle
  env_config env_cfg_h;

  // Variable: env_h
  // Handle for environment
  env env_h;


  //-------------------------------------------------------
  // Externally defined Tasks and Functions
  //-------------------------------------------------------
  extern function new(string name = "base_test", uvm_component parent = null);
  extern virtual function void build_phase(uvm_phase phase);
  extern virtual function void setup_env_cfg();
  extern virtual function void setup_master_agent_cfg();
  extern virtual function void setup_slave_agents_cfg();
  extern virtual function void end_of_elaboration_phase(uvm_phase phase);
  extern virtual task run_phase(uvm_phase phase);

endclass : base_test
```

Fig 9.14 Base test

Figure 82.

B) In build phase, calling the setup_env_cfg and constructing the environment handle

C) Inside setup_env_cfg function, constructing the environment config class handle. With the help of this env_cfg_h handle all the required fields in the config class have been set up with respective values and then calling the setup_master_agent_config and setup_slave_agent_config functions.

```
function void base_test::setup_env_cfg();
 //env_cfg_h = env_config::type_id::create("env_cfg_h");
 env_cfg_h.no_of_slaves = NO_OF_SLAVES;
 env_cfg_h.has_scoreboard = 1;
 env_cfg_h.has_virtual_seqr = 1;

 // Setup the master agent cfg
 //env_cfg_h.master_agent_cfg_h = master_agent_config::type_id::create("master_agent_cfg_h");
 setup_master_agent_cfg();
 uvm_config_db #(master_agent_config)::set(this,"*master_agent*","master_agent_config",env_cfg_h.master_agent_cfg_h);
 env_cfg_h.master_agent_cfg_h.print();

 // Setup the slave agent(s) cfg
 env_cfg_h.slave_agent_cfg_h = new[env_cfg_h.no_of_slaves];
 foreach(env_cfg_h.slave_agent_cfg_h[i]) begin
   env_cfg_h.slave_agent_cfg_h[i] = slave_agent_config::type_id::create($sformatf("slave_agent_cfg_h[%0d]",i));
 end
 setup_slave_agents_cfg();
 foreach(env_cfg_h.slave_agent_cfg_h[i]) begin
   uvm_config_db #(slave_agent_config)::set(this,$sformatf("*slave_agent_h[%0d]*",i),
                                            "slave_agent_config", env_cfg_h.slave_agent_cfg_h[i]);
  env_cfg_h.slave_agent_cfg_h[i].print();
 end

 // set method for env_cfg
 uvm_config_db #(env_config)::set(this,"*","env_config",env_cfg_h);
 env_cfg_h.print();
endfunction: setup_env_cfg
```

Figure 83.

Fig 9.15 Setup env_cfg

D) In setup_master_agent_config function, master_agent_config class handle which is in env_config class has been constructed with the help of this handle all the required fields(spi_modes.shift_directions,c2t,t2c,wdelay,has_coverage,no.of slaves,is_active) in master_agent_config class has been setup.

```
function void base_test::setup_master_agent_cfg();
 //env_cfg_h.master_agent_cfg_h = master_agent_config::type_id::create("master_agent_cfg_h");
 // Configure the Master agent configuration
 env_cfg_h.master_agent_cfg_h.is_active           = uvm_active_passive_enum'(UVM_ACTIVE);
 env_cfg_h.master_agent_cfg_h.no_of_slaves        = NO_OF_SLAVES;
 env_cfg_h.master_agent_cfg_h.spi_mode            = operation_modes_e'(CPOL0_CPHA0);
 env_cfg_h.master_agent_cfg_h.shift_dir           = shift_direction_e'(LSB_FIRST);
 env_cfg_h.master_agent_cfg_h.c2tdelay            = 1;
 env_cfg_h.master_agent_cfg_h.t2cdelay            = 1;
 env_cfg_h.master_agent_cfg_h.has_coverage        = 1;

 // baudrate_divisor_divisor = (secondary_prescalar+1) * (2 ** (primary_prescalar+1))
 // baudrate = busclock / baudrate_divisor_divisor;
 env_cfg_h.master_agent_cfg_h.set_baudrate_divisor(.primary_prescalar(0), .secondary_prescalar(0));

// uvm_config_db #(master_agent_config)::set(this,"*master_agent*","master_agent_config",env_cfg_h.master_agent_cfg_h);
//env_cfg_h.master_agent_cfg_h.print();
endfunction: setup_master_agent_cfg
```

Figure 84.

Fig 9.16 Master_agent_cfg setup

E) In setup_slave_agent_config function, for each slave agent configuration trying to construct slave_agent_config class handle which is in env_config class with the help of this handle all the required fields(slave_id,shift_directions,spi_mode,has_coverage,is_active) in slave_agent_config class has been setup Followed by the end of the elaboration phase used to print the topology.

```
function void base_test::setup_slave_agents_cfg();
 // Create slave agent(s) configurations
 // env_cfg_h.slave_agent_cfg_h = new[env_cfg_h.no_of_slaves];
 // Setting the configuration for each slave
 foreach(env_cfg_h.slave_agent_cfg_h[i]) begin
   //env_cfg_h.slave_agent_cfg_h[i] = slave_agent_config::type_id::create($sformatf("salve_agent_cfg_h[%0d]",i));
   env_cfg_h.slave_agent_cfg_h[i].slave_id     = i;
   env_cfg_h.slave_agent_cfg_h[i].is_active    = uvm_active_passive_enum'(UVM_ACTIVE);
   env_cfg_h.slave_agent_cfg_h[i].spi_mode     = operation_modes_e'(CPOL0_CPHA0);
   env_cfg_h.slave_agent_cfg_h[i].shift_dir    = shift_direction_e'(LSB_FIRST);
   env_cfg_h.slave_agent_cfg_h[i].has_coverage = 1;

 end

endfunction: setup_slave_agents_cfg
```

Figure 85.

Fig 9.17 Slave_agent_cfg setup

Extend the 8bit_test from base test and declare virtual sequence handle then create virtual sequence in test, and start the virtual sequence in phase, raise and drop objection.

```
class spi_simple_fd_8b_test extends base_test;

  //Registering the spi_simple_fd_8b_test in the factory
  `uvm_component_utils(spi_simple_fd_8b_test)

  //-------------------------------------------------------
  // Declaring sequence handles
  //-------------------------------------------------------
  spi_fd_8b_virtual_seq spi_fd_8b_virtual_seq_h;


  //-------------------------------------------------------
  // Externally defined Tasks and Functions
  //-------------------------------------------------------
  extern function new(string name = "spi_simple_fd_8b_test", uvm_component parent);
  extern function void build_phase(uvm_phase phase);
  extern task run_phase(uvm_phase phase);

endclass : spi_simple_fd_8b_test
```

Figure 86.

Fig 9.18 Example for 8bit test

```
task spi_simple_fd_8b_test::run_phase(uvm_phase phase);
  spi_fd_8b_virtual_seq_h = spi_fd_8b_virtual_seq::type_id::create("spi_fd_8b_virtual_seq_h");

  phase.raise_objection(this);

  //vseq1_fd_8b_h.start(env_h.vseqr);
  spi_fd_8b_virtual_seq_h.start(env_h.virtual_seqr_h);

  phase.drop_objection(this);

endtask : run_phase
```

Figure 87.

Fig 9.19 Run_phase of 8bit_test

sections	Test Names	Description
Data transfers	spi_simple_fd_8b_test	Extend test from base test and declare virtual sequence then create virtual sequence in run_phase, and start the virtual sequence with virtual sequencer handle in phase, raise and drop objections.

		For 8bit it generates only miso[0] and mosi[0]
	Spi_simple_fd_16b_test	Extend test from base test and declare virtual sequence then create virtual sequence handle in run_phase, and start the virtual sequence with virtual sequencer handle in phase, raise and drop objection.For 16bit it generates the data as mosi[0],mosi[1]and miso[0],miso[1]
	spi_simple_fd_32b_test	Extend test from base test and declare virtual sequence then create virtual sequence in run_phase, and start the virtual sequence with virtual sequencer handle in phase, raise and drop objections.For 32bit it generates data as mosi[0],miso[1],mosi[2],mosi[3] and miso[0],miso[1].mosi[2],miso[3]
	spi_simple_fd_64b_test	Extend test from base test and declare virtual sequence then create virtual sequence in run_phase, and start the virtual sequence with virtual sequencer handle in phase, raise and drop objections.For 16bit it generates the data as mosi[0],miso[1],mosi[2],mosi[3],mosi[4],mosi[5],mosi[6],mosi[7] and miso[0],miso[1].miso[2],miso[3],miso[4].miso[5],miso[6],miso[7].
	spi_simple_fd_maximum_bits_test	Extend test from base test and declare virtual sequence then create virtual sequence in run_phase, and start the virtual sequence with virtual sequencer handle in phase, raise and drop objections.
Continuous transfers	spi_simple_fd_8b_ct_test	Extended the test from base test and declare virtual sequence then create virtual sequence in run_phase, and start the virtual sequence with virtual sequencer handle in phase, raise and drop objections. continuously generates the transfer of bits
Discontinuous transfers	spi_simple_fd_dct_test	Extended the test from base test and declare virtual sequence then create virtual sequence in test, and start the virtual sequence in phase, raise and drop objection.
Shift directions	spi_simple_msb_test	Extended the test from base test and then master agent and slave agent configurations and then set configuration for each slave by the shift direction.
	spi_simple_lsb_test	Extended the test from base test and then master agent and slave agent configurations and then set configuration for each slave by the shift direction.
Configurations	spi_simple_fd_cpol0_cpha0_test	Extended test from base test and then master agent and slave agent configurations and then set configuration for each slave by the operation mode.When cpol =0.cpha=0 miso should happen on the trailing edge and mosi should happen on the leading edge

	spi_simple_fd_cpol0_cpha1_test	Extended test from base test and then master agent and slave agent configurations and then set configuration for each slave by the operation mode.When cpol=0.cpha=1 miso should happen on the leading edge and mosi should happen on the trailing edge
	spi_simple_fd_cpol0_cpha1_test	Extended test from base test and then master agent and slave agent configurations and then set configuration for each slave by the operation mode.When cpol=1.cpha=0 miso should happen on the trailing edge and mosi should happen on the leading edge.
	spi_simple_fd_cpol1_cpha0_test	Extended test from base test and then master agent and slave agent configurations and then set configuration for each slave by the operation mode
	spi_simple_fd_cpol1_cpha1_test	Extended test from base test and then master agent and slave agent configurations and then set configuration for each slave by the operation mode.When cpol=1.cpha=1 miso should happen on the leading edge and mosi should be on the trailing edge
Delays	spi_simple_fd_c2t_test	Extended the test from base test and then configure master agent configuration and then set c2t delay
	spi_simple_fd_t2c_test	Extended the test from base test and then configure master agent configuration and then set t2c delay
	spi_simple_fd_wdelay_test	Extended the test from base test and then configure master agent configuration and set wdelay.
	spi_simple_fd_baudrate_test	Extended the test from base test and then configure master agent configuration and set Baud rate.
Cross testing	spi_simple_fd_cross_test	Extended the test from base test and then configure master agent configuration and check for cross testing like cpol0_cpha1,msb first,c2t,t2c,wdelay,baud rate and number of slaves

Table 34. Describing Test cases

9.7 Testlists

Regression list for Simple SPI - Full duplex

TestCase Names	Description
#Directed test cases	

spi_simple_fd_8b_test	Checking for 8bit data transfer
spi_simple_fd_16b_test	Checking for 16bit data transfer
spi_simple_fd_32b_test	Checking for 32bit data transfer
spi_simple_fd_64b_test	Checking for 64bit data transfer
spi_simple_fd_maximum_bits_test	Checking for maximum bits (128) data transfer
spi_simple_fd_ct_test	Checking for continuous data transfer(mosi followed by miso)with character length
spi_simple_fd_dct_test	Checking for discontinuous data transfer(mosi followed by miso depends on cs)with different character length.
spi_simple_fd_msb_test	Checking for the msb first in 8bit data transfer
spi_simple_fd_lsb_test	Checking for the lsb first in 8bit data transfer
spi_simple_fd_cpol0_cpha0_test	When cpol is low, sclk should start from idle state to active high clock pulse, when cphase is low miso should happen on trailing edge and mosi should happen on leading edge.
spi_simple_fd_cpol0_cpha1_test	When cpol is low sclk should starts from idle state to active high clock pulse, when cphase is high miso should be happen on leading edge and mosi should happen on trailing edge
spi_simple_fd_cpol1_cpha0_test	When cpol is high sclk should starts from active high clock pulse to idle state, when cphase is low miso should be happen on trailing edge and mosi should happen on leading edge
spi_simple_fd_cpol1_cpha1_test	When cpol is high sclk should starts from active high clock pulse to idle state, when cphase is high miso should be happen on leading edge and mosi should happen on trailing edge
spi_simple_fd_c2t_delay_test	Check the Delay between active low cs and first edge of sclk .If c2t=1 there should be 1sclk delay
spi_simple_fd_t2c_delay_test	Check the Delay between last edge of sclk and active high cs. If t2c=1 there should be 1sclk delay
spi_simple_fd_baudrate_test	Check the baud rate with respect to primary and secondary prescaler. when baudrate=2,1sclk=2 pclks
#Cross testing	
spi_simple_fd_cross_test	Checking the cross test for configurations, shift direction, and delays for fixed values(cpol0_cpha1, c2t=2, t2c=2, wdelay=2, baud rate)

Table 35. Testlists

Chapter 10

User Guide

The user guide is the document that explains how to run tests on different platforms like Questa sim, cadence, and synopsis and also explains how to view waves, coverage.

Chapter 11

References