2022

PULPINO SUBSYSTEM VERIFICATION

Contents

List of Table

List of Figures

List of Abbreviations

Abbreviation	Description
uvm	universal verification methodology
axi4	advanced peripheral bus
avip	accelerated verification intellectual property
hdl	hardware descriptive language
hvl	hardware verification language
bfm	bus functional model
tlm	transaction level modelling
pclk	System clock

Chapter 1

Introduction

PULP(Parallel Ultra Low Power) Subsystem is a reusable system that consists of the peripherals axi4, AXI4 and SPI. It supports both the RISC-V RI5CY and zero-risc v core.

The SPI master Pulpino subsystem is a low frequency device that uses an axi4 bus of 32 bits wide as a bridge to connect to the SPI master. For data-storage, the subsystem uses a memory-map which will be controlled by the assigned addresses. It includes a FIFO to store the data that is received and needs to be transmitted via the TX and RX registers. It contains an event unit that can put the core to sleep and wake it up when there is an event or interrupt from a peripheral.

1.1 Key features

1. Standard SPI mode
2. Full Duplex System
3. Single master SPI and single slave SPI
4. Single master axi4 and single slave axi4
5. Serial transfer in SPI
6. Clock divider
7. Asynchronous active low hard reset
8. Software reset

9. SPI transfer length
10. Appending Dummy Data
11. Separate TXFIFO and RXFIFO
12. Interrupt Configuration

Chapter 2

Architecture

2.1 Pulpino SPI Master SUBSYSTEM Verification 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 the RTL. so, it is named HDL TOP and the unsynthesizable part is pushed into the HVL TOP. As 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 mode of operation.

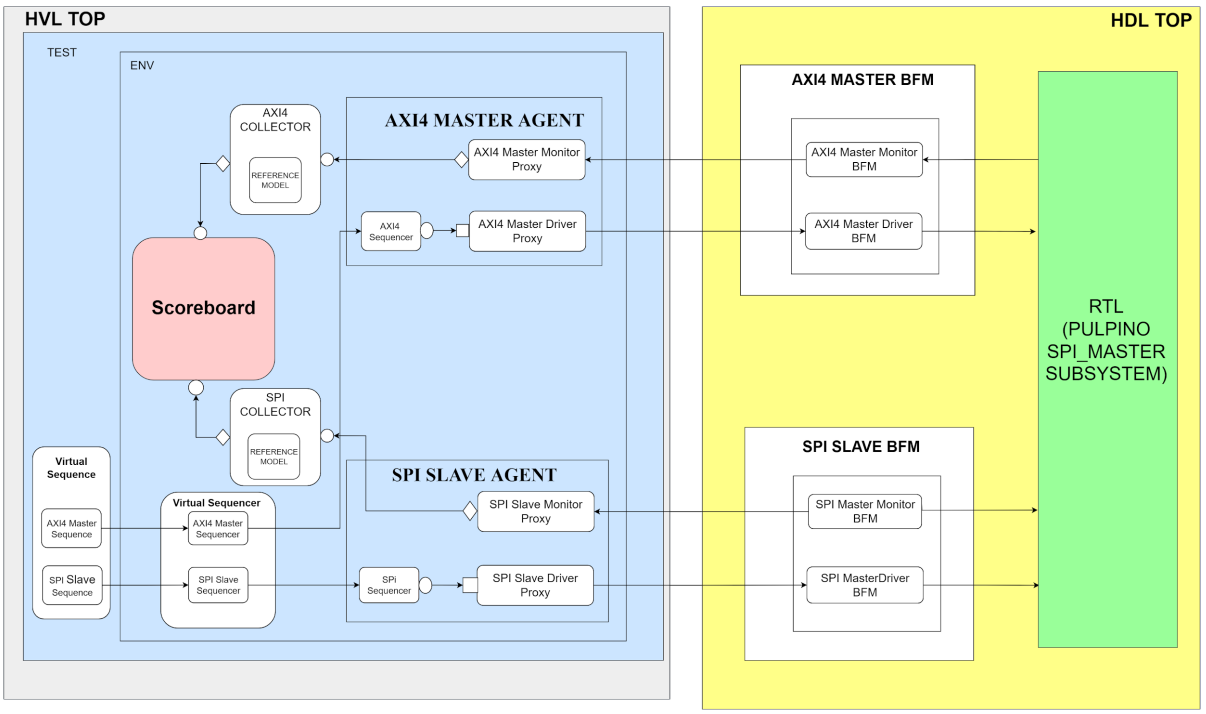


Fig 2.1 Pulpino_SPI_Master_Subsystem_Verification AVIP Architecture

HVL TOP has the design which is untimed and the transactions flow from both master virtual sequence and slave virtual sequence onto the axi4 and 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 its proxy to call non-blocking methods which are defined in the proxy.

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 and 3.2 shows the axi4 pins and SPI pins respectively used to interface to external devices.

Signals	Source	Description
aclk	Clock source	System generated clock
areset_n	System bus equivalent	It is an active low reset generated by system
awid	AXI4 bridge	This signal is the identification tag for the write address group of signals.
awaddr	AXI4 bridge	The write address gives the address of the first transfer in a write burst transaction.

awlen	AXI4 bridge	The burst length gives the exact number of transfers in a burst. This information determines the number of data transfers associated with the address.
awsize	AXI4 bridge	This signal indicates the size of each transfer in the burst.
awburst	AXI4 bridge	The burst type and the size information, determine how the address for each transfer within the burst is calculated.
awlock	AXI4 bridge	Provides additional information about the atomic characteristics of the transfer.
awcache	AXI4 bridge	This signal indicates how transactions are required to progress through a system.
awprot	AXI4 bridge	This signal indicates the privilege and security level of the transaction, and whether the transaction is a data access or an instruction access.
awqos	AXI4 bridge	The qos identifier is sent for each write transaction.
awregion	AXI4 bridge	Permits a single physical interface on a slave to be used for multiple logical interfaces.
awuser	AXI4 bridge	Optional User-defined signal in the write address channel.
awvalid	AXI4 bridge	This signal indicates that the channel is signalling valid write address and control information.
awready	AXI4 bridge	This signal indicates that the slave is ready to accept an address and associated control signals.
wdata		Write data.
wstrb	AXI4 bridge	This signal indicates which byte lanes hold valid data. There is one write strobe bit for each eight bits of the write data bus.
wlast	AXI4 bridge	This signal indicates the last transfer in a write burst.
wuser	AXI4 bridge	Optional User-defined signal in the write data channel.
wvalid	AXI4 bridge	This signal indicates that valid write data and strobes are available.
wready	AXI4 bridge	This signal indicates that the slave can accept the write data.

Table 3.1 axi4 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_	input	input	1 bit	It is an active low reset

n				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 synchronise 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

Table 3.2 SPI pins used to interface to external devices

3.2 Testbench Components

The testbench components of the PULPINO SPI MASTER SUBSYSTEM verification-avip are discussed below.

3.2.1 PULPINO SPI MASTER SUBSYSTEM Verification Hdl Top

Hdl top is synthesizable, where generation of the clock and reset is done. Instantiation of the axi4 interface handle, master agent bfm handle and slave agent bfm handle is done as shown in Figure 3.1.

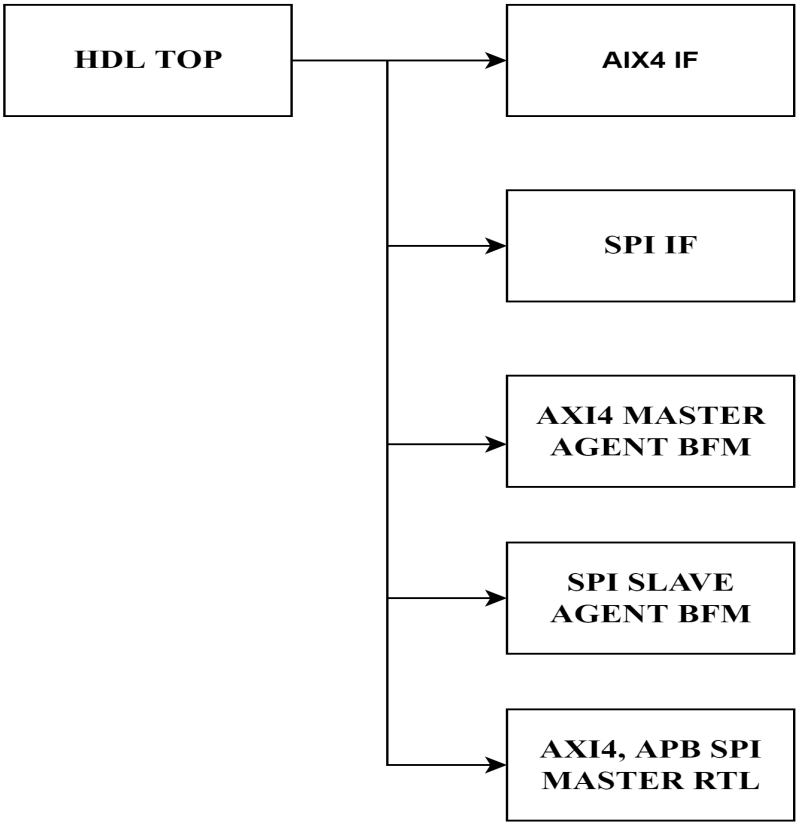


Fig. 3.1 HDL Top

3.2.2 AXI4 Interface

Importing the global packages
Passing Signals: aclk, areset_n
Declaration of signals: All five channels(write address, write data, write response, read address, and read data) are declared as logic type.

3.2.3 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.

3.2.4 AXI Master Agent BFM Module

Instantiates the below two interfaces here

a) axi4 master driver bfm and
b) axi4 master monitor bfm.

Instantiates the axi4 master assertions and binds it with the axi4 master monitor bfm handle and maps the signals of axi4 master assertions with the axi4 interface signals. The axi4 interface signals are passed to the axi4 master driver and monitor bfm in instantiations as shown in fig. 3.2 and fig.3.3

```
axi4_master_driver_bfm axi4_master_drv_bfm_h (.aclk(intf.aclk),
                                              .aresetn(intf.aresetn),
                                              .awid(intf.awid),
                                              .awaddr(intf.awaddr),
                                              .awlen(intf.awlen),
                                              .awsize(intf.awsize),
                                              .awburst(intf.awburst),
                                              .awlock(intf.awlock),
                                              .awcache(intf.awcache),
                                              .awprot(intf.awprot),
                                              .awqos(intf.awqos),
                                              .awregion(intf.awregion),
                                              .awuser(intf.awuser),
                                              .awvalid(intf.awvalid),
                                              .awready(intf.awready),
                                              .wdata(intf.wdata),
                                              .wstrb(intf.wstrb),
                                              .wlast(intf.wlast),
                                              .wuser(intf.wuser),
                                              .wvalid(intf.wvalid),
                                              .wready(intf.wready),
                                              .bid(intf.bid),
                                              .bresp(intf.bresp),
                                              .buser(intf.buser),
                                              .bvalid(intf.bvalid),
                                              .bready(intf.bready),
                                              .arid(intf.arid),
                                              .araddr(intf.araddr),
                                              .arlen(intf.arlen),
                                              .arsize(intf.arsize),
                                              .arburst(intf.arburst),
                                              .arlock(intf.arlock),
                                              .arcache(intf.arcache),
                                              .arprot(intf.arprot),
                                              .arqos(intf.arqos),
                                              .arregion(intf.arregion),
                                              .aruser(intf.aruser),
                                              .arvalid(intf.arvalid),
```

Fig 3.2 AXI4 driver bfm instantiation in axi4 master agent bfm code snippet

```
axi4_master_monitor_bfm axi4_master_mon_bfm_h (.aclk(intf.aclk),
                                               .aresetn(intf.aresetn),
                                               .awid(intf.awid),
                                               .awaddr(intf.awaddr),
                                               .awlen(intf.awlen),
                                               .awsize(intf.awsize),
                                               .awburst(intf.awburst),
                                               .awlock(intf.awlock),
                                               .awcache(intf.awcache),
                                               .awprot(intf.awprot),
                                               .awvalid(intf.awvalid),
                                               .awready(intf.awready),
                                               .wdata(intf.wdata),
                                               .wstrb(intf.wstrb),
                                               .wlast(intf.wlast),
                                               .wuser(intf.wuser),
                                               .wvalid(intf.wvalid),
                                               .wready(intf.wready),
                                               .bid(intf.bid),
                                               .bresp(intf.bresp),
                                               .buser(intf.buser),
                                               .bvalid(intf.bvalid),
                                               .bready(intf.bready),
                                               .arid(intf.arid),
                                               .araddr(intf.araddr),
                                               .arlen(intf.arlen),
                                               .arsize(intf.arsize),
                                               .arburst(intf.arburst),
                                               .arlock(intf.arlock),
                                               .arcache(intf.arcache),
                                               .arprot(intf.arprot),
```

Fig 3.3 AXI4 monitor bfm instantiation in axi4 master agent bfm code snippet

3.2.5 AXI Master Driver BFM Interface

Axi4 master driver bfm is an interface where it will get the signals from the axi4 interface. It has a method drive_to_bfm which will be called by the axi4 master driver proxy which drives the awaddr and awdata to the axi4 interface. Fig. 3.4 gives the flowchart of the write address channel for master bfm. Fig. 3.5 gives the flowchart of the write data channel for master bfm. Fig. 3.6 gives the flowchart of the write response channel for master bfm. Fig. 3.7 gives the flowchart of the read address channel for master bfm. Fig. 3.8 gives the flowchart of the read data channel for master bfm.

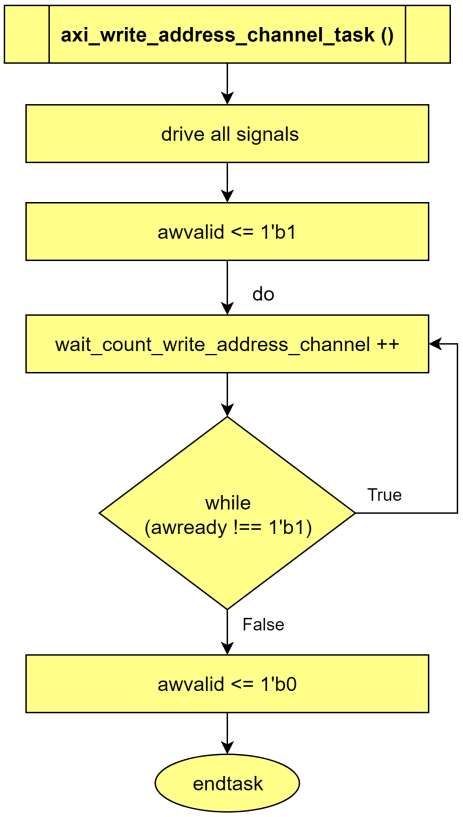


Fig 3.4 Flowchart of axi4_write_address_channel_task

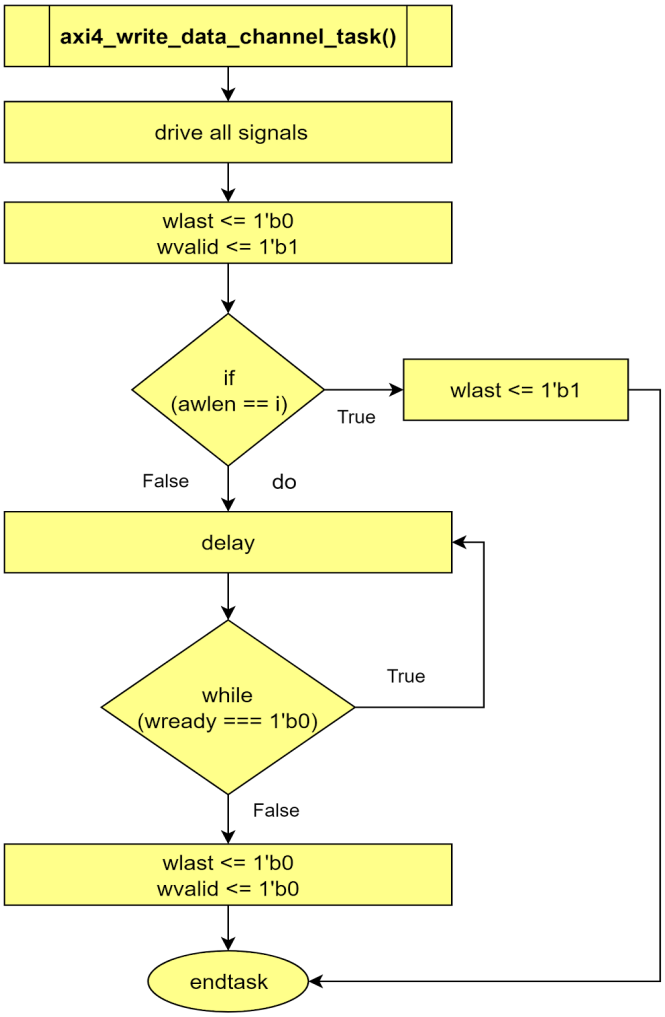


Fig 3.5 Flowchart of axi4_write_data_channel_task

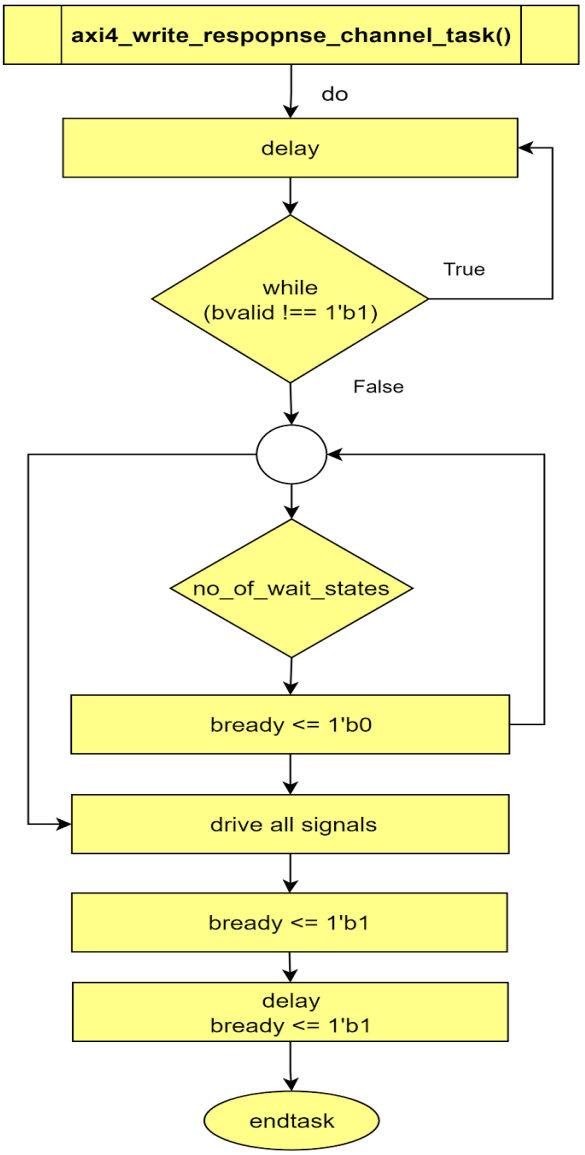


Fig 3.6 Flowchart of axi4_write_response_channel_task

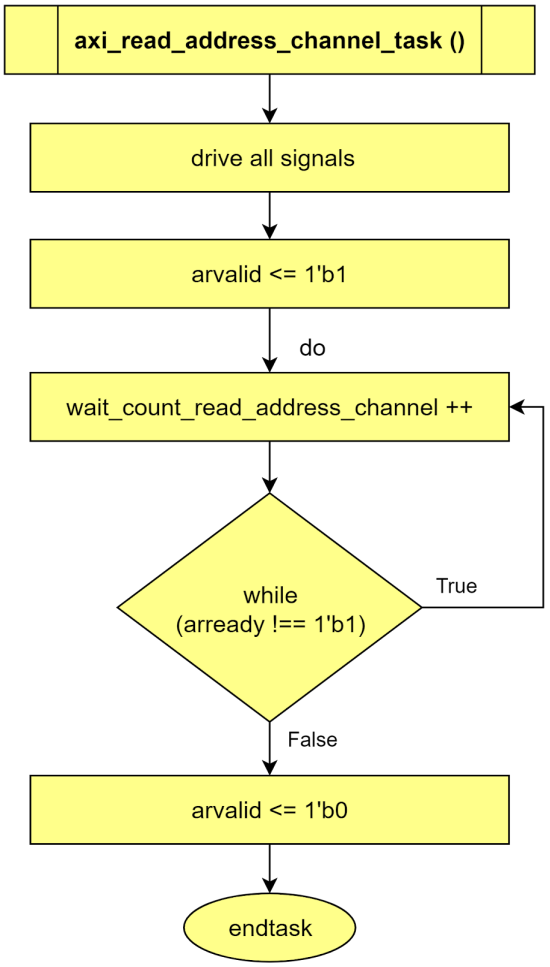


Fig 3.7 Flowchart of axi4_read_address_channel_task

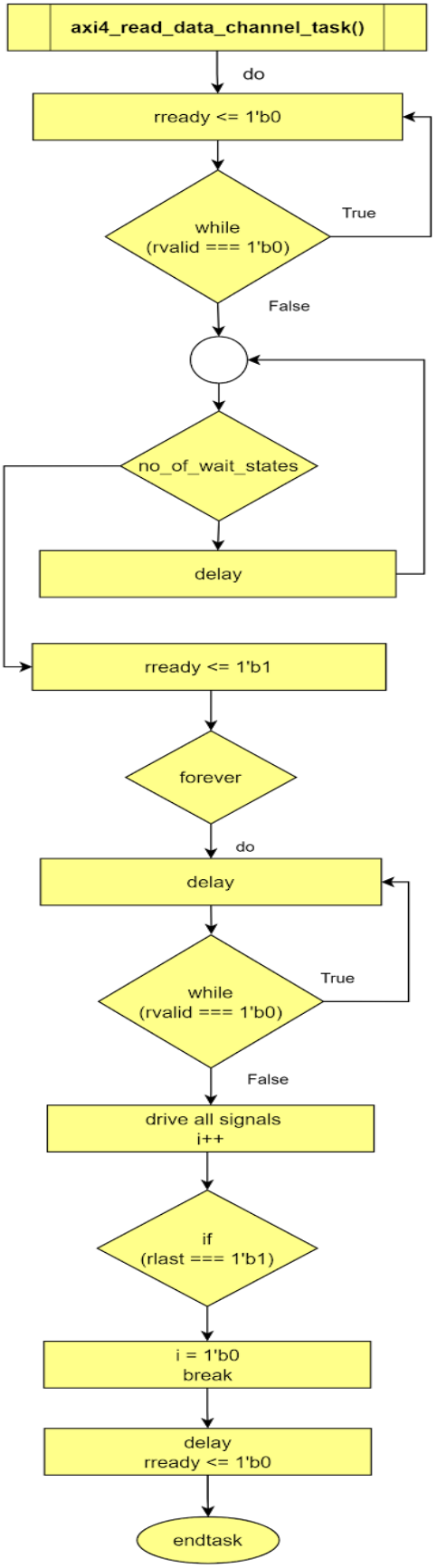


Fig 3.8 Flowchart of axi4_read_data_channel_task

3.2.6 AXI Master Monitor BFM Interface

Axi4 master monitor bfm is an interface where it will get the signals from the axi4 interface. It has a method sample_data which will be called by the axi4 master monitor proxy which samples the {paddr, pselx, pwdata and prdata} data from the axi4 interface. After sampling the data, the axi4 master monitor bfm interface sends the data to the axi4 master monitor proxy using the output port of sample_data task.

3.2.7 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 3.9 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 3.10 Spi slave monitor bfm instantiation in spi slave agent bfm code snippet

3.2.8 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.

3.2.9 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.

3.2.10 Pulpino SPI Master SUBSYSTEM Verification HVL_TOP

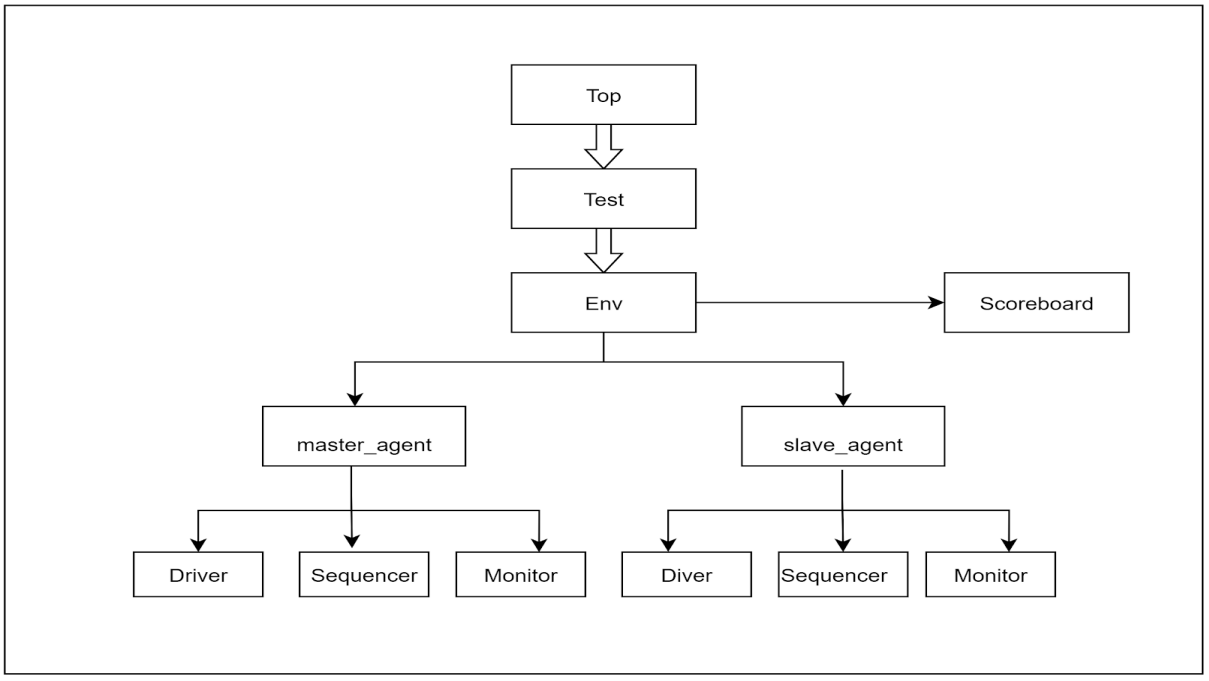


Fig 3.11 HVL Top

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

3.2.11 Pulpino SPI Master SUBSYSTEM Verification Environment

Environment has the below components

a. pulpino_spi_master_subsystem_verification_scoreboard
b. pulpino_spi_master_subsystem_verification_virtual_sequencer
c. axi4_master_agent

d. spi_slave_agent
e. axi4_master_collector
f. spi_slave_collector
g. axi4_reg_predictor

In the build phase, env_cfg handle will be called and create the memory for the above declared components.
In the connect phase, the axi4_master_monitor_proxy is connected to axi4_master_collector using uvm analysis port of axi4_master_monitor_proxy to uvm analysis import of axi4_master_collector. The axi4_master_collector is connected to the pulpino_spi_master_ip_verification_scoreboard using tlm_fifo of pulpino_spi_master_ip_verification_scoreboard and analysis_port of axi4 master collector. The spi_slave_monitor_proxy is connected to spi_salve_collector using uvm analysis port of spi_slave_monitor_proxy to uvm analysis import of spi_salve_collector. The axi4_master_collector is connected to the pulpino_spi_master_ip_verification_scoreboard using tlm_fifo of scoreboard and analysis_port of axi4 master collector. as shown in fig 3.7. The reg_map inside the env_config is passed to the axi4_master_collector and axi4_reg_predictor. The axi4_master_adapter is connected to the adapter in axi4_reg_predictor.

3.2.12 APB Register Predictor

APB register predictor component is used to find the register sampled from the monitor proxy to send to the coverage sampling using RAL. The APB reg predictor calls the bus2reg method to convert the apb bus leve transaction to register level transaction so that it can be passed to coverage sampling.

3.2.13 axi4 Master Collector

axi4 master collector component is used to sample the axi4 master monitor proxy sent data and access the registers using the address sent in the respective transaction and gets the data of the command_register, address_register, dummy_register and tx_fifo register. It concatenates all the four registers data in the respective order i,e., command, address ,dummy data ,tx_fifo data. This concatenated data is stored into the pulpino_spi_master_ip_global_pkg's coll_pkt struct packet. This coll_pkt struct packet is passed to the scoreboard for the comparisons. This process is shown in the below figure

3.2.14 SPI Slave Collector

axi4 master collector is a component which samples the data from the spi slave monitor proxy and sends the data to the scoreboard for the comparisons.

3.2.15 Pulpino SPI Master SUBSYSTEM Verification 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 PULPINO SPI MASTER SUBSYSTEM verification-AVIP project is to

1. Compare the data from the slave and master
2. The compared data is concatenation of command, address, dummy cycles and tx_fifo data.
3. Keep track of pass and failure rates identified in the comparison process
4. 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 collector class. fig. 3.7 shows the connection between the analysis port and analysis fifo.

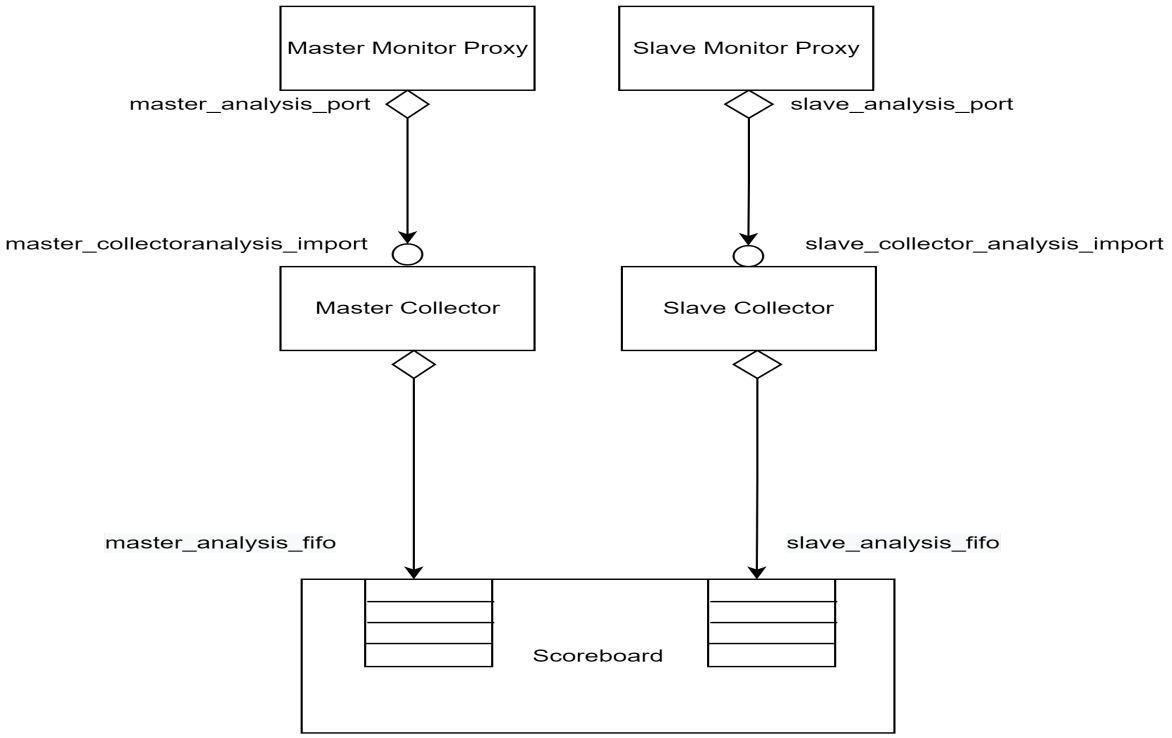


Fig 3.12 connection of the analysis ports of the monitor to the scoreboard analysis fifo

In the monitor proxy class of master and slave, five 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.

```
// Declaring analysis port for the monitor port
uvm_analysis_port#(axi4_master_tx) axi4_master_read_address_analysis_port;
uvm_analysis_port#(axi4_master_tx) axi4_master_read_data_analysis_port;
uvm_analysis_port#(axi4_master_tx) axi4_master_write_address_analysis_port;
uvm_analysis_port#(axi4_master_tx) axi4_master_write_data_analysis_port;
uvm_analysis_port#(axi4_master_tx) axi4_master_write_response_analysis_port;

//Declaring Monitor Analysis Import
uvm_analysis_port #(spi_slave_tx) spi_slave_analysis_port;
```

Fig 3.13 Declaration of slave and master analysis port in the slave and master monitor proxy

```
//variable : axi4_master_coll_analysis_port
//used to send the data from the axi4_master_collector
uvm_analysis_port#(collector_packet_s) axi4_master_coll_analysis_port;

//variable : axi4_master_coll_imp_port
//used to get the data from the axi4_master_monitor_proxy
uvm_analysis_imp#(axi4_master_tx, axi4_master_collector) axi4_master_coll_imp_port;
```

:Fig 3.14 Declaration of analysis port and import in the axi4 master collector

```
//variable : spi_slave_coll_analysis_port
//Used to send the data from the spi_slave_collector
uvm_analysis_port#(spi_slave_tx) spi_slave_coll_analysis_port;

//variable : spi_slave_coll_imp_port
//Used to get the data from the spi_slave_monitor_proxy
uvm_analysis_imp#(spi_slave_tx, spi_slave_collector) spi_slave_coll_imp_port;
```

Fig 3.15 Declaration of analysis port and import in the spi slave collector

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.

```
//Variable : axi4_master_write_address_analysis_fifo
//Used to store the axi4_master_data
uvm_tlm_analysis_fifo#(collector_packet_s) axi4_master_analysis_fifo;

  //Variable : spi_slave_analysis_fifo
//Used to store the spi_slave_data
uvm_tlm_analysis_fifo#(spi_slave_tx) spi_slave_analysis_fifo;
```

Fig 3.16 shows the declaration of master and slave analysis fifo in the scoreboard

s

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 scoreboard::new(string name = "scoreboard",uvm_component parent = null);
  super.new(name, parent);
  axi4_master_analysis_fifo  = new("axi4_master_analysis_fifo",this);
  spi_slave_analysis_fifo    = new("spi_slave_analysis_fifo",this);

endfunction : new
```

Fig 3.17 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.

```
axi4_master_agent_h.axi4_master_mon_proxy_h.axi4_master_write_address_analysis_port.connect
                                          (axi4_master_coll_h.axi4_master_coll_imp_port);
foreach(spi_slave_agent_h[i]) begin
  spi_slave_agent_h[i].spi_slave_mon_proxy_h.spi_slave_analysis_port.connect
                                          (spi_slave_coll_h.spi_slave_coll_imp_port);
end
```

Fig 3.18 Connection done between the analysis port and analysis fifo export in 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 scoreboard::run_phase(uvm_phase phase);
  super.run_phase(phase);

  forever begin
    bit [150:0]axi4_data;
    int        axi4_data_width;

    bit [150:0]spi_data;
    int        spi_data_width;

    `uvm_info(get_type_name(),$sformatf("before calling master's analysis fifo get method"),
                                        UVM_HIGH)
    axi4_master_analysis_fifo.get(axi4_data_packet);
    axi4_data = axi4_data_packet.data;
    axi4_data_width = axi4_data_packet.data_width;
    axi4_master_tx_count++;

    `uvm_info(get_type_name(),$sformatf("after calling master's analysis fifo get method"),
                                        UVM_HIGH)
    `uvm_info(get_type_name(),$sformatf("printing axi4_data = %0h",axi4_data),UVM_HIGH)
    `uvm_info(get_type_name(),$sformatf("before calling slave's analysis_fifo"),UVM_HIGH)
    spi_slave_analysis_fifo.get(spi_slave_tx_h);
    spi_slave_tx_count++;
    `uvm_info(get_type_name(),$sformatf("after calling slave's analysis fifo get method"),
                                        UVM_HIGH)
    `uvm_info(get_type_name(),$sformatf("printing spi_slave_tx_h, \n %s",spi_slave_tx_h.sprint()),
                                        UVM_HIGH)
```

Fig 3.19 Use of get method to get the packet from monitor analysis port

The Comparison of the awaddr, awdata and ardata from the master monitor and slave monitor is done in the run phase. Fig 3.13 shows the comparison of the master awdata with slave awdata.

```
  //Verifying awdata in master and slave
  if(axi4_data == spi_data) begin
    `uvm_info(get_type_name(),$sformatf("axi4_awdata from axi4_master and master_out_slave_in
                                        from spi_slave is equal"),UVM_HIGH);
    `uvm_info("SB_axi4_DATA_MATCHED WITH MOSI0", $sformatf("Master axi4_DATA = 'h%0x and Slave
                                            SPI_DATA = 'h%0x",axi4_data,spi_data), UVM_HIGH);

    byte_data_cmp_verified_master_awdata_slave_mosi_count++;
  end

  else begin
    `uvm_error(get_type_name(),$sformatf("axi4_awdata from axi4_master and master_out_slave_in
                                        from slave is not equal"));
    `uvm_error("SB_axi4_DATA_NOT_MATCHED WITH MOSI0", $sformatf("Master axi4_DATA = 'h%0x and
                                            Slave SPI_DATA = 'h%0x",axi4_data,spi_data));
    byte_data_cmp_failed_master_awdata_slave_mosi_count++;
  end
```

Fig 3.20 The comparison of the master axi4_data with slave spi_data

Similarly, the comparison is done for the signals as well.

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

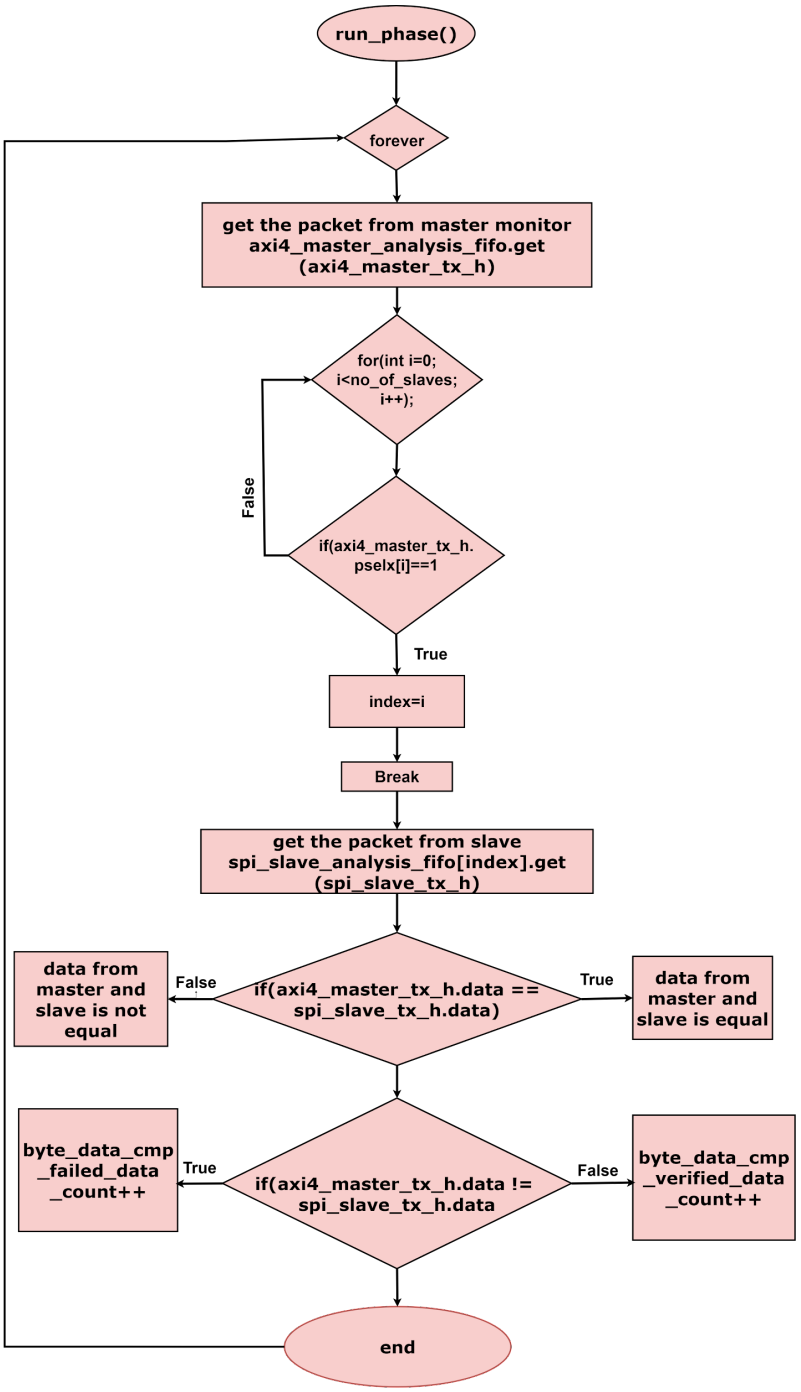


Fig 3.21 Flow chart of the scoreboard run phase

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.

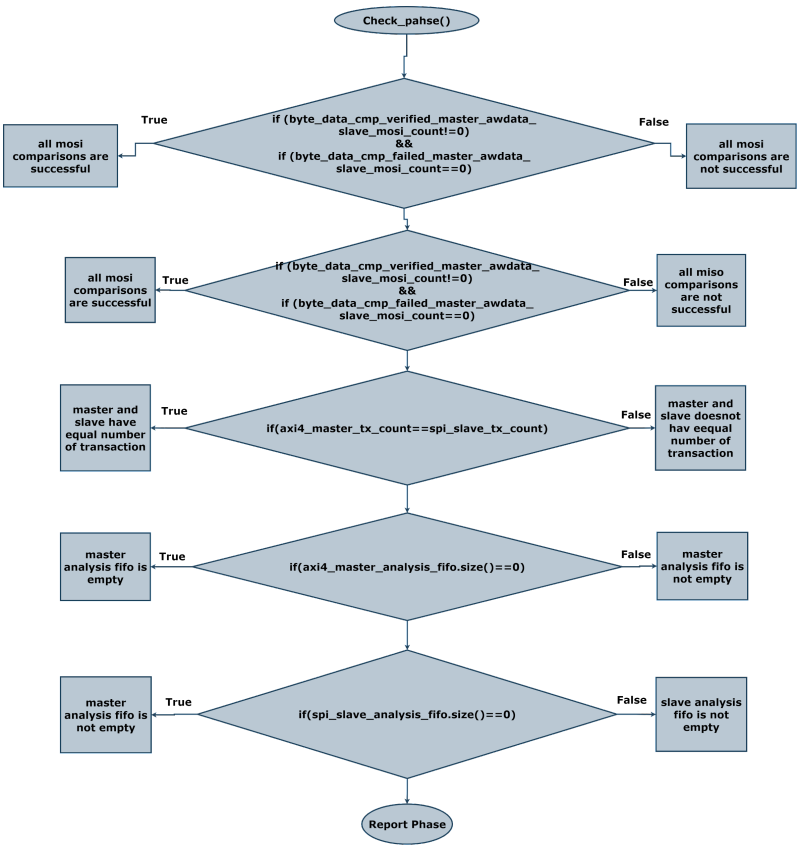


Fig 3.22 Flow chart of the scoreboard report phase

3.2.16 PULPINO SPI MASTER SUBSYSTEM Verification Virtual Sequencer

In virtual sequencer , declaring the handles for environment_configuration, master_sequencer and slave_sequencer.

3.2.17 AXI4 Master Agent

axi4 master agent component is a class extending from uvm_agent. It gets the axi4 master agent config handle and based on that we will create and connect the components. It creates the axi4 master sequencer and axi4 master driver only if the axi4 master agent is active which will depend on the value of is_active variable declared in the axi4 master agent configuration file. The axi4 master coverage is created in build_phase if the has_ coverage variable is 1 which is declared in the axi4 master agent configuration file. Please refer to figure 3.16 for the axi4 master agent build_phase code snippet.

axi4 master agent build phase has creation of,

a. axi4 master sequencer
b. axi4 master driver proxy
c. axi4 master monitor proxy
d. axi4 master coverage components.

```
function void axi4_master_agent::build_phase(uvm_phase phase);
  super.build_phase(phase);

 // if(!uvm_config_db #(axi4_master_agent_config)::get(this,"","axi4_master_agent_config",axi4_master_agent_cfg_h)) begin
 //    `uvm_fatal("FATAL_MA_CANNOT_GET_MASTER_AGENT_CONFIG","cannot get axi4_master_agent_cfg_h from uvm_config_db");
 // end

  if(axi4_master_agent_cfg_h.is_active == UVM_ACTIVE) begin
    axi4_master_drv_proxy_h=axi4_master_driver_proxy::type_id::create("axi4_master_drv_proxy_h",this);
    axi4_master_write_seqr_h=axi4_master_write_sequencer::type_id::create("axi4_master_write_seqr_h",this);
    axi4_master_read_seqr_h=axi4_master_read_sequencer::type_id::create("axi4_master_read_seqr_h",this);
  end

  axi4_master_mon_proxy_h=axi4_master_monitor_proxy::type_id::create("axi4_master_mon_proxy_h",this);

  if(axi4_master_agent_cfg_h.has_coverage) begin
   axi4_master_cov_h = axi4_master_coverage ::type_id::create("axi4_master_cov_h",this);
  end


endfunction : build_phase
```

Fig 3.23 axi4 master agent build phase code snippet

axi4 master agent configuration handles declared in the above created components will be mapped here in the connect phase. The axi4 master driver proxy and axi4 master sequencer are connected using TLM ports if the axi4 master agent is active. The axi4 master coverage's analysis_export will be connected to axi4 master monitor proxy's master_analysis_port in connect_phase.

```
function void axi4_master_agent::connect_phase(uvm_phase phase);
  super.connect_phase(phase);
  if(axi4_master_agent_cfg_h.is_active == UVM_ACTIVE) begin
    axi4_master_drv_proxy_h.axi4_master_agent_cfg_h = axi4_master_agent_cfg_h;
    axi4_master_write_seqr_h.axi4_master_agent_cfg_h = axi4_master_agent_cfg_h;
    axi4_master_read_seqr_h.axi4_master_agent_cfg_h = axi4_master_agent_cfg_h;
    axi4_master_cov_h.axi4_master_agent_cfg_h = axi4_master_agent_cfg_h;

    //Connecting the ports
    axi4_master_drv_proxy_h.axi_write_seq_item_port.connect(axi4_master_write_seqr_h.seq_item_export);
    axi4_master_drv_proxy_h.axi_read_seq_item_port.connect(axi4_master_read_seqr_h.seq_item_export);
  end

  if(axi4_master_agent_cfg_h.has_coverage) begin
    axi4_master_cov_h.axi4_master_agent_cfg_h = axi4_master_agent_cfg_h;
     // Connecting monitor_proxy port to coverage export
    axi4_master_mon_proxy_h.axi4_master_read_address_analysis_port.connect(axi4_master_cov_h.analysis_export);
    axi4_master_mon_proxy_h.axi4_master_read_data_analysis_port.connect(axi4_master_cov_h.analysis_export);
    axi4_master_mon_proxy_h.axi4_master_write_address_analysis_port.connect(axi4_master_cov_h.analysis_export);
    axi4_master_mon_proxy_h.axi4_master_write_data_analysis_port.connect(axi4_master_cov_h.analysis_export);
    axi4_master_mon_proxy_h.axi4_master_write_response_analysis_port.connect(axi4_master_cov_h.analysis_export);
  end

  axi4_master_mon_proxy_h.axi4_master_agent_cfg_h = axi4_master_agent_cfg_h;

endfunction : connect_phase
```

Fig 3.24 axi4 master agent connect phase code snippet

3.2.18 AXI4 Master Sequencer

axi4 master sequencer component is a parameterised class of type axi4 master transaction, extending uvm_sequencer. axi4 sequencer sends the data from the axi4 master sequences to the axi4 driver proxy.

3.2.19 AXI4 Master Driver Proxy

axi4 master driver proxy component is a parameterised class of type axi4 master transaction, extending uvm_driver. It gets the axi4 master agent config handle and based on the configurations we will drive and sample the paddr, pselx, pwdata and prdata signals respectively. It gets the master transaction into the axi4 driver proxy using get_next_item() method.

As the axi4 driver bfm interface cannot access the class based axi4 master transaction data, so we have to convert that into struct data type. Similarly, it converts the axi4 master configuration values into struct data type. axi4 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 axi4 global package) and then will pass it to axi4 master driver bfm using drive_to_bfm method declared in axi4 master driver bfm. The drive to bfm method starts the drive_to_bfm (data_packet, configuration packet) which is declared in axi4 driver bfm.

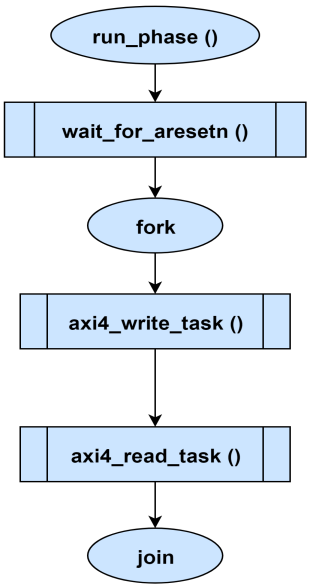


Fig 3.25 Flowchart for run phase of axi4 master driver proxy

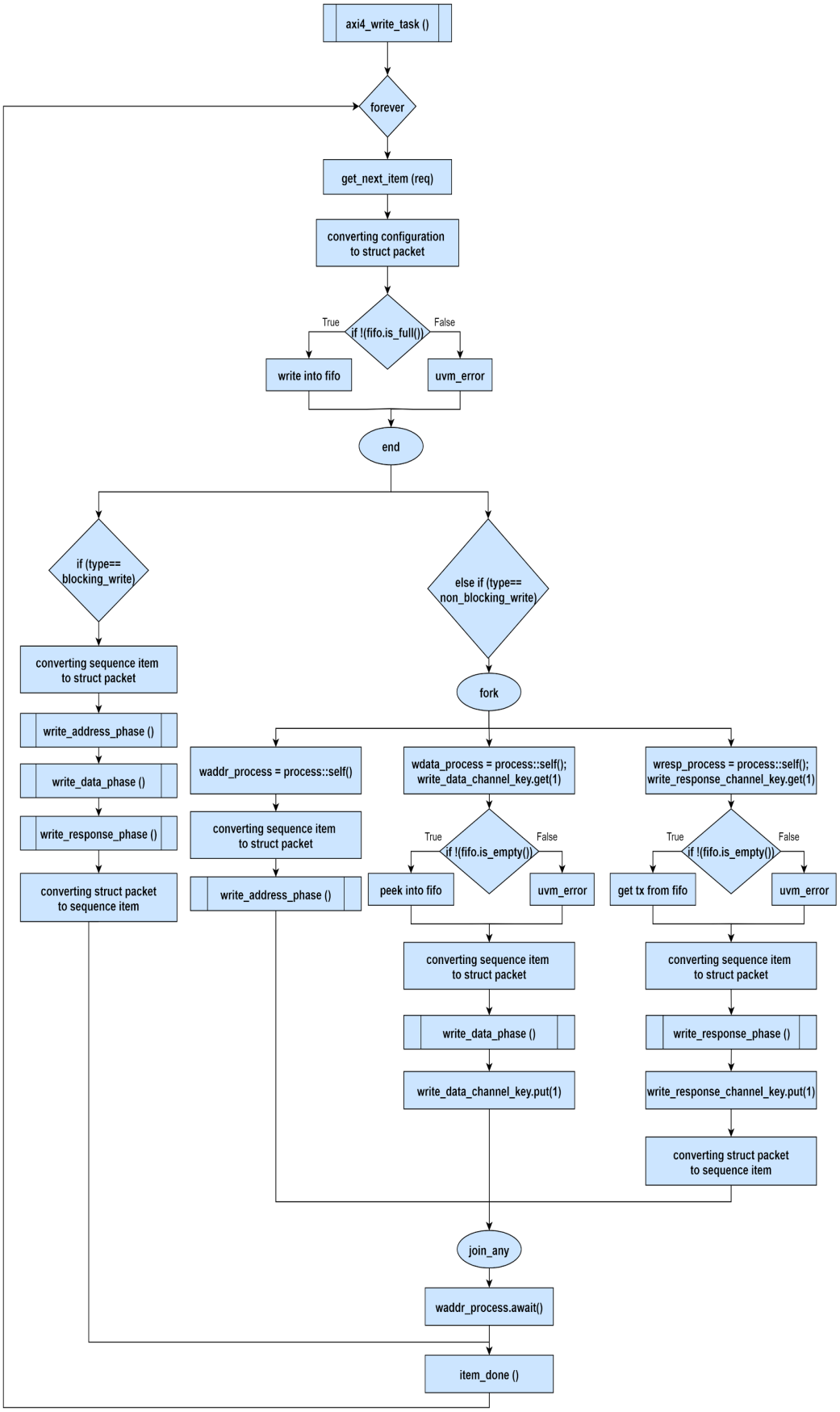


Fig 3.26 flowchart of write_task()

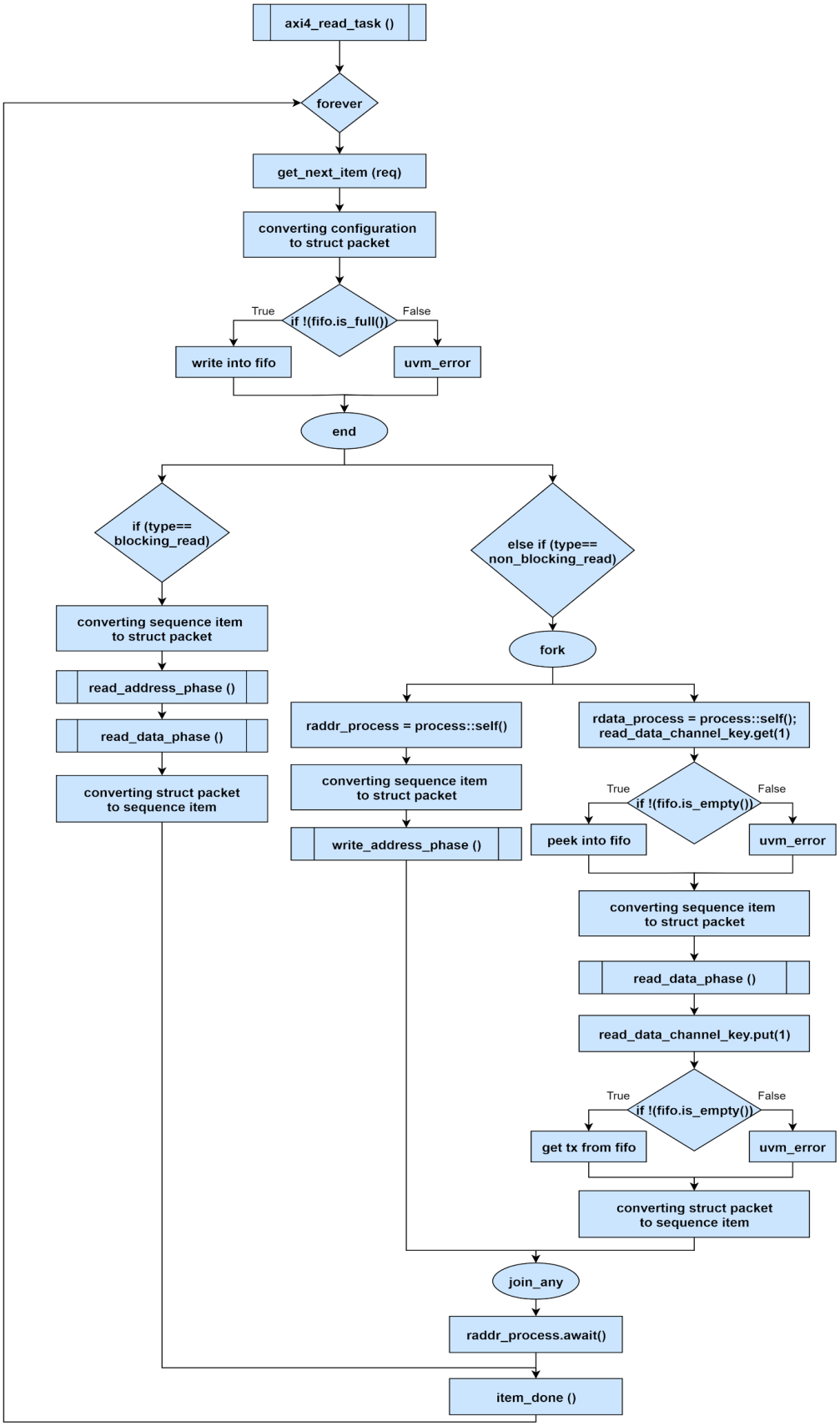


Fig 3.27 Flowchart of communication between axi4 master driver proxy and axi4 master driver bfm

```
task axi4_master_driver_proxy::run_phase(uvm_phase phase);

  //waiting for system reset
  axi4_master_drv_bfm_h.wait_for_aresetn();

  fork
    axi4_write_task();
    axi4_read_task();
  join

endtask : run_phase
```

Fig 3.28 run phase of axi4 master driver proxy code snippet

3.2.20 AXI4 Master Monitor Proxy

axi4 master monitor proxy component is a class extending uvm_monitor. It gets the axi4 master agent config handle and based on the configurations we will sample the pwdata and prdata signals. It declares and creates the axi4 master analysis port to send the sampled data.

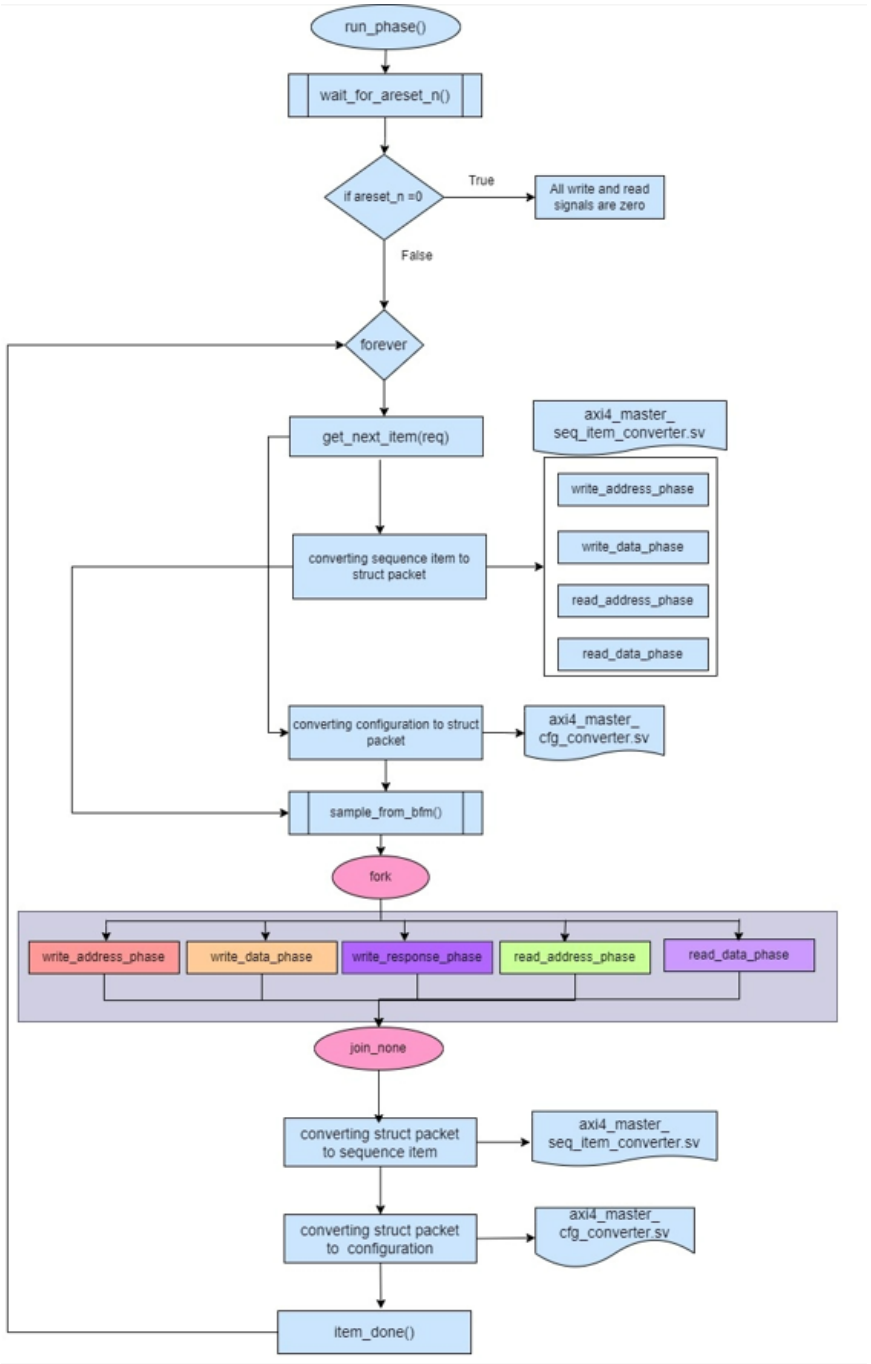


Fig 3.29 Flowchart of axi4 master monitor proxy and axi4 master monitor bfm communication

3.2.21 AXI4 Master Adapter

AXI4 master adapter is a class component that extends from uvm_reg_predictor. Its functionality is to convert the register_level_transactions to the apb bus_level_trasactions using bus2reg method. It is also used to convert the apb bus_level_transactions to the register_level_trasactions using reg2bus method.

3.2.22 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 3.30 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 3.31 Spi slave agent connect phase code snippet

3.2.23 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.

3.2.24 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.

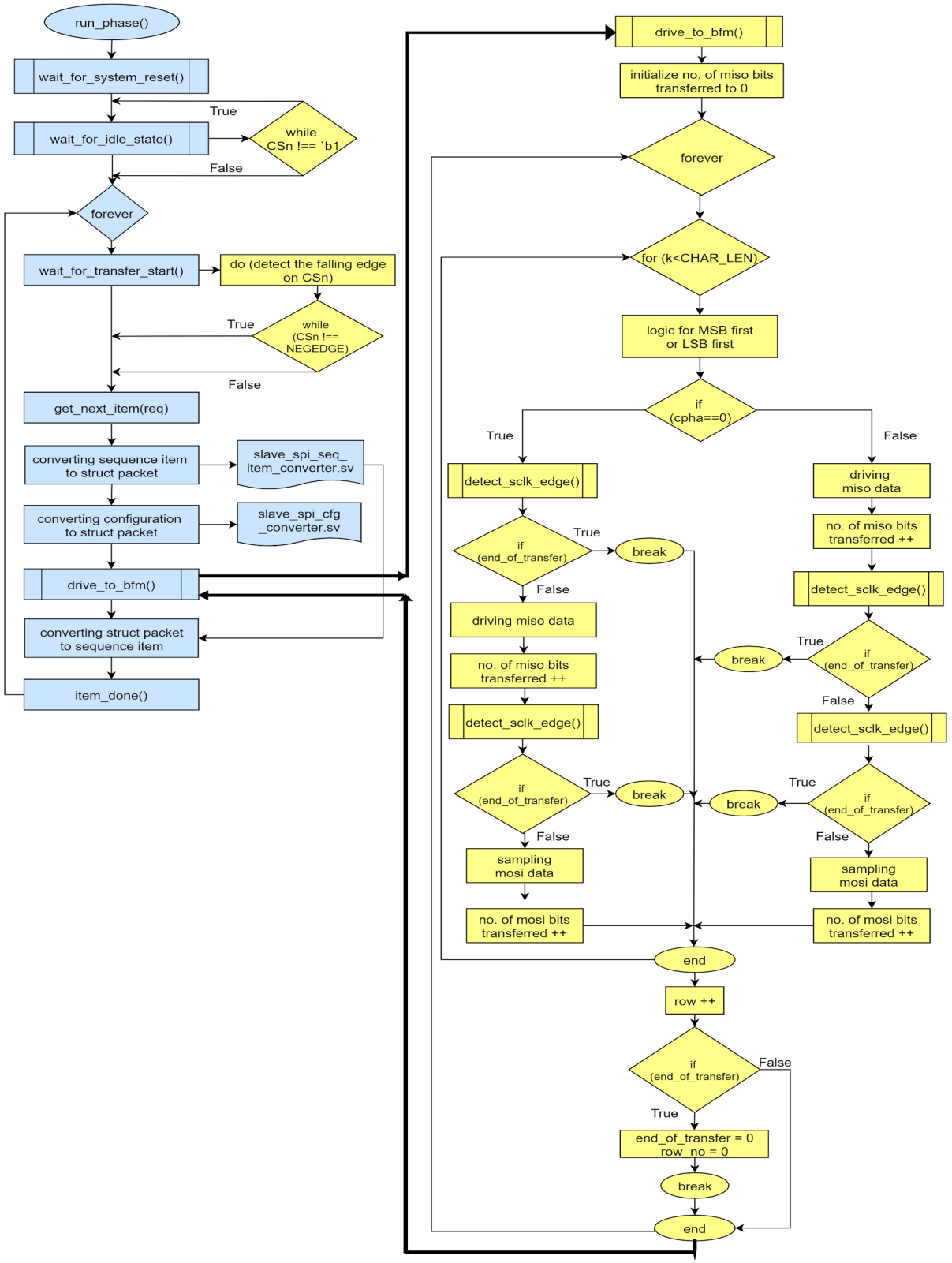


Fig 3.32 Flowchart of spi slave driver bfm and slave driver proxy communication

```
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 3.33 Spi slave driver proxy build phase code snippet

3.2.25 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.

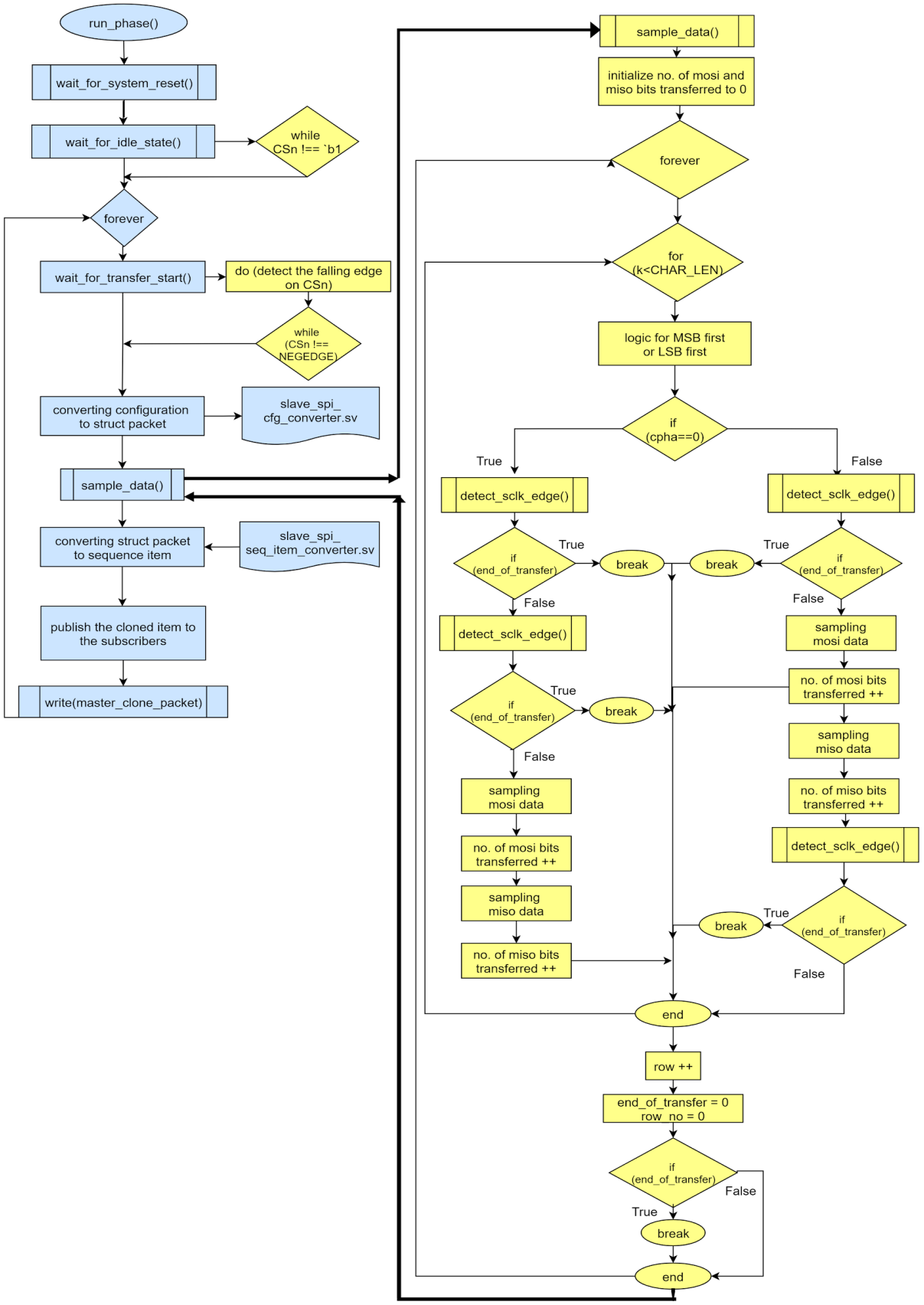


Fig 3.34 Flowchart of spi slave monitor bfm and slave monitor proxy communication

```
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 3.35 run phase of spi slave monitor proxy code snippet

3.2.26 SPI Slave Adapter

SPI slave adapter is a class component that extends from uvm_reg_predictor. Its functionality is to convert the register_level_transactions to the spi bus_level_trasactions using bus2reg method. It is also used to convert the spi bus_level_transactions to the register_level_trasactions using reg2bus method.

3.2.27 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.

Chapter 4

Directory Structure

4.1.Package Content

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

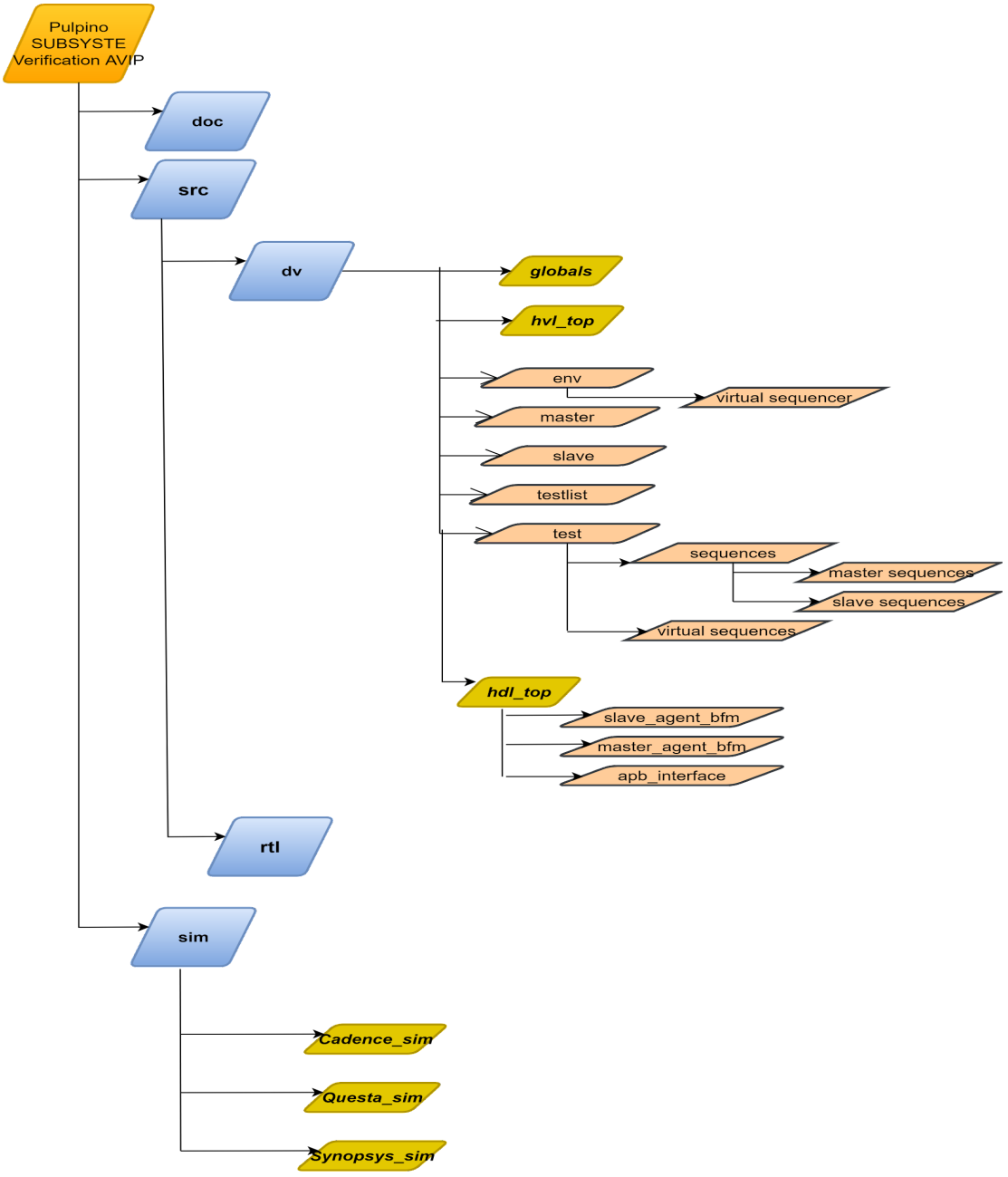


Figure 4.1. Package Structure of Pulpino_SPI_Master_Subsystem_AVIP

Table 4.1 Directory Path

Directory	Description
axi4_avip/doc	contains test bench architecture and components description and verification plan and assertion plan
axi4_avip/sim	Contains all simulating tools and axi4_compile.f file which contain all directories and compiling files
axi4_avip/src/dv/globals	Contains global package parameters(names,modes)
axi4_avip/src/dv/hvl_top	Contain all tb component folder(master,env,slave,test)
axi4_avip/src/dv/hdl_top	Contain all bfm files and assertions files
axi4_avip/src/dv/hdl_top/master_agent_bfm	Contain master agent, driver and monitor bfm files
axi4_avip/src/dv/hdl_top/slave_agent_bfm	Contains slave agent, driver and monitor bfm files
axi4_avip/src/dv/hdl_top/axi4_interface	Contain axi4 interface file
axi4_avip/src/dv/hvl_top/test	Contains all test cases files
axi4_avip/src/dv/hvl_top/test/sequences/master_sequences	Contain all master sequence test files
axi4_avip/src/dv/hvl_top/test/sequences/slave_sequences	Contain all slave sequence test files
axi4_avip/src/dv/hvl_top/test/sequences/virtual_sequences	Contain all virtual sequence test files
axi4_avip/src/dv/hvl_top/env	Contain env config files and score board file
axi4_avip/src/dv/hvl_top/env/virtual_sequencer	Contain virtual sequencer file
axi4_avip/src/dv/hvl_top/master	Contain master agent files , coverage file
axi4_avip/src/dv/hvl_top/slave	Contain slave agent files , coverage file

Chapter 5

Configuration

5.1 Global package variables

5.1.1 AXI4 Global Package Variables

Table 5.1 AXI4 Global package variables

Name	Type	Description
awburst_e	bit	Used to declare the enum type of write burst type WRITE_FIXED = 2'b00 WRITE_INCR = 2'b01 WRITE_WRAP = 2'b10 WRITE_RESERVED = 2'b11
arburst_e	bit	Used to declare the enum type of read burst type READ_FIXED = 2'b00 READ_INCR = 2'b01 READ_WRAP = 2'b10 READ_RESERVED = 2'b11
awsize_e	bit	Used to declare the enum type for write transfer size WRITE_1_BYTE = 3'b000 WRITE_2_BYTES = 3'b001 WRITE_4_BYTES = 3'b010 WRITE_8_BYTES = 3'b011 WRITE_16_BYTES = 3'b100 WRITE_32_BYTES = 3'b101 WRITE_64_BYTES = 3'b110 WRITE_128_BYTES = 3'b111
arsize_e	bit	Used to declare the enum type for read transfer size READ_1_BYTE = 3'b000 READ_2_BYTES = 3'b001 READ_4_BYTES = 3'b010 READ_8_BYTES = 3'b011 READ_16_BYTES = 3'b100 READ_32_BYTES = 3'b101 READ_64_BYTES = 3'b110 READ_128_BYTES = 3'b111
awlock_e	bit	Used to declare the enum type for write lock access WRITE_NORMAL_ACCESS = 1'b0 WRITE_EXCLUSIVE_ACCESS = 1'b1
arlock_e	bit	Used to declare the enum type for read lock access READ_NORMAL_ACCESS = 1'b0 READ_EXCLUSIVE_ACCESS = 1'b1
awcache_e	bit	Used to declare the enum type for write cache access WRITE_BUFFERABLE WRITE_MODIFIABLE WRITE_OTHER_ALLOCATE WRITE_ALLOCATE
arcache_e	bit	Used to declare the enum type for read cache access READ_BUFFERABLE READ_MODIFIABLE READ_OTHER_ALLOCATE READ_ALLOCATE

awprot_e	bit	Used to represent the protection type for transaction WRITE_NORMAL_SECURE_DATA = 3'b000 WRITE_NORMAL_SECURE_INSTRUCTION = 3'b001 WRITE_NORMAL_NONSECURE_DATA = 3'b010 WRITE_NORMAL_NONSECURE_INSTRUCTION = 3'b011 WRITE_PRIVILEGED_SECURE_DATA = 3'b100 WRITE_PRIVILEGED_SECURE_INSTRUCTION = 3'b101

arprot_e	bit	Used to represent the protection type for transaction READ_NORMAL_SECURE_DATA = 3'b000 READ_NORMAL_SECURE_INSTRUCTION = 3'b001 READ_NORMAL_NONSECURE_DATA = 3'b010 READ_NORMAL_NONSECURE_INSTRUCTION = 3'b011 READ_PRIVILEGED_SECURE_DATA = 3'b100 READ_PRIVILEGED_SECURE_INSTRUCTION = 3'b101
bresp_e	bit	Represents slave error signals for write channel WRITE_OKAY = 2'b00 WRITE_EXOKAY = 2'b01 WRITE_SLVERR = 2'b10 WRITE_DECERR = 2'b11
rresp_e	bit	Represents slave error signals for read channel READ_OKAY = 2'b00 READ_EXOKAY = 2'b01 READ_SLVERR = 2'b10 READ_DECERR = 2'b11
endian_e	bit	LITTLE_ENDIAN = 1'b0 : lsb bit will store in first address location BIG_ENDIAN = 1'b1 : msb bit will store in first address location
tx_type_e	bit	WRITE = 1'b1 : write transfer happens READ = 1'b0 : read transfer happens
transfer_type_e	bit	Used to determine the transfer type BLOCKING_WRITE = 2'b00 BLOCKING_READ = 2'b01 NON_BLOCKING_WRITE = 2'b10 NON_BLOCKING_READ = 2'b11

5.1.2 SPI Global Package Variables

Table 5.2 SPI 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.

5.1.3 Pulpino SPI Master SUBSYSTEM Global Package Variables

It has the struct packet to store the AXI4 Collector packet data.

Configuration used

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

5.2 Master agent configuration

Table 5.3 AXI4 Master_agent_config

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.
has_coverage	integer	‘d1	Used for enabling the master agent coverage

5.3 Slave agent configuration

Table 5.4 Env_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)

5.4 Environment configuration

Table 5.5 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 APB interface
spi_master_reg_block	spi_master_apb_if	null	Registers block handle for spi master module

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 Pulpino Specifications:

Pulpino subsystem Verification Specifications

6.2 Registers used in Pulpino:

1. Status Register
2. Clock Divider
3. SPI Command
4. SPI Address
5. SPI Length
6. Dummy Cycles
7. Tx FIFO
8. Rx FIFO
9. Interrupt Configurations

6.3 Features Supported:

1. Reset

1. Hard Reset
2. Soft Reset

2. Spi Modes for Writes and Reads

1. Standard Mode
2. Quad Mode

3. Clock Divider

1. Even CLKDIV (2)
2. Odd CLKDIV (1,3)

4. SPI Transfer Length

1. Zero_Command - Zero_Address - Non_Zero_Data(8x)
2. Zero_Command - Non_Zero_Address - Non_Zero_Data(8x)
3. Non_Zero_Command - Non_Zero_Address - Non_Zero_Data(8x)

5. SPI Dummy Cycles

5.1 For Writes

1. DUMMYWR_Zero_Clk_Cycles
2. DUMMYWR_Two_Clk_Cycles

5.2 For Reads

1. DUMMYRD_Zero_Clk_Cycles
2. DUMMYRD_Two_Clk_Cycles

6. Interrupt Configuration

1. Threshold Configurations
   1) THTX
   2) RHTX
2. Counter Configuration
   1) CNTTX

2) CNTRX
3. Enable
   1) EN=1
   2) EN=0

7. Tx & Rx FIFO

6.4 Template of Verification Plan

Verification Plan Link:

[Pulpino Subsystem Verification vplan]

In the below Figure

Section A shows the S.No

Section B shows the Sections

Section C shows the Features

Section D shows the Description

Section E & F shows the Test cases names

Section G shows the Status of the test cases

Sl.no	Sections	Features
2	**Clock Divider**	
2.1	For Writes and Reads	Even CLKDIV (2)
2.2		ODD CLKDIV (1,3)
3	**SPI Transfer Length**	
3.1	Zero_Command - Zero_Address - Non_Zero_Data(8x)	Zero_Zero_Eight
		Zero_Zero_Sixteen
		Zero_Zero_Thirtytwo
3.2	Zero_Command - Non_Zero_Address - Non_Zero_Data(8x)	Zero_Eight_Eight
		Zero_Sixteen_Sixteen
		Zero_Thirtytwo_Thirtytwo

Fig 6.1 features in verification plan

D
Description
Even value used to divide the SoC clock for the SPI transfers.This register should not be modified wh transfer is in progress.
Odd value used to divide the SoC clock for the SPI transfers.This register should not be modified whi transfer is in progress.
Used to transfer Zero command and Zero Address and Eight bits of Data
Used to transfer Zero command and Zero Address and Sixteen bits of Data
Used to transfer Zero command and Zero Address and bits of Thirtytwo Data
Used to transfer Zero command and Eight bits of Address and Eight bits of Data
Used to transfer Zero command and Sixteen bits of Address and Sixteen bits of Data
Used to transfer Zero command and Thirtytwo Address and bits of Thirtytwo Data

Fig 6.2 description of features

E	F	G
TestCase Names for Writes	**TestCase Names for Reads**	**Status**
pulpino_spi_master_ip_std_mode_write_even_clkdiv_test(reg_test)	pulpino_spi_master_ip_std_mode_read_even_clkdiv_test(reg_test)	TODO
pulpino_spi_master_ip_std_mode_write_odd_clkdiv_test(reg_test)	pulpino_spi_master_ip_std_mode_read_odd_clkdiv_test(reg_test)	TODO
pulpino_spi_master_ip_std_mode_write_0_cmd_0_addr_8_data_length_reg_test	pulpino_spi_master_ip_std_mode_read_0_cmd_0_addr_8_data_length_reg_test	TODO
pulpino_spi_master_ip_std_mode_write_0_cmd_0_addr_16_data_length_reg_test	pulpino_spi_master_ip_std_mode_read_0_cmd_0_addr_16_data_length_reg_test	TODO
pulpino_spi_master_ip_std_mode_write_0_cmd_0_addr_32_data_length_reg_test	pulpino_spi_master_ip_std_mode_read_0_cmd_0_addr_32_data_length_reag_test	TODO
pulpino_spi_master_ip_std_mode_write_0_cmd_8_addr_8_data_length_reg_test	pulpino_spi_master_ip_std_mode_read_0_cmd_8_addr_8_data_length_reg_test	TODO
pulpino_spi_master_ip_std_mode_write_0_cmd_16_addr_16_data_length_reg_test	pulpino_spi_master_ip_std_mode_read_0_cmd_16_addr_16_data_length_reg_test	TODO

Fig 6.3 test case names

6.5 Sections for different test scenarios

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

6.5.1 Directed test cases

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

Directed test names

Table no: 6.1

Sl.No	Feature	Testname	Description
1	SPI Modes	basic_write_test	Checks the mode of operation SPI is working
2	CLKDIV	std_mode_read_even_clkdiv_test	Checks the Clock divisor for the System clock
3	SPI Length	std_mode_write_8_cmd_16_addr_32_data_length_test	Checks the command length and address length and data length to transfer
4	SPI Dummy	std_mode_write_8_dummy_write_test	Adds the Dummy cycles for the writes after address and command sent
5	Interrupt Configuration	std_mode_write_thtx_rhtx_cnttx_cntrx_value_2_test	Adds the threshold value and counter value for transmitter and receiver and enables the interrupts.

6.5.2 Random test cases

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 names

This test describes the randomization of modes and clock divider and transfer lengths and dummy cycles

Table no: 6.1

Sl.No	Feature	Testname	Description
1	SPI modes, CLKDIV, SPI Length, Dummy Cycles, Tx fifo, and Interrupts	rand_reg_test	Randomizes the Spi modes and transfer lengths and clock divider value and dummy cycles for writes and reads and interrupt values for setting up threshold values for transmitter and Receiver

6.5.3 Cross test cases

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 names

This test describes the Cross test between the spi modes and CLKDIV and transfer length and dummy cycles

Table no: 6.1

Sl.No	Feature	Testname	Description
1	SPI modes, CLKDIV, Dummy cycles	spi_modes_clkdiv_dummy_cycles_cross_reg_test	Checks the cross coverage between spi modes and clock divider and dummy cycles
2	SPI Length, Interrupt configurations	spi_modes_transfer_length_interupts_cross_reg_test	Checks the cross coverage between spi transfer length and interrupts

6.5.4 Negative test cases

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 spi modes and transfer lengths

Table no: 6.1

Sl.No	Feature	Testname	Description
1	SPI Modes and Transfer length	spi_modes_transfer_length_negative_test	Checks the negative scenarios for the different spi modes and different spi transfer lengths

6.5.5 Register access test cases

This section defines the register access test cases which are used to access the registers of dut using Register Abstraction Layer(RAL). And you used to verify the configurations of registers and functionality of the dut.

Register access test names

The below tests shows how to write and read into the registers using RAL model

Sl.No	Feature	Testname	Description
1	SPI Modes	basic_write_read_reg_test	Checks the mode of operation SPI is working
2	CLKDIV	std_mode_read_even_clkdiv_reg_test	Checks the Clock divisor for the System clock
3	SPI Length	std_mode_write_8_cmd_16_addr_32_data_length_reg_test	Checks the command length and address length and data length to transfer
4	SPI Dummy	std_mode_write_16_dummy_write_reg_test	Adds the Dummy cycles for the writes after address and command sent
5	Interrupt Configuration	std_mode_write_thtx_rhtx_cnttx_cntrx_value_2_reg_test	Adds the threshold value and counter value for transmitter and receiver and enables the interrupts.

6.5.6 Stress test cases

This section describes about the stress test scenarios for the given dut where the design may broke

Stress test names

Sl.No	Feature	Testname	Description
1	Continuous transfers	pulpino_spi_master_ip_continous_transfer_stress_test	Check for the back to back spi transfers between SPI master and slave

For more information about Verification plan refer below link

Chapter 7

Coverage

7.1 Template of Coverage Plan

Template for Coverage plan is done in an excel sheet and refer to link below:

7.2 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.

7.3 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
```

Fig 7.1. Uvm_subscriber

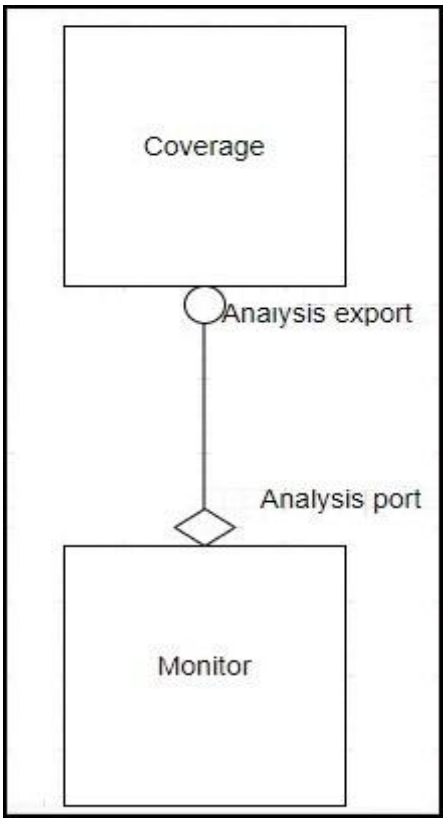


Fig 7.2. Monitor and coverage connection

7.3.1 Analysis export

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

7.3.2 Write function

The write function is to process the incoming transactions.

```
function void axi4_master_coverage::write(axi4_master_tx t);
 `uvm_info(get_type_name(),$sformatf("Before calling SAMPLE METHOD"),UVM_HIGH);

  axi4_master_covergroup.sample(axi4_master_agent_cfg_h,t);

  `uvm_info(get_type_name(),"After calling SAMPLE METHOD",UVM_HIGH);
endfunction: write
```

Fig 7.3: Write function

7.4 Covergroup

```
covergroup axi4_master_covergroup with function sample (axi4_master_agent_config cfg,
                                                        axi4_master_tx packet);
  option.per_instance = 1;

  //-------------------------------------------------------
  // Write channel signals
  //-------------------------------------------------------

  AWLEN_CP : coverpoint packet.awlen {
    option.comment = "Write Address Length values";
    bins AWLEN_1       = {0};
    bins AWLEN_2       = {1};
    bins AWLEN_4       = {3};
    bins AWLEN_8       = {7};
    bins AWLEN_16      = {15};
    bins AWLEN_32      = {31};
    bins AWLEN_64      = {63};
    bins AWLEN_128     = {127};
    bins AWLEN_256     = {255};
    bins AWLEN_DEFAULT = default ;
  }
```

Fig 7.4. Covergroup

The above red mark points in Figure covergroup 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 axi4_master_covergroup with function sample (axi4_master_agent_config cfg,
                                                        axi4_master_tx packet);
  option.per_instance = 1;
```

Figure 7.5 option.per_instance

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

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

Coverpoint: AWID_CP

Search:

Bin Name	At Least	Hits
AWID[AWID_0]	1	624
AWID[AWID_1]	1	0
AWID[AWID_2]	1	0
AWID[AWID_3]	1	0
AWID[AWID_4]	1	0
AWID[AWID_5]	1	0
AWID[AWID_6]	1	0
AWID[AWID_7]	1	0
AWID[AWID_8]	1	0
AWID[AWID_9]	1	0
AWID[AWID_10]	1	0
AWID[AWID_11]	1	0
AWID[AWID_12]	1	0
AWID[AWID_13]	1	0
AWID[AWID_14]	1	0
AWID[AWID_15]	1	0

Fig 7.6 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.

7.4 Bucket

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

```
AWID_CP : coverpoint packet.awid {
  option.comment = "Write Address ID values";
  bins AWID[] = {[0:$]};
}
```

Fig 7.7 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

7.5 Coverpoints

There we created the bins based on the write and read operation.

```
AWID_CP : coverpoint packet.awid {
  option.comment = "Write Address ID values";
  bins AWID[] = {[0:$]};
}
```

Fig 7.8 Coverpoint

7.6 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.

Cross coverage between paddr, prdata and pwdata:

```
AWLENGTH_CP_X_AWSIZE_X_AWBURST    :cross AWLEN_CP,AWSIZE_CP,AWBURST_CP;
ARLENGTH_CP_X_ARSIZE_X_ARBURST    :cross ARLEN_CP,ARSIZE_CP,ARBURST_CP;
```

Fig 7.9 Cross Coverpoints

7.6.1 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.

7.7 Creation of the covergroup

```
function axi4_master_coverage::new(string name = "axi4_master_coverage",uvm_component parent = null);
  super.new(name, parent);
  axi4_master_covergroup =new();
endfunction : new
```

Fig 7.10 Creation of covergroup

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

7.8 Sampling of the covergroup

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

```
function void axi4_master_coverage::write(axi4_master_tx t);
 `uvm_info(get_type_name(),$sformatf("Before calling SAMPLE METHOD"),UVM_HIGH);

  axi4_master_covergroup.sample(axi4_master_agent_cfg_h,t);

  `uvm_info(get_type_name(),"After calling SAMPLE METHOD",UVM_HIGH);
endfunction: write
```

Fig 7.11 Sampling of the covergroup

7.9 Checking for the coverage

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

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

Fig 7.12 Simulation log file path

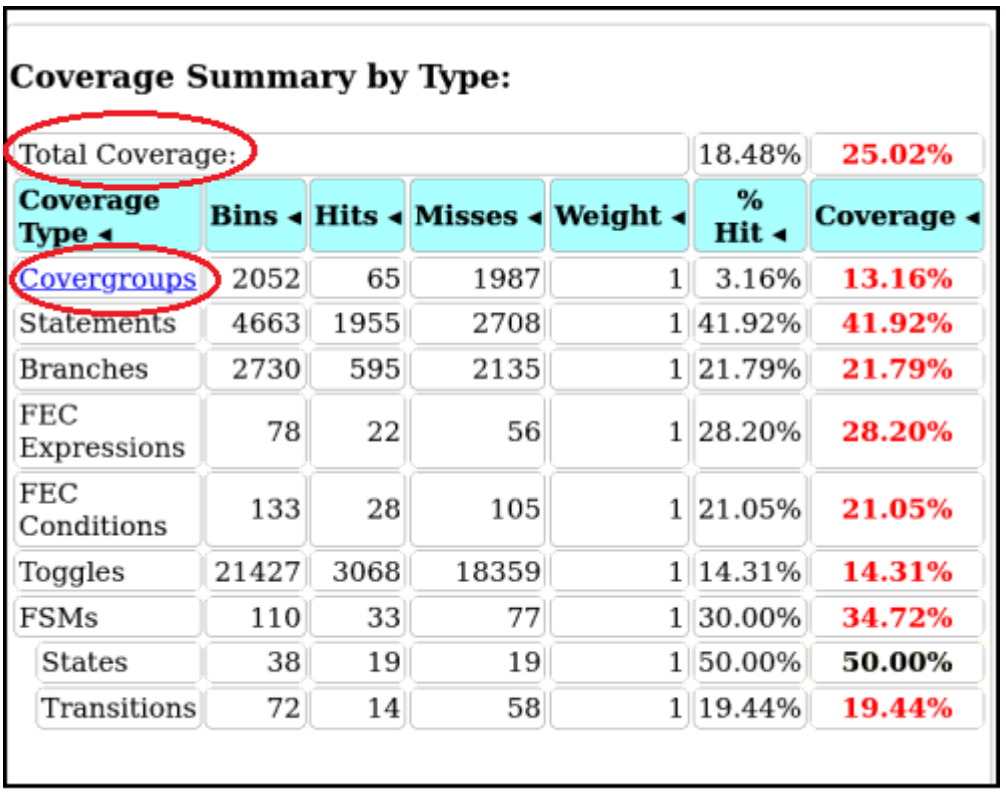
4. Search for the coverage (There it will be the full coverage) in the log file.

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

```
Coverage report: firefox basic_write_read_reg_test/html_cov_report/index.html &
```

Fig 7.13 Coverage report path

Then new html window will open



Coverage Summary by Type:

Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Total Coverage:					18.48%	25.02%
Covergroups	2052	65	1987	1	3.16%	13.16%
Statements	4663	1955	2708	1	41.92%	41.92%
Branches	2730	595	2135	1	21.79%	21.79%
FEC Expressions	78	22	56	1	28.20%	28.20%
FEC Conditions	133	28	105	1	21.05%	21.05%
Toggles	21427	3068	18359	1	14.31%	14.31%
FSMs	110	33	77	1	30.00%	34.72%
States	38	19	19	1	50.00%	50.00%
Transitions	72	14	58	1	19.44%	19.44%

Fig 7.14 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.

Search:

CoverPoints	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
ARBURST_CP	3	1	2	33.33%	33.33%	33.33%
ARCACHE_CP	4	1	3	25.00%	25.00%	25.00%
ARID_CP	16	1	15	6.25%	6.25%	6.25%
ARLEN_CP	9	1	8	11.11%	11.11%	11.11%
ARLOCK_CP	2	1	1	50.00%	50.00%	50.00%
ARPROT_CP	8	1	7	12.50%	12.50%	12.50%
ARSIZE_CP	8	1	7	12.50%	12.50%	12.50%
AWBURST_CP	3	1	2	33.33%	33.33%	33.33%
AWCACHE_CP	4	1	3	25.00%	25.00%	25.00%
AWID_CP	16	1	15	6.25%	6.25%	6.25%
AWLEN_CP	9	1	8	11.11%	11.11%	11.11%

Fig 7.15 All coverpoints present in the Covergroup

Coverpoint: AWID_CP

Comment:
Write Address ID values

Search:

Bin Name	At Least	Hits
AWID[AWID_0]	1	24
AWID[AWID_1]	1	0
AWID[AWID_2]	1	0
AWID[AWID_3]	1	0
AWID[AWID_4]	1	0
AWID[AWID_5]	1	0
AWID[AWID_6]	1	0
AWID[AWID_7]	1	0
AWID[AWID_8]	1	0
AWID[AWID_9]	1	0
AWID[AWID_10]	1	0

Figure 7.16 Individual Coverpoint Hit

Excluding the coverage

Chapter 8

Test Cases

8.1 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

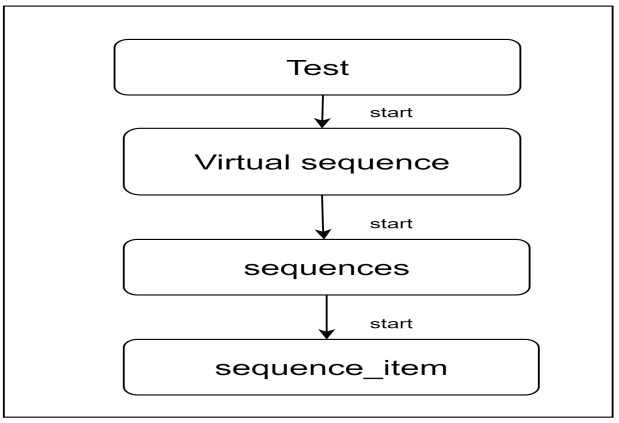


Fig 8.1 Test flow

8.2 Test Cases Flowchart

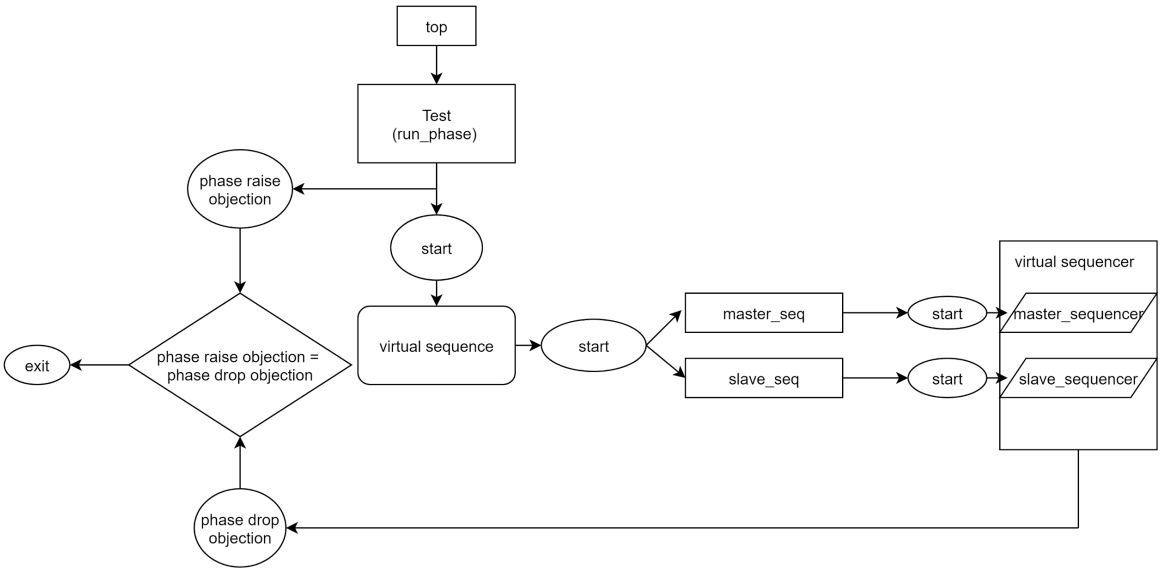


Fig 8.2 test cases flow chart

8.3 Transaction

8.3.1 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(of all the 5 channels, like write address, write data, write response, read address, read data)

  Write address channel
- Constraint declared for awburst to get burst type between fixed, incr and wrap types.
- Constraint declared for awlen to restrict the transfer size.
- Constraint declared for awlen to get multiples of 2 in wrap type burst
- Constraint declared for awlock to get locked transfers

➢ Constraint declared for awburst to select the type of burst

➢ Constraint declared for awsize to get the size of transfer

```
constraint awburst_c1 { awburst != WRITE_RESERVED;
                      }
constraint awlength_c2 { if(awburst==WRITE_FIXED || WRITE_WRAP)
                           awlen inside {[0:15]};
                         else if(awburst == WRITE_INCR)
                           awlen inside {[0:255]};
                       }

constraint awlength_c3 { if(awburst == WRITE_WRAP)
                           awlen + 1 inside {2,4,8,16};
                       }

constraint awlock_c4 { soft awlock == WRITE_NORMAL_ACCESS;
                     }

constraint awburst_c5 { soft awburst == WRITE_INCR;
                      }

constraint awsize_c6 { soft awsize inside {[0:2]};
                      }
```

Fig 8.3 Constraint for write address

Write data channel

➢ Constraint declared for wdata to restrict data based on awlen

➢ Constraint declared for no of wait states to restrict the wait states for response

```
constraint wdata_c1 { wdata.size() == awlen + 1;}

constraint no_of_wait_states_c3 { no_of_wait_states inside  {[0:3]};}
```

Fig 8.4 Constraint for write data

Read address channels

- ➢ Constraint declared for arburst for restricting burst type between incr, wrap and fixed
- ➢ Constraint declared for arlen to restrict the transfer size.
- ➢ Constraint declared for arlen to get multiples of 2 in wrap type burst
- ➢ Constraint declared for arlock to get locked transfers
- ➢ Constraint declared for arburst to select the type of burst
- ➢ Constraint declared for arsize to get the size of transfer

```
constraint arburst_c1 { arburst != READ_RESERVED;
                      }
constraint arlength_c2 { if(arburst==READ_FIXED || READ_WRAP)
                          arlen inside {[0:15]};
                         else if(arburst == READ_INCR)
                          arlen inside {[0:255]};
                       }
constraint arlength_c3 { if(arburst == READ_WRAP)
                          arlen + 1 inside {2,4,8,16};
                       }
constraint arlock_c4 { soft arlock == READ_NORMAL_ACCESS;
                     }
constraint arburst_c5 { soft arburst == READ_INCR;
```

Fig 8.5 Constraint for read address

Memory constraint

- ➢ Adding constraint for selecting the endianness

```
constraint endian_c1 { soft endian == LITTLE_ENDIAN;
                     }
```

Fig 8.6 Constraint for memory

Table 9.1 Describing constraint in master and slave transactions

Constraint	Description
awburst_c1	Constraint declared for awburst to get burst type between fixed, incr and wrap types.
awlength_c2	Constraint declared for awlen to restrict the transfer size.
awlength_c3	Constraint declared for awlen to get multiples of 2 in wrap type burst
awlock_c4	Constraint declared for awlock to get locked transfers
awburst_c5	Constraint declared for awburst to select the type of burst
awsize_c6	Constraint declared for awsize to get the size of transfer
wdata_c1	Constraint declared for wdata to restrict data based on awlen
no_of_wait_states_c3	Constraint declared for no of wait states to restrict the wait states for response
arburst_c1	Constraint declared for arburst for restricting burst type between incr, wrap and fixed
arlength_c2	Constraint declared for arlen to restrict the transfer size.
arlength_c3	Constraint declared for arlen to get multiples of 2 in wrap type burst
arlock_c4	Constraint declared for arlock to get locked transfers
arburst_c5	Constraint declared for arburst to select the type of burst
endian_c1	Adding constraint for selecting the endianness
rdata_c1	Constraint declared for ardata to restrict the data based on the arlen

bresp_c1	Constraint declared for the bresp to select the type of write response
rresp_c1	Constraint declared for the rresp to select the type of write response
wait_state_c1	Constraint to randomise the wait state in between 0 to 3

Master tx do_copy,do_compare and do_print methods

- 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 data values.

```
function void axi4_master_tx::do_copy(uvm_object rhs);
  axi4_master_tx axi4_master_tx_copy_obj;

  if(!$cast(axi4_master_tx_copy_obj,rhs)) begin
    `uvm_fatal("do_copy","cast of the rhs object failed")
  end
  super.do_copy(rhs);

  //WRITE ADDRESS CHANNEL
  awid    = axi4_master_tx_copy_obj.awid;
  awaddr  = axi4_master_tx_copy_obj.awaddr;
  awlen   = axi4_master_tx_copy_obj.awlen;
  awsize  = axi4_master_tx_copy_obj.awsize;
  awburst = axi4_master_tx_copy_obj.awburst;
  awlock  = axi4_master_tx_copy_obj.awlock;
  awcache = axi4_master_tx_copy_obj.awcache;
  awprot  = axi4_master_tx_copy_obj.awprot;
  awqos   = axi4_master_tx_copy_obj.awqos;
```

```
//WRITE DATA CHANNEL
wdata = axi4_master_tx_copy_obj.wdata;
wstrb = axi4_master_tx_copy_obj.wstrb;

//WRITE RESPONSE CHANNEL
bid   = axi4_master_tx_copy_obj.bid;
bresp = axi4_master_tx_copy_obj.bresp;

//READ ADDRESS CHANNEL
arid    = axi4_master_tx_copy_obj.arid;
araddr  = axi4_master_tx_copy_obj.araddr;
arlen   = axi4_master_tx_copy_obj.arlen;
arsize  = axi4_master_tx_copy_obj.arsize;
arburst = axi4_master_tx_copy_obj.arburst;
arlock  = axi4_master_tx_copy_obj.arlock;
arcache = axi4_master_tx_copy_obj.arcache;
arprot  = axi4_master_tx_copy_obj.arprot;
arqos   = axi4_master_tx_copy_obj.arqos;

//READ DATA CHANNEL
rid    = axi4_master_tx_copy_obj.rid;
rdata = axi4_master_tx_copy_obj.rdata;
rresp = axi4_master_tx_copy_obj.rresp;
```

Fig 8.7 do_copy method

```
function bit axi4_master_tx::do_compare (uvm_object rhs, uvm_comparer comparer);
  axi4_master_tx axi4_master_tx_compare_obj;

  if(!$cast(axi4_master_tx_compare_obj,rhs)) begin
    `uvm_fatal("FATAL_axi_MASTER_TX_DO_COMPARE_FAILED","cast of the rhs object failed")
    return 0;
  end

  return super.do_compare(axi4_master_tx_compare_obj, comparer) &&
  //WRITE ADDRESS CHANNEL
  awid    == axi4_master_tx_compare_obj.awid    &&
  awaddr  == axi4_master_tx_compare_obj.awaddr  &&
  awlen   == axi4_master_tx_compare_obj.awlen   &&
  awsize  == axi4_master_tx_compare_obj.awsize  &&
  awburst == axi4_master_tx_compare_obj.awburst &&
  awlock  == axi4_master_tx_compare_obj.awlock  &&
  awcache == axi4_master_tx_compare_obj.awcache &&
  awprot  == axi4_master_tx_compare_obj.awprot  &&
  awqos   == axi4_master_tx_compare_obj.awqos   &&

  //WRITE DATA CHANNEL
```

```
  //WRITE DATA CHANNEL
  wdata == axi4_master_tx_compare_obj.wdata &&
  wstrb == axi4_master_tx_compare_obj.wstrb &&

  //WRITE RESPONSE CHANNEL
  bid   == axi4_master_tx_compare_obj.bid   &&
  bresp == axi4_master_tx_compare_obj.bresp &&

  //READ ADDRESS CHANNEL
  arid    == axi4_master_tx_compare_obj.arid    &&
  araddr  == axi4_master_tx_compare_obj.araddr  &&
  arlen   == axi4_master_tx_compare_obj.arlen   &&
  arsize  == axi4_master_tx_compare_obj.arsize  &&
  arburst == axi4_master_tx_compare_obj.arburst &&
  arlock  == axi4_master_tx_compare_obj.arlock  &&
  arcache == axi4_master_tx_compare_obj.arcache &&
  arprot  == axi4_master_tx_compare_obj.arprot  &&
  arqos   == axi4_master_tx_compare_obj.arqos   &&

  //READ DATA CHANNEL
  rid   == axi4_master_tx_compare_obj.rid   &&
  rdata == axi4_master_tx_compare_obj.rdata &&
  rresp == axi4_master_tx_compare_obj.rresp;
endfunction : do_compare
```

Fig 8.8 do_compare method

```
function void axi4_master_tx::do_print(uvm_printer printer);
  //super.do_print(printer);
  //`uvm_info("------------------------------------------WRITE_ADDRESS_CHANNEL","------------------
  printer.print_string("tx_type",tx_type.name());
  if(tx_type == WRITE) begin
    printer.print_string("awid",awid.name());
    printer.print_field("awaddr",awaddr,$bits(awaddr),UVM_HEX);
    printer.print_field("awlen",awlen,$bits(awlen),UVM_DEC);
    printer.print_string("awsize",awsize.name());
    printer.print_string("awburst",awburst.name());
    printer.print_string("awlock",awlock.name());
    printer.print_string("awcache",awcache.name());
    printer.print_string("awprot",awprot.name());
    printer.print_field("awqos",awqos,$bits(awqos),UVM_HEX);
    //`uvm_info("-------------------------------------------WRITE_DATA_CHANNEL","------------------

    foreach(wdata[i])begin
      printer.print_field($sformatf("wdata[%0d]",i),wdata[i],$bits(wdata[i]),UVM_HEX);
    end
    foreach(wstrb[i])begin
      // MSHA: printer.print_field($sformatf("wstrb[%0d]",i),wstrb[i],$bits(wstrb[i]),UVM_HEX);
      printer.print_field($sformatf("wstrb[%0d]",i),wstrb[i],$bits(wstrb[i]),UVM_DEC);
    end
    //`uvm_info("------------------------------------------WRITE_RESPONSE_CHANNEL","---------------
    printer.print_field("no_of_wait_states",no_of_wait_states,$bits(no_of_wait_states),UVM_DEC);
    printer.print_string("bid",bid.name());
    printer.print_string("bresp",bresp.name());
  end
```

```
  if(tx_type == READ) begin
    //`uvm_info("----------------------------------------READ_ADDRESS_CHANNEL","----------
    printer.print_string("arid",arid.name());
    printer.print_field("araddr",araddr,$bits(araddr),UVM_HEX);
    printer.print_field("arlen",arlen,$bits(arlen),UVM_DEC);
    printer.print_string("arsize",arsize.name());
    printer.print_string("arburst",arburst.name());
    printer.print_string("arlock",arlock.name());
    printer.print_string("arcache",arcache.name());
    printer.print_string("arprot",arprot.name());
    printer.print_field("arqos",arqos,$bits(arqos),UVM_HEX);
    //`uvm_info("----------------------------------------READ_DATA_CHANNEL","----------
    foreach(rdata[i])begin
      printer.print_field($sformatf("rdata[%0d]",i),rdata[i],$bits(rdata[i]),UVM_HEX);
    end
    //printer.print_field("rdata",rdata,$bits(rdata),UVM_HEX);
    printer.print_string("rid",rid.name());
    printer.print_string("rresp",rresp.name());
    printer.print_field("no_of_wait_states",no_of_wait_states,$bits(no_of_wait_states),UVM_DEC);
  end
  printer.print_string("transfer_type",transfer_type.name());
endfunction : do_print
```

Fig 8.9 do_print method

8.3.2 Slave_tx

➢ Declared all the variables(cs, MOSI, MISO)

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.

➢ 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;}
```

Fig 8.10 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).

➢ 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.

8.4 Sequences

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

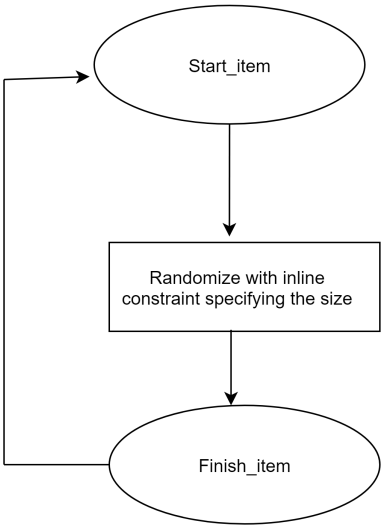


Fig 8.11 Flow chart for sequence methods

8.4.1 Register Sequences

This section defines the sequences used for register stimulus generation.

uvm_reg_sequence	This class provides base functionality for both user-defined RegModel test sequences and "register translation sequences"

When used as a base for user-defined RegModel test sequences, this class provides convenience methods for reading and writing registers and memories.

Users implement the body() method to interact directly with the RegModel model or indirectly via the delegation methods in this class.

Methods:

Method	Description
new	Creates and initializes a new sequence object
Write	Writes the given register *rg* using uvm_reg::write, supplying 'this' as the *parent* argument.
Read	Reads the given register *rg* using uvm_reg::read, supplying 'this' as the *parent* argument.
Update	Updates the given register *rg* using uvm_reg::update, supplying 'this' as the *parent* argument.

Register Masking:

Register Masking allows to mask the fields of each register(to be set or to be clear each bit) which are declared in the Register Model.
The below figure shows the define file with the position and mask of each field in the given register

```
//
//     CS                       4   11:08 RW   0x0        Specify the chip select signal that should be used for the
//                                                        next transfer
//
//     RESERVED5                3   07:05 RO   0x0        Reserved for further usage
//
//     SRST                     1       4 RW   0x0        Clear FIFOs and abort active transfers
//
//     QWR                      1       3 RW   0x0        Perform a write using Quad SPI mode
//
//     QRD                      1       2 RW   0x0        Perform a read using Quad SPI mode
//
//     WR                       1       1 RW   0x0        Perform a write using standard SPI mode
//
//     RD                       1       0 RW   0x0        Perform a read using standard SPI mode
//
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

`define POS_STATUS_CS                                          8
`define POS_STATUS_SRST                                        4
`define POS_STATUS_QWR                                         3
`define POS_STATUS_QRD                                         2
`define POS_STATUS_WR                                          1
`define POS_STATUS_RD                                          0

`define MASK_STATUS_CS                                  (4'hf<<`POS_STATUS_CS)
`define MASK_STATUS_SRST                                (1'h1<<`POS_STATUS_SRST)
`define MASK_STATUS_QWR                                 (1'h1<<`POS_STATUS_QWR)
`define MASK_STATUS_QRD                                 (1'h1<<`POS_STATUS_QRD)
`define MASK_STATUS_WR                                  (1'h1<<`POS_STATUS_WR)
`define MASK_STATUS_RD                                  (1'h1<<`POS_STATUS_RD)
```

Fig 8.12 reg sequence masking concept

Sections	Master sequences	Slave sequences	Description
base_seq	axi4_master_base_reg_seq.sv	spi_slave_base_seq.sv	Base class is extended from uvm_reg_ sequence for master and uvm_sequence for slave and both are parameterized with transaction (axi4_master_ tx, spi_slave_ tx)
Data transfers and std mode	axi4_master_std_mode_write_0_cmd_0_addr_16_data_length_reg_seq.sv	spi_fd_basic_slave_seq.sv	Extended from register base sequences. Based on a request from the driver, the task will drive the transactions. Each field of the registers in a register block will be updated with the specified values respectively. In this SPILEN register is updated with 0 cmd and 0 addr and 16 data length. And with the help of adapters it will convert into the reg2bus transactions and in slave randomizing the mosi size.(size is 4 for 32b) .
	axi4_master_std_mode_write_0_cmd_16_addr_16_data_length_reg_seq.sv	spi_fd_basic_slave_seq.sv	Extended from register base sequences. Based on a request from the driver, the task will drive the transactions. Each field of the registers in a register block will be updated with the specified values respectively. In this SPILEN register is updated with 0 cmd and 16 addr and 16 data length. And with the help of adapters it will convert into the reg2bus transactions and in slave randomizing the mosi size.(size is 4 for 32b) .

	axi4_master_std_mode_write_0_cmd_0_addr_32_data_length_reg_seq.sv	spi_fd_basic_slave_seq.sv	Extended from register base sequences. Based on a request from the driver, the task will drive the transactions. Each field of the registers in a register block will be updated with the specified values respectively. In this SPILEN register is updated with 0 cmd and 0 addr and 32 data length. And with the help of adapters it will convert into the reg2bus transactions and in slave randomizing the mosi size.(size is 4 for 32b) .
	axi4_master_std_mode_write_0_cmd_32_addr_32_data_length_reg_seq.sv	spi_fd_basic_slave_seq.sv	Extended from register base sequences. Based on a request from the driver, the task will drive the transactions. Each field of the registers in a register block will be updated with the specified values respectively. In this SPILEN register is updated with 0 cmd and 32 addr and 32 data length. And with the help of adapters it will convert into the reg2bus transactions and in slave randomizing the mosi size.(size is 4 for 32b) .
	axi4_master_std_mode_write_16_cmd_16_addr_16_data_length_reg_seq.sv	spi_fd_basic_slave_seq.sv	Extended from register base sequences. Based on a request from the driver, the task will drive the transactions. Each field of the registers in a register block will be updated with the specified values respectively. In this SPILEN register is updated with 16 cmd and 16 addr and 16 data length. And with the help of adapters it will convert into the reg2bus transactions and in slave randomizing the mosi size.(size is 4 for 32b) .
	axi4_master_std_mode_write_16_cmd_16_addr_32_data_length_reg_seq.sv	spi_fd_basic_slave_seq.sv	Extended from register base sequences. Based on a request from the driver, the task will drive the transactions. Each field of the registers in a register block will be updated with the specified values respectively. In this SPILEN register is updated with 16 cmd and 16 addr and 32 data length. And with the help of adapters it will convert into the reg2bus transactions and in slave randomizing the mosi size.(size is 4 for 32b) .
	axi4_master_std_mode_write_32_cmd_32_addr_32_data_length_reg_seq.sv	spi_fd_basic_slave_seq.sv	Extended from register base sequences. Based on a request from the driver, the task will drive the transactions. Each field of the registers in a register block will be updated with the specified values respectively. In this SPILEN register is updated with 32 cmd and 32 addr and 32 data length. And with the help of adapters it will convert into the reg2bus transactions and in slave randomizing the mosi size.(size is 4 for 32b)

	axi4_master_std_mode_write_8_cmd_16_addr_32_data_length_reg_seq.sv	spi_fd_basic_slave_seq.sv	Extended from register base sequences. Based on a request from the driver, the task will drive the transactions. Each field of the registers in a register block will be updated with the specified values respectively. In this SPILEN register is updated with 8 cmd and 16 addr and 32 data length. And with the help of adapters it will convert into the reg2bus transactions and in slave randomizing the mosi size.(size is 4 for 32b)
	axi4_master_std_mode_write_8_cmd_32_addr_32_data_length_reg_seq.sv	spi_fd_basic_slave_seq.sv	Extended from register base sequences. Based on a request from the driver, the task will drive the transactions. Each field of the registers in a register block will be updated with the specified values respectively. In this SPILEN register is updated with 8 cmd and 32 addr and 32 data length. And with the help of adapters it will convert into the reg2bus transactions and in slave randomizing the mosi size.(size is 4 for 32b)
	axi4_master_std_mode_write_8_cmd_8_addr_32_data_length_reg_seq.sv	spi_fd_basic_slave_seq.sv	Extended from register base sequences. Based on a request from the driver, the task will drive the transactions. Each field of the registers in a register block will be updated with the specified values respectively. In this SPILEN register is updated with 8 cmd and 32 addr and 32 data length. And with the help of adapters it will convert into the reg2bus transactions and in slave randomizing the mosi size.(size is 4 for 32b)
Dummy Cycles	axi4_master_std_mode_write_0_dummy_write_reg_seq.sv	spi_fd_basic_slave_seq.sv	Extended from register base sequences. Based on a request from the driver, the task will drive the transactions. Each field of the registers in a register block will be updated with the specified values respectively. In this SPIDUM register is updated with 0 write dummy cycles. And with the help of adapters it will convert into the reg2bus transactions and in slave randomizing the mosi size.(size is 4 for 32b)
	axi4_master_std_mode_write_8_dummy_write_reg_seq.sv	spi_fd_basic_slave_seq.sv	Extended from register base sequences. Based on a request from the driver, the task will drive the transactions. Each field of the registers in a register block will be updated with the specified values respectively. In this SPIDUM register is updated with 8 write dummy cycles. And with the help of adapters it will convert into the reg2bus transactions and in slave randomizing the mosi size.(size is 4 for 32b)

	axi4_master_std_mode_write_16_dummy_write_reg_seq.sv	spi_fd_basic_slave_seq.sv	Extended from register base sequences. Based on a request from the driver, the task will drive the transactions. Each field of the registers in a register block will be updated with the specified values respectively. In this SPIDUM register is updated with 16 write dummy cycles. And with the help of adapters it will convert into the reg2bus transactions and in slave randomizing the mosi size.(size is 4 for 32b)
Clock Divider	axi4_master_std_mode_write_even_clkdiv_reg_seq.sv	spi_fd_basic_slave_seq.sv	Extended from register base sequences. Based on a request from the driver, the task will drive the transactions. Each field of the registers in a register block will be updated with the specified values respectively. In this CLKDIV register is updated with even divisor value. And with the help of adapters it will convert into the reg2bus transactions and in slave randomizing the mosi size.(size is 4 for 32b)
Random Sequence	axi4_master_rand_reg_seq.sv	spi_fd_basic_slave_seq.sv	Extended from register base sequences. Based on a request from the driver, the task will drive the transactions. Each field of the registers in a register block will be updated with the specified values respectively. Here all the registers in the register model are randomized. And with the help of adapters it will convert into the reg2bus transactions and in slave randomizing the mosi size.(size is 4 for 32b)

Master_seq body holds the handle for user defined register model and cast the default or parent register model handle with the child or user defined register model handle and it also has the handle for register map which has the access to all the register addresses along with that it contains wdata and rdata both of register type. And in this the actual masking of each bit or the certain value can be done as shown in below figs…..

```
task axi4_master_basic_write_read_reg_seq::body();
 // super.body();
  spi_master_apb_if spi_master_reg_block;
  uvm_reg_map spi_reg_map;

  uvm_status_e status;
  uvm_reg_data_t wdata;
  uvm_reg_data_t rdata;

  $cast(spi_master_reg_block, model);

  spi_reg_map = spi_master_reg_block.get_map_by_name("SPI_MASTER_MAP_ABP_IF");
```

Fig 8.13 master_seq body method

```
// Writing into the register
begin

  bit [5:0] cmd_length;
  bit [5:0] addr_length;
  bit [15:0] data_length;
  cmd_length  = 6'h08;
  addr_length = 6'h08;
  data_length = 16'h20;

  `uvm_info(get_type_name(), $sformatf("Write :: Register cmd_length  = %0h",cmd_length) , UVM_LOW)
  `uvm_info(get_type_name(), $sformatf("Write :: Register addr_length = %0h",addr_length), UVM_LOW)
  `uvm_info(get_type_name(), $sformatf("Write :: Register data_length = %0h",data_length), UVM_LOW)

  // Clearing and writing the required feilds
  wdata = (wdata & (~`MASK_SPILEN_DATALEN)) | (data_length << `POS_SPILEN_DATALEN) ;
  wdata = (wdata & (~`MASK_SPILEN_ADDRLEN)) | (addr_length << `POS_SPILEN_ADDRLEN);
  wdata = (wdata & (~`MASK_SPILEN_CMDLEN))  | (cmd_length << `POS_SPILEN_CMDLEN)  ;

  //setting the required feilds
  //wdata = wdata | (data_length << `POS_SPILEN_CMDLEN) | (addr_length << `POS_SPILEN_ADDRLEN)
  //(cmd_length << `POS_SPILEN_CMDLEN);

end

//Writing into the SPI_LEN Register
spi_master_reg_block.SPILEN.write(.status(status)      ,
                                  .value(wdata)        ,
                                  .path(UVM_FRONTDOOR) ,
                                  .map(spi_reg_map)    ,
                                  .parent(this)
                                 );
```

Fig 8.14 reg master sequence

8.5 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 virtual_base_seq extends uvm_sequence;
  `uvm_object_utils(virtual_base_seq)

  //Declaring p_sequencer
  `uvm_declare_p_sequencer(virtual_sequencer)

  //variable : apb_master_vsqr_h
  //Declaring handle to the virtual sequencer
  axi4_master_write_sequencer  axi4_master_write_seqr_h;
  //axi4_master_read_sequencer  axi4_master_read_seqr_h;

  //variable : spi_slave_vsqr_h
  //Declaring handle to the virtual sequencer
  spi_slave_sequencer spi_slave_seqr_h;

  //-------------------------------------------------------
  // Externally defined Tasks and Functions
  //-------------------------------------------------------
  extern function new(string name = "virtual_base_seq");
  extern task body();
endclass : virtual_base_seq
```

Fig 8.15 base virtual seq

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 sequencer with local master sequencer and slave sequencer.

```
task virtual_base_seq::body();
  if(!$cast(p_sequencer,m_sequencer))begin
    `uvm_error(get_full_name(),"Virtual sequencer pointer cast failed")
  end

  axi4_master_write_seqr_h = p_sequencer.axi4_master_write_seqr_h;
  //axi4_master_read_seqr_h = p_sequencer.axi4_master_read_seqr_h;

  spi_slave_seqr_h  = p_sequencer.spi_slave_seqr_h;

endtask : body
```

Fig 8.16 virtual_base_seq 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 virtual_basic_write_read_reg_seq::body();
  super.body();

  fork
    forever begin : SLAVE_SEQ
      `uvm_info("slave_vseq",$sformatf("started slave vseq"),UVM_HIGH)
      write_read_key.get(1);
      spi_fd_basic_slave_seq_h =
                        spi_fd_basic_slave_seq::type_id::create("spi_fd_basic_slave_seq_h");
      spi_fd_basic_slave_seq_h.start(p_sequencer.spi_slave_seqr_h);
      write_read_key.put(1);
      `uvm_info("slave_vseq",$sformatf("ended slave vseq"),UVM_HIGH)
    end
  join_none

  repeat(1) begin
   `uvm_info("master_vseq",$sformatf("started master vseq"),UVM_HIGH)
   write_read_key.get(1);
   axi4_master_basic_write_read_reg_seq_h =
   axi4_master_basic_write_read_reg_seq::type_id::create("axi4_master_basic_write_read_reg_seq_h");
   axi4_master_basic_write_read_reg_seq_h.model = p_sequencer.env_config_h.spi_master_reg_block;
   axi4_master_basic_write_read_reg_seq_h.start(p_sequencer.axi4_master_write_seqr_h);
   //axi4_master_basic_write_read_reg_seq_h.start(p_sequencer.axi4_master_read_seqr_h);
   write_read_key.put(1);
   `uvm_info("master_vseq",$sformatf("ended master vseq"),UVM_HIGH)
 end
 endtask : body
```

Fig 8.17 virtual basic write read reg seq

For register sequences in virtual sequence we try to connect the default register block handle with the main register block handle which is inside the environment with the help of p_sequencer.

Sections	Virtual sequences	Description
Data transfer	virtual_std_mode_write_0_cmd_0_addr_32_data_length_reg_seq.sv	Inside the std_mode_write_0_cmd_0_addr_32_data_length_reg virtual sequence, extending from base class. Declaring handles of sequences and inside body method constructing handles of sequence and Connecting the sequence default register model handle with the actual register model using p_sequencer. And starting the master and slave sequences on the real master and slave sequencers.
	virtual_std_mode_write_16_cmd_16_addr_16_data_length_reg_seq.sv	Inside the std_mode_write_16_cmd_16_addr_16_data_length_reg virtual sequence, extending from base class. Declaring handles of sequences and inside body method constructing handles of sequence and Connecting the sequence default register model handle with the actual register model using p_sequencer. And starting the master and slave sequences on the real master and slave sequencers.
	virtual_std_mode_write_16_cmd_16_addr_32_data_length_reg_seq.sv	Inside the std_mode_write_16_cmd_16_addr_32_data_length_reg virtual sequence, extending from base class. Declaring handles of sequences and inside body method constructing handles of sequence and Connecting the sequence default register model handle with the actual register model using p_sequencer. And starting the master and slave sequences on the real master and slave sequencers.
	virtual_std_mode_write_0_cmd_0_addr_16_data_length_reg_seq.sv	Inside the std_mode_write_0_cmd_0_addr_16_data_length_reg virtual sequence, extending from base class. Declaring handles of sequences and inside body method constructing handles of sequence and Connecting the sequence default register model handle with the actual register model using p_sequencer. And starting the master and slave sequences on the real master and slave sequencers
	virtual_std_mode_write_0_cmd_16_addr_16_data_length_reg_seq.sv	Inside the std_mode_write_0_cmd_16_addr_16_data_length_reg virtual sequence, extending from base class. Declaring handles of sequences and inside body method constructing handles of sequence and Connecting the sequence default register model handle with the actual register model using p_sequencer. And starting the master and slave sequences on the real master and slave sequencers
	virtual_std_mode_write_0_cmd_32_addr_32_data_length_reg_seq.sv	Inside the std_mode_write_0_cmd_32_addr_32_data_length_reg virtual sequence, extending from base class. Declaring handles of sequences and inside body method constructing handles of sequence and Connecting the sequence default register model handle with the actual register model using p_sequencer. And starting the master and slave sequences on the real master and slave sequencers.

	virtual_std_mode_write_32_cmd_32_addr_32_data_length_reg_seq.sv	Inside the std_mode_write_32_cmd_32_addr_32_data_length_reg virtual sequence, extending from base class. Declaring handles of sequences and inside body method constructing handles of sequence and Connecting the sequence default register model handle with the actual register model using p_sequencer. And starting the master and slave sequences on the real master and slave sequencers.
	virtual_std_mode_write_8_cmd_16_addr_32_data_length_reg_seq.sv	Inside the std_mode_write_8_cmd_16_addr_32_data_length_reg virtual sequence, extending from base class. Declaring handles of sequences and inside body method constructing handles of sequence and Connecting the sequence default register model handle with the actual register model using p_sequencer. And starting the master and slave sequences on the real master and slave sequencers.
	virtual_std_mode_write_8_cmd_32_addr_32_data_length_reg_seq.sv	Inside the std_mode_write_8_cmd_32_addr_32_data_length_reg virtual sequence, extending from base class. Declaring handles of sequences and inside body method constructing handles of sequence and Connecting the sequence default register model handle with the actual register model using p_sequencer. And starting the master and slave sequences on the real master and slave sequencers.
	virtual_std_mode_write_8_cmd_8_addr_32_data_length_reg_seq.sv	Inside the std_mode_write_8_cmd_8_addr_32_data_length_reg virtual sequence, extending from base class. Declaring handles of sequences and inside body method constructing handles of sequence and Connecting the sequence default register model handle with the actual register model using p_sequencer. And starting the master and slave sequences on the real master and slave sequencers.
Clock Divider	virtual_std_mode_write_even_clkdiv_reg_seq.sv	Inside the std_mode_write_even_clockdiv_reg virtual sequence, extending from base class. Declaring handles of sequences and inside body method constructing handles of sequence and Connecting the sequence default register model handle with the actual register model using p_sequencer. And starting the master and slave sequences on the real master and slave sequencers.
	virtual_std_mode_write_odd_clkdiv_seq.sv	Inside the std_mode_write_odd_clockdiv_reg virtual sequence, extending from base class. Declaring handles of sequences and inside body method constructing handles of sequence and Connecting the sequence default register model handle with the actual register model using p_sequencer. And starting the master and slave sequences on the real master and slave sequencers.
Dummy Cycles	virtual_std_mode_write_0_dummy_write_seq.sv	Inside the std_mode_write_0_dummy_write_reg virtual sequence, extending from base class. Declaring handles of sequences and inside body method constructing handles of sequence and Connecting the sequence default register model handle with the actual register model using p_sequencer. And starting the master and slave sequences on the real master and slave sequencers.

	virtual_std_mode_write_16_dummy_write_seq.sv	Inside the std_mode_write_16_dummy_write_reg virtual sequence, extending from base class. Declaring handles of sequences and inside body method constructing handles of sequence and Connecting the sequence default register model handle with the actual register model using p_sequencer. And starting the master and slave sequences on the real master and slave sequencers.
	virtual_std_mode_write_8_dummy_write_seq.sv	Inside the std_mode_write_8_dummy_write_reg virtual sequence, extending from base class. Declaring handles of sequences and inside body method constructing handles of sequence and Connecting the sequence default register model handle with the actual register model using p_sequencer. And starting the master and slave sequences on the real master and slave sequencers.

8.6 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_h
  //Declaring a handle for env
  env env_h;

  //Variable : env_cfg_h
  //Declaring a handle for env_cfg_h
  env_config env_cfg_h;

  // Variable: spi_master_reg_block
  // Registers block for spi master module
  spi_master_apb_if spi_master_reg_block;

  //-------------------------------------------------------
  // 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_config();
  extern virtual function void setup_axi4_master_agent_config();
  extern virtual function void setup_spi_slave_agent_config();
  extern virtual function void end_of_elaboration_phase(uvm_phase phase);
  extern virtual task run_phase(uvm_phase phase);

endclass : base_test
```

Fig 8.18 base_test

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 and creating the RAL model register block and enabling the coverage model in register block by calling the set_coverage function.

```
function void base_test::setup_env_config();
  env_cfg_h = env_config::type_id::create("env_cfg_h");
  env_cfg_h.no_of_spi_slaves  = 1;
  env_cfg_h.has_scoreboard    = 1;
  env_cfg_h.has_virtual_seqr  = 1;
  `uvm_info(get_type_name(),$sformatf("\nenv_CONFIG\n%s",env_cfg_h.sprint()),UVM_LOW);

  //setting up the configuration for master agent
  setup_axi4_master_agent_config();

  //Setting the master agent configuration into config db
  uvm_config_db#(axi4_master_agent_config)::set(this,
      "*master_agent*","axi4_master_agent_config",env_cfg_h.axi4_master_agent_cfg_h);

  `uvm_info(get_type_name(),$sformatf
      ("\naxi4_master_agent_CONFIG\n%s",env_cfg_h.axi4_master_agent_cfg_h.sprint()),UVM_LOW);

  setup_spi_slave_agent_config();

  uvm_config_db#(env_config)::set(this,"*","env_config",env_cfg_h);
  //`uvm_info(get_type_name(),$sformatf("\nenv_CONFIG\n%s",env_cfg_h.sprint()),UVM_LOW);

  // Creation of RAL
  if(spi_master_reg_block == null)
  begin
    uvm_reg::include_coverage("*",UVM_CVR_ALL);

    spi_master_reg_block = spi_master_apb_if::type_id::create("spi_master_reg_block");
    spi_master_reg_block.build();

    // Enables sampling of coverage
    void'(spi_master_reg_block.set_coverage(UVM_CVR_ALL));

    spi_master_reg_block.lock_model();
  end

  env_cfg_h.spi_master_reg_block = this.spi_master_reg_block;

endfunction : setup_env_config
```

Fig 8.19 setup_env config

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(has_coverage,no.of slaves,is_active) in master_agent_config class has been setup.

```
function void base_test::setup_axi4_master_agent_config();

  env_cfg_h.axi4_master_agent_cfg_h =
      axi4_master_agent_config::type_id::create("axi4_master_agent_config");
  if(MASTER_AGENT_ACTIVE === 1) begin
    env_cfg_h.axi4_master_agent_cfg_h.is_active    = uvm_active_passive_enum'(UVM_ACTIVE);
  end
  else begin
    env_cfg_h.axi4_master_agent_cfg_h.is_active    = uvm_active_passive_enum'(UVM_PASSIVE);
  end
  env_cfg_h.no_of_spi_slaves    = 1;
  env_cfg_h.axi4_master_agent_cfg_h.has_coverage    = 1;


endfunction : setup_axi4_master_agent_config
```

Fig 8.20 setup_master_agent config

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_spi_slave_agent_config();

  env_cfg_h.spi_slave_agent_cfg_h = new[env_cfg_h.no_of_spi_slaves];

  foreach(env_cfg_h.spi_slave_agent_cfg_h[i]) begin
    env_cfg_h.spi_slave_agent_cfg_h[i] = spi_slave_agent_config
            ::type_id::create($sformatf("slave_agent_config[%0d]",i));
    env_cfg_h.spi_slave_agent_cfg_h[i].slave_id     = i;
    env_cfg_h.spi_slave_agent_cfg_h[i].is_active    = uvm_active_passive_enum'(UVM_ACTIVE);
    env_cfg_h.spi_slave_agent_cfg_h[i].spi_mode     = operation_modes_e'(CPOL0_CPHA1);
    env_cfg_h.spi_slave_agent_cfg_h[i].shift_dir    = shift_direction_e'(MSB_FIRST);
    env_cfg_h.spi_slave_agent_cfg_h[i].has_coverage = 1;
    env_cfg_h.spi_slave_agent_cfg_h[i].spi_type     = SIMPLE_SPI;

    uvm_config_db #(spi_slave_agent_config)::set(this,$sformatf("*spi_slave_agent_h[%0d]*",i),
            "spi_slave_agent_config", env_cfg_h.spi_slave_agent_cfg_h[i]);

    `uvm_info(get_type_name(),$sformatf("\npulpino_spi_master_subsystem_SLAVE_CONFIG[%0d]\n%s",
          i,env_cfg_h.spi_slave_agent_cfg_h[i].sprint()),UVM_LOW);
  end
endfunction : setup_spi_slave_agent_config
```

Fig 8.21 setup_slave_agent config

Extend the std_mode_write_even_clkdiv_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 basic_write_read_reg_test extends base_test;
  `uvm_component_utils(basic_write_read_reg_test)

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

endclass : basic_write_read_reg_test
```

Fig 8.22 basic_write_read_reg test

```
task basic_write_read_reg_test::run_phase(uvm_phase phase);
  virtual_basic_write_read_reg_seq virtual_basic_write_read_reg_seq_h;
  virtual_basic_write_read_reg_seq_h =
  virtual_basic_write_read_reg_seq::type_id::create("virtual_basic_write_read_reg_seq_h");

  `uvm_info(get_type_name(),$sformatf("basic_write_read_reg_test"),UVM_LOW);

  phase.raise_objection(this);

  virtual_basic_write_read_reg_seq_h.start(env_h.virtual_seqr_h);

  phase.drop_objection(this);

endtask : run_phase
```

Fig 8.23 run_phase of basic_write_read-reg_test

sections	Test Names	Description
Data transfers	std_mode_write_0_cmd_0_addr_16_ data_length_reg_test.sv	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. The data 0 cmd and 0 addr and 16 bit data length
	std_mode_write_0_cmd_0_addr_32_ data_length_reg_test.sv	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. The data contains 0 cmd and 0 addr and 32 bit data length

	std_mode_write_0_cmd_16_addr_16_data_length_reg_test.sv	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. The data contains 0 cmd and 16 bit addr and 16 bit data length
	std_mode_write_0_cmd_32_addr_32_data_length_reg_test.sv	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. The data contains 0 cmd and 16 bit addr and 16 bit data length
	std_mode_write_16_cmd_16_addr_32_data_length_reg_test.sv	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. The data contains 16 cmd and 16 addr and 32 bit data length
	std_mode_write_16_cmd_16_addr_16_data_length_reg_test.sv	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. The data contains 16 cmd and 16 addr and 16 bit data length
	std_mode_write_32_cmd_32_addr_32_data_length_reg_test.sv	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. The data contains 32 cmd and 32 addr and 32 bit data length
	std_mode_write_8_cmd_16_addr_32_data_length_reg_test.sv	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. The data contains 8 cmd and 16 addr and 32 bit data length
	std_mode_write_8_cmd_32_addr_32_data_length_reg_test.sv	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. The data contains 8 cmd and 32 addr and 32 bit data length

	std_mode_write_8_cmd_8_addr_32_data_length_reg_test.sv	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. The data contains 8 cmd and 8 addr and 32 bit data length
Clock Divider	std_mode_write_even_clkdiv_reg_test.sv	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. The data contains 32 bit data for even clock divider
	std_mode_write_odd_clkdiv_reg_test.sv	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. The data contains 32 bit data for odd clock divider
Dummy Cycles	std_mode_write_0_dummy_write_reg_test.sv	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. The data contains 32 bit data for 0 dummy write cycle
	std_mode_write_8_dummy_read_reg_test.sv	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. The data contains 32 bit data for 2 dummy write cycle
	std_mode_write_16_dummy_read_reg_test.sv	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. The data contains 32 bit data for 16 dummy write cycle
Interrupt Configurations	std_mode_read_thtx_rhtx_cnttx_cntrx_value_2_reg_test.sv	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. The data contains 32 bit data for transmitter and receiver fifos
tx_fifo	`std_mode_write_tx_fifo_reg_test.sv`	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. The data contains 32 bit data for transmitter fifos

8.7 Testlists:

Regression list for PULPINO SPI MASTER SUBSYSTEM Verification

Table 6.7 Testlists

TestCase Names	Description
basic_write_reg_test	Checks for standard mode write spi transfers
basic_write_read_reg_test	Checks for standard mode write read spi transfers
rand_reg_test	Checks for random register values of each register
std_mode_read_0_dummy_read_reg_test	Checks for standard mode read with 0 dummy read cycles
std_mode_read_8_dummy_read_reg_test	Checks for standard mode read with 8 dummy read cycles
std_mode_read_even_clkdiv_reg_test	Checks for standard mode read with even clock divider
std_mode_read_odd_clkdiv_reg_test	Checks for standard mode read with odd clock divider
std_mode_write_0_cmd_0_addr_32_data_length_reg_test	Checks for standard mode write with 0 command bits and 0 address bits and 32 data bits
std_mode_write_0_cmd_0_addr_16_data_length_reg_test	Checks for standard mode write with 0 command bits and 0 address bits and 16 data bits
std_mode_write_0_cmd_16_addr_16_data_length_reg_test	Checks for standard mode write with 0 command bits and 16 address bits and 16 data bits
std_mode_write_16_cmd_16_addr_16_data_length_reg_test	Checks for standard mode write with 16 command bits and 16 address bits and 16 data bits
std_mode_write_8_cmd_8_addr_32_data_length_reg_test	Checks for standard mode write with 8 command bits and 8 address bits and 16 data bits
std_mode_write_8_cmd_32_addr_32_data_length_reg_test	Checks for standard mode write with 8 command bits and 32 address bits and 32 data bits
std_mode_write_16_cmd_16_addr_32_data_length_reg_test	Checks for standard mode write with 16 command bits and 16 address bits and 32 data bits
std_mode_write_8_cmd_16_addr_32_data_length_reg_test	Checks for standard mode write with 8 command bits and 16 address bits and 32 data bits
std_mode_write_0_cmd_32_addr_32_data_length_reg_test	Checks for standard mode write with 0 command bits and 32 address bits and 32 data bits
std_mode_write_32_cmd_32_addr_32_data_length_reg_test	Checks for standard mode write with 32 command bits and 32 address bits and 32 data bits
std_mode_write_tx_fifo_reg_test	Checks for standard mode write with transmitter fifo data
std_mode_write_0_dummy_write_reg_test	Checks for standard mode write with 0 dummy write cycles

std_mode_write_8_dummy_write_test	Checks for standard mode write with 8 dummy write cycles
std_mode_write_16_dummy_write_test	Checks for standard mode write with 16 dummy write cycles
spi_modes_clkdiv_dummy_cycles_cross_reg_test	Checks for cross between different SPI modes with different CLKDIV and different Dummy cycles
spi_modes_transfer_length_interupts_cross_reg_test	Checks for cross between SPI modes and different spi lengths and different interrupts

Chapter 9

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.

subsystem user guide

Chapter 10

References