2021

APB AVIP Project

Contents

Contents 1

List of Table 6

List of Figures 6

List of Abbreviations 9

Chapter 1 10

Introduction 10

1.1 Key features 10

Chapter 2 11

Architecture 11

2.1 APB AVIP Testbench Architecture 11

Chapter 3 13

Implementation 13

3.1 Pin Interface 13

3.2 Testbench Components 14

3.2.1 APB Hdl Top 14

3.2.2 APB Interface 14

3.2.3 APB Master Agent BFM Module 15

3.2.4 APB Master Driver BFM Interface 16

3.2.5 APB Master Monitor BFM Interface 16

3.2.6 APB Slave Agent BFM Module 16

3.2.7 APB Slave Driver BFM Interface 17

3.2.8 APB Slave Monitor BFM Interface 17

3.2.9 APB HVL_TOP 18

3.2.10 APB Environment 18

3.2.11 APB Scoreboard 18

3.2.12 APB Virtual Sequencer 24

3.2.13 APB Master Agent 24

3.2.14 APB Master Sequencer 25

3.2.15 APB Master Driver Proxy 25

3.2.16 APB Master Monitor Proxy 27

3.2.17 APB Slave Agent 27

3.2.18 APB Slave Sequencer 29

3.2.19 APB Slave Driver Proxy 29

3.2.20 APB Slave Monitor Proxy 31

3.2.21 UVM Verbosity 32

Chapter 4 **34**
Directory Structure 34
4.1.Package Content 34
Chapter 5 **36**
Configuration 36
5.1 Global package variables 36
5.2 Master agent configuration 37
5.3 Slave agent configuration 37
5.4 Environment configuration 37
5.5 Memory Mapping 38
5.6 Little Endian and Big Endian 41
Chapter 6 **43**
Verification Plan 43
6.1 Verification plan: 43
6.2 Specifications for APB 43
6.3 Template of Verification Plan 43
6.4 Sections for different test Scenarios 44
6.4.1 Directed test 44
Chapter 7 **45**
7.1 Template of Coverage Plan 45
7.2 Functional Coverage 45
7.3 Uvm_Subscriber 45
7.3.1 Analysis export 47
7.3.2 Write function 47
7.4 Covergroup 47
7.4 Bucket 48
7.5 Coverpoints 49
7.6 Cross coverpoints 49
7.6.1 Illegal bins 49
7.7 Creation of the covergroup 49
7.8 Sampling of the covergroup 50
7.9 Checking for the coverage 50
7.10 Errors 52
Chapter 8 **53**
Test Cases 53
8.1 Test Flow 53
8.2 APB Test Cases FlowChart 53
8.3 Transaction 54

8.3.1 Master_tx 54
8.3.2 Slave_tx 57
8.4 Sequences 57
8.4.1 Methods 57
8.5 Virtual sequences 60
8.6 Test Cases 62
8.7 Testlists 65

Chapter 9 **65**
User Guide 65

Chapter 10 **66**
References 66

List of Table

Table No	Name of the Table	Pg no
Table 3.1	Pin interface signals……...….	13
Table 3.2	UVM Verbosity Priorities……	32
Table 3.3	Descriptions of each Verbosity level…….	33
Table 4.1	Directory path…….	35
Table 5.1	Global package variables……	36
Table 5.2	Master_agent_config……..…	37
Table 5.3	Slave_agent_config…….	37
Table 5.4	Env_config……..	37
Table 5.6.1	Big Endian……..…….	42
Table 5.6.2	Little Endian……..…….	42
Table 6.4.1	Checking coverage closure for No of bits transfers……	44
Table 8.1	Sequence methods……..	57
Table 8.2	Describing master and slave sequences…….	58
Table 8.3	Describing virtual sequences…….	58
Table 8.4	Testlists……	65

List of Figures

Fig no	Name of the Figure	Pg no
Fig 2.1	APB AVIP Architecture	10
Fig 3.1	HDL top	13
Fig 3.2	APB driver bfm instantiation in apb master agent bfm code snippet	14
Fig 3.3	APB monitor bfm instantiation in apb master agent bfm code snippet	14
Fig 3.4	APB driver bfm instantiation in apb slave agent bfm code snippet	15
Fig 3.5	APB monitor bfm instantiation in apb slave agent bfm code snippet	16
Fig 3.6	HVL top	17
Fig 3.7	Connection of the analysis port of the monitor to the scoreboard analysis fifo	18
Fig 3.8	Shows the declaration of slave & master analysis port in the slave & master monitor proxy	18
Fig 3.9	Shows the declaration of master & slave analysis fifo in the scoreboard	19
Fig 3.10	Shows the creation of the master & slave analysis port	19
Fig 3.11	Connection done between the analysis port & analysis fifo export in the env	19
Fig 3.12	Use of get method to get the packet from monitor analysis port	20
Fig 3.13	The comparison of the master pwdata with slave pwdata	20
Fig 3.14	Flow chart of the scoreboard run phase	21
Fig 3.15	Flow chart of scoreboard report phase	22
Fig 3.16	APB master agent build phase code snippet	23
Fig 3.17	APB master agent connect phase code snippet	24
Fig 3.18	Flow chart of communication between apb master driver proxy & apb master driver bfm	25
Fig 3.19	Run phase of apb master driver proxy code snippet	26

Fig 3.20	Flow chart of communication between apb master monitor proxy & apb master monitor bfm	27
Fig 3.21	APB slave agent build phase code snippet	28
Fig 3.22	APB slave agent connect phase code snippet	28
Fig 3.23	Flow chart of communication between apb slave driver proxy & apb slave driver bfm	30
Fig 3.24	Run phase of apb slave driver proxy code snippet	31
Fig 3.25	Flow chart of communication between apb slave monitor proxy & apb slave monitor bfm	31
Fig 3.26	Run phase of apb slave driver proxy code snippet	32
Fig 4.1	Package Structure of APB_AVIP	34
Fig 5.5.1	Memory Mapping Example	38
Fig 5.5.2	Global Parameter declaration	39
Fig 5.5.3	Associative array declaration	39
Fig 5.5.4	Function for memory mapping for max and min value	39
Fig 5.5.5	Local variable declaration in function	40
Fig 5.5.6	Memory mapping procedure in master agent configuration	40
Fig 5.5.7	Declaration slave max & min address range	40
Fig 5.5.8	Memory mapping procedure in slave agent configuration	41
Fig 5.6	Random data indicating msb & lsb bits	41
Fig 6.3.1	Verification plan template	44
Fig 7.1	UVM subscriber	46
Fig 7.2	Monitor & coverage connection	46
Fig 7.3	Write function	47
Fig 7.4.1	covergroup	47
Fig 7.4.2	option.per_instance	48
Fig 7.4.3	Option comment	48
Fig 7.4.4	Bucket	48

Fig 7.5	Coverpoint	49
Fig 7.6	Cross coverpoint	49
Fig 7.7	Creation of covergroup	49
Fig 7.8	Sampling of the covergroup	50
Fig 7.9.1	Simulation log file path	50
Fig 7.9.2	Coverage report path	50
Fig 7.9.3	HTML window showing all coverage	51
Fig 7.9.4	All coverpoints present in the covergroup	51
Fig 7.9.5	Individual coverpoint hit	52
Fig 8.1	Test flow	53
Fig 8.2	APB test cases flowchart	53
Fig 8.3	Constraint for pselx and transfer_size	55
Fig 8.4	do_compare method	56
Fig 8.5	do_copy method	56
Fig 8.6	do_print method	57
Fig 8.7	Flow chart for sequence methods	58
Fig 8.8	Master seq body method	59
Fig 8.9	Slave seq body method	59
Fig 8.10	Virtual base sequence	60
Fig 8.11	Virtual base sequence body	61
Fig 8.12	Virtual 8bit sequence body	61
Fig 8.13	Base test	62
Fig 8.14	Setup env_cfg	63
Fig 8.15	Master_agent_cfg setup	63
Fig 8.16	Slave_agent_cfg setup	64
Fig 8.17	Example for 8bit test	64
Fig 8.18	Run_phase of 8bit_test	64

List of Abbreviations

Abbreviation	Description
uvm	universal verification methodology
apb	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

The Advanced Peripheral Bus (APB) is part of the Advanced Microcontroller Bus Architecture (AMBA) protocol family. It defines a low-cost interface that is optimised for minimal power consumption and reduced interface complexity. The APB protocol is not pipelined, use it to connect to low-bandwidth peripherals that do not require the high performance of the AXI protocol.APB is a non pipelined structure.

1.1 Key features

1. It supports addresses up to 32 bit wide.
2. APB4 support for write strobe signal to enable sparse data transfer on the write data bus.
3. Single Master - Multiple Slaves.
4. Programmable Wait state insertion.
5. Slave supports fine grain control of response per address or per transfer.
6. Programmable character length(multiple of 8 bits).
7. Simple Interface.
8. Suitable for many Peripherals.
9. APB4 supports protected access.
10. Random PSLVERR insertion.
11. Flexibility to send completely configured data.
12. In APB every transfer takes at least two cycles(Setup Phase and Access Phase).

Chapter 2

Architecture

2.1 APB AVIP Testbench Architecture

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

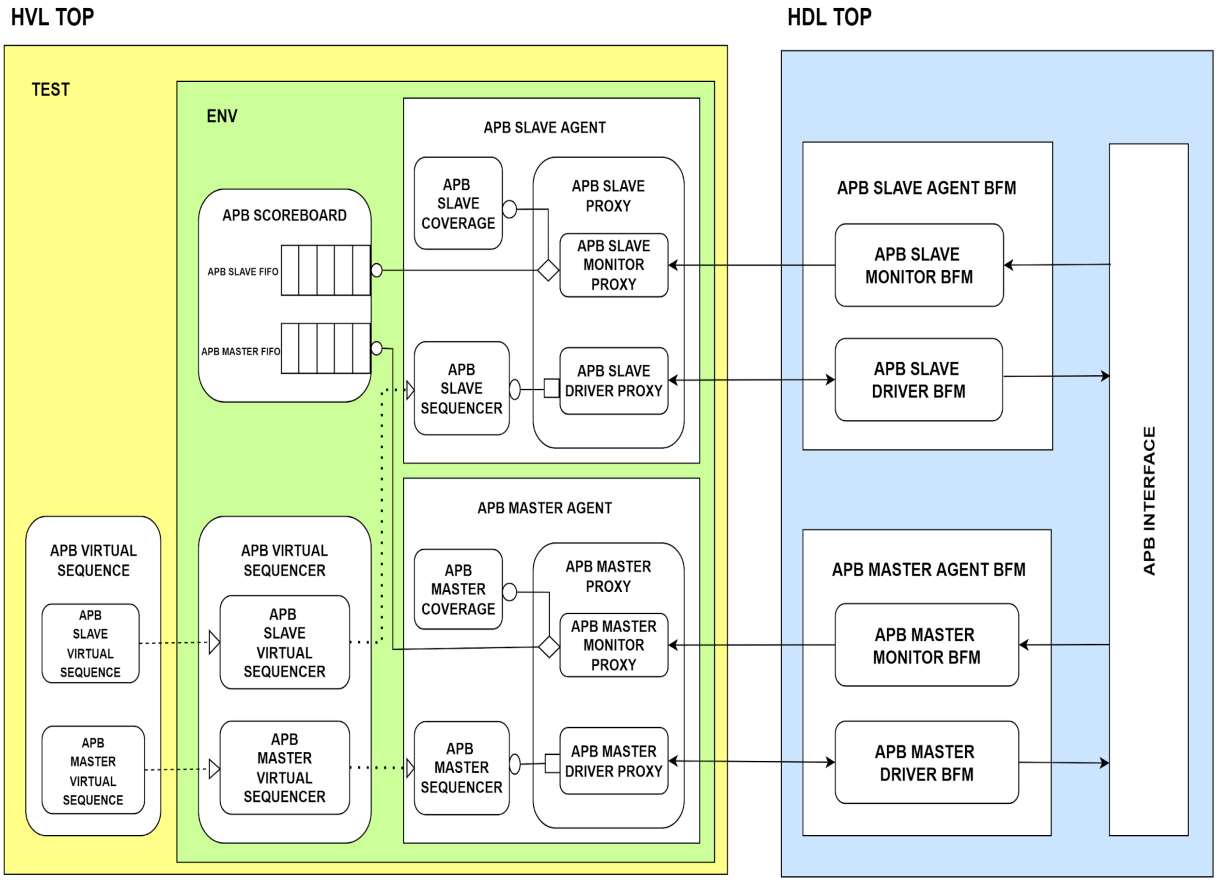


Fig 2.1 APB_AVIP Architecture

HVL TOP has the design which is untimed and the transactions flow from both master virtual sequence and slave virtual sequence onto the APB 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 shows the APB pins used to interface to external devices.

Signals	Source	Description
pclk	Clock source	Clock. The rising edge of PCLK times all transferon the APB.
preset_n	System bus equivalent	Reset. The APB reset signal is active low. This signal is normally connected directly to the system bus reset signal
paddr	APB bridge	Address. This is the APB address bus.It can be up to 32 bits wide and is a data access or an instruction access.
pprot	APB bridge	Protection type. This signal indicates the normal, privileged, or secure protection level of the transaction and whether the transaction is a data access or an instruction access.
pselx	APB bridge	Select. The APB bridge unit generates this signal to each peripheral bus slave. It indicates that the slave device is selected and that a data transfer is required. There is a pselx signal for each slave.
penable	APB bridge	Enable. This signal indicates the second and subsequent cycle of an APB transfer.
pwrite	APB bridge	Direction. This signal indicates an APB write access when HIGH and an APB read access when LOW.
pwdata	APB bridge	Write data. This bus is driven by the peripheral bus bridge unit during the write cycle when pwrite is HIGH. This bus can be up to 32 bits wide.
pstrb	APB bridge	Write strobes. This signal indicates when byte lanes to update during a write transfer. There is one write strobe for each eight bits of the write data bus. Therefore, pstrb[n] corresponds to pwdata[(8n+7):(8n)]. Write strobes must not be active during a read

		transfer.
pready	Slave interface	Ready. The Slave uses this signal to extend an APB transfer.
prdata	Slave interface	Read Data.The selected slave drives this bus during read cycles when pwrite is LOW. This bus can be up to 32-bits wide.
pslverr	Slave interface	This signal indicates a transfer failure. APB peripherals are not required to support the pslverr pin. This is true for both existing and new APB peripheral designs. Where a peripheral does not include this PIN then the appropriate input to the APB bridge is tied LOW.

Table 3.1 APB pins used to interface to external devices

3.2 Testbench Components

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

3.2.1 APB Hdl Top

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

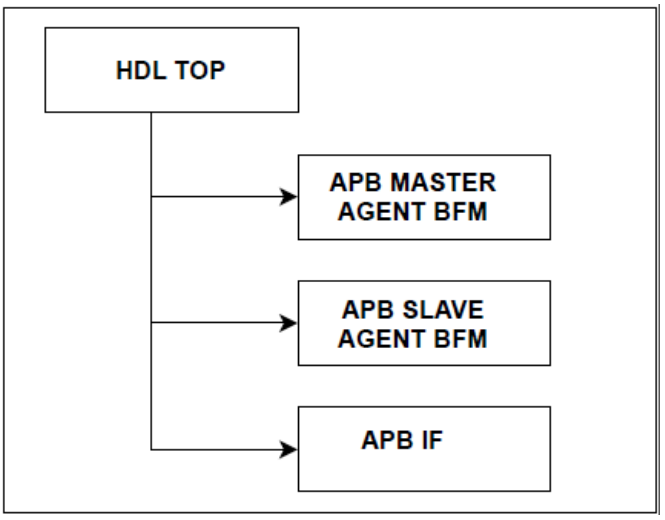


Fig. 3.1 HDL Top

3.2.2 APB Interface

Importing the global packages
Passing Signals: pclk, preset_n

Declaration of signals: paddr, pwrite, pwdata, pstrb, penable, pready, prdata, pslverr, pprot are declared as logic type.

3.2.3 APB Master Agent BFM Module

Instantiates the below two interfaces here

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

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

```
apb_master_driver_bfm apb_master_drv_bfm_h (.pclk(intf.pclk),
                                            .presetn(intf.presetn),
                                            .pselx(intf.pselx),
                                            .penable(intf.penable),
                                            .pprot(intf.pprot),
                                            .paddr(intf.paddr),
                                            .pwrite(intf.pwrite),
                                            .pwdata(intf.pwdata),
                                            .pstrb(intf.pstrb),
                                            .pslverr(intf.pslverr),
                                            .pready(intf.pready),
                                            .prdata(intf.prdata)
                                            );
```

Fig. 3.2 APB driver bfm instantiation in apb master agent bfm code snippet

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

```
apb_master_monitor_bfm apb_master_mon_bfm_h (.pclk(intf.pclk),
                                             .presetn(intf.presetn),
                                             .pselx(intf.pselx),
                                             .paddr(intf.paddr),
                                             .pwrite(intf.pwrite),
                                             .pwdata(intf.pwdata),
                                             .pstrb(intf.pstrb),
                                             .pslverr(intf.pslverr),
                                             .pready(intf.pready),
                                             .prdata(intf.prdata),
                                             .penable(intf.penable),
                                             .pprot(intf.pprot)
                                             );
```

Fig. 3.3 APB monitor bfm instantiation in apb master agent bfm code snippet

3.2.4 APB Master Driver BFM Interface

Apb master driver bfm is an interface where it will get the signals from the apb interface. It has a method drive_to_bfm which will be called by the apb master driver proxy which drives the paddr and pwdata data to the apb interface. fig.3.2 gives the reference of the instantiation of apb master driver bfm.

3.2.5 APB Master Monitor BFM Interface

Apb master monitor bfm is an interface where it will get the signals from the apb interface. It has a method sample_data which will be called by the apb master monitor proxy which samples the paddr, pselx, pwdata and prdata data from the apb interface. After sampling the data, the apb master monitor bfm interface sends the data to the apb master monitor proxy using the output port of sample_data task. fig.3.3 gives the reference of the instantiation of apb master monitor bfm.

3.2.6 APB Slave Agent BFM Module

Instantiates the below two interfaces here

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

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

```
apb_slave_driver_bfm apb_slave_drv_bfm_h(.pclk(intf.pclk),
                                         .presetn(intf.presetn),
                                         .pselx(intf.pselx),
                                         .penable(intf.penable),
                                         .pprot(intf.pprot),
                                         .paddr(intf.paddr),
                                         .pwrite(intf.pwrite),
                                         .pwdata(intf.pwdata),
                                         .pstrb(intf.pstrb),
                                         .pslverr(intf.pslverr),
                                         .pready(intf.pready),
                                         .prdata(intf.prdata));
```

Fig 3.4 APB slave driver bfm instantiation in apb slave agent bfm code snippet

```
apb_slave_monitor_bfm apb_slave_mon_bfm_h (.pclk(intf.pclk),
                                           .preset_n(intf.preset_n),
                                           .psel(intf.pselx),
                                           .paddr(intf.paddr),
                                           .pwrite(intf.pwrite),
                                           .pwdata(intf.pwdata),
                                           .pstrb(intf.pstrb),
                                           .pslverr(intf.pslverr),
                                           .pready(intf.pready),
                                           .prdata(intf.prdata),
                                           .penable(intf.penable),
                                           .pprot(intf.pprot)
                                           );
```

Fig 3.5 APB slave monitor bfm instantiation in apb slave agent bfm code snippet

3.2.7 APB Slave Driver BFM Interface

APB slave driver bfm is an interface where it will get the signals from the apb interface. It has a method drive_to_bfm which will be called by the apb slave driver proxy which drives the prdata data to the apb interface. fig.3.4 gives the reference for the instantiation of apb slave driver bfm.

3.2.8 APB Slave Monitor BFM Interface

APB slave monitor bfm is an interface where it will get the signals from the apb interface. It has a method sample_data which will be called by the apb slave monitor proxy which samples the pwdata and prdata from the apb interface. After sampling the data, the apb slave monitor bfm interface sends the data to the apb slave monitor proxy using the output port of sample_data task. fig.3.5 gives the reference for the instantiation of apb slave monitor bfm.

3.2.9 APB HVL_TOP

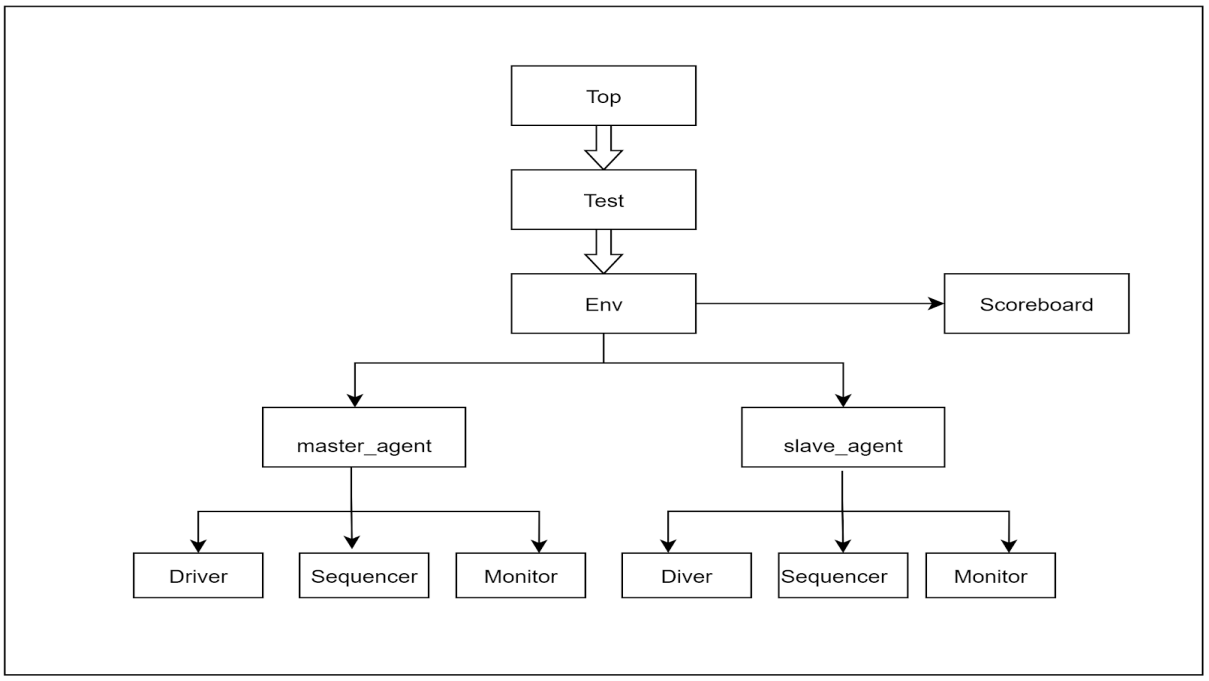


Fig 3.6 HVL Top

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

3.2.10 APB Environment

Environment has the below components

a. apb_scoreboard
b. apb_virtual_sequencer
c. apb_master_agent
d. apb_slave_agent

In the build phase, env_cfg handle will be called and create the memory for the above declared components.
In the connect phase, the apb_master_monitor_proxy is connected to apb_scoreboard and apb_slave_monitor_proxy to apb_scoreboard using analysis port and analysis fifo as shown in fig 3.7.

3.2.11 APB 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 APB-AVIP project is to

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

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

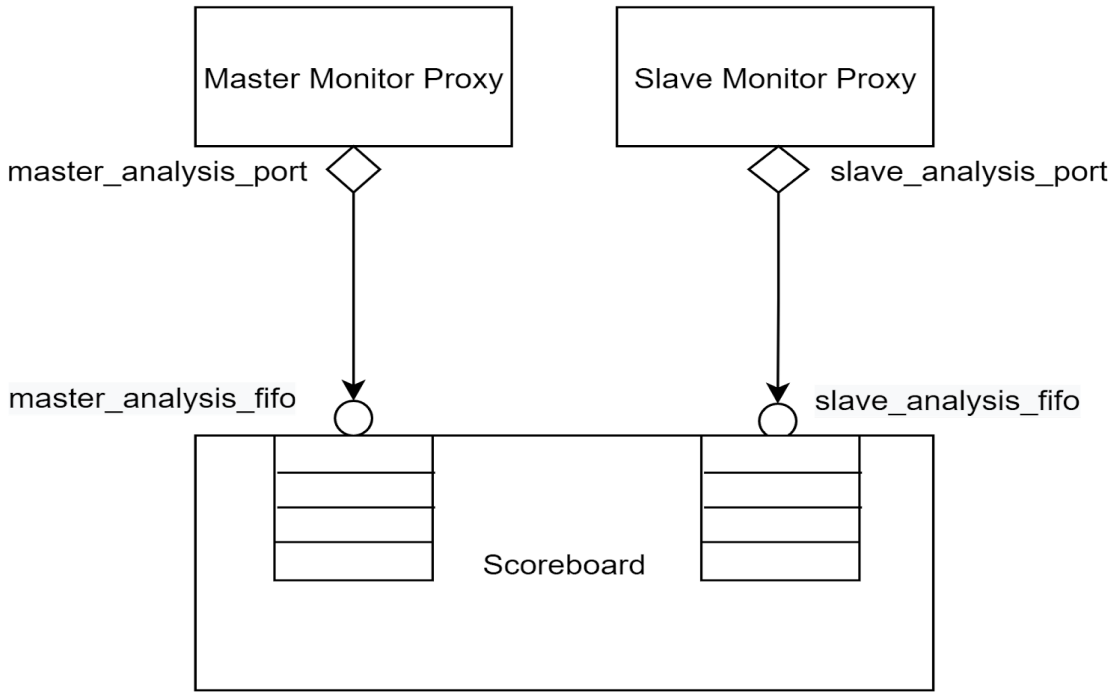


Fig 3.7 connection of the analysis ports of the monitor to the scoreboard analysis fifo

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

```
uvm_analysis_port#(apb_master_tx) apb_master_analysis_port;

uvm_analysis_port#(apb_slave_tx) apb_slave_analysis_port;
```

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

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

```
//Variable : apb_master_analysis_fifo
//Used to store the apb_master_data
uvm_tlm_analysis_fifo#(apb_master_tx) apb_master_analysis_fifo;

//Variable : apb_slave_analysis_fifo
//Used to store the apb_slave_data
uvm_tlm_analysis_fifo#(apb_slave_tx) apb_slave_analysis_fifo[];
```

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

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

```
function apb_scoreboard::new(string name = "apb_scoreboard",uvm_component parent = null);
  super.new(name, parent);
  apb_master_analysis_fifo = new("apb_master_analysis_fifo",this);
  apb_slave_analysis_fifo  = new("apb_slave_analysis_fifo",this);
endfunction : new
```

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

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

```
apb_master_agent_h.apb_master_mon_proxy_h.apb_master_analysis_port.connect(apb_scoreboard_h.apb_master_analysis_fifo.
                                                                           analysis_export);

foreach(apb_slave_agent_h[i]) begin
  apb_slave_agent_h[i].apb_slave_mon_proxy_h.apb_slave_analysis_port.connect(apb_scoreboard_h.apb_slave_analysis_fifo[i].
                                                                             analysis_export);
end
```

Fig 3.11 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 apb_scoreboard::run_phase(uvm_phase phase);
  super.run_phase(phase);
  forever begin
  `uvm_info(get_type_name(),$sformatf("before calling master's analysis fifo get method"),UVM_HIGH)
   apb_master_analysis_fifo.get(apb_master_tx_h);
   `uvm_info(get_type_name(),$sformatf("after calling master's analysis fifo get method"),UVM_HIGH)
   `uvm_info(get_type_name(),$sformatf("printing apb_master_tx_h, \n %s",apb_master_tx_h.sprint()),UVM_HIGH)
   `uvm_info(get_type_name(),$sformatf("before calling slave's analysis_fifo"),UVM_HIGH)
   for(int i=0;i<NO_OF_SLAVES;i++) begin
     if(apb_master_tx_h.pselx[i]==1) begin
       index = i;
       break;
     end
   end
   apb_slave_analysis_fifo[index].get(apb_slave_tx_h);
   `uvm_info(get_type_name(),$sformatf("after calling slave's analysis fifo get method"),UVM_HIGH)
   `uvm_info(get_type_name(),$sformatf("printing apb_slave_tx_h, \n %s",apb_slave_tx_h.sprint()),UVM_HIGH)
```

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

The Comparison of the paddr, pwrite, pwdata and prdata from the master monitor and slave monitor is done in the run phase. Fig 3.13 shows the comparison of the master pwdata with slave pwdata.

```
  if(apb_master_tx_h.pwdata != apb_slave_tx_h.pwdata) begin
    `uvm_error("ERROR_SC_PWDATA_MISMATCH",$sformatf("Master PWDATA = 'h%0x and Slave PWDATA = 'h%0x",
                apb_master_tx_h.pwdata,apb_slave_tx_h.pwdata));
    byte_data_cmp_failed_master_pwdata_count++;
  end
  else begin
    `uvm_info("SB_PWDATA_MATCH",$sformatf("Master PWDATA = 'h%0x and Slave PWDATA = 'h%0x",
               apb_master_tx_h.pwdata,apb_slave_tx_h.pwdata), UVM_HIGH);

    byte_data_cmp_verified_master_pwdata_count++;
  end
```

Fig 3.13 The comparison of the master pwdata with slave pwdata

Similarly, the comparison is done for the signals as well.
Fig 3.14 explains the flow chart of the run phase in the scoreboard.

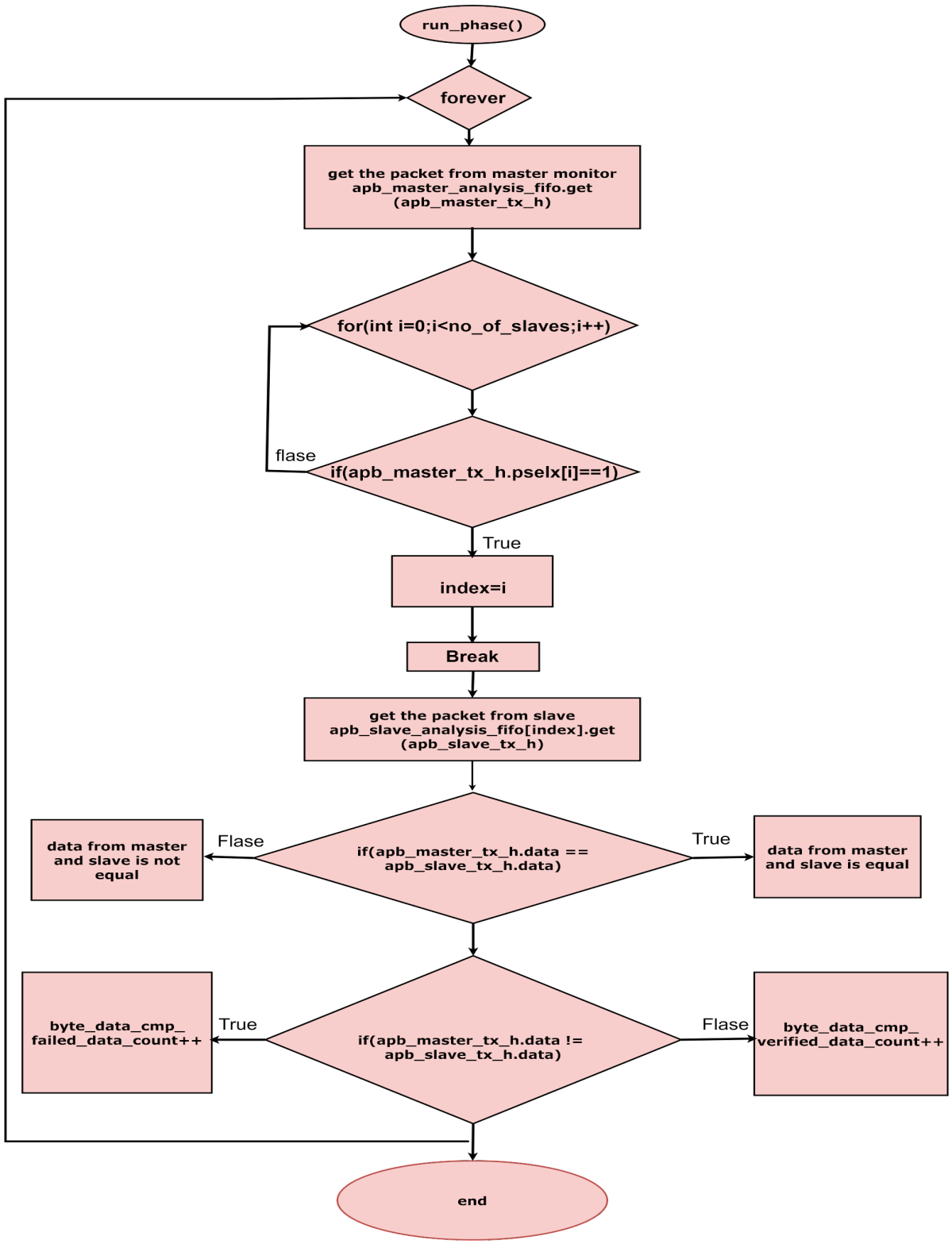


Fig 3.14 Flow chart of the scoreboard run phase

In the run phase, inside the forever loop, the scoreboard master analysis fifo gets the transaction from the master monitor analysis port using the get() method. Whenever the packet is received the transaction counter i.e, master_tx_count will be incremented.

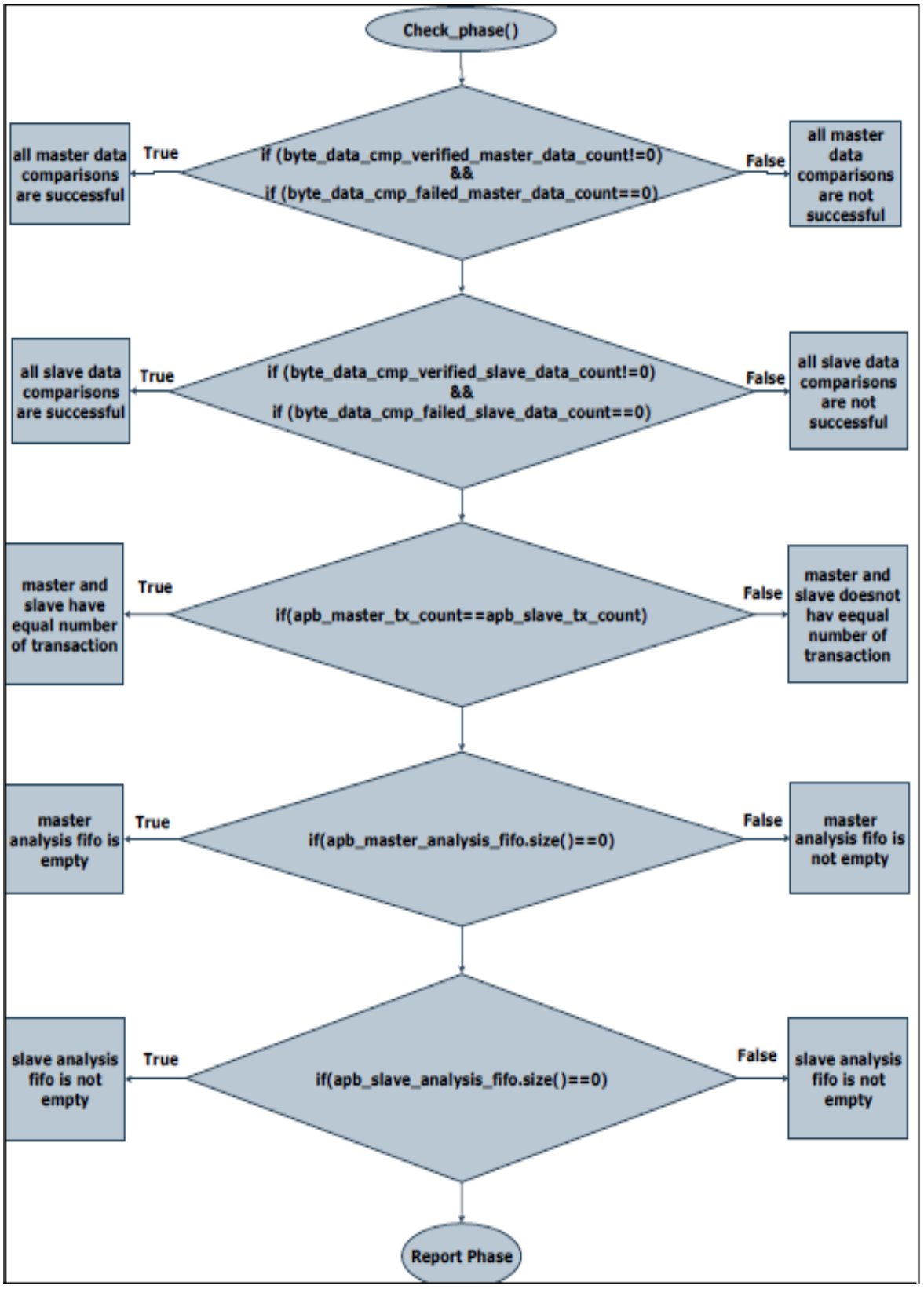


Fig 3.15 Flow chart of the scoreboard report phase

3.2.12 APB Virtual Sequencer

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

3.2.13 APB Master Agent

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

APB master agent build phase has creation of,

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

```
function void apb_master_agent::build_phase(uvm_phase phase);
  super.build_phase(phase);
  if(!uvm_config_db #(apb_master_agent_config)::get(this,"","apb_master_agent_config",apb_master_agent_cfg_h)) begin
    `uvm_fatal("FATAL_MA_CANNOT_GET_APB_MASTER_AGENT_CONFIG","cannot get apb_master_agent_cfg_h from uvm_config_db");
  end
  // Printing the values of the apb_master_agent_config
  // Print method is declared in apb_master_agent_config class and calling it from here
  //`uvm_info(get_type_name(), $sformatf("The apb_master_agent_config \n %s",apb_master_agent_cfg_h.sprint),UVM_LOW);

  if(apb_master_agent_cfg_h.is_active == UVM_ACTIVE) begin
    apb_master_drv_proxy_h=apb_master_driver_proxy::type_id::create("apb_master_drv_proxy_h",this);
    apb_master_seqr_h=apb_master_sequencer::type_id::create("apb_master_seqr_h",this);
  end
  apb_master_mon_proxy_h=apb_master_monitor_proxy::type_id::create("apb_master_mon_proxy_h",this);
  if(apb_master_agent_cfg_h.has_coverage) begin
    apb_master_cov_h = apb_master_coverage::type_id::create("apb_master_cov_h",this);
  end
endfunction : build_phase
```

Fig 3.16 APB master agent build phase code snippet

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

```
function void apb_master_agent::connect_phase(uvm_phase phase);
  if(apb_master_agent_cfg_h.is_active == UVM_ACTIVE) begin
    apb_master_drv_proxy_h.apb_master_agent_cfg_h = apb_master_agent_cfg_h;
    apb_master_seqr_h.apb_master_agent_cfg_h = apb_master_agent_cfg_h;

    // Connecting driver_proxy port to sequencer export
    apb_master_drv_proxy_h.seq_item_port.connect(apb_master_seqr_h.seq_item_export);
  end

  apb_master_mon_proxy_h.apb_master_agent_cfg_h = apb_master_agent_cfg_h;

  if(apb_master_agent_cfg_h.has_coverage) begin
    apb_master_cov_h.apb_master_agent_cfg_h = apb_master_agent_cfg_h;

    // Connecting monitor_proxy port to coverage export
    apb_master_mon_proxy_h.apb_master_analysis_port.connect(apb_master_cov_h.apb_master_analysis_export);
  end

endfunction: connect_phase
```

Fig 3.17 APB master agent connect phase code snippet

3.2.14 APB Master Sequencer

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

3.2.15 APB Master Driver Proxy

APB master driver proxy component is a parameterised class of type apb master transaction, extending uvm_driver. It gets the apb 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 apb driver proxy using get_next_item() method.

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

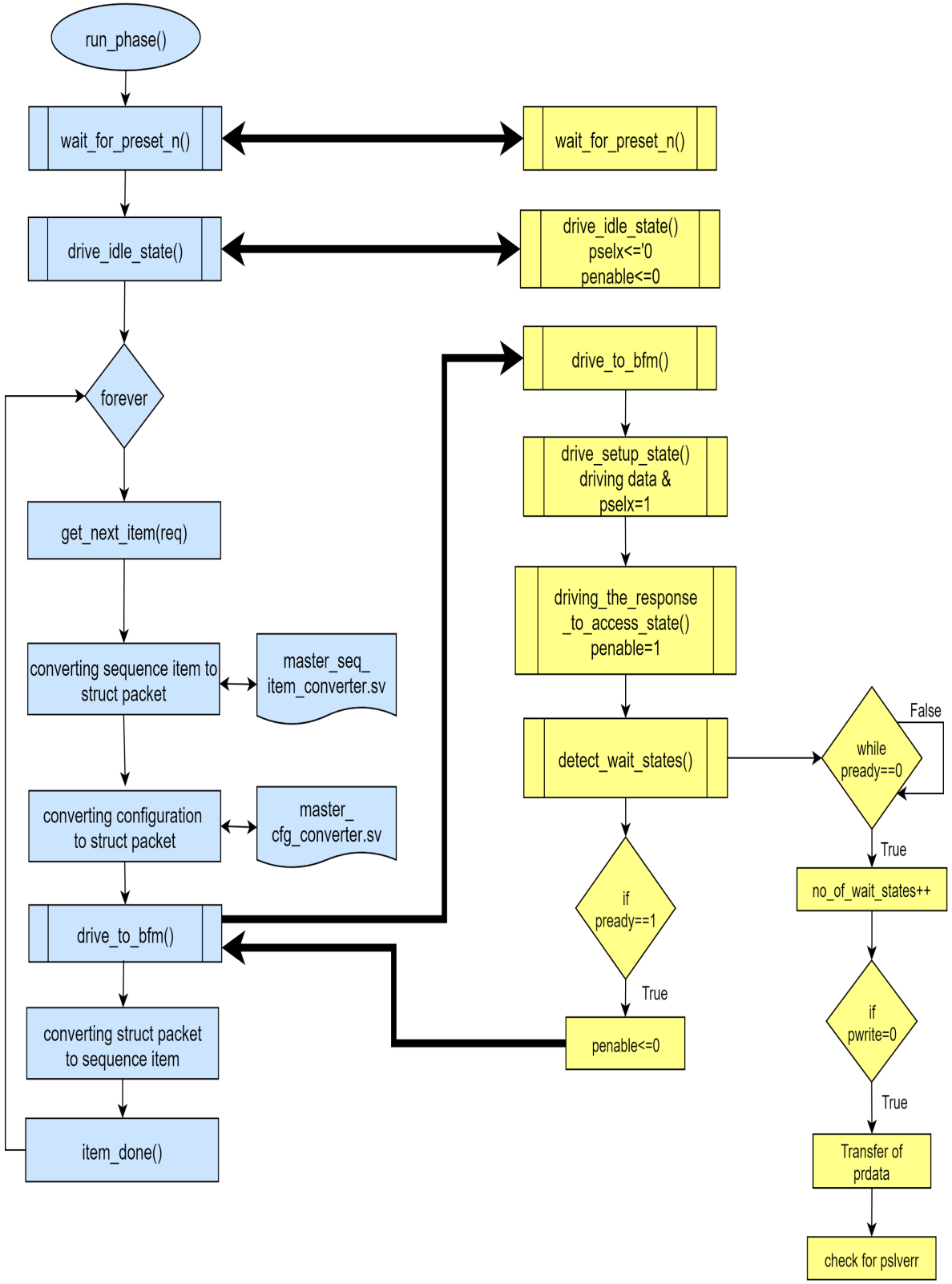


Fig 3.18 Flowchart of communication between apb master driver proxy and apb master driver bfm

```
task apb_master_driver_proxy::run_phase(uvm_phase phase);
  //wait for system reset
  apb_master_drv_bfm_h.wait_for_preset_n();
  forever begin
    apb_transfer_char_s struct_packet;
    apb_transfer_cfg_s struct_cfg;
    seq_item_port.get_next_item(req);
    //Printing the req item
    `uvm_info(get_type_name(), $sformatf("REQ-MASTER_TX \n %s",req.sprint),UVM_LOW);
    //Printing master agent config
    `uvm_info(get_type_name(), $sformatf("apb_master_agent_config \n %s",apb_master_agent_cfg_h.sprint),UVM_LOW);
    //Converting transaction to struct data_packet
    apb_master_seq_item_converter::from_class(req, struct_packet);
    //Converting configurations to struct cfg_packet
    apb_master_cfg_converter::from_class(apb_master_agent_cfg_h, struct_cfg);
    apb_master_drv_bfm_h.drive_to_bfm(struct_packet,struct_cfg);
    //Converting struct to transaction
    apb_master_seq_item_converter::to_class(struct_packet, req);
    seq_item_port.item_done();
  end
endtask : run_phase
```

Fig 3.19 run phase of apb master driver proxy code snippet

3.2.16 APB Master Monitor Proxy

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

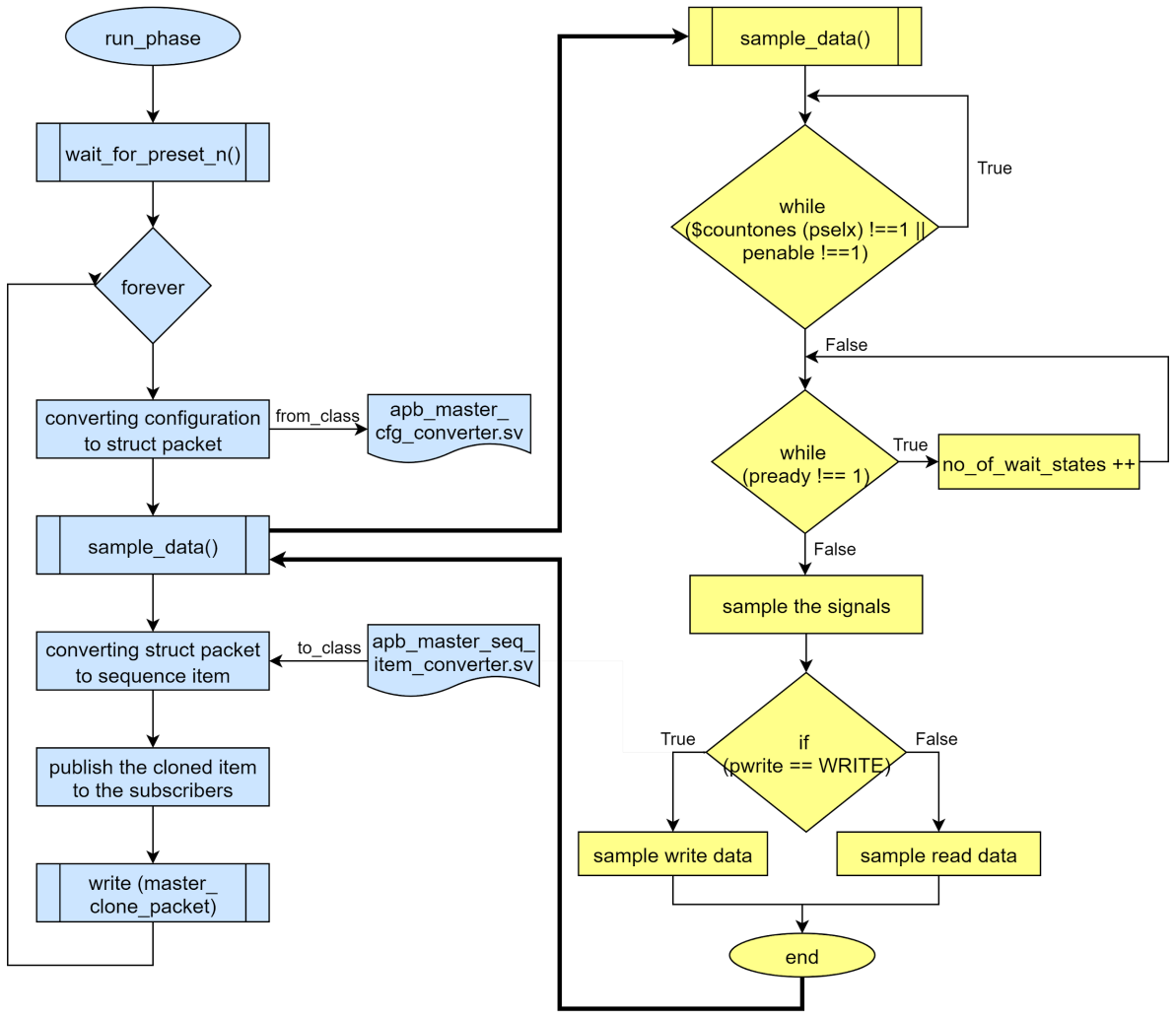


Fig 3.20 Flowchart of apb master monitor proxy and apb master monitor bfm communication

3.2.17 APB Slave Agent

APB slave agent component is a class extending uvm_agent. It gets the apb slave agent configuration and based on that we will create and connect the components. It creates the apb slave sequencer and apb slave driver only if the apb slave agent is active which will depend on the value of is_active variable declared in the apb slave agent configuration file. The apb slave coverage is created in build_phase if has_covergae variable is 1 which is declared in the apb slave agent configuration file.

APB slave agent build phase has creation of,

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

```
function void apb_slave_agent::build_phase(uvm_phase phase);
  super.build_phase(phase);
  if(!uvm_config_db #(apb_slave_agent_config)::get(this,"","apb_slave_agent_config",apb_slave_agent_cfg_h)) begin
    `uvm_fatal("FATAL_SA_AGENT_CONFIG", $sformatf("Couldn't get the apb_slave_agent_config from config_db"))
  end
  // Have a print method in master_agent_config class and call it from here
  //`uvm_info(get_type_name(), $sformatf("The apb_slave_agent_config.slave_id = %0d",slave_agent_cfg_h.slave_id), UVM_LOW);

   if(apb_slave_agent_cfg_h.is_active == UVM_ACTIVE) begin
     apb_slave_drv_proxy_h = apb_slave_driver_proxy::type_id::create("apb_slave_drv_proxy_h",this);
     apb_slave_seqr_h = apb_slave_sequencer::type_id::create("apb_slave_seqr_h",this);
   end
   apb_slave_mon_proxy_h = apb_slave_monitor_proxy::type_id::create("apb_slave_mon_proxy_h",this);

  if(apb_slave_agent_cfg_h.has_coverage) begin
    apb_slave_cov_h = apb_slave_coverage::type_id::create("apb_slave_cov_h",this);
  end
endfunction : build_phase
```

Fig 3.21 APB slave agent build phase code snippet

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

```
function void apb_master_agent::connect_phase(uvm_phase phase);
  if(apb_master_agent_cfg_h.is_active == UVM_ACTIVE) begin
    apb_master_drv_proxy_h.apb_master_agent_cfg_h = apb_master_agent_cfg_h;
    apb_master_seqr_h.apb_master_agent_cfg_h = apb_master_agent_cfg_h;

    // Connecting driver_proxy port to sequencer export
    apb_master_drv_proxy_h.seq_item_port.connect(apb_master_seqr_h.seq_item_export);
  end

  apb_master_mon_proxy_h.apb_master_agent_cfg_h = apb_master_agent_cfg_h;

  if(apb_master_agent_cfg_h.has_coverage) begin
    apb_master_cov_h.apb_master_agent_cfg_h = apb_master_agent_cfg_h;

    // Connecting monitor_proxy port to coverage export
    apb_master_mon_proxy_h.apb_master_analysis_port.connect(apb_master_cov_h.apb_master_analysis_export);
  end

endfunction: connect_phase
```

Fig 3.22 APB slave agent connect phase code snippet

3.2.18 APB Slave Sequencer

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

3.2.19 APB Slave Driver Proxy

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

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

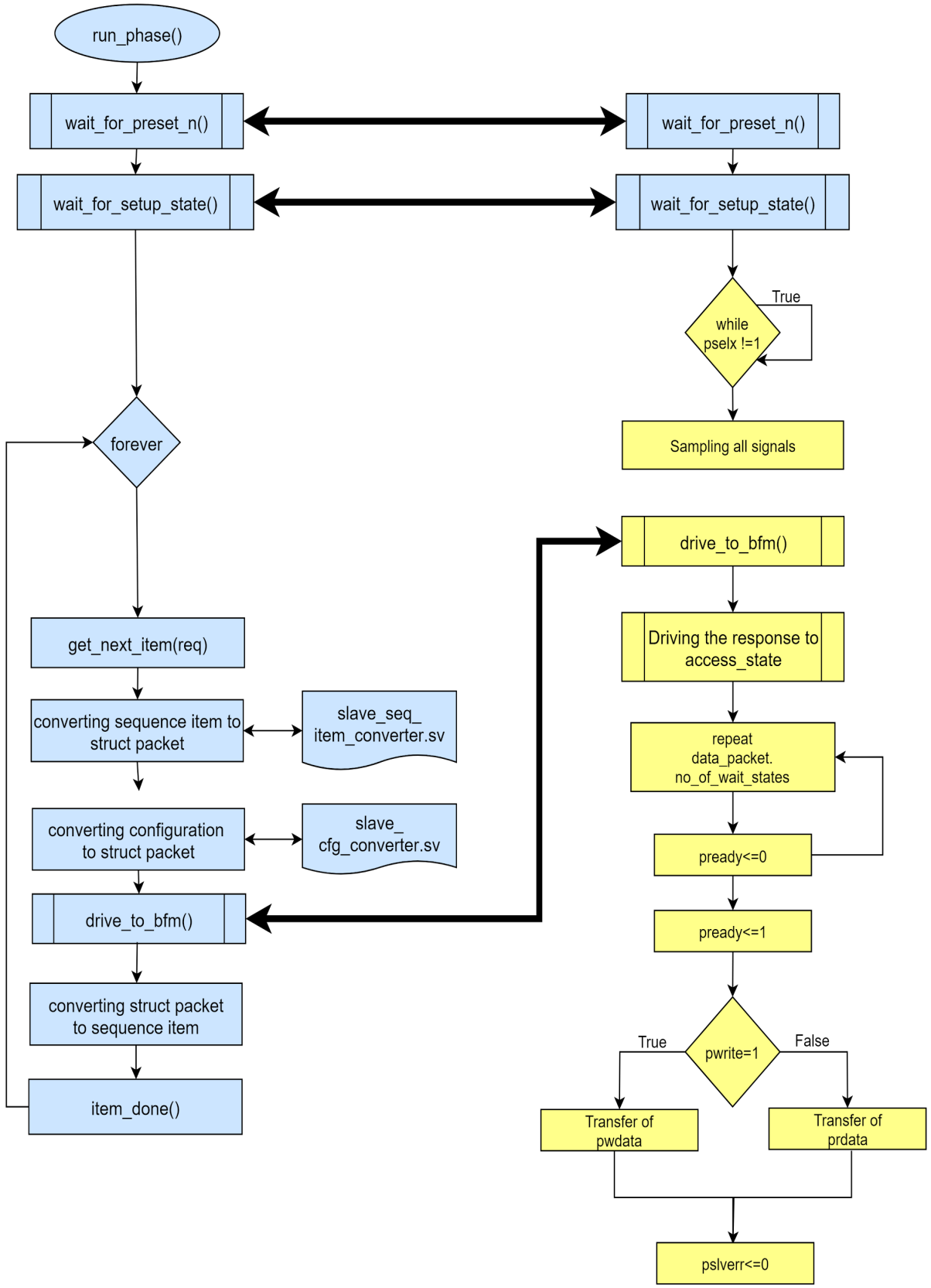


Fig 3.23 Flowchart of apb slave driver bfm and slave driver proxy communication

```
task apb_slave_driver_proxy::run_phase(uvm_phase phase);
  //wait for system reset
  apb_slave_drv_bfm_h.wait_for_presetn();
  forever begin
    apb_transfer_char_s struct_packet;
    apb_transfer_cfg_s struct_cfg;
    seq_item_port.get_next_item(req);
    //Printing the req item
    //Printing slave agent config
    `uvm_info(get_type_name(),$sformatf("\n apb_slave_agent_config\n%s",apb_slave_agent_cfg_h.sprint),UVM_LOW);
    //Converting transaction to struct data_packet
    apb_slave_seq_item_converter::from_class(req, struct_packet);
    //Converting configurations to struct cfg_packet
    apb_slave_cfg_converter::from_class(apb_slave_agent_cfg_h, struct_cfg);
    //drive the converted data packets to the slave driver bfm
    apb_slave_drv_bfm_h.drive_to_bfm(struct_packet,struct_cfg);
    //converting the struct data items into transcations
    apb_slave_seq_item_converter::to_class(struct_packet, req);
    seq_item_port.item_done();
  end
  endtask : run_phase
```

Fig 3.24 APB slave driver proxy run phase code snippet

3.2.20 APB Slave Monitor Proxy

APB slave monitor proxy component is a class extending uvm_monitor. It gets the apb slave agent config handle and based on the configurations we will sample the pwdata and prdata signals. It declares and creates the apb slave analysis port to send the sampled data. The apb slave monitor proxy will get the sampled data from apb master monitor bfm as shown in figure 3.25.

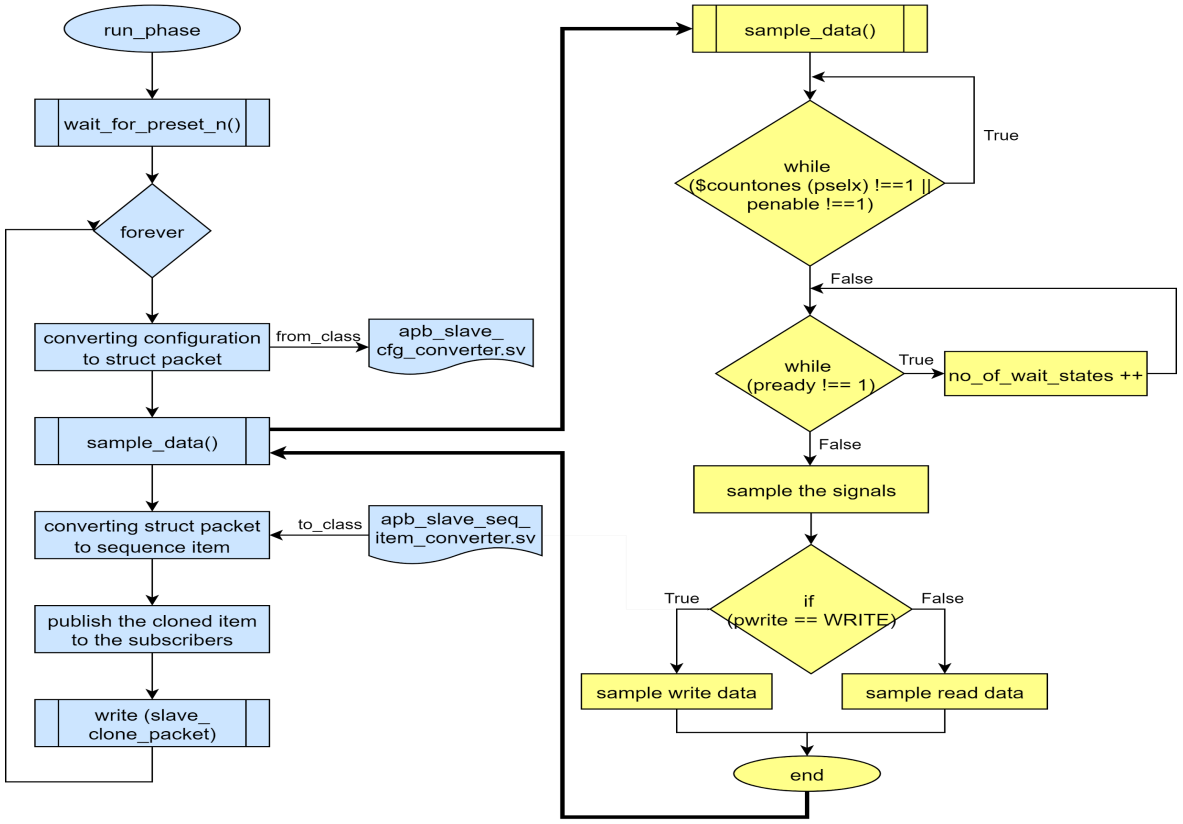


Fig 3.25 Flowchart of apb slave monitor bfm and slave monitor proxy communication

```
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 3.26 run phase of apb slave monitor proxy code snippet

3.2.21 UVM Verbosity

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

Table 3.2 UVM verbosity Priorities

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

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

Table 3.3 Descriptions of each Verbosity level

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

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.

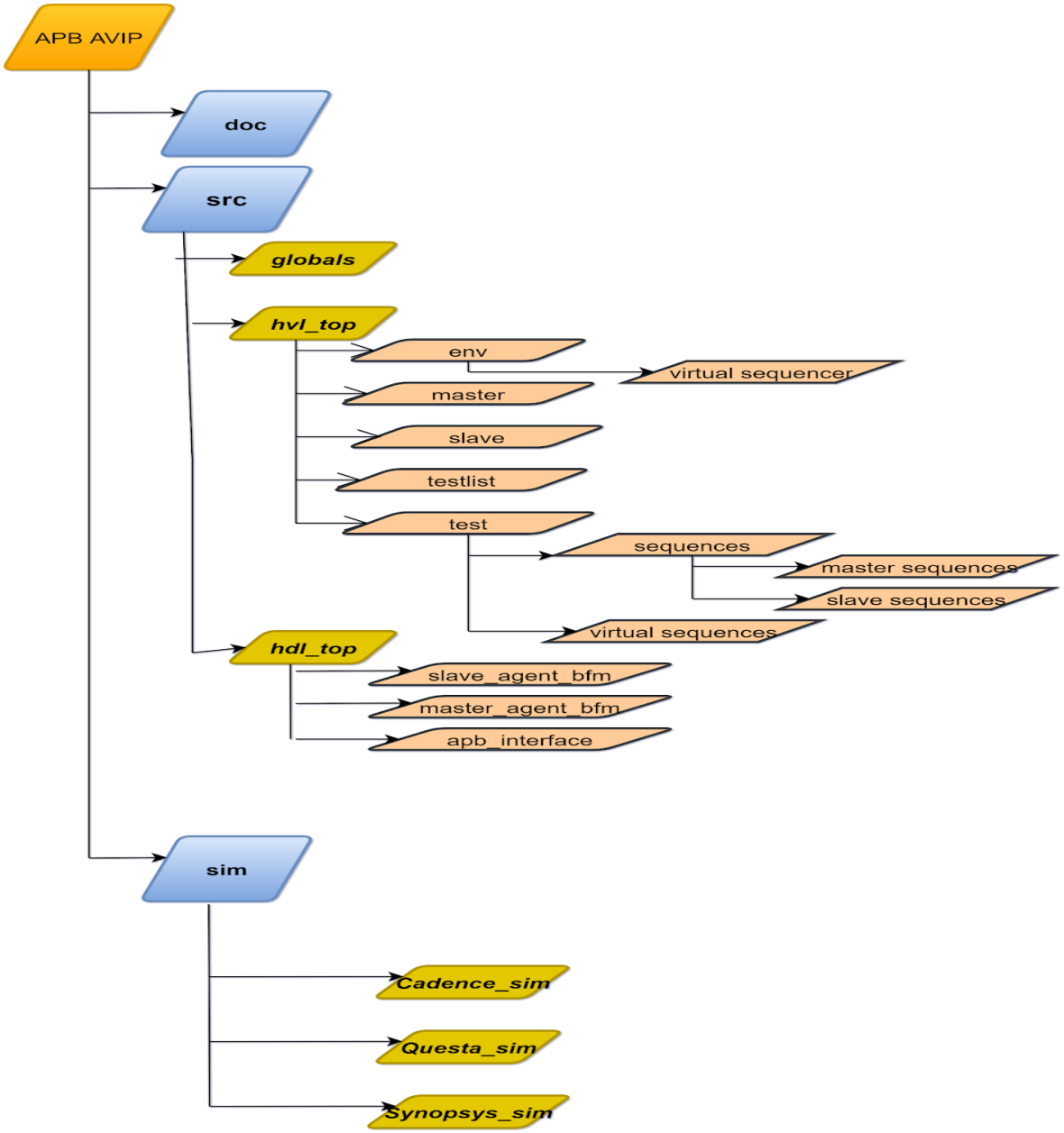


Figure 4.1. Package Structure of APB_AVIP

Table 4.1 Directory Path

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

Chapter 5

Configuration

5.1 Global package variables

Table 5.1 Global package variables

Name	Type	Description
NO_OF_SLAVES	integer	specifies no of slaves connected to the APB interface
APB_transfer_char_s	struct	Structure to hold the packet data.
APB_transfer_cfg_s	struct	Structure to hold the configuration data.
apb_fsm_state_e	enum	Represents the type of state IDLE SETUP ACCESS
slave_no_e	enum	Used to select the one slave at a time by one hot encoding SLAVE_0 = 16'b0000_0000_0000_0001 SLAVE_1 = 16'b0000_0000_0000_0010 SLAVE_2 = 16'b0000_0000_0000_0100 SLAVE_3 = 16'b0000_0000_0000_1000 SLAVE_4 = 16'b0000_0000_0001_0000 SLAVE_5 = 16'b0000_0000_0010_0000
endian_e	enum	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	enum	WRITE=1'b1 : write transfer happen READ=1'b0 : read transfer happen
protectiontype_e	enum	Used to represent the protection type for transaction NORMAL_SECURE_DATA = 3'b000 NORMAL_SECURE_INSTRUCTION = 3'b001 NORMAL_NONSECURE_DATA = 3'b010 NORMAL_NONSECURE_INSTRUCTION = 3'b011 PRIVILEGED_SECURE_DATA = 3'b100 PRIVILEGED_SECURE_INSTRUCTION = 3'b101
slave_error_e	enum	Represents slave error signal NO_ERROR=1'b0; ERROR=1'b1;

Configuration used

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

5.2 Master agent configuration

Table 5.2 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.
no_of_slaves	integer	‘d1	Used for specifying the number of slaves connected to the master
has_coverage	integer	‘d1	Used for enabling the master agent coverage

5.3 Slave agent configuration

Table 5.3 Slave_agent_config

Name	Type	Default value	Description
is_active	enum	UVM_ACTIVE	It will be used for configuring agent as an active agent means it has sequencer,driver and monitor and if it’s a passive agent then it will have only monitor
slave _id	integer	‘d0	Used for indicating the ID of this slave e.g. slave 0 is selected.
has_coverage	integer	‘d1	Used for enabling the slave agent coverage.

5.4 Environment configuration

Table 5.4 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

5.5 Memory Mapping

Memory-mapping is a mechanism that maps a portion of a file, or an entire file, on disk to a range of addresses within an application's address space.

In APB, the memory mapping means that the slave's address ranges are stored in the master's associative arrays so that master has access to each slave's address range.

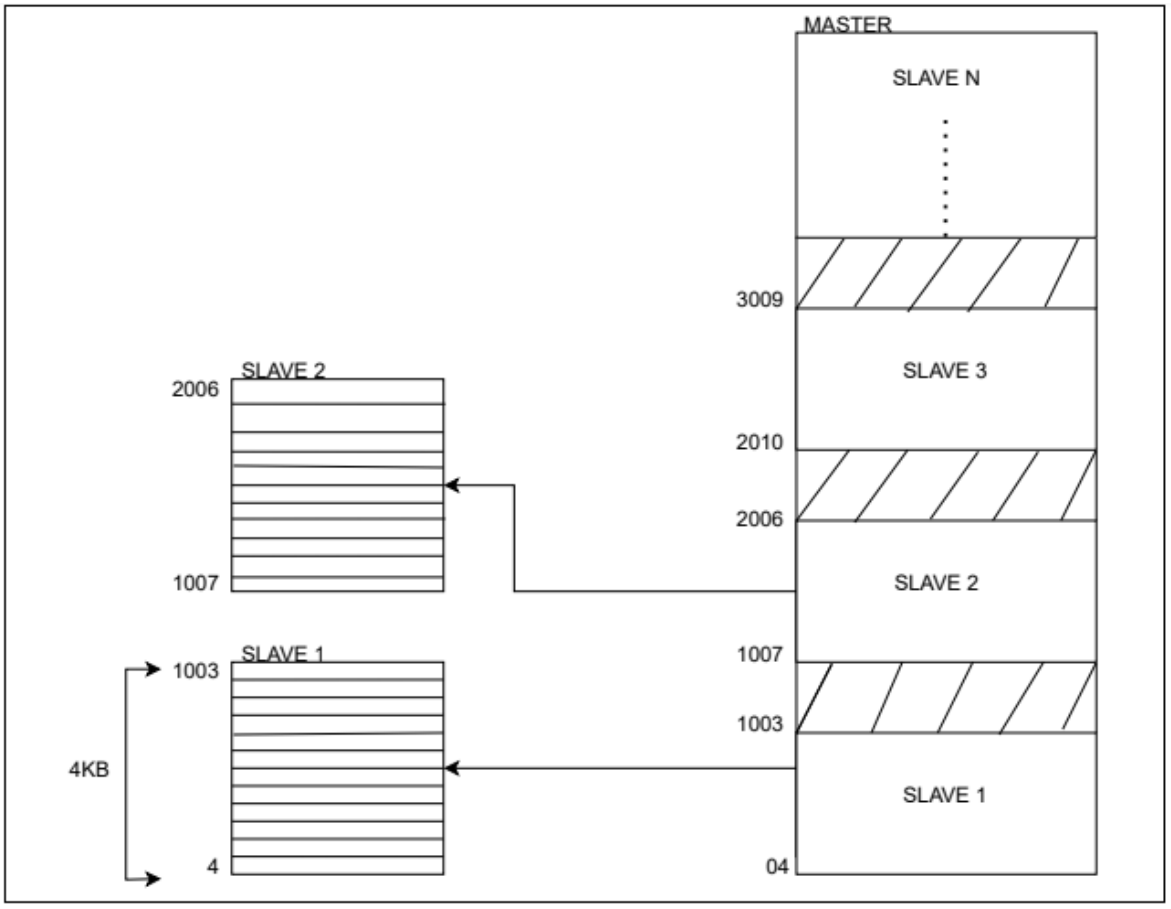


Fig. 5.5.1: Memory mapping example

Memory mapping in APB

An example of memory mapping is as shown in fig. 1. Initially in global package, the memory is taken as ***4KB***, i.e., the slave memory size is taken as ***12***, because (2**ADDRESS_DEPTH = MEMORY_SIZE) i.e., **(2**12 = 4096)** as shown in fig. 5.5.2.

Each Slave memory is given a gap of 2 locations, so that each memory mapping can be differentiated easily as shown in fig. 5.5.2.

```
//Parameter : SLAVE_MEMORY_SIZE
//Sets the memory size of the slave in KB
parameter int SLAVE_MEMORY_SIZE = 12;

//Parameter : SLAVE_MEMORY_GAP
//Sets the memory gap size of the slave
parameter int SLAVE_MEMORY_GAP = 2;
```

Fig. 5.5.2: Global parameter declaration

An associative array is used to store the ***max and min address ranges*** of every slave in master agent configuration, where [int] is the index type as shown in fig. 5.5.3.

```
//Variable : master_memory
//Used to store all the data from the slaves
//Each location of the master memory stores 8 bit data
bit [MEMORY_WIDTH-1:0]master_memory[(SLAVE_MEMORY_SIZE+SLAVE_MEMORY_GAP)*NO_OF_SLAVES:0];

//Variable : master_min_array
//An associative array used to store the min address ranges of every slave
//Index - type    - int
//        stores  - slave number
//Value - stores the minimum address range of that slave.
bit [ADDRESS_WIDTH-1:0]master_min_addr_range_array[int];

//Variable : master_max_array
//An associative array used to store the max address ranges of every slave
//Index - type    - int
//        stores  - slave number
//Value - stores the maximum address range of that slave.
bit [ADDRESS_WIDTH-1:0]master_max_addr_range_array[int];
```

Fig.5.5.3: Associative array declaration

In master agent configuration, two functions are written so that the value obtained will be stored in the master array as shown in fig. 5.5.4.

```
function void apb_master_agent_config::master_max_addr_range(int slave_number, bit[ADDRESS_WIDTH-1:0]slave_max_address_range);
  master_max_addr_range_array[slave_number] = slave_max_address_range;
endfunction : master_max_addr_range

//-------------------------------------------------------------------------------------------
// Function : master_min_addr_range_array
// Used to store the minimum address ranges of the slaves in the array
// Parameters :
//  slave_number           - int
//  slave_min_address_range - bit [63:0]
//-------------------------------------------------------------------------------------------
function void apb_master_agent_config::master_min_addr_range(int slave_number, bit[ADDRESS_WIDTH-1:0]slave_min_address_range);
  master_min_addr_range_array[slave_number] = slave_min_address_range;
endfunction : master_min_addr_range
```

Fig.5.5.4: Functions for memory mapping for max and min value

The memory mapping is done in base_test as shown in fig. 5. In a setup_apb_master_agent_config(), initially, we declare 2 local variables to store the min and max address used for each iteration as shown in fig. 5.5.5.

```
function void apb_base_test::setup_apb_master_agent_config();
  bit [63:0]local_min_address;
  bit [63:0]local_max_address;
  apb_env_cfg_h.apb_master_agent_cfg_h = apb_master_agent_config::type_id::create("apb_master_agent_config");
  if(MASTER_AGENT_ACTIVE === 1) begin
    apb_env_cfg_h.apb_master_agent_cfg_h.is_active    = uvm_active_passive_enum'(UVM_ACTIVE);
  end
  else begin
    apb_env_cfg_h.apb_master_agent_cfg_h.is_active    = uvm_active_passive_enum'(UVM_PASSIVE);
  end
  apb_env_cfg_h.apb_master_agent_cfg_h.no_of_slaves   = NO_OF_SLAVES;
  apb_env_cfg_h.apb_master_agent_cfg_h.has_coverage   = 1;
```

Fig 5.5.5: Local variable declaration in function

The function setup_apb_master_agent_config will start pushing the maximum and minimum address ranges to the respective associative arrays by adding a memory gap of 4 and making sure that start address is mod of 4 as shown in fig. 5.5.6.

```
  for(int i =0; i<NO_OF_SLAVES; i++) begin

    apb_env_cfg_h.apb_master_agent_cfg_h.master_min_addr_range(i,local_max_address + 2**SLAVE_MEMORY_GAP);
    local_min_address = apb_env_cfg_h.apb_master_agent_cfg_h.master_min_addr_range_array[i];

    apb_env_cfg_h.apb_master_agent_cfg_h.master_max_addr_range(i,local_max_address+ 2**(SLAVE_MEMORY_SIZE)-1 + 2**SLAVE_MEMORY_GAP
);
    local_max_address = apb_env_cfg_h.apb_master_agent_cfg_h.master_max_addr_range_array[i];

  end
```

Fig 5.5.6: Memory mapping procedure in master agent configuration

In slave agent configuration, the ***slave max and min address range*** is declared as shown in fig. 5.5.7. Created the slave memory of type associative array so that each salve can store the data received from master with the respective address as key.

```
//Variable : max_address
//Used to store the maximum address value of this slave
bit [ADDRESS_WIDTH-1:0]max_address;

//Variable : min_address
//Used to store the minimum address value of this slave
bit [ADDRESS_WIDTH-1:0]min_address;

//Variable : slave_memory
//Declaration of slave_memory to store the data from master
bit [7:0]slave_memory[longint];
```

Fig 5.5.7: Declaration of slave max and min address range

Similarly for Slave, the mapping is done as shown in fig. 5.5.8. The same index value is mapped for the slave memory, so that the slave stores the data in the same address range for memory. Each slave's minimum and maximum addressess are sent to the respective slave agent configurations from the stored maximum and minimum address ranges in the master agent configuration.

```
function void apb_base_test::setup_apb_slave_agent_config();
  apb_env_cfg_h.apb_slave_agent_cfg_h = new[apb_env_cfg_h.no_of_slaves];
  foreach(apb_env_cfg_h.apb_slave_agent_cfg_h[i]) begin
    apb_env_cfg_h.apb_slave_agent_cfg_h[i] = apb_slave_agent_config::type_id::create($sformatf("apb_slave_agent_config[%0d]",i));
    apb_env_cfg_h.apb_slave_agent_cfg_h[i].slave_id       = i;
    apb_env_cfg_h.apb_slave_agent_cfg_h[i].slave_selected = 0;
    apb_env_cfg_h.apb_slave_agent_cfg_h[i].min_address    = apb_env_cfg_h.apb_master_agent_cfg_h.master_min_addr_range_array[i];
    apb_env_cfg_h.apb_slave_agent_cfg_h[i].max_address    = apb_env_cfg_h.apb_master_agent_cfg_h.master_max_addr_range_array[i];
    if(SLAVE_AGENT_ACTIVE === 1) begin
      apb_env_cfg_h.apb_slave_agent_cfg_h[i].is_active  = uvm_active_passive_enum'(UVM_ACTIVE);
    end
    else begin
      apb_env_cfg_h.apb_slave_agent_cfg_h[i].is_active  = uvm_active_passive_enum'(UVM_PASSIVE);
    end
    apb_env_cfg_h.apb_slave_agent_cfg_h[i].has_coverage = 1;
    uvm_config_db #(apb_slave_agent_config)::set(this,$sformatf("*env*"),$sformatf("apb_slave_agent_config[%0d]",i),apb_env_cfg_h.
apb_slave_agent_cfg_h[i]);
    `uvm_info(get_type_name(),$sformatf("\nAPB_SLAVE_CONFIG[%0d]\n%s",i,apb_env_cfg_h.apb_slave_agent_cfg_h[i].sprint()),UVM_LOW);
  end
endfunction : setup_apb_slave_agent_config
```

Fig.5.5.8: Memory mapping procedure in slave agent configuration

5.6 Little Endian and Big Endian

ENDIAN CONCEPT

Endian refers to the bytes order in which data is stored in the memory and also describes the order of byte transmission over a digital link.

Basically Endian comes in two types: Little endian and Big endian

Fig. 5.6. Indicates the MSB and LSB bits of a Data.

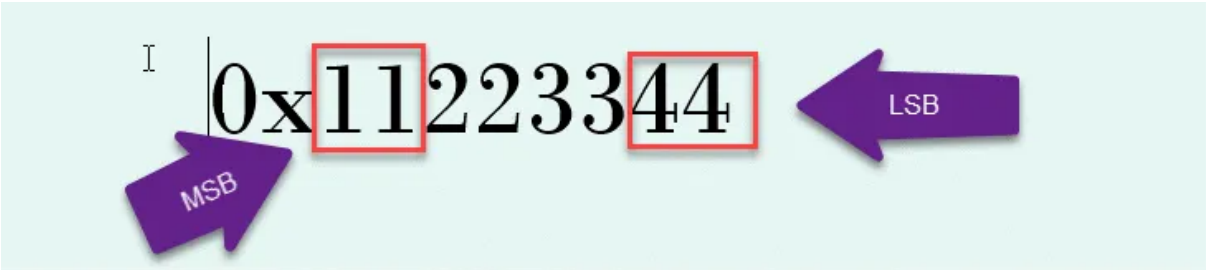


Fig. 5.6: Random Data indicating MSB and LSB bits

If your machine is big-endian then the MSB byte store first (means at lower address) and if the machine is the little-endian then LSB byte store first (means at lower address).

Big-endian

In big-endian MSB Byte will store first. It means the ***MSB Byte*** will store at the ***lowest memory*** address location as shown in table 1.

Address	Value
00	0x11
01	0x22
02	0x33
03	0x44

Table 5.6.1: Big-endian

Little endian

In the little endian machine, LSB byte will store first. So the LSB Byte will store at the lowest memory address as shown in table 2.

Address	Value
00	0x44
01	0x33
02	0x22
03	0x11

Table5.6.2: Little-endian

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 APB Specification:
APB-Protocol-Specification-v2-0

6.2 Specifications for APB

Bits of transfers (8, 16, 24, maximum bits 32)

Four lanes of PSTRB

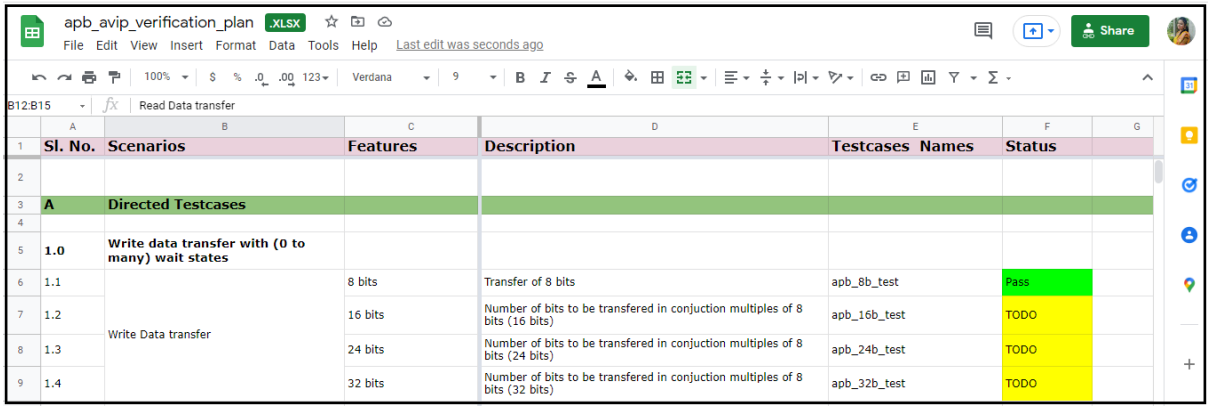
- LANE 0
- LANE 1
- LANE 2
- LANE 3

Wait state & No wait state

6.3 Template of Verification Plan

For more information of Verification Plan click below link:
APB-Verification Plan

In the below Figure,
Section A represents the Sl.No
Section B represents the Scenarios
Section C represents the Features
Section D represents the Description
Section E represents the Test Cases names
Section F represents the Status



apb_avip_verification_plan .XLSX

File Edit View Insert Format Data Tools Help Last edit was seconds ago

Share

B12:B15 Read Data transfer

	A	B	C	D	E	F	G
1	Sl. No.	Scenarios	Features	Description	Testcases Names	Status	
2							
3	A	Directed Testcases					
4							
5	1.0	Write data transfer with (0 to many) wait states					
6	1.1	Write Data transfer	8 bits	Transfer of 8 bits	apb_8b_test	Pass	
7	1.2		16 bits	Number of bits to be transfered in conjuction multiples of 8 bits (16 bits)	apb_16b_test	TODO	
8	1.3		24 bits	Number of bits to be transfered in conjuction multiples of 8 bits (24 bits)	apb_24b_test	TODO	
9	1.4		32 bits	Number of bits to be transfered in conjuction multiples of 8 bits (32 bits)	apb_32b_test	TODO	

Fig. 6.3.1 Verification plan template

6.4 Sections for different test Scenarios

6.4.1 Directed test

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

No of bits transfers -

This tests describes the different combinations of number of bits transfer

S.NO	No of bit transfers	Test names	Description
1	8bit	apb_8b_write_test	Checking the 8 bit transfer
2	16bit	apb_16b_write_test	Checking the 16bit transfer
3	24bit	apb_24b_write_test	Checking the 24bit transfer
4	32bit	apb_32b_write_test	Checking the 32 bit transfers

Table 6.4.1. Checking coverage closure for No of bits transfers

Chapter 7

Coverage

7.1 Template of Coverage Plan

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

apb_avip_coverage_plan

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

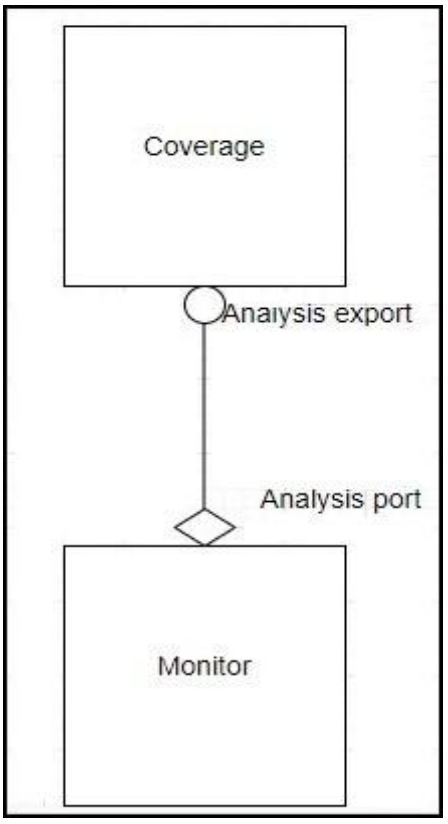


Fig 7.2. Monitor and coverage connection

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 apb_master_coverage::write(apb_master_tx t);
  `uvm_info(get_type_name(),$sformatf("Before calling SAMPLE METHOD"),UVM_HIGH);

  apb_master_covergroup.sample(apb_master_agent_cfg_h,t);

  `uvm_info(get_type_name(),"After calling SAMPLE METHOD",UVM_HIGH);
  // cg.sample(master_agent_cfg_h, master_tx_cov_data);
endfunction : write
```

Fig 7.3: Write function

7.4 Covergroup

```
covergroup apb_master_covergroup with function sample (apb_master_agent_config cfg, apb_master_tx packet);
3 option.per_instance = 1;

  // To check the number slaves we used
  PSEL_CP : coverpoint slave_no_e'(packet.pselx) {
4   option.comment = " psel of apb";
    bins APB_PSELX[] = {[0:NO_OF_SLAVES]};
  }
```

Fig 7.4.1: 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 apb_master_covergroup with function sample (apb_master_agent_config cfg, apb_master_tx packet);
  option.per_instance = 1;
```

Figure 7.4.2 : option.per_instance

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

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

Comment:
apb protection sin=gnal

Search:

Bin Name	At Least	Hits
APB_PPROT[NORMAL_SECURE_DATA]	1	0
APB_PPROT[NORMAL_SECURE_INSTRUCTION]	1	0
APB_PPROT[NORMAL_NONSECURE_DATA]	1	1
APB_PPROT[NORMAL_NONSECURE_INSTRUCTION]	1	1
APB_PPROT[PRIVILEGED_SECURE_DATA]	1	0
APB_PPROT[PRIVILEGED_SECURE_INSTRUCTION]	1	1
APB_PPROT[PRIVILEGED_NONSECURE_DATA]	1	1
APB_PPROT[PRIVILEGED_NONSECURE_INSTUCTION]	1	1

Fig 7.4.3: 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.

```
bins APB_PADDR[] ={[0:7]};  // 8 bins will be created one-one for the each values
bins APB_PADDR  = {[0:ADDRESS_WIDTH-1]}; //one bin is created for 0 to address_width
```

Fig 7.4.4: 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.

```
PWRITE_CP : coverpoint tx_type_e'(packet.pwrite) {
  option.comment = "apb write or read operation";
  bins READ_DATA = {0};
  bins WRITE_DATA = {1};
}
```

Fig 7.5: 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:

```
PADDR_CP_X_PWDATA_CP : cross PADDR_CP , PWDATA_CP;
PADDR_CP_X_PRDATA_CP : cross PADDR_CP , PRDATA_CP;
```

Fig 7.6: 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 apb_master_coverage::new(string name = "apb_master_coverage", uvm_component parent = null);
  super.new(name, parent);
  apb_master_covergroup = new();
  //apb_master_analysis_export = new("apb_master_analysis_export",this);
endfunction : new
```

Fig 7.7: 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 apb_master_coverage::write(apb_master_tx t);
  `uvm_info(get_type_name(),$sformatf("Before calling SAMPLE METHOD"),UVM_HIGH);

  apb_master_covergroup.sample(apb_master_agent_cfg_h,t);

  `uvm_info(get_type_name(),"After calling SAMPLE METHOD",UVM_HIGH);
  // cg.sample(master_agent_cfg_h, master_tx_cov_data);
endfunction : write
```

Fig 7.8.: Sampling of the covergroup

7.9 Checking for the coverage

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

```
Log file path: apb_8b_write_test/apb_8b_write_test.log
```

Fig 7.9.1: Simulation log file path

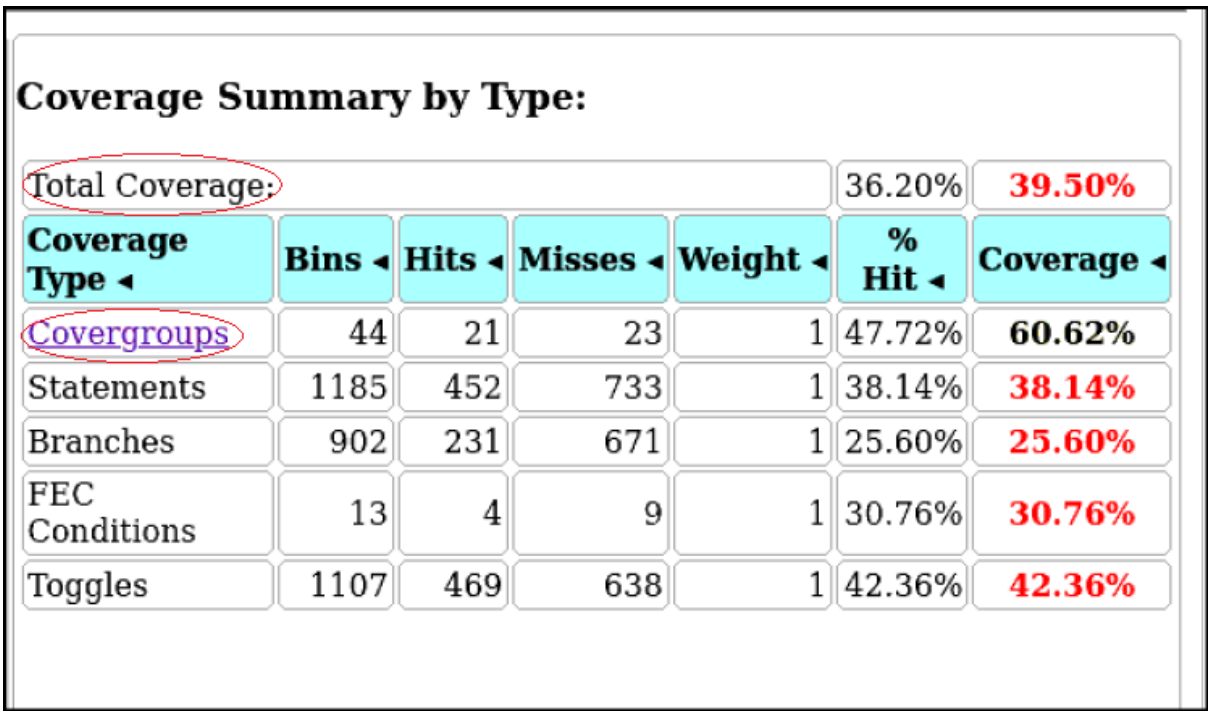
4. Search for the coverage (There it will be the full coverage) in the log file.

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

```
Coverage report: firefox apb_8b_write_test/html_cov_report/index.html &
```

Fig 7.9.2: Coverage report path

Then new html window will open

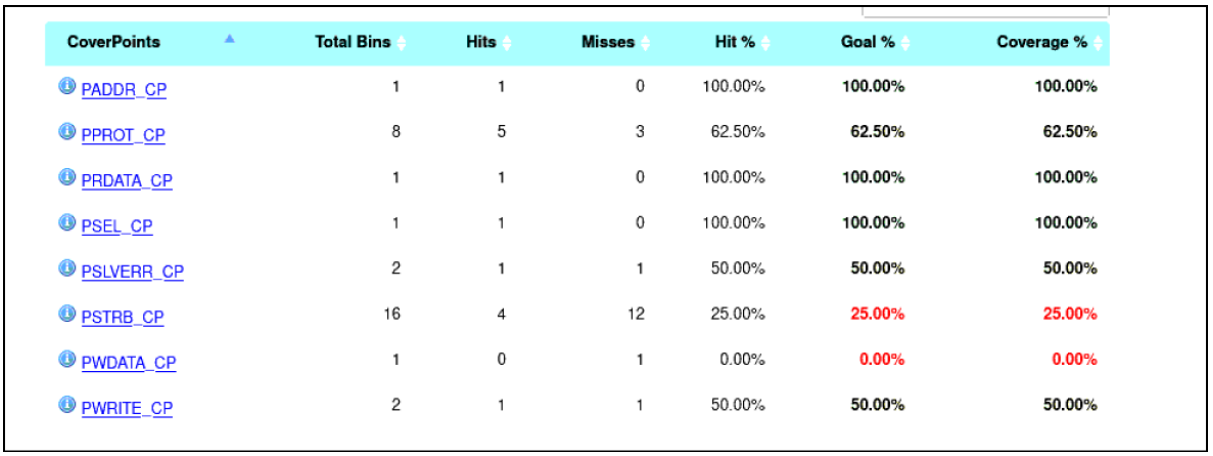


Coverage Summary by Type:

Total Coverage:					36.20%	39.50%
Coverage Type	**Bins**	**Hits**	**Misses**	**Weight**	**% Hit**	**Coverage**
Covergroups	44	21	23	1	47.72%	**60.62%**
Statements	1185	452	733	1	38.14%	**38.14%**
Branches	902	231	671	1	25.60%	**25.60%**
FEC Conditions	13	4	9	1	30.76%	**30.76%**
Toggles	1107	469	638	1	42.36%	**42.36%**

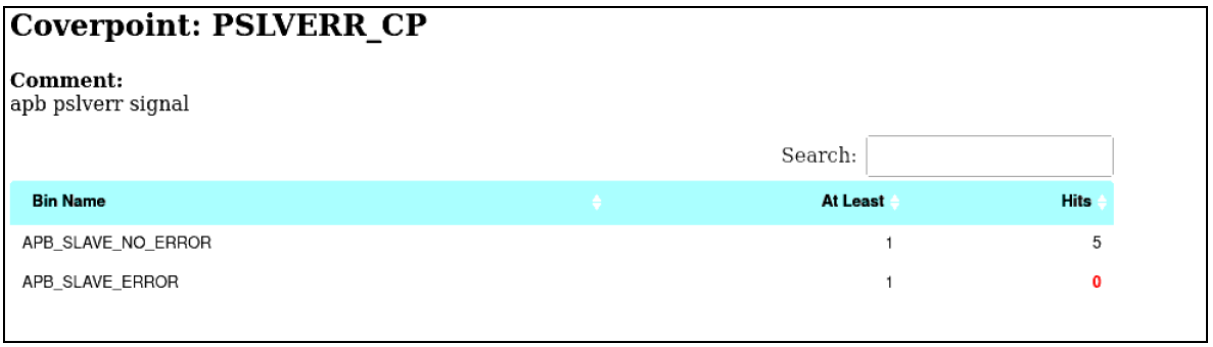
Fig 7.9.3: HTML window showing all coverage

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



CoverPoints	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
PADDR_CP	1	1	0	100.00%	100.00%	100.00%
PPROT_CP	8	5	3	62.50%	62.50%	62.50%
PRDATA_CP	1	1	0	100.00%	100.00%	100.00%
PSEL_CP	1	1	0	100.00%	100.00%	100.00%
PSLVERR_CP	2	1	1	50.00%	50.00%	50.00%
PSTRB_CP	16	4	12	25.00%	25.00%	25.00%
PWDATA_CP	1	0	1	0.00%	0.00%	0.00%
PWRITE_CP	2	1	1	50.00%	50.00%	50.00%

Fig 7.9.4: All coverpoints present in the Covergroup



Coverpoint: PSLVERR_CP

Comment:
apb pslverr signal

Search:

Bin Name	At Least	Hits
APB_SLAVE_NO_ERROR	1	5
APB_SLAVE_ERROR	1	0

Figure 7.9.5: Individual Coverpoint Hit

7.10 Errors

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

- The 2nd error was the declaration of the handle which is declared to access the sample.

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

Chapter 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.

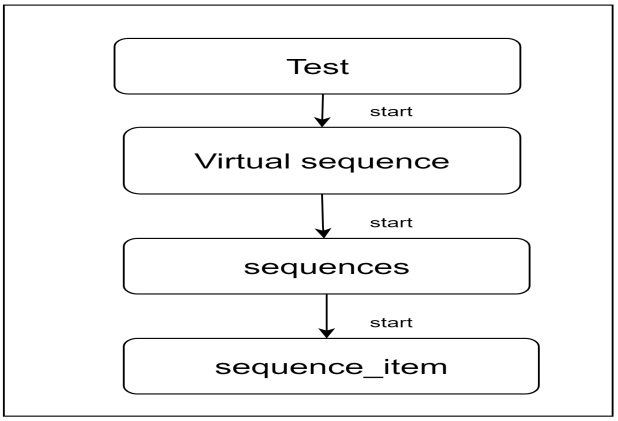


Fig 8.1 Test flow

8.2 APB Test Cases FlowChart

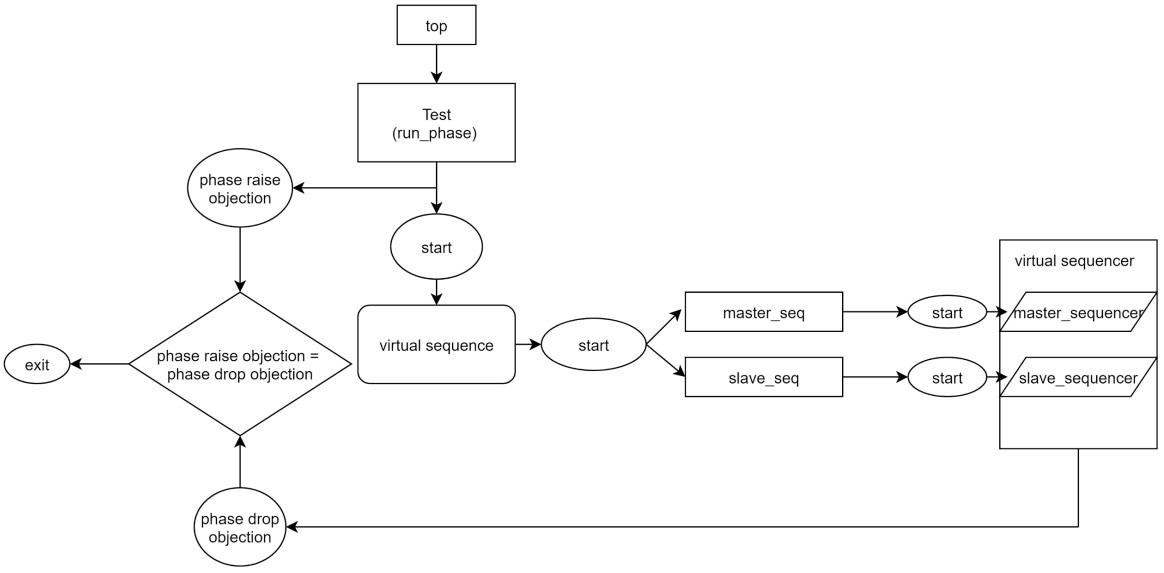


Fig 8.2: APB test cases flow chart

8.3 Transaction

Variables	Type	Description
pselx	bit	Master asserts the pselx to select the slave device
pwrite	enum	Pwrite signal decides whether write data transfer happens from the master side or read data transfer happens to the master.
paddr	bit	Address. This is the APB address bus. It can be up to 32 bits wide and is a data access or an instruction access.
pprot	enum	Protection type. This signal indicates the normal, privileged, or secure protection level of the transaction and whether the transaction is a data access or an instruction access.
penable	bit	Enable.This signal indicates the second and subsequent cycle of an APB transfer.
pwdata	bit	Write data. This bus is driven by the peripheral bus bridge unit during the write cycle when pwrite is HIGH. This bus can be up to 32 bits wide.
pstrb	enum	Write strobes. This signal indicates when byte lanes to update during a write transfer. There is one write strobe for each eight bits of the write data bus. Therefore, pstrb[n] corresponds to pwdata[(8n+7):(8n)]. Write strobes must not be active during a read transfer.
pready	bit	Ready. The Slave uses this signal to extend an APB transfer.
prdata	bit	Read Data.The selected slave drives this bus during read cycles when pwrite is LOW. This bus can be up to 32-bits wide.
pslverr	enum	This signal indicates a transfer failure. APB peripherals are not required to support the pslverr pin. This is true for both existing and new APB peripheral designs. Where a peripheral does not include this PIN then the appropriate input to the APB bridge is tied LOW.

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 (pselx, paddr, pwrite, pwdata, pready pslverr, pprot, pstrobe).
- ➢ Constraint declared for slave select and data transfer based on transfer size.

```
constraint pselx_c1  { $countones(pselx) == 1; }

constraint pselx_c2 { pselx >0 && pselx < 2**NO_OF_SLAVES; }

constraint pwdata_c3 { soft pwdata inside {[0:100]}; }

//This constraint is used to decide the pwdata size based om transfer size
constraint transfer_size_c4 {if(transfer_size == BIT_8)
                               $countones (pstrb) == 1;
                             else if(transfer_size == BIT_16)
                               $countones (pstrb) == 2;
                             else if(transfer_size == BIT_24)
                               $countones (pstrb) == 3;
                             else
                               $countones (pstrb) == 4;
                            }
```

Fig 8.3: Constraint for pselx and transfer_size

Table 8.3.2 Describing constraint for pselx and transfer size

Constraint	Description
pselx_c1	Declaring constraint for to select one slave at a time
pselx_c2	Declaring constraint for pselx should be in specified range
transfer_size_c4	This constraint is used to decide the pwdata based on the transfer size. (whether it is 8bit, 16bit, 24bit etc..)

➢ 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 master tx signals.

```
function bit apb_master_tx::do_compare (uvm_object rhs, uvm_comparer comparer);
  apb_master_tx apb_master_tx_compare_obj;

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

  return super.do_compare(apb_master_tx_compare_obj, comparer) &&
  paddr   == apb_master_tx_compare_obj.paddr &&
  pprot   == apb_master_tx_compare_obj.pprot &&
  pselx   == apb_master_tx_compare_obj.pselx &&
  pwrite  == apb_master_tx_compare_obj.pwrite &&
  pwdata  == apb_master_tx_compare_obj.pwdata &&
  pstrb   == apb_master_tx_compare_obj.pstrb &&
  prdata  == apb_master_tx_compare_obj.prdata &&
  pslverr == apb_master_tx_compare_obj.pslverr;

endfunction : do_compare
```

Fig 8.4: do_compare method

```
function void apb_master_tx::do_copy (uvm_object rhs);
  apb_master_tx apb_master_tx_copy_obj;

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

  paddr   = apb_master_tx_copy_obj.paddr;
  pprot   = apb_master_tx_copy_obj.pprot;
  pselx   = apb_master_tx_copy_obj.pselx;
  pwrite  = apb_master_tx_copy_obj.pwrite;
  pwdata  = apb_master_tx_copy_obj.pwdata;
  pstrb   = apb_master_tx_copy_obj.pstrb;
  prdata  = apb_master_tx_copy_obj.prdata;
  pslverr = apb_master_tx_copy_obj.pslverr;

endfunction : do_copy
```

Fig 8.5: do_copy method

```
function void apb_master_tx::do_print(uvm_printer printer);
  //super.do_print(printer);
  printer.print_string ("pselx",    pselx.name());
  printer.print_field  ("paddr",    paddr,       $bits(paddr),   UVM_HEX);
  printer.print_string ("pwrite",   pwrite.name());
  printer.print_field  ("pwdata",   pwdata,      $bits(pwdata),  UVM_HEX);
  printer.print_string ("transfer_size",transfer_size.name());
  printer.print_field  ("pstrb",    pstrb,       4,   UVM_BIN);
  printer.print_string ("pprot",    pprot.name());
  printer.print_field  ("prdata",   prdata,      $bits(prdata),  UVM_HEX);
  printer.print_string ("pslverr",  pslverr.name());
  printer.print_field  ("no_of_wait_states_detected", no_of_wait_states_detected, $bits(no_of_wait_states_detected), UVM_DEC);
  endfunction : do_print
```

Fig 8.6: do_print method

8.3.2 Slave_tx

- ➢ Slave_tx class is extended from the uvm_sequence_item holds the data items required to drive stimulus to dut
- ➢ Declared all the variables (pselx, paddr, pwrite, pwdata, pready, pslverr, pprot, pstrobe)

8.4 Sequences

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

8.4.1 Methods

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

Table 8.1. Sequence methods

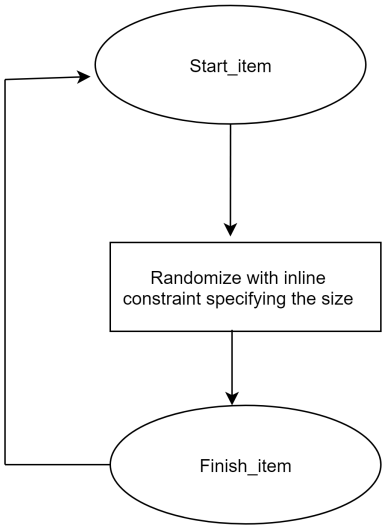


Fig 8.7 : Flow chart for sequence methods

Sections	Master sequences	Slave sequences	Description
base_seq	apb_base_master_seq	apb_base_slave_seq	Base class is extended from uvm_ sequence and parameterized with transaction (master_ tx, slave_ tx)
Data transfers	apb_8b_write_master_seq	apb_8b_write_slave_seq	Extended from base sequence. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomizing the req with the transfer size is BIT_8 and selecting number of slaves
	apb_16b_write_master_seq	apb_16b_write_slave_seq	Extended from base sequence. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomizing the req with the transfer size is BIT_16 and selecting number of slaves
	apb_24b_write_master_seq	apb_24b_write_slave_seq	Extended from base sequence. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomizing the req with the transfer size is BIT_24 and selecting number of slaves

Table 8.2. Describing master and slave sequences

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

```
task apb_master_8b_write_seq::body();
  super.body();
  req=apb_master_tx::type_id::create("req");
  req.apb_master_agent_cfg_h = p_sequencer.apb_master_agent_cfg_h;
  start_item(req);
  if(!req.randomize() with {req.pselx == SLAVE_0;
                            req.transfer_size == BIT_8;
                            req.pwrite == WRITE;}) begin
    `uvm_fatal("APB","Rand failed");
  end
  //req.print();
  finish_item(req);
```

Fig 8.8: Master seq body method

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

```
task apb_slave_8b_write_seq::body();
  req=apb_slave_tx::type_id::create("req");
  start_item(req);
  if(!req.randomize())
  begin
    `uvm_error(get_type_name(),"randomization failed");
  end
  req.print();
  finish_item(req);
endtask : body
```

Fig 8.9: Slave seq body method

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 apb_virtual_base_seq extends uvm_sequence;
  `uvm_object_utils(apb_virtual_base_seq)

  //Declaring p_sequencer
  `uvm_declare_p_sequencer(apb_virtual_sequencer)

    apb_master_sequencer apb_master_seqr_h;

    apb_slave_sequencer apb_slave_seqr_h;

    extern function new(string name = "apb_virtual_base_seq");
  extern task body();
endclass : apb_virtual_base_seq
```

Fig 8.10: Virtual base sequence

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

```
task apb_virtual_base_seq::body();
  if(!$cast(p_sequencer,m_sequencer))begin
    `uvm_error(get_full_name(),"Virtual sequencer pointer cast failed")
  end
  apb_slave_seqr_h  = p_sequencer.apb_slave_seqr_h;
  apb_master_seqr_h = p_sequencer.apb_master_seqr_h;
endtask
```

Fig 8.11: Virtual base sequence body

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

```
task apb_virtual_8b_write_seq::body();
  super.body();
  apb_master_8b_seq_h=apb_master_8b_write_seq::type_id::create("apb_master_8b_seq_h");
  apb_slave_8b_seq_h=apb_slave_8b_seq::type_id::create("apb_slave_8b_seq_h");
   fork
    forever begin
      apb_slave_8b_seq_h.start(p_sequencer.apb_slave_seqr_h);
    end
  join_none

  repeat(5) begin
    apb_master_8b_seq_h.start(p_sequencer.apb_master_seqr_h);
  end
 endtask : body
```

Fig 8.12:Virtual 8bit sequence body

Sections	Virtual sequences	Description
Data transfer	apb_virtual_8b_write_seq	Inside the 8bit virtual sequence, extending from base class. Declaring handles of sequences and inside body method constructing handles of sequence. Configuring the master and slave sequencers.
	apb_virtual_16b_write_seq	Inside the 16bit virtual sequence, extending from base class. Declaring handles of sequences and inside body method constructing handles of sequence. Configuring the master and slave sequencers.

	apb_virtual_24b_write_seq	Inside the 24bit virtual sequence, extending from base class. Declaring handles of sequences and inside body method constructing handles of sequence. Configuring the master and slave sequencers.

Table 8.3. Describing virtual sequences

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 apb_base_test extends uvm_test;
  `uvm_component_utils(apb_base_test)

   apb_env apb_env_h;

   apb_env_config apb_env_cfg_h;

  extern function new(string name = "apb_base_test", uvm_component parent = null);
  extern virtual function void build_phase(uvm_phase phase);
  extern virtual function void setup_apb_env_config();
  extern virtual function void setup_apb_master_agent_config();
  extern virtual function void setup_apb_slave_agent_config();
  extern virtual function void end_of_elaboration_phase(uvm_phase phase);
  extern virtual task run_phase(uvm_phase phase);

endclass : apb_base_test
```

Fig 8.13: 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.

```
function void apb_base_test::setup_apb_env_config();
  apb_env_cfg_h = apb_env_config::type_id::create("apb_env_cfg_h");
  apb_env_cfg_h.no_of_slaves      = NO_OF_SLAVES;
  apb_env_cfg_h.has_scoreboard    = 1;
  apb_env_cfg_h.has_virtual_seqr  = 1;

  //setting up the configuration for master agent
  setup_apb_master_agent_config();
  //Setting the master agent configuration into config_db
  uvm_config_db#(apb_master_agent_config)::set(this,"*master_agent*","apb_master_agent_config",apb_env_cfg_h.apb_master_agent_cfg_h);
  //Displaying the master agent configuration
  `uvm_info(get_type_name(),$sformatf("\nAPB_MASTER_AGENT_CONFIG\n%s",apb_env_cfg_h.apb_master_agent_cfg_h.sprint()),UVM_LOW);

  setup_apb_slave_agent_config();

  uvm_config_db#(apb_env_config)::set(this,"*","apb_env_config",apb_env_cfg_h);
  `uvm_info(get_type_name(),$sformatf("\nAPB_ENV_CONFIG\n%s",apb_env_cfg_h.sprint()),UVM_LOW);

endfunction : setup_apb_env_config
```

Fig 8.14: Setup env_cfg

D) In setup_master_agent_config function, master_agent_config class handle which is in env_config class has been constructed with the help of this handle all the required fields(has_coverage,no.of slaves,is_active,min and max address) in master_agent_config class has been setup.

```
function void apb_base_test::setup_apb_master_agent_config();
  bit [63:0]local_min_address;
  bit [63:0]local_max_address;
  apb_env_cfg_h.apb_master_agent_cfg_h = apb_master_agent_config::type_id::create("apb_master_agent_config");
  if(MASTER_AGENT_ACTIVE === 1) begin
    apb_env_cfg_h.apb_master_agent_cfg_h.is_active    = uvm_active_passive_enum'(UVM_ACTIVE);
  end
  else begin
    apb_env_cfg_h.apb_master_agent_cfg_h.is_active    = uvm_active_passive_enum'(UVM_PASSIVE);
  end
  apb_env_cfg_h.apb_master_agent_cfg_h.no_of_slaves   = NO_OF_SLAVES;
  apb_env_cfg_h.apb_master_agent_cfg_h.has_coverage   = 1;
  for(int i =0; i<NO_OF_SLAVES; i++) begin

    apb_env_cfg_h.apb_master_agent_cfg_h.master_min_addr_range(i,local_max_address + 2**SLAVE_MEMORY_GAP);
    local_min_address = apb_env_cfg_h.apb_master_agent_cfg_h.master_min_addr_range_array[i];
    apb_env_cfg_h.apb_master_agent_cfg_h.master_max_addr_range(i,local_max_address+ 2**(SLAVE_MEMORY_SIZE)-1 + 2**SLAVE_MEMORY_GAP);
    local_max_address = apb_env_cfg_h.apb_master_agent_cfg_h.master_max_addr_range_array[i];

  end
endfunction : setup_apb_master_agent_config
```

Fig 8.15: Master_agent_cfg setup

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

```
function void apb_base_test::setup_apb_slave_agent_config();
  apb_env_cfg_h.apb_slave_agent_cfg_h = new[apb_env_cfg_h.no_of_slaves];
  foreach(apb_env_cfg_h.apb_slave_agent_cfg_h[i]) begin
    apb_env_cfg_h.apb_slave_agent_cfg_h[i] = apb_slave_agent_config::type_id::create($sformatf("apb_slave_agent_config[%0d]",i));
    apb_env_cfg_h.apb_slave_agent_cfg_h[i].slave_id       = i;
    apb_env_cfg_h.apb_slave_agent_cfg_h[i].slave_selected = 0;
    apb_env_cfg_h.apb_slave_agent_cfg_h[i].min_address    = apb_env_cfg_h.apb_master_agent_cfg_h.master_min_addr_range_array[i];
    apb_env_cfg_h.apb_slave_agent_cfg_h[i].max_address    = apb_env_cfg_h.apb_master_agent_cfg_h.master_max_addr_range_array[i];
    if(SLAVE_AGENT_ACTIVE === 1) begin
      apb_env_cfg_h.apb_slave_agent_cfg_h[i].is_active  = uvm_active_passive_enum'(UVM_ACTIVE);
    end
    else begin
      apb_env_cfg_h.apb_slave_agent_cfg_h[i].is_active  = uvm_active_passive_enum'(UVM_PASSIVE);
    end
    apb_env_cfg_h.apb_slave_agent_cfg_h[i].has_coverage = 1;
    uvm_config_db #(apb_slave_agent_config)::set(this,$sformatf("*env*"),$sformatf("apb_slave_agent_config[%0d]",i),apb_env_cfg_h.apb_slave_agent_cfg_h[i]
);
   `uvm_info(get_type_name(),$sformatf("\nAPB_SLAVE_CONFIG[%0d]\n%s",i,apb_env_cfg_h.apb_slave_agent_cfg_h[i].sprint()),UVM_LOW);
  end
endfunction : setup_apb_slave_agent_config
```

Fig 8.16: Slave_agent_cfg setup

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

```
class apb_8b_write_test extends apb_base_test;
  `uvm_component_utils(apb_8b_write_test)

  //Variable : apb_virtual_8b_seq_h'
  //Instatiation of apb_virtual_8b_seq
  apb_virtual_8b_write_seq apb_virtual_8b_seq_h;

  //-------------------------------------------------------
  // Externally defined Tasks and Functions
  //-------------------------------------------------------
  extern function new(string name = "apb_8b_write_test", uvm_component parent = null);
  extern virtual task run_phase(uvm_phase phase);

endclass : apb_8b_write_test
```

Fig 8.17: Example for 8bit test

```
task apb_8b_write_test::run_phase(uvm_phase phase);

  apb_virtual_8b_seq_h = apb_virtual_8b_write_seq::type_id::create("apb_virtual_8b_seq_h");
  `uvm_info(get_type_name(),$sformatf("apb_8b_write_test"),UVM_LOW);
  phase.raise_objection(this);
    apb_virtual_8b_seq_h.start(apb_env_h.apb_virtual_seqr_h);
  phase.drop_objection(this);

endtask : run_phase
```

Fig 8.18 :Run_phase of 8bit_test

sections	Test Names	Description
Data transfers	apb_8b_write_test	Extend test from base test and declare virtual sequence then create virtual sequence in run_phase, and start the virtual sequence with virtual sequencer handle in phase, raise and drop objections. The wdata contains 8 bit data
	apb_16b_write_test	Extend test from base test and declare virtual sequence then create virtual sequence handle in run_phase, and start the virtual sequence with virtual sequencer handle in phase, raise and drop objection.The wdata contains 16 bit data
	apb_24b_write_test	Extend test from base test and declare virtual sequence then create virtual sequence handle in run_phase, and start the virtual sequence with virtual sequencer handle in phase, raise and drop objection.The wdata contains 24 bit data

8.7 Testlists

Regression list for APB

TestCase Names	Description
#Directed test cases	
apb_8b_write_test	Checking for 8 bit data transfer
apb_16b_write_test	Checking for 16bit data transfer
apb_24b_write_test	Checking for 24bit data transfer
apb_8b_read_test	Checking for 8b read test

Table 8.4. Testlists

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.
APB AVIP USER GUIDE

Chapter 10

References

ABP Accelerated VIP APB4 Document-
https://developer.arm.com/documentation/ihi0024/c/