

AHB-AVIP

Table of Contents

Table of Contents	1
List of Tables :	5
List of Abbreviations :	6
Chapter 1	7
Introduction	7
1.1 Key features	7
Chapter 2	8
Architecture	8
2.1 AHB AVIP Testbench Architecture	8
Chapter 3	10
Implementation	10
3.1 Pin Interface	10
3.2 Testbench Components	12
Chapter 4	35
Directory Structure	35
4.1 Package Content	35
Chapter 5	38
Configuration	38
5.1 Global package variables	38
5.2 Master agent configuration	40
5.3 Slave agent configuration	40
5.4 Environment configuration	40
Chapter 6	42
Verification Plan	42
6.1 Verification plan	42
6.2 Template of Verification Plan	42
6.3 Sections for different test Scenarios	43
Chapter 7	44
Assertion Plan	44
7.1 Assertion Plan overview	44
7.2 Template of Assertion Plan	45

7.3 Master Assertion Condition	45
7.4 Slave Assertion Condition	47
7.4.1 checkSlaveHrdataValid	47
Chapter 8	49
Coverage Plan	49
8.1 Template of Coverage Plan	49
8.2 Functional Coverage	49
8.3 Uvm_Subscriber	49
8.4 Covergroup	51
8.4 Bucket	52
8.5 Coverpoints	53
8.6 Cross Coverpoints	53
8.7 Creation of the covergroup	54
8.8 Sampling of the covergroup	54
8.9 Checking for the coverage	54
Chapter 9	58
Test Cases	58
9.1 Test Flow	58
9.2 AHB Test Cases FlowChart	58
9.3 Transaction	59
9.4 Sequences	65
9.5 Virtual sequences	68
9.6 Test Cases	71
Chapter 10	76
Simulation Results and Waveform	76
Chapter 11	77
References	77

List of Figures :

Fig 2.1 AHP_AVIP Architecture.....	8
Fig 3. 1 HDL Top	13
Fig 3.2 AHB driver bfm instantiation in AHB master agent bfm code snippet.....	14
Fig 3.3 AHB monitor bfm instantiation in AHB master agent bfm code snippet.....	14
Fig 3.4 AHB slave driver bfm instantiation in AHB slave agent bfm code snippet.....	15
Fig 3.5 AHB slave monitor bfm instantiation in AHB slave agent bfm code snippet.....	16
Fig 3.6 HVL Top	17
Fig 3.7 Connection of the analysis ports of the monitor to the scoreboard analysis fifo.....	18
Fig 3. 8 Shows the declaration of slave and master analysis port in the slave and master monitor proxy	19
Fig 3.9 Shows the declaration of master and slave analysis fifo in the scoreboard.....	19
Fig 3.10 Creation of the master and slave analysis port	19
Fig 3.11 Connection done between the analysis port and analysis fifo export in the env class	20
Fig 3.12 Use of get method to get the packet from monitor analysis port.....	20
Fig 3.13 The comparison of the master hwdata with slave hwdata	20
Fig 3.14 Flow chart of the scoreboard run phase.....	21
Fig 3.15 Flow chart of the scoreboard report phase.....	22
Fig 3.16 AHB master agent build phase code snippet.....	23
Fig 3.17 AHB master agent connect phase code snippet.....	24
Fig 3.18 Flowchart of communication between ahb master driver proxy and ahb master driver bfm	25
Fig 3.19 run phase of ahb master driver proxy code snippet	26
Fig 3.20 Flowchart of ahb master monitor proxy and ahb master monitor bfm communication	27
Fig 3.21 AHB slave agent build phase code snippet	28
Fig 3.22 AHB slave agent connect phase code snippet	28
Fig 3.23 Flowchart of ahb slave driver bfm and slave driver proxy communication	30
Fig 3.24 AHB slave driver proxy run phase code snippet	31
Fig 3.25 Flowchart of ahb slave monitor bfm and slave monitor proxy communication.....	32
Fig 3.26 run phase of ahb slave monitor proxy code snippet	32
Fig 4.1 Package Structure of AHB_AVIP	35
Fig 5.1 Verification plan template	42
Fig 7.1 checkHaddrAlignment Assertion	45
Fig 7.2 checkStrobe Assertion.....	46
Fig 7.3 checkHrespOKayForValid Assertion.....	47
Fig 7.4 checkSlaveHrdataValid Assertion.....	47
Fig 8. 1 uvm_subscriber	50
Fig 8.2 Monitor and coverage connection	50
Fig 8.3 Write function.....	51
Fig 8.4 Covergroup.....	51
Fig 8.5 option.comment.....	52

Fig 8.6 Bucket.....	52
Fig 8.7 Coverpoint.....	53
Fig 8. 8 Cross Coverpoints	53
Fig 8.9 Illegal bins	53
Fig 8.10 Creation of covergroup.....	54
Fig 8.11 Sampling of the covergroup	54
Fig 8.12 Simulation log file path	54
Fig 8.13 Coverage report path	55
Fig 8.14 HTML window showing all coverage.....	55
Fig 8.15 All coverpoints present in the Master Covergroup.....	56
Fig 8.16 All coverpoints present in the Slave Covergroup.....	56
Fig 8.17 Individual Coverpoint Hit.....	57
Fig 9.1 Test flow.....	58
Fig 9.2 AHB test cases flow chart	58
Fig 9.3 Constraints of Ahb Master transaction	61
Fig 9.4 do_compare method of Master Transaction	62
Fig 9.5 do_copy method of Master Transaction	62
Fig 9.6 Constraints of Ahb Slave transaction	63
Fig 9.7 do_compare method of Slave Transaction	64
Fig 9.8 do_copy method of Slave Transaction	64
Fig 9.9 Flow chart for sequence methods	65
Fig 9.10 Master sequence body method	66
Fig 9.11 Constraints Of Master Sequence	67
Fig 9.12 Slave sequence body method.....	68
Fig 9.13 Virtual base sequence	68
Fig 9.14 Virtual base sequence body	69
Fig 9.15 Virtual Single Write sequence body	69
Fig 9.16 Base test.....	71
Fig 9.17 Setup Environment Config	72
Fig 9.18 Master Agent Config setup.....	72
Fig 9.19 Slave Agent Config setup.....	73
Fig 9.20 Example for Single Write test	73
Fig 9.21 Run phase of Single Write test	74

List of Tables :

Table 1: AHB pins used to interface to external devices	10
Table 2: UVM verbosity Priorities	33
Table 3: Descriptions of each Verbosity level	33
Table 4: Directory Path	36
Table 5: Global package variables	38
Table 6: AhbMasterAgentConfig.....	40
Table 7: AhbSlaveAgentConfig	40
Table 8: AhbEnvironmentConfig.....	40
Table 9: Checking coverage closure for the different transactions and burst transfers.....	43
Table 10: Transaction Signals	59
Table 11: Describing constraints of AhbMasterTransaction.....	61
Table 12: Constraints of Ahb Slave transaction.....	63
Table 13: Sequence methods.....	65
Table 14: Describing master and slave sequences	66
Table 15: Describing virtual sequences	69
Table 16: Tests	74
Table 17: Testlists	75

List of Abbreviations :

Abbreviation	Description
uvm	universal verification methodology
ahb	advanced high performance
avip	accelerated verification intellectual property
hdl	hardware descriptive language
hvl	hardware verification language
bfm	bus functional model
tlm	transaction level modelling
hclk	system clock

Chapter 1

Introduction

The Advanced High-Performance Bus (AHB) is part of the Advanced Microcontroller Bus Architecture (AMBA) protocol family developed by ARM. It defines a high-performance bus interface that is designed for efficient communication and high-speed data transfer between system components. AHB is used to connect high-bandwidth, high-performance components like CPUs, memory controllers, and DMA controllers.

1.1 Key features

- Supports 32-bit address bus.
- Pipelined structure for high-performance data transfers.
- Burst transfers for efficient block data movement.
- Programmable wait states for slower slaves.
- Error reporting using HRESP signal.
- Separate address and data phases for improved efficiency.
- Supports large data widths (32, 64 bits or more).
- Suitable for high-speed peripherals like GPUs, memory, and accelerators.

Architecture

2.1 AHB AVIP Testbench Architecture

The **Accelerated VIP (AVIP)** for AHB is structured into two distinct top modules: **HVL TOP** and **HDL TOP**, as depicted in Figure 2.1. This division separates the synthesizable and un-synthesizable components of the testbench, optimizing its functionality for both simulation and emulation modes.

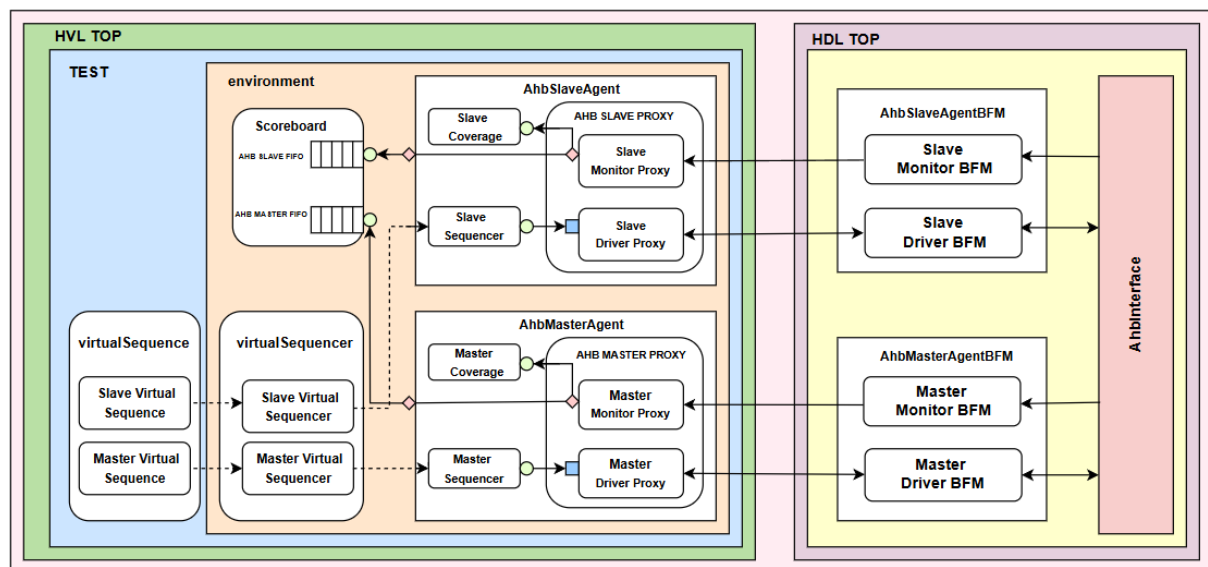


Fig 2.1 AHP_AVIP Architecture

The **HDL TOP** contains the synthesizable components, including the interface, clock, and reset signal generation, allowing it to run efficiently in emulators. It houses the Bus Functional Models (BFMs), which consist of drivers and monitors with back-pointers to their proxies, enabling non-blocking method calls defined in the HVL.

In contrast, the **HVL TOP** includes the un-synthesizable and untimed components, handling transaction flow from the master and slave virtual sequences through the BFM Proxy and BFM to the AHB interface. Data collected by the monitor BFMs is passed to the scoreboard for checking and coverage collection, ensuring robust verification.

Communication between the two modules is transaction-based, enabling the exchange of information-rich transactions. Tasks and functions within the HDL TOP interact seamlessly with their proxies in the HVL TOP, facilitating efficient data transfer.

The clock generation within the HDL TOP inside the emulator ensures the emulator operates at full speed, enabling faster execution of longer tests. This modular approach provides a clear separation of concerns, ensuring scalability and efficiency in verifying AHB designs across simulation and emulation platforms.

Chapter 3

Implementation

3.1 Pin Interface

Table 1: AHB pins used to interface to external devices

Signal Name	Source	Width	Description
hclk	Clock source	1	Bus clock that times all transfers. All timings are related to its rising edge.
hresetn	Reset controller	1	Active LOW reset signal for the system and the bus.
haddr	Master	ADDR_WIDTH	Byte address of the transfer. ADDR_WIDTH is recommended between 10 and 64.
hburst	Master	HBURST_WIDTH	Indicates burst transfer count and address increment type. HBURST_WIDTH must be 0 or 3.
hmastlock	Master	1	Indicates if transfer is part of a locked sequence.
hprot	Master	HPROT_WIDTH	Protection control signal providing access type information.

hsize	Master	3	Indicates the size of the transfer.
hnonsec	Master	1	Indicates if transfer is Non-secure or Secure.
hexcl	Master	1	Indicates if the transfer is part of an Exclusive Access sequence.
hmaster	Master	HMASTER_WIDTH	Master identifier. Modified by interconnect for unique identification during Exclusive Transfers.
htrans	Master	2	Indicates transfer type (IDLE, BUSY, NONSEQUENTIAL, SEQUENTIAL).
hwdata	Master	DATA_WIDTH	Transfers data from Master to Slave during writes. Recommended range: 32 to 256 bits.
hwstrb	Master	DATA_WIDTH/8	Write strobes indicating valid data lanes.
hwrite	Master	1	Indicates transfer direction: HIGH for write, LOW for read.
hrdata	Slave	DATA_WIDTH	Transfers data from Slave to Master during reads.
hreadyout	Slave	1	Indicates if the transfer on the bus is finished.
hresp	Slave	1	Indicates transfer status: LOW (OKAY) or HIGH (ERROR).

hexokay	Slave	1	Indicates success or failure of an Exclusive Transfer.
hselx	Master	1	Indicates selection of the current Slave for the transfer.
hready	Mux/Slave	1	Indicates to Master and Slaves that the previous transfer is complete.
hresp	Slave	1	Selected transfer response from the Slave.
hexokay	Slave	1	Selected Exclusive Transfer status from the Slave.

3.2 Testbench Components

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

3.2.1 AHB HDL Top

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

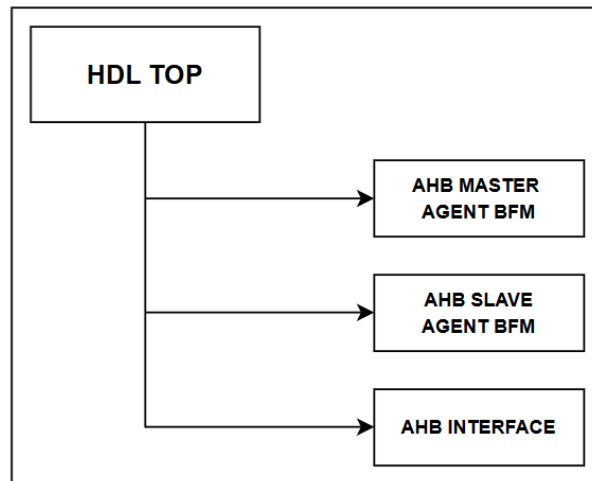


Fig 3. 1 HDL Top

3.2.2 AHB Interface

Importing the global packages

Passing Signals: hclk, hresetn

Declaration of signals: haddr, hselx, hburst, hmastlock, hprot, hsize, hnonsec, hexcl, hmaster, htrans, hwdata, hwstrb, hwrite, hrdata, hreadyout, hresp, hexokay, hready declared as logic type.

3.2.3 AHB Master Agent BFM Module

Instantiates the below two interfaces here

- a) ahb master driver bfm and
- b) ahb master monitor bfm.

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

```

AhbMasterDriverBFM ahbMasterDriverBFM (
    .hclk(ahbInterface.hclk),
    .hresetsn(ahbInterface.hresetsn),
    .haddr(ahbInterface.haddr),
    .hburst(ahbInterface.hburst),
    .hmastlock(ahbInterface.hmastlock),
    .hprot(ahbInterface.hprot),
    .hsize(ahbInterface.hsize),
    .hnonsec(ahbInterface.hnonsec),
    .hexcl(ahbInterface.hexcl),
    .hmaster(ahbInterface.hmaster),
    .htrans(ahbInterface.htrans),
    .hwdata(ahbInterface.hwdata),
    .hwstrb(ahbInterface.hwstrb),
    .hwrite(ahbInterface.hwrite),
    .hrdata(ahbInterface.hrdata),
    .hreadyout(ahbInterface.hreadyout),
    .hresp(ahbInterface.hresp),
    .hexokay(ahbInterface.hexokay),
    .hready(ahbInterface.hready),
    .hselx(ahbInterface.hselx)
);

```

Fig 3.2 AHB driver bfm instantiation in AHB master agent bfm code snippet

```

AhbMasterMonitorBFM ahbMasterMonitorBFM (
    .hclk(ahbInterface.hclk),
    .hresetsn(ahbInterface.hresetsn),
    .haddr(ahbInterface.haddr),
    .hburst(ahbInterface.hburst),
    .hmastlock(ahbInterface.hmastlock),
    .hprot(ahbInterface.hprot),
    .hsize(ahbInterface.hsize),
    .hnonsec(ahbInterface.hnonsec),
    .hexcl(ahbInterface.hexcl),
    .hmaster(ahbInterface.hmaster),
    .htrans(ahbInterface.htrans),
    .hwdata(ahbInterface.hwdata),
    .hwstrb(ahbInterface.hwstrb),
    .hwrite(ahbInterface.hwrite),
    .hrdata(ahbInterface.hrdata),
    .hreadyout(ahbInterface.hreadyout),
    .hresp(ahbInterface.hresp),
    .hexokay(ahbInterface.hexokay),
    .hready(ahbInterface.hready),
    .hselx(ahbInterface.hselx)
);

```

Fig 3.3 AHB monitor bfm instantiation in AHB master agent bfm code snippet

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

3.2.4 AHB Master Driver BFM Interface

Ahb master driver bfm is an interface where it will get the signals from the ahb interface. It has a method driveToBFM which will be called by the ahb master driver proxy which drives the haddr, hwdata, hwrite, hsize, hburst, htrans and other control signals to the ahb interface. fig.3.2 gives the reference of the instantiation of ahb master driver bfm.

3.2.5 AHB Master Monitor BFM Interface

Ahb master monitor bfm is an interface where it will get the signals from the ahb interface. It has a method sampleData which will be called by the ahb master monitor proxy which samples the haddr, hwrite, hsize, hburst, htrans, hmastlock, hready, hresp, hprot, hselx, hwstrb, hwdata and hrdata data from the ahb interface. After sampling the data, the ahb master monitor bfm interface sends the data to the ahb master monitor proxy using the output port of sampleData task. fig.3.3 gives the reference of the instantiation of ahb master monitor bfm.

3.2.6 AHB Slave Agent BFM Module

Instantiates the below two interfaces here

1. ahb slave driver bfm
2. ahb slave monitor bfm

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

```
AhbSlaveMonitorBFM ahbSlaveMonitorBFM(.hclk(ahbInterface.hclk),
                                         .hresetn(ahbInterface.hresetn),
                                         .hburst(ahbInterface.hburst),
                                         .hmastlock(ahbInterface.hmastlock),
                                         .haddr(ahbInterface.haddr),
                                         .hprot(ahbInterface.hprot),
                                         .hsize(ahbInterface.hsize),
                                         .hnonsec(ahbInterface.hnonsec),
                                         .hexcl(ahbInterface.hexcl),
                                         .hmaster(ahbInterface.hmaster),
                                         .htrans(ahbInterface.htrans),
                                         .hwdata(ahbInterface.hwdata),
                                         .hwstrb(ahbInterface.hwstrb),
                                         .hwrite(ahbInterface.hwrite),
                                         .hrdata(ahbInterface.hrdata),
                                         .hreadyout(ahbInterface.hreadyout),
                                         .hresp(ahbInterface.hresp),
                                         .hexokay(ahbInterface.hexokay),
                                         .hready(ahbInterface.hready),
                                         .hselx(ahbInterface.hselx)
);
```

Fig 3.4 AHB slave driver bfm instantiation in AHB slave agent bfm code snippet


```

AhbSlaveMonitorBFM ahbSlaveMonitorBFM(.hclk(ahbInterface.hclk),
                                        .hresetn(ahbInterface.hresetn),
                                        .hburst(ahbInterface.hburst),
                                        .hmastlock(ahbInterface.hmastlock),
                                        .haddr(ahbInterface.haddr),
                                        .hprot(ahbInterface.hprot),
                                        .hsize(ahbInterface.hsize),
                                        .hnonsec(ahbInterface.hnonsec),
                                        .hexcl(ahbInterface.hexcl),
                                        .hmaster(ahbInterface.hmaster),
                                        .htrans(ahbInterface.htrans),
                                        .hwdata(ahbInterface.hwdata),
                                        .hwstrb(ahbInterface.hwstrb),
                                        .hwrite(ahbInterface.hwrite),
                                        .hrdata(ahbInterface.hrdata),
                                        .hreadyout(ahbInterface.hreadyout),
                                        .hresp(ahbInterface.hresp),
                                        .hexokay(ahbInterface.hexokay),
                                        .hready(ahbInterface.hready),
                                        .hselx(ahbInterface.hselx)
);

```

Fig 3.5 AHB slave monitor bfm instantiation in AHB slave agent bfm code snippet

3.2.7 AHB Slave Driver BFM Interface

AHB slave driver bfm is an interface where it will get the signals from the ahb interface. It has a method `slaveDriveToBFM` which will be called by the ahb slave driver proxy which drives the `hready`, `hresp`, `hready` to the ahb interface. fig.3.4 gives the reference for the instantiation of ahb slave driver bfm.

3.2.8 AHB Slave Monitor BFM Interface

AHB slave monitor bfm is an interface where it will get the signals from the ahb interface. It has a method `slaveSampleData` which will be called by the ahb slave monitor proxy which samples the `hselx`, `haddr`, `hburst`, `hwrite`, `hsize`, `htrans`, `hnonsec`, `hprot`, `hresp`, `hreadyout`, `hwdata`, `hrdata`, `hwstrb` from the ahb interface. After sampling the data, the ahb slave monitor bfm interface sends the data to the ahb slave monitor proxy using the output port of `slaveSampleData` task. fig.3.5 gives the reference for the instantiation of ahb slave monitor bfm.

3.2.9 AHB

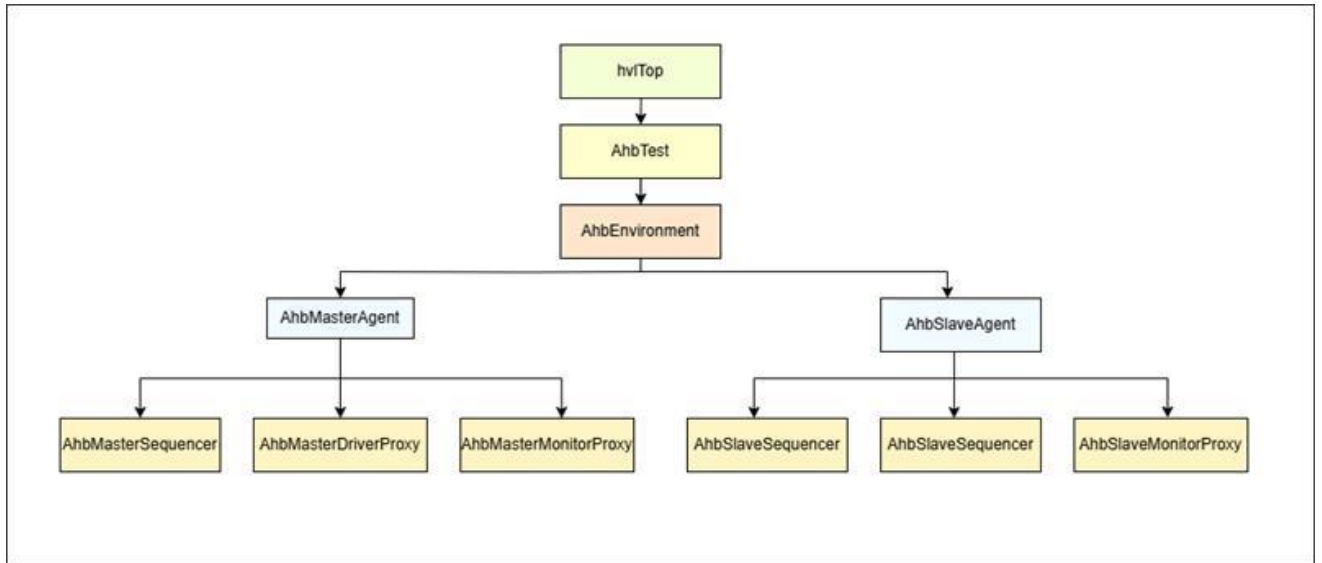


Fig 3.6 HVL Top

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

3.2.10 AHB Environment

Environment has the below components

- a. AhbScoreboard
- b. AhbVirtualSequencer
- c. AhbMasterAgent
- d. AhbSlaveAgent

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

In the connect phase, the **ahbMasterMonitorProxy** is connected to **ahbScoreboard** and **ahbSlaveMonitorProxy** to **ahbScoreboard** using analysis port and analysis fifo as shown in fig 3.7.

3.2.11 AHB 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 AHB-AVIP project is to

1. Compare the HWDATA, HADDR, HWRITE and HRDATA 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.

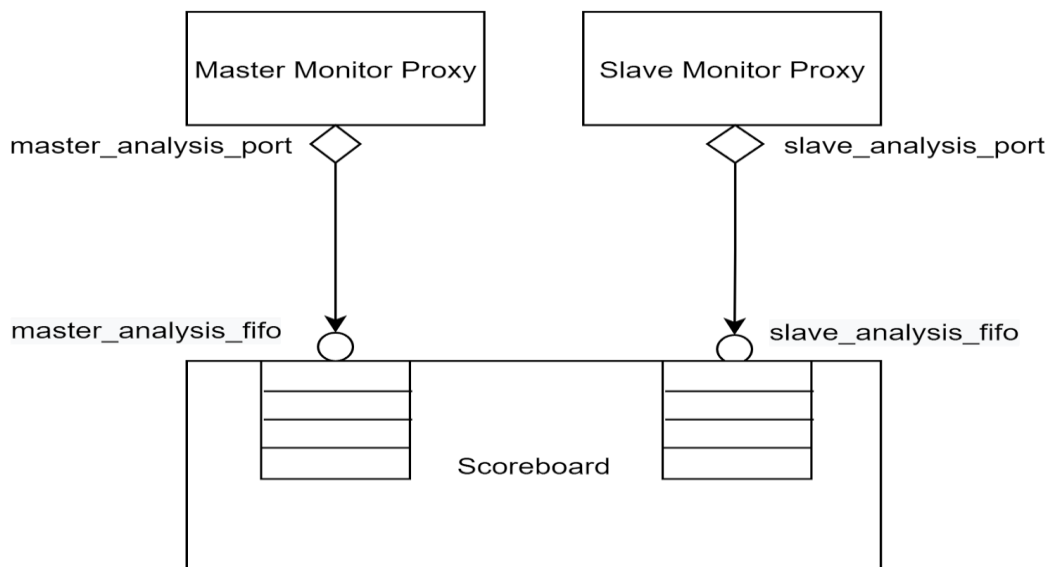


Fig 3.7 Connection of the analysis ports of the monitor to the scoreboard analysis fifo

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#(AhbSlaveTransaction) ahbSlaveAnalysisPort;
uvm_analysis_port#(AhbMasterTransaction) ahbMasterAnalysisPort;

```

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 fifo and slave analysis fifo in the scoreboard.

```

uvm_tlm_analysis_fifo#(AhbMasterTransaction) ahbMasterAnalysisFifo[];
uvm_tlm_analysis_fifo#(AhbSlaveTransaction) ahbSlaveAnalysisFifo[];

```

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 AhbScoreboard::new(string name = "AhbScoreboard",uvm_component parent = null);
    super.new(name, parent);
    ahbMasterAnalysisFifo = new[NO_OF_MASTERS];
    ahbSlaveAnalysisFifo = new[NO_OF_SLAVES];

    foreach(ahbMasterAnalysisFifo[i]) begin
        ahbMasterAnalysisFifo[i] = new($sformatf("ahbMasterAnalysisFifo[%0d]",i),this);
    end

    foreach(ahbSlaveAnalysisFifo[i]) begin
        ahbSlaveAnalysisFifo[i] = new($sformatf("ahbSlaveAnalysisFifo[%0d]",i),this);
    end
endfunction : new

```

Fig 3.10 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 in the connect phase of the env class.

```

foreach(ahbMasterAgent[i]) begin
    ahbMasterAgent[i].ahbMasterMonitorProxy.ahbMasterAnalysisPort.connect(ahbScoreboard.ahbMasterAnalysisFifo[i].analysis_export);
end

foreach(ahbSlaveAgent[i]) begin
    ahbSlaveAgent[i].ahbSlaveMonitorProxy.ahbSlaveAnalysisPort.connect(ahbScoreboard.ahbSlaveAnalysisFifo[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 AhbScoreboard::run_phase(uvm_phase phase);
    super.run_phase(phase);

    forever begin
        for(int j = 0; j < NO_OF_MASTERS; j++) begin
            ahbMasterAnalysisFifo[j].get(ahbMasterTransaction);
            ahbMasterTransactionCount++;
            `uvm_info(get_type_name(), $sformatf("after calling master's analysis fifo get method"), UVM_HIGH);
        end

        for(int i = 0; i < NO_OF_SLAVES; i++) begin
            ahbSlaveAnalysisFifo[i].get(ahbSlaveTransaction);
            ahbSlaveTransactionCount++;
            `uvm_info(get_type_name(), $sformatf("after calling slave's analysis fifo get method"), UVM_HIGH);
        end
    end
end

```

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

The Comparison of the haddr, hwrite, hwddata and hrdata from the master monitor and slave monitor is done in the run phase. Fig 3.13 shows the comparison of the master hwddata with slave hwddata.

```

if(ahbMasterTransaction.hwddata == ahbSlaveTransaction.hwddata) begin
    `uvm_info(get_type_name(), $sformatf("ahb HWDATA from master and slave is equal"), UVM_HIGH);
    `uvm_info("SB HWDATA MATCHED", $sformatf("Master HWDATA = %0p and Slave HWDATA = %0p",
        ahbMasterTransaction.hwddata, ahbSlaveTransaction.hwddata), UVM_HIGH);
    VerifiedMasterHwdataCount++;
end

else begin
    `uvm_info(get_type_name(), $sformatf("ahb HWDATA from master and slave is not equal"), UVM_HIGH);
    `uvm_error("ERROR SC HWDATA MISMATCH", $sformatf("Master HWDATA = %0p and Slave HWDATA = %0p",
        ahbMasterTransaction.hwddata, ahbSlaveTransaction.hwddata));
    FailedMasterHwdataCount++;
end
end

```

Fig 3.13 The comparison of the master hwddata with slave hwddata

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

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

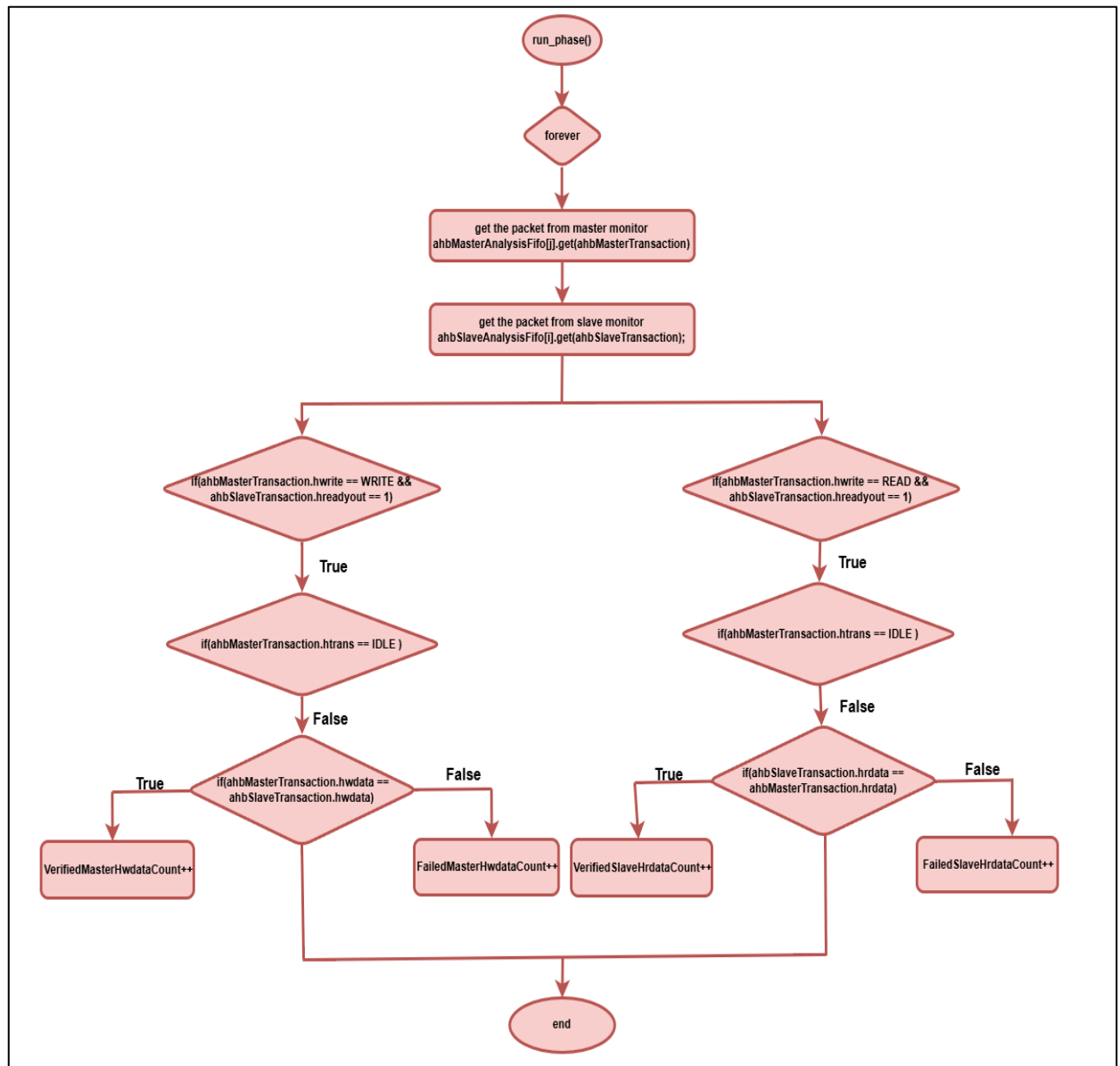


Fig 3.14 Flow chart of the scoreboard run phase

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 master packet is compared with the slave packet.

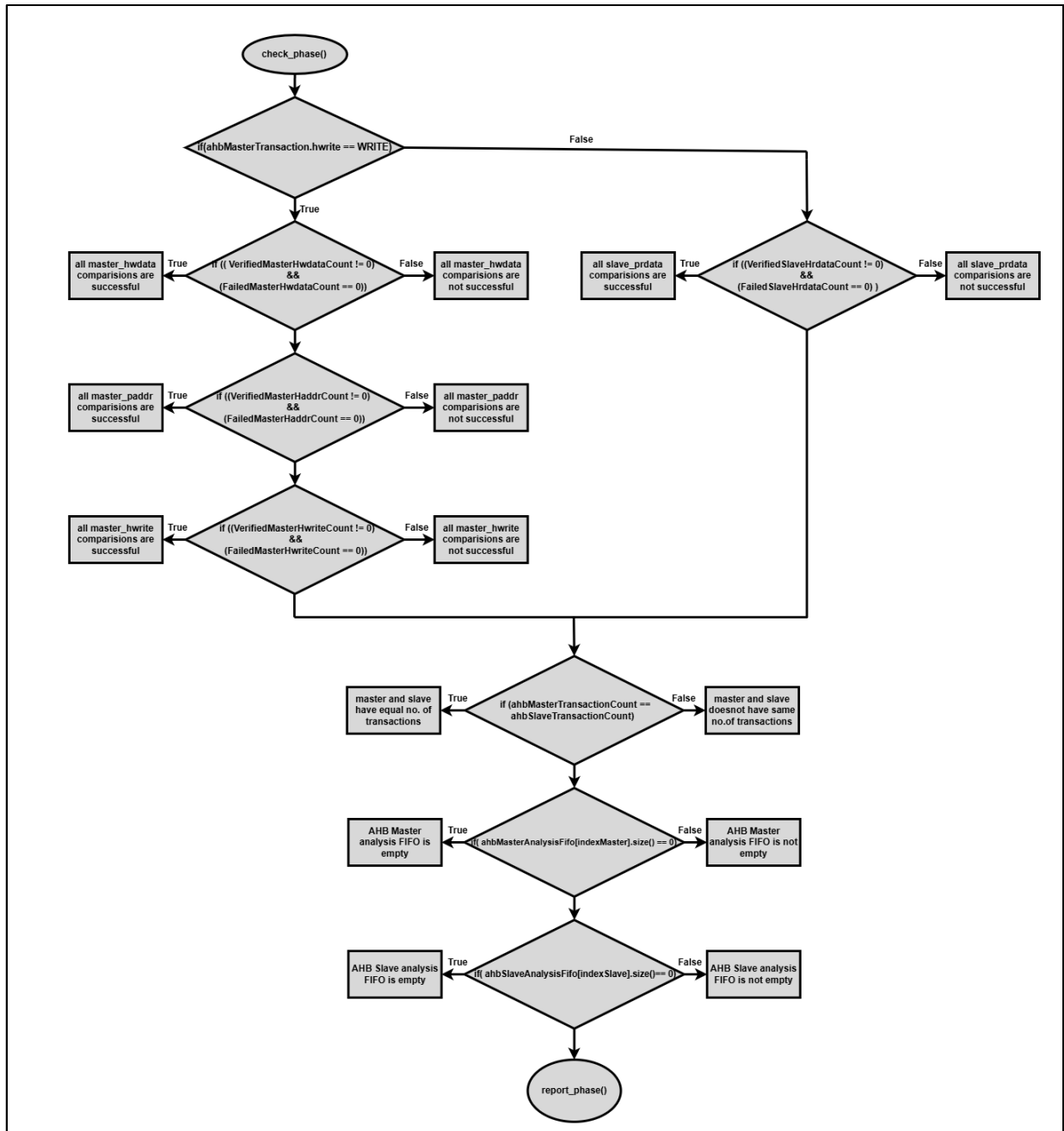


Fig 3.15 Flow chart of the scoreboard report phase

3.2.12 AHB Virtual Sequencer

It coordinates stimulus for the AHB Master and AHB Slave Sequencers. It declares their handles and initializes them in the build_phase.

3.2.13 AHB Master Agent

The AHB Master Agent is a UVM component that extends `uvm_agent`. It declares a handle for `AhbMasterAgentConfig`, which determines the creation and connection of components. The `AhbMasterSequencer` and `AhbMasterDriverProxy` are instantiated only if the agent is active, based on the `is_active` variable in `AhbMasterAgentConfig`. Additionally, the `AhbMasterCoverage` component is created in the build_phase if the `hasCoverage` variable is set to 1. Please refer to figure 3.16 for the `AhbMasterAgent` build_phase code snippet

The AHB Master Agent build phase involves the creation of the following components:

- a. `AhbMasterSequencer`
- b. `AhbMasterDriverProxy`
- c. `AhbMasterMonitorProxy`
- d. `AhbMasterCoverage`

```
function void AhbMasterAgent::build_phase(uvm_phase phase);
    super.build_phase(phase);
    if (ahbMasterAgentConfig.is_active == UVM_ACTIVE) begin
        ahbMasterSequencer = AhbMasterSequencer::type_id::create("ahbMasterSequencer", this);
        ahbMasterDriverProxy = AhbMasterDriverProxy::type_id::create("ahbMasterDriverProxy", this);
    end

    ahbMasterMonitorProxy = AhbMasterMonitorProxy::type_id::create("ahbMasterMonitorProxy", this);

    if (ahbMasterAgentConfig.hasCoverage) begin
        ahbMasterCoverage = AhbMasterCoverage::type_id::create("ahbMasterCoverage", this);
    end
endfunction : build_phase
```

Fig 3.16 AHB master agent build phase code snippet

In the connect_phase, configuration handles from the above components are mapped. If the AHB Master Agent is active, the `AhbMasterDriverProxy` and `AhbMasterSequencer` are connected using TLM ports. Additionally, the `AhbMasterMonitorProxy`'s

ahbMasterAnalysisPort is connected to the AhbMasterCoverage component's analysis_export port for transaction analysis and coverage collection.

```
function void AhbMasterAgent::connect_phase(uvm_phase phase);
  if(ahbMasterAgentConfig.is_active == UVM_ACTIVE) begin
    ahbMasterDriverProxy.ahbMasterAgentConfig = ahbMasterAgentConfig;
    ahbMasterSequencer.ahbMasterAgentConfig = ahbMasterAgentConfig;
    ahbMasterDriverProxy.seq_item_port.connect(ahbMasterSequencer.seq_item_export);
  end

  ahbMasterMonitorProxy.ahbMasterAgentConfig = ahbMasterAgentConfig;
  ahbMasterMonitorProxy.ahbMasterAgentConfig = ahbMasterAgentConfig;

  if(ahbMasterAgentConfig.hasCoverage) begin
    ahbMasterCoverage.ahbMasterAgentConfig = ahbMasterAgentConfig;
    ahbMasterMonitorProxy.ahbMasterAnalysisPort.connect(ahbMasterCoverage.analysis_export);
  end
endfunction : connect_phase
```

Fig 3.17 AHB master agent connect phase code snippet

3.2.14 AHB Master Sequencer

AhbMasterSequencer component is a parameterised class of type AhbMasterTransaction, extending uvm_sequencer. AHB sequencer sends the data from the ahb master sequences to the ahb driver proxy.

3.2.15 AHB Master Driver Proxy

The AhbMasterDriverProxy component is a parameterized class of type AhbMasterTransaction, extending uvm_driver. It obtains the AhbMasterAgentConfig handle and, based on the configuration, drives and samples signals such as haddr, hwddata, hwrite, hsize, hburst, and htrans. The master transaction is fetched into the AhbMasterDriverProxy using the get_next_item() method.

Since the ahbMasterDriverBFM interface cannot directly access class-based AhbMasterTransaction data, it is converted into a struct data type. Similarly, the AhbMasterAgentConfig values are also converted into a struct data type. The AhbMasterDriverProxy invokes the converter class to transform both the master transaction packet and master configuration packet into their respective struct formats (declared in the AHB global package) before passing them to ahbMasterDriverBFM. The driveToBFM

method, defined in ahbMasterDriverBFM, is then called to initiate driveToBFM(dataPacket, configPacket), facilitating the structured communication between the driver and the BFM.

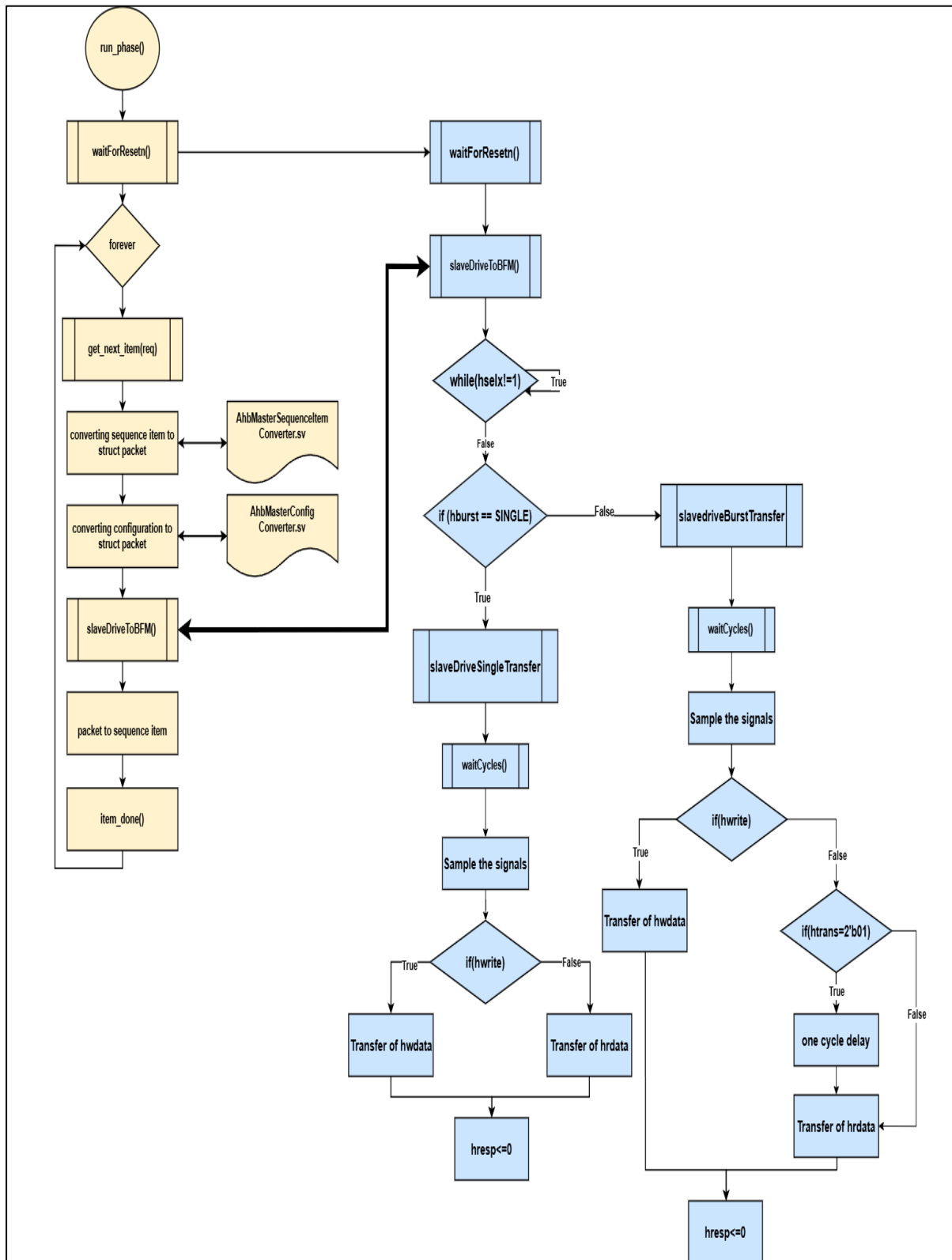


Fig 3.18 Flowchart of communication between ahb master driver proxy and ahb master driver bfm

```
task AhbMasterDriverProxy::run_phase(uvm_phase phase);
    ahbMasterDriverBFM.waitForResetn();
    forever begin
        ahbTransferCharStruct dataPacket;
        ahbTransferConfigStruct configPacket;
        seq_item_port.get_next_item(req);
        `uvm_info(get_type_name(), $sformatf("REQ-MASTERTX \n %s", req.sprint), UVM_HIGH);
        AhbMasterSequenceItemConverter::fromClass(req, dataPacket);
        AhbMasterConfigConverter::fromClass(ahbMasterAgentConfig, configPacket);
        ahbMasterDriverBFM.driveToBFM(dataPacket, configPacket);
        AhbMasterSequenceItemConverter::toClass(dataPacket, req);
        seq_item_port.item_done();
    end
endtask : run_phase
```

Fig 3.19 run phase of ahb master driver proxy code snippet

3.2.16 AHB Master Monitor Proxy

AhbMasterMonitorProxy component is a class extending uvm_monitor. It gets the AhbMasterAgentConfig handle and based on the configurations we will sample the hwdata and hrdata signals. It declares and creates the ahbMasterAnalysisPort to send the sampled data.

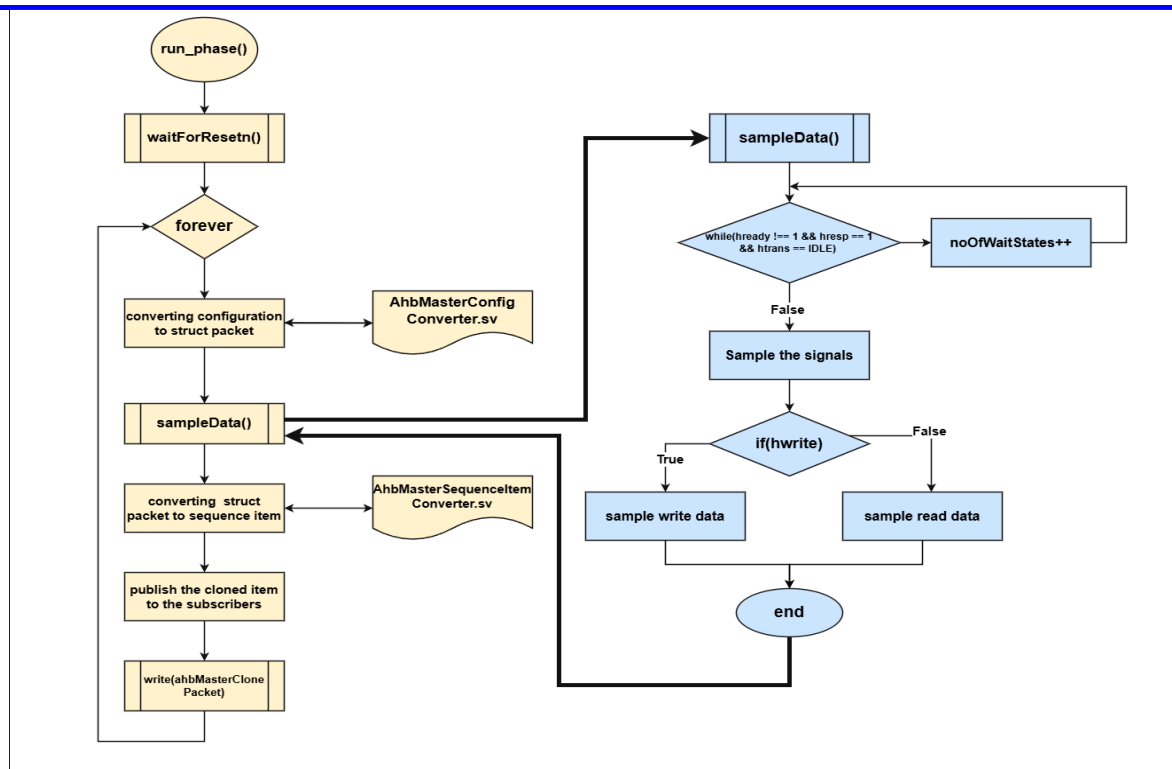


Fig 3.20 Flowchart of ahb master monitor proxy and ahb master monitor bfm communication

3.2.17 AHB Slave Agent

The AhbSlaveAgent is a class extending uvm_agent and is responsible for managing the AHB Slave components in a UVM testbench. It retrieves configuration settings from the AhbSlaveAgentConfig and, based on these settings, creates and connects the required components. The AhbSlaveSequencer and AhbSlaveDriverProxy are instantiated only if the agent is active, determined by the is_active variable in the configuration. Additionally, if the hasCoverage variable is set to 1, the AhbSlaveCoverage component is created during the build_phase.

The build phase of the AhbSlaveAgent includes the creation of:

- a. AhbSlaveSequencer
- b. AhbSlaveDriverProxy
- c. AhbSlaveMonitorProxy
- d. AhbSlaveCoverage

```

function void AhbSlaveAgent::build_phase(uvm_phase phase);
    super.build_phase(phase);

    if(ahbSlaveAgentConfig.is_active == UVM_ACTIVE) begin
        ahbSlaveSequencer = AhbSlaveSequencer::type_id::create("ahbSlaveSequencer",this);
        ahbSlaveDriverProxy = AhbSlaveDriverProxy::type_id::create("ahbSlaveDriverProxy",this);
    end

    ahbSlaveMonitorProxy = AhbSlaveMonitorProxy::type_id::create("ahbSlaveMonitorProxy",this);

    if(ahbSlaveAgentConfig.hasCoverage) begin
        ahbSlaveCoverage = AhbSlaveCoverage::type_id::create("ahbSlaveCoverage",this);
    end
endfunction : build_phase

```

Fig 3.21 AHB slave agent build phase code snippet

During the connect_phase, the AhbSlaveAgentConfig handles declared in these components are mapped accordingly. If the agent is active, the AhbSlaveDriverProxy and AhbSlaveSequencer are connected using TLM ports to facilitate transaction flow. The ahbSlaveMonitorProxy's ahbSlaveAnalysisPort is connected to the ahbSlaveCoverage's analysis_export to ensure coverage data is properly recorded.

```

function void AhbSlaveAgent::connect_phase(uvm_phase phase);

    if(ahbSlaveAgentConfig.is_active == UVM_ACTIVE) begin
        ahbSlaveDriverProxy.ahbSlaveAgentConfig = ahbSlaveAgentConfig;
        ahbSlaveSequencer.ahbSlaveAgentConfig = ahbSlaveAgentConfig;
        ahbSlaveDriverProxy.seq_item_port.connect(ahbSlaveSequencer.seq_item_export);
    end

    ahbSlaveMonitorProxy.ahbSlaveAgentConfig = ahbSlaveAgentConfig;
    ahbSlaveMonitorProxy.ahbSlaveAgentConfig = ahbSlaveAgentConfig;

    if(ahbSlaveAgentConfig.hasCoverage) begin
        ahbSlaveCoverage.ahbSlaveAgentConfig = ahbSlaveAgentConfig;
        ahbSlaveMonitorProxy.ahbSlaveAnalysisPort.connect(ahbSlaveCoverage.analysis_export);
    end
endfunction : connect_phase

```

Fig 3.22 AHB slave agent connect phase code snippet

3.2.18 AHB Slave Sequencer

AhbSlaveSequencer component is a parameterised class of type AhbSlaveTransaction, extending uvm_sequencer. AhbSlaveSequencer sends the data from the slaveSequences to the ahbSlaveDriverProxy.

3.2.19 AHB Slave Driver Proxy

AhbSlaveDriverProxy component is a parameterised class of type AhbSlaveTransaction, extending uvm_driver. It gets AhbSlaveAgentConfig handle and based on the configurations drives and samples the hwdatas and hrdatas signals. The driver proxy receives transactions using the get_next_item() method from the sequencer.

Since the AhbSlaveDriverBFM cannot directly process class-based transactions, the driver proxy converts transactions into a struct format before sending them to the BFM. Similarly, the AhbSlaveAgentConfig values are also converted into a struct format. The AhbSlaveDriverProxy utilizes a converter class to transform the slave transaction packet and slave configuration packet into struct data packets. These converted packets are then passed to the AhbSlaveDriverBFM using the slaveDriveToBFM() method. This method initiates the transaction by calling slaveDriveToBFM(structPacket, structConfig), ensuring that the converted transaction and configuration packets are correctly driven onto the AHB bus.

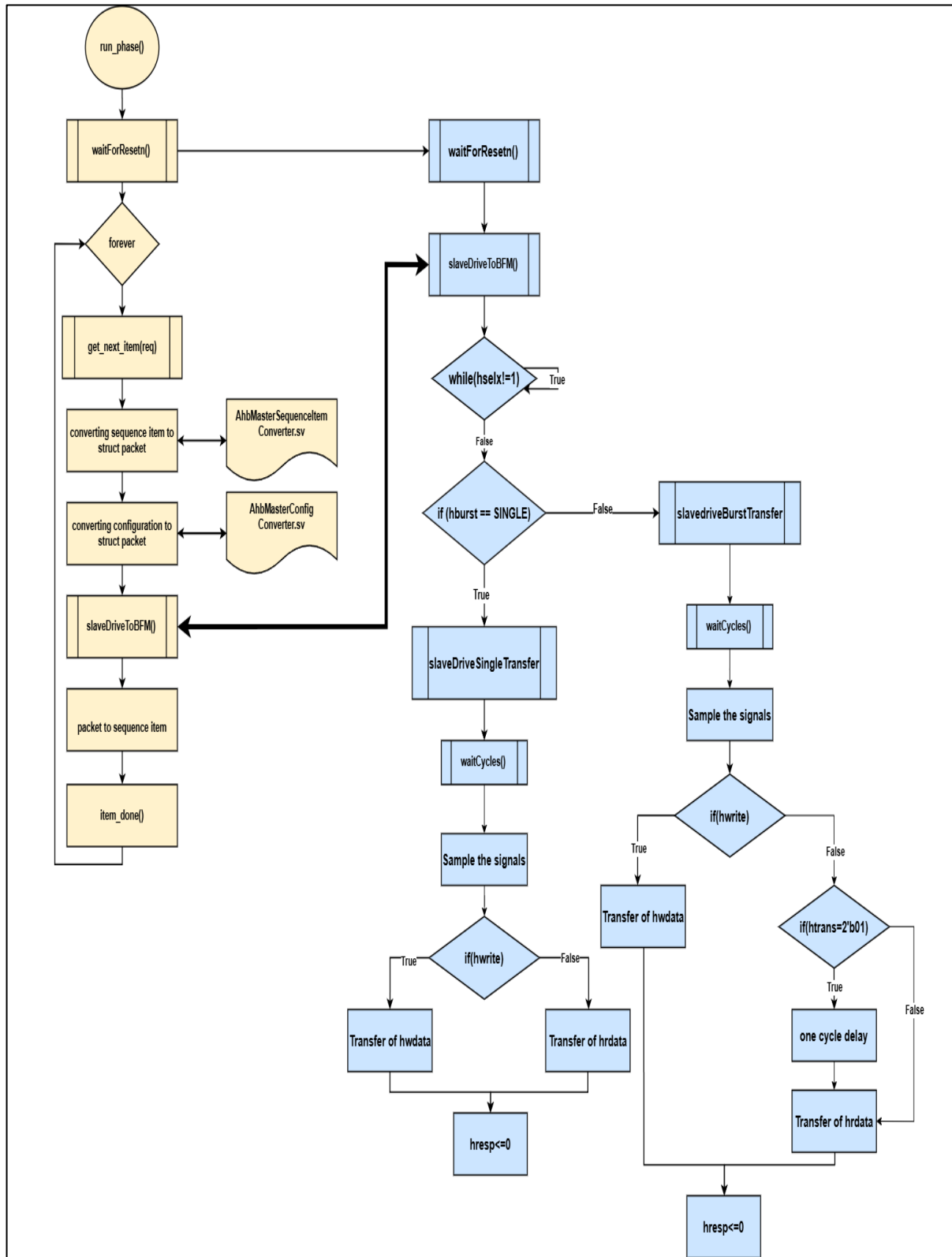


Fig 3.23 Flowchart of ahb slave driver bfm and slave driver proxy communication

```

task AhbSlaveDriverProxy::run_phase(uvm_phase phase);
`uvm_info(get_type_name(), $sformatf(" BEFORERESET \n "), UVM_NONE);

    ahbSlaveDriverBFM.waitForResetn();

    forever begin
        ahbTransferCharStruct structPacket;
        ahbTransferConfigStruct structConfig;

        seq_item_port.get_next_item(req);
        if(req.choosePacketData) begin

            AhbSlaveSequenceItemConverter::fromClass(req, structPacket);
            AhbSlaveConfigConverter::fromClass(ahbSlaveAgentConfig, structConfig);

            ahbSlaveDriverBFM.slaveDriveToBFM(structPacket, structConfig);

            AhbSlaveSequenceItemConverter::toClass(structPacket, req);

            `uvm_info("SONAL", $sformatf("STRUCTPACKET = %p", req), UVM_LOW)

            if(structPacket.hwrite == WRITE)begin
                `uvm_info("AIL", "ENTERED WRITE LOOP", UVM_LOW)
                taskWrite(structPacket);
            end
            else begin
                taskRead(structPacket);
            end
        end
        else
        begin
            AhbSlaveSequenceItemConverter::fromClass(req, structPacket);
            AhbSlaveConfigConverter::fromClass(ahbSlaveAgentConfig, structConfig);
            ahbSlaveDriverBFM.slaveDriveToBFM(structPacket, structConfig);
            AhbSlaveSequenceItemConverter::toClass(structPacket, req);
        end
        seq_item_port.item_done();
    end
endtask : run_phase

```

Fig 3.24 AHB slave driver proxy run phase code snippet

3.2.20 AHB Slave Monitor Proxy

The AhbSlaveMonitorProxy component extends uvm_monitor. It retrieves the AhbSlaveAgentConfig handle and, based on the configuration, samples key signals such as hwddata and hradta. The monitor declares and creates the ahbSlaveAnalysisPort, which is used to send the captured transactions to other verification components, such as scoreboards and coverage collectors. The AhbSlaveMonitorProxy will get the sampled data from

AhbSlaveMonitorBFM as shown in figure 3.25. The sampled transaction is converted using the AhbSlaveSequenceItemConverter and then sent through the ahbSlaveAnalysisPort.

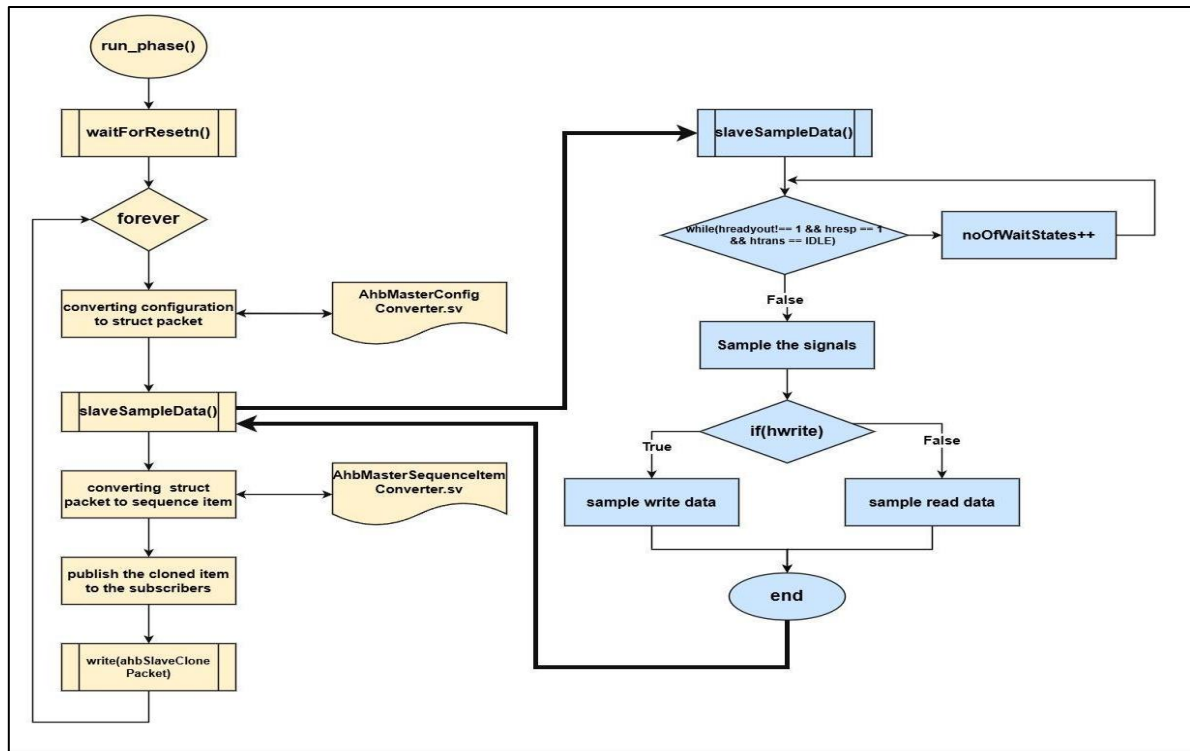


Fig 3.25 Flowchart of ahb slave monitor bfm and slave monitor proxy communication

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

    AhbSlaveTransaction ahbSlavePacket;

    ahbSlavePacket = AhbSlaveTransaction::type_id::create("slave Packet");
    ahbSlaveMonitorBFM.waitForResetrn();

    forever begin
        ahbTransferCharStruct structDataPacket;
        ahbTransferConfigStruct structConfigPacket;
        AhbSlaveTransaction ahbSlaveClonePacket;

        AhbSlaveConfigConverter :: fromClass (ahbSlaveAgentConfig, structConfigPacket);
        ahbSlaveMonitorBFM.slaveSampleData (structDataPacket, structConfigPacket);

        $display("values inside monitor proxy %p",structDataPacket);
        AhbSlaveSequenceItemConverter :: toClass (structDataPacket, ahbSlavePacket);

        $cast(ahbSlaveClonePacket, ahbSlavePacket.clone());
        `uvm_info(get_type_name(),$sformatf("Sending packet via analysis_port: , \n %s",
            ahbSlaveClonePacket.sprint()),UVM_HIGH)
        ahbSlaveAnalysisPort.write(ahbSlaveClonePacket);
    end
endtask : run_phase
```

Fig 3.26 run phase of ahb 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 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 shows the Verbosity levels that have used in this particular project

Table 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.

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.

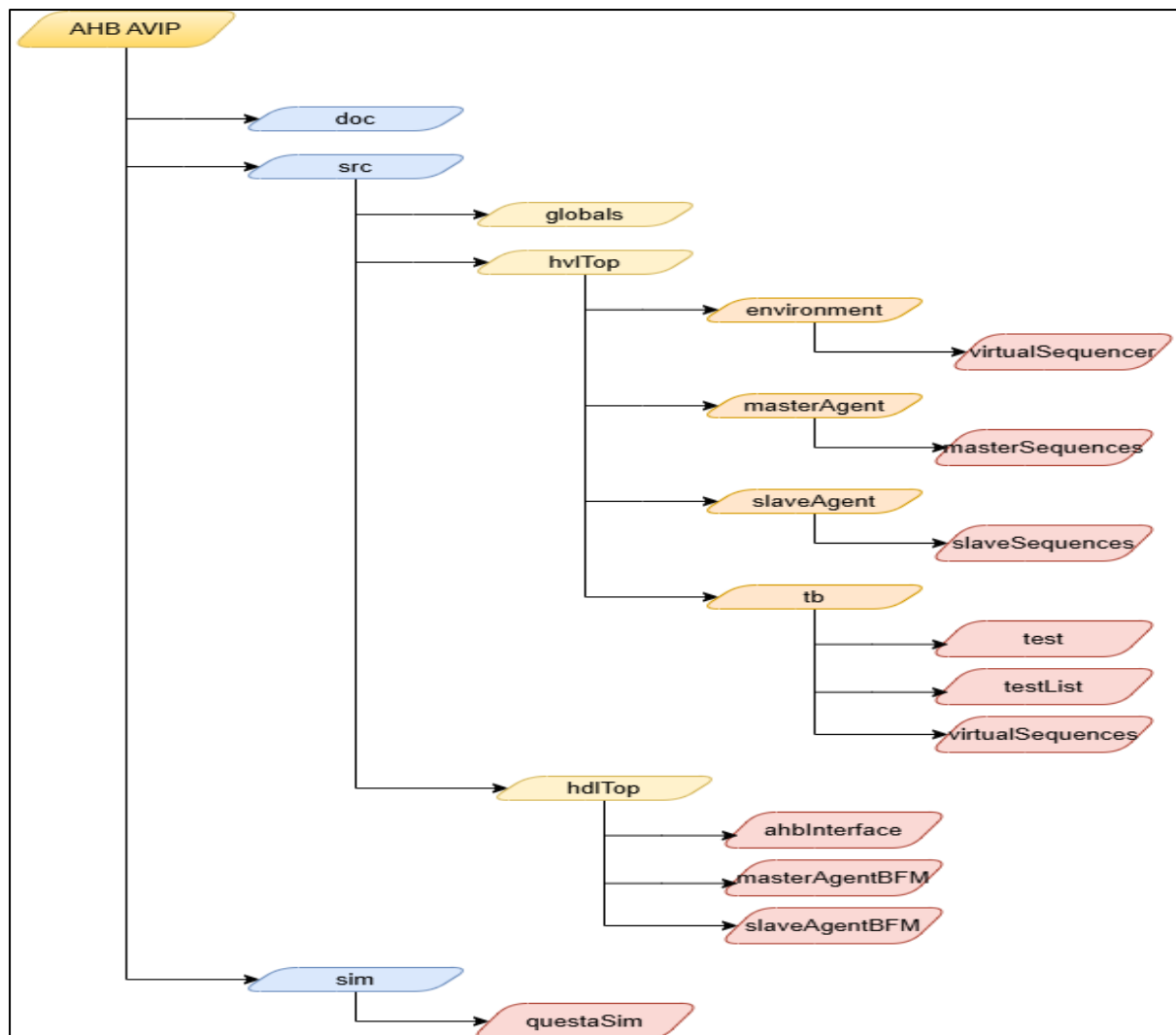


Fig 4.1 Package Structure of AHB_AVIP

Table 4: Directory Path

Directory	Description
ahb_avip/doc	Contains test bench architecture and components description and verification plan and assertion plan
ahb_avip/sim	Contains all simulating tools and ahb_compile.f file which contain all directories and compiling files
ahb_avip/src/globals	Contains global package parameters(names,modes)
ahb_avip/src/hvlTop	Contains all tb component folders (environment, master agent, slave agent, tb)
ahb_avip/src/hdlTop	Contains all bfm files and interface
ahb_avip/src/hdlTop/masterAgentBFM	Contains master agent , driver and monitor bfm files, master coverproperty and assertion files.
ahb_avip/src/hdlTop/slaveAgentBFM	Contains slave agent, driver and monitor bfm files. Slave coverproperty and assertion files.
ahb_avip/src/hdlTop/ahbInterface	Contains ahb interface file
ahb_avip/src/hvlTop/tb	Contains testbench for master and slave assertions, coverproperty and ahbAssertion.f file. Additionally, it includes test files, test lists, and virtual sequences
ahb_avip/src/hvlTop/tb/test	Contains all the test files
ahb_avip/src/hvlTop/tb/testList	Contains test list for Regression test.
ahb_avip/src/hvlTop/tb/virtualSequences	Contain all virtual sequence test files
ahb_avip/src/hvlTop/environment	Contains environment along with its config and package files and Scoreboard file and virtual sequencer folder
ahb_avip/src/hvlTop/environment/virtualSequencer	Contains master and slave virtual sequencer and base virtual sequencer
src/hvlTop/masterAgent	Contains master agent, agent config, conifg converter, coverage, diver proxy, monitor proxy, package, sequence item converter, sequencer, transactions and a master sequence folder
src/hvlTop/masterAgent/masterSequences	Contains all master test sequences

src/hvlTop/slaveAgent	Contains slave agent, agent config, conifg converter, coverage, diver proxy, monitor proxy, package, sequence item converter, sequencer, transactions and a slave sequence folder.
src/hvlTop/slaveAgent/slaveSequences	Contains all slave test sequences

Configuration

5.1 Global package variables

Table 5: Global package variables

Name	Type	Description
NO_OF_MASTERS	integer	Specifies no of master connected to the AHB interface
NO_OF_SLAVES	integer	Specifies no of slave connected to the AHB interface
MASTER_AGENT_ACTIVE	bit	Determines whether master agent is active or not
SLAVE_AGENT_ACTIVE	bit	Determines whether slave agent is active or not
ADDR_WIDTH	int	Specifies the address width
DATA_WIDTH	int	Specifies the data width
HMASTER_WIDTH	int	Determines the required bit-width for the HMASTER signal
HPROT_WIDTH	int	Specifies the bit-width of the HPROT signal
SLAVE_MEMORY_SIZE	int	Determines the size of the addressable memory region for a slave
SLAVE_MEMORY_GAP	int	Determines the gap or spacing between the address regions assigned to consecutive slaves
MEMORY_WIDTH	int	Determines the data width of a memory module
LENGTH	int	Specifies the maximum size of the array to store burst transfers
ahbBurstEnum	enum	Used to represent the type of burst transaction SINGLE = 3'b000 INCR = 3'b001 WRAP4 = 3'b010 INCR4 = 3'b011 WRAP8 = 3'b100 INCR8 = 3'b101 WRAP16 = 3'b110 INCR16 = 3'b111
ahbTransferEnum	enum	Used to represent the type of transfer IDLE = 2'b00 BUSY = 2'b01 NONSEQ = 2'b10

		SEQ = 2'b11
ahbRespEnum	enum	Used to represent the type of resp OKAY = 1'b0 ERROR = 1'b1
ahbHsizeEnum	enum	Used to represent the size of the transaction BYTE = 3'b000 // 8 bits HALFWORD = 3'b001 // 16 bits WORD = 3'b010 // 32 bits DOUBLEWORD = 3'b011 // 64 bits LINE4 = 3'b110 // 128 bits (4-word line) LINE8 = 3'b101, // 256 bits (8-word line) LINE16 = 3'b110, // 512 bits LINE32 = 3'b111 // 1024 bits
ahbProtectionEnum	enum	Used to represent the protection type for transaction NORMAL_SECURE_DATA = 4'b0000, NORMAL_SECURE_INSTRUCTION = 4'b0001, NORMAL_NONSECURE_DATA = 4'b0010, NORMAL_NONSECURE_INSTRUCTION = 4'b0011, PRIVILEGED_SECURE_DATA = 4'b0100, PRIVILEGED_SECURE_INSTRUCTION = 4'b0101, PRIVILEGED_NONSECURE_DATA = 4'b0110, PRIVILEGED_NONSECURE_INSTRUCTION = 4'b0111
ahbWriteEnum	enum	WRITE=1'b1 : write transfer happen READ=1'b0 : read transfer happen
ahbTransferCharStruct	struct	Structure to hold the packet data.
ahbTransferConfigStruct	struct	Structure to hold the configuration data.

Configuration used

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

5.2 Master agent configuration

Table 6: AhbMasterAgentConfig

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.
hasCoverage	bit	`d1	Used for enabling the master agent coverage
noOfWaitStates	int	`d0	Defines the number of extra wait states before initiating a transaction.

5.3 Slave agent configuration

Table 7: AhbSlaveAgentConfig

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
hasCoverage	bit	`d1	Used for enabling the slave agent coverage.
noOfWaitStates	int	`d0	Defines the number of wait states the slave introduces (declared as rand for randomization).

5.4 Environment configuration

Table 8: AhbEnvironmentConfig

Name	Type	Default value	Description
hasScoreboard	bit	1	Enables the scoreboard, it usually receives the transaction level objects via TLM ANALYSIS PORT.

hasVirtualSequencer	bit	1	Enables the virtual sequencer which has master and slave sequencer
noOfSlaves	integer	'h1	Number of slaves connected to the SPI interface
noOfMasters	integer	'h1	Number of Masters connected to the SPI interface

Chapter 6

Verification Plan

6.1 Verification plan

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

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

6.2 Template of Verification Plan

For more information of Verification Plan click below link:

[Ahb Verification Plan](#)

SI No	Sections	Features	Sub-Features	Description	Priority	Status	TestcasesNames	AssetProperty
DIRECTED TESTCASES								
1	1. Write Data Transfers	32-bit Transfer	8-bit Transfer	Verify write data transfer where the data bus width (HWDATA) is 8 bits.	2	Pass	Ah08b05SingleWriteTest	checkHeaderValid
2			16-bit Transfer	Verify write data transfer where the data bus width (HWDATA) is 16 bits.	2	Pass	Ah16b05SingleWriteTest	
3			32-bit Transfer	Verify write data transfer where the data bus width (HWDATA) is 32 bits.	1	Pass	Ah32b05SingleWriteTest	
4			64-bit Transfer	Verify write data transfer where the data bus width (HWDATA) is 64 bits.	1	Pending	Ah64b05SingleWriteTest	
5	2. Read Data Transfers	32-bit Transfer	8-bit Transfer	Verify read data transfer where the data bus width (HRDATA) is 8 bits.	2	Pass	Ah08b05SingleReadTest	checkHeaderValid
6			16-bit Transfer	Verify read data transfer where the data bus width (HRDATA) is 16 bits.	2	Pass	Ah16b05SingleReadTest	
7			32-bit Transfer	Verify read data transfer where the data bus width (HRDATA) is 32 bits.	1	Pass	Ah32b05SingleReadTest	
8			64-bit Transfer	Verify read data transfer where the data bus width (HRDATA) is 64 bits.	1	Pending	Ah64b05SingleReadTest	
9	3. Transfer Direction	HWRITE	i) 1 (Write operation)	Verify that HWRITE = 1 when a write operation is initiated.	1	Pass	Ah32b05Hr4WriteTest	checkHeaderValid
10			ii) 0 (Read operation)	Verify that HWRITE = 0 when a read operation is initiated.	1	Pass	Ah32b05Hr4ReadTest	checkHeaderValid
11	4. Transfer status	HREADY	i) 1 (Ready for transfer)	Verify that HREADY = 1 when the bus is ready to complete a transfer.	1	Pass	Ah32b05Hr4WriteTest	
12			ii) 0 (Not ready for transfer)	Verify that HREADY = 0 when the bus is not ready, causing a delay in transfer.	1	Pass	Ah32b05SingleWriteWithWaitStateTest	
13			iii) 0 for multiple cycles	Verify that HREADY = 0 for multiple cycles causes the transfer to be delayed.	1	Pass	Ah32b05Hr4WriteWithWaitStateTest	NA
14			iv) 1 with long transfer delay	Verify that HREADY = 1 is asserted after a long transfer delay (multiple cycles).	1	Pending		NA
15			v) 0 and HTRANS = IDLE	Verify that HREADY = 0 during HTRANS = IDLE results in no transfer occurring.	1	Pending		NA

Fig 5.1 Verification plan template

6.3 Sections for different test Scenarios

6.3.1 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.

This test describes the different transactions and burst transfers.

Table 9: Checking coverage closure for the different transactions and burst transfers

Sl. no	Test names	Description
1	AhbWriteTest	Verifies AHB write operation for burst types WRAP4, INCR4, WRAP8, INCR8, WRAP16, INCR16
2	AhbReadTest	Verifies AHB read operation for burst types WRAP4, INCR4, WRAP8, INCR8, WRAP16, INCR16
3	AhbSingleWriteTest	Verifies AHB write operation for SINGLE burst transfer
4	AhbSingleReadTest	Verifies AHB read operation for SINGLE burst transfer
5	AhbWriteWithBusyTest	Verifies AHB write operation with BUSY transaction
6	AhbReadWithBusyTest	Verifies AHB read operation with BUSY transaction
7	AhbSingleWriteWithWaitStateTest	Verifies AHB write operation with wait state for Single Transfer
8	AhbSingleReadWithWaitStateTest	Verifies AHB read operation with wait state for Single Transfer
9	AhbWriteWithWaitStateTest	Verifies AHB write operation with wait state
10	AhbReadWithWaitStateTest	Verifies AHB read operation with wait state

Assertion Plan

7.1 Assertion Plan overview

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

7.1.1 What are Assertions?

- An assertion specifies the behavior of the system.
- It is a piece of verification code that monitors a design implementation for compliance with the specifications.
- It is a directive to a verification tool to prove, assume, or count a given property using formal methods.
- Helps in detecting functional bugs early in the design cycle.
- Can be used in simulation, formal verification, and emulation environments.

7.1.2 Why Do We Use Assertions?

- Assertions are primarily used to validate the behavior of a design.
- They help in detecting functional bugs early and locating their source faster.
- Assertions provide functional coverage to ensure all scenarios are exercised.
- They flag invalid input stimulus that does not conform to assumed requirements.
- Enable formal verification by proving or checking properties automatically.

7.1.3 Benefits of Assertions

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

7.2 Template of Assertion Plan

The Assertion Plan template is provided in an Excel sheet. Please refer to the link below.

[AHB Assertion Plan](#)

7.3 Master Assertion Condition

7.3.1 checkHaddrAlignment

```
property checkHaddrAlignment;
  @(posedge hclk) disable iff (!hresetn)
    (hready && (htrans != 2'b00) && hburst != 3'b000 && hsize != 3'b000) |-> ((hsize == 3'b001) && (haddr[0] == 1'b0)) ||
    ((hsize == 3'b010) && (haddr[1:0] == 2'b00));
endproperty

assert property (checkHaddrAlignment)
  $info("HADDR is aligned based on HSIZE");
else $error("HADDR is not aligned based on HSIZE!");
```

Fig 7.1 checkHaddrAlignment Assertion

The checkHaddrAlignment property is evaluated as follows:

- Initially, it will check for posedge of hclk, and the property is disabled if hresetn is low.
- When hready is high, htrans is not IDLE (2'b00), hburst is not SINGLE (3'b000), and hsize is not BYTE (3'b000), then at the same cycle, it will check the alignment of HADDR based on HSIZE:
 - If hsize is HALFWORD (3'b001), then haddr[0] should be 0.
 - If hsize is WORD (3'b010), then haddr[1:0] should be 00.
- If the above alignment conditions hold, then the property is true. Otherwise, the property will fail, and an error message is raised.

7.3.2 checkStrobe

```
property checkStrobe;
  @(posedge hclk) disable iff (!hresetn)
    (htrans != 2'b00) |->((hsize == 3'b000 -> $countones(hwstrb)== 1 )) ||
      ((hsize ==3'b001 -> $countones(hwstrb)==2)) ||
      ((hsize==3'b010 -> $countones(hwstrb)==4));
endproperty

assert property (checkStrobe)
  $info("Hwstrb valid ");|
else $error("Hwstrb is not valid for hsize");
```

Fig 7.2 checkStrobe Assertion

The checkStrobe property is evaluated as follows:

- Initially, it will check for posedge of hclk, and the property is disabled if hresetn is low.
- When htrans is not IDLE (2'b00), then at the same cycle, it will check the validity of hwstrb based on hsize:
 - If hsize is BYTE (3'b000), then hwstrb should have exactly 1 bit set (1'b1 count = 1).
 - If hsize is HALFWORD (3'b001), then hwstrb should have exactly 2 bits set (1'b1 count = 2).
 - If hsize is WORD (3'b010), then hwstrb should have exactly 4 bits set (1'b1 count = 4).
- If the above strobe conditions hold, then the property is true. Otherwise, the property will fail, and an error message is raised.

7.4 Slave Assertion Condition

7.4.1 checkHrespOKayForValid

```
property checkHrespOkayForValid;
  @(posedge hclk) disable iff (!hresetn)
    (hreadyout && (htrans != 2'b00)) | => (hresp == 1'b0);
endproperty

assert property (checkHrespOkayForValid)
  $info("HRESP is OKAY for valid transactions!");
else $error("HRESP is not OKAY for valid transactions!");
```

Fig 7.3 checkHrespOKayForValid Assertion

The checkHrespOKayForValid property is evaluated as follows:

- Initially, it will check for posedge of hclk, and the property is disabled if hresetn is low.
- When hreadyout is high and htrans is not IDLE (2'b00), then at the same cycle, it will check that hresp is 0 (indicating an OKAY response).
- If hresp is 0 (OKAY response), then the property is true. Otherwise, the property will fail, and an error message is raised.

7.4.1 checkSlaveHrdataValid

```
property checkSlaveHrdataValid;
  @(posedge hclk) disable iff (!hresetn)
    (!hwrite && hreadyout && (htrans != 2'b00) && hselx) ##1 $stable(hreadyout)
    |-> (!$isunknown(hrdata));
endproperty

assert property (checkSlaveHrdataValid)
  $info("HRDATA is valid during read transfer!");
else $error("HRDATA is invalid during read transfer!");
```

Fig 7.4 checkSlaveHrdataValid Assertion

The checkSlaveHrdataValid property is evaluated as follows:

- Initially, it will check for posedge of hclk, and the property is disabled if hresetn is low.

-
- When hwrite is low (indicating a read transfer), hreadyout is high, htrans is not IDLE (2'b00) and hselx is 1 then at next clock cycle, check whether hreadyout is stable if true check whether is unknown.
 - If hrdata is valid (not unknown), then the property is true, Otherwise, the property will fail, and an error message is raised.

Coverage Plan

8.1 Template of Coverage Plan

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

[AHB Coverage Plan](#)

8.2 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.

8.3 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 8.1 uvm_subscriber

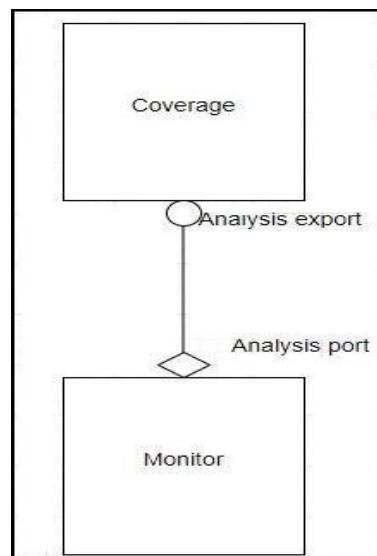


Fig 8.2 Monitor and coverage connection

8.3.1 Analysis export

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

8.3.2 Write function

The write function is to process the incoming transactions.

```
function void AhbMasterCoverage::write(AhbMasterTransaction t);  
  `uvm_info(get_type_name(), $sformatf("Before calling SAMPLE METHOD"), UVM_HIGH);  
  ahbMasterCovergroup.sample(ahbMasterAgentConfig, t);  
  `uvm_info(get_type_name(), "After calling SAMPLE METHOD", UVM_HIGH);  
  
endfunction : write
```

Fig 8.3 Write function

8.4 Covergroup

```
covergroup ahbMasterCovergroup with function sample (AhbMasterAgentConfig ahbMasterAgentConfig,  
                                                    AhbMasterTransaction ahbMasterTransaction);  
  
  option.per_instance = 1;
```

Fig 8.4 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.

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

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

Coverpoint: HADDR_CP

Comment:
AHB Address Coverage

Search:

Bin Name	At Least	Hits
default_bin ahbAddrDefault	--	0
ahbAddrByteAligned	1	4015
ahbAddrHalfWordAligned	1	3663
ahbAddrWordAligned	1	2739
ahbAddrAllZeroes	1	420

Fig 8.5 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.

8.4 Bucket

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

```
HBURST_CP:coverpoint ahbMasterTransaction.hburst{  
  option.comment = " ahb burst";  
  bins ahbSingle={0};  
  bins ahbWrap4={2};  
  bins ahbIncr4={3};  
  bins ahbWrap8={4};  
  bins ahbIncr8={5};  
  bins ahbWrap16={6};  
  bins ahbIncr16={7};  
}
```

Fig 8.6 Bucket

- In the above points we can see that the Mode[] have the bins for each value.

8.5 Coverpoints

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

```
HWRITE_CP:coverpoint ahbMasterTransaction.hwrite{
    option.comment = " ahb write";
    bins ahbRead ={0};
    bins ahbWrite ={1};
}
```

Fig 8.7 Coverpoint

8.6 Cross Coverpoints

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

Cross coverage between coverpoints HSIZE_CP and HBURST_CP

```
HEXCL_CP:coverpoint ahbMasterTransaction.hexcl{
    option.comment = " ahb excl";
    bins ahbExcl0={0};
}

HSIZE_CP_x_HBURST_CP:cross HSIZE_CP ,HBURST_CP;
```

Fig 8. 8 Cross Coverpoints

8.6.1 Illegal bins

```
HSIZE_CP:coverpoint ahbMasterTransaction.hsize{
    option.comment = " ahb size";
    bins ahbByte    ={0};
    bins ahbHalfWord  ={1};
    bins ahbWord    ={2};
    illegal_bins illegalBinsOfAhbSize64Bytes = {6};
    illegal_bins illegalBinsOfAhbSize128Bytes = {7};
}
```

Fig 8.9 Illegal bins

Illegal bins are used when we don't want to have the particular value eg - we don't want to have the hsize to be value 6 and 7, so we create the illegal bin for it.

8.7 Creation of the covergroup

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

Fig 8.10 Creation of covergroup

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

8.8 Sampling of the covergroup

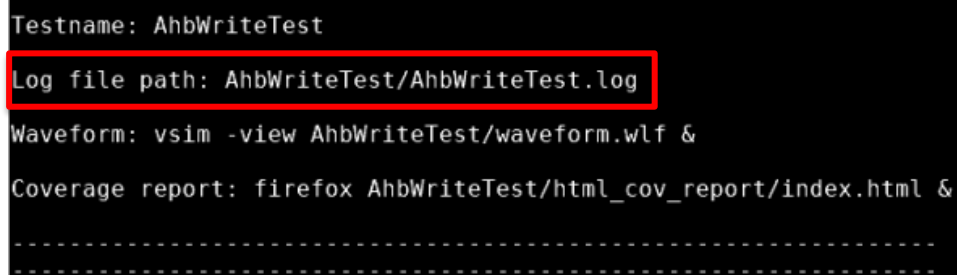
In this the sampling of the covergroup is done in the write function as shown below

```
function void AhbMasterCoverage::write(AhbMasterTransaction t);  
    `uvm_info(get_type_name(), $sformatf("Before calling SAMPLE METHOD"), UVM_HIGH);  
    ahbMasterCovergroup.sample(ahbMasterAgentConfig, t);  
    `uvm_info(get_type_name(), "After calling SAMPLE METHOD", UVM_HIGH);  
endfunction : write
```

Fig 8.11 Sampling of the covergroup

8.9 Checking for the coverage

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



```
Testname: AhbWriteTest  
Log file path: AhbWriteTest/AhbWriteTest.log  
Waveform: vsim -view AhbWriteTest/waveform.wlf &  
Coverage report: firefox AhbWriteTest/html_cov_report/index.html &  
-----  
-----
```

Fig 8.12 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 :-

```

Testname: AhbWriteTest
Log file path: AhbWriteTest/AhbWriteTest.log
Waveform: vsim -view AhbWriteTest/waveform.wlt &
Coverage report: firefox AhbWriteTest/html_cov_report/index.html &
.....

```

Fig 8.13 Coverage report path

Then new html window will open

Coverage Summary by Type:						
Total Coverage:					73.79%	76.27%
Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Covergroups	55	55	0	1	100.00%	100.00%
Directives	11	11	0	1	100.00%	100.00%
Statements	1371	878	493	1	64.04%	64.04%
Branches	984	337	647	1	34.24%	34.24%
FEC Conditions	21	8	13	1	38.09%	38.09%
Toggles	2140	2088	52	1	97.57%	97.57%
Assertions	16	16	0	1	100.00%	100.00%

Fig 8.14 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.

ahbMasterCovergroup

Summary	Total Bins	Hits	Hit %
Coverpoints	30	30	100.00%
Crosses	21	21	100.00%

Search:

CoverPoints	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
HADDR_CP	4	4	0	100.00%	100.00%	100.00%
HBURST_CP	7	7	0	100.00%	100.00%	100.00%
HEXCL_CP	1	1	0	100.00%	100.00%	100.00%
HMASTLOCK_CP	2	2	0	100.00%	100.00%	100.00%
HNONSEC_CP	1	1	0	100.00%	100.00%	100.00%
HPROT_CP	3	3	0	100.00%	100.00%	100.00%
HSIZE_CP	3	3	0	100.00%	100.00%	100.00%
HTRANS_CP	4	4	0	100.00%	100.00%	100.00%
HWRITE_CP	2	2	0	100.00%	100.00%	100.00%
HWSTRB_CP_0	3	3	0	100.00%	100.00%	100.00%

Fig 8.15 All coverpoints present in the Master Covergroup

ahbSlaveCovergroup

Summary	Total Bins	Hits	Hit %
Coverpoints	4	4	100.00%
Crosses	0	0	0.00%

Search:

CoverPoints	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
HREADY_CP	2	2	0	100.00%	100.00%	100.00%
HRESP_CP	2	2	0	100.00%	100.00%	100.00%

Fig 8.16 All coverpoints present in the Slave Covergroup

Coverpoint: HWRITE_CP			
		Search:	<input type="text"/>
Bin Name	At Least	Hits	
ahbRead	1	314	
ahbWrite	1	197	

Fig 8.17 Individual Coverpoint Hit

Test Cases

9.1 Test Flow

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

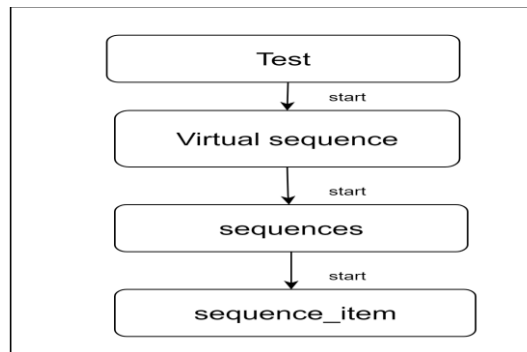


Fig 9.1 Test flow

9.2 AHB Test Cases FlowChart

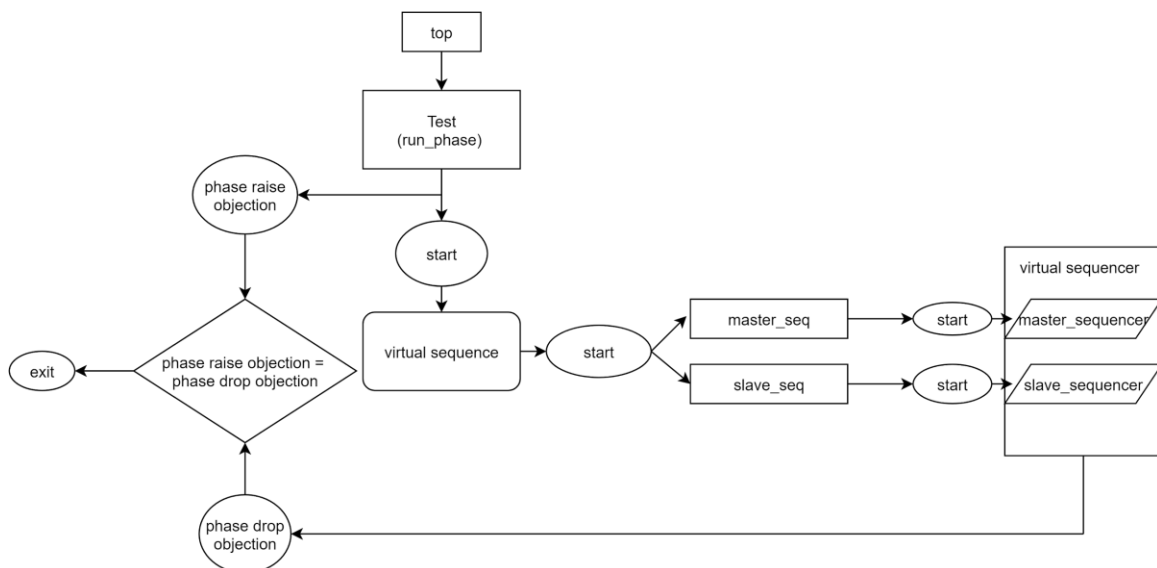


Fig 9.2 AHB test cases flow chart

9.3 Transaction

Table 10: Transaction Signals

Variables	Type	Description
haddr	bit	Address. This is the AHB address bus. It can be up to 32 bits wide.
hselx	bit	Master asserts the hselx to select the slave device hwrite signal decides whether write data transfer happens from the master side or read data transfer happens to the master.
hburst	enum	Burst type: Defines whether the transfer is a single transaction or part of a burst.
hmastlock	bit	Indicates a locked transfer. When 1, the master keeps control of the bus until the transaction completes. When 0, normal arbitration applies.
hprot	enum	Protection control: Defines the type of access for a transaction.
hsize	enum	Transfer size: Defines the number of bytes in each transfer (byte, halfword, word, etc.).
hnonsec	bit	Security attribute: Defines whether the transfer is secure or non-secure.
hmaster	bit	Indicates which master is performing the transfer in a multi-master system.
htrans	enum	Indicates the type of transfer on the bus. It defines whether the transfer is Busy, idle, sequential, or non-sequential.
hwdata	bit	Write data bus: Carries data from the master to the slave during a write operation.

hwstrb	bit	Write strobe signals: Indicate which byte lanes in hwddata are valid during a write transfer.
hwrite	enum	Write enable signal: Defines whether the transaction is a read (0) or write (1) operation.
hrdata	bit	Read data bus: Carries data from the slave to the master during a read operation.
hreadyout	bit	Slave Ready Signal: Indicates whether the slave is ready to complete the current transfer.
hresp	enum	Transfer response: Indicates whether the current transfer was successful or encountered an error.
hexokay	bit	Exclusive Access OK Signal: Indicates whether an exclusive transfer was successful.
hready	bit	Final Ready Signal: Indicates whether the current transfer is complete and the next transfer can begin.
noOfWaitStatesDetected	int	Counts the number of wait states inserted before a transfer is completed.
busyControl	bit	Indicates if the master is in a BUSY state. Prevents new transactions from starting.

9.3.1 AhbMasterTransaction

- AhbMasterTransaction class is extended from the uvm_sequence_item holds the data items required to drive stimulus to dut.
- Declared all the variables (haddr, hselx, hburst, hmastlock, hprot, hsize, hnonsec, hmaster, htrans, hwddata, hwstrb, hwrite, hrdata, hreadyout, hresp, hexokay, hready, noOfWaitStatesDetected, busyControl).
- Constraint declared for slave select and data transfer based on transfer size.

```

constraint strobleValue{foreach(hwstrb[i]) { if(hsize == BYTE) $countones(hwstrb[i]) == 1;
                                           else if(hsize == HALFWORD) $countones(hwstrb[i]) == 2;
                                           else if(hsize == WORD) $countones(hwstrb[i]) == 4;
                                           else if(hsize == DOUBLEWORD) $countones(hwstrb[i]) == 8;
                                           }
}

constraint burstsize{if(hburst == WRAP4 || hburst == INCR4) hwdatasize() == 4;
                    else if(hburst == WRAP8 || hburst == INCR8) hwdatasize() == 8;
                    else if(hburst == WRAP16 || hburst == INCR16) hwdatasize() == 16;
                    else hwdatasize() == 1;
}

constraint strobesize{if(hburst == WRAP4 || hburst == INCR4) hwstrb.size() == 4;
                    else if(hburst == WRAP8 || hburst == INCR8) hwstrb.size() == 8;
                    else if(hburst == WRAP16 || hburst == INCR16) hwstrb.size() == 16;
                    else hwstrb.size() == 1;
}

constraint busyState{if(hburst == WRAP4 || hburst == INCR4) busyControl.size() == 4;
                    else if(hburst == WRAP8 || hburst == INCR8) busyControl.size() == 8;
                    else if(hburst == WRAP16 || hburst == INCR16) busyControl.size() == 16;
                    else busyControl.size() == 1;
}

```

Fig 9.3 Constraints of Ahb Master transaction

Table 11: Describing constraints of AhbMasterTransaction

Constraint	Description
strobleValue	The constraint ensures that the number of active write strobes (hwstrb) matches the transfer size (hsize), setting 1, 2, 4, or 8 active bits for BYTE, HALFWORD, WORD, and DOUBLEWORD transfers, respectively.
burstsize	The constraint ensures that the hwdatasize corresponds to the burst type, setting 4, 8, 16, or 1 beats for WRAP4/INCR4, WRAP8/INCR8, WRAP16/INCR16, and other cases, respectively.
strobesize	The constraint ensures that hwstrb size matches the burst type, while the busyState constraint enforces the same size mapping for busyControl, setting values of 4, 8, or 16 for WRAP4/INCR4, WRAP8/INCR8, and WRAP16/INCR16, respectively.
busyState	The constraint ensures that the busyControl size corresponds to the burst type, assigning values 4, 8, or 16 for WRAP4/INCR4, WRAP8/INCR8, and WRAP16/INCR16, respectively.

- 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 AhbMasterTransaction signals.

```

function bit AhbMasterTransaction::do_compare (uvm_object rhs, uvm_comparer comparer);
    AhbMasterTransaction ahbMasterTransaction;

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

    return super.do_compare(ahbMasterTransaction, comparer) &&
        haddr == ahbMasterTransaction.haddr &&
        hselx == ahbMasterTransaction.hselx &&
        hburst == ahbMasterTransaction.hburst &&
        hmastlock == ahbMasterTransaction.hmastlock &&
        hprot == ahbMasterTransaction.hprot &&
        hsize == ahbMasterTransaction.hsize &&
        hnonsec == ahbMasterTransaction.hnonsec &&
        hexcl == ahbMasterTransaction.hexcl &&
        hmaster == ahbMasterTransaction.hmaster &&
        htrans == ahbMasterTransaction.htrans &&
        hwdata == ahbMasterTransaction.hwdata &&
        hwstrb == ahbMasterTransaction.hwstrb &&
        hwrite == ahbMasterTransaction.hwrite &&
        hrdata == ahbMasterTransaction.hrdata &&
        hreadyout == ahbMasterTransaction.hreadyout &&
        hresp == ahbMasterTransaction.hresp &&
        hexokay == ahbMasterTransaction.hexokay &&
        hready == ahbMasterTransaction.hready &&
        noOfWaitStatesDetected == ahbMasterTransaction.noOfWaitStatesDetected;

endfunction : do_compare

```

Fig 9.4 do_compare method of Master Transaction

```

function void AhbMasterTransaction::do_copy (uvm_object rhs);
    AhbMasterTransaction ahbMasterTransaction;

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

    haddr = ahbMasterTransaction.haddr;
    hselx = ahbMasterTransaction.hselx;
    hburst = ahbMasterTransaction.hburst;
    hmastlock = ahbMasterTransaction.hmastlock;
    hprot = ahbMasterTransaction.hprot;
    hsize = ahbMasterTransaction.hsize;
    hnonsec = ahbMasterTransaction.hnonsec;
    hexcl = ahbMasterTransaction.hexcl;
    hmaster = ahbMasterTransaction.hmaster;
    htrans = ahbMasterTransaction.htrans;
    hwdata = ahbMasterTransaction.hwdata;
    hwstrb = ahbMasterTransaction.hwstrb;
    hwrite = ahbMasterTransaction.hwrite;
    hrdata = ahbMasterTransaction.hrdata;
    hreadyout = ahbMasterTransaction.hreadyout;
    hresp = ahbMasterTransaction.hresp;
    hexokay = ahbMasterTransaction.hexokay;
    hready = ahbMasterTransaction.hready;
    noOfWaitStatesDetected = ahbMasterTransaction.noOfWaitStatesDetected;

endfunction : do_copy

```

Fig 9.5 do_copy method of Master Transaction

9.3.2 AhbSlaveTransaction

- AhbSlaveTransaction class is extended from the uvm_sequence_item holds the data items required to drive stimulus to dut
- Declared all the variables (haddr, hselx, hburst, hmastlock, hprot, hsize, hnonsec, hmaster, htrans, hwdata, hwstrb, hwrite, hrdata, hreadyout, hresp, hexokay, hready, hready, noOfWaitStatesDetected, busyControl)

```
constraint chooseDataPacketC1 {soft choosePacketData==0;},
constraint readDataSize{hrdata.size() == 16;}
|
constraint waitState{soft noOfWaitStates == 0;}
```

Fig 9.6 Constraints of Ahb Slave transaction

Table 12: Constraints of Ahb Slave transaction

Constraint	Description
chooseDataPacketC1	This constraint, chooseDataPacketC1, ensures that the variable choosePacketData is softly assigned the value 0 by default
readDataSize	This constraint, ensures that the size of the hrdata array is always 16. This means that any read operation must retrieve exactly 16 data elements, enforcing a fixed data width for read transactions
waitState	This constraint, waitState, defines a soft condition on noOfWaitStates such that its default or preferred value is 0


```

function bit AhbSlaveTransaction::do_compare(uvm_object rhs, uvm_comparer comparer);
    AhbSlaveTransaction ahbSlaveTransaction;

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

    return super.do_compare(ahbSlaveTransaction, comparer) &&
    haddr    == ahbSlaveTransaction.haddr    &&
    hselx    == ahbSlaveTransaction.hselx    &&
    hburst   == ahbSlaveTransaction.hburst   &&
    hmastlock == ahbSlaveTransaction.hmastlock &&
    hprot    == ahbSlaveTransaction.hprot    &&
    hsize    == ahbSlaveTransaction.hsize    &&
    hnonsec  == ahbSlaveTransaction.hnonsec  &&
    hexcl    == ahbSlaveTransaction.hexcl    &&
    hmaster  == ahbSlaveTransaction.hmaster  &&
    htrans   == ahbSlaveTransaction.htrans   &&
    hwdata   == ahbSlaveTransaction.hwdata   &&
    hwstrb   == ahbSlaveTransaction.hwstrb   &&
    hwrite   == ahbSlaveTransaction.hwrite   &&
    hrdata   == ahbSlaveTransaction.hrdata   &&
    hreadyout == ahbSlaveTransaction.hreadyout &&
    hresp    == ahbSlaveTransaction.hresp    &&
    hexokay  == ahbSlaveTransaction.hexokay  &&
    hready   == ahbSlaveTransaction.hready;
endfunction : do_compare

```

Fig 9.7 do_compare method of Slave Transaction

```

function void AhbSlaveTransaction::do_copy(uvm_object rhs);
    AhbSlaveTransaction ahbSlaveTransaction;

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

    haddr    = ahbSlaveTransaction.haddr;
    hselx    = ahbSlaveTransaction.hselx;
    hburst   = ahbSlaveTransaction.hburst;
    hmastlock = ahbSlaveTransaction.hmastlock;
    hprot    = ahbSlaveTransaction.hprot;
    hsize    = ahbSlaveTransaction.hsize;
    hnonsec  = ahbSlaveTransaction.hnonsec;
    hexcl    = ahbSlaveTransaction.hexcl;
    hmaster  = ahbSlaveTransaction.hmaster;
    htrans   = ahbSlaveTransaction.htrans;
    hwdata   = ahbSlaveTransaction.hwdata;
    hwstrb   = ahbSlaveTransaction.hwstrb;
    hwrite   = ahbSlaveTransaction.hwrite;
    hrdata   = ahbSlaveTransaction.hrdata;
    hreadyout = ahbSlaveTransaction.hreadyout;
    hresp    = ahbSlaveTransaction.hresp;
    hexokay  = ahbSlaveTransaction.hexokay;
    hready   = ahbSlaveTransaction.hready;
endfunction : do_copy

```

Fig 9.8 do_copy method of Slave Transaction

9.4 Sequences

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

9.4.1 Methods

Table 13: Sequence methods

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

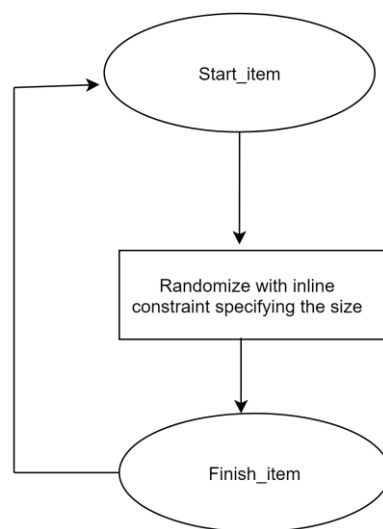


Fig 9.9 Flow chart for sequence methods

Table 14: Describing master and slave sequences

Sections	Master sequences	Slave sequences	Description
BaseSequence	AhbMasterBaseSequence	AhbSlaveBaseSequence	Base class is extended from uvm_sequence and parameterized with transaction (AhbMasterTransaction, AhbSlaveTransaction)
Data transfers	AhbMasterSequence	AhbSlaveSequence	Extended from base sequence. Based on a request from the driver, the task will drive the transactions. In between start_item and finish_item, the task body randomizes the transaction object (req) with inline constraints, assigning values from sequence variables, and reports a fatal error if randomization fails.

In the master sequence body, req is created, and start_item(req) initiates the sequence. The transaction (req) is then randomized with inline constraints, assigning values from sequence variables (master signals), followed by finish_item(req) to complete the sequence.

```

task AhbMasterSequence::body();
    super.body();
    req = AhbMasterTransaction::type_id::create("req");

    start_item(req);
    `uvm_info("AHB", $sformatf("req is of type: %s", req.get_type_name()), UVM_LOW)

    if (!req.randomize() with { haddr      == haddrSeq;
                               hselx      == hselxSeq;
                               hburst     == hburstSeq;
                               hmastlock  == hmastlockSeq;
                               hprot      == hprotSeq;
                               hsize      == hsizeSeq;
                               hnonsec    == hnonsecSeq;
                               hexcl      == hexclSeq;
                               htrans     == htransSeq;
                               hwrite     == hwriteSeq;
                               hexokay    == hexokaySeq;
                               foreach(hwdataSeq[i])
                                   hwdata[i] == hwdataSeq[i];
                               foreach(hwstrbSeq[i])
                                   hwstrb[i] == hwstrbSeq[i];
                               foreach(busyControlSeq[i])
                                   busyControl[i] == busyControlSeq[i];
                               }) begin
        `uvm_fatal("AHB", "Randomization failed for 32-bit Write")
    end
    finish_item(req);
endtask: body

```

Fig 9.10 Master sequence body method

```

constraint haddr_alignment1 {
    if (hsizeSeq == HALFWORD) {
        haddrSeq[0] == 1'b0;
    } else if (hsizeSeq == WORD) {
        haddrSeq[1:0] == 2'b00;
    } else if (hsizeSeq == DOUBLEWORD) {
        haddrSeq[2:0] == 3'b000;
    } else if (hsizeSeq == LINE4) {
        haddrSeq[3:0] == 4'b0000;
    } else if (hsizeSeq == LINE8) {
        haddrSeq[4:0] == 5'b00000;
    } else if (hsizeSeq == LINE16) {
        haddrSeq[5:0] == 6'b000000;
    } else if (hsizeSeq == LINE32) {
        haddrSeq[6:0] == 7'b0000000;
    }
}

constraint first_trans_tpy1 {
    if (hburstSeq == SINGLE) {
        soft htransSeq inside {IDLE, NONSEQ};
    } else {
        soft htransSeq == NONSEQ;
    }
}

constraint incr_trans_tpy1 {
    if (hburstSeq != SINGLE) {
        if (htransSeq == IDLE)
            soft htransSeq == NONSEQ;
        else
            soft htransSeq == SEQ;
    }
}

constraint hselx_logic1 {
    if (htransSeq == IDLE)
        soft hselxSeq == '0;
    else
        $onehot(hselxSeq);
}

```

Fig 9.11 Constraints Of Master Sequence

In the slave sequence body, req is created, and start_item(req) initiates the sequence. The transaction (req) is then randomized with inline constraints, assigning values from sequence variables (slave signals), followed by finish_item(req) to complete the sequence.

```

function AhbSlaveSequence::new(string name="AhbSlaveSequence");
    super.new(name);
endfunction : new

task AhbSlaveSequence::body();
    req = AhbSlaveTransaction::type_id::create("req");
    start_item(req);
    if(!req.randomize() with { foreach(hrdataSeq[i])
        hrdata[i] == hrdataSeq[i];
        hreadyout == hreadyoutSeq;
        hexokay == hexokaySeq;
        choosePacketData == choosePacketDataSeq;
        noOfWaitStates == noOfWaitStatesSeq;
    }) begin
        `uvm_fatal("AHB","Rand failed");
    end
    req.print();
    finish_item(req);
endtask : body

```

Fig 9.12 Slave sequence body method

9.5 Virtual sequences

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

9.5.1 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 AhbVirtualBaseSequence extends uvm_sequence;
    `uvm_object_utils(AhbVirtualBaseSequence)
    `uvm_declare_p_sequencer(AhbVirtualSequencer)

    AhbEnvironmentConfig ahbEnvironmentConfig;

    extern function new(string name = "AhbVirtualBaseSequence");
    extern task body();

endclass : AhbVirtualBaseSequence

```

Fig 9.13 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 AhbVirtualBaseSequence::body();
    if (!uvm_config_db#(AhbEnvironmentConfig)::get(null, get_full_name(), "AhbEnvironmentConfig", ahbEnvironmentConfig )) begin
        `uvm_fatal("AHBENVIRONMENTCONFIG", "cannot get() ENV_cfg from uvm_config_db.Have you set() it?")
    end

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

Fig 9.14 Virtual base sequence body

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

```
task AhbVirtualSingleWriteSequence::body();
    super.body();
    ahbMasterSequence = AhbMasterSequence::type_id::create("ahbMasterSequence");
    ahbSlaveSequence = AhbSlaveSequence::type_id::create("ahbSlaveSequence");
    repeat(40) begin
        if(!ahbMasterSequence.randomize() with {
            hsizeSeq dist {BYTE:=1, HALFWORD:=1, WORD:=1};
            hwriteSeq ==1;
            htransSeq == NONSEQ;
            hburstSeq == SINGLE;
            foreach(busyControlSeq[i]) busyControlSeq[i] dist {0:=100, 1:=0};
        }) begin
            `uvm_error(get_type_name(), "Randomization failed : Inside AhbVirtualSingleWriteSequence")
        end
        fork
            ahbSlaveSequence.start(p_sequencer.ahbSlaveSequencer);
            ahbMasterSequence.start(p_sequencer.ahbMasterSequencer);
        join
    end
endtask : body
```

Fig 9.15 Virtual Single Write sequence body

Table 15: Describing virtual sequences

Virtual sequences	Description
AhbVirtualSingleWriteSequence	Inside the Single Write Virtual Sequence, extending from base class. Declaring handles for master and slave sequences, Using inline constraints to randomize the master sequence with BYTE, HALFWORD, and WORD transfer sizes, enforcing a write operation, NONSEQ transfer type, and SINGLE burst type.

AhbVirtualSingleReadSequence	Inside the Single Write Virtual Sequence, extending from base class. Declaring handles for master and slave sequences. Using inline constraints to randomize the master sequence with BYTE, HALFWORD, and WORD transfer sizes, enforcing a read operation, NONSEQ transfer type, and SINGLE burst type.
AhbVirtualWriteSequence	Inside the Write Virtual Sequence, extending from base class. Declaring handles for master and slave sequences. Using inline constraints to randomize the master sequence with BYTE, HALFWORD, and WORD transfer sizes, enforcing a write operation, NONSEQ transfer type, and distributing the burst type across multiple values like WRAP4, INCR4, WRAP8, INCR8, WRAP16, INCR16.
AhbVirtualReadSequence	Inside the Read Virtual Sequence, extending from base class. Declaring handles for master and slave sequences. Using inline constraints to randomize the master sequence with BYTE, HALFWORD, and WORD transfer sizes, enforcing a read operation, NONSEQ transfer type, and distributing the burst type across multiple values like WRAP4, INCR4, WRAP8, INCR8, WRAP16, INCR16.
AhbVirtualWriteWithBusySequence	Inside the Write Virtual Sequence, extending from base class. Declaring and constructing master and slave sequence handles. Using inline constraints to randomize the master sequence with BYTE, HALFWORD, and WORD transfer sizes, enforcing a write operation, NONSEQ transfer type, and distributing the burst type across multiple values like WRAP4, INCR4, WRAP8, INCR8, WRAP16, INCR16. Additionally, busyControlSeq is randomized to introduce bus busy conditions
AhbVirtualReadWithBusySequence	Inside the Read Virtual Sequence, extending from base class. Declaring and constructing master and slave sequence handles. Using inline constraints to randomize the master sequence with BYTE, HALFWORD, and WORD transfer sizes, enforcing a read operation, NONSEQ transfer type, and distributing the burst type across multiple values like WRAP4, INCR4, WRAP8, INCR8, WRAP16, INCR16. Additionally, busyControlSeq is randomized to introduce bus busy conditions
AhbSingleVirtualWriteWithWaitStateSequence	Inside the Write with wait state Virtual Sequence, extending from base class. Declaring constructing master and slave sequence handles. Using inline constraints to randomize the master sequence with BYTE, HALFWORD, and WORD transfer sizes, enforcing a write operation, NONSEQ transfer type and SINGLE burst type.
AhbSingleVirtualReadWithWaitStateSequence	Inside the Read with wait state Virtual Sequence, extending from base class. Declaring constructing master and slave sequence handles. Using inline constraints to randomize the master sequence with BYTE, HALFWORD, and WORD transfer sizes, enforcing a read operation, NONSEQ transfer type and SINGLE burst type.

AhbVirtualWriteWithWaitState Sequence	Inside the Write with wait state Virtual Sequence, extending from base class. Declaring constructing master and slave sequence handles. Using inline constraints to randomize the master sequence with BYTE, HALFWORD, and WORD transfer sizes, enforcing a write operation, NONSEQ transfer type, and distributing the burst type across multiple values like WRAP4, INCR4, WRAP8, INCR8, WRAP16, INCR16.
AhbVirtualReadWithWaitState Sequence	Inside the Read with wait state Virtual Sequence, extending from base class. Declaring constructing master and slave sequence handles. Using inline constraints to randomize the master sequence with BYTE, HALFWORD, and WORD transfer sizes, enforcing a read operation, NONSEQ transfer type, and distributing the burst type across multiple values like WRAP4, INCR4, WRAP8, INCR8, WRAP16, INCR16.
AhbWriteFollowedByReadVirtualSequence	Inside the Write Followed by Read Virtual Sequence, extending from base class. Declaring handles for master and slave sequences, Using inline constraints to randomize the master sequence with BYTE, HALFWORD, and WORD transfer sizes, enforcing a write operation followed by read operation, NONSEQ transfer type, and SINGLE burst type.

9.6 Test Cases

The uvm_test class defines the test scenario and verification goals.

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

```

class AhbBaseTest extends uvm_test;
  `uvm_component_utils(AhbBaseTest)

  AhbEnvironment ahbEnvironment;

  AhbEnvironmentConfig ahbEnvironmentConfig;

  extern function new(string name = "AhbBaseTest", uvm_component parent = null);
  extern virtual function void build_phase(uvm_phase phase);
  extern virtual function void setupAhbEnvironmentConfig();
  extern virtual function void setupAhbMasterAgentConfig();
  extern virtual function void setupAhbSlaveAgentConfig();
  extern virtual function void end_of_elaboration_phase(uvm_phase phase);
  extern virtual task run_phase(uvm_phase phase);

endclass : AhbBaseTest

```

Fig 9.16 Base test

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

C) Inside setupAhbEnvironmentConfig function, constructing the environment config class handle. With the help of this ahbEnvironmentConfig handle all the required fields in the

config class have been set up with respective values and then calling the setupAhbMasterAgentConfig and setupAhbSlaveAgentConfig functions.

```
function void AhbBaseTest::setupAhbEnvironmentConfig();
ahbEnvironmentConfig = AhbEnvironmentConfig::type_id::create("ahbEnvironmentConfig");
ahbEnvironmentConfig.noOfSlaves      = NO_OF_SLAVES;
ahbEnvironmentConfig.noOfMasters     = NO_OF_MASTERS;
ahbEnvironmentConfig.hasScoreboard   = 1;
ahbEnvironmentConfig.hasVirtualSequencer = 1;

ahbEnvironmentConfig.ahbMasterAgentConfig = new[ahbEnvironmentConfig.noOfMasters];
foreach(ahbEnvironmentConfig.ahbMasterAgentConfig[i]) begin
    ahbEnvironmentConfig.ahbMasterAgentConfig[i] = AhbMasterAgentConfig::type_id::create($sformatf("AhbMasterAgentConfig[%0d]",i));
end
setupAhbMasterAgentConfig();

foreach(ahbEnvironmentConfig.ahbMasterAgentConfig[i]) begin
    uvm_config_db #(AhbMasterAgentConfig)::set(this,"*", $sformatf("AhbMasterAgentConfig[%0d]",i), ahbEnvironmentConfig.ahbMasterAgentConfig[i]);
    `uvm_info(get_type_name(), $sformatf("\nAHB_MASTER_CONFIG[%0d]\n%s", i, ahbEnvironmentConfig.ahbMasterAgentConfig[i].sprint()), UVM_LOW);
end

ahbEnvironmentConfig.ahbSlaveAgentConfig = new[ahbEnvironmentConfig.noOfSlaves];
foreach(ahbEnvironmentConfig.ahbSlaveAgentConfig[i]) begin
    ahbEnvironmentConfig.ahbSlaveAgentConfig[i] = AhbSlaveAgentConfig::type_id::create($sformatf("AhbSlaveAgentConfig[%0d]",i));
end

setupAhbSlaveAgentConfig();

foreach(ahbEnvironmentConfig.ahbSlaveAgentConfig[i]) begin
    uvm_config_db #(AhbSlaveAgentConfig)::set(this,"*", $sformatf("AhbSlaveAgentConfig[%0d]",i), ahbEnvironmentConfig.ahbSlaveAgentConfig[i]);
    `uvm_info(get_type_name(), $sformatf("\nAHB_SLAVE_CONFIG[%0d]\n%s", i, ahbEnvironmentConfig.ahbSlaveAgentConfig[i].sprint()), UVM_LOW);
end

uvm_config_db#(AhbEnvironmentConfig)::set(this,"*", "AhbEnvironmentConfig", ahbEnvironmentConfig);
`uvm_info(get_type_name(), $sformatf("\nAHB_ENV_CONFIG\n%s", ahbEnvironmentConfig.sprint()), UVM_LOW);

endfunction : setupAhbEnvironmentConfig
```

Fig 9.17 Setup Environment Config

In setupAhbMasterAgentConfig function, AhbMasterAgentConfig class handle which is in AhbEnvironmentConfig class has been constructed with the help of this handle all the required fields(hasCoverage, is_active) in AhbMasterAgentConfig class has been setup.

```
function void AhbBaseTest::setupAhbMasterAgentConfig();

foreach(ahbEnvironmentConfig.ahbMasterAgentConfig[i]) begin
    if(MASTER_AGENT_ACTIVE == 1) begin
        ahbEnvironmentConfig.ahbMasterAgentConfig[i].is_active = uvm_active_passive_enum'(UVM_ACTIVE);
    end
    else begin
        ahbEnvironmentConfig.ahbMasterAgentConfig[i].is_active = uvm_active_passive_enum'(UVM_PASSIVE);
    end
    ahbEnvironmentConfig.ahbMasterAgentConfig[i].hasCoverage = 1;
end

endfunction : setupAhbMasterAgentConfig
```

Fig 9.18 Master Agent Config setup

D) In setupAhbSlaveAgentConfigfunction, for each slave agent configuration trying to construct AhbSlaveAgentConfig class handle which is in AhbEnvironmentConfig class with the help of this handle all the required fields (hasCoverage, is_active) in AhbSlaveAgentConfig class has been setup followed by the end of the elaboration phase used to print the topology.

```
function void AhbBaseTest::setupAhbSlaveAgentConfig();
|
|   foreach(ahbEnvironmentConfig.ahbSlaveAgentConfig[i]) begin
|       if(SLAVE_AGENT_ACTIVE === 1) begin
|           ahbEnvironmentConfig.ahbSlaveAgentConfig[i].is_active = uvm_active_passive_enum'(UVM_ACTIVE);
|       end
|       else begin
|           ahbEnvironmentConfig.ahbSlaveAgentConfig[i].is_active = uvm_active_passive_enum'(UVM_PASSIVE);
|       end
|       ahbEnvironmentConfig.ahbSlaveAgentConfig[i].hasCoverage = 1;
|   end
endfunction : setupAhbSlaveAgentConfig
```

Fig 9.19 Slave Agent Config setup

Extend the AhbSingleWrite from base test and declare virtual sequence handle then create virtual sequence in test, and start the virtual sequence in run_phase, raise and drop objection.

```
class AhbSingleWriteTest extends AhbBaseTest;
`uvm_component_utils(AhbSingleWriteTest)

    AhbVirtualSingleWriteSequence ahbVirtualSingleWriteSequence;

extern function new(string name = "AhbSingleWriteTest", uvm_component parent = null);
extern virtual task run_phase(uvm_phase phase);

endclass : AhbSingleWriteTest
```

Fig 9.20 Example for Single Write test

```

task AhbSingleWriteTest::run_phase(uvm_phase phase);

    ahbVirtualSingleWriteSequence = AhbVirtualSingleWriteSequence::type_id::create("ahbVirtualSingleWriteSequence");
    `uvm_info(get_type_name(), $sformatf("AhbSingleWriteTest"), UVM_LOW);
    phase.raise_objection(this);
    ahbVirtualSingleWriteSequence.start(ahbEnvironment.ahbVirtualSequencer);
    #10;
    phase.drop_objection(this);

endtask : run_phase

`endif

```

Fig 9.21 Run phase of Single Write test

Table 16: Tests

Test names	Description
AhbSingleWriteTest	Extend test from base test and created the Single Write virtual sequence handle and starting the sequences in between phase raise and drop objection.
AhbSingleReadTest	Extend test from base test and created the Single Read virtual sequence handle and starting the sequences in between phase raise and drop objection.
AhbWriteTest	Extend test from base test and created the Write virtual sequence handle and starting the sequences in between phase raise and drop objection.
AhbReadTest	Extend test from base test and created the Read virtual sequence handle and starting the sequences in between phase raise and drop objection.
AhbWriteWithBusyTest	Extend test from base test and created the Write with Busy virtual sequence handle and starting the sequences in between phase raise and drop objection.
AhbReadWithBusyTest	Extend test from base test and created the Read with Busy virtual sequence handle and starting the sequences in between phase raise and drop objection.
AhbWriteWithWaitStateTest	Extend test from base test and created the Write with wait state virtual sequence handle and starting the sequences in between phase raise and drop objection.

AhbReadWithWaitStateTest	Extend test from base test and created the Read with wait state virtual sequence handle and starting the sequences in between phase raise and drop objection.
AhbWriteFollowedByReadTest	Extend test from base test and created the Single Write virtual sequence handle and Single Read virtual sequence handle and starting the sequences in between phase raise and drop objection.

9.7 Testlists

Regression list for AHB

Table 17: Testlists

TestCase Names	Description
AhbSingleWriteTest	Checks for a Single Transfer Write operation
AhbSingleReadTest	Checks for a Single Transfer Read operation
AhbWriteTest	Checks for Other Burst Transfers Write operation
AhbReadTest	Checks for Other Burst Transfers Read operation
AhbWriteWithBusyTest	Check for Busy Trans for Write Operation
AhbReadWithBusyTest	Check for Busy Trans for Read Operation
AhbSingleWriteWithWaitStateTest	Check for Wait state between transfers for Write Operation of Single Burst Transfer
AhbSingleReadWithWaitStateTest	Check for Wait state between transfers for Read Operation of Single Burst Transfer
AhbWriteWithWaitStateTest	Check for Wait state between transfers for Write Operation
AhbReadWithWaitStateTest	Check for Wait state between transfers for Read Operation
AhbWriteFollowedByReadTest	Checks for a Single Transfer Write operation followed by Read operation

Chapter 10

Simulation Results and Waveform

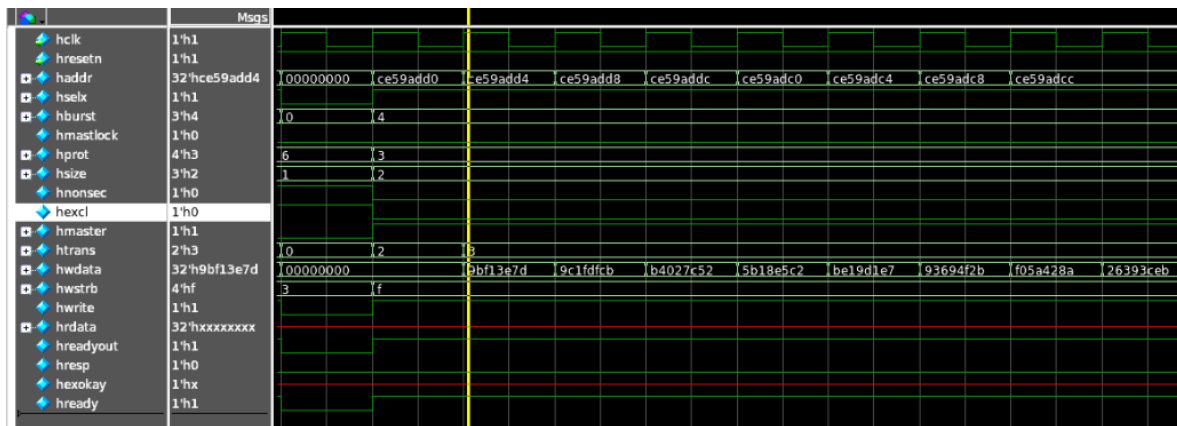


Fig 10.1 AhbWriteTest

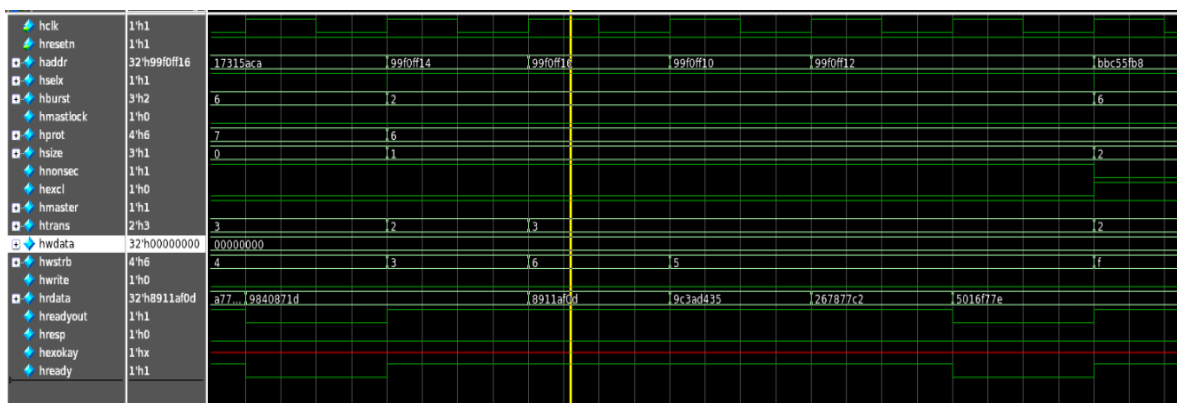


Fig 10.2 *AhbReadTest*

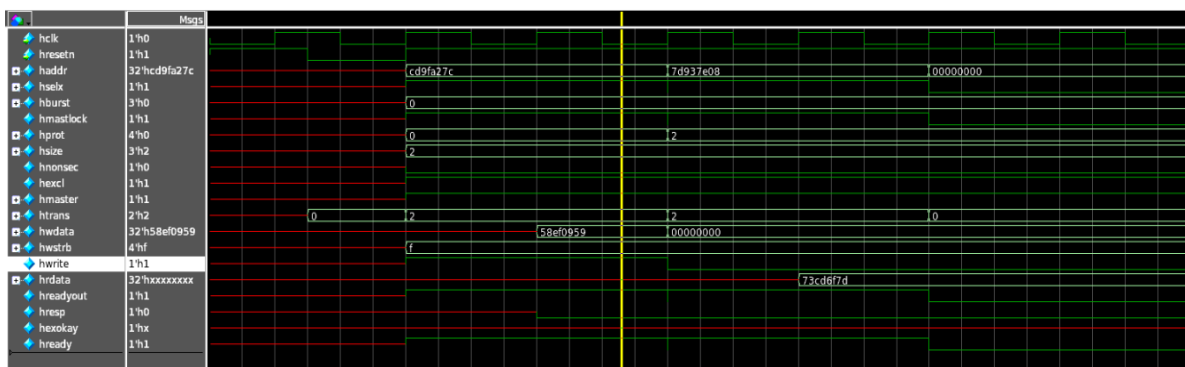


Fig 10.3 *AhbWriteFollowedByReadTest*

References

[AHB Protocol Specification](#)