2021-22

AXI4-AVIP

Contents

List of Tables

List of Figures

Table 7.1 Assertion Table

Assertion Label	Description
AXI_WA_STABLE_SIGNALS_CHECK	All signals must remain stable after **AWVALID** is asserted until **AWREADY** IS LOW
AXI_WA_UNKNOWN_SIGNALS_CHECK	A value of X on signals is not permitted when **AWVALID** is HIGH
AXI_WA_VALID_STABLE_CHECK	When **AWVALID** is asserted, then it must remain asserted until **AWREADY** is HIGH

Chapter 1

INTRODUCTION

AMBA (Advanced Microcontroller Bus Architecture) is open standard for communication and management of the functional blocks in SoC, provide different on chip communication protocols like CHI (Coherence Hub Interface), AXI (Advanced eXtensible Interface), ACE (Advanced Coherency Extension), AHB (Advanced High Performance Bus), APB (Advanced Peripheral Bus) developed by ARM (Advanced RISC Machine) . Flexibility of AMBA protocols is IP reuse for different SoC designs with different area, power and performance requirements.

As the AMBA protocols are widely used open standards which ensures compatibility between IPs of different suppliers for the SoC , with compatibility it enables low friction integration and reuse of IP which catalyse the faster time to market. The AMBA AXI protocol specification is defined to implement a high frequency, high bandwidth interface across a wide variety of applications in embedded, automotive and cellphones. It does not require complex bridge implementation for different peripheral devices. The AXI protocol includes some new features which extend previous versions and is compatible to complement CHI.

There are 3 types of AXI4-Interfaces (AMBA 4.0):

- AXI4 (Full AXI4): For high-performance memory-mapped requirements.
- AXI4-Lite: For simple, low-throughput memory-mapped communication (for example, to and from control and status registers).
- AXI4-Stream: For high-speed streaming data

1.1 AXI Read and Write Channels

The AXI protocol defines 5 channels:

- 2 are used for Read transactions
    - read address
    - read data
- 3 are used for Write transactions
    - Write address
    - Write data
    - Write response

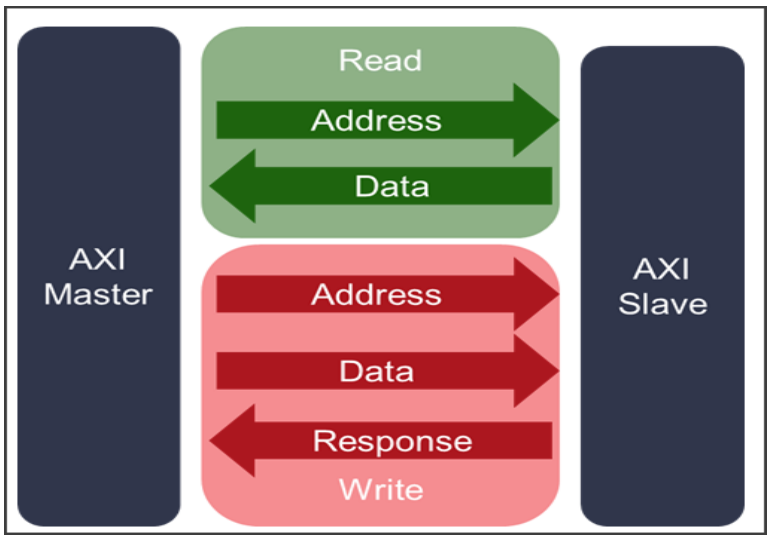


Fig 1.1 AXI Read and write Channels

The AXI is a burst-based protocol that defines the following transaction channels independently:

1. Address Read(AR)
2. Read data(R)
3. Address Write(AW)
4. Write data(W)
5. Write response(B)

Control information about the kind of data to be delivered is carried by an address channel. AXI protocol entails the following steps:

1. Enables the distribution of address information before the actual data transmission
2. Supports a large number of open transactions
3. Allows transactions to be completed out of order

A channel is an independent collection of AXI signals associated with the VALID and READY signals.

Note: An AXI4/AXI3/AXI4-Lite Interface can be read only (only includes the 2 Read channels) or write only (only includes the 3 Write channels).

A piece of data transmitted on a single channel is called a transfer. A transfer happens when both the VALID and READY signals are high while there is a rising edge of the clock.

1.2 AXI Read Transactions

An AXI Read transaction requires multiple transfers on the 2 Read channels.

- First, the **Address Read Channel** is sent from the Master to the Slave to set the address and some control signals.

- Then the data for this address is transmitted from the Slave to the Master on the **Read data channel**.

Note that, as per the figure below, there can be multiple data transfers per address. This type of transaction is called a **burst**.

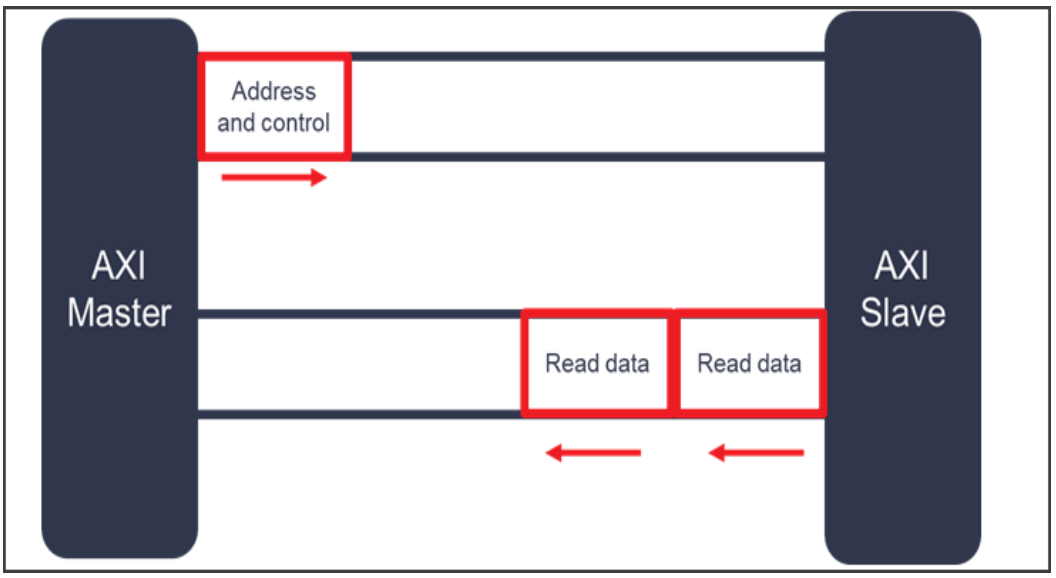


Fig 1.2 Read Channels of AXI

1.3 AXI Write Transactions

An AXI Write transaction requires multiple transfers on the 3 Read channels.

- First, the **Address Write Channel** is sent Master to the Slave to set the address and some control signals.
- Then the data for this address is transmitted Master to the Slave on the **Write data channel**.
- Finally the write response is sent from the Slave to the Master on the **Write Response Channel** to indicate if the transfer was successful.

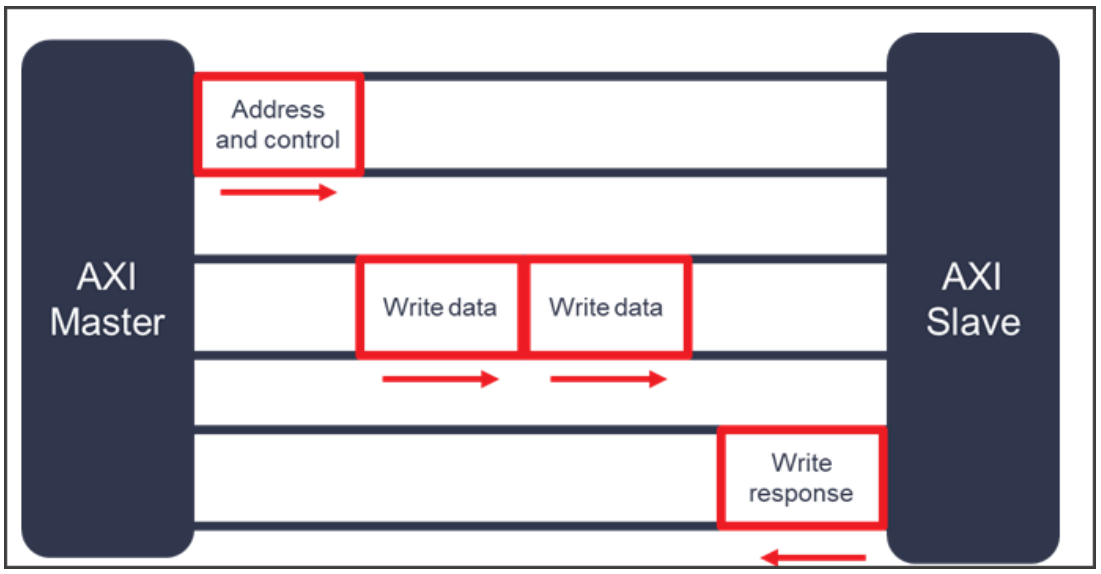


Fig 1.3 Write Channels of AXI

The possible response values on the Write Response Channel are:

- OKAY (0b00): Normal access success. Indicates that a normal access has been successful
- EXOKAY (0b01): Exclusive access okay.
- SLVERR (0b10): Slave error. The slave was reached successfully but the slave wishes to return an error condition to the originating master (for example, data read not valid).
- DECERR (0b11): Decode error. Generated, typically by an interconnect component, to indicate that there is no slave at the transaction address

Note: Read transactions also have a response value but this response is transmitted as part of the Read Response Channel

1.4 Interface Signal Definition

Table 1.1: Write Address Signals

Signal	Source	Supporting Version	Definition of Signal
AWID[x:0]	Manager / Master	AXI3 and AXI4	Provides ID for each transaction
AWADDR[31:0]	Manager / Master	AXI3 and AXI4	Address for write request
AWLEN[7:0]	Manager / Master	AXI4	Number of transfers support in each transaction
AWSIZE[2:0]	Manager / Master	AXI3 and AXI4	No. of bytes to be transfer in each beat
AWBURST[1:0]	Manager / Master	AXI3 and AXI4	Indicates type of burst to be performed
AWLOCK	Manager / Master	AXI4	Indicates the atomic characteristics of the transaction
AWCACHE[3:0]	Manager / Master	AXI3 and AXI4	This signal indicates the system performance
AWPROT[2:0]	Manager / Master	AXI3 and AXI4	Provides system level security and privileged access to each transaction.
AWQoS[3:0]	Manager / Master	AXI4	Use to prioritise the transactions
AWREGION[3:0]	Manager / Master	AXI4	Region identifier
AWVALID	Manager / Master	AXI3 and AXI4	Use to validate the associated signal inorder to pass valid information.

AWREADY	Subordinat e/slave	AXI3 and AXI4	Indicates weather slave is ready for the transactions.

Table 1.2: Write data Signals

Signal	Source	Supporting Version	Definition of Signal
AWDATA[x:0]	Manager / Master	AXI3 and AXI4	Write data signal
AWSTRB[x:0]	Manager / Master	AXI3 and AXI4	Used to send valid bytes for each transfer
AWLAST	Manager / Master	AXI3 and AXI4	Indicates last transfer for the transaction
AWVALID	Manager / Master	AXI3 and AXI4	Used to validate the associated signal in order to pass valid information.
AWREADY	Subordinate/ slave	AXI3 and AXI4	Indicates weather slave is ready for the transactions.

Table 1.3 : Write Response Signals

Signal	Source	Supporting Version	Definition of Signal
BID[x:0]	Subordinate/slave	AXI3 and AXI4	Provides ID for each write transaction
BRSEP[1:0]	Subordinate/slave	AXI3 and AXI4	Write Response signal
BVALID	Subordinate/slave	AXI3 and AXI4	Use to validate the associated signal in order to pass valid information.
BREADY	Manager / Master	AXI3 and AXI4	Indicates weather master is ready for the transactions.

Table 1.4 : Read Address Signals

Signal	Source	Supporting Version	Definition of Signal
ARID[x:0]	Manager / Master	AXI3 and AXI4	Provides ID for each transaction
ARADDR[31:0]	Manager / Master	AXI3 and AXI4	Address for write request
ARLEN[7:0]	Manager / Master	AXI4	No.of transfers support in each transaction
ARSIZE[2:0]	Manager / Master	AXI3 and AXI4	No. of bytes to be transfer in each beat
ARBURST[1:0]	Manager / Master	AXI3 and AXI4	Indicates type of burst to be performed
ARLOCK	Manager / Master	AXI4	Indicates the atomic characteristics of the transaction
ARCACHE[3:0]	Manager / Master	AXI3 and AXI4	This signal indicates the system performance
ARPROT[2:0]	Manager / Master	AXI3 and AXI4	Provides system level security and privileged access to each transaction.
ARQoS[3:0]	Manager / Master	AXI4	Use to prioritise the transactions
ARREGION[3:0]	Manager / Master	AXI4	Region identifier
ARVALID	Manager / Master	AXI3 and AXI4	Use to validate theassociated signal inorder to pass valid information.
ARREADY	Subordinate/slave	AXI3 and AXI4	Indicates weather slave is ready for thetransactions.

Table 1.5 : Read data Signals

Signal	Source	Supporting Version	Definition of Signal
RID[x:0]	Subordinate/ slave	AXI3 and AXI4	Provides ID for each read transaction
RDATA[x:0]	Subordinate/ slave	AXI3 and AXI4	Read data signal
RESP[1:0]	Subordinate/ slave	AXI3 and AXI4	Read response signal
RLAST	Subordinate/ slave	AXI3 and AXI4	Indicates last transfer for the transaction
RVALID	Subordinate/ slave	AXI3 and AXI4	Use to validate the associated signal in order to pass valid information.
READY	Manager / Master	AXI3 and AXI4	Indicates weather master is ready for the transactions.

1.5 Key Features

1. Axi4 supports write and read in parallel
2. Blocking and Non Blocking Transfers
3. Separate address/control and data phases
4. Support for unaligned data transfers, using byte strobes
5. Uses burst-based transactions with only the start address issued
6. Support for issuing multiple outstanding addresses
7. Support for narrow transfers

TO_DO Key features :

1.Out-of-Order

2. Low-power features

1.6 Reset

Reset assertion is asynchronous to clock and deassertion is synchronous to clock

1.7 Write Channel Signal Descriptions :

Awid : Master - Write Address ID

1. Each transaction has its own id.

Awaddr : Master - Write Address

1. Write address for the first **transfer**.
2. Drives Address of the first-byte in the tx to the slave.
3. The slave must calculate the subsequent transfers in the burst.
4. **Burst must not cross a 4KB address boundary**.

Awlength : Master - Burst Length

1. Gives the exact number of transfers in a burst.
2. Burst_length = AWLEN + 1
3. AWLEN[7:0], for write transfers for INCR burst(new feature for AXI4)
4. AWLEN[3:0], for write transfers for FIXED, WRAP burst
5. AWLEN for different burst types :
    a. FIXED : 1 to 16
    b. INCR : 1 to 256 transfers
    c. WRAP : 2,4,8 or 16
6. Early termination of burst is not supported.
    a. Hence, master can reduce data transfer in write burst by deasserting pstobe.

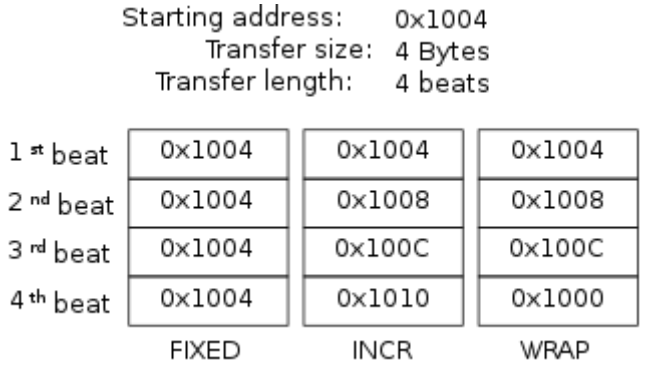


Fig 1.4 Burst types

Awsize : **Master - Size of each transfer**

1. Gives the size of each transfer in the burst.

AxSIZE[2:0]	Bytes in transfer
0b000	1
0b001	2
0b010	4

0b011	8
0b100	16
0b101	32
0b110	64
0b111	128

2. No of address bytes of transfer in the burst will be calculated as two powers of AWSIZE of Address Channel bus.
3. If the AXI bus is wider than the burst size, the AXI interface must determine from the transfer address which byte lanes of the data bus to use for each transfer. See Data read and write structure.
4. The size of any transfer must not exceed the data bus width of either agent in the transaction.

Awburst: MASTER - Type of transfer

fixed :

1. The start address is the same for all transfers in the burst.
2. The byte lanes still may differ based on the assertion of WSTRB
3. This burst type is used for repeated accesses to the same location such as when loading or emptying a FIFO.

Incr :

1. The address will be incremented for the next beat/transfer from the start address.
2. The increment will be based on the size of the transfer.

3. This burst type is used for access to normal sequential memory.

wrap :

1. A wrapping burst is similar to an incrementing burst, except that the address wraps around to a lower address if an upper address limit is reached.
2. Restrictions to wrap burst :
    a. Start address must be aligned to the size of each transfer.
    b. Length of bursts must be 2,4,8 or 16b transfers
3. Lowest address used by burst is aligned to the total size of the data to be transferred.
    a. Lowest address is the **wrap boundary**.
    b. **Wrap Boundary** = size of each transfer on the burst * total num of transfers
4. It increments till the wrap boundary from the start address, i.e.,
    a. **Incremented address** = wrap_boundary + total size of data to be transferred
5. The start address can be higher than the wrap boundary. This means that the address wraps for any WRAP burst for which the first address is higher than the wrap boundary.
6.

AxBURST[1:0]	**Burst type**
0b00	FIXED
0b01	INCR

0b10	WRAP
0b11	Reserved

Write Data channel

1. Data bus width : 8,16,32,64,128,256,512,1024 bits wide
2. Different bus width with for write and read data channels
3. Write strobe feature
4. Different combination of strobes
5. Valid scenario - 8bits data on 32 bits data bus
6. Invalid scenarios - 16bits data on 32 bits data bus but strobes like 111 - invalid

Note: The read channel doesn't require buffer because the data sent from slave with the ID will be routed correctly by the interconnect. Hence, no buffer on this side.

Write Response

1. Valid default value is 0, other signals can be anything
2. Ready can be low or high in default state.
3. It should be high, only if master can accept the response in 1 clock cycle - transfer in 1 clock cycle
4. If low, it takes 2 clock cycles min.
5. Response is driven only after the write-data transaction is completed.

Read Address:

1. Valid default value is 0, other signals can be anything
2. Ready can be low or high in default state.
3. If high, it should be able to accept the data.- transfer in 1 clock cycle
4. If low, it takes 2 clock cycles min. Hence, not recommended but based on implementation

Read Response:

1. Valid default value is 0, other signals can be anything
2. Ready can be low or high in default state.
3. If high, master should be able to accept the data.- transfer in 1 clock cycle
4. If low, it takes 2 clock cycles min.
5. Response, along with data, is driven only after the read address transaction is completed.

1.8 Handshake Process:

All five transaction channels use the same VALID/READY handshake process to transfer address, data, and control information. This two-way flow control mechanism means both the master and slave can control the rate at which the information moves between master and slave. The source generates the VALID signal to indicate when the address, data or control information is available. The destination generates the READY signal to indicate that it can accept the information. Transfer occurs only when both the VALID and READY signals are HIGH.

In Figure below the source presents the address, data or control information after T1 and asserts the VALID signal. The destination asserts the READY signal after T2, and the source must keep its information stable until the transfer occurs at T3, when this assertion is recognized.

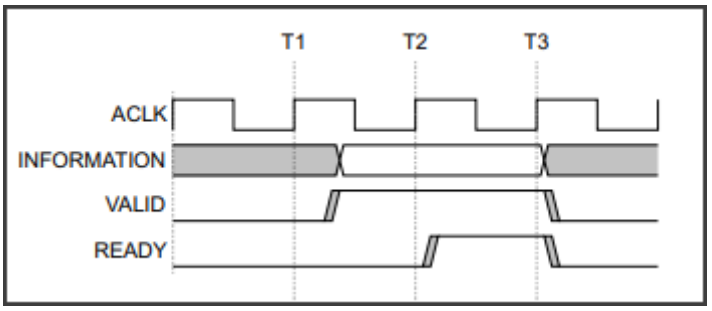


Fig 1.5 Valid before Ready

Once VALID is asserted it must remain asserted until the handshake occurs, at a rising clock edge at which VALID and READY are both asserted.

In Figure below , the destination asserts READY, after T1, before the address, data or control information is valid, indicating that it can accept the information. The source presents the information, and asserts VALID, after T2, and the transfer occurs at T3, when this assertion is recognized. In this case, transfer occurs in a single cycle

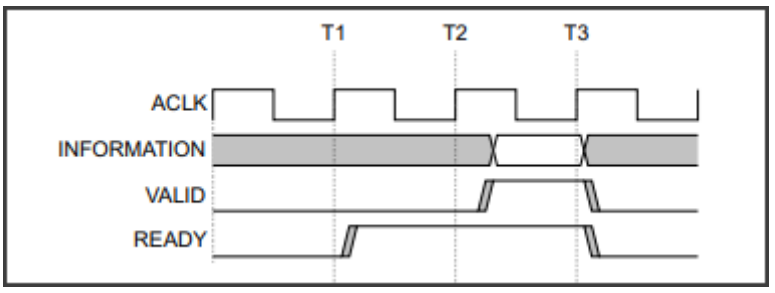


Fig 1.6 Ready before valid

A destination is permitted to wait for VALID to be asserted before asserting the serted, it is permitted to deassert REcorresponding READY. If READY is asADY before VALID is asserted.

In Figure below both the source and destination happen to indicate, after T1, that they can transfer the address, data or control information. In this case the transfer occurs at the rising clock edge when the assertion of both VALID and READY can be recognized. This means the transfer occurs at T2.

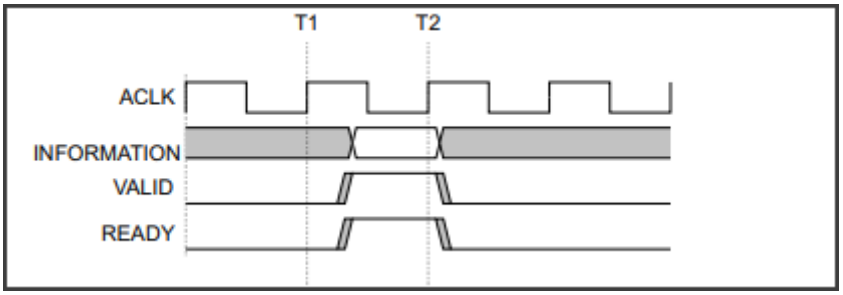


Fig 1.7 Ready with valid

1.9 Relationships between the channels:

Read transaction dependencies

Figure below shows the read transaction handshake signal dependencies, and shows that, in a read transaction:

1. The master must not wait for the slave to assert ARREADY before asserting ARVALID
2. The slave can wait for ARVALID to be asserted before it asserts ARREADY
3. The slave can assert ARREADY before ARVALID is asserted
4. The slave must wait for both ARVALID and ARREADY to be asserted before it asserts RVALID to indicate that valid data is available
5. The slave must not wait for the master to assert RREADY before asserting
6. The master can wait for RVALID to be asserted before it asserts RREADY
7. The master can assert RREADY before RVALID is asserted.

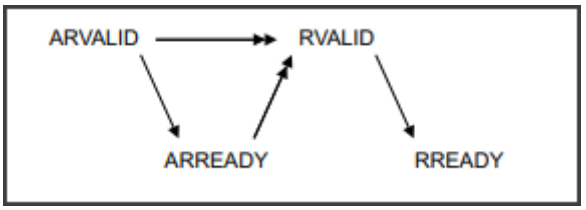


Fig 1.8 Read transaction dependencies

Write transaction dependencies

1. The master must not wait for the slave to assert AWREADY or WREADY before asserting AWVALID or WVALID
2. The slave can wait for AWVALID or WVALID, or both before asserting AWREADY
3. The slave can assert AWREADY before AWVALID or WVALID, or both, are asserted
4. The slave can wait for AWVALID or WVALID, or both, before asserting WREADY
5. The slave can assert WREADY before AWVALID or WVALID, or both, are asserted
6. The slave must wait for both WVALID and WREADY to be asserted before asserting BVALID the slave must also wait for WLAST to be asserted before asserting BVALID, because the write response, BRESP, must be signaled only after the last data transfer of a write transaction
7. The slave must not wait for the master to assert BREADY before asserting BVALID
8. The master can wait for BVALID before asserting BREADY
9. The master can assert BREADY before BVALID is asserted.

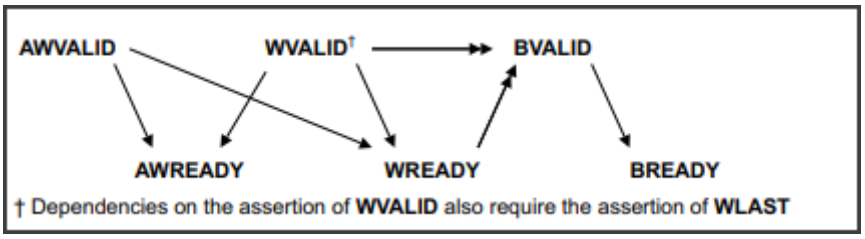
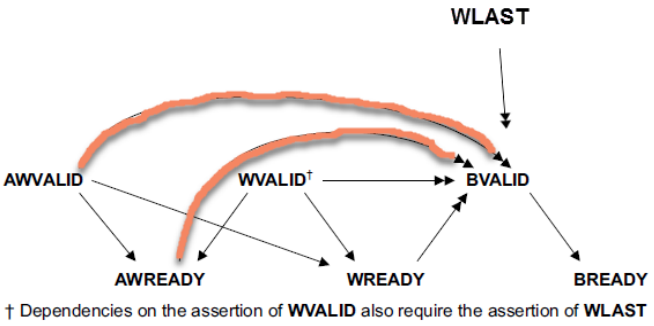


Fig 1.9 Write transaction dependencies

Write Response dependencies

The master must not wait for the slave to assert AWREADY or WREADY before asserting AWVALID or WVALID

1. The slave can wait for AWVALID or WVALID, or both, before asserting AWREADY
2. The slave can assert AWREADY before AWVALID or WVALID, or both, are asserted
3. The slave can wait for AWVALID or WVALID, or both, before asserting WREADY
4. The slave can assert WREADY before AWVALID or WVALID, or both, are asserted
5. The slave must wait for AWVALID, AWREADY, WVALID, and WREADY to be asserted before asserting BVALID the slave must also wait for WLAST to be asserted before asserting BVALID because the write response, BRESP must be signaled only after the last data transfer of a write transaction
6. The slave must not wait for the master to assert BREADY before asserting BVALID
7. The master can wait for BVALID before asserting BREADY
8. The master can assert BREADY before BVALID is asserted.



\ Fig 1.10 Write response dependencies

Chapter 2

ARCHITECTURE

2.1 AXI4 AVIP Testbench Architecture

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

HVL TOP has the design which is untimed and the transactions flow from both master virtual sequence and slave virtual sequence onto the AXI4 I/F through the BFM Proxy and BFM and gets the data from monitor BFM and uses the data to do checks using scoreboard and coverage

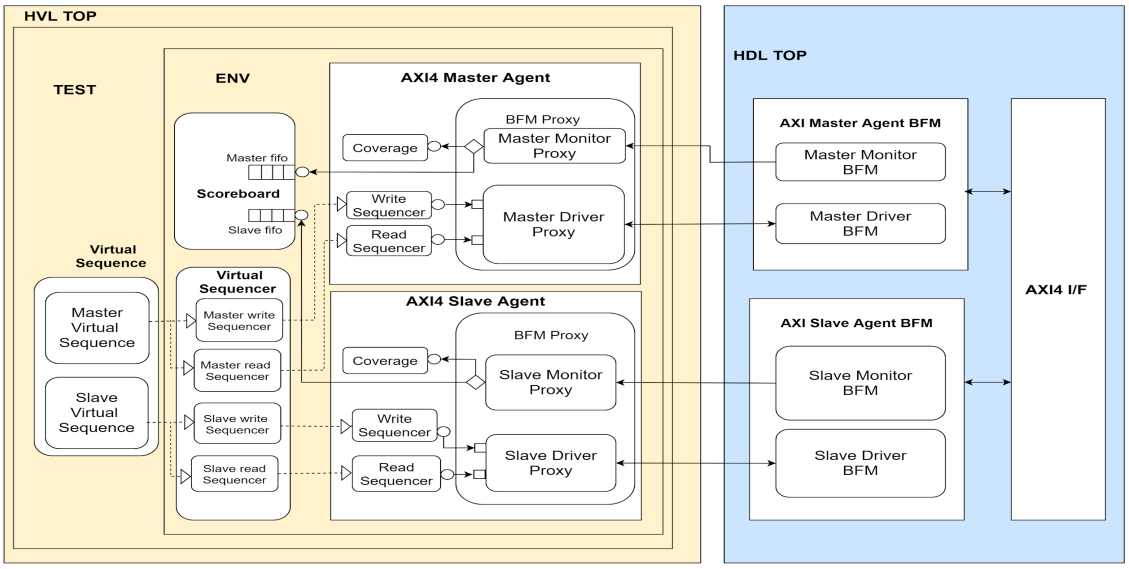


Fig 2.1 AXI4 AVIP Testbench Architecture

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

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

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

..

Chapter 3

IMPLEMENTATION

3.1 Pin Interface

Table 3.1 shows the AXI4 pins used to interface to external device

Signal	Master Direction	Slave Direction	Width of signals	Description
aclk	input	input	1 bit	System generated clock
areset_n	input	input	1 bit	It is an active low reset generated by system
awid	output	input	4 bits	This signal is the identification tag for the write address group of signals.
awaddr	output	input	32 bits	The write address gives the address of the first transfer in a write burst transaction.
awlen	output	input	4 bits	The burst length gives the exact number of transfers in a burst. This information determines the number of data transfers associated with the address.
awsize	output	input	3 bits	This signal indicates the size of each transfer in the burst.
awburst	output	input	2 bits	The burst type and the size information, determine how the address for each transfer within the burst is calculated.
awlock	output	input	2 bits	Provides additional information about the atomic characteristics of the transfer.
awcache	output	input	4 bits	This signal indicates how transactions are required to progress through a system.
awprot	output	input	3 bits	This signal indicates the privilege and security level of the transaction, and whether the transaction is a data access or an instruction access.
awqos	output	input	4 bits	The qos identifier is sent for each write transaction.
awregion	output	input	4 bits	Permits a single physical interface on a slave to be used for multiple logical interfaces.
awuser	output	input	4 bits	Optional User-defined signal in the write address channel.
awvalid	output	input	1 bit	This signal indicates that the channel is signalling valid write address and control information.
awready	input	output	1 bit	This signal indicates that the slave is ready to accept an address and associated control signals.
wdata	output	input	32 bits	Write data.
wstrb	output	input	4 bits	This signal indicates which byte lanes hold valid data. There is one write strobe bit for each eight bits

				of the write data bus.
wlast	output	input	1 bit	This signal indicates the last transfer in a write burst.
wuser	output	input	4 bits	Optional User-defined signal in the write data channel.
wvalid	output	input	1 bit	This signal indicates that valid write data and strobes are available.
wready	input	output	1 bit	This signal indicates that the slave can accept the write data.
bid	input	output	4 bits	This signal is the ID tag of the write response.
bresp	input	output	2 bits	This signal indicates the status of the write transaction.
buser	input	output	4 bits	Optional User-defined signal in the write response channel.
bvalid	input	output	1 bit	This signal indicates that the channel is signalling a valid write response.
bready	output	input	1 bit	This signal indicates that the master can accept a write response.
arid	output	input	4 bits	This signal is the identification tag for the read address group of signals.
araddr	output	input	32 bits	The read address gives the address of the first transfer in a read burst transaction.
arlen	output	input	8 bits	This signal indicates the exact number of transfers in a burst.
arsize	output	input	3 bits	This signal indicates the size of each transfer in the burst.
arburst	output	input	2 bits	The burst type and the size information determine how the address for each transfer within the burst is calculated.
arlock	output	input	2 bits	This signal provides additional information about the atomic characteristics of the transfer.
arcache	output	input	4 bits	This signal indicates how transactions are required to progress through a system.
arprot	output	input	3 bits	This signal indicates the privilege and security level of the transaction, and whether the transaction is a data access or an instruction access.
arqos	output	input	4 bits	qos identifier sent for each read transaction.
arregion	output	input	4 bits	Permits a single physical interface on a slave to be used for multiple logical interfaces.
aruser	output	input	4 bits	Optional User-defined signal in the read address channel.
arvalid	output	input	1 bit	This signal indicates that the channel is signalling valid read address and control information.

arready	input	output	1 bit	This signal indicates that the slave is ready to accept an address and associated control signals.
rid	input	output	4 bits	This signal is the identification tag for the read data group of signals generated by the slave.
rdata	input	output	32 bits	Read data.
rresp	input	output	2 bits	This signal indicates the status of the read transfer.
rlast	input	output	1 bit	This signal indicates the last transfer in a read burst.
ruser	input	output	4 bits	Optional User-defined signal in the read data channel.
rvalid	input	output	1 bit	This signal indicates that the channel is signalling the required read data.
rready	output	input	1 bit	This signal indicates that the master can accept the read data and response information.

For more details refer - axi4_avip_verification_plan

3.2 Testbench Components

In this section, testbench components of the axi4-avip are discussed

3.2.1 AXI HDL Top

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

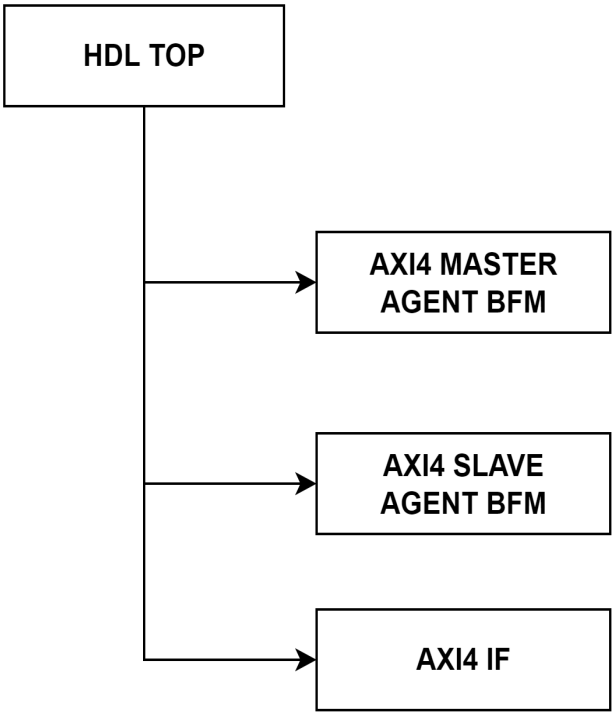


Fig 3.1 HDL Top

3.2.2 AXI Interface

Importing the global packages
Passing Signals: aclk, aresetn
Declaration of signals: awid, awaddr, awlen, awsize, awburst, awlock, awcache, awprot, awvalid, awready, wdata, wstrb, wlast, wuser, valid, wready, bid, bresp,buser, bvalid, bready, arid, araddr, arlen, arsize, arburst, arlock, arcache, arprot, arqos, arregion, aruser, arvalid, arready, rid, rdata, rresp, rlast, ruser, rvali, rready, are declared as logic type.

3.2.3 AXI Master Agent BFM Module

Instantiates the below two interfaces here

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

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

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

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

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

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

3.2.4 AXI Master Driver BFM Interface

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

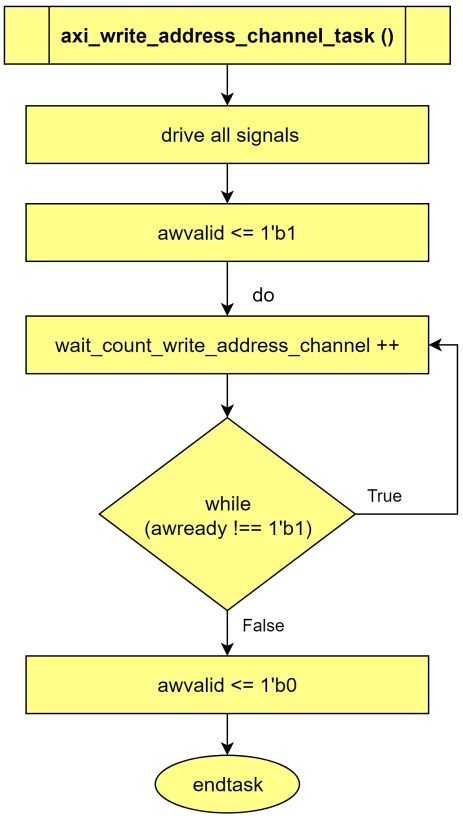


Fig 3.4 Flowchart of axi4_write_address_channel_task

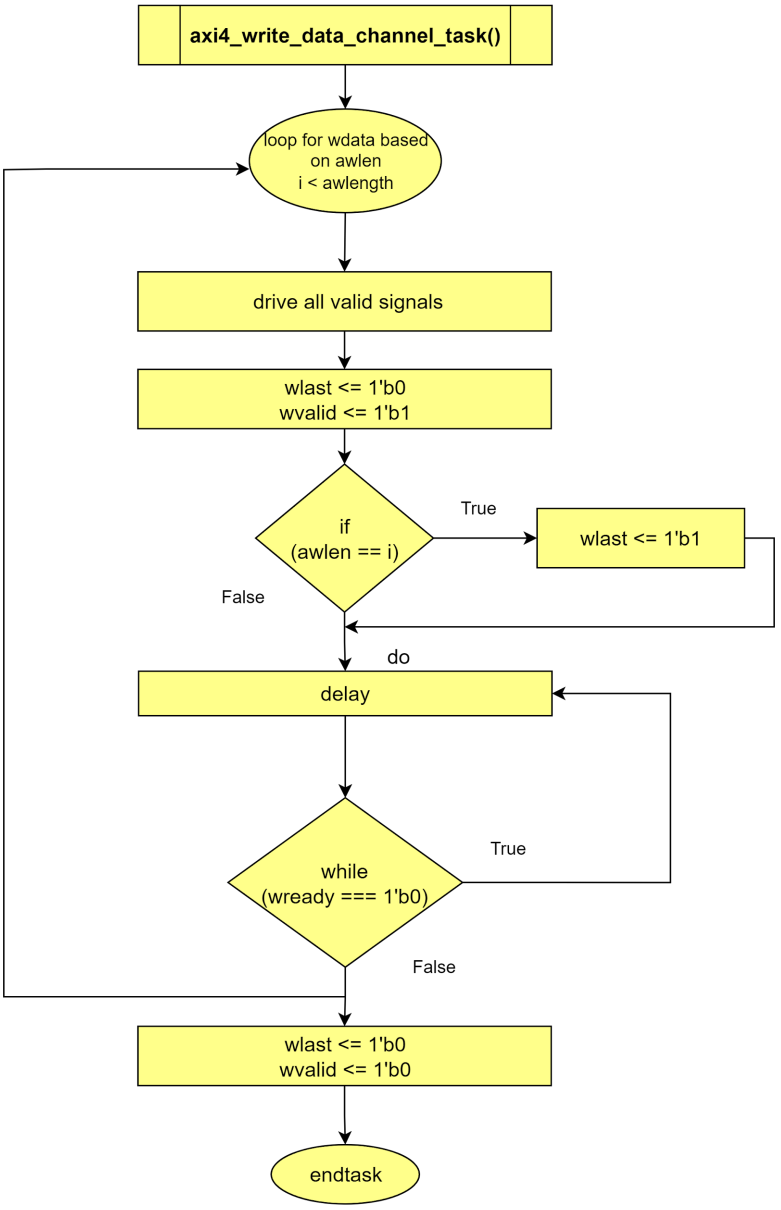


Fig 3.5 Flowchart of axi4_write_data_channel_task

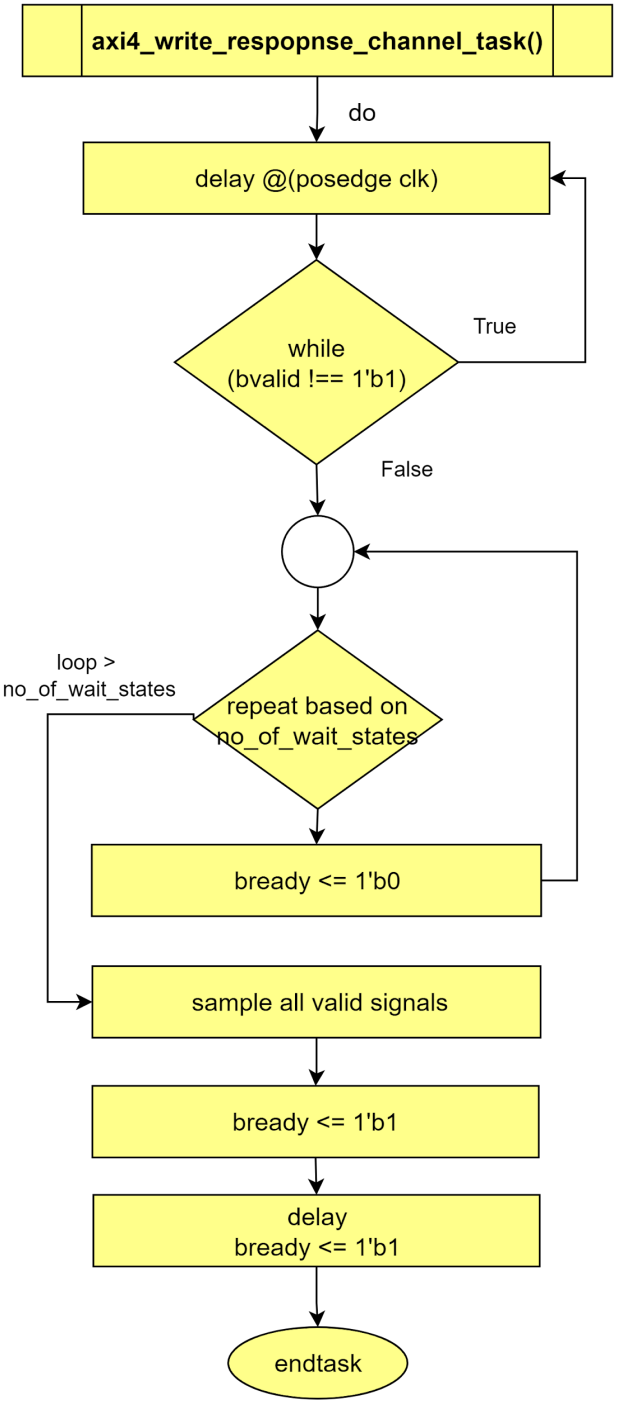


Fig 3.6 Flowchart of axi4_write_response_channel_task

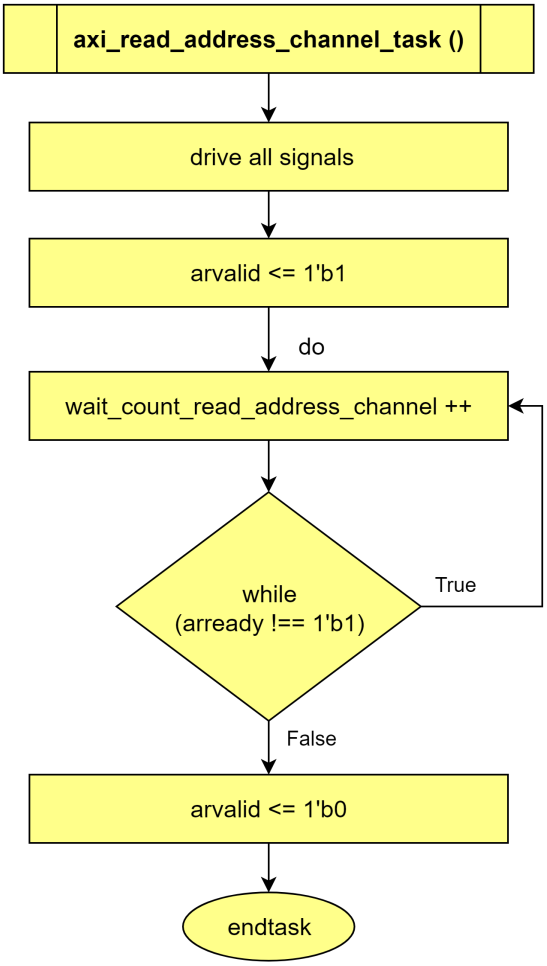


Fig 3.7 Flowchart of axi4_read_address_channel_task

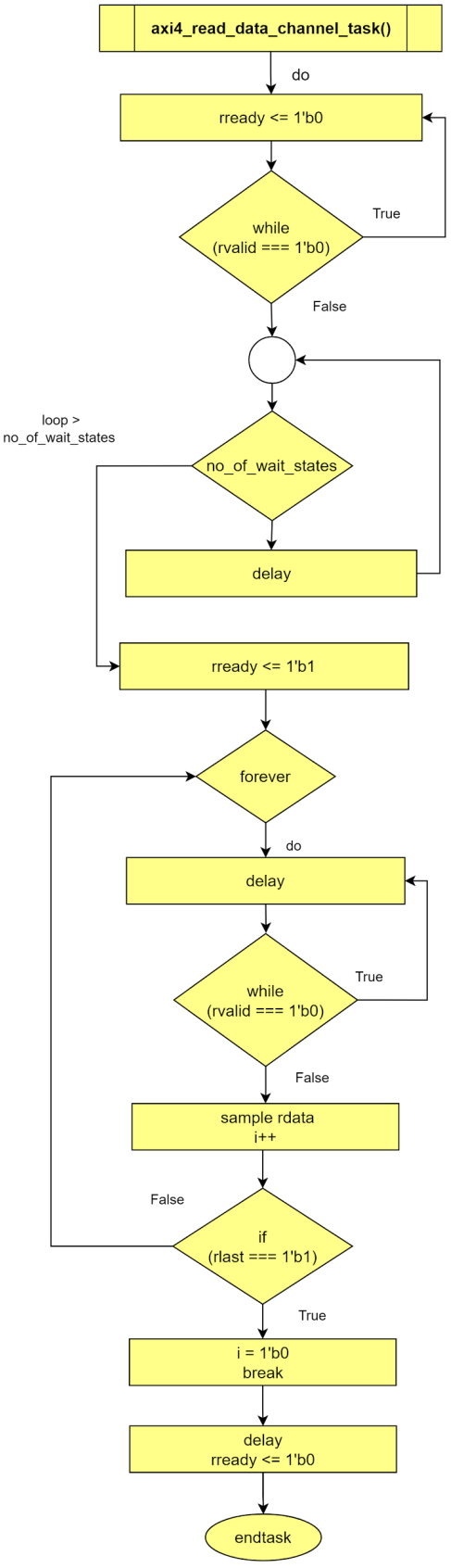


Fig 3.8 Flowchart of axi4_read_data_channel_task

3.2.5 AXI Master Monitor BFM Interface

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

3.2.6 AXI Slave Agent BFM Module

Instantiates the below two interfaces here

1. axi4 slave driver bfm and
2. axi4 slave monitor bfm.

Instantiates the axi4 slave assertions and binds it with the axi4 slave monitor bfm handle and maps the signals of axi4 slave assertions with the axi4 interface signals. The axi4 interface signals are passed to the axi4 slave driver and monitor bfm in instantiations as shown in fig. 3.9 and fig. 3.10.

```
axi4_slave_driver_bfm axi4_slave_drv_bfm_h (.aclk     (intf.aclk)     ,
                                            .aresetn  (intf.aresetn)  ,
                                            .awid     (intf.awid)     ,
                                            .awaddr   (intf.awaddr)   ,
                                            .awlen    (intf.awlen)    ,
                                            .awsize   (intf.awsize)   ,
                                            .awburst  (intf.awburst)  ,
                                            .awlock   (intf.awlock)   ,
                                            .awcache  (intf.awcache)  ,
                                            .awprot   (intf.awprot)   ,
                                            .awvalid  (intf.awvalid)  ,
                                            .awready  (intf.awready)  ,

                                            .wdata    (intf.wdata)    ,
                                            .wstrb    (intf.wstrb)    ,
                                            .wlast    (intf.wlast)    ,
                                            .wuser    (intf.wuser)    ,
                                            .wvalid   (intf.wvalid)   ,
                                            .wready   (intf.wready)   ,

                                            .bid      (intf.bid)      ,
                                            .bresp    (intf.bresp)    ,
                                            .buser    (intf.buser)    ,
                                            .bvalid   (intf.bvalid)   ,
                                            .bready   (intf.bready)   ,

                                            .arid     (intf.arid)     ,
                                            .araddr   (intf.araddr)   ,
                                            .arlen    (intf.arlen)    ,
                                            .arsize   (intf.arsize)   ,
```

Fig 3.9 AXI4 slave driver bfm instantiation in axi4 slave agent bfm code snippet

```
axi4_slave_monitor_bfm axi4_slave_mon_bfm_h (.aclk(intf.aclk),
                                             .aresetn(intf.aresetn),
                                             .awid     (intf.awid)     ,
                                             .awaddr   (intf.awaddr)   ,
                                             .awlen    (intf.awlen)    ,
                                             .awsize   (intf.awsize)   ,
                                             .awburst  (intf.awburst)  ,
                                             .awlock   (intf.awlock)   ,
                                             .awcache  (intf.awcache)  ,
                                             .awprot   (intf.awprot)   ,
                                             .awvalid  (intf.awvalid)  ,
                                             .awready  (intf.awready)  ,

                                             .wdata    (intf.wdata)    ,
                                             .wstrb    (intf.wstrb)    ,
                                             .wlast    (intf.wlast)    ,
                                             .wuser    (intf.wuser)    ,
                                             .wvalid   (intf.wvalid)   ,
                                             .wready   (intf.wready)   ,

                                             .bid      (intf.bid)      ,
                                             .bresp    (intf.bresp)    ,
                                             .buser    (intf.buser)    ,
                                             .bvalid   (intf.bvalid)   ,
                                             .bready   (intf.bready)   ,

                                             .arid     (intf.arid)     ,
                                             .araddr   (intf.araddr)   ,
                                             .arlen    (intf.arlen)    ,
                                             .arsize   (intf.arsize)   ,
                                             .arburst  (intf.arburst)  ,
                                             .arlock   (intf.arlock)   ,
```

Fig 3.10 AXI4 slave monitor bfm instantiation in axi4 slave agent bfm code snippet

3.2.7 AXI Slave Driver BFM Interface

AXI4 slave driver bfm is an interface where it will get the signals from the axi4 interface. It has a five method calls 3 for Write Channels and 2 for Read Channels

1. Write address phase
2. Write data phase
3. Write response phase
4. Read Address phase
5. Read data phase

which will be called by the axi4 slave driver proxy which drives the address, data and response of write and read channels to the axi4 interface.

Figure 3.11 gives the reference for the instantiation of axi4 slave driver bfm.

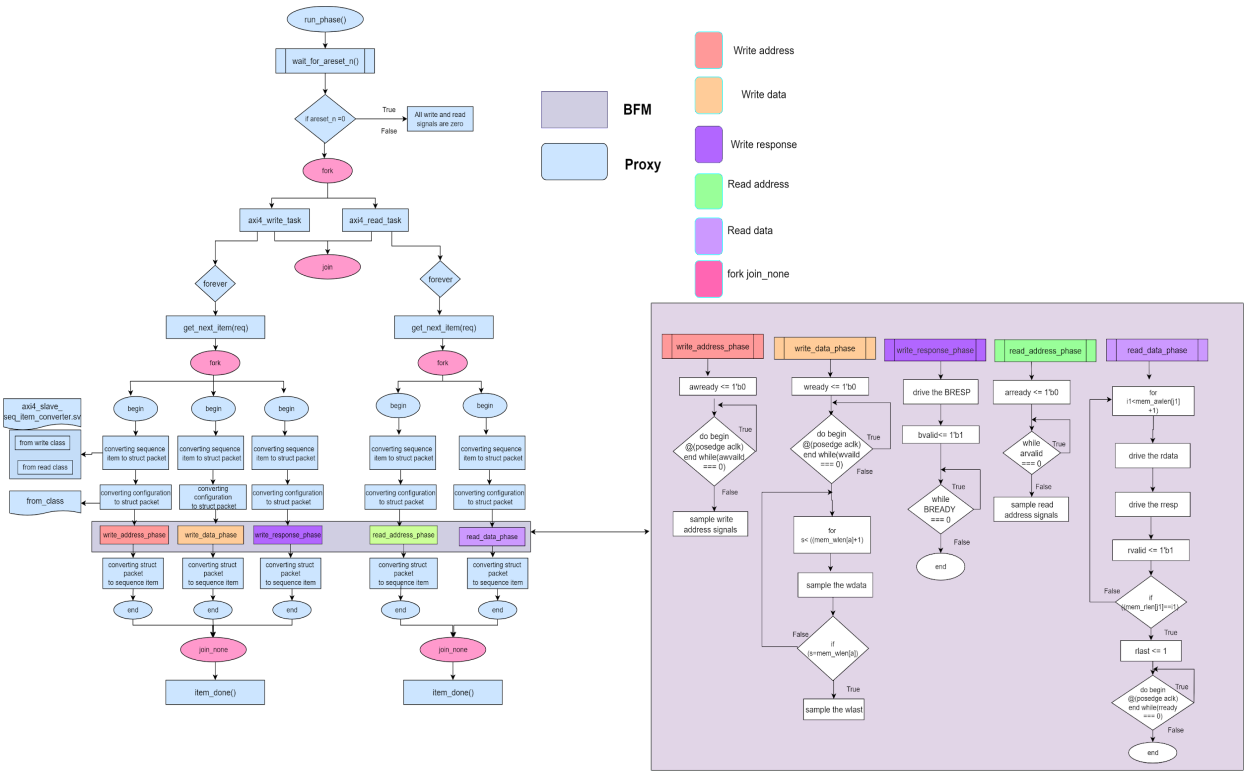


Fig 3.11 Slave driver Proxy and BFM flow chart

The above figure describes the flowchart of slave driver proxy and slave driver bfm and detailed flow chart of slave driver proxy and slave driver bfm is explained below:

Write Address Phase:

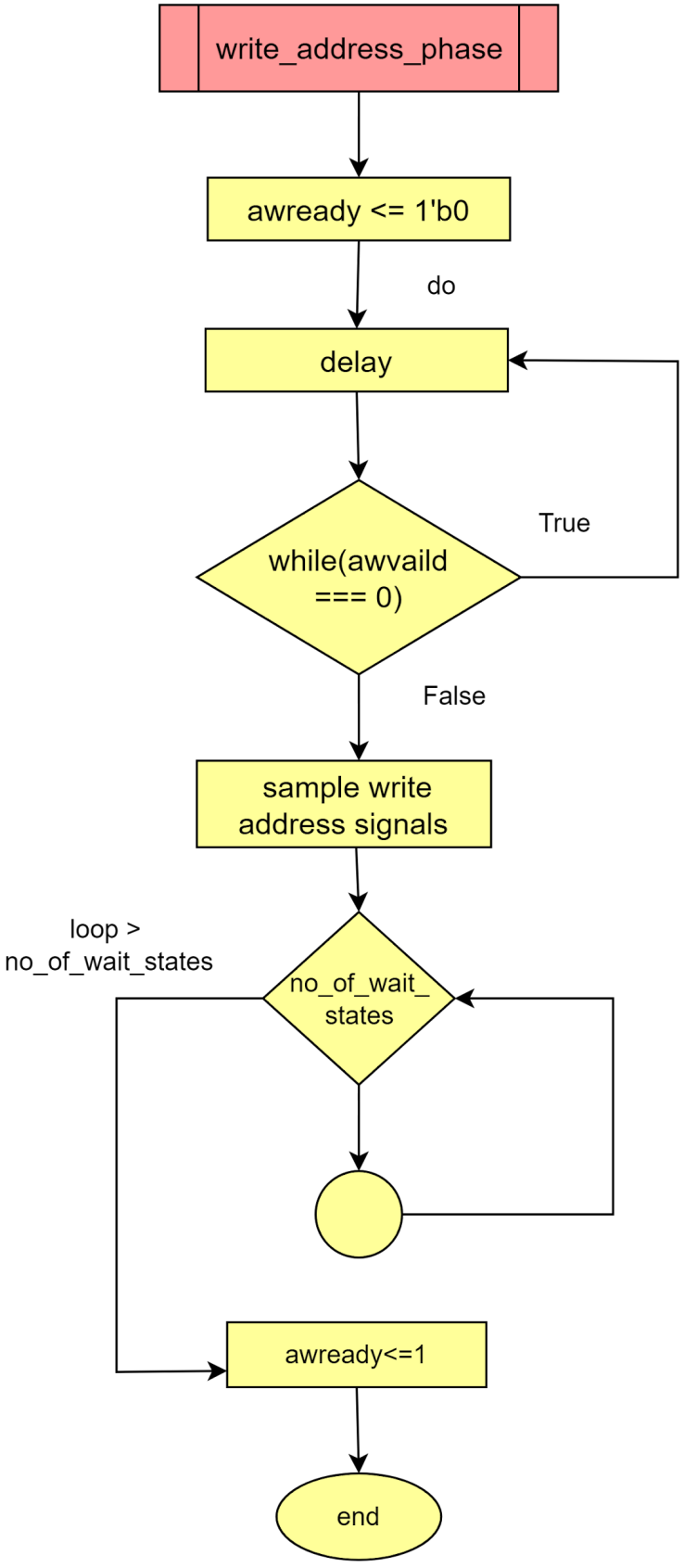


Fig 3.12 Write_address_phase

Write data Phase:

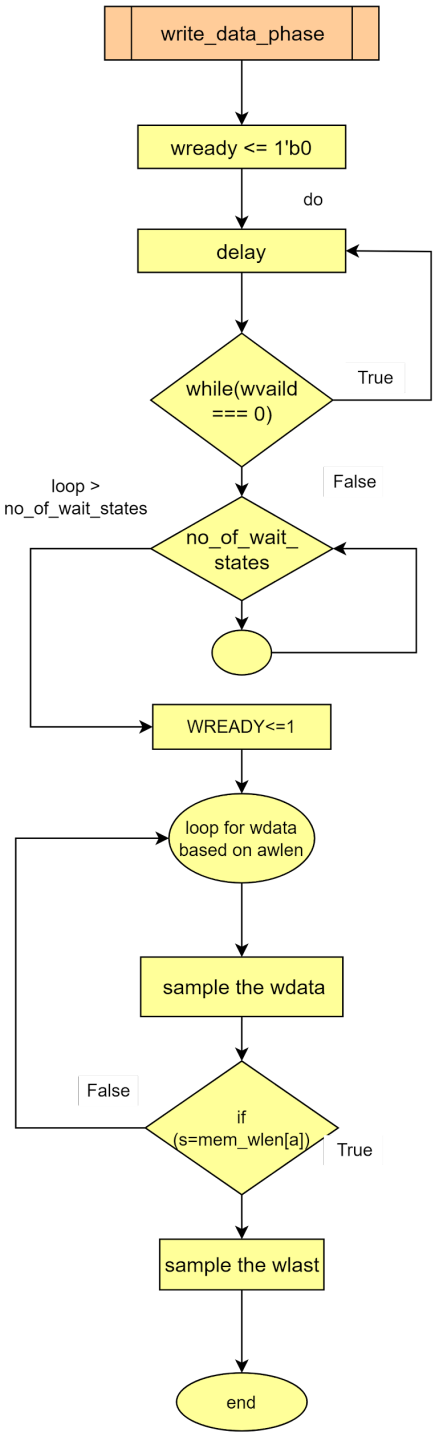


Fig 3.13 Write_data_phase

Write Response Phase:

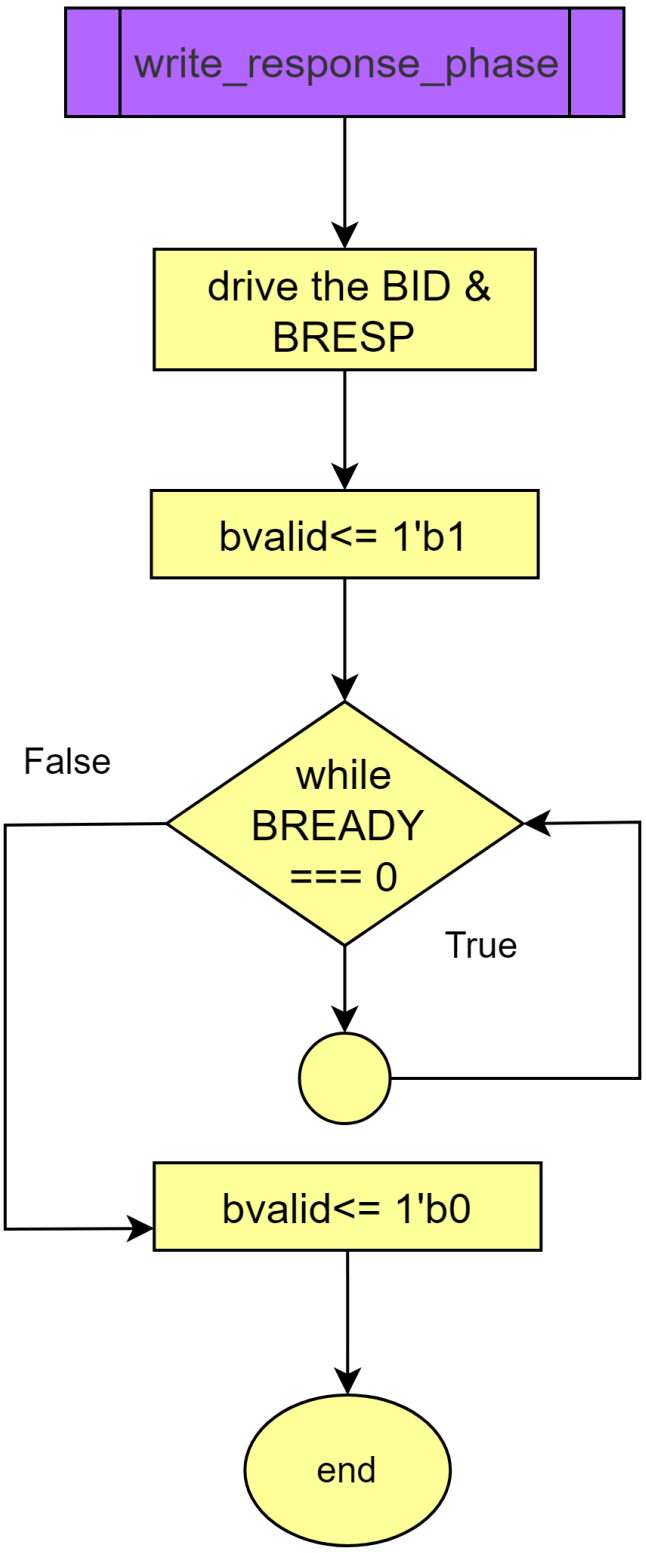


Fig 3.14 Write_response_phase

Read Address Phase:

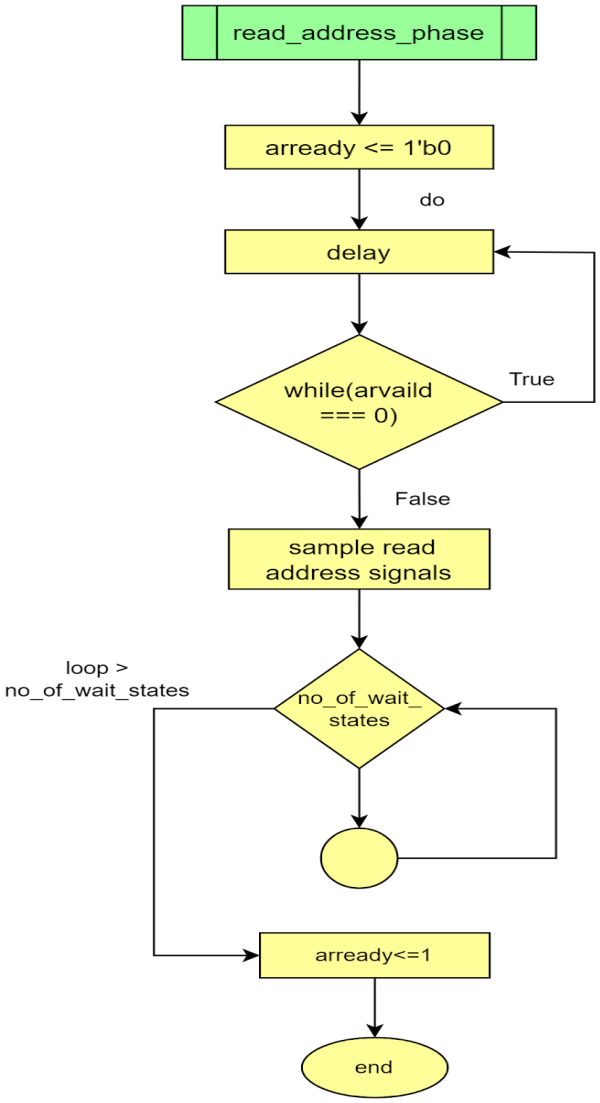


Fig 3.15 Read_address_phase

Read data Phase:

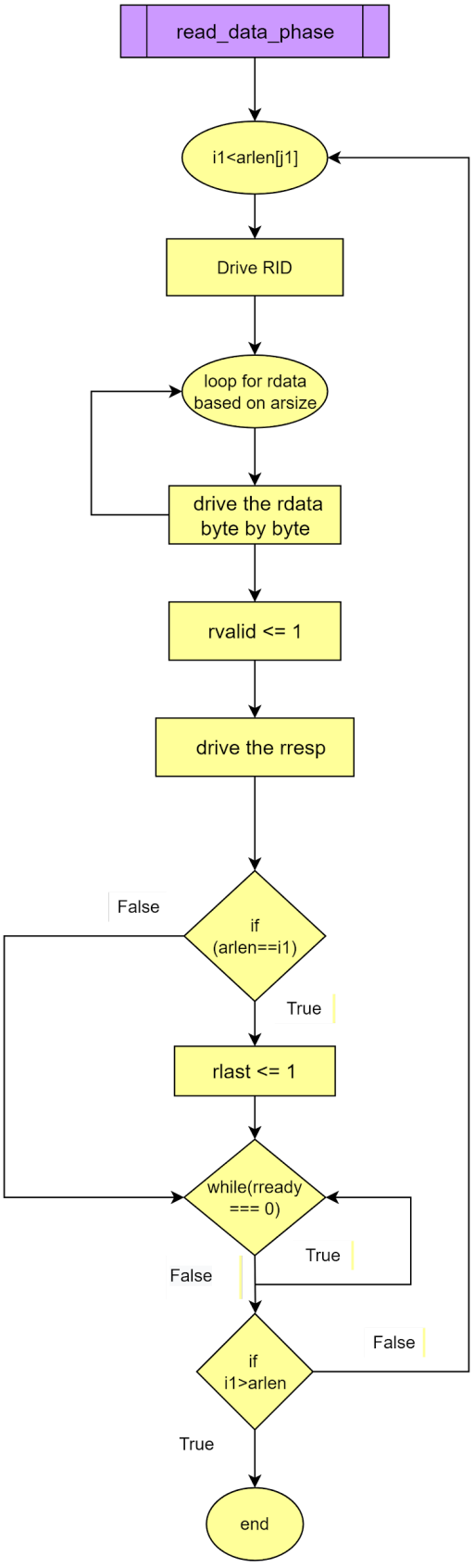


Fig 3.16 Read_data_phase

3.2.8 AXI Slave Monitor BFM Interface

AXI4 slave monitor bfm is an interface where it will get the signals from the axi4 interface. It has a method sample_data which will be called by the axi4 slave monitor proxy which samples the {pwdata and prdata} from the axi4 interface. After sampling the data, the axi4 slave monitor bfm interface sends the data to the axi4 slave monitor proxy using the output port of sample_data task. fig. gives the reference for the instantiation of axi4 slave monitor bfm.

3.2.9 AXI HVL_TOP

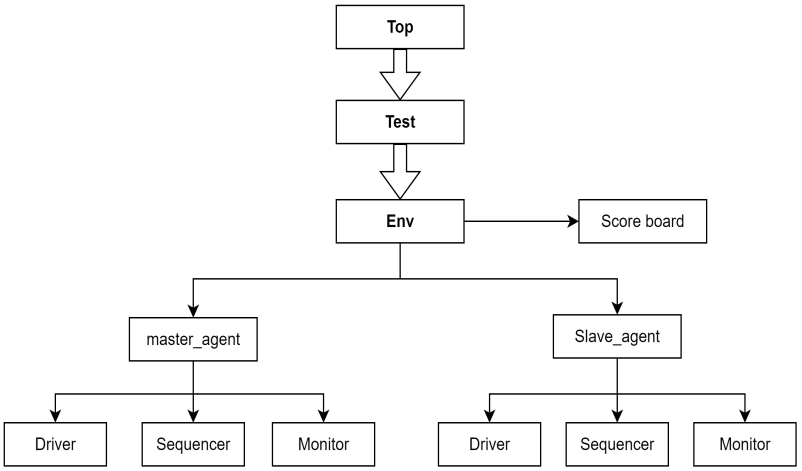


Fig 3.17 HVL Top

In top test is running by using the **run_test("test_name")** method, which will start the whole tb components. Fig. 3.17 give the HVL top hierarchy.

3.2.10 AXI Environment

Environment has the below components

a. axi4_scoreboard
b. axi4_virtual_sequencer
c. axi4_master_agent
d. axi4_slave_agent

In the build phase, env_cfg handle will be called and create the memory for the above declared components.
In the connect phase, the axi4_master_monitor_proxy is connected to axi4_scoreboard and axi4_slave_monitor_proxy to axi4_scoreboard using analysis port and analysis fifo.

3.2.11 AXI 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 AXI4-AVIP project is to

1. Compare the Write address, write data,write response, read address and read 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 five analysis fifo's which receive the packets from the analysis port of the monitor class. fig. 3.18 shows the connection between the analysis port and analysis fifo.

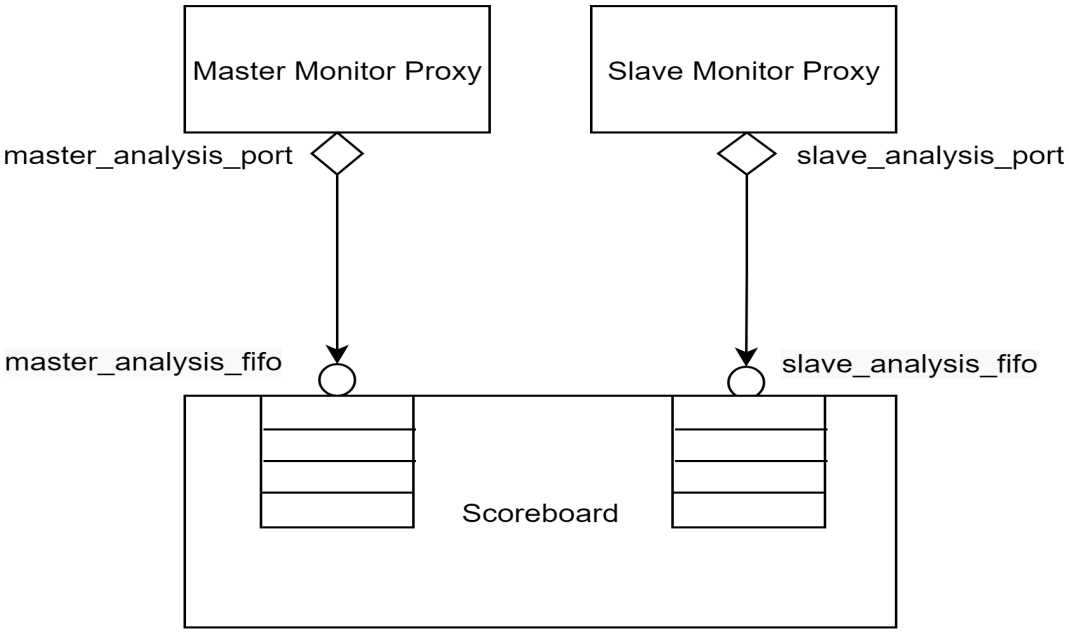


Fig 3.18 connection of the analysis ports of the monitor to the scoreboard analysis fifo

In the monitor proxy class of master and slave, five analysis ports are declared. Fig 3.19 shows the declaration of master analysis port and slave analysis port in the master monitor proxy and slave monitor proxy

```
27   // Declaring analysis port for the monitor port
28   uvm_analysis_port#(axi4_master_tx) axi4_master_read_address_analysis_port;
29   uvm_analysis_port#(axi4_master_tx) axi4_master_read_data_analysis_port;
30   uvm_analysis_port#(axi4_master_tx) axi4_master_write_address_analysis_port;
31   uvm_analysis_port#(axi4_master_tx) axi4_master_write_data_analysis_port;
32   uvm_analysis_port#(axi4_master_tx) axi4_master_write_response_analysis_port;
33
```

```
23  // Variable: axi4_slave_analysis_port
24  // Declaring analysis port for the monitor port
25  uvm_analysis_port#(axi4_slave_tx) axi4_slave_write_address_analysis_port;
26  uvm_analysis_port#(axi4_slave_tx) axi4_slave_write_data_analysis_port;
27  uvm_analysis_port#(axi4_slave_tx) axi4_slave_write_response_analysis_port;
28  uvm_analysis_port#(axi4_slave_tx) axi4_slave_read_address_analysis_port;
29  uvm_analysis_port#(axi4_slave_tx) axi4_slave_read_data_analysis_port;
30
```

Fig. 3.19 declaration of master and slave analysis port

In the scoreboard, five analysis fifo's are declared. Fig 3.20 shows the declaration of master analysis fiifo and slave analysis fifo in the scoreboard.

```
24  //Variable : axi4_master_analysis_fifo
25  //Used to store the axi4_master_data
26  uvm_tlm_analysis_fifo#(axi4_master_tx) axi4_master_read_address_analysis_fifo;
27  uvm_tlm_analysis_fifo#(axi4_master_tx) axi4_master_read_data_analysis_fifo;
28  uvm_tlm_analysis_fifo#(axi4_master_tx) axi4_master_write_address_analysis_fifo;
29  uvm_tlm_analysis_fifo#(axi4_master_tx) axi4_master_write_data_analysis_fifo;
30  uvm_tlm_analysis_fifo#(axi4_master_tx) axi4_master_write_response_analysis_fifo;
31
32  //Variable : axi4_slave_analysis_fifo
33  //Used to store the axi4_slave_data
34  uvm_tlm_analysis_fifo#(axi4_slave_tx) axi4_slave_read_address_analysis_fifo;
35  uvm_tlm_analysis_fifo#(axi4_slave_tx) axi4_slave_read_data_analysis_fifo;
36  uvm_tlm_analysis_fifo#(axi4_slave_tx) axi4_slave_write_address_analysis_fifo;
37  uvm_tlm_analysis_fifo#(axi4_slave_tx) axi4_slave_write_data_analysis_fifo;
38  uvm_tlm_analysis_fifo#(axi4_slave_tx) axi4_slave_write_response_analysis_fifo;
39
```

Fig 3.20 shows the declaration of master and slave analysis fifo in the scoreboard

In the constructor, create objects for the five declared analysis fifo's. Fig 3.21 shows the creation of the master and slave analysis port.

```
148 function axi4_scoreboard::new(string name = "axi4_scoreboard",
149                                 uvm_component parent = null);
150   super.new(name, parent);
151   axi4_master_write_address_analysis_fifo = new("axi4_master_write_address_analysis_fifo",this);
152   axi4_master_write_data_analysis_fifo = new("axi4_master_write_data_analysis_fifo",this);
153   axi4_master_write_response_analysis_fifo= new("axi4_master_write_response_analysis_fifo",this);
154   axi4_master_read_address_analysis_fifo = new("axi4_master_read_address_analysis_fifo",this);
155   axi4_master_read_data_analysis_fifo = new("axi4_master_read_data_analysis_fifo",this);
156
157   axi4_slave_write_address_analysis_fifo = new("axi4_slave_write_address_analysis_fifo",this);
158   axi4_slave_write_data_analysis_fifo = new("axi4_slave_write_data_analysis_fifo",this);
159   axi4_slave_write_response_analysis_fifo= new("axi4_slave_write_response_analysis_fifo",this);
160   axi4_slave_read_address_analysis_fifo = new("axi4_slave_read_address_analysis_fifo",this);
161   axi4_slave_read_data_analysis_fifo = new("axi4_slave_read_data_analysis_fifo",this);
162
163   write_address_key = new(1);
164   write_data_key = new(1);
165   write_response_key = new(1);
166   read_address_key = new(1);
167   read_data_key = new(1);
168
169 endfunction : new
```

Fig 3.21 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 3.22 shows the connection made between the monitor analysis port and the scoreboard fifo's in the connect phase of the env class.

```
140  foreach(axi4_master_agent_h[i]) begin
141    axi4_master_agent_h[i].axi4_master_mon_proxy_h.axi4_master_read_address_analysis_port.connect(axi4_scoreboard_h.axi4_master_read_address_analysis_fifo.analysis_export);
142    axi4_master_agent_h[i].axi4_master_mon_proxy_h.axi4_master_read_data_analysis_port.connect(axi4_scoreboard_h.axi4_master_read_data_analysis_fifo.analysis_export);
143    axi4_master_agent_h[i].axi4_master_mon_proxy_h.axi4_master_write_address_analysis_port.connect(axi4_scoreboard_h.axi4_master_write_address_analysis_fifo.analysis_export);
144    axi4_master_agent_h[i].axi4_master_mon_proxy_h.axi4_master_write_data_analysis_port.connect(axi4_scoreboard_h.axi4_master_write_data_analysis_fifo.analysis_export);
145    axi4_master_agent_h[i].axi4_master_mon_proxy_h.axi4_master_write_response_analysis_port.connect(axi4_scoreboard_h.axi4_master_write_response_analysis_fifo.analysis_export);
146  end
147
148  foreach(axi4_slave_agent_h[i]) begin
149    axi4_slave_agent_h[i].axi4_slave_mon_proxy_h.axi4_slave_write_address_analysis_port.connect(axi4_scoreboard_h.axi4_slave_write_address_analysis_fifo.analysis_export);
150    axi4_slave_agent_h[i].axi4_slave_mon_proxy_h.axi4_slave_write_data_analysis_port.connect(axi4_scoreboard_h.axi4_slave_write_data_analysis_fifo.analysis_export);
151    axi4_slave_agent_h[i].axi4_slave_mon_proxy_h.axi4_slave_write_response_analysis_port.connect(axi4_scoreboard_h.axi4_slave_write_response_analysis_fifo.analysis_export);
152    axi4_slave_agent_h[i].axi4_slave_mon_proxy_h.axi4_slave_read_address_analysis_port.connect(axi4_scoreboard_h.axi4_slave_read_address_analysis_fifo.analysis_export);
153    axi4_slave_agent_h[i].axi4_slave_mon_proxy_h.axi4_slave_read_data_analysis_port.connect(axi4_scoreboard_h.axi4_slave_read_data_analysis_fifo.analysis_export);
154  end
```

Fig 3.22 Connection done between the analysis port and analysis fifo exportin the env class

In the run phase of the scoreboard, the get() method is used to get the data packet from the monitor write() method. Fig 3.23 shows the use of the get() method to get the transaction from the monitor analysis port.

```
task axi4_scoreboard::run_phase(uvm_phase phase);

  super.run_phase(phase);

  forever begin
    `uvm_info(get_type_name(),$sformatf("calling analysis fifo in scoreboard"),UVM_HIGH);

    fork
      begin : WRITE_ADDRESS_CHANNEL
        write_address_key.get(1);
        axi4_master_write_address_analysis_fifo.get(axi4_master_tx_h1);
        `uvm_info(get_type_name(),$sformatf("scoreboard's axi4_master_write_address_channel \n%s",axi4_master_tx_h1.sprint()),UVM_HIGH)
        axi4_slave_write_address_analysis_fifo.get(axi4_slave_tx_h1);
        `uvm_info(get_type_name(),$sformatf("scoreboard's axi4_slave_write_address_channel \n%s",axi4_slave_tx_h1.sprint()),UVM_HIGH)
        axi4_write_address_comparision(axi4_master_tx_h1,axi4_slave_tx_h1);
        write_address_key.put(1);
      end
```

Fig 3.23 Use of get method to get the packet from monitor analysis port

The Comparison of the write address and write address from the master monitor and slave monitor is done in the run phase. Fig 3.24 shows the comparison of the master write address with slave write address.

```
task axi4_scoreboard::axi4_write_address_comparision(input axi4_master_tx axi4_master_tx_h1,input axi4_slave_tx axi4_slave_tx_h1);
  //$display("------------------------------------------------------------------------------");
  //$display("SCOREBOARD WRITE ADDRESS CHANNEL COMPARISIONS");
  //$display("------------------------------------------------------------------------------");
  if(axi4_master_tx_h1.awid == axi4_slave_tx_h1.awid)begin
    `uvm_info(get_type_name(),$sformatf("axi4_awid from master and slave is equal"),UVM_HIGH);
    `uvm_info("SB_AWID_MATCHED", $sformatf("Master AWID = 'h%0x and Slave AWID = 'h%0x",axi4_master_tx_h1.awid,axi4_slave_tx_h1.awid), UVM_HIGH);
    byte_data_cmp_verified_awid_count++;
  end
  else begin
    `uvm_info(get_type_name(),$sformatf("axi4_awid from master and slave is  not equal"),UVM_HIGH);
    `uvm_info("SB_AWID_NOT_MATCHED", $sformatf("Master AWID = 'h%0x and Slave AWID = 'h%0x",axi4_master_tx_h1.awid,axi4_slave_tx_h1.awid), UVM_HIGH);
    byte_data_cmp_failed_awid_count++;
  end

```

Fig 3.24 The comparison of the master write address with slave write address

Similarly, the comparison is done for the write data,write response ,read address and read data as well.

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

a. Whether all write address ,write data,write response,read address and read data comparisons are successful
b. Whether all transactions from master and slave monitors are equal
c. Whether both FIFO's are empty or not

Fig 3.25 is the flow chart of the scoreboard report phase, which explains checks made to identify the success/failure rates.

scoreboard run phase

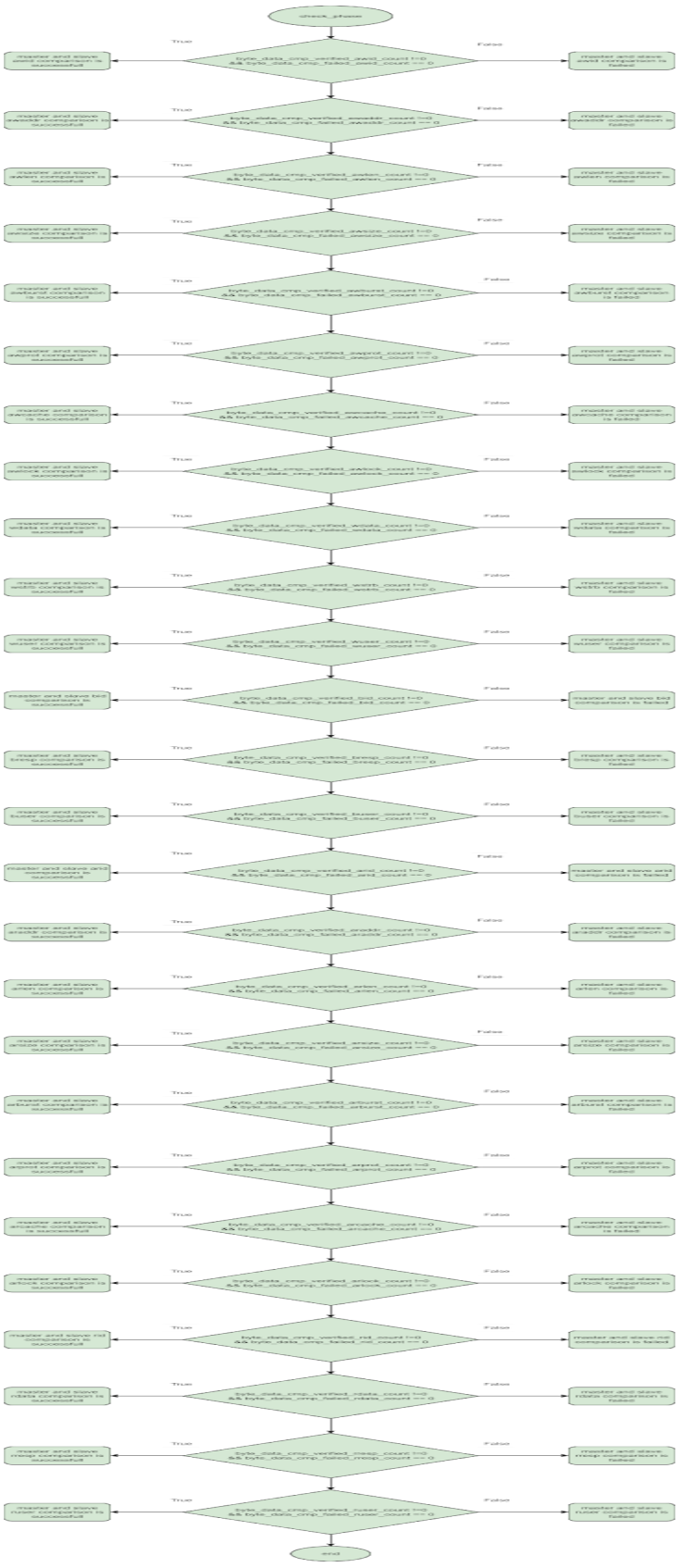


Fig 3.25 Flow chart of the scoreboard report phase

3.2.12 AXI Virtual Sequencer

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

3.2.13 AXI Master Agent

AXI4 master agent component is a class extending from uvm_agent. It gets the axi4 master agent config handle and based on that we will create and connect the components. It creates the axi4 master sequencer and axi4 master driver only if the axi4 master agent is active which will depend on the value of is_active variable declared in the axi4 master agent configuration file. The axi4 master coverage is created in build_phase if the has_coverage variable is 1 which is declared in the axi4 master agent configuration file. Please refer to figure 3.26 for the axi4 master agent build_phase code snippet.

AXI4 master agent build phase has creation of,

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

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

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

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

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

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


endfunction : build_phase
```

Fig 3.26 AXI4 master agent build phase code snippet

AXI4 master agent configuration handles declared in the above created components will be mapped here in the connect phase. The axi4 master driver proxy and axi4 master sequencer are connected using TLM ports if the axi4 master agent is active. The axi4 master coverage's analysis_export will be connected to axi4 master monitor proxy's master_analysis_port in connect_phase as shown in fig. 3.27

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

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

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

  axi4_master_mon_proxy_h.axi4_master_agent_cfg_h = axi4_master_agent_cfg_h;

endfunction : connect_phase
```

Fig 3.27 AXI$ master agent connect phase code snippet

3.2.14 AXI Master Sequencer

AXI4 master sequencer component is a parameterised class of type axi4 master transaction, extending uvm_sequencer. AXI4 sequencer sends the data from the axi4 master sequences to the axi4 driver proxy.

3.2.15 AXI Master Driver Proxy

Axi4 master driver proxy has the connection to master driver bfm in the end of elaboration phase. The master driver proxy run phase calls the write and read tasks in parallel using fork join as shown in fig. 3.28 and fig. 3.29

The write task checks for the transfer type, if it is BLOCKING_WRITE then write_address, write_data and write_response will be called sequentially. The req packet will be converted to struct packet and then passed to each task. The received response will be converted to a struct packet using to_class as shown in fig. 3.30

If the transfer type is NON_BLOCKING_WRITE, then write_address, write_data and write_response will be called in parallel using the fork join_any process control statements. Initially, the req packet is converted to struct packet and kept in write fifo so that write data and write response channels can make use of it.

The write address channel uses the process awaddr_process to control the fork join . The write address channel gets the req packet and sends it to the bfm write address channel. Later it takes the output and converts it back to the req packet.

The write data channel uses the process wdata_process to control the fork join . Before starting the transfer, it gets the semaphore key of the wdata_process. The write data channel peeks the req packet from the analysis write fifo and sends it to the bfm write data channel. Later it takes the output and converts it back to the req packet. As the transfer is completed it will put back the key in the semaphore.

The write response channel uses the process wresponse_process to control the fork join . Before starting the transfer, it gets the semaphore key of the wresponse_process. The write response channel gets the req packet from the analysis write fifo and sends it to the bfm write data channel. Later, it takes the output and converts it back to the req packet. As the transfer is completed it will put back the key in the semaphore.

After the completion of the fork, join_any awaddr.await() method will make sure that the write address process has to be completed.

Now the master driver proxy calls the item_done method to complete the transaction.

The similar transaction process will be happening in the READ TRANSFER as well as shown in fig. 3.31

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

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

  fork
    axi4_write_task();
    axi4_read_task();
  join

endtask : run_phase
```

Fig 3.28 run phase of axi4 master driver proxy code snippet

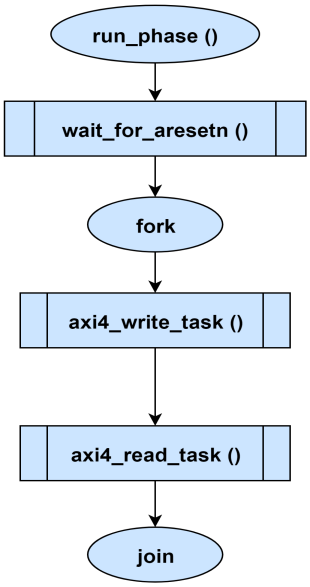


Fig 3.29 Flowchart for run phase of axi4 master driver proxy

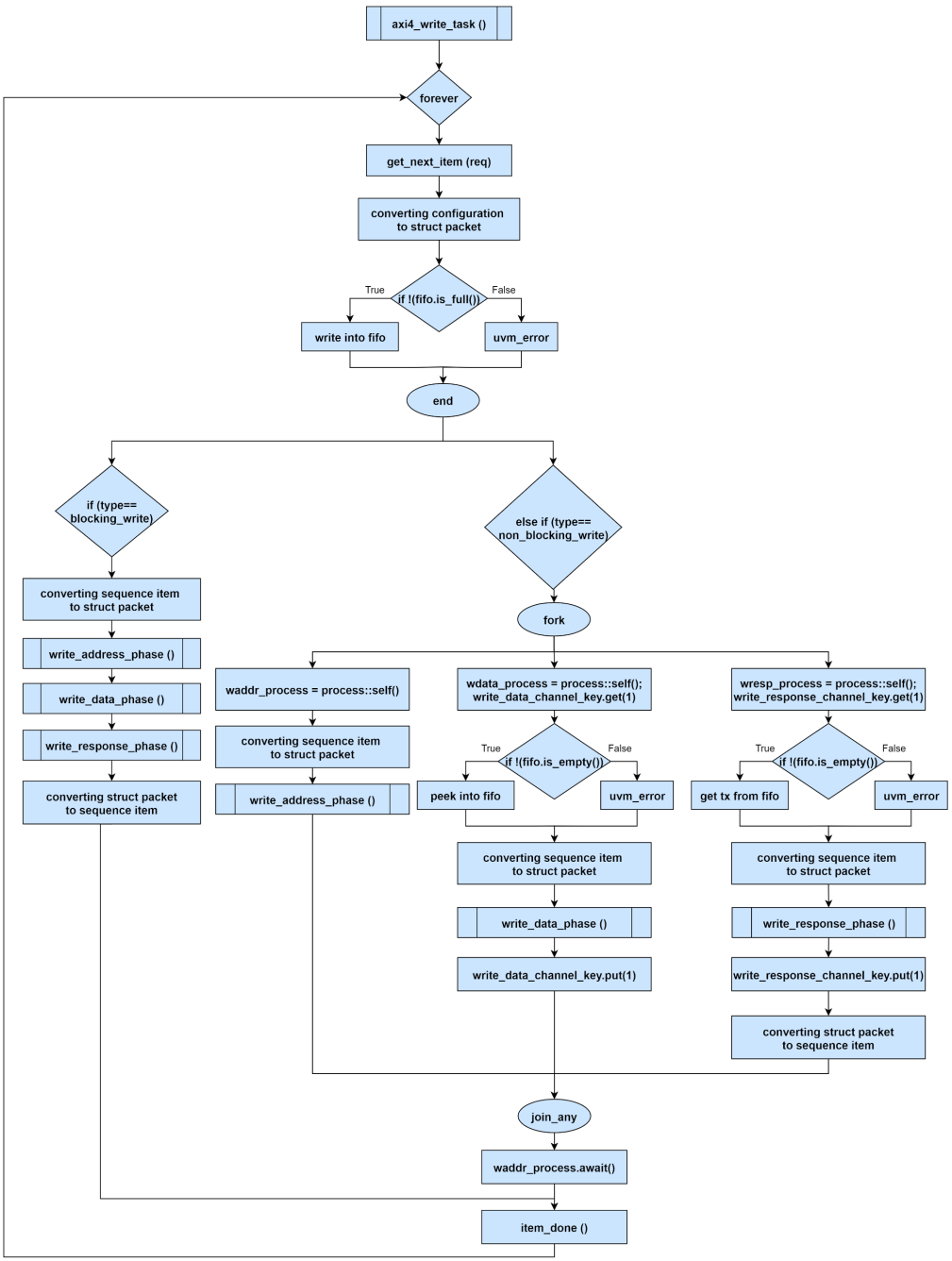


Fig 3.30 flowchart of write_task()

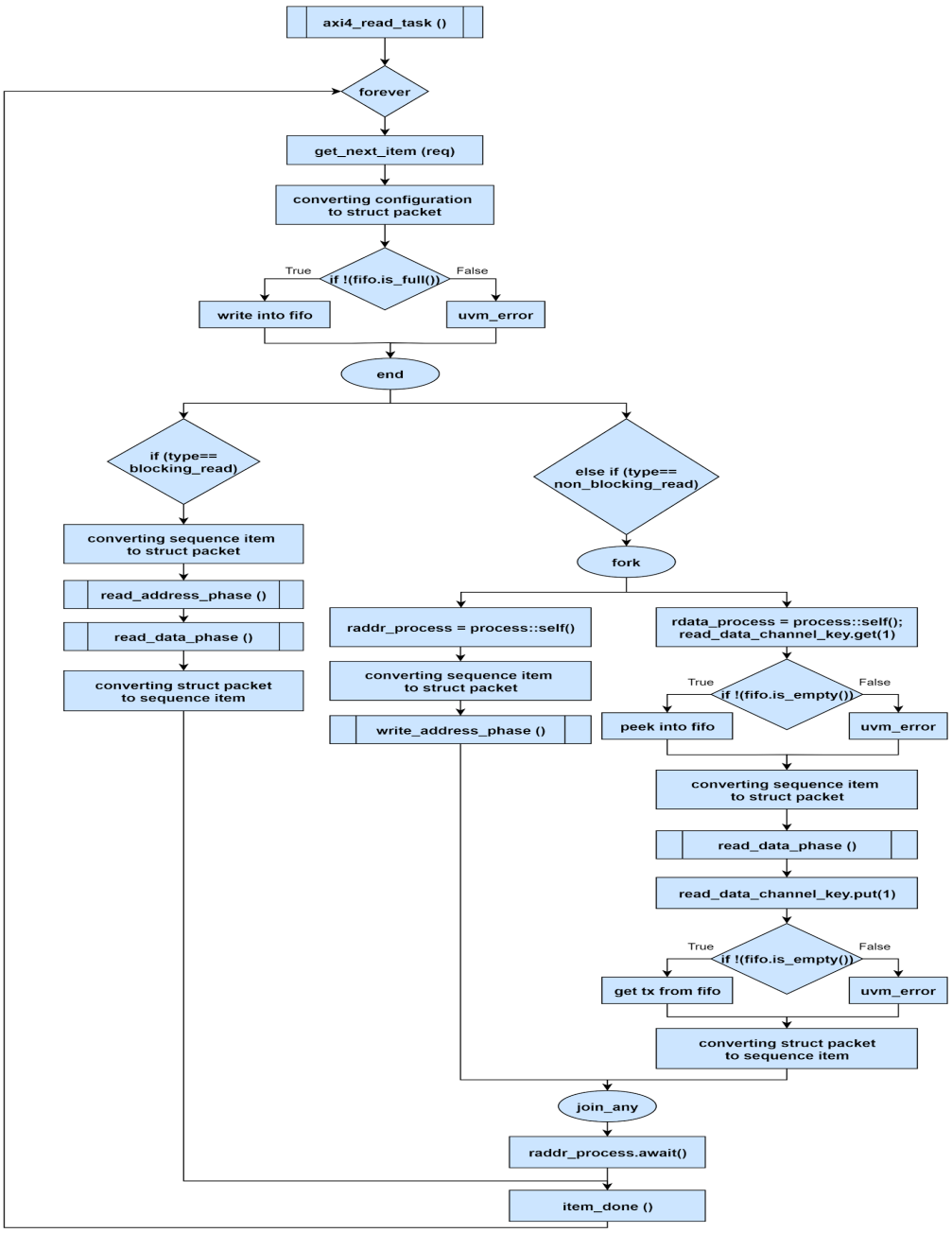


Fig 3.31 flowchart of read_task()

3.2.16 AXI Master Monitor Proxy

AXI4 master monitor proxy component is a class extending uvm_monitor. It gets the axi4 master agent config handle and based on the configurations we will sample the signals. It declares and creates the axi4 master analysis ports to send the sampled data. The axi4 slave monitor proxy will get the sampled data from axi4 master monitor bfm as shown in figure 3.32

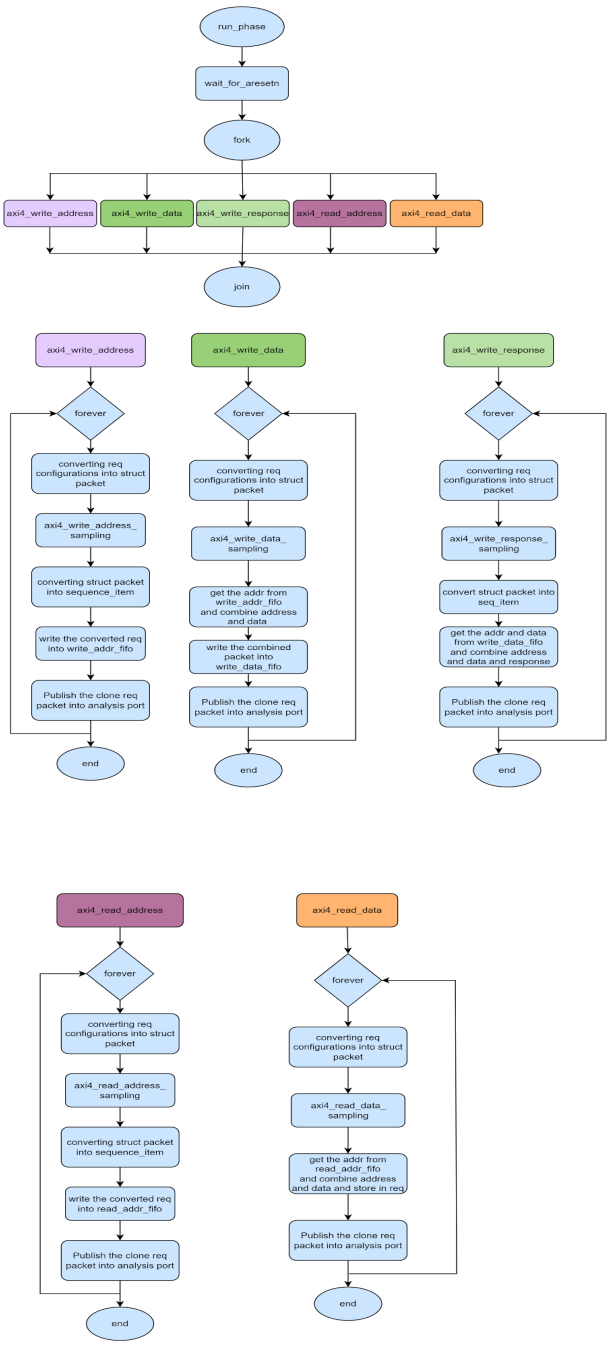


Fig 3.32 flowchart of master_monitor_proxy

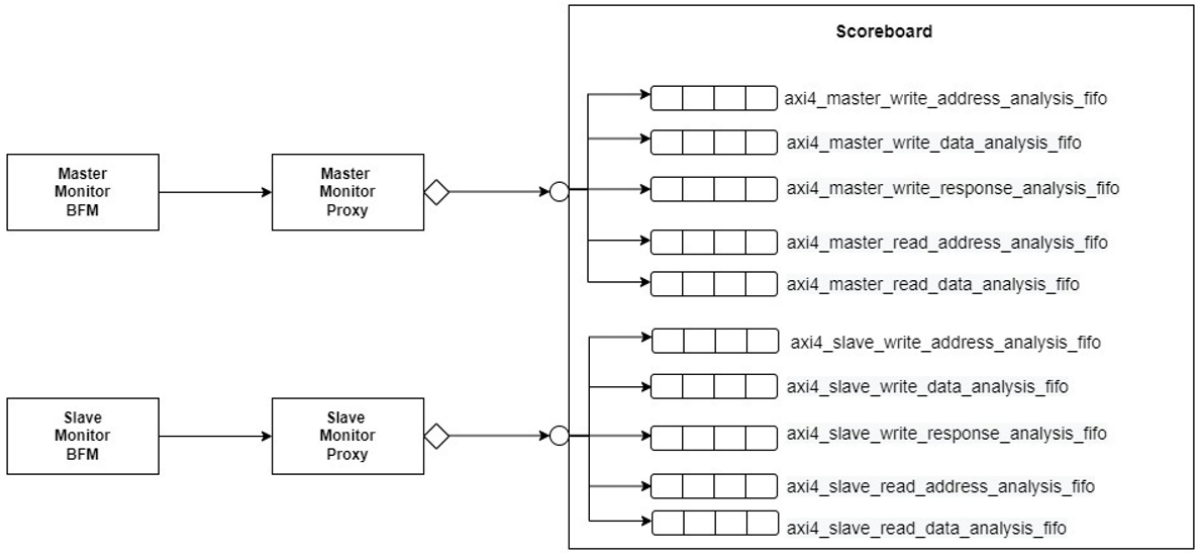


Fig 3.33 Connection between master monitor and slave monitor to scoreboard

The data sampled in the master monitor proxy from master monitor bfm will be combined and sent to scoreboard by using the analysis port. For getting five channels of data into the scoreboard we use five fifos to store the data. The axi4_master_write_address_analysis_fifo gets the write address and stores it, axi4_master_write_data_analysis_fifo gets write data and stores it, axi4_master_write_response_analysis_fifo and stores response in it. Similarly the fifos store the read address in axi4_master_read_address_analysis_fifo and read data in axi4_master_read_data_analysis_fifo respectively. The data received in the fifos are called using the get method and call their respective tasks which compares the master and slave data as shown in fig.3.34. We use semaphore to synchronization the tasks. All the tasks must be done concurrently to get data in upcoming channels as shown in fig 3.34.

The connections made between the master monitor proxy and scoreboard are shown in the above fig.3.33.

```
write_data_key.get(1);  Obtain key from Bucket (Semaphore)
axi4_master_write_data_analysis_fifo.get(axi4_master_tx_h2);  Getting data from Master fifo
`uvm_info(get_type_name(),$sformatf("scoreboard's axi4_master_write_data_channel \n%s",axi4_master_tx_h2.sprint()),UVM_HIGH)
axi4_slave_write_data_analysis_fifo.get(axi4_slave_tx_h2);  Getting data from Slave fifo
`uvm_info(get_type_name(),$sformatf("scoreboard's axi4_slave_write_data_channel \n%s",axi4_slave_tx_h2.sprint()),UVM_HIGH)
axi4_write_data_comparision(axi4_master_tx_h2,axi4_slave_tx_h2);  Sending both master and slave data received from fifo into a task which compares data
axi4_master_tx_wdata_count++;
`uvm_info(get_type_name(),$sformatf("scoreboard's axi4_master_write_data_channel count \n %0d",axi4_master_tx_wdata_count),UVM_
H)
axi4_slave_tx_wdata_count++;
`uvm_info(get_type_name(),$sformatf("scoreboard's axi4_slave_write_data_channel count \n %0d",axi4_slave_tx_wdata_count),UVM_HI

write_data_key.put(1);  Return key into Bucket (Semaphore)
```

Fig 3.34 Semaphore and fios in Scoreboard

3.2.17 AXI Slave Agent

AXI4 slave agent component is a class extending uvm_agent. It gets the axi4 slave agent configuration and based on that we will create and connect the components. It creates the axi4 slave sequencer and axi4 slave driver only if the axi4 slave agent is active which will depend on the value of is_active variable declared in the axi4 slave agent configuration file. The axi4 slave coverage is created in build_phase if has_covergae variable is 1 which is declared in the axi4 slave agent configuration file as shown in fig. 3.35

AXI4 slave agent build phase has creation of,

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

```
function void axi4_slave_agent::build_phase(uvm_phase phase);
  super.build_phase(phase);

 // if(!uvm_config_db #(axi4_slave_agent_config)::get(this,"","axi4_slave_agent_config",axi4_slave_agent_cfg_h)) begin
 //   `uvm_fatal("FATAL_SA_AGENT_CONFIG", $sformatf("Couldn't get the axi4_slave_agent_config from config_db"))
 // end

   if(axi4_slave_agent_cfg_h.is_active == UVM_ACTIVE) begin
     axi4_slave_drv_proxy_h = axi4_slave_driver_proxy::type_id::create("axi4_slave_drv_proxy_h",this);
     axi4_slave_write_seqr_h=axi4_slave_write_sequencer::type_id::create("axi4_slave_write_seqr_h",this);
     axi4_slave_read_seqr_h=axi4_slave_read_sequencer::type_id::create("axi4_slave_read_seqr_h",this);
   end

   axi4_slave_mon_proxy_h = axi4_slave_monitor_proxy::type_id::create("axi4_slave_mon_proxy_h",this);

   if(axi4_slave_agent_cfg_h.has_coverage) begin
    axi4_slave_cov_h = axi4_slave_coverage::type_id::create("axi4_slave_cov_h",this);
   end
endfunction : build_phase
```

Fig 3.35 AXI4 slave agent build phase code snippet

AXI4 slave agent configuration handles declared in the above created components will be mapped here in the connect phase. The axi4 slave driver proxy and axi4 slave sequencer is connected using tlm ports if the axi4 slave agent is active. The axi4 slave coverage's analysis_export will be connected to axi4 slave monitor proxy's slave_analysis_port in connect_phase as shown in fig 3.36.

```
function void axi4_slave_agent::connect_phase(uvm_phase phase);
  super.connect_phase(phase);

  if(axi4_slave_agent_cfg_h.is_active == UVM_ACTIVE) begin
    axi4_slave_drv_proxy_h.axi4_slave_agent_cfg_h = axi4_slave_agent_cfg_h;
    axi4_slave_write_seqr_h.axi4_slave_agent_cfg_h = axi4_slave_agent_cfg_h;
    axi4_slave_read_seqr_h.axi4_slave_agent_cfg_h = axi4_slave_agent_cfg_h;
    axi4_slave_cov_h.axi4_slave_agent_cfg_h = axi4_slave_agent_cfg_h;

    // Connecting the ports
    axi4_slave_drv_proxy_h.axi_write_seq_item_port.connect(axi4_slave_write_seqr_h.seq_item_export);
    axi4_slave_drv_proxy_h.axi_read_seq_item_port.connect(axi4_slave_read_seqr_h.seq_item_export);
  end

  if(axi4_slave_agent_cfg_h.has_coverage) begin
    axi4_slave_cov_h.axi4_slave_agent_cfg_h = axi4_slave_agent_cfg_h;
    // Connecting monitor_proxy port to coverage export
    axi4_slave_mon_proxy_h.axi4_slave_read_address_analysis_port.connect(axi4_slave_cov_h.analysis_export);
    axi4_slave_mon_proxy_h.axi4_slave_read_data_analysis_port.connect(axi4_slave_cov_h.analysis_export);
    axi4_slave_mon_proxy_h.axi4_slave_write_address_analysis_port.connect(axi4_slave_cov_h.analysis_export);
    axi4_slave_mon_proxy_h.axi4_slave_write_data_analysis_port.connect(axi4_slave_cov_h.analysis_export);
    axi4_slave_mon_proxy_h.axi4_slave_write_response_analysis_port.connect(axi4_slave_cov_h.analysis_export);
  end

  axi4_slave_mon_proxy_h.axi4_slave_agent_cfg_h = axi4_slave_agent_cfg_h;

endfunction: connect_phase
```

Fig 3.36 AXI4 slave agent connect phase code snippet

3.2.18 AXI Slave Sequencer

AXI4 slave sequencer component is a parameterised class of type axi4 slave transaction, extending uvm_sequencer. AXI4 sequencer sends the data from the axi4 slave sequences to the axi4 driver proxy as shown in fig. 3.37.

3.2.19 AXI Slave Driver Proxy

Axi4 slave driver proxy has the connection to slave driver bfm in the end of elaboration phase. The slave driver proxy run phase has the write and read tasks calls in parallel using fork join.

Write task:

Write address:

p 1: The write address channel uses a fine grain concept called pSterocess addr_tx to control the address phase which is in fork join.
Step 2 : Convert the transactions , configurations into structure type,
Step 3: Sample the write address phase from the bfm.
Step 4: Converting the structure type into the transaction type,
Step 5: Then put the sampled write address into fifo.

Write data:

Step 1: The write data channel uses a fine grain concept called process data_tx to check the status of the write data phase which is in fork join.
Step 2: Get a semaphore key and get data from input fifo.
Step 3: Convert the transactions , configurations into structure type,

Step 4: Sample the write data phase from the bfm.
Step 5: Converting the structure type into the transaction type,
Step 6 : Then put the sampled write data into output fifo and put back the semaphore key

Write Response:

Step 1: The write response channel uses a fine grain concept called process response_tx to check the status of the write response phase which is in fork join.
Step 2 : Get a semaphore key and get a response from fifo.
Step 3 : Convert the transactions , configurations into structure type,
Step 4 : Drive the write response phase from the bfm.
Step 5: Convert the structure type into the transaction type, and get write address and write data from the fifo and combine write address,write data and write response into Packet

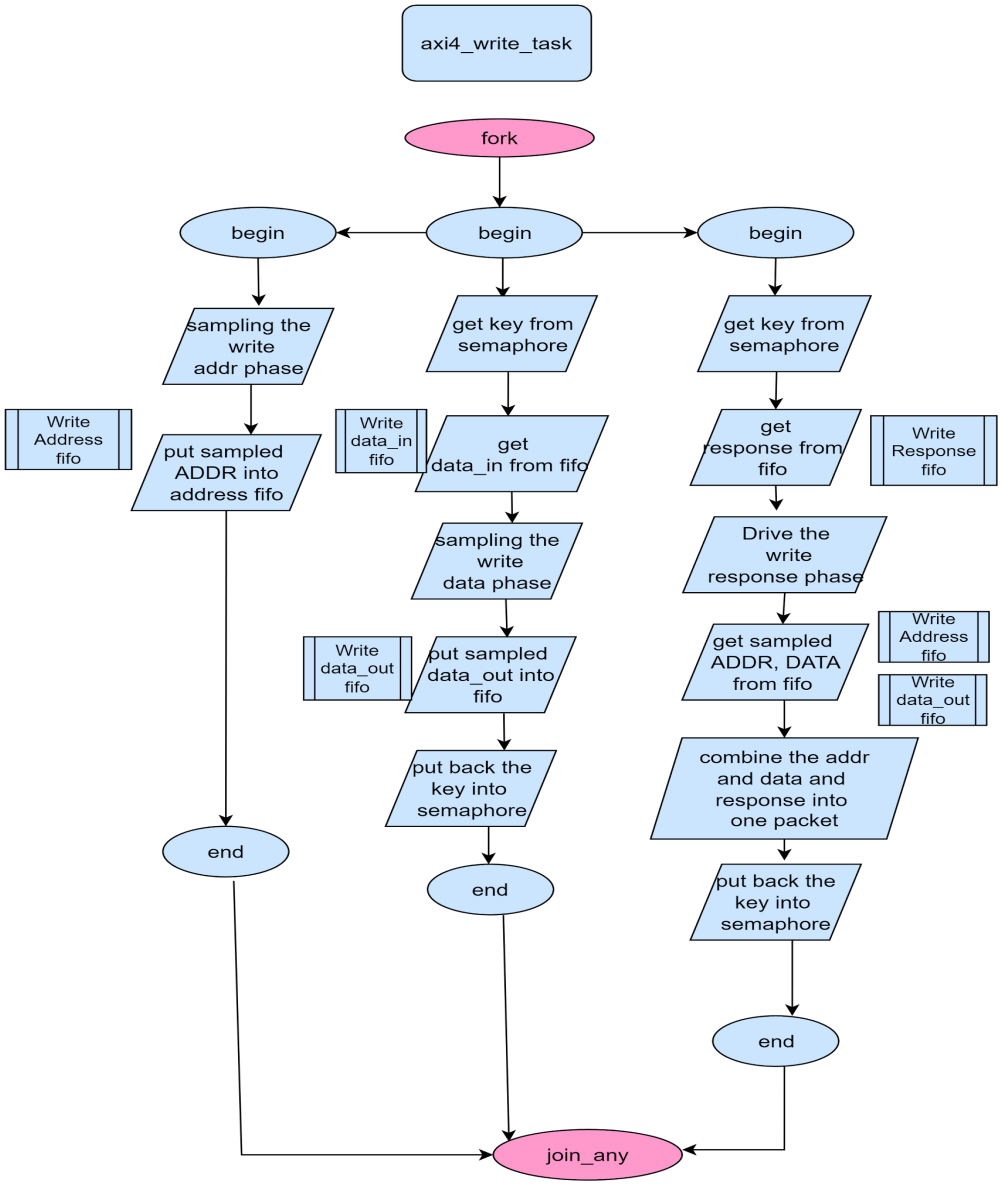


Fig 3.37 Flow chart for slave driver proxy write task using semaphore

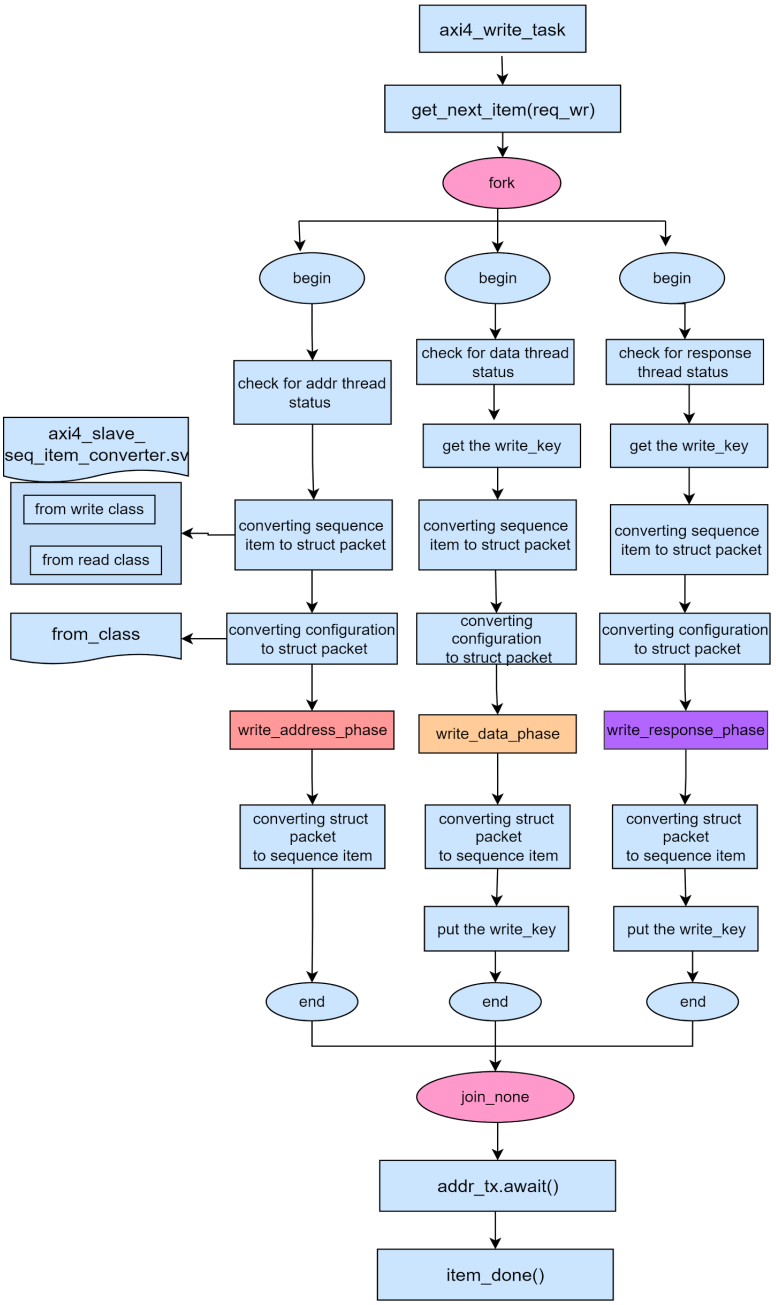


Fig 3.38 Slave driver proxy write_task

Read Task:

Read Address:

Step1 : The read channel uses a fine grain concept called process rd_addr to control the read address phase which is in fork join.

Step 2 : Convert the transactions , configurations into structure type,

Step 3: Sample the read address phase from the bfm.

Step 4: Converting the structure type into the transaction type,

Step 5: Then put the sampled read address into fifo.

Read data:

Step 1: The read data channel uses a fine grain concept called process rd_data to check the status of the read data phase which is in the fork join.
Step 2 : Get a semaphore key and get data from fifo.
Step 3 : Convert the transactions , configurations into structure type,
Step 4 : Drive the read data phase from the bfm.
Step 5 : Convert the structure type into the transaction type, and get the read address and read data from the fifo and Combine the read address and read data into Packet.
Step 6 :Then Put back the semaphore key.

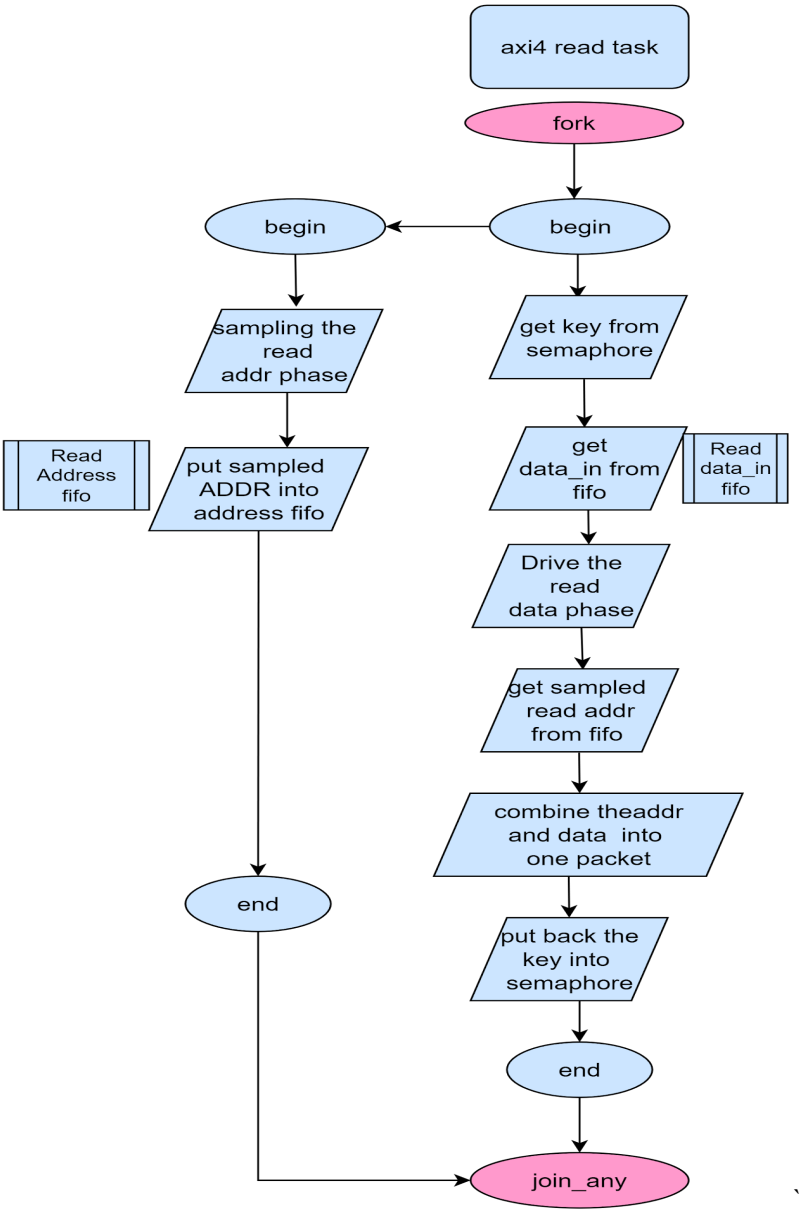


Fig 3.39 Flowchart for slave driver proxy read task using semaphore

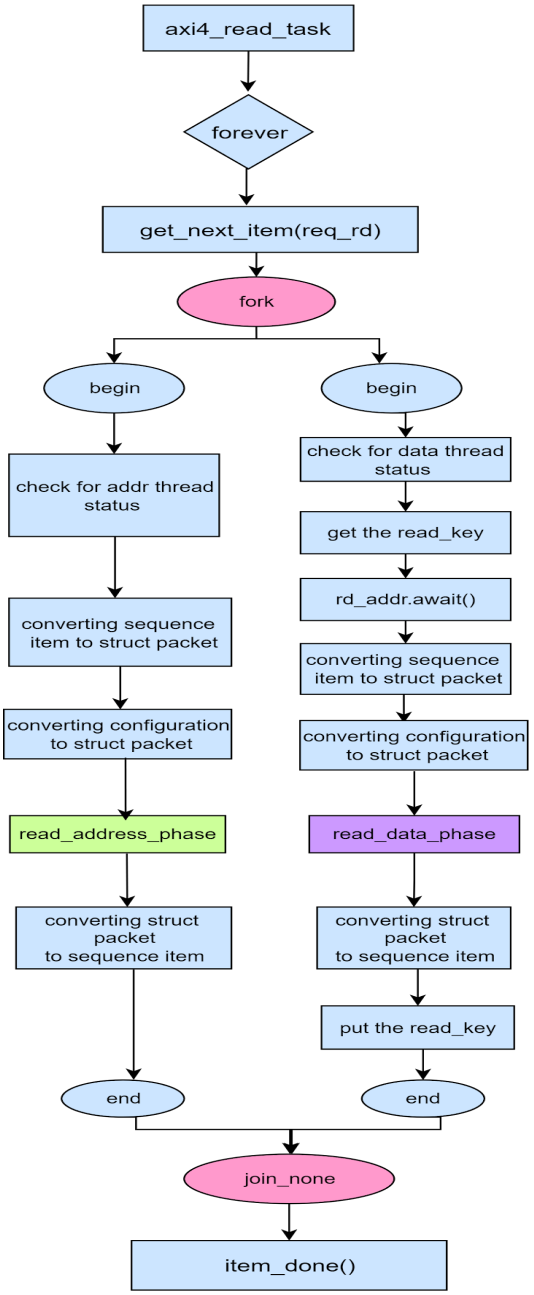


Fig 3.40 Slave driver proxy read_task

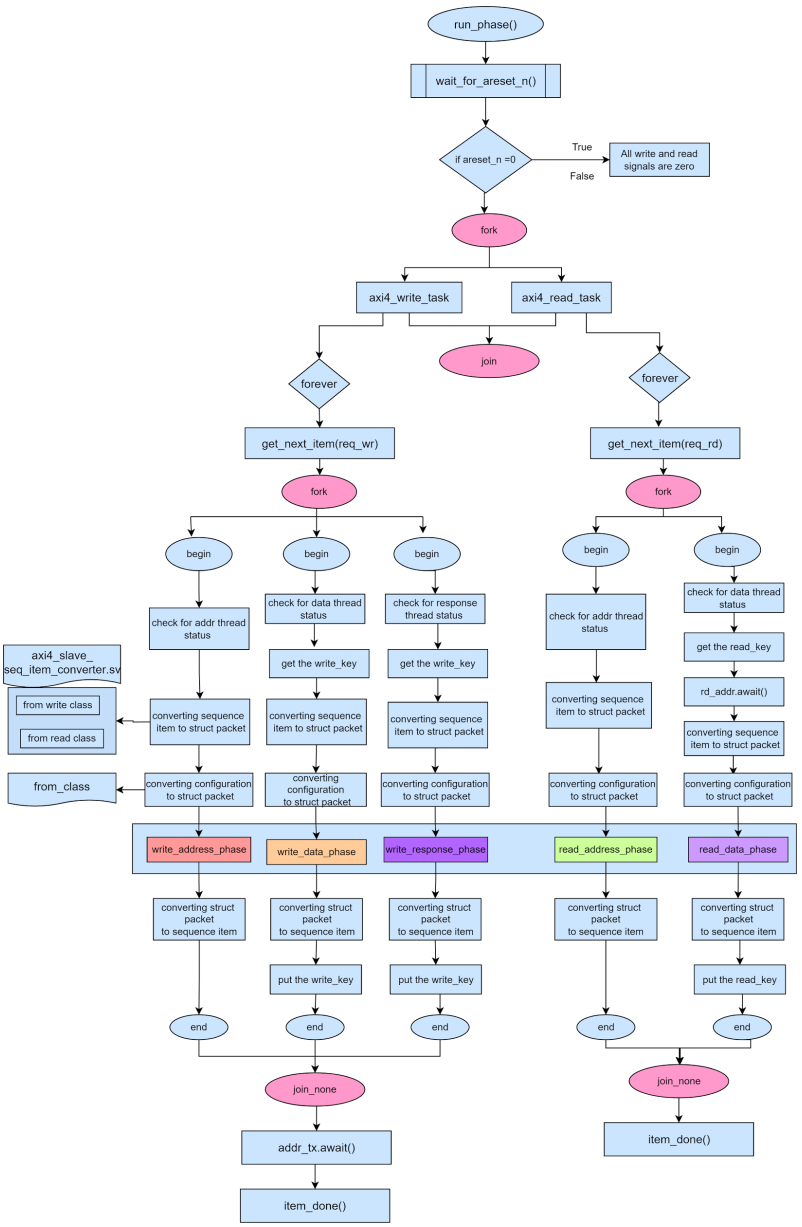


Fig 3.41 AXI4 slave driver proxy Flow chart for write and read

3.2.20 AXI Slave Monitor Proxy

Axi4 slave monitor proxy component is a class extending uvm_monitor. It gets the Axi4 slave agent config handle and based on the configurations we will sample the signals. It declares and creates the axi4 slave analysis port to send the sampled data. The axi4 slave monitor proxy will get the sampled data from axi4 master monitor bfm as shown in figure 3.42.

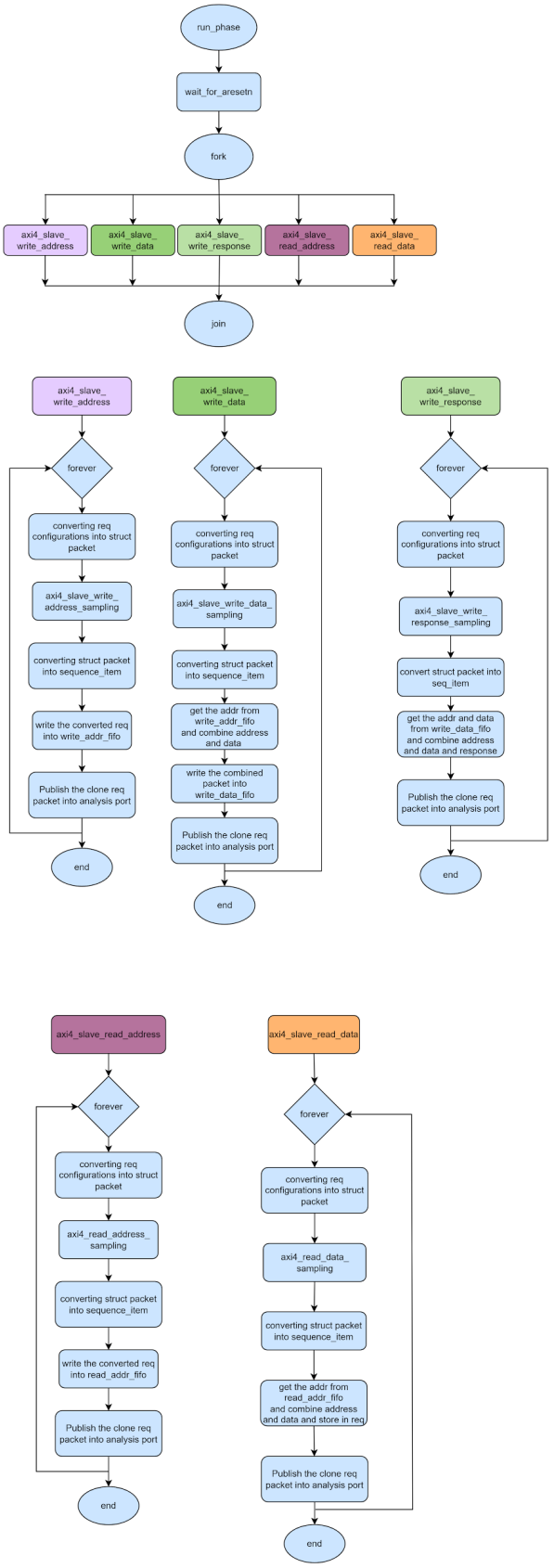


Fig 3.42 flowchart of slave_monitor_proxy

The connections made between the slave monitor proxy and scoreboard are shown in the above fig.3.33.

The data sampled in the slave monitor proxy from slave monitor bfm will be combined and sent to scoreboard by using the analysis port. For getting five channels of data into the scoreboard we use five fifos to store the data. The axi4_slave_write_address_analysis_fifo gets the write address and stores it, axi4_slave_write_data_analysis_fifo gets write data and stores it, axi4_slave_write_response_analysis_fifo and stores response in it. Similarly the fifos store the read address in axi4_slave_read_address_analysis_fifo and read data in axi4_slave_read_data_analysis_fifo respectively. The data received in the fifos are called using the get method and call their respective tasks which compares the master and slave data as shown in fig.3.34. We use semaphore to synchronization the tasks. All the tasks must be done concurrently to get data in upcoming channels as shown in fig 3.34.

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 3.2

Table 3.2 UVM verbosity Priorities

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

Chapter 4

Directory Structure

4.1.Package Content

The package structure diagram navigates users to find out the file locations, where it is located and which folder as shown in fig. 4.1

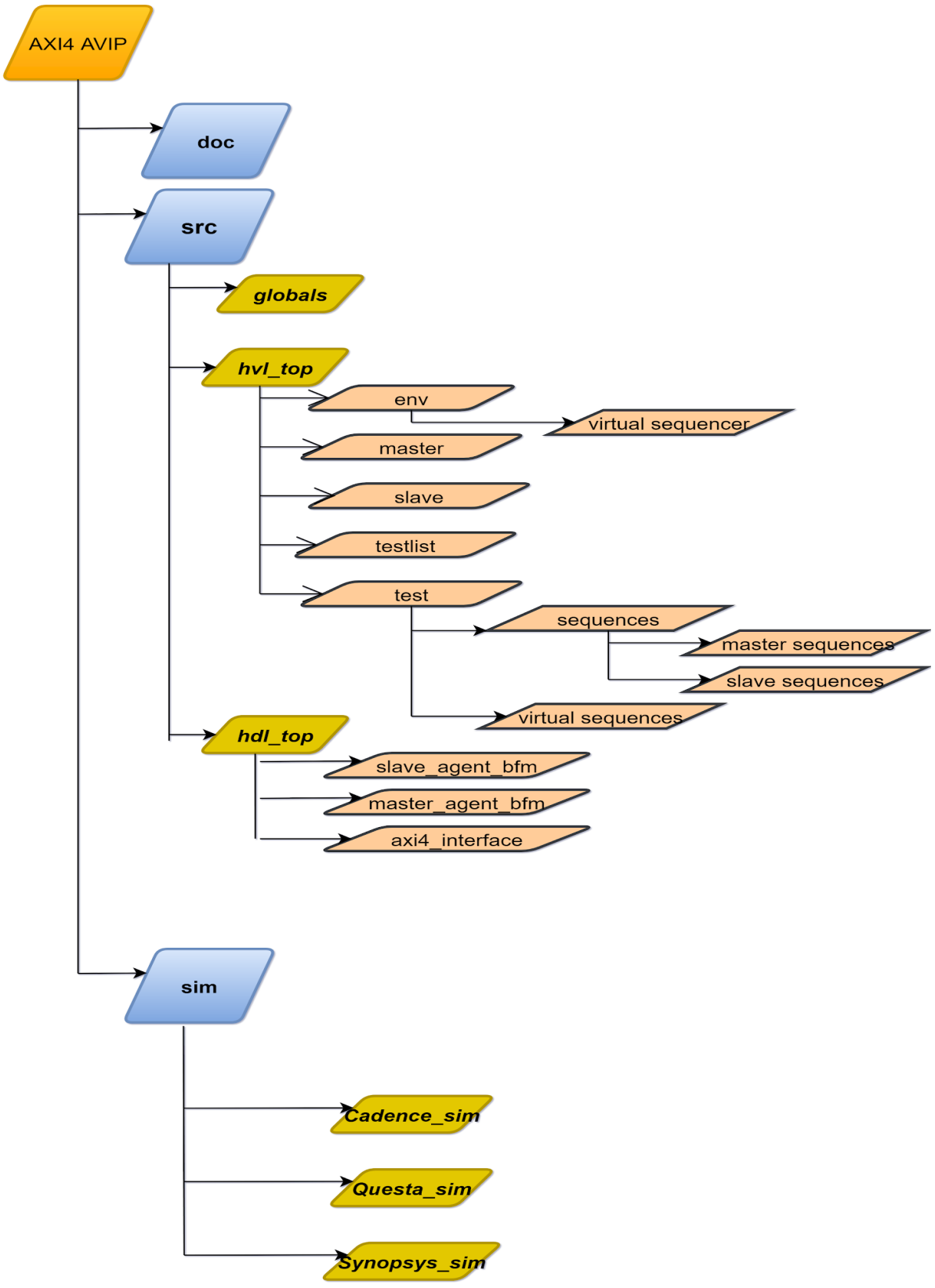


Fig. 4.1. Package Structure of AXI4_AVIP

Table 4.1 Directory Path

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

Chapter 5

Configuration

5.1 Global package variables

The global variables declared in the global package file are given in this chapter.

Table 5.1 Global package variables

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

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

Configuration used

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

5.2 Master agent configuration

Table 5.2 Master_agent_config

Name	Type	Default value	Description
is_active	enum	UVM_ACTIVE	It will be used for configuring an agent as an active agent means it has sequencer, driver and monitor or passive agent which has monitor only.
has_coverage	integer	'd1	Used for enabling the master agent coverage

5.3 Slave agent configuration

Table 5.3 Slave_agent_config

Name	Type	Default value	Description
is_active	enum	UVM_ACTIVE	It will be used for configuring agent as an active agent means it has sequencer,driver and monitor and if it's a passive agent then it will have only monitor
has_coverage	integer	'd1	Used for enabling the slave agent coverage.

5.4 Environment configuration

Table 5.4 Env_config

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

5.5 Memory Mapping

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

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

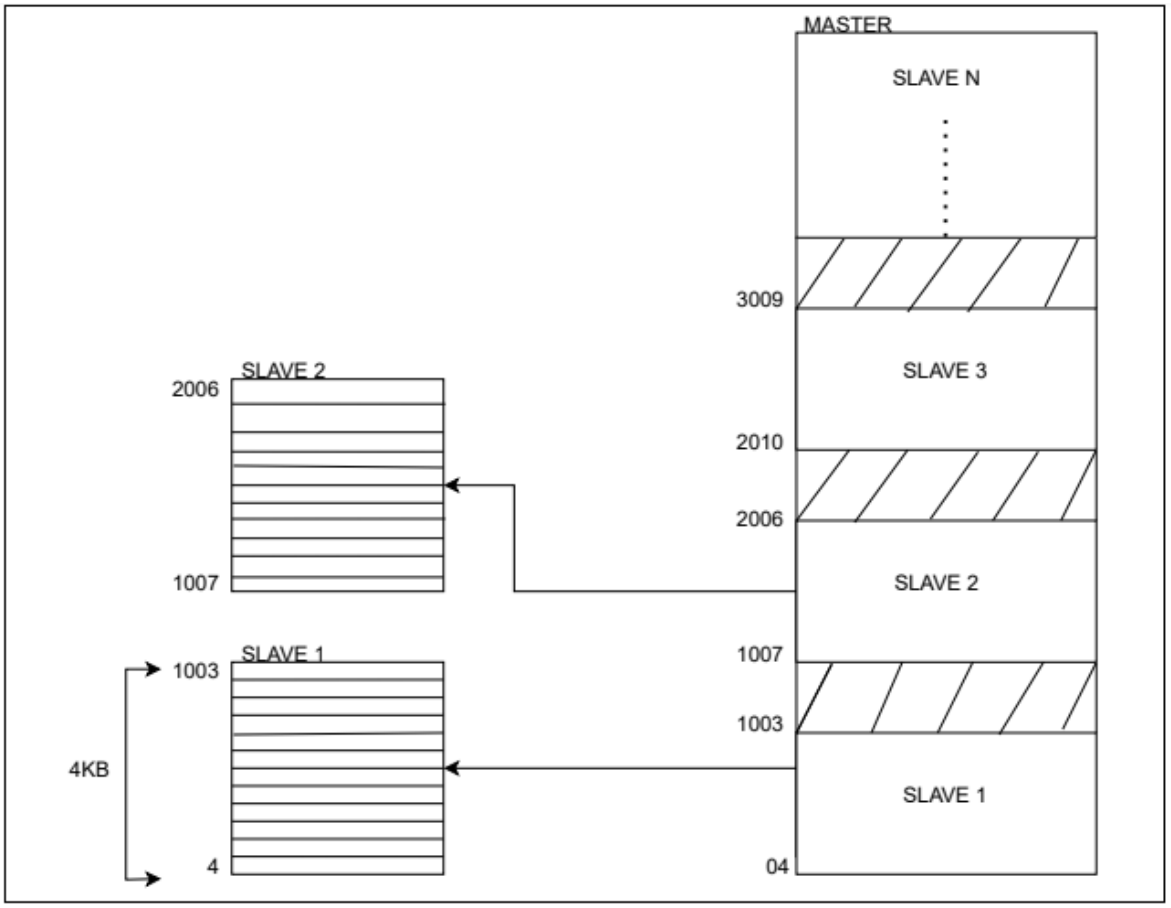


Fig. 5.5.1: Memory mapping example

Memory mapping in AXI

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

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

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

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

Fig. 5.5.2: Global parameter declaration

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

```
//Variable : master_memory
//Used to store all the data from the slaves
//Each location of the master memory stores 32 bit data
bit [MEMORY_WIDTH-1:0]master_memory[(SLAVE_MEMORY_SIZE+SLAVE_MEMORY_GAP)*NO_OF_SLAVES:0];

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

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

Fig.5.5.3: Associative array declaration

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

```
function void axi4_master_agent_config::master_max_addr_range(int slave_number, bit[ADDRESS_WIDTH-1:0]slave_max_address_range);
  master_max_addr_range_array[slave_number] = slave_max_address_range;
endfunction : master_max_addr_range

//-------------------------------------------------------------------------------------------
// Function : master_min_addr_range_array
// Used to store the minimum address ranges of the slaves in the array
// Parameters :
//  slave_number           - int
//  slave_min_address_range - bit [63:0]
//-------------------------------------------------------------------------------------------
function void axi4_master_agent_config::master_min_addr_range(int slave_number, bit[ADDRESS_WIDTH-1:0]slave_min_address_range);
  master_min_addr_range_array[slave_number] = slave_min_address_range;
endfunction : master_min_addr_range
```

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

The memory mapping is done in base_test as shown in fig. 5. In a setup_axi4_master_agent_config(), initially, we declare 2 local variables to store the min and max address used for each iteration as shown in fig. 5.5.5.

Fig 5.5.5 Local variable declaration in function

The function setup_axi4_master_agent_cfg will start pushing the maximum and minimum address ranges to the respective associative arrays by adding a memory gap of 4 and making sure that start address is mod of 4 as shown in fig. 5.5.6.

```
  for(int i =0; i<NO_OF_SLAVES; i++) begin
    if(i == 0) begin
      axi4_env_cfg_h.axi4_master_agent_cfg_h[i].master_min_addr_range(i,0);
      local_min_address = axi4_env_cfg_h.axi4_master_agent_cfg_h[i].master_min_addr_range_array[i];
      axi4_env_cfg_h.axi4_master_agent_cfg_h[i].master_max_addr_range(i,2**(SLAVE_MEMORY_SIZE)-1 );
      local_max_address = axi4_env_cfg_h.axi4_master_agent_cfg_h[i].master_max_addr_range_array[i];
    end
    else begin
      axi4_env_cfg_h.axi4_master_agent_cfg_h[i].master_min_addr_range(i,local_max_address + SLAVE_MEMORY_GAP);
      local_min_address = axi4_env_cfg_h.axi4_master_agent_cfg_h[i].master_min_addr_range_array[i];
      axi4_env_cfg_h.axi4_master_agent_cfg_h[i].master_max_addr_range(i,local_max_address+ 2**(SLAVE_MEMORY_SIZE)-1 +
                                                                      SLAVE_MEMORY_GAP);
      local_max_address = axi4_env_cfg_h.axi4_master_agent_cfg_h[i].master_max_addr_range_array[i];
    end
    `uvm_info(get_type_name(),$sformatf("\nAXI4_MASTER_CONFIG[%0d]\n%s",i,axi4_env_cfg_h.axi4_master_agent_cfg_h[i].sprint()),UVM_L
OW);
  end
```

Fig 5.5.6: Memory mapping procedure in master agent configuration

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

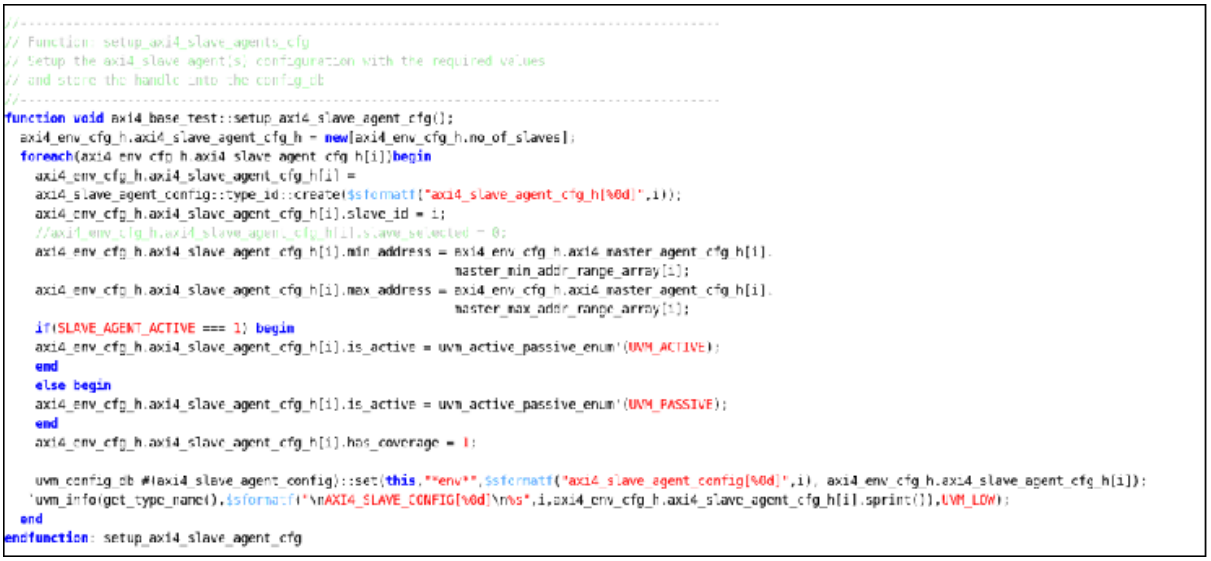
```
//Variable : max_address
//Used to store the maximum address value of this slave
bit [ADDRESS_WIDTH-1:0]max_address;

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

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

Fig 5.5.7: Declaration of slave max and min address range

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



```
//-------------------------------------------------------------------------------------
// Function: setup_axi4_slave_agents_cfg
// Setup the axi4 slave agent(s) configuration with the required values
// and store the handle into the config_db
//-------------------------------------------------------------------------------------
function void axi4_base_test::setup_axi4_slave_agent_cfg();
  axi4_env_cfg_h.axi4_slave_agent_cfg_h = new[axi4_env_cfg_h.no_of_slaves];
  foreach(axi4_env_cfg_h.axi4_slave_agent_cfg_h[i])begin
    axi4_env_cfg_h.axi4_slave_agent_cfg_h[i] =
    axi4_slave_agent_config::type_id::create($sformatf("axi4_slave_agent_cfg_h[%0d]",i));
    axi4_env_cfg_h.axi4_slave_agent_cfg_h[i].slave_id = i;
    //axi4_env_cfg_h.axi4_slave_agent_cfg_h[i].slave_selected = 0;
    axi4_env_cfg_h.axi4_slave_agent_cfg_h[i].min_address = axi4_env_cfg_h.axi4_master_agent_cfg_h[i].
                                                           master_min_addr_range_array[i];
    axi4_env_cfg_h.axi4_slave_agent_cfg_h[i].max_address = axi4_env_cfg_h.axi4_master_agent_cfg_h[i].
                                                           master_max_addr_range_array[i];
    if(SLAVE_AGENT_ACTIVE === 1) begin
    axi4_env_cfg_h.axi4_slave_agent_cfg_h[i].is_active = uvm_active_passive_enum'(UVM_ACTIVE);
    end
    else begin
    axi4_env_cfg_h.axi4_slave_agent_cfg_h[i].is_active = uvm_active_passive_enum'(UVM_PASSIVE);
    end
    axi4_env_cfg_h.axi4_slave_agent_cfg_h[i].has_coverage = 1;

    uvm_config_db #(axi4_slave_agent_config)::set(this,"*env*",$sformatf("axi4_slave_agent_config[%0d]",i), axi4_env_cfg_h.axi4_slave_agent_cfg_h[i]);
   `uvm_info(get_type_name(),$sformatf("\nAXI4_SLAVE_CONFIG[%0d]\n%s",i,axi4_env_cfg_h.axi4_slave_agent_cfg_h[i].sprint()),UVM_LOW);
  end
endfunction: setup_axi4_slave_agent_cfg
```

Fig.5.5.8: Memory mapping procedure in slave agent configuration

Chapter 6

Verification Plan

6.1 Verification plan

Verification Plan Link:

axi4 avip vplan

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

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

Refer the below link for AXI Specifications:

AXI4 Specifications

AXI4 Specifications(Arm)

AXI4 Write Channel Transfers

1. Write data Transfers
2. Burst Type Transfers
    1) Fixed
    2) INCR
    3) WRAP
3. Write Response
4. Locked Transfers
5. Quality of Service

AXI4 Read Channel Transfers

1. Read data Transfers
2. Burst Type Transfers
    1) FIXED
    2) INCR
    3) WRAP
3. Read Response

4. Locked Transfers
5. Quality of Service

Outstanding Addresses:

When the master sends the next transaction without waiting to complete the previous transaction.

Addressing Types

1) Aligned
2) Unaligned

Memory Access

1) Big Endian
2) Little Endian
3) Byte Invariance

TO_DO:

Out of order:

Response doesn't need to come in the same order as the master sent, only if id is different to the transactions(based on different ID's to the same slave)

Memory Access

1) Big Endian
2) Byte Invariance

Protection

1) Un Privilege/Privilege
2) Secure/Non Secure
3) Data/Instruction

Cache Memory Access

1) Bufferable
2) Modifiable
3) Write Allocate
4) Read Allocate

6.2 Template of Verification Plan

Verification Plan Link:

axi4 avip vplan

In the below Figure

Section A shows the S.No

Section B shows the Features

Section C shows the Sub Features

S.No	Sections	Features	Sub-Features	Description
A	Directed Testcases			
1	Basic Write Transfers	Burst type	i.FIXED	Starting address remains same throughtout the transfers
			ii.INCR	Address of each transfer increments depends on previous address
			iii.WRAP	Based on lower wrap boundary and upper wrap boundary, address will be incremented
			(Depends on AWLEN, AWSIZE)	
2	Transfers	Locked Transfers	i.Normal access	Based on prioritization data can be read or write
			ii.Exclusive access	Master send exclusive access saying that no another master should send a request to the same address while performing exclusive access
3	QOS	Quality of service (depends on Axlock)		Prioritize the transactions based on QoS value

Fig 6.2.1 Verification plan Template

Section D represents the Description

Sections	Features	Sub-Features	Description
Directed Testcases			
Basic Write Transfers	Burst type	i.FIXED	Starting address remains same throughtout the transfers
		ii.INCR	Address of each transfer increments depends on previous address
		iii.WRAP	Based on lower wrap boundary and upper wrap boundary, address will be incremented
		(Depends on AWLEN, AWSIZE)	
Transfers	Locked Transfers	i.Normal access	Based on prioritization data can be read or write
		ii.Exclusive access	Master send exclusive access saying that no another master should send a request to the same address while performing exclusive access

Fig 6.2.2 Verification plan Section D is Description for tests

Section G and H shows the Test Cases names and Status for heading(Directed Test cases)

G	H	I
Non Blocking Testcases Names(Write and Read)	**Blocking Test Status**	**Non Blocking Test Status**
axi4_non_blocking_fixed_burst_write_read_test	DONE	DONE
axi4_non_blocking_incr_burst_write_read_test	DONE	DONE
axi4_non_blocking_wrap_burst_write_read_test	DONE	DONE

Fig 6.2.3 Verification plan Section G and H is for Test Names and Status

6.3 Sections for different test Scenarios

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

6.3.1 Directed test

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

Directed test names for Blocking:

This tests describes the different combinations of number of bits transfer for Blocking

Table 6.1 : Directed test names for Blocking Transfers

S.NO	Test names	Description
1	axi4_blocking_8b_write_read_test	Checking the 8 bit transfer
2	axi4_blocking_16b_write_read_test	Checking the 16bit transfer
3	axi4_blocking_32b_write_read_test	Checking the 24bit transfer
4	axi4_blocking_64b_write_read_test	Checking the 32 bit transfers
5	axi4_blocking_fixed_burst_write_read_test	Checking for fixed burst transfer
6	axi4_blocking_incr_burst_write_read_test	Checking for incr burst transfer

7	axi4_blocking_wrap_burst_write_read_test	Checking for wrap burst transfer
8	axi4_blocking_slave_error_write_read_test	Checking for slave error response
9	axi4_blocking_unaligned_addr_write_read_test	Checking for unaligned address

Directed test names for Non Blocking:

This tests describes the different combinations of number of bits transfer for Non Blocking

Table 6.2 : Directed test names for Non Blocking Transfers

S.NO	Test names	Description
1	axi4_non_blocking_8b_write_read_test	Checking the 8 bit transfer
2	axi4_non_blocking_16b_write_read_test	Checking the 16bit transfer
3	axi4_non_blocking_32b_write_read_test	Checking the 24bit transfer
4	axi4_non_blocking_64b_write_read_test	Checking the 32 bit transfers
5	axi4_non_blocking_fixed_burst_write_read_test	Checking for fixed burst transfer
6	axi4_non_blocking_incr_burst_write_read_test	Checking for incr burst transfer
7	axi4_non_blocking_wrap_burst_write_read_test	Checking for wrap burst transfer
8	axi4_non_blocking_slave_error_write_read_test	Checking for slave error response
9	axi4_non_blocking_okay_response_write_read_test	Checking fo Okay response of write read transfers
10	axi4_non_blocking_unaligned_addr_write_read_test	Checking for unaligned address

6.3.2 Random test

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

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

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

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

Random test name

This tests describes the random write read transfers for Blocking

Table 6.3 : Random test name for Blocking Transfers

S.NO	Test names	Description
1	axi4_blocking_write_read_rand_test	Checking the random write read transfers

This tests describes the random write read transfers for Non Blocking

Table 6.4 : Random test name for Non Blocking Transfers

S.NO	Test names	Description
1	axi4_non_blocking_write_read_rand_test	Checking the random write read transfers

6.3.3 Cross test

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

This tests describes the cross test for Axlength, Axsize and Axburst write read transfers for Non Blocking

Table 6.5 : Cross test name for Non Blocking Transfers

S.NO	Test names	Description
1	axi4_non_blocking_cross_write_read_test	Checking the cross of AxLength X Axburst and Axsize of write read transfers

For more information about Verification plan refer below link

axi4 avip vplan

Chapter 7

Assertion Plan

7.1 Assertion Plan overview

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

7.1.1 What are assertions?

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

7.1.2 Why do we use it?

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

7.1.3 Benefits of Assertions

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

7.2 Template of Assertion Plan

Template for Assertion plan is done in an excel sheet and refer to link below: axi4_avip_assertion_plan

7.3 Assertion Condition

```
//------------------------------------------------------------------------------------------
// Assertion properties written for various checks in write address channel
//------------------------------------------------------------------------------------------
//Assertion:   AXI_WA_STABLE_SIGNALS_CHECK
//Description: All signals must remain stable after AWVALID is asserted until AWREADY IS LOW
property if_write_address_channel_signals_are_stable;
  @(posedge aclk) disable iff (!aresetn)
  (awvalid==1 && awready==0) |=> ($stable(awid) && $stable(awaddr) && $stable(awlen) && $stable(awsize) &&
                                  $stable(awburst) && $stable(awlock) && $stable(awcache) && $stable(awprot));
endproperty : if_write_address_channel_signals_are_stable
AXI_WA_STABLE_SIGNALS_CHECK: assert property (if_write_address_channel_signals_are_stable);
```

Fig. 7.1 Assertion code for stable signals check

Property AXI_WA_STABLE_SIGNALS_CHECK is evaluated as follows:

- Initially it will check for posedge of aclk and aresetn should be high.
- When awvalid is high and awready is low ,at the same cycle it will check for awid,awaddr,awlen,awsize,awburst,awlock,awcache and awprot signal should be stable. Then property is true.
- Otherwise the property will fail.

7.3.2. AXI_WA_UNKNOWN_SIGNALS_CHECK

A value of X on signals is not permitted when **AWVALID** is high as shown in fig. 7.2

```
//Assertion:   AXI_WA_UNKNOWN_SIGNALS_CHECK
//Description: A value of X on signals is not permitted when AWVALID is HIGH
property if_write_address_channel_signals_are_unknown;
  @(posedge aclk) disable iff (!aresetn)
  (awvalid==1) |-> (!($isunknown(awid)) && !($isunknown(awaddr)) && !($isunknown(awlen)) && !($isunknown(awsize))
                && !($isunknown(awburst)) && !($isunknown(awlock)) && !($isunknown(awcache)) && !($isunknown(awprot)));
endproperty : if_write_address_channel_signals_are_unknown
AXI_WA_UNKNOWN_SIGNALS_CHECK: assert property (if_write_address_channel_signals_are_unknown);
```

Fig. 7.2 Assertion code for unknown signals check

Property AXI_WA_UNKNOWN_SIGNALS_CHECK is evaluated as follows:

- Initially it will check for posedge of aclk and aresetn should be high.
- When awvalid is high,at the same cycle it will check for awid,awaddr,awlen,awsize,awburst,awlock,awcache and awprot signal should be unknown. Then property is true.
- Otherwise the property will fail.

7.3.3. AXI_WA_VALID_STABLE_CHECK

When **AWVALID** is asserted, then it must remain asserted until **AWREADY** is high as shown in fig. 7.3

Fig. 7.3 Assertion code for valid stable check

```
//Assertion:   AXI_WA_VALID_STABLE_CHECK
//Description: When AWVALID is asserted, then it must remain asserted until AWREADY is HIGH
//Assertion stays asserted from the time awvalid becomes high and till awready becomes high using s_until_with keyword
property axi_write_address_channel_valid_stable_check;
  @(posedge aclk) disable iff (!aresetn)
  $rose(awvalid) |-> awvalid s_until_with awready;
endproperty : axi_write_address_channel_valid_stable_check
AXI_WA_VALID_STABLE_CHECK : assert property (axi_write_address_channel_valid_stable_check);
```

Property AXI_WA_VALID_STABLE_CHECK is evaluated as follows:

- Initially it will check for posedge of aclk and aresetn should be high.
- When awvalid is changed from zero to high,at the same cycle it will check for awvalid should be high until awready signal is high. Then property is true.
- Otherwise the property will fail.

Chapter 8

Coverage

8.1 Functional Coverage

- Functional coverage is the coverage data generated from the user defined functional coverage model and assertions usually written in System Verilog During simulation, the simulator generates functional coverage based on the stimulus. Looking at the functional coverage data, one can identify the portions of the DUT [Features] verified. Also, it helps us to target the DUT features that are unverified.
- The reason for switching to the functional coverage is that we can create the bins manually as per our requirement while in the code coverage it is generated by the system by itself.

8.2 Uvm_Subscriber

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

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

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

uvm_analysis_imp #(T, this_type) analysis_export;

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

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

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

 pure virtual function void write(T t);

 endclass
```

Figure8.1 . Uvm_subscriber

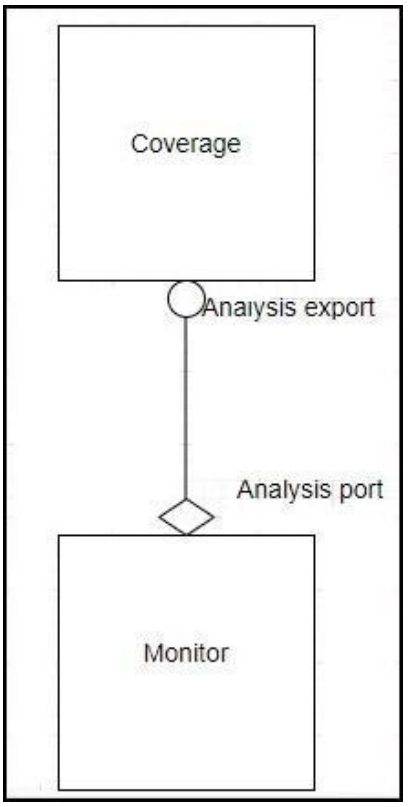


Figure8.2 . Monitor and coverage connection

8.2.1 Analysis export

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

8.2.2 Write function

The write function is to process the incoming transactions.

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

  axi4_master_covergroup.sample(axi4_master_agent_cfg_h,t);

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

Figure 8.3. Write function

8.3 Covergroup

```
covergroup axi4_master_covergroup with function sample (axi4_master_agent_config cfg, axi4_master_tx packet);
 option.per_instance = 1;
     3                          1                              2
 //-------------------------------------------------------
 // Write channel signals
 //-------------------------------------------------------

 AWLEN_CP : coverpoint packet.awlen {
  option.comment = "awlen";  4
  bins AW_LEN[16]={[0:$]};
 }
```

Figure 8.4. Covergroup

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

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

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

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

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

Questa Covergroup Coverage Report

Search: cvg:axi4_master_covergroup

Covergroups/Instances	Total Bins	Hits	Misses	Hits %	Goal %	Coverage %
Covergroup axi4_master_covergroup	2336	47	2289	2.01%	22.81%	22.81%
Instance \/axi4_master_pkg::axi4_master....erage::axi4_master_covergroup	2336	47	2289	2.01%	22.81%	22.81%

Figure 8.5. option.per_instance

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

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

Coverpoint: ARBURST_CP

Comment:
arburst

Search:

Bin Name	At Least	Hits
ARBURST[READ_FIXED]	1	12
ARBURST[READ_INCR]	1	0
ARBURST[READ_WRAP]	1	0
ARBURST[READ_RESERVED]	1	0

Figure 8.6 option.comment

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

8.4 Coverpoints

There we are created the bins based on the write_address_length, write_address_size, write_address_burst

```
AWLEN_CP : coverpoint packet.awlen {
  option.comment = "awlen";
  bins AW_LEN[16]={[0:$]};
}

AWBURST_CP : coverpoint packet.awburst {
  option.comment = "awburst";
  bins AWBURST[]={[0:$]};
}

AWSIZE_CP : coverpoint packet.awsize {
  option.comment = "awsize";
  bins AWSIZE[]={[0:$]};
```

Figure 8.7. Coverpoint

8.5 Illegal bins

illegal_bins illegal_bin = {0};

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

8.6 Creation of the covergroup

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

Figure 8.8. Creation of covergroup

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

8.7 Sampling of the covergroup

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

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

  axi4_master_covergroup.sample(axi4_master_agent_cfg_h,t);

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

Figure 8.9. Sampling of the covergroup

8.8 Checking for the coverage

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

```
Log file path: axi4_blocking_write_read_test/axi4_blocking_write_read_test.log
```

Figure 8.10. Log file

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

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

```
Coverage report: firefox axi4_blocking_write_read_test/html_cov_report/index.html &
```

Figure 8.11. Coverage report

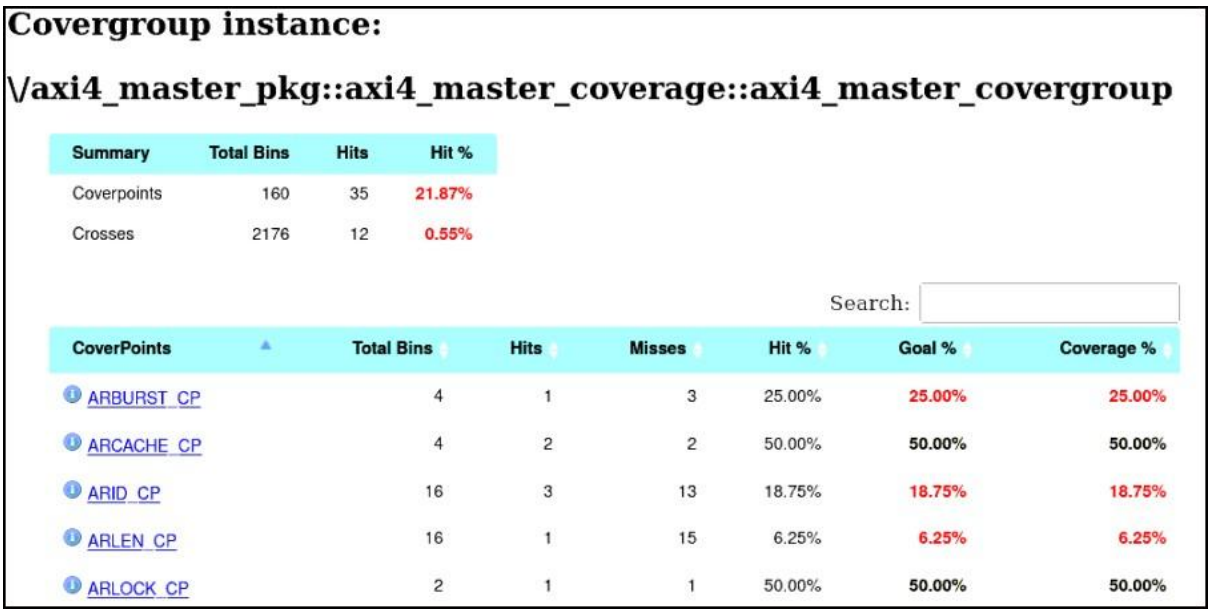
Then new html window will open

Coverage Summary by Type:

Total Coverage:					19.77%	**42.48%**
Coverage Type	**Bins**	**Hits**	**Misses**	**Weight**	**% Hit**	**Coverage**
Covergroups	4672	88	4584	1	1.88%	**20.93%**
Statements	5287	1360	3927	1	25.72%	**25.72%**
Branches	3567	395	3172	1	11.07%	**11.07%**
FEC Conditions	20	20	0	1	100.00%	**100.00%**
Toggles	2336	1278	1058	1	54.70%	**54.70%**

Figure 8.12. 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.



Covergroup instance:

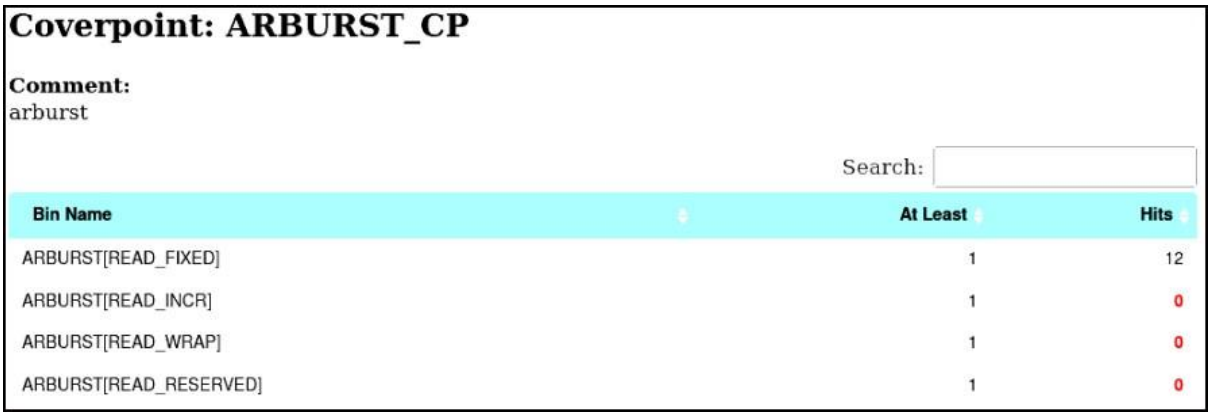
\/axi4_master_pkg::axi4_master_coverage::axi4_master_covergroup

Summary	Total Bins	Hits	Hit %
Coverpoints	160	35	21.87%
Crosses	2176	12	0.55%

Search:

CoverPoints	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
ARBURST_CP	4	1	3	25.00%	25.00%	25.00%
ARCACHE_CP	4	2	2	50.00%	50.00%	50.00%
ARID_CP	16	3	13	18.75%	18.75%	18.75%
ARLEN_CP	16	1	15	6.25%	6.25%	6.25%
ARLOCK_CP	2	1	1	50.00%	50.00%	50.00%

Figure 8.13. All coverpoints present in the Covergroup



Coverpoint: ARBURST_CP

Comment:
arburst

Search:

Bin Name	At Least	Hits
ARBURST[READ_FIXED]	1	12
ARBURST[READ_INCR]	1	0
ARBURST[READ_WRAP]	1	0
ARBURST[READ_RESERVED]	1	0

Figure 8.14. Individual Coverpoint Hit

Chapter 9

Test Cases

9.1 Test Flow

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

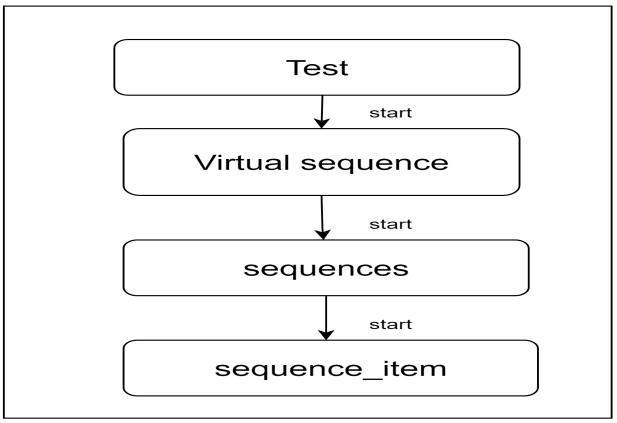


Fig 9.1 Test flow

9.2 AXI4 Test Cases Flow Chart

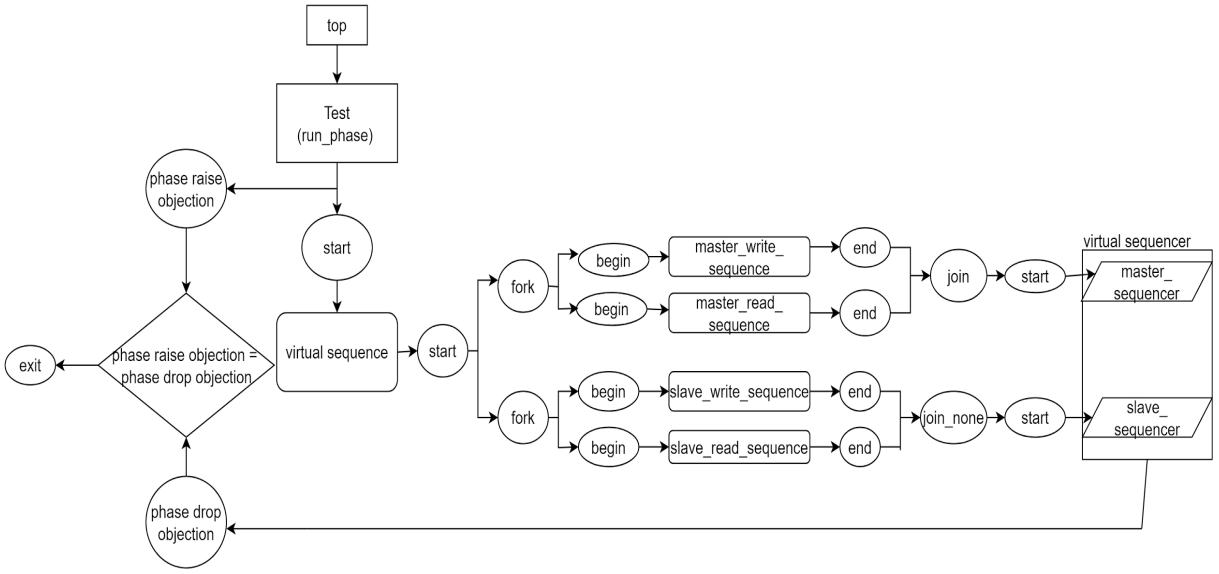


Fig 9.2 AXI4 test cases flow chart

9.3 Transaction

Two types of transactions are there

- ➢ Master_tx
- ➢ Slave_tx

9.3.1 Master_tx

- ➢ Master_tx class is extended from the uvm_sequence_item holds the data items

  required to drive stimulus to dut

- ➢ Declared all the variables(of all the 5 channels, like write address, write data, write response, read address, read data)

  Write address channel

- ➢ Constraint declared for awburst to get burst type between fixed, incr and wrap types.
- ➢ Constraint declared for awlen to restrict the transfer size.

- Constraint declared for awlen to get multiples of 2 in wrap type burst
- Constraint declared for awlock to get locked transfers
- Constraint declared for awburst to select the type of burst
- Constraint declared for awsize to get the size of transfer

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

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

constraint awlock_c4 { soft awlock == WRITE_NORMAL_ACCESS;
                     }

constraint awburst_c5 { soft awburst == WRITE_INCR;
                      }

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

Fig 9.3 Constraint for write address

Write data channel

- Constraint declared for wdata to restrict data based on awlen

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

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

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

Fig 9.4 Constraint for write data

Read address channels

➢ Constraint declared for arburst for restricting burst type between incr, wrap and fixed

➢ Constraint declared for arlen to restrict the transfer size.

➢ Constraint declared for arlen to get multiples of 2 in wrap type burst

➢ Constraint declared for arlock to get locked transfers

➢ Constraint declared for arburst to select the type of burst

➢ Constraint declared for arsize to get the size of transfer

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

Fig 9.5 Constraint for read address

Memory constraint

➢ Adding constraint for selecting the endianness

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

Fig 9.6 Constraint for memory

9.3.2 Slave_tx

- Slave_tx class is extended from the uvm_sequence_item holds the data items required to drive stimulus to dut
- Declared all the variables(of all the 5 channels, like write address, write data, write response, read address, read data)
- Constraint declared for ardata to restrict the data based on the arlen
- Constraint declared for the bresp to select the type of write response
- Constraint declared for the rresp to select the type of write response
- Constraint to randomise the wait state in between 0 to 3

```
constraint rdata_c1 { rdata.size() == arlen+1;
                      rdata.size() != 0;
                    }
constraint bresp_c1 {soft bresp == WRITE_OKAY;
                    }
constraint rresp_c1 {soft rresp == READ_OKAY;
                    }
constraint wait_states_c1 {soft no_of_wait_states inside {[0:3]};}
```

Fig 9.7 Constraints for read data response and wait states

Table 9.1 Describing constraint in master and slave transactions

Constraint	**Description**

awburst_c1	Constraint declared for awburst to get burst type between fixed, incr and wrap types.
awlength_c2	Constraint declared for awlen to restrict the transfer size.
awlength_c3	Constraint declared for awlen to get multiples of 2 in wrap type burst
awlock_c4	Constraint declared for awlock to get locked transfers
awburst_c5	Constraint declared for awburst to select the type of burst
awsize_c6	Constraint declared for awsize to get the size of transfer
wdata_c1	Constraint declared for wdata to restrict data based on awlen
no_of_wait_sta tes_c3	sponse
arburst_c1	Constraint declared for arburst for restricting burst type between incr, wrap and fixed
arlength_c2	Constraint declared for arlen to restrict the transfer size.
arlength_c3	Constraint declared for arlen to get multiples of 2 in wrap type burst
arlock_c4	Constraint declared for arlock to get locked transfers
arburst_c5	Constraint declared for arburst to select the type of burst
endian_c1	Adding constraint for selecting the endianness
rdata_c1	Constraint declared for ardata to restrict the data based on the arlen
bresp_c1	Constraint declared for the bresp to select the type of write response
rresp_c1	Constraint declared for the rresp to select the type of write response
wait_state_c1	Constraint to randomise the wait state in between 0 to 3

Master tx do_copy,do_compare and do_print methods

➢ Written functions for do_copy, do_compare, do_print methods, $casting is used to copy the data member values and compare the data member values and by using a printer , printing the data values.

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

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

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

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

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

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

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

Fig 9.8 do_copy method

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

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

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

  //WRITE DATA CHANNEL
```

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

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

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

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

Fig 9.9 do_compare method

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

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

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

Fig 9.10 do_print method

Slave tx do_copy,do_compare and do_print methods

- Written functions for do_copy, do_compare, do_print methods, $casting is used to copy the data member values and compare the data member values and by using a printer , printing the data values.

```
function void axi4_slave_tx::do_copy (uvm_object rhs);
  axi4_slave_tx axi_slave_tx_copy_obj;

  if(!$cast(axi_slave_tx_copy_obj,rhs )) begin
    `uvm_fatal("do_copy","cast of the rhs object failed")
  end

  super.do_copy(rhs);
  //WRITE ADDRESS CHANNEL
  awaddr  = axi_slave_tx_copy_obj.awaddr;
  awid    = axi_slave_tx_copy_obj.awid;
  awlen   = axi_slave_tx_copy_obj.awlen;
  awsize  = axi_slave_tx_copy_obj.awsize;
  awburst = axi_slave_tx_copy_obj.awburst;
  //awready = axi_slave_tx_copy_obj.awready;
  //awvalid = axi_slave_tx_copy_obj.awvalid;
  awlock  = axi_slave_tx_copy_obj.awlock;
  awcache = axi_slave_tx_copy_obj.awcache;
  awqos   = axi_slave_tx_copy_obj.awqos;
  awprot  = axi_slave_tx_copy_obj.awprot;

  //WRITE DATA CHANNEL
  wdata   = axi_slave_tx_copy_obj.wdata;
  wstrb   = axi_slave_tx_copy_obj.wstrb;
  //WRITE RESPONSE CHANNEL
  bid     = axi_slave_tx_copy_obj.bid;
  bresp   = axi_slave_tx_copy_obj.bresp;
```

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

  //READ DATA CHANNEL
  rid   = axi_slave_tx_copy_obj.rid;
  rdata = axi_slave_tx_copy_obj.rdata;
  rresp = axi_slave_tx_copy_obj.rresp;
  //rready  = axi_slave_tx_copy_obj.rready;
  //rvalid  = axi_slave_tx_copy_obj.rvalid;
endfunction : do_copy
```

Fig 9.11: slave_tx do_copy method

```
function bit axi4_slave_tx::do_compare (uvm_object rhs, uvm_comparer comparer);
  axi4_slave_tx axi_slave_tx_compare_obj;

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

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

  //WRITE DATA CHANNEL
  wdata == axi_slave_tx_compare_obj.wdata  &&
  wstrb == axi_slave_tx_compare_obj.wstrb  &&

  //WRITE RESPONSE CHANNEL
  bid   == axi_slave_tx_compare_obj.bid    &&
  bresp == axi_slave_tx_compare_obj.bresp  &&
```

```
  //READ ADDRESS CHANNEL
  araddr  == axi_slave_tx_compare_obj.araddr &&
  arid    == axi_slave_tx_compare_obj.arid   &&
  arlen   == axi_slave_tx_compare_obj.arlen  &&
  arsize  == axi_slave_tx_compare_obj.arsize &&
  arburst == axi_slave_tx_compare_obj.arburst &&
  //arready == axi_slave_tx_compare_obj.arready &&
  //arvalid == axi_slave_tx_compare_obj.arvalid &&
  arlock  == axi_slave_tx_compare_obj.arlock &&
  arcache == axi_slave_tx_compare_obj.arcache &&
  arqos   == axi_slave_tx_compare_obj.arqos  &&
  arprot  == axi_slave_tx_compare_obj.arprot &&

  //READ DATA CHANNEL
  rid   == axi_slave_tx_compare_obj.rid  &&
  rdata == axi_slave_tx_compare_obj.rdata &&
  rresp == axi_slave_tx_compare_obj.rresp;
  //rready == axi_slave_tx_compare.obj.rready &&
  //rvalid == axi_slave_tx_compare.obj.rvalid ;
endfunction : do_compare
```

Fig 9.12 slave_tx do_compare method

```
function void axi4_slave_tx::do_print(uvm_printer printer);
  //super.do_print(printer);
  if(tx_type == WRITE)begin
    //`uvm_info("-----------------------------------------WRITE_ADDRESS_CHANNEL","-------------
    printer.print_string("awid",awid.name());
    printer.print_field("awaddr",awaddr,$bits(awaddr),UVM_HEX);
    printer.print_field("awlen",awlen,$bits(awlen),UVM_DEC);
    printer.print_string("awsize",awsize.name());
    printer.print_string("awburst",awburst.name());
    printer.print_string("awlock",awlock.name());
    printer.print_string("awcache",awcache.name());
    printer.print_string("awprot",awprot.name());
    printer.print_field("awqos",awqos,$bits(awqos),UVM_HEX);
    //`uvm_info("-----------------------------------------WRITE_DATA_CHANNEL","-----------------
    foreach(wdata[i])begin
      printer.print_field($sformatf("wdata[%0d]",i),wdata[i],$bits(wdata[i]),UVM_HEX);
    end
    foreach(wstrb[i])begin
      printer.print_field($sformatf("wstrb[%0d]",i),wstrb[i],$bits(wstrb[i]),UVM_HEX);
    end
    printer.print_field("wlast",wlast,$bits(wlast),UVM_DEC);
    //`uvm_info("-----------------------------------------WRITE_RESPONSE_CHANNEL","-------------
    printer.print_string("bid",bid.name());
    printer.print_string("bresp",bresp.name());
  end
```

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

    printer.print_field("no_of_wait_states",no_of_wait_states,$bits(no_of_wait_states),UVM_HEX);
    printer.print_string("TRNASFER_TYPE",transfer_type.name());
  end
endfunction : do_print
```

Fig 9.13 slave_tx do_print method

9.4 Sequences

A UVM Sequence is an object that contains a behaviour 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 9.2. Sequence methods

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

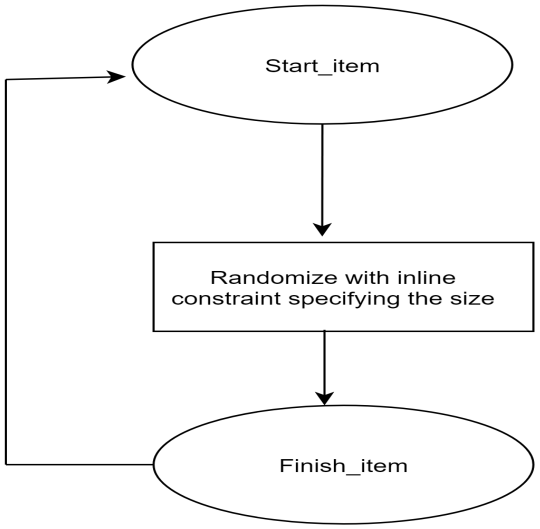


Fig 9.14 Flow chart for sequence methods

Table 9.3. Describing master sequences for Blocking and Non Blocking

Sections	Master sequences	Description
base_seq	axi4_master_base_seq	Base class is extended from uvm_ sequence and parameterized with transaction (axi4_master_ tx)
write	axi4_master_write_seq	Extended from master_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the awsize, awburst, transfer type, tx_type
read	axi4_master_read_seq	Extended from master_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the arsize, arburst, tx_type
Blocking base_seq	axi4_master_bk_base_seq	Base class is extended from uvm_ sequence and parameterized with transaction (axi4_master_ tx)and in task called sequence body method giving the req with transfer_types(BLOCKING_WRITE,BLOCKING_READ)
Blocking write	axi4_master_bk_write_seq	Extended from master_bk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the awsize, awburst, transfer type, tx_type(transfer_type=BLOCKING_WRITE)
Blocking read	axi4_master_bk_read_seq	Extended from master_bk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the arsize, arburst,tx_type,transfer_type.(transfer_type=BLOCKING_READ)
Blocking write data transfer	axi4_master_bk_write_8b_transfer_seq	Extended from master_bk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the awsize, awburst, transfer type, tx_type(awise is WRITE_1_BYTE for 8 bits)
	axi4_master_bk_write_16b_transfer_seq	Extended from master_bk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the awsize, awburst, transfer type, tx_type(awise is WRITE_2_BYTE for 16 bits)

	axi4_master_bk_write_32b_transfer_seq	Extended from master_bk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the awsize, awburst, transfer type, tx_type(awise is WRITE_4_BYTE for 32 bits)
	axi4_master_bk_write_64b_transfer_seq	Extended from master_bk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the awsize, awburst, transfer type, tx_type(awise is WRITE_8_BYTE for 64 bits)
Blocking read data transfer	axi4_master_bk_read_8b_transfer_seq	Extended from master_bk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the arsize, arburst, tx_type,transfer_type.(arsize is READ_1_BYTE for 8 bits)
	axi4_master_bk_read_16b_transfer_seq	Extended from master_bk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the arsize, arburst, tx_type,transfer_type.(arsize is READ_2_BYTE for 8 bits)
	axi4_master_bk_read_32b_transfer_seq	Extended from master_bk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the arsize, arburst, tx_type,transfer_type.(arsize is READ_4_BYTE for 8 bits)
	axi4_master_bk_read_64b_transfer_seq	Extended from master_bk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the arsize, arburst, tx_type,transfer_type.(arsize is READ_8_BYTE for 8 bits)
Blocking write burst transfers	axi4_master_bk_write_incr_burst_seq	Extended from master_bk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the awsize, awburst, transfer type, tx_type(awburst is WRITE_INCR)
	axi4_master_bk_write_wrap_burst_seq	Extended from master_bk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the awsize, awburst, transfer type, tx_type(awburst is WRITE_WRAP)

Blocking read burst transfers	axi4_master_bk_read_incr_burst_seq	Extended from master_bk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the arsize, arburst, tx_type,transfer_type.(aburst is READ_INCR)
	axi4_master_bk_read_wrap_burst_seq	Extended from master_bk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the arsize, arburst, tx_type,transfer_type.(aburst is READ_WRAP)
Blocking write responses	axi4_master_bk_write_okay_resp_seq	Extended from master_bk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the awsize, awburst, tx_type,transfer_type.
	axi4_master_bk_write_exokay_resp_seq	Extended from master_bk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the awsize, awburst, tx_type,transfer_type
Blocking read response	axi4_master_bk_read_okay_resp_seq	Extended from master_bk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the arsize, arburst, tx_type,transfer_type
	axi4_master_bk_read_exokay_resp_seq	Extended from master_bk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the arsize, arburst, tx_type,transfer_type
non_blocking base_seq	axi4_master_nbk_base_seq	Base class is extended from uvm_ sequence and parameterized with transaction (axi4_master_ tx)and in task called sequence body method giving the req with transfer_types(NON_BLOCKING_WRITE,NON_ BLOCKING_READ)
Non_blocking write	axi4_master_nbk_write_seq	Extended from master_nbk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the awsize, awburst, transfer type, tx_type(transfer_type=NON_BLOCKING_WRIT E)

Non_blocking read	axi4_master_nbk_read_seq	Extended from master_nbk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the arsize, arburst, transfer type, tx_type(transfer_type=NON_BLOCKING_READ)
non_blocking write data transfer	axi4_master_nbk_write_8b_transfer_seq	Extended from master_nbk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the awsize, awburst, tx_type,transfer_type.(awsize is WRITE_1_BYTE for 8 bits)
	axi4_master_nbk_write_32b_transfer_seq	Extended from master_nbk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the awsize, awburst, tx_type,transfer_type.(awsize is WRITE_4_BYTE for 8 bits)
	axi4_master_nbk_write_64b_transfer_seq	Extended from master_nbk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the awsize, awburst, tx_type,transfer_type.(awsize is WRITE_8_BYTE for 8 bits)
	axi4_master_nbk_write_16b_transfer_seq	Extended from master_nbk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the awsize, awburst, tx_type,transfer_type.(awsize is WRITE_2_BYTE for 8 bits)
non_blocking read data transfer	axi4_master_nbk_read_8b_transfer_seq	Extended from master_nbk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the arsize, arburst, tx_type,transfer_type.(arsize is READ_1_BYTE for 8 bits)
	axi4_master_nbk_read_16b_transfer_seq	Extended from master_nbk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the arsize, arburst, tx_type,transfer_type.(arsize is READ_2_BYTE for 8 bits)

	axi4_master_nbk_read_32b_transfer_seq	Extended from master_nbk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the arsize, arburst, tx_type,transfer_type.(arsize is READ_4_BYTE for 8 bits)
	axi4_master_nbk_read_64b_transfer_seq	Extended from master_nbk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the arsize, arburst, tx_type,transfer_type.(arsize is READ_8_BYTE for 8 bits)
non_blocking write burst transfers	axi4_master_nbk_write_incr_burst_seq	Extended from master_nbk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the awsize, awburst, transfer type, tx_type(awburst is WRITE_INCR)
	axi4_master_nbk_write_wrap_burst_seq	Extended from master_nbk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the awsize, awburst, transfer type, tx_type(awburst is WRITE_WRAP)
non_blocking read burst transfers	axi4_master_nbk_read_incr_burst_seq	Extended from master_nbk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the arsize, arburst, transfer type, tx_type(awburst is READ_INCR)
	axi4_master_nbk_read_wrap_burst_seq	Extended from master_nbk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the arsize, arburst, transfer type, tx_type(awburst is READ_WRAP)
non_blocking write responses	axi4_master_nbk_write_okay_resp_seq	Extended from master_nbk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the awsize, awburst, tx_type,transfer_type
	axi4_master_nbk_write_exokay_resp_seq	Extended from master_nbk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the awsize, awburst, tx_type,transfer_type

non_blocking read responses	axi4_master_nbk_read_okay_resp_seq	Extended from master_nbk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the arsize, arburst, tx_type,transfer_type
	axi4_master_nbk_read_exokay_resp_seq	Extended from master_nbk_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the arsize, arburst, tx_type,transfer_type

Table 9.4 : Describing slave sequences for Blocking and Non Blocking

Sections	Slave sequences	Description
base_seq	axi4_slave_base_seq	Base class is extended from uvm_ sequence and parameterized with transaction (axi4_slave_ tx)
write	axi4_slave_write_seq	Extended from slave_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the bresp.
read	axi4_slave_read_seq	Extended from slave_base_seq. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item using inline constraint and randomising the req with the rresp
Blocking base_seq	axi4_slave_bk_base_seq	Base class is extended from uvm_ sequence and parameterized with transaction (axi4_slave_ tx)and in task called sequence body method giving the req with the transfer_types(BLOCKING_WRITE,BLOCKING_READ)
Blocking write	axi4_slave_bk_write_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as BLOCKING_WRITE. In between start_ item and finish_ item randomising the req.
Blocking read	axi4_slave_bk_read_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as BLOCKING_READ. In between start_ item and finish_ item randomising the req.
Blocking write data transfer	axi4_slave_bk_write_8b_transfer_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as BLOCKING_WRITE. In between start_ item and finish_ item randomising the req.

	axi4_slave_bk_write_16b_transfer_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as BLOCKING_WRITE. In between start_ item and finish_ item randomising the req.
	axi4_slave_bk_write_32b_transfer_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as BLOCKING_WRITE. In between start_ item and finish_ item randomising the req.
	axi4_slave_bk_write_64b_transfer_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as BLOCKING_WRITE. In between start_ item and finish_ item randomising the req.
Blocking read data transfer	axi4_slave_bk_read_8b_transfer_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as BLOCKING_READ. In between start_ item and finish_ item using inline constraint and randomising the req with the arsize, arburst, transfer type, tx_type (arsize is READ_1_BYTE)
	axi4_slave_bk_read_16b_transfer_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as BLOCKING_READ. In between start_ item and finish_ item using inline constraint and randomising the req with the arsize, arburst, transfer type, tx_type (arsize is READ_2_BYTE)
	axi4_slave_bk_read_32b_transfer_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as BLOCKING_READ. In between start_ item and finish_ item using inline constraint and randomising the req with the arsize, arburst, transfer type, tx_type (arsize is READ_4_BYTE)
	axi4_slave_bk_read_64b_transfer_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as BLOCKING_READ. In between start_ item and finish_ item using inline constraint and randomising the req with the arsize, arburst, transfer type, tx_type (arsize is READ_8_BYTE)
Blocking write burst transfers	axi4_slave_bk_write_incr_burst_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as BLOCKING_WRITE. In between start_ item and finish_ item randomising the req.

	axi4_slave_bk_write_wrap_burst_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as BLOCKING_WRITE. In between start_ item and finish_ item randomising the req.
Blocking read burst transfers	axi4_slave_bk_read_incr_burst_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as BLOCKING_READ. In between start_ item and finish_ item randomising the req.
	axi4_slave_bk_read_wrap_burst_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as BLOCKING_READ. In between start_ item and finish_ item randomising the req.
Blocking write responses	axi4_slave_bk_write_okay_resp_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as BLOCKING_WRITE. In between start_ item and finish_ item randomising the req.
	axi4_slave_bk_write_exokay_resp_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as BLOCKING_WRITE. In between start_ item and finish_ item randomising the req.
Blocking read response	axi4_slave_bk_read_okay_resp_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as BLOCKING_READ. In between start_ item and finish_ item randomising the req.
	axi4_slave_bk_read_exokay_resp_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as BLOCKING_READ. In between start_ item and finish_ item randomising the req.
non_blocking base_seq	axi4_slave_nbk_base_seq	Base class is extended from uvm_ sequence and parameterized with transaction (axi4_slave_ tx)
Non_blocking write	axi4_slave_nbk_write_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as NON_BLOCKING_WRITE. In between start_ item and finish_ item randomising the req.
Non_blocking read	axi4_slave_nbk_read_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as NON_BLOCKING_READ. In between start_ item and finish_ item randomising the req.

non_blocking write data transfer	axi4_slave_nbk_write_8b_transfer_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as NON_BLOCKING_WRITE. In between start_ item and finish_ item randomising the req.
	axi4_slave_nbk_write_32b_transfer_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as NON_BLOCKING_WRITE. In between start_ item and finish_ item randomising the req.
	axi4_slave_nbk_write_64b_transfer_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as NON_BLOCKING_WRITE. In between start_ item and finish_ item randomising the req.
	axi4_slave_nbk_write_16b_transfer_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as NON_BLOCKING_WRITE. In between start_ item and finish_ item randomising the req.
non_blocking read data transfer	axi4_slave_nbk_read_8b_transfer_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as BLOCKING_READ. In between start_ item and finish_ item using inline constraint and randomising the req with the arsize, arburst, transfer type, tx_type (arsize is READ_1_BYTE)
	axi4_slave_nbk_read_16b_transfer_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as BLOCKING_READ. In between start_ item and finish_ item using inline constraint and randomising the req with the arsize, arburst, transfer type, tx_type (arsize is READ_2_BYTE)
	axi4_slave_nbk_read_32b_transfer_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as BLOCKING_READ. In between start_ item and finish_ item using inline constraint and randomising the req with the arsize, arburst, transfer type, tx_type (arsize is READ_4_BYTE)
	axi4_slave_nbk_read_64b_transfer_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as BLOCKING_READ. In between start_ item and finish_ item using inline constraint and randomising the req with the arsize, arburst, transfer type, tx_type (arsize is READ_8_BYTE)

non_blocking write burst transfers	axi4_slave_nbk_write_incr_burst_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as NON_BLOCKING_WRITE. In between start_ item and finish_ item randomising the req.
	axi4_slave_nbk_write_wrap_burst_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as NON_BLOCKING_WRITE. In between start_ item and finish_ item randomising the req.
non_blocking read burst transfers	axi4_slave_nbk_read_incr_burst_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as NON_BLOCKING_READ. In between start_ item and finish_ item randomising the req.
	axi4_slave_nbk_read_wrap_burst_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as NON_BLOCKING_READ. In between start_ item and finish_ item randomising the req.
non_blocking write responses	axi4_slave_nbk_write_okay_resp_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as NON_BLOCKING_WRITE. In between start_ item and finish_ item randomising the req.
	axi4_slave_nbk_write_exokay_resp_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as NON_BLOCKING_WRITE. In between start_ item and finish_ item randomising the req.
non_blocking read responses	axi4_slave_nbk_read_okay_resp_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as NON_BLOCKING_READ. In between start_ item and finish_ item randomising the req.
	axi4_slave_nbk_read_exokay_resp_seq	Extended from slave_bk_base_seq. Based on a request from the driver, the task will drive the transactions and give the req with transfer_type as NON_BLOCKING_READ. In between start_ item and finish_ item randomising the req.

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

```
task axi4_master_bk_base_seq::body();
//  if(!$cast(p_sequencer,m_sequencer))begin
  //super.body();
  req = axi4_master_tx::type_id::create("req");
  req.transfer_type=BLOCKING_WRITE;
  req.transfer_type=BLOCKING_READ;

//    `uvm_error(get_full_name(),"master_agent_config pointer cast failed")
//  end
endtask
```

Fig 9.15 Master blocking seq body method

```
task axi4_master_nbk_base_seq::body();
//  if(!$cast(p_sequencer,m_sequencer))begin
//    `uvm_error(get_full_name(),"master_agent_config pointer cas
  req = axi4_master_tx::type_id::create("req");
  req.transfer_type = NON_BLOCKING_WRITE ;
  req.transfer_type = NON_BLOCKING_READ ;
//  end
endtask
```

Fig 9.16 Master non_blocking sequence

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

```
task axi4_slave_bk_base_seq::body();
//  end
  req = axi4_slave_tx::type_id::create("req");
  req.transfer_type=BLOCKING_WRITE;
  req.transfer_type=BLOCKING_READ;
endtask
```

Fig 9.17 Slave blocking seq body method

```
task axi4_slave_nbk_base_seq::body();
  req = axi4_slave_tx::type_id::create("req");
endtask
```

Fig 9.18 Slave non blocking seq body method

9.5 Virtual sequences

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

Virtual sequence base class

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

```
class axi4_virtual_base_seq extends uvm_sequence;
  `uvm_object_utils(axi4_virtual_base_seq)

   //p_sequencer macro declaration
   `uvm_declare_p_sequencer(axi4_virtual_sequencer)

   axi4_env_config env_cfg_h;

  //-------------------------------------------------------------------------------
  // Externally defined tasks and functions
  //-------------------------------------------------------------------------------
  extern function new(string name="axi4_virtual_base_seq");
  extern task body();

endclass:axi4_virtual_base_seq
```

Fig 9.19 Virtual base sequence

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

```
task axi4_virtual_base_seq::body();

   if(!uvm_config_db#(axi4_env_config) ::get(null,get_full_name(),"axi4_env_config",env_cfg_h))
    `uvm_fatal("CONFIG","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
  // MSHA:axi4_slave_write_seqr_h = p_sequencer.axi4_slave_write_seqr_h;
  // MSHA:axi4_slave_read_seqr_h = p_sequencer.axi4_slave_read_seqr_h;
endtask:body
```

Fig 9.20 Virtual base sequence body

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

```
task axi4_virtual_bk_8b_data_read_seq::body();
  axi4_master_bk_read_8b_transfer_seq_h = axi4_master_bk_read_8b_transfer_seq::type_id::create
                                          ("axi4_master_bk_read_8b_transfer_seq_h");
  axi4_slave_bk_read_8b_transfer_seq_h =
  axi4_slave_bk_read_8b_transfer_seq::type_id::create("axi4_slave_bk_read_8b_transfer_seq_h");

  `uvm_info(get_type_name(), $sformatf("DEBUG_MSHA :: Insdie axi4_virtual_bk_read_8b_transfer_seq"
                                       ), UVM_NONE);

  fork
    begin : T2_SL_RD
      forever begin
        axi4_slave_bk_read_8b_transfer_seq_h.start(p_sequencer.axi4_slave_read_seqr_h);
      end
    end
  join_none


  fork
    begin: T2_READ
      repeat(3) begin
      axi4_master_bk_read_8b_transfer_seq_h.start(p_sequencer.axi4_master_read_seqr_h);
      end
    end
  join
 endtask : body
```

Fig 9.21 axi4_virtual_bk_8b_read_data_seq body

```
task axi4_virtual_nbk_8b_data_read_seq::body();
  axi4_master_nbk_read_8b_transfer_seq_h = axi4_master_nbk_read_8b_transfer_seq::type_id::create
                                          ("axi4_master_nbk_read_8b_transfer_seq_h");
  axi4_slave_nbk_read_8b_transfer_seq_h =
  axi4_slave_nbk_read_8b_transfer_seq::type_id::create("axi4_slave_nbk_read_8b_transfer_seq_h");

  `uvm_info(get_type_name(), $sformatf("DEBUG_MSHA :: Insdie axi4_virtual_nbk_read_8b_transfer_seq"
                                      ), UVM_NONE);

  fork
    begin : T2_SL_RD
      forever begin
        axi4_slave_nbk_read_8b_transfer_seq_h.start(p_sequencer.axi4_slave_read_seqr_h);
      end
    end
  join_none

  fork
    begin: T2_READ
      repeat(3) begin
      axi4_master_nbk_read_8b_transfer_seq_h.start(p_sequencer.axi4_master_read_seqr_h);
      end
    end
  join
 endtask : body
```

Fig 9.22 axi4_virtual_nbk_8b_read_data_seq body method

Table 9.5: . Describing virtual sequences for Blocking and Non Blocking

Sections	Virtual sequences	Description
Burst type	axi4_virtual_bk_incr_burst_write_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(bk_write_incr_burst sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none.
	axi4_virtual_bk_incr_burst_read_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(bk_read_incr_burst sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_bk_incr_burst_write_read_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(bk_write_incr_burst, read_incr_burst sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_bk_wrap_burst_write_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(bk_write_wrap_burst sequences) with p_sequencer, master can be repeatedfor multiple

		times, both master and slave can be started within fork join_none
	axi4_virtual_bk_wrap_burst_read_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(bk_read_incr_burst sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_bk_wrap_burst_write_read_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(bk_write_incr_burst, bk_read_incr_burst sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
Responses	axi4_virtual_bk_okay_resp_write_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(bk_write_okay_resp sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_bk_okay_resp_read_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(bk_read_okay_resp sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_bk_okay_resp_write_read_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(bk_write_okay_resp, bk_read_okay_resp sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_bk_exokay_resp_write_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(bk_write_exokay_resp sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_bk_exokay_resp_read_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(bk_read_exokay_resp sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none

	axi4_virtual_bk_exokay_resp_write_read_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(bk_write_exokay_resp, bk_read_exokay_resp sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
Data transfer	axi4_virtual_bk_8b_write_data_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(bk_write_8b_transfer sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_bk_8b_read_data_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(bk_read_8b_transfer sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_bk_8b_write_read_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(bk_write_8b_transfer, bk_read_8b_transfer sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_bk_16b_write_data_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(bk_write_16b_transfer sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_bk_16b_read_data_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(bk_read_16b_transfer sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_bk_16b_write_read_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(bk_write_16b_transfer, bk_read_16b_transfer sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_bk_32b_write_data_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave

		and master sequences(bk_write_32b_transfer sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_bk_32b_read_data_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(bk_read_32b_transfer sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_bk_32b_write_read_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(bk_write_32b_transfer, bk_read_32b_transfer sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_bk_64b_write_data_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(bk_write_64b_transfer sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_bk_64b_read_data_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(bk_read_64b_transfer sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_bk_64b_write_read_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(bk_write_64b_transfer, bk_read_64b_transfer sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
Burst type	axi4_virtual_nbk_incr_burst_write_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(nbk_write_incr_burst sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none.
	axi4_virtual_nbk_incr_burst_read_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(nbk_read_incr_burst sequences) with p_sequencer, master can be repeatedfor multiple

		times, both master and slave can be started within fork join_none.
	axi4_virtual_nbk_incr_burst_write_read_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(nbk_write_incr_burst, nbk_read_incr_burst sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none.
	axi4_virtual_nbk_wrap_burst_write_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(nbk_write_incr_burst sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_nbk_incr_burst_read_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(nbk_read_incr_burst sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_nbk_incr_burst_write_read_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(nbk_write_wrap_burst, nbk_read_wrap_burst sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none.
Responses	axi4_virtual_nbk_okay_resp_write_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(nbk_write_okay_resp sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_nbk_okay_resp_read_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(nbk_read_okay_resp sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_nbk_okay_resp_write_read_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(nbk_write_okay_resp, nbk_read_okay_resp sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none

	axi4_virtual_nbk_exokay_resp_write_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(nbk_write_exokay_resp sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_nbk_exokay_resp_read_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(nbk_read_exokay_resp sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_nbk_exokay_resp_write_read_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(nbk_write_exokay_resp, nbk_read_exokay_resp sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
Data transfer	axi4_virtual_nbk_8b_write_data_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(nbk_write_8b_transfer sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_nbk_8b_read_data_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(nbk_read_8b_transfer sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_nbk_8b_write_read_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(nbk_write_8b_transfer, nbk_read_8b_transfer sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_nbk_16b_write_data_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(nbk_write_16b_transfer sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_nbk_16b_read_data_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave

		and master sequences(nbk_read_16b_transfer sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_nbk_16b_write_read_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(nbk_write_16b_transfer, nbk_read_16b_transfer sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_nbk_32b_write_data_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(nbk_write_32b_transfer sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_nbk_32b_read_data_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(nbk_read_32b_transfer sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_nbk_32b_write_read_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(nbk_write_32b_transfer, nbk_read_32b_transfer sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_nbk_64b_write_data_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(nbk_write_64b_transfer sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_nbk_64b_read_data_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(nbk_read_64b_transfer sequences) with p_sequencer, master can be repeatedfor multiple times, both master and slave can be started within fork join_none
	axi4_virtual_nbk_64b_write_read_seq	Extending from virtual base class. Declaring handles of sequences and inside body method constructing handles of sequence.In the sequence body method start the slave and master sequences(nbk_write_64b_transfer, nbk_read_64b_transfer sequences) with p_sequencer,

		master can be repeatedfor multiple times, both master and slave can be started within fork join_none

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 axi4_base_test extends uvm_test;

  `uvm_component_utils(axi4_base_test)

  // Declaring environment config handle
  axi4_env_config axi4_env_cfg_h;

  // Handle for environment
  axi4_env axi4_env_h;

  //-------------------------------------------------------
  // Externally defined Tasks and Functions
  //-------------------------------------------------------
  extern function new(string name = "axi4_base_test", uvm_component parent = null);
  extern virtual function void build_phase(uvm_phase phase);
  extern virtual function void setup_axi4_env_cfg();
  extern virtual function void setup_axi4_master_agent_cfg();
  extern virtual function void setup_axi4_slave_agent_cfg();
  extern virtual function void end_of_elaboration_phase(uvm_phase phase);
  extern virtual task run_phase(uvm_phase phase);

endclass : axi4_base_test
```

Fig 9.23 Base test

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

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

```
function void axi4_base_test:: setup_axi4_env_cfg();
  axi4_env_cfg_h = axi4_env_config::type_id::create("axi4_env_cfg_h");
  // axi4_env_cfg_h = axi4_env_config::type_id::create("axi4_env_cfg_h");

  axi4_env_cfg_h.has_scoreboard = 1;
  axi4_env_cfg_h.has_virtual_seqr = 1;
  axi4_env_cfg_h.no_of_masters = NO_OF_MASTERS;
  axi4_env_cfg_h.no_of_slaves = NO_OF_SLAVES;

  // Setup the axi4_master agent cfg
  setup_axi4_master_agent_cfg();

  setup_axi4_slave_agent_cfg();
  // set method for axi4_env_cfg
  uvm_config_db #(axi4_env_config)::set(this,"*","axi4_env_config",axi4_env_cfg_h);
  `uvm_info(get_type_name(),$sformatf("\nAXI4_ENV_CONFIG\n%s",axi4_env_cfg_h.sprint()),UVM_LOW);
endfunction: setup_axi4_env_cfg
```

Fig 9.24 Setup env_cfg

D) In setup_master_agent_config function, master_agent_config class handle which is in env_config class has been constructed with the help of this handle all the required fields in master_agent_config class has been setup.

```
function void axi4_base_test::setup_axi4_master_agent_cfg();
  bit [63:0]local_min_address;
  bit [63:0]local_max_address;
  axi4_env_cfg_h.axi4_master_agent_cfg_h = new[axi4_env_cfg_h.no_of_masters];
  foreach(axi4_env_cfg_h.axi4_master_agent_cfg_h[i])begin
    axi4_env_cfg_h.axi4_master_agent_cfg_h[i] =
    axi4_master_agent_config::type_id::create($sformatf("axi4_master_agent_cfg_h[%0d]",i));
    axi4_env_cfg_h.axi4_master_agent_cfg_h[i].is_active    = uvm_active_passive_enum'(UVM_ACTIVE);
    axi4_env_cfg_h.axi4_master_agent_cfg_h[i].has_coverage = 1;
    //uvm_config_db#(axi4_master_agent_config)::set(this,"*axi4_master_agent*","axi4_master_agent_c
    uvm_config_db#(axi4_master_agent_config)::set(this,"*env*",$sformatf
                  ("axi4_master_agent_config[%0d]",i),axi4_env_cfg_h.axi4_master_agent_cfg_h[i]);
  end

  for(int i =0; i<NO_OF_SLAVES; i++) begin
    if(i == 0) begin
      axi4_env_cfg_h.axi4_master_agent_cfg_h[i].master_min_addr_range(i,0);
      local_min_address = axi4_env_cfg_h.axi4_master_agent_cfg_h[i].master_min_addr_range_array[i];
      axi4_env_cfg_h.axi4_master_agent_cfg_h[i].master_max_addr_range(i,2**(SLAVE_MEMORY_SIZE)-1 );
      local_max_address = axi4_env_cfg_h.axi4_master_agent_cfg_h[i].master_max_addr_range_array[i];
    end
    else begin
      axi4_env_cfg_h.axi4_master_agent_cfg_h[i].master_min_addr_range
                                        (i,local_max_address + SLAVE_MEMORY_GAP);
      local_min_address = axi4_env_cfg_h.axi4_master_agent_cfg_h[i].master_min_addr_range_array[i];
      axi4_env_cfg_h.axi4_master_agent_cfg_h[i].master_max_addr_range
                             (i,local_max_address+ 2**(SLAVE_MEMORY_SIZE)-1 + SLAVE_MEMORY_GAP);
      local_max_address = axi4_env_cfg_h.axi4_master_agent_cfg_h[i].master_max_addr_range_array[i];
    end
   `uvm_info(get_type_name(),$sformatf("\nAXI4_MASTER_CONFIG[%0d]\n%s",
                                  i,axi4_env_cfg_h.axi4_master_agent_cfg_h[i].sprint()),UVM_LOW);
  end
endfunction: setup_axi4_master_agent_cfg
```

Fig 9.25 Master_agent_cfg setup

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

```
function void axi4_base_test::setup_axi4_slave_agent_cfg();
  axi4_env_cfg_h.axi4_slave_agent_cfg_h = new[axi4_env_cfg_h.no_of_slaves];
  foreach(axi4_env_cfg_h.axi4_slave_agent_cfg_h[i])begin
    axi4_env_cfg_h.axi4_slave_agent_cfg_h[i] =
    axi4_slave_agent_config::type_id::create($sformatf("axi4_slave_agent_cfg_h[%0d]",i));
    axi4_env_cfg_h.axi4_slave_agent_cfg_h[i].slave_id = i;
    //axi4_env_cfg_h.axi4_slave_agent_cfg_h[i].slave_selected = 0;
    axi4_env_cfg_h.axi4_slave_agent_cfg_h[i].min_address=axi4_env_cfg_h.axi4_master_agent_cfg_h[i].
                                                         master_min_addr_range_array[i];
    axi4_env_cfg_h.axi4_slave_agent_cfg_h[i].max_address=axi4_env_cfg_h.axi4_master_agent_cfg_h[i].
                                                         master_max_addr_range_array[i];
    if(SLAVE_AGENT_ACTIVE === 1) begin
    axi4_env_cfg_h.axi4_slave_agent_cfg_h[i].is_active = uvm_active_passive_enum'(UVM_ACTIVE);
    end
    else begin
    axi4_env_cfg_h.axi4_slave_agent_cfg_h[i].is_active = uvm_active_passive_enum'(UVM_PASSIVE);
    end
    axi4_env_cfg_h.axi4_slave_agent_cfg_h[i].has_coverage = 1;

    uvm_config_db #(axi4_slave_agent_config)::set(this,"*env*",$sformatf
                    ("axi4_slave_agent_config[%0d]",i), axi4_env_cfg_h.axi4_slave_agent_cfg_h[i]);
   `uvm_info(get_type_name(),$sformatf("\nAXI4_SLAVE_CONFIG[%0d]\n%s",i,
                                       axi4_env_cfg_h.axi4_slave_agent_cfg_h[i].sprint()),UVM_LOW);
  end
endfunction: setup_axi4_slave_agent_cfg
```

Fig 9.26 Slave_agent_cfg setup

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

```
class axi4_blocking_8b_data_read_test extends axi4_base_test;
  `uvm_component_utils(axi4_blocking_8b_data_read_test)

  //Variable : axi4_virtual_write_seq_h
  //Instatiation of axi4_virtual_write_seq
  axi4_virtual_bk_8b_data_read_seq axi4_virtual_bk_8b_data_read_seq_h;

  //-------------------------------------------------------
  // Externally defined Tasks and Functions
  //-------------------------------------------------------
  extern function new(string name = "axi4_blocking_8b_data_read_test", uvm_component parent = null)
  extern virtual task run_phase(uvm_phase phase);

endclass : axi4_blocking_8b_data_read_test
```

Fig 9.27 Example for 8bit read data test

```
task axi4_blocking_8b_data_read_test::run_phase(uvm_phase phase);

  axi4_virtual_bk_8b_data_read_seq_h=axi4_virtual_bk_8b_data_read_seq::type_id::create
                                          ("axi4_virtual_bk_8b_data_read_seq_h");
  `uvm_info(get_type_name(),$sformatf("axi4_blocking_8b_data_read_test"),UVM_LOW);
  phase.raise_objection(this);
  axi4_virtual_bk_8b_data_read_seq_h.start(axi4_env_h.axi4_virtual_seqr_h);
  phase.drop_objection(this);

endtask : run_phase
```

Fig 9.29 Run_phase of 8bit_test

Table 9.6 : Describing Test cases

Sections	Test Names	Description
Base test	axi4_base_test	Extending the base test from the uvm_test and creating the env_config, master_agent_cfg and slave_agent_cfg
	axi4_blocking_write_read_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_non_blocking_write_read_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
Burst type	axi4_blocking_incr_burst_write_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_blocking_incr_burst_read_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_blocking_incr_burst_write_read_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_blocking_wrap_burst_write_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_blocking_wrap_burst_read_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.

	axi4_blocking_wrap_burst_write_read_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
Responses	axi4_blocking_okay_response_write_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_blocking_okay_response_read_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_blocking_okay_response_write_read_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_blocking_exokay_response_write_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_blocking_exokay_response_read_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_blocking_exokay_response_write_read_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
Data transfer	axi4_blocking_8b_write_data_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_blocking_8b_read_data_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_blocking_8b_write_read_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_blocking_16b_write_data_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_blocking_16b_read_data_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_blocking_16b_write_read_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.

	axi4_blocking_32b_write_data_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_blocking_32b_read_data_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_blocking_32b_write_read_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_blocking_64b_write_data_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_blocking_64b_write_data_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_blocking_64b_read_data_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
Burst type	axi4_non_blocking_incr__burst_write_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_non_blocking_incr__burst_read_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_non_blocking_incr_burst_write_read_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_non_blocking_wrap_burst_write_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_non_blocking_wrap_burst_read_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_non_blocking_wrap_burst_write_read_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
Responses	axi4_non_blocking_okay_response_write_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.

	axi4_non_blocking_okay_response_read_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_non_blocking_okay_response_write_read_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_non_blocking_exokay_response_write_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_non_blocking_exokay_response_read_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_non_blocking_exokay_response_write_read_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
Data transfer	axi4_non_blocking_8b_write_data_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_non_blocking_8b_wread_data_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_non_blocking_8b_write_read_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_non_blocking_16b_write_data_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_non_blocking_16b_read_data_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_non_blocking_16b_write_read_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_non_blocking_32b_write_data_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_non_blocking_32b_read_data_test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.

	axi4_non_blocking_32b_write_read _test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_non_blocking_64b_write_data _test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_non_blocking_64b_read_data_ test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.
	axi4_non_blocking_64b_write_read _test	Extended from the axi4_base_test and created the virtual sequence handle and starting the sequences in between phase raise and drop objection.

9.7 Testlists

Regression list for axi4

Table 9.7 Regression list of Test cases

Test Names	**Description**
axi4_blocking_incr_burst_write_test	Checking for incr bust type write transac- tion
axi4_blocking_incr_burst_read_test	Checking for incr bust type read transaction
axi4_blocking_incr_burst_write_read_test	Checking for incr bust type write read transaction
axi4_blocking_wrap_burst_write_test	Checking for wrap bust type write transac- tion
axi4_blocking_wrap_burst_read_test	Checking for wrap bust type read transac- tion
axi4_blocking_wrap_burst_write_read_test	Checking for wrap bust type write read transaction
axi4_blocking_okay_response_write_test	Checking for okay response type write transaction
axi4_blocking_okay_response_read_test	Checking for okay response type read transaction
axi4_blocking_okay_response_write_read_test	Checking for okay response type write read transaction

axi4_blocking_8b_write_data_test	Checking for 8b write data transaction
axi4_blocking_8b_data_read_test	Checking for 8b read data transaction
axi4_blocking_8b_write_read_test	Checking for 8b write read transaction
axi4_blocking_16b_write_data_test	Checking for 16b write data transaction
axi4_blocking_16b_data_read_test	Checking for 16b read data transaction
axi4_blocking_16b_write_read_test	Checking for 16b write read transaction
axi4_blocking_32b_write _data_test	Checking for 32b write data transaction
axi4_blocking_32b_data_read_test	Checking for 32b read data transaction
axi4_blocking_32b_write_read_test	Checking for 32b write read transaction
axi4_blocking_64b_write_data_test	Checking for 64b write data transaction
axi4_blocking_64b_data_read_test	Checking for 64b read data transaction
axi4_blocking_64b_write_read_test	Checking for 64b write read transaction
axi4_non_blocking_incr_burst_write_test	Checking for incr bust type write transac- tion
axi4_non_blocking_incr_burst_read_test	Checking for incr bust type read transaction
axi4_non_blocking_incr_burst_write_read_test	Checking for incr bust type write read transaction
axi4_non_blocking_wrap_burst_write_test	Checking for wrap bust type write transac- tion
axi4_non_blocking_wrap_burst_read_test	Checking for wrap bust type read transac- tion
axi4_non_blocking_wrap_burst_write_read_test	Checking for wrap bust type write read transaction
axi4_non_blocking_okay_response_write_test	Checking for okay response type write transaction

axi4_non_blocking_okay_response_read_test	Checking for okay response type read transaction
axi4_non_blocking_okay_response_write_read_test	Checking for okay response type write read transaction
axi4_non_blocking_8b_write_data_test	Checking for 8b write data transaction
axi4_non_blocking_8b_data_read_test	Checking for 8b read data transaction
axi4_non_blocking_8b_write_read_test	Checking for 8b write read transaction
axi4_non_blocking_16b_write_data_test	Checking for 16b write data transaction
axi4_non_blocking_16b_data_read_test	Checking for 16b read data transaction
axi4_non_blocking_16b_write_read_test	Checking for 16b write read transaction
axi4_non_blocking_32b_write_data_test	Checking for 32b write data transaction
axi4_non_blocking_32b_data_read_test	Checking for 32b read data transaction
axi4_non_blocking_32b_write_read_test	Checking for 32b write read transaction
axi4_non_blocking_64b_write_data_test	Checking for 64b write data transaction
axi4_non_blocking_64b_data_read_test	Checking for 64b read data transaction
axi4_non_blocking_64b_write_read_test	Checking for 64b write read transaction
axi4_non_blocking_unaligned_addr_write_read_test	Checking for unaligned address write read transactions
axi4_non_blocking_slave_error_write_read_test	Checking for slave error write read transactions
axi4_non_blocking_write_read_rand_test	Checking for write read random transactions
axi4_non_blocking_cross_write_read_test	Checking for cross of length X burst X size write read transactions

Chapter 10

User Guide

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

User Guide Link

Chapter 11

References

Reference Link 1

Reference Link 2

Reference Link 3

Reference Link 4