AXI4
Verification Components

User Guide for Release 2025.04

By

Jim Lewis

SynthWorks VHDL Training

Jim@SynthWorks.com

http://www.SynthWorks.com

Table of Contents

1. Terminology: Manager and Subordinate

OSVVM AXI4 VCs and this document use the more recent AXI terminology, Manager and Subordinate. An AXI Manager is the agent that initiates transactions, and a Subordinate is the agent that receives and responds to requests.

2. Overview

The OSVVM AXI4 Verification Components (VCs) facilitate testing the interface and functionality of AXI4 devices. The OSVVM AXI4 VCs include Axi4Manager, Axi4Subordinate, and Axi4Memory. These VCs are intended to be part of a structured test environment.

The AXI4 Manager verification component implements the complete AXI4 Manager interface capability. It supports both single word and burst transfers. For bursting it uses the BurstFifos in AddressBusRecType. The signals xAddr, xData, xStrb, xLen, xLast, xValid, and xReady, are set on a transaction-by-transaction basis. The signals xID, xSize, xBurst, xLock, xCache, xProt, xQOS, xRegion, xUser, and xResp are set using values inside the model.

The Axi4Subordinate is a transaction-based Subordinate. The Axi4Memory is a memory Subordinate that responds to operations on the AxiBus inteface.

The Axi4Memory responds to AXI4 accesses on AxiBus by either writing to or reading from its internal memory data structure (see OSVVM's MemoryGenericPkg). It handles both AXI4 single word and burst transfers. For bursting, it currently only handles incrementing bursts. The memory spans the complete address range of the verification component. MemoryPkg currently limits the memory address to 40 bits.

In addition to the AXI4 bus interface (connected to the DUT), the Axi4Memory also supports a transaction interface (TransRec) to the test sequencer (TextCtrl) that provides a "backdoor" (non-functional) transaction interface to the memory. This interface accesses the memory contents on a delta cycle basis.

The Axi4Subordinate verification component is a transaction-based component which allows the test sequencer to program an arbitrary sequence of responses. Currently the Axi4Subordinate only supports AXI4 single word transfers – primarily since burst transfers are handled by the Axi4Memory.

For the test case programming API (used in a test sequencer), the AXI4 VCs support the complete set of OSVVM Address Bus Model Independent Transactions. Using this interface ensures uniformity and consistency with other OSVVM VCs and improves verification test case reuse.

We are going to start with a brief overview and a demo of the AXI4 test environment.

PDF documents referenced in this document are in the directory OsvvmLibraries/Documentation.

3. OSVVM Testbench Architecture

3.1 Test Architecture Overview

The objective of any verification framework is to make the Device Under Test (DUT) "feel like" it has been plugged into the board. Hence, the framework must be able to produce the same waveforms and sequence of waveforms that the DUT will see on the board.

The OSVVM testbench framework looks identical to other frameworks, including SystemVerilog. It includes verification components (Axi4Manager, Axi4Subordinate, and Axi4Memory) and TestCtrl (the test sequencer) as shown in Figure 1. The top level of the testbench connects the components together (using the same methods as in RTL design) and is often called a test harness. Connections between the verification components and TestCtrl use VHDL records (which we call the transaction interface). Connections between the verification components and the DUT are the DUT interfaces (such as AxiStream, UART, AXI4, SPI, and I2C).

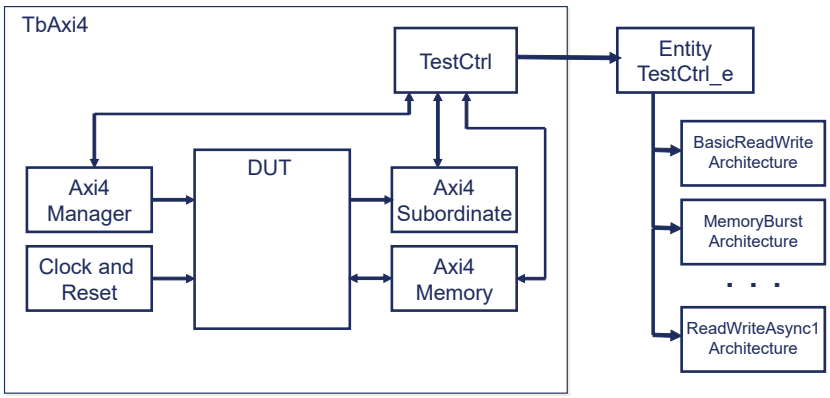

Figure 1. OSVVM Testbench Framework

3.2 Writing Tests

Writing tests is all about creating waveforms at an interface. In a basic test approach, each test directly drives and wiggles interface waveforms. This is tedious and error prone.

In OSVVM, signal wiggling is replaced by transactions. A transaction is an abstract representation of an interface waveform (such as Write) or a directive to the VC (such as wait for clock). A transaction is initiated using a procedure call. In a VC based approach, the procedure call collects the transaction information and passes it to the Axi4 VCs via a transaction interface (a record). The Axi4 VC then decodes this information and creates the corresponding interface waveforms.

Using transactions simplifies creating tests and increases their readability. Figure 2 shows calls to the Write, Read, and ReadCheck transactions for an Axi4Manager VC. In this test, the responses are provided by an Axi4Memory VC. Note that in the calls to Write, Read, and ReadCheck that the size of the data parameter determines the maximum size of the transactions on the interface.

```
ManagerProc : process
Begin
    . . .
    log("Write and Read with ByteAddr = 0, 4 Bytes") ;
    Write(ManagerRec, X"0000_0000", X"5555_5555" ) ;
    Read(ManagerRec,  X"0000_0000", Data) ;
    AffirmIfEqual(Data, X"5555_5555", "Super Read Data: ") ;

    log("Write and Read with 1 Byte, and ByteAddr = 1") ;
    Write(ManagerRec, X"0000_0011", X"22" ) ;
    ReadCheck(ManagerRec, X"0000_0011", X"22" ) ;

    log("Write and Read with 3 Bytes and ByteAddr = 0") ;
    Write(ManagerRec, X"0000_0050", X"33_2211" ) ;
    ReadCheck(ManagerRec, X"0000_0050", X"33_2211" ) ;
```

Figure 2. Calls to Axi4Manager Write, Read, and ReadCheck Transactions

3.3 **Address Bus Transaction Interface**

Each Axi4 Verification Component receives transactions from the test sequencer via a Transaction Interface. OSVVM implements the transaction interface as a record.

AddressBus Transaction Interface, AddressBusRecType, is used to connect the verification component to TestCtrl. AddressBusRecType, shown in Figure 3, is defined in the Address Bus Model Independent Transaction package, AddressBusTransactionPkg.vhd, which is in the directory OsvvmLbraries/Common/Src.

```
  type AddressBusRecType is record
    -- Handshaking controls
    --   Used by RequestTransaction in the Transaction Procedures
    --   Used by WaitForTransaction in the Verification Component
    --   RequestTransaction and WaitForTransaction are in osvvm.TbUtilPkg
    Rdy                : RdyType ;
    Ack                : AckType ;
    -- Transaction Type
    Operation          : AddressBusOperationType ;
    -- Address to verification component and its width
    -- Width may be smaller than Address
    Address            : std_logic_vector_max_c ;
    AddrWidth          : integer_max ;
    -- Data to and from the verification component and its width.
    -- Width will be smaller than Data for byte operations
    -- Width size requirements are enforced in the verification component
    DataToModel        : std_logic_vector_max_c ;
    DataFromModel      : std_logic_vector_max_c ;
    DataWidth          : integer_max ;
    -- Burst FIFOs
    WriteBurstFifo     : ScoreboardIdType ;
    ReadBurstFifo      : ScoreboardIdType ;
    -- StatusMsgOn provides transaction messaging override.
    -- When true, print transaction messaging independent of
```

```
    -- other verification based based controls.
    StatusMsgOn        : boolean_max ;
    -- Verification Component Options Parameters - used by SetModelOptions
    IntToModel         : integer_max ;
    IntFromModel       : integer_max ;
    BoolToModel        : boolean_max ;
    BoolFromModel      : boolean_max ;
    -- Verification Component Options Type
    Options            : integer_max ;
  end record AddressBusRecType ;
```

Figure 3. AddressBusRecType

Note that Address, DataToModel, and DataFromModel are unconstrained. Hence, when they are used in a signal declaration they must be constrained. Address needs to be sized to match the maximum (AWADDR'length, ARADDR'length). DataToModel and DataFromModel need to be sized to match the maximum(WDATA'length, RDATA'length).

Figure 4 shows the declaration ManagerRec (which connects the Axi4Manager to TestCtrl) and SubordinateRec (which connects the Axi4Subordinate to TestCtrl).

```
  signal ManagerRec, SubordinateRec  : AddressBusRecType(
          Address      (AXI_ADDR_WIDTH-1 downto 0),
          DataToModel  (AXI_DATA_WIDTH-1 downto 0),
          DataFromModel(AXI_DATA_WIDTH-1 downto 0)
        ) ;
```

Figure 4. ManagerRec and SubordinateRec

3.4 **AxiBus : Axi4RecType**

AxiBus is defined as a record of type Axi4RecType, shown in Figure 5 is defined in the package Axi4InterfacePkg.vhd which is in the directory OsvvmLbraries/AXI4/common/src.

```
  type Axi4WriteAddressRecType is record
    -- AXI4 Lite
    Addr      : std_logic_vector ;
    Prot      : Axi4ProtType ;
    Valid     : std_logic ;
    Ready     : std_logic ;
    -- AXI4 Full
    -- User Config - AXI recommended 3:0 for Manager, 7:0 at slave
    ID        : std_logic_vector ;
    -- BurstLength = AxLen+1.  AXI4: 7:0,  AXI3: 3:0
    Len       : std_logic_vector(7 downto 0) ;
    -- #Bytes in transfer = 2**AxSize
    Size      : std_logic_vector(2 downto 0) ;
    -- AxBurst Binary Encoded (Fixed, Incr, Wrap, NotDefined)
    Burst     : std_logic_vector(1 downto 0) ;
    Lock      : std_logic ;
    -- AxCache bits (Write-Allocate, Read-Allocate, Modifiable, Bufferable)
    Cache     : std_logic_vector(3 downto 0) ;
    QOS       : std_logic_vector(3 downto 0) ;
```

```
    Region    : std_logic_vector(3 downto 0) ;
    User      : std_logic_vector ; -- User Config
  end record Axi4WriteAddressRecType ;

  type Axi4WriteDataRecType is record
    -- AXI4 Lite
    Data      : std_logic_vector ;
    Strb      : std_logic_vector ;
    Valid     : std_logic ;
    Ready     : std_logic ;
    -- AXI 4 Full
    Last      : std_logic ;
    User      : std_logic_vector ;
    -- AXI3
    ID        : std_logic_vector ;
  end record Axi4WriteDataRecType ;

  type Axi4WriteResponseRecType is record
    -- AXI4 Lite
    Valid     : std_logic ;
    Ready     : std_logic ;
    Resp      : Axi4RespType ;
    -- AXI 4 Full
    ID        : std_logic_vector ;
    User      : std_logic_vector ;
  end record Axi4WriteResponseRecType ;

  type Axi4ReadAddressRecType is record
    -- AXI4 Lite
    Addr      : std_logic_vector ;
    Prot      : Axi4ProtType ;
    Valid     : std_logic ;
    Ready     : std_logic ;
    -- AXI 4 Full
    -- User Config - AXI recommended 3:0 for Manager, 7:0 at slave
    ID        : std_logic_vector ;
    -- BurstLength = AxLen+1.  AXI4: 7:0,  AXI3: 3:0
    Len       : std_logic_vector(7 downto 0) ;
    -- #Bytes in transfer = 2**AxSize
    Size      : std_logic_vector(2 downto 0) ;
    -- AxBurst Binary Encoded (Fixed, Incr, Wrap, NotDefined)
    Burst     : std_logic_vector(1 downto 0) ;
    Lock      : std_logic ;
    -- AxCache bits (Write-Allocate, Read-Allocate, Modifiable, Bufferable)
    Cache     : std_logic_vector(3 downto 0) ;
    QOS       : std_logic_vector(3 downto 0) ;
    Region    : std_logic_vector(3 downto 0) ;
    User      : std_logic_vector ; -- User Config
  end record Axi4ReadAddressRecType ;

  type Axi4ReadDataRecType is record
    -- AXI4 Lite
    Data      : std_logic_vector ;
```

```
    Resp       : Axi4RespType ;
    Valid      : std_logic ;
    Ready      : std_logic ;
    -- AXI 4 Full
    Last       : std_logic ;
    User       : std_logic_vector ;
    ID         : std_logic_vector ;
  end record Axi4ReadDataRecType ;

  type Axi4BaseRecType is record
    WriteAddress   : Axi4WriteAddressRecType ;
    WriteData      : Axi4WriteDataRecType ;
    WriteResponse  : Axi4WriteResponseRecType ;
    ReadAddress    : Axi4ReadAddressRecType ;
    ReadData       : Axi4ReadDataRecType ;
  end record Axi4BaseRecType ;

  alias Axi4RecType is Axi4BaseRecType ;
```

Figure 5. Axi4BaseRecType

Figure 6 shows the declaration AxiBus. The numerous unconstrained elements of Axi4BaseRecType are constrained in the signal declaration.

```
  signal   AxiBus : Axi4RecType(
    WriteAddress(
      Addr(AXI_ADDR_WIDTH-1 downto 0),
      ID  (7 downto 0),
      User(7 downto 0)
    ),
    WriteData   (
      Data(AXI_DATA_WIDTH-1 downto 0),
      Strb(AXI_STRB_WIDTH-1 downto 0),
      User(7 downto 0),
      ID  (7 downto 0)
    ),
    WriteResponse(
      ID  (7 downto 0),
      User(7 downto 0)
    ),
    ReadAddress (
      Addr(AXI_ADDR_WIDTH-1 downto 0),
      ID  (7 downto 0),
      User(7 downto 0)
    ),
    ReadData    (
      Data(AXI_DATA_WIDTH-1 downto 0),
      ID  (7 downto 0),
      User(7 downto 0)
    )
  ) ;
```

Figure 6. AxiBus

3.5 **Axi4 Context Declaration**

To simplify the usage of OSVVM AXI4 packages, a context declaration that references all of the OSVVM AXI4 packages is provided. Using a context declaration allows the packages to be refactored without impacting the designs that reference the packages using the context. Figure 7 shows the Axi4Context as defined in Axi4Context.vhd.

```
context Axi4Context is
    library osvvm_common ;
    context osvvm_common.OsvvmCommonContext; -- Address Bus Transactions

    library osvvm_axi4 ;
    use osvvm_axi4.Axi4CommonPkg.all ;       -- AXI handshaking
    use osvvm_axi4.Axi4InterfacePkg.all ;    -- Interface definition
    use osvvm_axi4.Axi4OptionsPkg.all ;      -- Model parameters
    use osvvm_axi4.Axi4ModelPkg.all ;        -- Model support

    use osvvm_axi4.Axi4ComponentPkg.all ;    -- Port Based Interface
    use osvvm_axi4.Axi4ComponentVtiPkg.all ; -- Virtual Interface

    -- Package of aliases to maintain compatibility with the past
    use osvvm_axi4.Axi4VersionCompatibilityPkg.all ;
end context Axi4Context ;
```

Figure 7. Axi4Context

3.6 **Component Declarations for Axi4 Verification Components**

OSVVM prefers to use component instances. One good reason is they support configuration declarations and direct entity instances do not.

To make usage of component instances easier than direct entity instances, component declarations for each verification component is provided in a package and the package is referenced by Axi4Context.

4. Demo Preparation: Getting and Building the OSVVM Libraries

OSVVM is available on GitHub at https://github.com/OSVVM as a git repository or at https://osvvm.org/downloads as a ZIP file. Retrieve OSVVM from GitHub using git as shown in Figure 8. Note that the "—recursive" option is required since the OSVVM repositories are submodules of OsvvmLibraries. Submodules greatly simplify development and deployment of the libraries.

```
git clone --recursive https://github.com/OSVVM/OsvvmLibraries.git
```

Figure 8. Retrieving OSVVM from GitHub

Prior to starting the OSVVM scripting environment, create a directory named sim in which to run your simulations. Start your simulator and go to the sim directory. Once there, use the steps in Figure 9 to build the OSVVM Libraries (utility and verification component). These directions are supported in Mentor QuestaSim/ModelSim or Aldec RivieraPRO. Aldec's ActiveHDL is also supported but requires a few extra steps. For these steps and additional details of the OSVVM scripting environment see Script_user_guide.pdf (in OsvvmLibraries/Documentation).

```
cd sim
source ../OsvvmLibraries/Scripts/StartUp.tcl
build  ../OsvvmLibraries
```

Figure 9. Building (Compiling) OSVVM

The intent of the OSVVM scripting is to make compiling and running your simulations independent of the simulator you are using. We hope to update the scripting environment to support Synopsys and Cadence tools in the first half of 2021.

GHDL can be run using tclsh. In windows, using MSYS2/MinGW64 start tclsh using "winpty tclsh".

5. TbAxi4: Port Based Interface

In the OSVVM Port Based Interfaces approach, the transaction interfaces are record ports of the verification components (VCs) and the test sequencer (TestCtrl). The testbench then simply connects the ports together using, just like we do for RTL design. OSVVM and its predecessor within SynthWorks has used this transaction interface methodology since 1997.

The OSVVM Port Based Interface approach works well when the testbench components are external to the device being tested. OSVVM's Virtual Transaction Interfaces (see next section) provide a simplified means to connect to a verification component that is internal to the design – such as an embedded processor core.

OSVVM components with Virtual Transaction Interfaces interoperate well with OSVVM components with Port Based Interfaces.

5.1 Demo: Running the AXI4 Testbenches

Prior to doing this step, do the steps in section 3, Demo Preparation.

Use the steps in Figure 10 to compile and run the tests for the Axi4 verification components in Mentor QuestaSim/ModelSim or Aldec RivieraPRO. If you have not exited the simulator, you only need to do the "build" step.

```
cd sim
do ../OsvvmLibraries/startup.tcl
build ../OsvvmLibraries/AXI4/Axi4/RunDemoTests.pro
```

Figure 10. Compiling and Running OSVVM

5.2 **TbAxi4: Axi4 Test Environment - Port Based Interface**

In the previous section, you ran the Axi4 Testbench (TbAxi4.vhd). It is in the directory OsvvmLibraries/AXI4/Axi4/testbench. It is structured as shown in Figure 11.

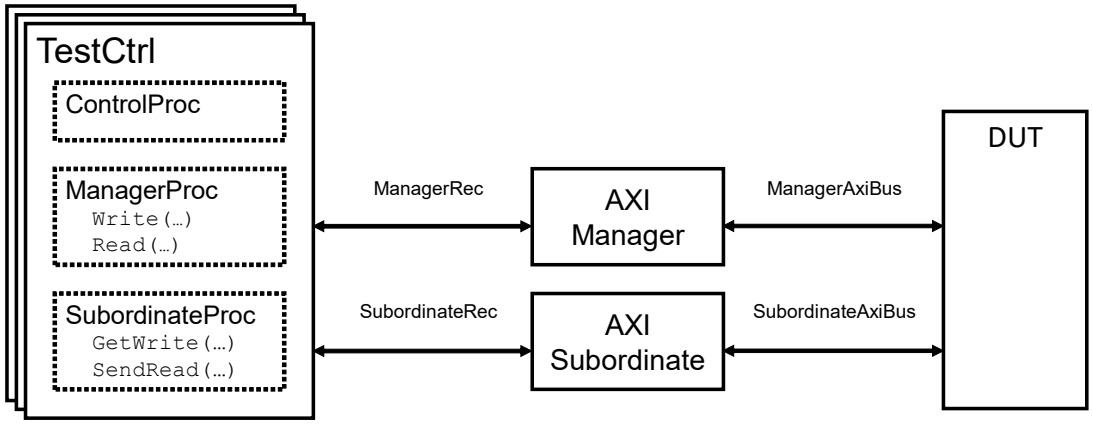

Figure 11. TbAxi4

TbAxi4 is a test harness that connects components together. In an RTL design, this code is also called structural code or a netlist. A sketch of TbAxi4.vhd is shown in Figure 12. For more details, see TbAxi4.vhd. Note that OSVVM uses VHDL-2008 external names and the order of instantiation is important. First instantiate the design under test, next the verification components, and then finally the test sequencer (TestCtrl).

```
library osvvm_Axi4 ;
  context osvvm_axi4.Axi4Context ;
. . .
entity  TbAxi4  is
end entity TbAxi4 ;
architecture TestHarness of TbAxi4 is
  signal ManagerRec, SubordinateRec  : AddressBusRecType(
          Address      (AXI_ADDR_WIDTH-1 downto 0),
          DataToModel  (AXI_DATA_WIDTH-1 downto 0),
          DataFromModel(AXI_DATA_WIDTH-1 downto 0)
        ) ;
  signal AxiBus : Axi4RecType( . . . ) ;
  . . .
begin
  osvvm.TbUtilPkg.CreateClock(Clk, tperiod_Clk) ;
  osvvm.TbUtilPkg.CreateReset(nReset, . . .) ;
  DUT :           Axi4PassThru    port map (…, …) ;
  Subordinate_1 : Axi4Subordinate port map (…, AxiBus, SubordinateRec) ;
  Manager_1 :     Axi4Manager     port map (…, AxiBus, ManagerRec);
  TestCtrl_1 :    TestCtrl        port map (nReset, ManagerRec, SubordinateRec) ;
end TestHarness ;
```

Figure 12. A sketch of TbAxi4.vhd

By default, each OSVVM verification component uses its instance label as the name it reports when an alert or log within the model is called. This allows each message to be tracked to a unique verification component. AlertLogIDs can be looked up using this name, so picking a good instance label will simplify looking up the AlertLogID for each verification component from the test sequencer (TestCtrl). These names can also be set by the generic MODEL_ID_NAME. The only reason to do this is to allow verification components to share the same AlertLogID.

We recommend using the "ComponentName_1". In this case we shortened the names to Manager_1 and Subordinate_1. Our intent is to reuse some of the same test cases with other Manager and Subordinate verification components (such as Avalon and Wishbone) – and hence the more generic naming.

5.3 TestCtrl Entity – Port Based Interface

Tests are written as architectures of the test sequencer, TestCtrl. The entity for TestCtrl, shown in Figure 13, consists of transaction interface connections. It uses records for the transaction interfaces (ManagerRec and SubordinateRec). These records connect to the Axi4Manager and Axi4Subordinate/Axi4Memory components.

```
library ieee ;
   use . . .

library osvvm ;
  context osvvm.OsvvmContext ;
  use osvvm.ScoreboardPkg_slv.all ;

library OSVVM_AXI4 ;
  context OSVVM_AXI4.Axi4Context ;

entity TestCtrl is
  port (
    -- Global Signal Interface
    nReset          : in    std_logic ;

    -- Transaction Interfaces
    ManagerRec       : inout AddressBusRecType ;
    SubordinateRec    : inout AddressBusRecType
  ) ;

  -- Derive AXI interface properties from the interface
  constant AXI_ADDR_WIDTH : integer := ManagerRec.Address'length ;
  constant AXI_DATA_WIDTH : integer := ManagerRec.DataToModel'length ;
  constant AXI_DATA_BYTE_WIDTH : integer := AXI_DATA_WIDTH / 8 ;
  constant AXI_BYTE_ADDR_WIDTH : integer :=
      integer(ceil(log2(real(AXI_DATA_BYTE_WIDTH)))) ;

  -- Simplifying access to Burst FIFOs using aliases
  alias WriteBurstFifo : ScoreboardIdType is ManagerRec.WriteBurstFifo ;
  alias ReadBurstFifo  : ScoreboardIdType is ManagerRec.ReadBurstFifo ;

end entity TestCtrl ;
```

Figure 13. TestCtrl_e.vhd

Note the reference to "osvvm.ScoreboardPkg_slv" is required with the 2021.06 update.

5.4 **Axi4Manager Entity Interface – Port Based Interface**

The Axi4Manager entity interface is shown in Figure 14. It uses records for both the AXI4 (AxiBus) and transaction interface (TransRec). The AxiBus implements all signals in the AXI interface.

Transactions are initiated by the test sequencer (TestCtrl) by calling a procedure from the transaction API (defined by Address Bus Model Independent Transactions). These are passed through the transaction interface (TransRec) to the Axi4Manager which then creates the requested interactions with the DUT via AxiBus.

For generics, it has MODEL_ID_NAME which optionally specifies the model AlertLogID name. If the MODEL_ID_NAME is not specified, the component instance label will be used (preferred method). The remaining generics specify timing. Tperiod_Clk specifies the clock frequency. Tpd_Clk_* specifies the delay for each interface output. See the file Axi4Manager.vhd for the details of the generics.

```
entity Axi4Manager is
generic (
  MODEL_ID_NAME    : string :="" ;
  tperiod_Clk      : time   := 10 ns ;

  tpd_Clk_AWAddr   : time   := 2 ns ;
  -- . . .  see entity for remaining generics
  tpd_Clk_RReady   : time   := 2 ns
) ;
port (
  -- Globals
  Clk         : in   std_logic ;
  nReset      : in   std_logic ;

  -- AXI Manager Functional Interface
  AxiBus      : inout Axi4RecType ;

  -- Testbench Transaction Interface
  TransRec    : inout AddressBusRecType
) ;

  -- Model Configuration
  -- Access via transactions or external name
  shared variable params : ModelParametersPType ;

end entity Axi4Manager ;
```

Figure 14. Axi4Manager

5.5 Axi4Subordinate Entity Interface – Port Based Interface

The Axi4Subordinate entity interface is shown in Figure 15. It uses records for both the AxiBus and transaction interfaces (TransRec). The AxiBus implements all signals in the AXI interface. It supports single word operations. For bursts, see Axi4Memory.

Transactions are initiated by the DUT via the AxiBus interface. The test sequencer (TestCtrl) responds to operations AxiBus by calling a procedure from the transaction API (defined by Address Bus Model Independent Transactions). The procedures pass the transaction information to Axi4Subordinate via the transaction interface (TransRec).

For generics, it has MODEL_ID_NAME which optionally specifies the model AlertLogID name. If the MODEL_ID_NAME is not specified, the component instance label will be used (preferred method). The remaining generics specify timing. Tperiod_Clk specifies the clock frequency. Tpd_Clk_* specifies the delay for each interface output. See the file Axi4Manager.vhd for the details of the generics.

```
entity Axi4Subordinate is
generic (
  MODEL_ID_NAME   : string :="" ;
  tperiod_Clk     : time := 10 ns ;

  tpd_Clk_AWReady : time := 2 ns ;
  -- . . .  see entity for remaining generics
  tpd_Clk_RResp   : time := 2 ns
) ;
port (
  -- Globals
  Clk         : in   std_logic ;
  nReset      : in   std_logic ;

  -- AXI Manager Functional Interface
  AxiBus      : inout Axi4RecType ;

  -- Testbench Transaction Interface
  TransRec    : inout AddressBusRecType
) ;

  -- Model Configuration
  -- Access via transactions or external name
  shared variable Params : ModelParametersPType ;

end entity Axi4Subordinate ;
```

Figure 15. Axi4Subordinate

5.6 **Axi4Memory Entity Interface – Port Based Interface**

The Axi4Memory entity interface is shown in Figure 16. It uses records for both the AxiBus and transaction interfaces (TransRec). The AxiBus implements all signals in the AXI interface. Axi4Memory is a subordinate AXI4 VC that internally has memory in the entire address range.

The Axi4Memory responds to AXI4 accesses on AxiBus by either writing to or reading from its internal memory data structure (see OSVVM's MemoryGenericPkg). It handles both AXI4 single word and burst transfers. For bursting, it currently only handles incrementing bursts. The memory spans the complete address range of the verification component. MemoryPkg currently limits the memory address to 40 bits.

In addition to the AXI4 bus interface (connected to the DUT), the Axi4Memory also supports a transaction interface (TransRec) to the test sequencer (TextCtrl) that provides a "backdoor" (non-functional) transaction interface to the memory. This interface accesses the memory contents on a delta cycle basis.

For generics, it has MODEL_ID_NAME which optionally specifies the model AlertLogID name. If the MODEL_ID_NAME is not specified, the component instance label will be used (preferred method). The remaining generics specify timing. Tperiod_Clk specifies the clock frequency. Tpd_Clk_* specifies the delay for each interface output. See the file Axi4Manager.vhd for the details of the generics.

```
entity Axi4Memory is
generic (
  MODEL_ID_NAME   : string :="" ;
  MEMORY_NAME     : string := "" ;
  tperiod_Clk     : time := 10 ns ;

  tpd_Clk_AWReady : time := 2 ns ;
  -- . . .  see entity for remaining generics
  tpd_Clk_RLast   : time := 2 ns
) ;
port (
  -- Globals
  Clk         : in   std_logic ;
  nReset      : in   std_logic ;

  -- AXI Manager Functional Interface
  AxiBus      : inout Axi4RecType ;

  -- Testbench Transaction Interface
  TransRec    : inout AddressBusRecType
) ;

  -- Model Configuration
  -- Access via transactions or external name
  shared variable Params : ModelParametersPType ;

end entity Axi4Memory ;
```

Figure 16. Axi4Memory

5.6.1 Setting the Name of the Axi4Memory Model

The Axi4Memory name is the value of MEMORY_NAME if it is set (other than ""). If MEMORY_NAME is not set, the memory name is Axi4Memory'PATH_NAME.

If two memory models have the same name, they use the same instance of the memory and share the same memory space.

6. TbAxi4: AXI4 - Virtual Transaction Interface

In the Virtual Transaction Interface approach, the transaction interfaces are internal record signals of the verification components (VCs). The test sequencer (TestCtrl) connects to these using VHDL-2008 external names (hierarchical references). OSVVM Virtual Transaction Interfaces are a new feature of the OSVVM 2020.12 release.

OSVVM's Virtual Transaction Interfaces provide a simplified means to connect to a verification component that is internal to the design – such as an embedded processor core. They also simplify any testbench since they remove the need to use hierarchical connections.

OSVVM components with Virtual Transaction Interfaces interoperate well with OSVVM components with Port Based Interfaces.

6.1 Demo: Running the AXI4 Testbenches

Prior to doing this step, do the steps in section 3, Demo Preparation.

Use the steps in Figure 17 to compile and run the tests for the Axi4 verification components in Mentor QuestaSim/ModelSim or Aldec RivieraPRO. If you have not exited the simulator, you only need to do the "build" step.

```
cd sim
do ../OsvvmLibraries/startup.tcl
build ../OsvvmLibraries/AXI4/Axi4/RunDemoTestsVti.pro
```

Figure 17. Compiling and Running OSVVM

6.2 **TbAxi4: Axi4 Test Environment – Virtual Transaction Interface**

In the previous section, you ran TbAxi4.vhd, the Axi4 Testbench. It is in the directory OsvvmLibraries/AXI4/Axi4/testbenchVti. It is structured as shown in Figure 18. The record connections (shown with dotted lines) use external names rather than direct signal connections.

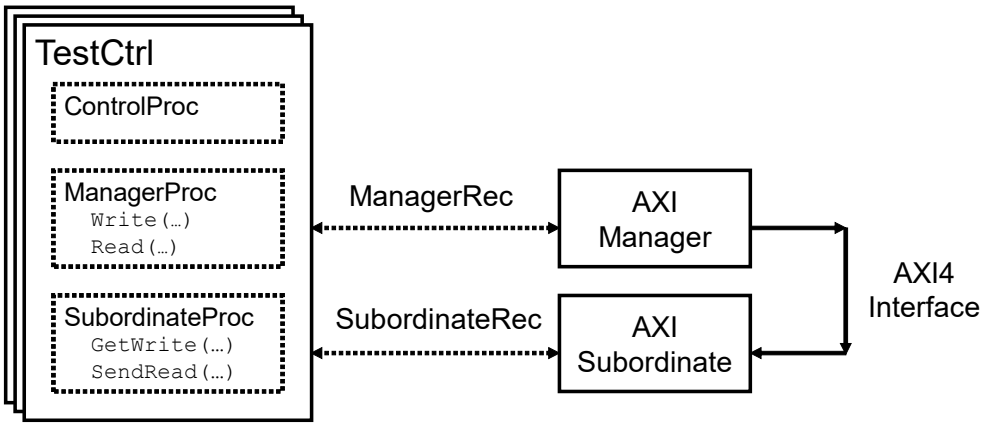

Figure 18. TbAxi4 for Virtual Transaction Interfaces

TbAxi4 is a test harness that connects components together. In an RTL design, this code is also called structural code or a netlist. A sketch of TbAxi4.vhd is shown in Figure 19. For more details, see Axi4/testbenchVti/TbAxi4.vhd. Note that there are no transaction interface signals to connect between the verification components (Axi4Manager and Axi4Subordinate/Axi4Memory) and the test sequencer (TestCtrl). Instead, these are now connected via VHDL-2008 external names. See the entity declaration for TestCtrl for details. Since external names are used, the order of instantiation is important. First instantiate the design under test, next the verification components, and then finally the test sequencer (TestCtrl).

```
library osvvm_Axi4 ;
  context osvvm_axi4.Axi4Context ;
. . .
entity  TbAxi4  is
end entity TbAxi4 ;
architecture TestHarness of TbAxi4 is
  signal AxiBus : Axi4RecType( . . . ) ;
  . . .
begin
  osvvm.TbUtilPkg.CreateClock(Clk, tperiod_Clk) ;
  osvvm.TbUtilPkg.CreateReset(nReset, . . .) ;
  Subordinate_1 : Axi4Subordinate(…, AxiBus) ;
  Manager_1 :    Axi4Manager   (…, AxiBus);
  TestCtrl_1 :  TestCtrl     (nReset) ;
end TestHarness ;
```

Figure 19. A sketch of TbAxi4.vhd for Virtual Transaction Interfaces

6.3 TestCtrl Entity – Virtual Transaction Interface (External Names)

Tests are written as architectures of the test sequencer, TestCtrl (testbenchVTI). The entity for TestCtrl, shown in Figure 20, consists of transaction interface connections.

```
library ieee ;
   use . . .

library osvvm ;
  context osvvm.OsvvmContext ;
  use osvvm.ScoreboardPkg_slv.all ;

library OSVVM_AXI4 ;
  context OSVVM_AXI4.Axi4Context ;

entity TestCtrl is
  port (
    -- Global Signal Interface
    nReset          : in    std_logic
  ) ;

  -- Connect transaction interfaces using external names
  alias ManagerRec is <<signal ^.Manager_1.TransRec : AddressBusRecType>>;
  alias SubordinateRec is <<signal ^.Subordinate_1.TransRec :
      AddressBusRecType>>;

  -- Derive AXI interface properties from the ManagerRec
  constant AXI_ADDR_WIDTH : integer := ManagerRec.Address'length ;
  constant AXI_DATA_WIDTH : integer := ManagerRec.DataToModel'length ;
  constant AXI_DATA_BYTE_WIDTH : integer := AXI_DATA_WIDTH / 8 ;
  constant AXI_BYTE_ADDR_WIDTH : integer :=
      integer(ceil(log2(real(AXI_DATA_BYTE_WIDTH)))) ;

  -- Simplifying access to Burst FIFOs using aliases
  alias WriteBurstFifo : ScoreboardIdType is ManagerRec.WriteBurstFifo ;
  alias ReadBurstFifo  : ScoreboardIdType is ManagerRec.ReadBurstFifo ;

end entity TestCtrl ;
```

Figure 20. TestCtrl_e.vhd

Note the reference to "osvvm.ScoreboardPkg_slv" is required with the 2021.06 update.

6.4 **Axi4ManagerVti Entity Interface – Virtual Transaction Interface (External Names)**

The Axi4ManagerVti entity interface is shown in Figure 21. It uses a record for the AXI4 (AxiBus) port. The AxiBus implements all signals in the AXI interface. The transaction interface (TransRec) is declared as a signal in the entity declarative region. It is accessed using an external name in the test sequencer (TestCtrl).

For generics, it has MODEL_ID_NAME which optionally specifies the model AlertLogID name. If the MODEL_ID_NAME is not specified, the component instance label will be used (preferred method). The remaining generics specify timing. Tperiod_Clk specifies the clock frequency. Tpd_Clk_* specifies the delay for each interface output. See the file Axi4ManagerVti.vhd for the details of the generics.

```
entity Axi4ManagerVti is
generic (
  MODEL_ID_NAME    : string :="" ;
  tperiod_Clk      : time   := 10 ns ;
  tpd_Clk_AWAddr   : time   := 2 ns ;
  -- . . .  see entity for remaining generics
) ;
port (
  -- Globals
  Clk         : in   std_logic ;
  nReset      : in   std_logic ;

  -- AXI Manager Functional Interface
  AxiBus      : inout Axi4RecType
) ;

  -- Model Configuration
  -- Access via transactions or external name
  shared variable params : ModelParametersPType ;

  -- Derive AXI interface properties from the AxiBus
  alias AxiAddr is AxiBus.WriteAddress.Addr ;
  alias AxiData is AxiBus.WriteData.Data ;
  constant AXI_ADDR_WIDTH      : integer := AxiAddr'length ;
  constant AXI_DATA_WIDTH      : integer := AxiData'length ;

  -- Testbench Transaction Interface - Access via external names
  signal TransRec : AddressBusRecType (
          Address      (AXI_ADDR_WIDTH-1 downto 0),
          DataToModel  (AXI_DATA_WIDTH-1 downto 0),
          DataFromModel(AXI_DATA_WIDTH-1 downto 0)
        ) ;
end entity Axi4ManagerVti ;
```

Figure 21. Axi4Manager

6.5 Axi4SubordinateVti Entity Interface – Virtual Transaction Interface (External Names)

The Axi4SubordinateVti entity interface is shown in Figure 22. It uses records for the AXI4 (AxiBus) port. The AxiBus implements all signals in the AXI interface. The transaction interface (TransRec) is declared as a signal in the entity declarative region. It is accessed using an external name in the test sequencer (TestCtrl).

For generics, it has MODEL_ID_NAME which optionally specifies the model AlertLogID name. If the MODEL_ID_NAME is not specified, the component instance label will be used (preferred method). The remaining generics specify timing. Tperiod_Clk specifies the clock frequency. Tpd_Clk_* specifies the delay for each interface output. See the file Axi4SubordinateVti.vhd for the details of the generics.

```
entity Axi4SubordinateVti is
generic (
  MODEL_ID_NAME   : string :="" ;
  tperiod_Clk     : time := 10 ns ;
  tpd_Clk_AWReady : time := 2 ns ;
  -- . . .  see entity for remaining generics
  tpd_Clk_RResp   : time := 2 ns
) ;
port (
  -- Globals
  Clk         : in   std_logic ;
  nReset      : in   std_logic ;

  -- AXI Manager Functional Interface
  AxiBus      : inout Axi4RecType
) ;

  -- Model Configuration
  -- Access via transactions or external name
  shared variable Params : ModelParametersPType ;

  -- Derive AXI interface properties from the AxiBus
  alias    AxiAddr is AxiBus.WriteAddress.Addr ;
  alias    AxiData is AxiBus.WriteData.Data ;
  constant AXI_ADDR_WIDTH : integer := AxiAddr'length ;
  constant AXI_DATA_WIDTH : integer := AxiData'length ;

  -- Testbench Transaction Interface
  -- Access via external names
  signal TransRec : AddressBusRecType (
          Address      (AXI_ADDR_WIDTH-1 downto 0),
          DataToModel  (AXI_DATA_WIDTH-1 downto 0),
          DataFromModel(AXI_DATA_WIDTH-1 downto 0)
        ) ;
end entity Axi4SubordinateVti ;
```

Figure 22. Axi4Subordinate

6.6 **Axi4MemoryVti Entity Interface – Virtual Transaction Interface (External Names)**

The Axi4MemoryVti entity interface is shown in Figure 23. It uses a record for the AXI4 (AxiBus) port. The AxiBus implements all signals in the AXI interface. The transaction interface (TransRec) is declared as a signal in the entity declarative region. It is accessed using an external name in the test sequencer (TestCtrl).

The Axi4Memory responds to AXI4 accesses on AxiBus by either writing to or reading from its internal memory data structure (see OSVVM's MemoryGenericPkg). It handles both AXI4 single word and burst transfers. For bursting, it currently only handles incrementing bursts. The memory spans the complete address range of the verification component. MemoryPkg currently limits the memory address to 40 bits.

In addition to the AXI4 bus interface (connected to the DUT), the Axi4Memory also supports a transaction interface (TransRec) to the test sequencer (TextCtrl) that provides a "backdoor" (non-functional) transaction interface to the memory. This interface accesses the memory contents on a delta cycle basis.

For generics, it has MODEL_ID_NAME which optionally specifies the model AlertLogID name. If the MODEL_ID_NAME is not specified, the component instance label will be used (preferred method). The remaining generics specify timing. Tperiod_Clk specifies the clock frequency. Tpd_Clk_* specifies the delay for each interface output. See the file Axi4MemoryVti.vhd for the details of the generics.

```
entity Axi4MemoryVti is
generic (
  MODEL_ID_NAME   : string :="" ;
  MEMORY_NAME     : string := "" ;
  tperiod_Clk     : time := 10 ns ;
  tpd_Clk_AWReady : time := 2 ns ;
  -- . . .  see entity for remaining generics
  tpd_Clk_RLast   : time := 2 ns
) ;
port (
  -- Globals
  Clk         : in   std_logic ;
  nReset      : in   std_logic ;

  -- AXI Manager Functional Interface
  AxiBus      : inout Axi4RecType
) ;
  -- Derive AXI interface properties from the AxiBus
  constant AXI_ADDR_WIDTH : integer := AxiBus.WriteAddress.Addr'length ;
  constant AXI_DATA_WIDTH : integer := AxiBus.WriteData.Data'length ;

  -- Testbench Transaction Interface - Access via external names
  signal TransRec : AddressBusRecType (
          Address      (AXI_ADDR_WIDTH-1 downto 0),
          DataToModel  (AXI_DATA_WIDTH-1 downto 0),
          DataFromModel(AXI_DATA_WIDTH-1 downto 0)
        ) ;
end entity Axi4MemoryVti ;
```

Figure 23. Axi4MemoryVti

6.6.1 Setting the Name of the Axi4Memory Model

The Axi4Memory name is the value of MEMORY_NAME if it is set (other than ""). If MEMORY_NAME is not set, the memory name is Axi4Memory'PATH_NAME.

If two memory models have the same name, they use the same instance of the memory and share the same memory space.

7. Writing Tests Using the Axi4 VCs

Tests are written by calling transactions in an architecture of TestCtrl (the test sequencer). Each separate test is a separate architecture of TestCtrl. Each test generates a sequence of waveforms that verify a particular aspect of the design. Hence, an entire test is visible in a single file, improving readability.

The TestCtrl architecture consists of a control process plus one process per independent interface, see Figure 24. The control process is used for test initialization and finalization. Each test process creates interface waveform sequences by calling the transaction procedures (Write, WriteBurst, Read, ReadBurst, Check …). This test architecture is based on the test TbAxi4_MemoryReadWrite1.vhd in the directory OsvvmLibraries/AXI4/Axi4/testbench.

For more details on writing tests, see the OSVVM Test Writers User Guide. For more details on writing tests, see the OSVVM Test Writers User Guide. For examples of using specific Address Bus transactions, see the Address Bus Independent Transactions User Guide.

```
architecture MemoryReadWrite1 of TestCtrl is
  . . .
begin
  ControlProc : process
  begin
    . . .
    WaitForBarrier(TestDone, 35 ms) ;
    ReportAlerts ;
    std.env.stop;
  end process ;

  ManagerProc : process
  begin
    WaitForClock(ManagerRec, 2) ;
    . . .
    Write(ManagerRec, X"0000_0000", X"5555_5555" ) ;
    Read (ManagerRec, X"0000_0000", Data) ;
    AffirmIfEqual(Data, X"5555_5555", "Manager Read Data: ") ;

    Write(ManagerRec, X"0000_0010", X"11" ) ;
    ReadCheck(ManagerRec, X"0000_0010", X"11" ) ;
    . . .
    WaitForBarrier(ManagerDone) ;
    WaitForBarrier(TestDone) ;
  end process ManagerProc;

  SubordinateProc : process
  begin
    WaitForBarrier(ManagerDone) ;
    -- Backdoor transaction access to Axi4Memory.vhd
    ReadCheck(SubordinateRec, X"0000_0000", X"5555_5555" ) ;
    ReadCheck(SubordinateRec, X"0000_0010", X"11" ) ;
    . . .
    WaitForBarrier(TestDone) ;
  end process ReceiverProc ;
end MemoryReadWrite1 ;
```

Figure 24. TestCtrl Architecture

8. AXI4 VC Transactions

The AXI4 VCs implement the OSVVM Address Bus Model Independent Transactions. The following is a summary of the supported transactions supported by each VC. For details of the transactions, parameter types, and examples, see the Address Bus Model Independent Transactions User Guide in the documentation repository.

8.1 Transactions Supported by Axi4Manager

The oData and iData in Manager transactions should match the size of the transfer. For example, if the transfer is for a byte of data, then only 8 bits should be present. Using the address byte address and the data width, the specification of interface characteristics such byte enables in the transaction interface is not necessary (as it is redundant information).

8.1.1 Interface Independent Transactions

An interface independent transaction can be implemented by any interface. An interface independent transaction contains all the information needed for an interface operation.

8.1.1.1 Interface Independent Write Transactions

```
Write(TRec, iAddr, iData, [MsgOn])
```

```
WriteAsync(TRec, iAddr, iData, [MsgOn])
```

```
WriteBurst         (TRec, iAddr, iNumFifoWords, [MsgOn])
```

```
WriteBurstAsync    (TRec, iAddr, iNumFifoWords, [StatusMsgOn])
```

```
WriteBurstVector   (TRec, iAddr, VectorOfWords, [MsgOn])
WriteBurstIncrement(TRec, iAddr, FirstWord, iNumFifoWords, [MsgOn])
WriteBurstRandom   (TRec, iAddr, FirstWord, iNumFifoWords, [MsgOn])
WriteBurstRandom   (TRec, iAddr, CoverID, iNumFifoWords, FifoWidth, [MsgOn])
```

```
WriteBurstVectorAsync   (TRec, iAddr, VectorOfWords, [StatusMsgOn])
WriteBurstIncrementAsync(TRec, iAddr, FirstWord, iNumFifoWords, [MsgOn])
WriteBurstRandomAsync   (TRec, iAddr, FirstWord, iNumFifoWords, [MsgOn])
WriteBurstRandomAsync   (TRec, iAddr, CoverID, iNumFifoWords, FifoWidth, [MsgOn])
```

The parameter TRec is an abbreviation for the TransactionRec parameter. The parameter MsgOn is an abbreviation for the StatusMsgOn parameter.

For WriteBurst and WriteBurstAsync, iNumFifoWords (input) specifies the number of words in the BurstFifo. For pattern based WriteBurst (Vector, Increment, Random), iNumFifoWords (input) specifies the number of words to be put into the FIFO by the pattern generator. Note that when in the mode, ADDRESS_BUS_BURST_BYTE_MODE, this will be the number of bytes in the transfer, otherwise it is the number of words in the transfer.

8.1.1.2 Interface Independent Read Transactions

```
Read(TransactionRec, iAddr, oData, [StatusMsgOn])
```

```
ReadCheck(TransactionRec, iAddr, iData, [StatusMsgOn])
```

```
ReadBurst (TransactionRec, iAddr, iNumFifoWords, [StatusMsgOn])
```

```
ReadBurstVector   (TRec, iAddr, VectorOfWords, [StatusMsgOn])
ReadBurstIncrement(TRec, iAddr, FirstWord, iNumFifoWords, [MsgOn])
ReadBurstRandom   (TRec, iAddr, FirstWord, iNumFifoWords, [MsgOn])
ReadBurstRandom   (TRec, iAddr, CoverID, iNumFifoWords, FifoWidth, [MsgOn])
```

The parameter TRec is an abbreviation for the TransactionRec parameter. The parameter MsgOn is an abbreviation for the StatusMsgOn parameter.

For GetBurst and TryGetBurst, NumFifoWords is only used as an output. For CheckBurst and TryCheckBurst, NumFifoWords is an input.

8.1.2 Interacting with the Burst FIFOs

The basic forms of WriteBurst, WriteBurstAsync and ReadBurst do not provide a means of interacting with the FIFO contents. The following FIFO fill and check patterns are implemented in FifoFillPkg_slv.vhd (in the osvvm_common library). Examples of their usage is in the Address Bus Model Independent Transaction User Guide.

8.1.2.1 Short Hand Names for the Burst FIFOs

To simplify access of the burst FIFO, the following aliases can be used.

```
-- Simplifying access to Burst FIFOs using aliases
alias WriteBurstFifo : ScoreboardIdType is ManagerRec.WriteBurstFifo ;
alias ReadBurstFifo  : ScoreboardIdType is ManagerRec.ReadBurstFifo ;
```

8.1.2.2 Filling the Burst FIFO

These are intended to be used prior to calling WriteBurst or WriteBurstAsync transactions to fill the FIFO with a set of values.

```
alias WriteBurstFifo : ScoreboardIdType is ManagerRec.WriteBurstFifo ;
. . .
Push(WriteBurstFifo, DataWord) ;
PushBurstVector   (WriteBurstFifo, VectorOfWords, FifoWidth)
PushBurstIncrement(WriteBurstFifo, FirstWord, Count, FifoWidth)
PushBurstRandom   (WriteBurstFifo, FirstWord, Count, FifoWidth)
PushBurstRandom   (WriteBurstFifo, CoverID,   Count, FifoWidth)
```

8.1.2.3 Checking the Burst FIFO

These are intended to be used after calling ReadBurst to check the contents of the FIFO.

```
alias ReadBurstFifo  : ScoreboardIdType is ManagerRec.ReadBurstFifo ;
. . .
CheckExpected(ReadBurstFifo, CheckWord) ;
CheckBurstVector   (ReadBurstFifo, VectorOfWords,    FifoWidth)
CheckBurstIncrement(ReadBurstFifo, FirstWord, Count, FifoWidth)
CheckBurstRandom   (ReadBurstFifo, FirstWord, Count, FifoWidth)
CheckBurstRandom   (ReadBurstFifo, CoverID,   Count, FifoWidth)
```

8.1.2.4 Reading the Burst FIFO

These are intended to be used after calling ReadBurst to manage the contents of the FIFO.

```
alias ReadBurstFifo  : ScoreboardIdType is ManagerRec.ReadBurstFifo ;
. . .
DataWord := Pop(ReadBurstFifo) ;
PopBurstVector(ReadBurstFifo, VectorOfWords,    FifoWidth)
```

8.1.3 Interface Specific Transactions

Interface specific transactions support split transaction interfaces by providing a transaction call for each aspect of the split transaction interface to be operated independently. This allows a WriteDataAsync transaction to be dispatched and then perhaps 4 clocks later a WriteAddressAsync transaction to be dispatched. This gives the test writer direct control of all aspects of the interface and facilitates testing of all interface characteristics.

8.1.3.1 Interface Specific Write Transactions

```
WriteAddressAsync(TransactionRec, iAddr, [StatusMsgOn])
```

```
WriteDataAsync(TransactionRec, iAddr, iData, [StatusMsgOn])
WriteDataAsync(TransactionRec, iAddr, [StatusMsgOn])
```

8.1.3.2 Interface Specific Read Transactions

```
ReadAddressAsync(TransactionRec, iAddr, [StatusMsgOn])
```

```
ReadData(TransactionRec, oData, [StatusMsgOn])
```

```
ReadCheckData(TransactionRec, iData, [StatusMsgOn])
```

```
TryReadData(TransactionRec, oData, Available, [StatusMsgOn])
```

```
TryReadCheckData(TransactionRec, iData, Available, [StatusMsgOn])
```

8.1.4 BurstMode Control Directives

BurstMode control directives configure the FIFO to be in either WORD or BYTE mode. WORD mode should match the exact size of the AXI data bus. BYTE mode is used to assemble bytes into words of a given interface. BYTE mode allows a test to be written in a fashion that is independent of the interface width.

`SetBurstMode (TransactionRec, ADDRESS_BUS_BURST_WORD_MODE) ;` `SetBurstMode (TransactionRec, ADDRESS_BUS_BURST_BYTE_MODE) ;`
`GetBurstMode (TransactionRec, OptVal)`

8.2 Transactions Supported by Axi4Subordinate

The Axi4Subordinate only supports single word transfers. Its interface is a little like a Manager Read /Write and a little like Stream Send/Get. For burst transactions, see the Axi4Memory Subordinate.

The oData and iData in Manager transactions should match the size of the transfer. For example, if the transfer is for a byte of data, then only 8 bits should be present.

8.2.1 Write Transactions

`GetWrite(TransactionRec, oAddr, oData, [StatusMsgOn])`
`TryGetWrite(TransactionRec, oAddr, oData, Available, [StatusMsgOn])`
`GetWriteAddress(TransactionRec, oAddr, [StatusMsgOn])`
`TryGetWriteAddress(TransactionRec, oAddr, Available, [StatusMsgOn])`
`GetWriteData(TransactionRec, iAddr, oData, [StatusMsgOn])`
`GetWriteData(TransactionRec, oData, [StatusMsgOn])`
`TryGetWriteData(TransactionRec, iAddr, oData, Available, [StatusMsgOn])`
`TryGetWriteData(TransactionRec, oData, Available, [StatusMsgOn])`

8.2.2 Read Transactions

`SendRead(TransactionRec, oAddr, iData, [StatusMsgOn])`
`TrySendRead(TransactionRec, oAddr, iData, Available, [StatusMsgOn])`
`GetReadAddress(TransactionRec, oAddr, [StatusMsgOn])`
`TryGetReadAddress (TransactionRec, oAddr, Available, [StatusMsgOn])`
`SendReadData(TransactionRec, iData, [StatusMsgOn])`
`SendReadDataAsync(TransactionRec, iData, [StatusMsgOn])`

8.3 Transactions Supported by Axi4Memory

The Axi4Memory VC supports the full capability of the AXI4 bus from the DUT side interface.

The transaction interface is a "backdoor" (non-functional) interface that gives the test sequencer (TestCtrl) access to the internal memory of the device. Only basic single word transactions are currently supported via this interface.

All transactions on the transaction interface (connecting to TestCtrl) complete without time passing (ie: in delta cycles). Take care not to do too many consecutive transactions as a delta cycle iteration limit in the simulator can be hit. In addition, there is no need for the VC to support Asynchronous (Async or Try) transactions. A future release may add Asynchronous transactions for symmetry to the Axi4Manager VC.

8.3.1 Write Transactions

`Write(TransactionRec, iAddr, iData, [StatusMsgOn])`

8.3.2 Read Transactions

`Read(TransactionRec, iAddr, oData, [StatusMsgOn])`
`ReadCheck(TransactionRec, iAddr, iData, [StatusMsgOn])`

8.4 AXI4 VC Shared Directives

8.4.1 General Directives

`WaitForTransaction(TransactionRec)`
`WaitForWriteTransaction(TransactionRec)`
`WaitForReadTransaction(TransactionRec)`

`WaitForClock(TransactionRec, NumberOfClocks)`

`GetTransactionCount(TransactionRec, Count)`
`GetWriteTransactionCount(TransactionRec, Count)`
`GetReadTransactionCount(TransactionRec, Count)`

`GetAlertLogID(TransactionRec, AlertLogID)`
`GetErrorCount(TransactionRec, ErrorCount)`

8.4.2 Address Bus Configuration Directives

`SetModelOptions(TransactionRec, Option, OptVal)`
`GetModelOptions(TransactionRec, Option, OptVal)`

Largely the configuration directives are used indirectly through the SetAxi4Options and GetAxi4Options directives. See Configuring the AXI4 VC. For AXI4, OptVal can have a type of boolean, integer, or std_logic_vector.

9. Configuring the AXI4 VC

The AXI4 Parameters configure the VC into a particular mode of operation or establish a default value for an interface object when it is not specified directly in the transaction. AXI4 Parameters are set using SetAxi4Options. Subordinate refers to either the Transaction Subordinate or the Memory Subordinate.

9.1 SetAxi4Options / GetAxi4Options

Model options are set using SetAxi4Options and retrieved using GetAxi4Options. These are an abstraction layer wrapped around the SetModelOptions and GetModelOptions. This allows values from the enumerated type to be used, rather than using integer constant values. These are implemented in the package Axi4OptionsPkg.vhd.

```
SetAxi4Options(TransactionRec, Option, OptVal)
GetAxi4Options(TransactionRec, Option, OptVal)
```

OptVal can be of type integer, std_logic_vector, or boolean.

9.2 Controlling Interface Signaling Characteristics and Timeouts

9.2.1 Delays for Valid and Ready

Delays for Valid and Ready can either be a static value (a fixed delay) or can be randomized.

9.2.1.1 Static Delay: Valid Delay Cycles

The xVALID _DELAY_CYCLES value specifies the amount of time to initiate a new transaction on an interface. For Burst transactions, xVALID_BURST_DELAY_CYCLES specifies the amount of time between valid within a burst data cycle.

VC	Name	Initial Value in VC
Manager	WRITE_ADDRESS_VALID_DELAY_CYCLES	0
Manager	WRITE_DATA_VALID_DELAY_CYCLES	0
Manager	WRITE_DATA_VALID_BURST_DELAY_CYCLES	0
Subordinate	WRITE_RESPONSE_VALID_DELAY_CYCLES	0
Manager	READ_ADDRESS_VALID_DELAY_CYCLES	0
Subordinate	READ_DATA_VALID_DELAY_CYCLES	0
Memory Subordinate	READ_DATA_VALID_BURST_DELAY_CYCLES	0

9.2.1.2 **Random Delay: TValid Delay Cycles**

Random delays for TValid are controlled by an array of Delay Coverage Models whose subtype is AxiDelayCoverageIdArrayType. For more information on delay coverage models, see DelayCoveragePkg_user_guide.pdf.

The delay coverage array type is indexed by the constants defined in the following table.

VC	Index into DelayCoverageIDArrayType
Manager	WRITE_ADDRESS_ID
Manager	WRITE_DATA_ID
Subordinate	WRITE_RESPONSE_ID
Manager	READ_ADDRESS_ID
Subordinate	READ_DATA_ID

To use random delays in the verification component, call SetUseRandomDelays. At this point the TValid delays will be randomized using the default coverage model.

```
SetUseRandomDelays(ManagerRec, TRUE) ;
```

The coverage model can be retrieved using GetDelayCoverageID. After getting the DelayCovID, it can be indexed using the constants in the table to access an individual delay coverage model. For example, DelayCovID(WRITE_ADDRESS_ID) accesses the delay coverage model for the manager write address interface.

```
variable DelayCovID  : AxiDelayCoverageIdArrayType ;
. . .
GetDelayCoverageID(ManagerRec, DelayCovID) ;
```

The delay coverage model can be changed using SetDelayCoverageID.

```
SetDelayCoverageID(ManagerRec, NewDelayCoverageID) ;
```

Axi4Manager sets the default delays to the following.

```
-- Write Address
    AddBins (WriteAddressDelayCov.BurstLengthCov,  GenBin(2,10,1)) ;
    AddBins (WriteAddressDelayCov.BeatDelayCov,    GenBin(0)) ;
    AddBins (WriteAddressDelayCov.BurstDelayCov,   GenBin(2,5,1)) ;
-- Write Data
    AddBins (WriteDataDelayCov.BurstLengthCov,  GenBin(2,10,1)) ;
    AddBins (WriteDataDelayCov.BurstDelayCov,   GenBin(2,5,1)) ;
    AddBins (WriteDataDelayCov.BeatDelayCov,    GenBin(0)) ;
-- Read Address
    AddBins (ReadAddressDelayCov.BurstLengthCov,  GenBin(2,10,1)) ;
    AddBins (ReadAddressDelayCov.BurstDelayCov,   GenBin(2,5,1)) ;
    AddBins (ReadAddressDelayCov.BeatDelayCov,    GenBin(0)) ;
```

Axi4Subordinate and AxiMemory sets the default delays to the following.

```
-- Write Response
    AddBins (WriteResponseDelayCov.BurstLengthCov,  GenBin(2,10,1)) ;
    AddBins (WriteResponseDelayCov.BurstDelayCov,   GenBin(2,5,1)) ;
    AddBins (WriteResponseDelayCov.BeatDelayCov,    GenBin(0)) ;
-- Read Data
    AddBins (ReadDataDelayCov.BurstLengthCov,  GenBin(2,10,1)) ;
    AddBins (ReadDataDelayCov.BurstDelayCov,   GenBin(2,5,1)) ;
    AddBins (ReadDataDelayCov.BeatDelayCov,    GenBin(0)) ;
```

9.2.1.3 **Static Delay: Ready Before Valid**

When xREADY_BEFORE_VALID is TRUE then the interface may assert xREADY before xVALID is received.

VC	Name	Initial Value in VC
Subordinate	WRITE_ADDRESS_READY_BEFORE_VALID	TRUE
Subordinate	WRITE_DATA_READY_BEFORE_VALID	TRUE
Manager	WRITE_RESPONSE_READY_BEFORE_VALID	TRUE
Subordinate	READ_ADDRESS_READY_BEFORE_VALID	TRUE
Manager	READ_DATA_READY_BEFORE_VALID	TRUE

9.2.1.4 **Static Delay: Ready Delay Cycles**

When xx_READY_BEFORE_VALID is TRUE, then xx_READY_DELAY_CYCLES is a relative to when the last transfer completed. Figure 25 shows xx_READY_DELAY_CYCLES = 2 when xx_READY_BEFORE_VALID is TRUE.

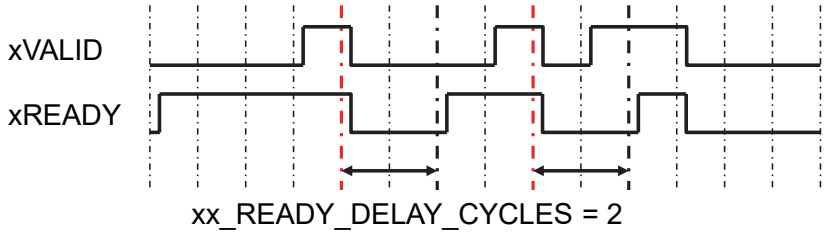

Figure 25. xx_READY_DELAY_CYCLES = 2 when xx_READY_BEFORE_VALID is TRUE

When xx_READY_BEFORE_VALID is FALSE, then xx_READY_DELAY_CYCLES is a relative to when xValid is asserted. Figure 26 shows xx_READY_DELAY_CYCLES = 2 when xx_READY_BEFORE_VALID is FALSE.

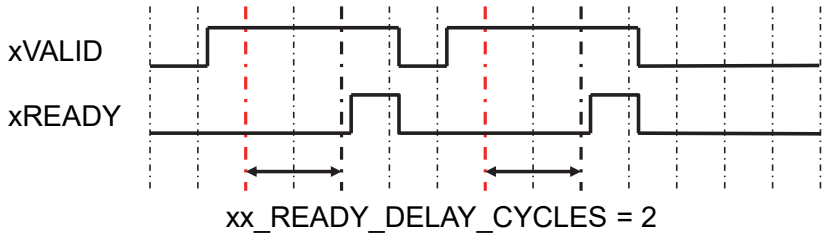

Figure 26. xx_READY_DELAY_CYCLES = 2 when xx_READY_BEFORE_VALID is FALSE

VC	Name	Initial Value in VC
Subordinate	WRITE_ADDRESS_READY_DELAY_CYCLES	0
Subordinate	WRITE_DATA_READY_DELAY_CYCLES	0
Manager	WRITE_RESPONSE_READY_DELAY_CYCLES	0
Subordinate	READ_ADDRESS_READY_DELAY_CYCLES	0
Manager	READ_DATA_READY_DELAY_CYCLES	0

9.2.1.5 **Random Delay: TReady Delay Cycles**

Random delays for TReady and random settings for RECEIVE_READY_BEFORE_VALID are controlled an array of Delay Coverage Models whose subtype is AxiDelayCoverageIdArrayType. For more information on delay coverage models, see DelayCoveragePkg_user_guide.pdf.

The delay coverage array type is indexed by the constants defined in the following table.

VC	Index into DelayCoverageIDArrayType
Manager	WRITE_ADDRESS_ID
Manager	WRITE_DATA_ID
Subordinate	WRITE_RESPONSE_ID
Manager	READ_ADDRESS_ID
Subordinate	READ_DATA_ID

To use random delays in the verification component, call SetUseRandomDelays. At this point TReady delays and RECEIVE_READY_BEFORE_VALID settings will be randomized.

```
SetUseRandomDelays(ManagerRec, TRUE) ;
```

The coverage model can be retrieved using GetDelayCoverageID. After getting the DelayCovID, it can be indexed using the constants in the table to access an individual delay coverage model. For example, DelayCovID(WRITE_RESPONSE_ID) accesses the delay coverage model for the manager write response interface.

```
variable DelayCovID  : AxiDelayCoverageIdArrayType ;
. . .
GetDelayCoverageID(ManagerRec, DelayCovID) ;
```

The delay coverage model can be changed using SetDelayCoverageID.

```
SetDelayCoverageID(ManagerRec, NewDelayCoverageID) ;
```

Axi4Manager sets the default delays to the following.

```
-- Write Response
    AddBins (WriteResponseDelayCov.BurstLengthCov,  GenBin(2,10,1)) ;
    AddCross(WriteResponseDelayCov.BurstDelayCov,   GenBin(0,1,1), GenBin(2,5,1)) ;
    AddCross(WriteResponseDelayCov.BeatDelayCov,    GenBin(0), GenBin(0)) ;
-- Read Data
    AddBins (ReadDataDelayCov.BurstLengthCov,  GenBin(2,10,1)) ;
    AddCross(ReadDataDelayCov.BurstDelayCov,   GenBin(0,1,1), GenBin(2,5,1)) ;
    AddCross(ReadDataDelayCov.BeatDelayCov,    GenBin(0),     GenBin(0)) ;
```

Axi4Subordinate and AxiMemory sets the default delays to the following.

```
-- Write Address
    AddBins (WriteAddressDelayCov.BurstLengthCov,  GenBin(2,10,1)) ;
    AddCross(WriteAddressDelayCov.BurstDelayCov,   GenBin(0,1,1), GenBin(2,5,1)) ;
    AddCross(WriteAddressDelayCov.BeatDelayCov,    GenBin(0),     GenBin(0)) ;
-- Write Data
    AddBins (WriteDataDelayCov.BurstLengthCov,  GenBin(2,10,1)) ;
    AddCross(WriteDataDelayCov.BurstDelayCov,   GenBin(0,1,1), GenBin(2,5,1)) ;
    AddCross(WriteDataDelayCov.BeatDelayCov,    GenBin(0),     GenBin(0)) ;
-- Read Address
    AddBins (ReadAddressDelayCov.BurstLengthCov,  GenBin(2,10,1)) ;
    AddCross(ReadAddressDelayCov.BurstDelayCov,   GenBin(0,1,1), GenBin(2,5,1)) ;
    AddCross(ReadAddressDelayCov.BeatDelayCov,    GenBin(0),     GenBin(0)) ;
```

9.2.2 **Timeout: No xReady in response xValid**

On the AXI4 interface, it is expected that after a xValid is asserted, an xReady will be asserted within a reasonable amount of time. The xx_READY_TIME_OUT specifies the number of clocks that xValid is asserted without xReady before a FAILURE is generated. The xx_READY_TIME_OUT value is an integer.

VC	Name	Initial Value in VC
Manager	WRITE_ADDRESS_READY_TIME_OUT	25
Manager	WRITE_DATA_READY_TIME_OUT	25
Subordinate	WRITE_RESPONSE_READY_TIME_OUT	25
Manager	READ_ADDRESS_READY_TIME_OUT	25
Responder	READ_DATA_READY_TIME_OUT	25

9.2.3 Timeout: No xValid for Write Response or Read Data to complete cycles

On the AXI4 interface, after a Write Address cycle and its corresponding Write Data cycle(s) have completed, it is expected that the Subordinate will initiate a Write Response cycle (signaled with BVALID) within a reasonable amount of time. Likewise, after a Read Address cycle is completed, it is expected that the Subordinate will initiate a Read Data cycle (signaled with RVALID) within a reasonable amount of time. The xx_VALID_TIME_OUT specifies the number of clocks within which the Subordinate must asssert xValid before a FAILURE is generated. xx_VALID_TIME_OUT value is an integer.

VC	Name	Initial Value in VC
Manager	WRITE_RESPONSE_VALID_TIME_OUT	25
Manager	READ_DATA_VALID_TIME_OUT	25

9.2.4 Setting Interface Signaling Characteristics and Timeouts

Figure 27 shows setting the Subordinate's Write Address interface so that AWREADY occurs two cycles after AWVALID is asserted.

```
SetAxi4Options(SubordinateRec, WRITE_ADDRESS_READY_BEFORE_VALID,   FALSE) ;
SetAxi4Options(SubordinateRec, WRITE_ADDRESS_READY_DELAY_CYCLES,   2) ;
```

Figure 27. Setting the Subordinate's Write Address Interface AWREADY Parameters

9.3 AXI4 Interface Default Values

For interface objects, a value set in the VC that drives the interface, establishes the default value to be driven if it is not specified by the transaction. Similarly, a value set in the VC that does not drive the interface, establishes a default expected value used for checking when checking is done for that item and it is not specified by the transaction.

9.3.1 Write Address Default Values

Signal	Driver	Configurable	AXI	Description
AWADDR	Manager	No	All	Write Address
AWVALID	Manager	No	All	Set when address channels has valid values
AWREADY	Subordinate	No	All	Set when Subordinate ready to accept values
AWPROT	Manager	Yes	All	Initial value = 0 Privilege level. Frequency of usage?
AWID	Manager	Yes	Full	Write Address ID. Initial value = 0. Used in reordering operations.
AWLEN	Manager	No	Full	Number of transfers in a burst. F(#Bytes, Word Width, Starting Address)

AWSIZE	Manager	Yes	Full	Set to log2(Data Bytes) = Full width of interface.
AWBURST	Manager	Yes	Full	Initial value = INCR Options: Fixed, Incr, Wrap.
AWLOCK	Manager	Yes	AXI3	Initial value = 0.
AWCACHE	Manager	Yes	Full	Initial value = 0. Cache Memory Access Types.
AWQOS	Manager	Yes	Full	Initial value = 0. Quality of Service.
AWREGION	Manager	Yes	Full	Initial value = 0. Partitions Subordinates into separate areas.
AWUSER	Manager	Yes	Full	Initial value = 0

9.3.2 **Write Data Default Values**

Signal	Driver	Configurable	AXI	Description
WDATA	Manager	No	All	Write Address
WSTRB	Manager	No	All	Currently only supports contiguous bursts. F(#Bytes, Starting Address) Upgrade to support Metavalue = 0
WVALID	Manager	No	All	Set when channel has valid values
WREADY	Subordinate	No	All	Set when Subordinate ready to accept values
WLAST	Manager	No	Full	Asserted on last data value of transaction. Single word – always asserted Burst – last word
WID	Manager	Yes	AXI3	Write Data ID. Initial value = 0. Used in reordering operations.
WUSER	Manager	Yes	Full	Initial value = 0.

9.3.3 **Write Response Default Values**

Signal	Driver	Configurable	AXI	Description
BRESP	Subordinate	Yes	All	Values: AXI4_RESP_OKAY (Initial) AXI4_RESP_EXOKAY AXI4_RESP_SLVERR AXI4_RESP_DECERR

BVALID	Subordinate	No	All	Set when Subordinate has valid values
BREADY	Manager	No	All	Set when Manager ready to accept values
BID	Subordinate	No	Full	Axi4Memory: Matches Address Write Axi4Subordinate: Set by configuration = 0
BUSER	Subordinate	No	Full	Axi4Memory: Matches Address Write Axi4Subordinate: Set by configuration = 0

9.3.4 **Read Address Default Values**

Signal	Driver	Configurable	AXI	Description
ARADDR	Manager	No	All	Read Address
ARVALID	Manager	No	All	Set when address channels has valid values
ARREADY	Subordinate	No	All	Set when Subordinate ready to accept values
ARPROT	Manager	Yes	All	Initial value = 0 Privilege level. Frequency of usage?
ARID	Manager	Yes	Full	Read Address ID. Initial value = 0. Used in reordering operations.
ARLEN	Manager	No	Full	Number of transfers in a burst. F(#Bytes, Word Width, Starting Address)
ARSIZE	Manager	Yes	Full	Set to log2(Data Bytes) = Full width of interface.
ARBURST	Manager	Yes	Full	Initial value = INCR Options: Fixed, Incr, Wrap.
ARLOCK	Manager	Yes	AXI3	Initial value = 0.
ARCACHE	Manager	Yes	Full	Initial value = 0. Cache Memory Access Types.
ARQOS	Manager	Yes	Full	Initial value = 0. Quality of Service.
ARREGION	Manager	Yes	Full	Initial value = 0. Partitions Subordinates into separate areas.
ARUSER	Manager	Yes	Full	Initial value = 0

9.3.5 **Read Data Default Values**

Signal	Driver	Configurable	AXI	Description
RDATA	Subordinate	No	All	Write Address
RRESP	Subordinate	Yes	All	Initial = OKAY
RVALID	Subordinate	No	All	Set when channel has valid values
RREADY	Manager	No	All	Set when Subordinate ready to accept values
RID	Subordinate	Yes	AXI3	Axi4Memory: Matches Address Read Axi4Subordinate: Set by configuration = 0
RUSER	Subordinate	Yes	Full	Axi4Memory: Matches Address Read Axi4Subordinate: Set by configuration = 0
RLAST	Subordinate	No	Full	Asserted on last data value of transaction. Single word – always asserted Burst – last word

9.3.6 **Setting Interface Default Values**

All AXI interface values are std_logic_vector and must match the size of the corresponding interface object. Figure 28 shows setting the AWPROT and AWUSER values.

```
SetAxi4Options(TransactionRec, AWPROT,   "010") ;
SetAxi4Options(TransactionRec, AWUSER,   X"02") ;
```

Figure 28. Setting values for AWPROT and AWUSER

9.3.7 **Setting BRESP and RRESP**

By default, the expected value for BRESP and RRESP are OKAY. When testing a subordinates response to an incorrect address, a SLVERR is expected. AXI4 models support the enumeration values OKAY, EXOKAY, SLVERR, and DECERR of type Axi4RespEnumType. Figure 29 shows how to set BRESP and RRESP to expect a SLVERR for a single transfer.

```
-- Write transfer that expects a SLVERR
SetAxi4Options(ManagerRec, BRESP,     SLVERR) ;
Write(ManagerRec, X"0002_0000", X"0002_0101" ) ;
SetAxi4Options(ManagerRec, BRESP,     OKAY) ;

-- Read transfer that expects a SLVERR
SetAxi4Options(ManagerRec, RRESP,     SLVERR) ;
Read(ManagerRec,    X"0004_000C", Data) ;
AffirmIfEqual(Data, X"0004_0404", "Manager Read Data: ") ;
SetAxi4Options(ManagerRec, RRESP,     OKAY) ;
```

Figure 29. Setting values for BRESP and RRESP to expect a SLVERR.

9.3.8 Setting WSTRB

WSTRB indicates which bytes are active in a transfer. It is a function of the number of bytes in a transfer and the starting address.

The current implementation of the AXI4 VC requires that Data bytes in a burst transfer are contiguous. Hence, only the starting and ending words in the transfer will not be all '1'.

10. About the OSVVM AXI4 VCs

The OSVVM AXI4 VCs were developed and is maintained by Jim Lewis of SynthWorks VHDL Training. It evolved from methodology and packages developed for SynthWorks' VHDL Testbenches and verification class. It is part of the Open Source VHDL Verification Methodology (OSVVM) model library, which brings leading edge verification techniques to the VHDL community.

Please support OSVVM by purchasing your VHDL training from SynthWorks.

11. About the Author - Jim Lewis

Jim Lewis, the founder of SynthWorks, has thirty plus years of design, teaching, and problem solving experience. In addition to working as a Principal Trainer for SynthWorks, Mr Lewis has done ASIC and FPGA design, custom model development, and consulting.

Mr. Lewis is chair of the IEEE 1076 VHDL Working Group (VASG) and is the primary developer of the Open Source VHDL Verification Methodology (OSVVM.org) packages. Neither of these activities generate revenue. Please support our volunteer efforts by buying your VHDL training from SynthWorks.

If you find bugs these packages or would like to request enhancements, you can reach me at jim@synthworks.com.

12. References

[1] Jim Lewis, VHDL Testbenches and Verification, student manual for SynthWorks' class.