OSVVM Model Library:
AXI4
Verification Components

User Guide for Release 2021.06

By

Jim Lewis

SynthWorks VHDL Training

Jim@SynthWorks.com

http://www.SynthWorks.com

Table of Contents

1 Overview 4
2 OSVVM Testbench Architecture 4
2.1 Test Architecture Overview 4
2.2 Writing Tests 5
2.3 Address Bus Transaction Interface 6
2.4 AxiBus : Axi4RecType 7
2.5 Axi4 Context Declaration 10
2.6 Component Declarations for Axi4 Verification Components 10
3 Demo Preparation: Getting and Building the OSVVM Libraries 11
4 TbAxi4: "Connected" Transaction Interface 12
4.1 Demo: Running the AXI4 Testbenches 12
4.2 TbAxi4: Axi4 Test Environment - Connected Transaction Interface 13
4.3 TestCtrl Entity – Connected Transaction Interface (Record Port) 14
4.4 Axi4Master Entity Interface – Connected Transaction Interface (Record Port) 15
4.5 Axi4Responder Entity Interface – Connected Transaction Interface (Record Port) 15
4.6 Axi4Memory Entity Interface – Connected Transaction Interface (Record Port) 17
5 TbAxi4: AXI4 - Virtual Transaction Interface 18
5.1 Demo: Running the AXI4 Testbenches 18
5.2 TbAxi4: Axi4 Test Environment – Virtual Transaction Interface 19
5.3 TestCtrl Entity – Virtual Transaction Interface (External Names) 20
5.4 Axi4MasterVti Entity Interface – Virtual Transaction Interface (External Names) .. 21
5.5 Axi4ResponderVti Entity Interface – Virtual Transaction Interface (External Names) 22
5.6 Axi4MemoryVti Entity Interface – Virtual Transaction Interface (External Names) 23
6 Writing Tests Using the Axi4 VCs 24
6.1 Test Initialization 25
6.2 A Simple Directed Test 25
6.3 Test Finalization 27
6.4 Test Wide Reporting 27
7 AXI4 VC Transactions 28
7.1 AXI4 VC Shared Directives 28
7.1.1 General Directives 28
7.1.2 Configuration Directives 28
7.2 AXI Master Transactions 28
7.2.1 BurstMode Control Directives 28
7.2.2 Write Transactions 29
7.2.3 Read Transactions 29
7.3 AXI4 Responder Transactions 30

7.3.1 Write Transactions 30
7.3.2 Read Transactions 30
7.4 AXI4 Memory Responder Transactions 30
7.4.1 Write Transactions 31
7.4.2 Read Transactions 31
8 Setting AXI4 Parameters 31
8.1 SetAxi4Options / GetAxi4Options 31
8.2 Controlling Interface Signaling Characteristics and Timeouts 31
8.2.1 Delay: Valid Delay Cycles 31
8.2.2 Control: Ready Before Valid 32
8.2.3 Delay: Ready Delay Cycles 32
8.2.4 Timeout: No xReady in response xValid 33
8.2.5 Timeout: No xValid for Write Response or Read Data to complete cycles 33
8.2.6 Setting Interface Signaling Characteristics and Timeouts 34
8.3 AXI4 Interface Default Values 34
8.3.1 Write Address Default Values 34
8.3.2 Write Data Default Values 35
8.3.3 Write Response Default Values 35
8.3.4 Read Address Default Values 35
8.3.5 Read Data Default Values 36
8.3.6 Setting Interface Default Values 37
8.3.7 Setting WSTRB 37
9 Burst Transactions 37
9.1 Run the Demo 37
9.2 Burst FIFOs are in the Interface 37
9.3 Interacting with the Burst FIFOs in the Test Sequencer 38
9.3.1 Accessing Burst FIFOs in a Test Case 38
9.3.2 Filling the Burst FIFO 38
9.3.3 Reading and/or Checking the Burst FIFO 38
9.3.4 Reading and Writing Bursts with the Axi4Master 38
9.4 Interacting with the Burst FIFOs in the Axi4Master VC 39
9.4.1 Accessing the BurstFifo in Axi4Master VC 39
10 About the OSVVM AXI4 VCs 39
11 About the Author - Jim Lewis 40
12 References 40

1 Overview

The OSVVM AXI4 Verification Components (VCs) facilitate testing the interface and functionality of AXI4 devices. The OSVVM AXI4 VCs include Axi4Master, Axi4Responder, and Axi4Memory. These VCs are intended to be part of a structured test environment.

The AXI4 Master verification component implements the complete AXI4 Master 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.

With respect to AXI4 terminology, we use the term Responder instead of Slave. The Axi4Responder is a transaction-based responder. The Axi4Memory is a memory responder.

The Axi4Memory verification component responds to AXI4 accesses by either writing to memory or reading from memory. 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 using OSVVM's MemoryPkg. Internally MemoryPkg implements a sparse memory data structure using a singleton which creates a dynamic array of memory models. Each memory model uses an array of pointers to storage locations (approximately 1024). Memory only gets allocated when a block is written. Hence, a test only used the locations it needs.

The Axi4Responder verification component is a transaction-based component which allows the test sequencer to program an arbitrary sequence of responses. Currently the Axi4Responder 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.

2 OSVVM Testbench Architecture

2.1 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 (Axi4Master, Axi4Responder, 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).

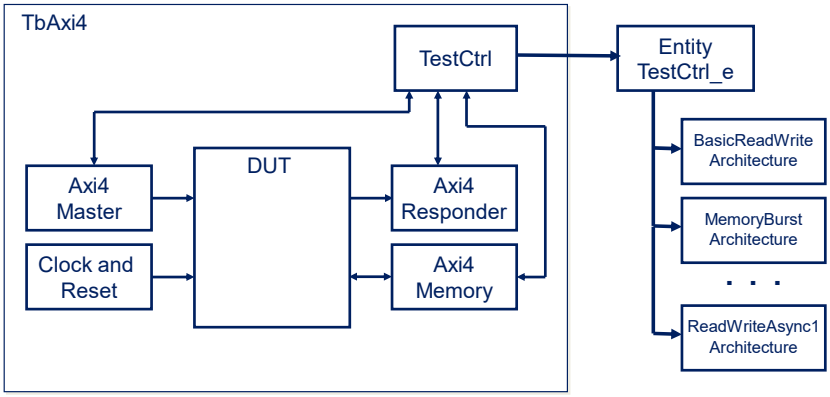

Figure 1. OSVVM Testbench Framework

2.2 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 Axi4Master 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.

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

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

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

Figure 2. Calls to Axi4Master Write, Read, and ReadCheck Transactions

2.3 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               : bit_max ;
    Ack               : bit_max ;
    -- 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 ;
    BoolToModel       : boolean_max ;
    IntFromModel      : integer_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 MasterRec (which connects the Axi4Master to TestCtrl) and ResponderRec (which connects the Axi4Responder to TestCtrl).

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

Figure 4. MasterRec and ResponderRec

2.4 AxiBus : Axi4RecType

AxiBus is defined as a record of type Axi4RecType, shown in Figure 5 is defined in the Address Bus Model Independent Transaction package, AddressBusTransactionPkg.vhd, which is in the directory OsvvmLbraries/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 master, 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 master, 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

2.5 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 ;    -- Connected 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

2.6 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.

3 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".

4 TbAxi4: "Connected" Transaction Interface

In the OSVVM Connected Transaction 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 Connected Transaction 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 Connected Transaction Interfaces.

4.1 Demo: Running the AXI4 Testbenches

The AXI4, Axi4Lite, AxiStream, and UART verification components all come with testbenches and the process to run them is similar to what is discussed here for AXI4.

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/testbench.pro
```

Figure 10. Compiling and Running OSVVM

4.2 TbAxi4: Axi4 Test Environment - Connected Transaction 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.

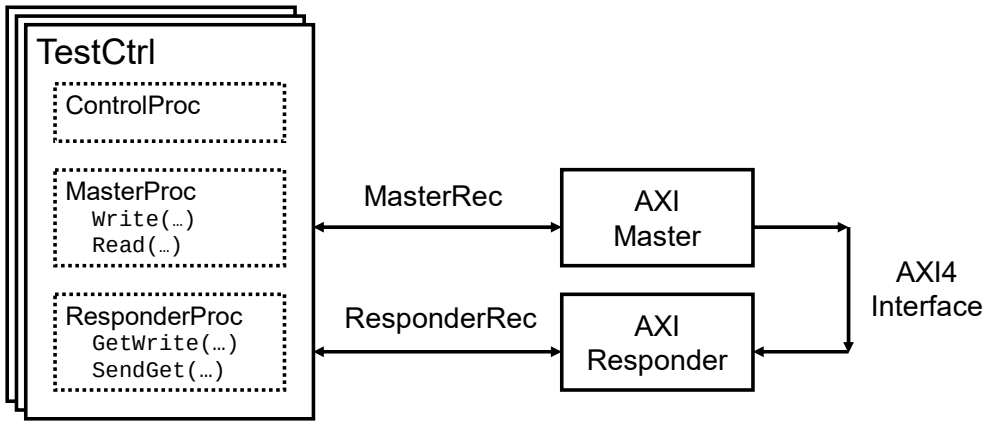

Figure 11. TbAxi4

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 MasterRec, ResponderRec  : 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, . . .) ;
  Responder_1 : Axi4Responder(…, AxiBus, ResponderRec) ;
  Master_1 :    Axi4Master   (…, AxiBus, MasterRec);
  TestCtrl_1 :  TestCtrl     (nReset, MasterRec, ResponderRec) ;
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 Master_1 and Responder_1. Our intent is to reuse some of the same test cases with other Master and Responder verification components (such as Avalon and Wishbone) – and hence the more generic naming.

4.3 TestCtrl Entity – Connected Transaction Interface (Record Port)

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 (MasterRec and ResponderRec). These records connect to the Axi4Master and Axi4Responder/Axi4Memory components.

```
library OSVVM_AXI4 ;
  context OSVVM_AXI4.Axi4Context ;
entity TestCtrl is
  port (
    -- Global Signal Interface
    nReset          : in    std_logic ;

    -- Transaction Interfaces
    MasterRec       : inout AddressBusRecType ;
    ResponderRec    : inout AddressBusRecType
  ) ;

  -- Derive AXI interface properties from the interface
  constant AXI_ADDR_WIDTH : integer := MasterRec.Address'length ;
  constant AXI_DATA_WIDTH : integer := MasterRec.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 MasterRec.WriteBurstFifo ;
  alias ReadBurstFifo  : ScoreboardIdType is MasterRec.ReadBurstFifo ;

end entity TestCtrl ;
```

Figure 13. TestCtrl_e.vhd

4.4 Axi4Master Entity Interface – Connected Transaction Interface (Record Port)

The Axi4Master entity interface is shown in Figure 14. It uses records for both the AXI4 (AxiBus) and transaction interfaces (TransRec). The AxiBus implements all signals in the AXI interface.

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 Axi4Master.vhd for the details of the generics.

```
entity Axi4Master 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 Master 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 Axi4Master ;
```

Figure 14. Axi4Master

4.5 Axi4Responder Entity Interface – Connected Transaction Interface (Record Port)

The Axi4Responder 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.

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 Axi4Master.vhd for the details of the generics.

```
entity Axi4Responder 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 Master 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 Axi4Responder ;
```

Figure 15. Axi4Responder

4.6 Axi4Memory Entity Interface – Connected Transaction Interface (Record Port)

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.

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 Axi4Master.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 Master 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

4.6.1 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.

5 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 Connected Transaction Interfaces.

5.1 Demo: Running the AXI4 Testbenches

The AXI4, Axi4Lite, AxiStream, and UART verification components all come with OSVVM testbenches and the process to run them is similar to what is discussed here for AXI4.

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/testbenchVTI
```

Figure 17. Compiling and Running OSVVM

5.2 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.

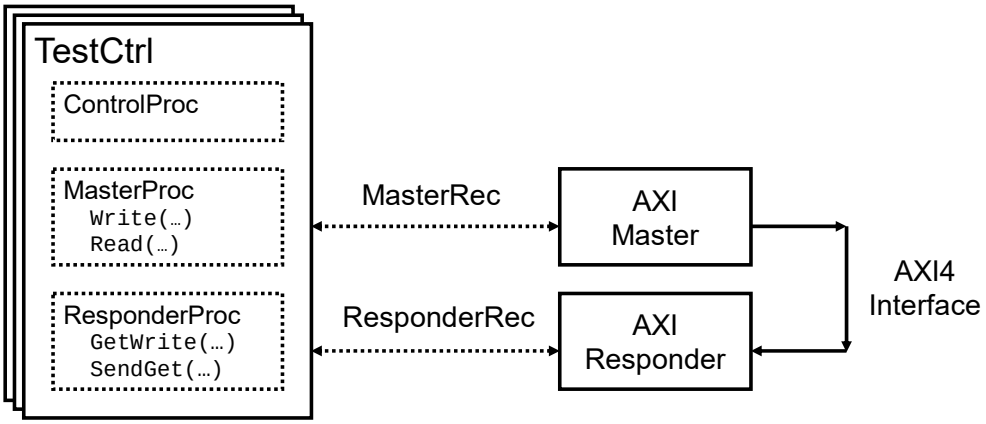

Figure 18. TbAxi4 for Virtual Transaction Interfaces

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 (Axi4Master and Axi4Responder/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, . . .) ;
  Responder_1 : Axi4Responder(…, AxiBus) ;
  Master_1 :    Axi4Master   (…, AxiBus);
  TestCtrl_1 :  TestCtrl     (nReset) ;
end TestHarness ;
```

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

5.3 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 OSVVM_AXI4 ;
  context OSVVM_AXI4.Axi4Context ;
entity TestCtrl is
  port (
    -- Global Signal Interface
    nReset          : in    std_logic
  ) ;

  -- Connect transaction interfaces using external names
  alias MasterRec is <<signal ^.Master_1.TransRec : AddressBusRecType>>;
  alias ResponderRec is <<signal ^.Responder_1.TransRec :
      AddressBusRecType>>;

  -- Derive AXI interface properties from the MasterRec
  constant AXI_ADDR_WIDTH : integer := MasterRec.Address'length ;
  constant AXI_DATA_WIDTH : integer := MasterRec.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 MasterRec.WriteBurstFifo ;
  alias ReadBurstFifo  : ScoreboardIdType is MasterRec.ReadBurstFifo ;

end entity TestCtrl ;
```

Figure 20. TestCtrl_e.vhd

5.4 Axi4MasterVti Entity Interface – Virtual Transaction Interface (External Names)

The Axi4MasterVti 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 Axi4MasterVti.vhd for the details of the generics.

```
entity Axi4MasterVti 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 Master 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 Axi4MasterVti ;
```

Figure 21. Axi4Master

5.5 Axi4ResponderVti Entity Interface – Virtual Transaction Interface (External Names)

The Axi4ResponderVti 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 Axi4ResponderVti.vhd for the details of the generics.

```
entity Axi4ResponderVti 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 Master 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 Axi4ResponderVti ;
```

Figure 22. Axi4Responder

5.6 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).

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 Master 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 Axi4MemoryVti ;
```

Figure 23. Axi4MemoryVti

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 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.

Since the processes are independent of each other, synchronization is required to create coordinated events on the different interfaces. This is accomplished by using synchronization primitives, such as WaitForBarrier (from TbUtilPkg in the OSVVM library).

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

  MasterProc : process
  begin
    WaitForClock(MasterRec, 2) ;
    . . .
    Write(MasterRec, X"0000_0000", X"5555_5555" ) ;
    Read (MasterRec, X"0000_0000", Data) ;
    AffirmIfEqual(Data, X"5555_5555", "Master Read Data: ") ;
    . . .
    WaitForBarrier(MasterDone) ;
    WaitForBarrier(TestDone) ;
  end process MasterProc;

  ResponderProc : process
  begin
    WaitForBarrier(MasterDone) ;
```

```
    -- Backdoor transaction access to Axi4Memory.vhd
    ReadCheck(ResponderRec, X"0000_0000", X"5555_5555" ) ;
    . . .
    WaitForBarrier(TestDone) ;
  end process ReceiverProc ;
end MemoryReadWrite1 ;
```

Figure 24. TestCtrl Architecture

6.1 Test Initialization

The ControlProc both initializes and finalizes a test. Test initialization is shown in Figure 25. This is based on the code in TbAxi4_MemoryReadWrite1.vhd. SetAlertLogName sets the test name. Each verification component calls GetAlertLogID to allocate an ID that allows it to accumulate errors separately within the AlertLog data structure. Calling GetAlertLogID here with the same name used by the component instance (Master_1) returns the same ID as in the verification component and allows its message filtering to be controlled directly from the testbench (via the calls to SetLogEnable). WaitForBarrier stops ControlProc until the test is complete. The value 35 ms is a watch dog timer that is set over the entire test case. See the finalization discussion for details.

```
ControlProc : process
begin
  SetAlertLogName("TbAxi4_MemoryReadWrite1");
  TBID <= GetAlertLogID("TB");
  MasterID <= GetAlertLogID("Master_1");
  SetLogEnable(PASSED, TRUE) ;
  SetLogEnable(MasterID, INFO, TRUE) ;

  -- Wait for simulation elaboration/initialization
  wait for 0 ns ;  wait for 0 ns ;
  TranscriptOpen("./results/TbAxi4_MemoryReadWrite1.txt") ;
  SetTranscriptMirror(TRUE) ;

  -- Wait for Design Reset
  wait until nReset = '1' ;
  ClearAlerts ;
  WaitForBarrier(TestDone, 35 ms) ;
  . . .
```

Figure 25. Test Initialization

6.2 A Simple Directed Test

In the TbAxi4_MemoryReadWrite1.vhd snippet shown in Figure 26, in the MasterProc, the Axi4Master writes four bytes and checks them Using the WaitForBarrier(MasterDone) as a blocking point, the ResponderProc waits until all transactions from the Axi4Master have completed and then the Axi4Memory VC "double" checks that the memory still has these 4 values using its backdoor transaction interface to the memory.

```
architecture MemoryReadWrite1 of TestCtrl is
```

```
  . . .
begin
  . . .
  MasterProc : process
  begin
    WaitForClock(MasterRec, 2) ;
    . . .
    log("Write and Read with 1 Byte, and ByteAddr = 0, 1, 2, 3") ;
    Write(MasterRec, X"0000_0010", X"11" ) ;
    Write(MasterRec, X"0000_0011", X"22" ) ;
    Write(MasterRec, X"0000_0012", X"33" ) ;
    Write(MasterRec, X"0000_0013", X"44" ) ;

    ReadCheck(MasterRec, X"0000_0010", X"11" ) ;
    ReadCheck(MasterRec, X"0000_0011", X"22" ) ;
    ReadCheck(MasterRec, X"0000_0012", X"33" ) ;
    ReadCheck(MasterRec, X"0000_0013", X"44" ) ;
    . . .
    WaitForBarrier(MasterDone) ;
    WaitForBarrier(TestDone) ;
  end process MasterProc;

  ResponderProc : process
  begin
    WaitForBarrier(MasterDone) ;
    -- Backdoor transaction access to Axi4Memory.vhd
    . . .
    ReadCheck(ResponderRec, X"0000_0010", X"11" ) ;
    ReadCheck(ResponderRec, X"0000_0011", X"22" ) ;
    ReadCheck(ResponderRec, X"0000_0012", X"33" ) ;
    ReadCheck(ResponderRec, X"0000_0013", X"44" ) ;
    . . .
    WaitForBarrier(TestDone) ;
  end process ReceiverProc ;
end MemoryReadWrite1 ;
```

Figure 26. A Simple Directed Test

The ReadCheck transaction checks the received value against the supplied expected value. It produces a log "PASSED" message if they are equal and alert "ERROR" message otherwise. A PASSED message is shown in Figure 27. "Master_1" is produced in the message since it matches the string that the verification component used to create its ModelID – see section Axi4 Verification Components for a discussion of how this happens.

```
%% Log   PASSED  in Master_1: Data Check, Read Data: 00000011  Read Address:
00000010  Prot: 0 at 200 ns
```

Figure 27. Messaging from ReadCheck

6.3 Test Finalization

Test finalization runs after the "WaitForBarrier(TestDone, 35 ms)" resumes. This occurs when either TestDone is signaled (normal completion) or in this case when the 35 ms timeout occurs. Representative code is shown in Figure 28. The first AlertIf logs a test error if the test finished due to timeout. The second AlertIf logs a test error if the test did not do any self-checking (reporting PASSED in this case would be misleading). Then it prints the a summary of the test results using ReportAlerts. See Test Wide Reporting for more details on ReportAlerts.

```
ControlProc : process
begin
  . . .
  -- Wait for test to finish
  WaitForBarrier(TestDone, 35 ms) ;
  AlertIf(now >= 35 ms, "Test finished due to timeout") ;
  AlertIf(GetAffirmCount < 1, "Test is not Self-Checking");

  TranscriptClose ;
--    AlertIfDiff("./results/…", "…", "") ;

  print("") ;
  -- Expecting two check errors at 128 and 256
  ReportAlerts(ExternalErrors => (0, -2, 0)) ;
  print("") ;
  std.env.stop ;
  wait ;
end process ControlProc ;
```

Figure 28. Test Finalization

6.4 Test Wide Reporting

The AlertLog data structure tracks FAILURE, ERROR, WARNING, and PASSED for the entire test as well as for each AlertLogID (see GetAlertLogID). Each OSVVM VC uses GetAlertLogID to allocate one or more IDs to report against. ReportAlerts prints a test completion message using this information. Figure 29 shows a representative PASSED and FAILED message that will be printed.

```
%% DONE  PASSED  TbAxi4_MemoryReadWrite  Passed: 40  Affirmations Checked: 40  at 440 ns
```

```
%% DONE  FAILED  TbAxi4_MemoryReadWrite1  Total Error(s) = 7  Failures: 0  Errors: 7  Warnings:
0  Passed: 33  Affirmations Checked: 40  at 440 ns
%%    Default                          Failures: 0  Errors: 0  Warnings: 0  Passed: 0
%%    OSVVM                            Failures: 0  Errors: 0  Warnings: 0  Passed: 0
%%    TB                               Failures: 0  Errors: 0  Warnings: 0  Passed: 0
%%    Master_1                         Failures: 0  Errors: 7  Warnings: 0  Passed: 0
%%      Master_1: Protocol Error       Failures: 0  Errors: 0  Warnings: 0  Passed: 0
%%      Master_1: Data Check           Failures: 0  Errors: 7  Warnings: 0  Passed: 33
%%      Master_1: No response          Failures: 0  Errors: 0  Warnings: 0  Passed: 0
%%    Responder_1                      Failures: 0  Errors: 0  Warnings: 0  Passed: 0
%%      Responder_1: Data Check        Failures: 0  Errors: 0  Warnings: 0  Passed: 0
%%      Responder_1: No response       Failures: 0  Errors: 0  Warnings: 0  Passed: 0
```

Figure 29. ReportAlerts for each AlertLogID

7 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. See Address_Bus_Model_Independent_Transactions_user_guide.pdf in the documentation repository for details.

7.1 AXI4 VC Shared Directives

7.1.1 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)

7.1.2 Configuration Directives

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

7.2 AXI Master Transactions

The oData and iData in Master 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.

7.2.1 BurstMode Control Directives

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

Largely these are used indirectly through the SetAxi4Options and GetAxi4Options directives. See setting AXI4 Master Parameters. For AXI4 Master, OptVal can have a type of boolean, integer, or std_logic_vector.

7.2.2 Write Transactions

Write(TransactionRec, iAddr, iData [, StatusMsgOn])
WriteBurst(TransactionRec, iAddr, iNumFifoWords [, StatusMsgOn])
WriteAsync(TransactionRec, iAddr, iData [, StatusMsgOn])
WriteAddressAsync(TransactionRec, iAddr [, StatusMsgOn])
WriteDataAsync(TransactionRec, iAddr, iData [, StatusMsgOn]) **WriteDataAsync(TransactionRec, iAddr [, StatusMsgOn])**
WriteBurstAsync(TransactionRec, iAddr, iNumFifoWords[, StatusMsgOn])

INumFifoWords (input) specifies the number of words in the BurstFifo. 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.

7.2.3 Read Transactions

Read(TransactionRec, iAddr, oData [, StatusMsgOn])
ReadCheck(TransactionRec, iAddr, iData [, StatusMsgOn])
ReadBurst(TransactionRec, iAddr, iNumFifoWords [, StatusMsgOn])
ReadAddressAsync(TransactionRec, iAddr [, StatusMsgOn])
ReadData(TransactionRec, oData [, StatusMsgOn])
ReadCheckData(TransactionRec, iData [, StatusMsgOn])
TryReadData(TransactionRec, oData, Available [, StatusMsgOn])
TryReadCheckData(TransactionRec, iData, Available [, StatusMsgOn])

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

7.3 AXI4 Responder Transactions

The AXI4 Responder only supports single word transfers. Its interface is a little like a Master Read /Write and a little like Stream Send/Get. For burst transactions, see the Memory Responder.

The oData and iData in Master 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.

7.3.1 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])

7.3.2 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])

7.4 AXI4 Memory Responder Transactions

The AXI4 Memory VC supports transactions as a "backdoor" interface (non-functional) to the internal memory of the device. Only basic single word transactions are currently supported.

As a "backdoor" interface, all transactions finish without time passing. Hence there is no need to support Asynchronous (Async or Try) type operations. In a future release they may be added for symmetry to the Master model.

7.4.1 Write Transactions

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

7.4.2 Read Transactions

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

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

8 Setting AXI4 Parameters

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.

8.1 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.

8.2 Controlling Interface Signaling Characteristics and Timeouts

8.2.1 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
Master	WRITE_ADDRESS_VALID_DELAY_CYCLES	0
Master	WRITE_DATA_VALID_DELAY_CYCLES	0
Master	WRITE_DATA_VALID_BURST_DELAY_CYCLES	0
Responder	WRITE_RESPONSE_VALID_DELAY_CYCLES	0
Master	READ_ADDRESS_VALID_DELAY_CYCLES	0
Responder	READ_DATA_VALID_DELAY_CYCLES	0

Responder	READ_DATA_VALID_BURST_DELAY_CYCLES	0

8.2.2 Control: 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
Responder	WRITE_ADDRESS_READY_BEFORE_VALID	TRUE
Responder	WRITE_DATA_READY_BEFORE_VALID	TRUE
Master	WRITE_RESPONSE_READY_BEFORE_VALID	TRUE
Responder	READ_ADDRESS_READY_BEFORE_VALID	TRUE
Master	READ_DATA_READY_BEFORE_VALID	TRUE

8.2.3 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 30 shows xx_READY_DELAY_CYCLES = 2 when xx_READY_BEFORE_VALID is TRUE.

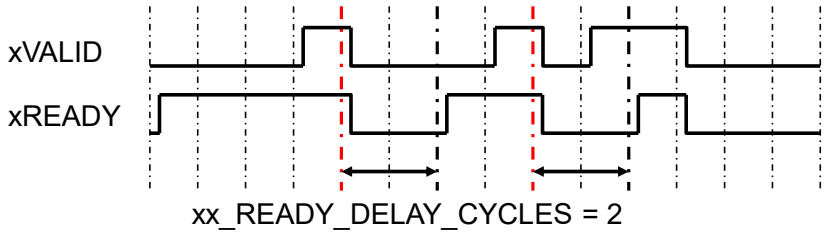

Figure 30. xx_READY_DELAY_CYCLES = 2 when xx_READY_BEFORE_VALID is TRUE

When xx_READY_BEFORE_VALID is FALSE, then xx_READY_DELAY_CYCLES is a relative to when xValid is asserted. Figure 31 shows xx_READY_DELAY_CYCLES = 2 when xx_READY_BEFORE_VALID is FALSE.

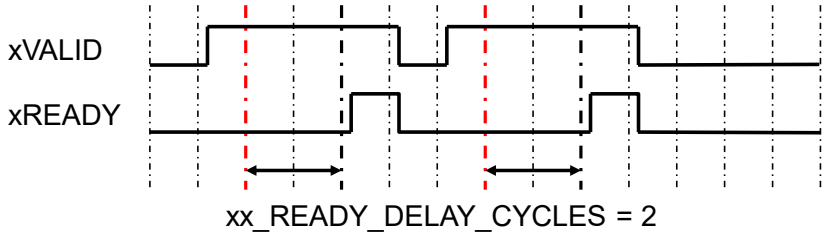

Figure 31. xx_READY_DELAY_CYCLES = 2 when xx_READY_BEFORE_VALID is FALSE

VC	Name	Initial Value in VC
Responder	WRITE_ADDRESS_READY_DELAY_CYCLES	0
Responder	WRITE_DATA_READY_DELAY_CYCLES	0
Master	WRITE_RESPONSE_READY_DELAY_CYCLES	0
Responder	READ_ADDRESS_READY_DELAY_CYCLES	0
Master	READ_DATA_READY_DELAY_CYCLES	0

8.2.4 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
Master	WRITE_ADDRESS_READY_TIME_OUT	25
Master	WRITE_DATA_READY_TIME_OUT	25
Responder	WRITE_RESPONSE_READY_TIME_OUT	25
Master	READ_ADDRESS_READY_TIME_OUT	25
Responder	READ_DATA_READY_TIME_OUT	25

8.2.5 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 responder 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 responder 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 responder must asssert xValid before a FAILURE is generated. xx_VALID_TIME_OUT value is an integer.

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

8.2.6 Setting Interface Signaling Characteristics and Timeouts

Figure 32 shows setting the Responder's Write Address interface so that AWREADY occurs two cycles after AWVALID is asserted.

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

Figure 32. Setting the Responder's Write Address Interface AWREADY Parameters

8.3 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.

8.3.1 Write Address Default Values

Signal	Driver	Configurable	AXI	Description
AWADDR	Master	No	All	Write Address
AWVALID	Master	No	All	Set when address channels has valid values
AWREADY	Responder	No	All	Set when responder ready to accept values
AWPROT	Master	Yes	All	Initial value = 0 Privilege level. Frequency of usage?
AWID	Master	Yes	Full	Write Address ID. Initial value = 0. Used in reordering operations.
AWLEN	Master	No	Full	Number of transfers in a burst. F(#Bytes, Word Width, Starting Address)
AWSIZE	Master	Yes	Full	Set to log2(Data Bytes) = Full width of interface.
AWBURST	Master	Yes	Full	Initial value = INCR Options: Fixed, Incr, Wrap.
AWLOCK	Master	Yes	AXI3	Initial value = 0.
AWCACHE	Master	Yes	Full	Initial value = 0. Cache Memory Access Types.
AWQOS	Master	Yes	Full	Initial value = 0. Quality of Service.
AWREGION	Master	Yes	Full	Initial value = 0. Partitions responders into separate areas.
AWUSER	Master	Yes	Full	Initial value = 0

8.3.2 Write Data Default Values

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

8.3.3 Write Response Default Values

Signal	Driver	Configurable	AXI	Description
BRESP	Responder	Yes	All	Initial = OKAY
BVALID	Responder	No	All	Set when responder has valid values
BREADY	Master	No	All	Set when master ready to accept values
BID	Responder	No	Full	Axi4Memory: Matches Address Write Axi4Responder: Set by configuration = 0
BUSER	Responder	No	Full	Axi4Memory: Matches Address Write Axi4Responder: Set by configuration = 0

8.3.4 Read Address Default Values

Signal	Driver	Configurable	AXI	Description
ARADDR	Master	No	All	Read Address
ARVALID	Master	No	All	Set when address channels has valid values

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

8.3.5 Read Data Default Values

Signal	Driver	Configurable	AXI	Description
RDATA	Responder	No	All	Write Address
RRESP	Responder	Yes	All	Initial = OKAY
RVALID	Responder	No	All	Set when channel has valid values
RREADY	Master	No	All	Set when responder ready to accept values
RID	Responder	Yes	AXI3	Axi4Memory: Matches Address Read Axi4Responder: Set by configuration = 0
RUSER	Responder	Yes	Full	Axi4Memory: Matches Address Read Axi4Responder: Set by configuration = 0
RLAST	Responder	No	Full	Asserted on last data value of transaction. Single word – always asserted Burst – last word

8.3.6 Setting Interface Default Values

All AXI interface values are std_logic_vector and must match the size of the corresponding interface object. Figure 33 shows setting the AWPROT and AWUSER values.

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

Figure 33. Setting values for AWPROT and AWUSER

8.3.7 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'.

9 Burst Transactions

The AXI4 Master VC supports burst transactions via the BurstFIFOs internal to AddressBusRecType. The AXI4 Memory VC responds to burst transactions and either writes to or reads from its internal memory.

9.1 Run the Demo

It is time to go back to demo mode. In the simulator run the test TbAxi4_SendGetBurst1 using the steps shown in Figure 34. You already compiled this test when you ran testbench.pro.

```
simulate TbAxi4_SendGetBurst1
```

Figure 34. Running TbAxi4_SendGetBurst1

9.2 Burst FIFOs are in the Interface

The AXI4 Master VC master implements two burst FIFOs, one for reading (ReadBurstFifo) and one for writing (WriteBurstFifo). Both Burst FIFOs are part of AddressBusRecType, see Figure 35. This makes the Burst FIFOs easily accessible to both the Axi4Master as well as the Test Sequencer (TestCtrl). The BurstFifo is implemented using a std_logic_vector based scoreboard. The FIFO is defined in ScoreboardPkg_slv.vhd (directory OsvvmLibraries/osvvm).

```
  type AddressBusRecType is record
    . . .
    -- Burst FIFOs
    WriteBurstFifo     : ScoreboardIdType ;
    ReadBurstFifo      : ScoreboardIdType ;
    . . .
  end record AddressBusRecType;
```

Figure 35. BurstFifo in AddressBusRecType

9.3 Interacting with the Burst FIFOs in the Test Sequencer

The Burst FIFOs support any operation the scoreboards support – such as push, pop, or check. See Scoreboard_user_guide.pdf. In addition, the package FifoFillPkg_slv.vhd adds support for burst operations, PushBurst, PopBurst, and CheckBurst. See OsvvmLibraries/common/src and the Address_Bus_Model_Independent_transactions_user_guide.pdf.

9.3.1 Accessing Burst FIFOs in a Test Case

To simplify access of the burst FIFO, the OSVVM Test Sequencer (TestCtrl) entity includes the aliases shown in Figure 36 in its entity declarative region. See the discussion on TestCtrl in 4.3 and 5.3. We recommend doing this in your testbenches.

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

Figure 36. Making the BurstFifos visible in the test sequencer (TestCtrl)

9.3.2 Filling the Burst FIFO

```
Push              (WriteBurstFifo, DataWord) ;
PushBurst         (WriteBurstFifo, VectorOfWords, FifoWidth)
PushBurstIncrement(WriteBurstFifo, FirstWord, Count, FifoWidth)
PushBurstRandom   (WriteBurstFifo, FirstWord, Count, FifoWidth)
```

9.3.3 Reading and/or Checking the Burst FIFO

```
DataWord := Pop(ReadBurstFifo) ;
Check              (ReadBurstFifo, CheckWord) ;
PopBurst           (ReadBurstFifo, VectorOfWords, FifoWidth)
CheckBurst         (ReadBurstFifo, VectorOfWords, FifoWidth)
CheckBurstIncrement(ReadBurstFifo, FirstWord, Count, FifoWidth)
CheckBurstRandom   (ReadBurstFifo, FirstWord, Count, FifoWidth)
```

9.3.4 Reading and Writing Bursts with the Axi4Master

For writing bursts with the WriteBurst transaction, first items must be pushed into the WriteBurstFIFO using the FIFO operations BurstFIFO.push, PushBurstIncrement, PushBurst, or PushBurstRandom before calling WriteBurst or WriteBurstAsync. When reading bursts, first call the ReadBurst transaction, and then items in the ReadBurstFIFO can be checked using the FIFO operations BurstFIFO.check, BurstFIFO.pop, CheckBurstIncrement, CheckBurst, or CheckBurstRandom. Figure 37 shows three calls to WriteBurst and ReadBurst that are similar to the ones in the test TbAxi4_MemoryBurst1.vhd.

```
constant WIDTH : integer := 32 ;
. . .

MasterProc : process
begin
  . . .
```

```
log("Write with ByteAddr = 8, 12 Bytes -- word aligned") ;
PushBurstIncrement(WriteBurstFifo, 3, 12) ;
WriteBurst(AxiInitiatorTransRec, X"0000_0008", 12) ;

ReadBurst (AxiInitiatorTransRec, X"0000_0008", 12) ;
CheckBurstIncrement(ReadBurstFifo, 3, 12) ;

log("Write with ByteAddr = x1A, 13 Bytes -- unaligned") ;
PushBurst(WriteBurstFifo, (1,3,5,7,9,11,13,15,17,19,21,23,25)) ;
WriteBurst(AxiInitiatorTransRec, X"0000_001A", 13) ;

ReadBurst (AxiInitiatorTransRec, X"0000_001A", 13) ;
CheckBurst(ReadBurstFifo, (1,3,5,7,9,11,13,15,17,19,21,23,25)) ;

log("Write with ByteAddr = 31, 12 Bytes -- unaligned") ;
PushBurstRandom(WriteBurstFifo, 7, 12) ;
WriteBurst(AxiInitiatorTransRec, X"0000_0031", 12) ;

ReadBurst (AxiInitiatorTransRec, X"0000_0031", 12) ;
CheckBurstRandom(ReadBurstFifo, 7, 12) ;
```

Figure 37. ReadBurst and WriteBurst as used in TbAxi4_MemoryBurst1.vhd

9.4 Interacting with the Burst FIFOs in the Axi4Master VC

9.4.1 Accessing the BurstFifo in Axi4Master VC

The Burst FIFOs are initialized in Axi4Master VC as shown Figure 38. Note that in Axi4Master VC the BurstFifos are accessed as record elements of TransRec.

```
TransactionDispatcher : process
 . . .
begin
  wait for 0 ns ;
  TransRec.WriteBurstFifo <= NewID(MODEL_INSTANCE_NAME &
      ": WriteBurstFifo", ModelID) ;
  TransRec.ReadBurstFifo  <= NewID(MODEL_INSTANCE_NAME &
      ": ReadBurstFifo",  ModelID) ;
  wait for 0 ns ;
```

Figure 38. Initialization of WriteBurstFifo and ReadBurstFifo in Axi4Master VC

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.