# OSVVM Model Library: AxiStream Verification Component

## User Guide for Release 2021.06

By

Jim Lewis

SynthWorks VHDL Training

Jim@SynthWorks.com

http://www.SynthWorks.com

## Table of Contents

# 1. Overview

The OSVVM AxiStream Verification Components (VCs) facilitate testing the interface and functionality of AxiStream devices. The OSVVM AxiStream VCs include AxiStreamTransmitter and AxiStreamReceiver. These VCs are intended to be part of a structured test environment.

The AxiStream verification components implement the complete AxiStream interface capability. They support bursting capability via BurstFifos in StreamRecType as well as through direct and algorithmic control of TLast during single word transfers. They support setting of TStrb and TKeep for either single word transfers or burst transfers. Within a burst transfer, they support sparse data streams (ie: TKeep=0 and/or TStrb=0). They support setting of TID, TDest, and TUser. For single word transfers, these can either be set from defaults in the VC or supplied as values to the transaction call. For Bursting, TID and TDest are intended to maintain their value throughout the entire transfer, and TUser can be set either for the entire transfer or on a word by word basis.

For the test case programming API (used in a test sequencer), the AxiStream VCs support the OSVVM Stream 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 AxiStream 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 (AxiStreamTransmitter and AxiStreamReceiver) 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).

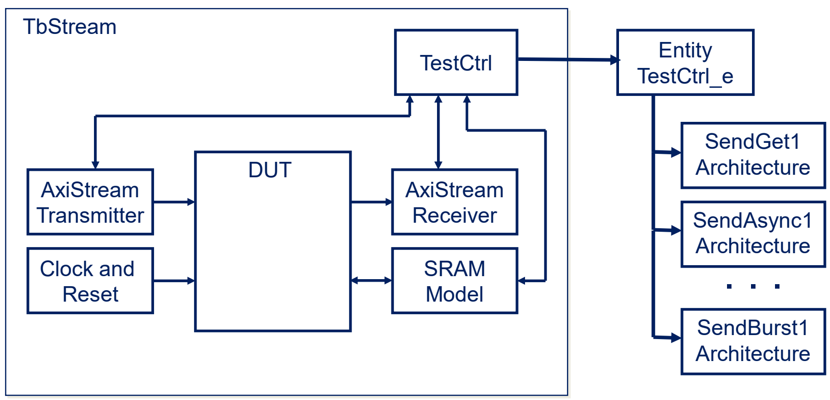


Figure 1. OSVVM Testbench Framework

## 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 Send) 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 AxiStream VC via a transaction interface (a record). The AxiStream 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 Send and WaitForClock transactions and the corresponding waveforms produced by the AxiStreamTransmitter verification component. Note this waveform implies that during the cycle in which data values A1, A3, A4, and A6 were sent, the AxiStream receiver was ready to receive TData and during the cycle in which data values A2 and A4 were sent, the AxiStream receiver was not ready to receive TData until a clock cycle later.

```
StreamTxProc : process
begin
  WaitForBarrier(StartTest) ;
  Send(StreamTxRec, X"A1") ;
  WaitForClock(StreamTxRec, 1) ;
  Send(StreamTxRec, X"A2") ;
  WaitForClock(StreamTxRec, 2) ;
  Send(StreamTxRec, X"A3") ;
  Send(StreamTxRec, X"A4") ;
  Send(StreamTxRec, X"A5") ;
  WaitForClock(StreamTxRec, 1) ;
  Send(StreamTxRec, X"A6") ;
```

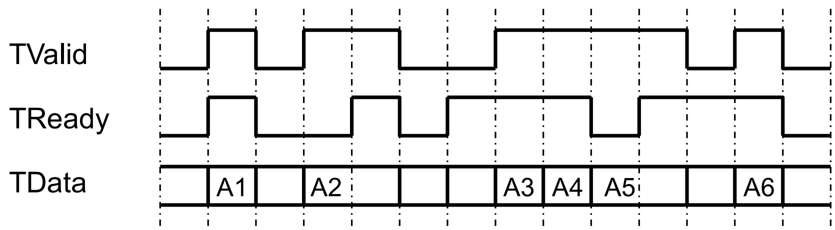


Figure 2. Waveform resulting from the calls to Send and WaitForClock

## 2.3 Stream Transaction Interface

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

Stream Transaction Interface, StreamRecType, is used to connect the verification component to TestCtrl. StreamRecType, shown in Figure 3, is defined in the Stream Model Independent Transaction package, StreamTransactionPkg.vhd, which is in the directory OsvvmLbraries/Common/Src.

```
  type StreamRecType is record
    -- Handshaking controls
    Rdy             : bit_max ;
    Ack             : bit_max ;
    -- Transaction Type
    Operation       : StreamOperationType ;
    -- Data and Transaction Parameter to and from the VC
    DataToModel     : std_logic_vector_max_c ;
    ParamToModel    : std_logic_vector_max_c ;
    DataFromModel   : std_logic_vector_max_c ;
    ParamFromModel  : std_logic_vector_max_c ;
    -- VC Options Parameters - used by SetModelOptions
    IntToModel      : integer_max ;
    BoolToModel     : boolean_max ;
    IntFromModel    : integer_max ;
    BoolFromModel   : boolean_max ;
    TimeToModel     : time_max ;
```

```
    TimeFromModel   : time_max ;
    -- Verification Component Options Type
    Options         : integer_max ;
  end record StreamRecType ;
```

Figure 3. StreamRecType

Note that DataToModel, ParamToModel, DataFromModel, and ParamFromModel are unconstrained. Hence, when they are used in a signal declaration they must be constrained. DataToModel and DataFromModel need to be sized to match TData. ParamToModel and ParamFromModel need to be sized to be (TID'length + TDest'length + TUser'length + 1) in length.

Figure 4 shows the declaration StreamTxRec (which connects the AxiStreamTransmitter to TestCtrl) and StreamRxRec (which connects the AxiStreamReceiver to TestCtrl).

```
constant AXI_PARAM_WIDTH : integer :=
      TID'length + TDest'length + TUser'length + 1;

signal StreamTxRec, StreamRxRec : StreamRecType(
    DataToModel   (AXI_DATA_WIDTH-1  downto 0),
    ParamToModel  (AXI_PARAM_WIDTH-1 downto 0),
    DataFromModel (AXI_DATA_WIDTH-1  downto 0),
    ParamFromModel(AXI_PARAM_WIDTH-1 downto 0)
  ) ;
```

Figure 4. StreamRecType

## 2.4 AxiStream Context Declaration

To simplify the usage of OSVVM AxiStream packages, a context declaration that references all of the OSVVM AxiStream 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 5 shows the AxiStreamContext as defined in AxiStreamContext.vhd.

```
context AxiStreamContext is
  library osvvm_common ;
    context osvvm_common.OsvvmCommonContext ;

  library osvvm_axi4 ;
    use osvvm_axi4.AxiStreamOptionsPkg.all ;
    use osvvm_axi4.Axi4CommonPkg.all ;
    use osvvm_axi4.AxiStreamComponentPkg.all ;
end context AxiStreamContext ;
```

Figure 5. AxiStreamContext

## 2.5 Component Declarations for AxiStream 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.

## 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 6. 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 6. 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 7 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 7. 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. TbStream: "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 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 AxiStream.

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

Use the steps in Figure 8 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/AxiStream/testbench.pro
```

Figure 8. Compiling and Running OSVVM

## 4.2 TbStream: AxiStream Test Environment - Connected Transaction Interface

In the previous section, you ran TbStream.vhd, the AxiStream Testbench. It is in the directory OsvvmLibraries/AXI4/AxiStream/testbench. It is structured as shown in Figure 9.

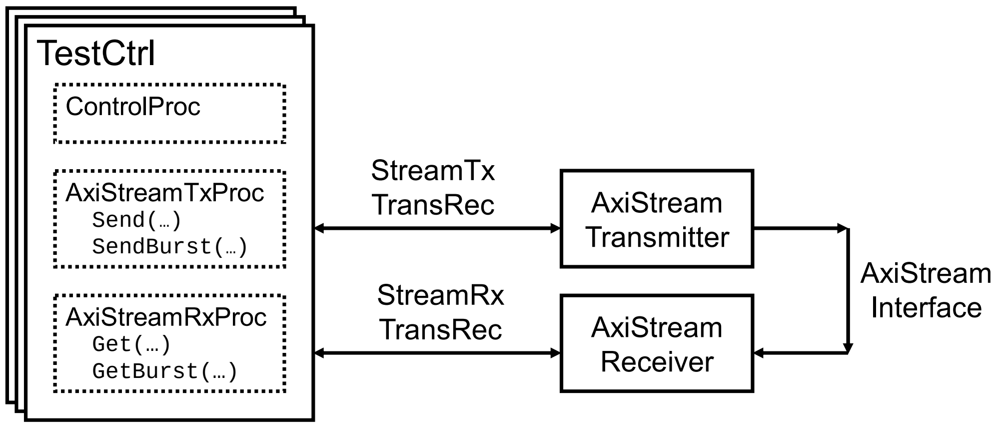


Figure 9. TbStream

TbStream 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 TbStream.vhd is shown in Figure 10. For more details, see TbStream.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.AxiStreamContext ;
. . .
entity  TbStream  is
end entity TbStream ;
architecture TestHarness of TbStream is
  signal StreamTxRec, StreamRxRec :
    StreamRecType( . . . ) ;
  . . .
begin
  osvvm.TbUtilPkg.CreateClock (Clk, tperiod_Clk) ;
  osvvm.TbUtilPkg.CreateReset (nReset, . . .) ;
  Receiver_1 :    AxiStreamReceiver    ( . . .,  StreamRxRec) ;
  Transmitter_1 : AxiStreamTransmitter ( . . .,  StreamTxRec) ;
  TestCtrl_1 :    TestCtrl (nReset, StreamTxRec, StreamRxRec) ;
end TestHarness ;
```

Figure 10. A sketch of TbStream.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 Transmitter_1 and Receiver_1. Our intent is to reuse some of the same test cases with other Transmitter and Receiver verification components (such as UART) – 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 11, consists of transaction interface connections.

```
library ieee ;
  use ieee.std_logic_1164.all ;
  use ieee.numeric_std.all ;
  use ieee.numeric_std_unsigned.all ;

library OSVVM ;
  context OSVVM.OsvvmContext ;

library osvvm_AXI4 ;
  context osvvm_AXI4.AxiStreamContext ;

entity TestCtrl is
  generic (
    ID_LEN       : integer ;
    DEST_LEN     : integer ;
    USER_LEN     : integer
  ) ;
  port (
      -- Global Signal Interface
      nReset             : In    std_logic ;

      -- Transaction Interfaces
      StreamTxRec        : InOut StreamRecType ;
      StreamRxRec        : InOut StreamRecType
  ) ;

  -- Derive AXI interface properties from the StreamTxRec
  constant DATA_WIDTH : integer := StreamTxRec.DataToModel'length ;
  constant DATA_BYTES : integer := DATA_WIDTH/8 ;

  -- Simplifying access to Burst FIFOs using aliases
  alias TxBurstFifo : ScoreboardIdType is StreamTxRec.BurstFifo ;
  alias RxBurstFifo : ScoreboardIdType is StreamRxRec.BurstFifo ;

end entity TestCtrl ;
```

Figure 11. TestCtrl.vhd

## 4.4 AxiStream Transmitter – Connected Transaction Interface (Record Port)

The AxiStreamTransmitter entity interface is shown in Figure 12. It has the full set of AxiStream interface signals as well as the transaction interface (TransRec).

```
entity AxiStreamTransmitter is
  generic (
    MODEL_ID_NAME  : string := "" ;
    INIT_ID        : std_logic_vector := "" ;
    INIT_DEST      : std_logic_vector := "" ;
    INIT_USER      : std_logic_vector := "" ;
    INIT_LAST      : natural := 0 ;

    tperiod_Clk    : time := 10 ns ;
    tpd_Clk_TValid : time := 2 ns ;
    -- . . .  see entity for remaining generics
    tpd_Clk_TValid : time := 2 ns
  ) ;
  port (
    -- Globals
    Clk       : in  std_logic ;
    nReset    : in  std_logic ;

    -- AXI Transmitter Functional Interface
    TValid    : out std_logic ;
    TReady    : in  std_logic ;
    TID       : out std_logic_vector ;
    TDest     : out std_logic_vector ;
    TUser     : out std_logic_vector ;
    TData     : out std_logic_vector ;
    TStrb     : out std_logic_vector ;
    TKeep     : out std_logic_vector ;
    TLast     : out std_logic ;

    -- Testbench Transaction Interface
    TransRec  : inout StreamRecType
  ) ;

  -- Derive AXI interface properties from interface signals
  constant AXI_STREAM_DATA_WIDTH : integer := TData'length ;

end entity AxiStreamTransmitter ;
```

Figure 12. AxiStreamTransmitter

## 4.5 AxiStream Receiver – Connected Transaction Interface (Record Port)

The AxiStreamReceiver entity interface is shown in Figure 13. It has the full set of AxiStream interface signals as well as the transaction interface (TransRec).

```
entity AxiStreamReceiver is
  generic (
    MODEL_ID_NAME  : string :="" ;
    INIT_ID        : std_logic_vector := "" ;
    INIT_DEST      : std_logic_vector := "" ;
    INIT_USER      : std_logic_vector := "" ;
    INIT_LAST      : natural := 0 ;

    tperiod_Clk    : time := 10 ns ;
    tpd_Clk_TReady : time := 2 ns
  ) ;
  port (
    -- Globals
    Clk       : in  std_logic ;
    nReset    : in  std_logic ;

    -- AXI Master Functional Interface
    TValid    : in  std_logic ;
    TReady    : out std_logic ;
    TID       : in  std_logic_vector ;
    TDest     : in  std_logic_vector ;
    TUser     : in  std_logic_vector ;
    TData     : in  std_logic_vector ;
    TStrb     : in  std_logic_vector ;
    TKeep     : in  std_logic_vector ;
    TLast     : in  std_logic ;

    -- Testbench Transaction Interface
    TransRec  : inout StreamRecType
  ) ;

  -- Derive AXI interface properties from interface signals
  constant AXI_STREAM_DATA_WIDTH   : integer := TData'length ;

end entity AxiStreamReceiver ;
```

Figure 13. AxiStreamReceiver

## 4.6 AxiStream Transmitter and Receiver Generics

AxiStreamTransmitter and AxiStreamReceiver both have a similar set of generics. The MODEL_ID_NAME optionally specifies the model AlertLogID name. If the MODEL_ID_NAME is not specified, the component instance label will be used (preferred method).

INIT_ID, INIT_DEST, INIT_USER, and INIT_LAST set the internal default values for TID, TDest, TUser, and TLast. In AxiStreamTransmitter, these are default driving values. In AxiStreamReceiver, these are default values for checking. Note that when INIT_ID, INIT_DEST, and INIT_USER are specified, they must be the same size as their corresponding interface signals.

The remaining generics specify timing. Tperiod_Clk specifies the clock frequency. Tpd_Clk_* specifies the delay for each interface output. See the file AxiStreamTransmitter.vhd and AxiStreamReceiver.vhd for the details of the generics.

## 5. TbStream: "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 testbenches and the process to run them is similar to what is discussed here for AxiStream.

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

Use the steps in Figure 14 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/AxiStream/testbench.pro
```

Figure 14. Compiling and Running OSVVM

## 5.2 TbStream: AxiStream Test Environment - Virtual Transaction Interface

In the previous section, you ran TbStream.vhd, the AxiStream Testbench. It is in the directory OsvvmLibraries/AXI4/AxiStream/testbenchVti. It is structured as shown in Figure 15. The record connections (shown with dotted lines) use external names rather than direct signal connections.

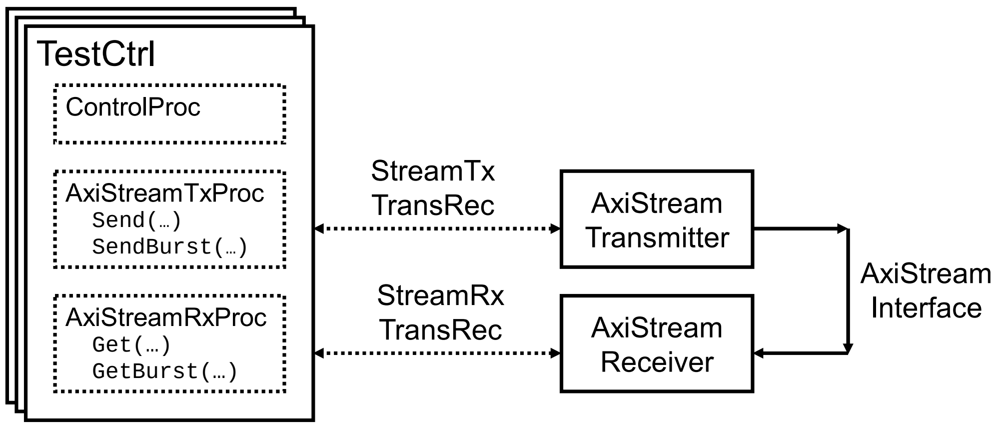


Figure 15. TbStream for Virtual Transaction Interfaces

TbStream 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 TbStream.vhd is shown in Figure 16. For more details, see AxiStream/testbenchVti/TbStream.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.AxiStreamContext ;
. . .
entity  TbStream  is
end entity TbStream ;
architecture TestHarness of TbStream is
  . . .
begin
  osvvm.TbUtilPkg.CreateClock (Clk, tperiod_Clk) ;
  osvvm.TbUtilPkg.CreateReset (nReset, . . .) ;
  Receiver_1 :    AxiStreamReceiver    ( . . .) ;
  Transmitter_1 : AxiStreamTransmitter ( . . .) ;
  TestCtrl_1 :    TestCtrl (nReset) ;
end TestHarness ;
```

Figure 16. A sketch of TbStream.vhd for Virtual Transaction Interfaces

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 Transmitter_1 and Receiver_1. Our intent is to reuse some of the same test cases with other Transmitter and Receiver verification components (such as UART) – and hence the more generic naming.

### 5.3 TestCtrl Entity – Virtual Transaction Interface (External Names)

Tests are written as architectures of the test sequencer, TestCtrl. The entity for TestCtrl, shown in Figure 17, consists of transaction interface connections.

```
library ieee ;
  use ieee.std_logic_1164.all ;
  use ieee.numeric_std.all ;
  use ieee.numeric_std_unsigned.all ;

library OSVVM ;
  context OSVVM.OsvvmContext ;

library osvvm_AXI4 ;
  context osvvm_AXI4.AxiStreamContext ;

entity TestCtrl is
  generic (
    ID_LEN       : integer ;
    DEST_LEN     : integer ;
    USER_LEN     : integer
  ) ;
  port (
      -- Global Signal Interface
      nReset             : In    std_logic
  ) ;

  -- Connect transaction interfaces using external names
  alias StreamTxRec is <<signal ^.Transmitter_1.TransRec : StreamRecType>>;
  alias StreamRxRec is <<signal ^.Receiver_1.TransRec : StreamRecType>> ;

  -- Derive AXI interface properties from the StreamTxRec
  constant DATA_WIDTH : integer := StreamTxRec.DataToModel'length ;
  constant DATA_BYTES : integer := DATA_WIDTH/8 ;

  -- Simplifying access to Burst FIFOs using aliases
  alias TxBurstFifo : ScoreboardIdType is StreamTxRec.BurstFifo ;
  alias RxBurstFifo : ScoreboardIdType is StreamRxRec.BurstFifo ;

end entity TestCtrl ;
```

Figure 17. TestCtrl.vhd for Virtual Transaction Interfaces

## 5.4 AxiStream Transmitter – Virtual Transaction Interface (External Names)

The AxiStreamTransmitter entity interface is shown in Figure 18. It has the full set of AxiStream interface signals. 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).

```
entity AxiStreamTransmitterVti is
  generic (
    MODEL_ID_NAME  : string := "" ;
    INIT_ID        : std_logic_vector := "" ;
    INIT_DEST      : std_logic_vector := "" ;
    INIT_USER      : std_logic_vector := "" ;
    INIT_LAST      : natural := 0 ;

    tperiod_Clk    : time := 10 ns ;
    tpd_Clk_TValid : time := 2 ns ;
    -- . . .  see entity for remaining generics
  ) ;
  port (
    -- Globals
    Clk       : in  std_logic ;
    nReset    : in  std_logic ;

    -- AXI Transmitter Functional Interface
    TValid    : out std_logic ;
    TReady    : in  std_logic ;
    TID       : out std_logic_vector ;
    TDest     : out std_logic_vector ;
    TUser     : out std_logic_vector ;
    TData     : out std_logic_vector ;
    TStrb     : out std_logic_vector ;
    TKeep     : out std_logic_vector ;
    TLast     : out std_logic
  ) ;

  -- Derive AXI interface properties from interface signals
  constant AXI_STREAM_DATA_WIDTH   : integer := TData'length ;
  constant AXI_STREAM_PARAM_WIDTH  : integer :=
      TID'length + TDest'length + TUser'length + 1 ;

  -- Testbench Transaction Interface - Access via external names
  signal TransRec  : StreamRecType (
    DataToModel   (AXI_STREAM_DATA_WIDTH-1  downto 0),
    DataFromModel (AXI_STREAM_DATA_WIDTH-1  downto 0),
    ParamToModel  (AXI_STREAM_PARAM_WIDTH-1 downto 0),
    ParamFromModel(AXI_STREAM_PARAM_WIDTH-1 downto 0)
  ) ;
  end entity AxiStreamTransmitterVti ;
```

Figure 18. AxiStreamTransmitterVti

 

## 5.5 AxiStream Receiver – Virtual Transaction Interface (External Names)

The AxiStreamReceiver entity interface is shown in Figure 19. It has the full set of AxiStream interface signals. 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).

```
 entity AxiStreamReceiverVti is
  generic (
    MODEL_ID_NAME  : string :="" ;
    INIT_ID        : std_logic_vector := "" ;
    INIT_DEST      : std_logic_vector := "" ;
    INIT_USER      : std_logic_vector := "" ;
    INIT_LAST      : natural := 0 ;

    tperiod_Clk    : time := 10 ns ;
    tpd_Clk_TReady : time := 2 ns
  ) ;
  port (
    -- Globals
    Clk       : in  std_logic ;
    nReset    : in  std_logic ;

    -- AXI Master Functional Interface
    TValid    : in  std_logic ;
    TReady    : out std_logic ;
    TID       : in  std_logic_vector ;
    TDest     : in  std_logic_vector ;
    TUser     : in  std_logic_vector ;
    TData     : in  std_logic_vector ;
    TStrb     : in  std_logic_vector ;
    TKeep     : in  std_logic_vector ;
    TLast     : in  std_logic
  ) ;

  -- Derive AXI interface properties from interface signals
  constant AXI_STREAM_DATA_WIDTH   : integer := TData'length ;
  constant AXI_STREAM_PARAM_WIDTH  : integer :=
      TID'length + TDest'length + TUser'length + 1 ;

  -- Testbench Transaction Interface - Access via external names
  signal TransRec  : StreamRecType (
    DataToModel   (AXI_STREAM_DATA_WIDTH-1  downto 0),
    DataFromModel (AXI_STREAM_DATA_WIDTH-1  downto 0),
    ParamToModel  (AXI_STREAM_PARAM_WIDTH-1 downto 0),
    ParamFromModel(AXI_STREAM_PARAM_WIDTH-1 downto 0)
  ) ;
end entity AxiStreamReceiverVti ;
```

Figure 19. AxiStreamReceiverVti

## 5.6 AxiStream Transmitter and Receiver Generics

AxiStreamTransmitterVti and AxiStreamReceiverVti both have a similar set of generics. The MODEL_ID_NAME optionally specifies the model AlertLogID name. If the MODEL_ID_NAME is not specified, the component instance label will be used (preferred method).

INIT_ID, INIT_DEST, INIT_USER, and INIT_LAST set the internal default values for TID, TDest, TUser, and TLast. In AxiStreamTransmitter, these are default driving values. In AxiStreamReceiver, these are default values for checking. Note that when INIT_ID, INIT_DEST, and INIT_USER are specified, they must be the same size as their corresponding interface signals.

The remaining generics specify timing. Tperiod_Clk specifies the clock frequency. Tpd_Clk_* specifies the delay for each interface output. See the file AxiStreamTransmitter.vhd and AxiStreamReceiver.vhd for the details of the generics.

## 6. Writing Tests Using the AxiStream VC

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 20. The control process is used for test initialization and finalization. Each test process creates interface waveform sequences by calling the transaction procedures (Send, SendBurst, Get, GetBurst, Check, CheckBurst …). This test architecture is based on the test TbStream_SendGet1.vhd in the directory OsvvmLibraries/AXI4/AxiStream/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 SendGet1 of TestCtrl is
  . . .
begin
  ControlProc : process
  begin
    . . .
    WaitForBarrier(TestDone, 35 ms) ;
    ReportAlerts ;
    std.env.stop;
  end process ;

  TransmitterProc : process
  begin
    WaitForClock(StreamTxRec, 2) ;
    . . .
    Send(StreamTxRec, Data) ;
    . . .
    WaitForBarrier(TestDone) ;
  end process TransmitterProc ;

  ReceiverProc : process
  begin
    WaitForClock(StreamRxRec, 2) ;
    . . .
    Get(StreamRxRec, RxData) ;
    . . .
    WaitForBarrier(TestDone) ;
  end process ReceiverProc ;
end SendGet1 ;
```

Figure 20. TestCtrl Architecture

## 6.1 Test Initialization

The ControlProc both initializes and finalizes a test. Test initialization is shown in Figure 21. This is based on the code in TbStream_SendGet1.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 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("TbStream_SendGet1");
  TBID <= GetAlertLogID("TB");
  TxID <= GetAlertLogID("AxiStreamTransmitter_1");
  SetLogEnable(PASSED, TRUE) ;
  SetLogEnable(TxID, INFO, TRUE) ;

  -- Wait for simulation elaboration/initialization
  wait for 0 ns ;  wait for 0 ns ;
  TranscriptOpen("./results/TbStream_SendGet1.txt") ;
  SetTranscriptMirror(TRUE) ;

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

Figure 21. Test Initialization

## 6.2 A Simple Directed Test

A simple test can be created by transmitting (send) values on one interface (here AxiStreamTransmitter) and receiving (Get) and checking (AffirmIfEqual) it on another interface (here AxiStreamReceiver). The receiving and checking can also be done using the Check transaction (Get plus check inside the VC). These are shown in Figure 22. A more complex variation of this is in TbStream_SendGet1.vhd.

```
TransmitterProc : process
  . . .
begin
  Send(StreamTxRec, X"10") ;


  Send(StreamTxRec, X"11") ;
  . . .
end process TransmitterProc ;
```

```
ReceiverProc : process
  . . .
begin
  Get(StreamRxRec, RxD) ;
  AffirmIfEqual(TBID, RxD, X"10");

  Check(StreamRxRec, X"11") ;
  . . .
end process ReceiverProc ;
```

Figure 22. A Simple Directed Test

The AffirmIfEqual checks its two parameters. It produces a log "PASSED" message if they are equal and alert "ERROR" message otherwise. An ERROR message is shown in Figure 23. "TB" is produced in the message since AffirmIfEqual uses the TBID and "TB" matches the string used with GetAlertLogID for TBID.

```
%% Alert ERROR  In TB, Received: 08 /= Expected: 10 at 2150 ns
```

Figure 23. Messaging from AffirmIfEqual

The Check 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 24. "AxiStreamReceiver_1" is produced in the message since it matches the string that the verification component used to create its ModelID – see section AxiStream Verification Components for a discussion of how this happens.

```
%% Log   PASSED In AxiStreamReceiver_1: Data Check, Received: 11 at 3150 ns
```

Figure 24. Messaging from Check

## 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 25. 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 25. 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 26 shows a representative PASSED and FAILED message that will be printed.

```
%% DONE  PASSED  TbStream_SendGet1  Passed: 48  Affirmations Checked: 48  at 100000 ns
```

```
%% DONE  FAILED  TbStream_SendGet1  Total Error(s) = 7  Failures: 0  Errors: 7  Warnings: 0
Passed: 41  Affirmations Checked: 48  at 100000 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
%%    AxiStreamTransmitter_1              Failures: 0  Errors: 0  Warnings: 0  Passed: 0
%%      AxiStreamTransmitter_1: No response   Failures: 0  Errors: 0  Warnings: 0  Passed: 0
%%    AxiStreamReceiver_1                 Failures: 0  Errors: 7  Warnings: 0  Passed: 0
%%      AxiStreamReceiver_1: Data Check   Failures: 0  Errors: 7  Warnings: 0  Passed: 41
%%      AxiStreamReceiver_1: No response  Failures: 0  Errors: 0  Warnings: 0  Passed: 0
%%      AxiStreamReceiver_1: BurstFifo    Failures: 0  Errors: 0  Warnings: 0  Passed: 0
```

Figure 26. ReportAlerts for each AlertLogID

# 7. AxiStream VC Transactions

## 7.1 AxiStream Supported Stream Independent Transactions

The AxiStream VC implements the OSVVM Stream Model Independent Transactions. The following is a summary of the supported transactions. See Stream_Model_Independent_Transactions_user_guide.pdf in the documentation repository for details.

### 7.1.1 General Directives

| |
|---|
| **WaitForTransaction(TransactionRec)** |
| **WaitForClock(TransactionRec, NumberOfClocks)** |
| **GetTransactionCount(TransactionRec, Count)** |
| **GetAlertLogID(TransactionRec, AlertLogID)** |
| **GetErrorCount(TransactionRec, ErrorCount)** |

### 7.1.2 BurstMode Control Directives

| |
|---|
| **SetBurstMode (TransactionRec, STREAM_BURST_WORD_MODE) ;**<br>**SetBurstMode (TransactionRec, STREAM_BURST_WORD_PARAM_MODE) ;**<br>**SetBurstMode (TransactionRec, STREAM_BURST_BYTE_MODE) ;** |
| **GetBurstMode (TransactionRec, OptVal)** |

### 7.1.3 AxiStream Configuration Directives

| |
|---|
| **SetModelOptions(TransactionRec, Option, OptVal)** |
| **GetModelOptions(TransactionRec, Option, OptVal)** |

Largely these are used indirectly through the SetAxiStreamOptions and GetAxiStreamOptions directives. See setting AxiStream Parameters. For AxiStream, OptVal can have a type of boolean, integer, or std_logic_vector.

### 7.1.4 Transmitter Transactions

| |
|---|
| **Send(TransactionRec, Data, Param[, StatusMsgOn])**<br>**Send(TransactionRec, Data[, StatusMsgOn])** |
| **SendAsync(TransactionRec, Data, Param[, StatusMsgOn])**<br>**SendAsync(TransactionRec, Data[, StatusMsgOn])** |
| **SendBurst(TransactionRec, NumFifoWords, Param[, StatusMsgOn])**<br>**SendBurst(TransactionRec, NumFifoWords[, StatusMsgOn])** |
| **SendBurstAsync(TransactionRec, NumFifoWords, Param[, StatusMsgOn])**<br>**SendBurstAsync(TransactionRec, NumFifoWords[, StatusMsgOn])** |

Here the Param (input) parameter specifies the values ID & Dest & User & Last (in that order). If less than ID'length + Dest'lengh + User'length + 1 values are specified, the left-most values will be filled with a '-'. As an input, a value of '-' indicates the corresponding field is not specified and will use the default value DEFAULT_ID, DEFAULT_DEST, DEFAULT_USER, and DEFAULT_LAST (see AxiStream Parameters). For SendBurst and SendBurstAsync when the BurstMode is STREAM_BURST_WORD_PARAM_MODE the value in the BurstFifo specifies (Data & User).

NumFifoWords (input) specifies the number of words in the BurstFifo. Note that when in the mode, STREAM_BURST_BYTE_MODE, this will be the number of bytes in the transfer, otherwise it is the number of words in the transfer.

Currently the AxiStreamTransmitter does not use the optional StatusMsgOn (boolean input) parameter.

### 7.1.5 Receiver Transactions

```
Get(TransactionRec, Data, Param[, StatusMsgOn])
Get(TransactionRec, Data[, StatusMsgOn])

TryGet(TransactionRec, Data, Param, Available[, StatusMsgOn])
TryGet(TransactionRec, Data, Available[, StatusMsgOn])

GetBurst(TransactionRec, NumFifoWords, Param[, StatusMsgOn])
GetBurst(TransactionRec, NumFifoWords[, StatusMsgOn])

TryGetBurst(TransactionRec, NumFifoWords, Param, Available[, StatusMsgOn])
TryGetBurst TransactionRec, NumFifoWords, Available[, StatusMsgOn])
```

```
Check(TransactionRec, Data, Param[, StatusMsgOn])
Check(TransactionRec, Data[, StatusMsgOn])

TryCheck(TransactionRec, Data, Param, Available[, StatusMsgOn])
TryCheck(TransactionRec, Data, Available[, StatusMsgOn])

CheckBurst(TransactionRec, NumFifoWords, Param[, StatusMsgOn])
CheckBurst(TransactionRec, NumFifoWords[, StatusMsgOn])

TryCheckBurst(TransactionRec, NumFifoWords, Param, Available[, StatusMsgOn])
TryCheckBurst TransactionRec, NumFifoWords, Available[, StatusMsgOn])
```

Here the Param parameter specifies the values ID & Dest & User & Last (in that order). For Get, TryGet, GetBurst, TryGetBurst the Param parameter returns the values that were received by TID, TDest, TUser, and TLast.

For Check, TryCheck, CheckBurst and TryCheckBurst, Param parameter is an input. If less than ID'length + Dest'lengh + User'length + 1 values are specified, the left-most values will be filled with a '-'. As an input, a value of '-' indicates the corresponding field is not specified and will use the default value DEFAULT_ID, DEFAULT_DEST, DEFAULT_USER, and DEFAULT_LAST (see AxiStream Parameters). For CheckBurst or TryCheckBurst when the BurstMode is STREAM_BURST_WORD_PARAM_MODE the value in the BurstFifo specifies (Data & User).

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

Currently the AxiStreamReceiver does not use the optional StatusMsgOn (boolean input) parameter.

## 7.2 AxiStream Parameters

The AxiStream 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.

### 7.2.1 SetAxiStreamOptions / GetAxiStreamOptions

Model options are set using SetAxiStreamOptions and retrieved using GetAxiStreamOptions. 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 AxiStreamOptionsPkg.vhd.

```
SetAxiStreamOptions(TransactionRec, Option, OptVal)
GetAxiStreamOptions(TransactionRec, Option, OptVal)
```

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

### 7.2.2 Options common to AxiStreamTransmitter and AxiStreamReceiver

| | |
|---|---|
| DEFAULT_ID<br>std_logic_vector | Default value for TID if not specified in a send or check.<br>Initial value = INIT_ID (generic) if set, otherwise 0 |
| DEFAULT_DEST<br>std_logic_vector | Default value for TDest if not specified in a send or check.<br>Initial value = INIT_DEST (generic) if set, otherwise 0. |
| DEFAULT_USER<br>std_logic_vector | Default value for TUser if not specified in a send or check.<br>Initial value = INIT_USER (generic) if set, otherwise 0 |
| DEFAULT_LAST<br>integer | Default value for TLast if not specified in a send or check.<br>If value <= 1, then TLast = ??(DEFAULT_LAST=1).<br>If value > 1, then TLast = ??(NumOperations mod DEFAULT_LAST = 0).<br>Initial value = INIT_LAST (generic) which defaults to 0. |

The following set defaults for TID, TDest, and TUser. Note that the std_logic_vector value must match the size of the corresponding interface object.

```
SetAxiStreamOptions(TransactionRec, DEFAULT_ID,   X"01") ;
SetAxiStreamOptions(TransactionRec, DEFAULT_DEST, X"2") ;
SetAxiStreamOptions(TransactionRec, DEFAULT_USER, X"1") ;
```

The following set default so that it generates TLast once every 16 transfers.

```
SetAxiStreamOptions(TransactionRec, DEFAULT_LAST, 16) ;
```

### 7.2.3 AxiStreamTransmitter

| | |
|---|---|
| TRANSMIT_VALID_DELAY_CYCLES<br>Integer. Initialized to 0 | Specifies the number of cycles before a transaction starts once the AxiStreamTransmitter receives the transaction. |
| TRANSMIT_VALID_BURST_DELAY_CYCLES<br>Integer. Initialized to 0 | Specifies the number of cycles between each word during a burst cycle. |
| TRANSMIT_READY_TIME_OUT<br>Integer. Initialized to integer'right | Generates FAILURE if TValid is asserted for more specified number of clocks without TReady. Disable by setting to Integer'right. |

### 7.2.4 AxiStreamReceiver

| | |
|---|---|
| RECEIVE_READY_BEFORE_VALID<br>Boolean. Initialized to TRUE. | If TRUE generate TReady even if TValid is not asserted. |
| RECEIVE_READY_DELAY_CYCLES<br>Integer. Initialized to 0. | Number of clock cycles to delay assertion of TReady. If READY_BEFORE_VALID is TRUE, then number of clocks from previous cycle ending.<br>If READY_BEFORE_VALID is FALSE, then the number of clocks after RValid. |
| DROP_UNDRIVEN<br>Boolean. Initialized to FALSE. | If TRUE, then undriven values in a burst stream are not copied to a BYTE wide burst FIFO. Has no impact if the BurstFifo is word oriented (default). |

When RECEIVE_READY_BEFORE_VALID is TRUE, then RECEIVE_READY_DELAY_CYCLES is a relative to when the last transfer completed. Figure 27 shows RECEIVE_READY_DELAY_CYCLES = 2 when RECEIVE_READY_BEFORE_VALID is TRUE.

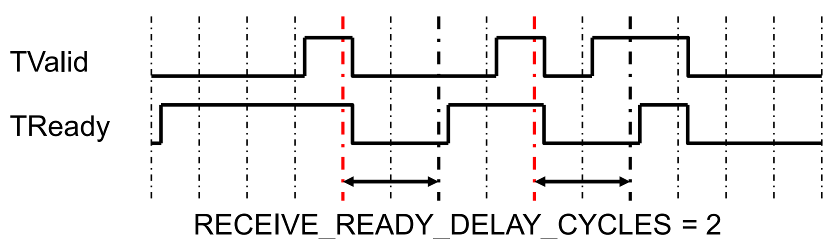


Figure 27. RECEIVE_READY_DELAY_CYCLES = 2 when RECEIVE_READY_BEFORE_VALID is TRUE

When RECEIVE_READY_BEFORE_VALID is FALSE, then RECEIVE_READY_DELAY_CYCLES is a relative to when TValid is asserted. Figure 28 shows RECEIVE_READY_DELAY_CYCLES = 2 when RECEIVE_READY_BEFORE_VALID is FALSE.

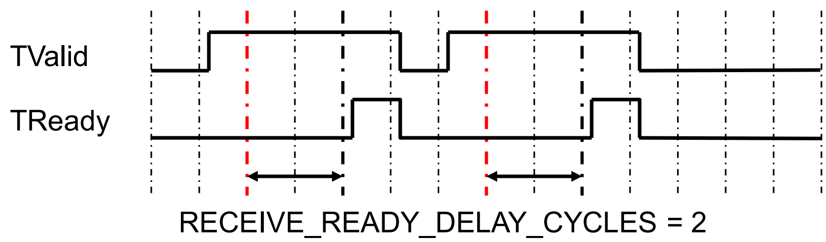


Figure 28. RECEIVE_READY_DELAY_CYCLES = 2 when RECEIVE_READY_BEFORE_VALID is FALSE

## 7.3 Setting and Checking TKeep and TStrb

On the AxiStream interface, a TStrb value that corresponds to a data value of '1' indicates the value contains valid data. A value of '0' indicates it is a filler value. A TKeep value of '1' indicates the value is either valid data or a filler value that may not be dropped. A value of '0' indicates the value may be dropped by the interface.

Rather than supplying this sort of information as a value in the transaction call, the OSVVM AxiStreamTransmitter VC uses a data value of X"UU" to indicate the data byte is to have TStrb = '0' and TKeep = '0' and a data value of X"WW" to indicate the data byte is to have TStrb = '0' and TKeep = '1'. This applies to values supplied either via the Send transaction or via the BurstFifo for a SendBurst transaction.

Similarly, in the AxiStreamReceiver VC, when TKeep = '0', then the corresponding data byte will be X"UU" and if TKeep = '1' and TStrb = '0', then the corresponding data byte will be X"WW". If the transaction is a GetBurst, and the BurstFIFO is configured to receive bytes, and the DropUndriven VC parameter is TRUE, and the received byte is X"UU", then it will be dropped (ie not put into the BurstFIFO).

## 8. Burst Transactions

The AxiStream VCs support burst transactions via the BurstFIFOs internal to StreamRecType.

### 8.1 Run the Demo

It is time to go back to demo mode. In the simulator run the test TbStream_SendGetBurst1 using the steps shown in Figure 29. You already compiled this test when you ran testbench.pro.

```
simulate TbStream_SendGetBurst1
```

Figure 29. Running TbStream_SendGetBurst1

### 8.2 BurstFifo is in the Interface

The BurstFifo is part of StreamRecType, see Figure 30. This makes the BurstFifo easily accessible to both the AxiStream VC as well as the Test Sequencer (TestCtrl). The BurstFifo is implemented using a std_logic_vector based scoreboard. This allows AxiStreamReceiver to use the BurstFifo as either a FIFO during GetBurst or a scoreboard for CheckBurst and TryCheckBurst. The FIFO is defined in ScoreboardPkg_slv.vhd (directory OsvvmLibraries/osvvm).

```
  type StreamRecType is record
    . . .
    -- BurstFifo
    BurstFifo       : ScoreboardIdType ;
    . . .
  end record StreamRecType ;
```

Figure 30. BurstFifo in StreamRecType

### 8.3 Interacting with the Burst FIFOs in the Test Sequencer

The BurstFifo supports 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 Stream_Model_Independent_transactions_user_guide.pdf.

#### 8.3.1 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 31 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 TxBurstFifo : ScoreboardIdType is StreamTxRec.BurstFifo ;
  alias RxBurstFifo : ScoreboardIdType is StreamRxRec.BurstFifo ;
```

Figure 31. Making the BurstFifos visible in the test sequencer (TestCtrl)

### 8.3.2 Filling the Burst FIFO

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

Note PushBurst, PushburstIncrement, and PushBurstRandom may also be used by RxBurstFifo when doing CheckBurst and TryCheckBurst transactions.

### 8.3.3 Reading and/or Checking the Burst FIFO

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

### 8.3.4 Sending Bursts via the AxiStreamTransmitter

For sending bursts with the SendBurst transaction, first items must be pushed into the BurstFIFO using the FIFO operations BurstFIFO.push, PushBurstIncrement, PushBurst, or PushBurstRandom before calling SendBurst or SendBurstAsync.   Figure 32 shows three calls to SendBurst that are similar to the ones in the test TbStream_SendGetBurst1.vhd.

```
constant WIDTH : integer := 32 ;
. . .
AxiTransmitterProc : process
begin
  . . .
  log("Transmit 32 Bytes -- word aligned") ;
  PushBurstIncrement(TxBurstFifo, 3, 32, WIDTH) ;
  SendBurst(StreamTxRec, 32) ;

  WaitForClock(StreamTxRec, 4) ;

  log("Transmit 30 Bytes -- unaligned") ;
  PushBurst(TxBurstFifo, (1,3,5,7,9,11,13,15,17,19,21,23,25,27,29), WIDTH);
  PushBurst(TxBurstFifo, (31,33,35,37,39,41,43,45,47,49,1,3,5,7,9), WIDTH);
  SendBurst(StreamTxRec, 30) ;

  WaitForClock(StreamTxRec, 4) ;

  log("Transmit 34 Bytes -- unaligned") ;
  PushBurstRandom(TxBurstFifo, 7, 34, WIDTH) ;
  SendBurst(StreamTxRec, 34) ;
```

Figure 32. SendBurst as used in TbStream_SendGetBurst1.vhd

### 8.3.5 Getting Bursts via the Receiver

When using GetBurst, first call the GetBurst or TryGetBurst transaction, and then items in the BurstFIFO can be checked using the FIFO operations BurstFIFO.check, BurstFIFO.pop, CheckBurstIncrement, CheckBurst, or CheckBurstRandom. Figure 33 shows three calls to GetBurst that are similar to the ones in the test TbStream_SendGetBurst1.vhd.

```
AxiReceiverProc : process
  variable NumBytes : integer ;
begin
  WaitForClock(StreamRxRec, 2) ;

--    log("Transmit 32 Bytes -- word aligned") ;
  GetBurst (StreamRxRec, NumBytes) ;
  AffirmIfEqual(NumBytes, 32, "Receiver: NumBytes Received") ;
  CheckBurstIncrement(RxBurstFifo, 3, NumBytes, WIDTH) ;

--    log("Transmit 30 Bytes -- unaligned") ;
  GetBurst (StreamRxRec, NumBytes) ;
  AffirmIfEqual(NumBytes, 30, "Receiver: NumBytes Received") ;
  CheckBurst(RxBurstFifo, (1,3,5,7,9,11,13,15,17,19,21,23,25,27,29), WIDTH);
  CheckBurst(RxBurstFifo, (31,33,35,37,39,41,43,45,47,49,1,3,5,7,9), WIDTH);

--    log("Transmit 34 Bytes -- unaligned") ;
  GetBurst (StreamRxRec, NumBytes) ;
  AffirmIfEqual(NumBytes, 34, "Receiver: NumBytes Received") ;
  CheckBurstRandom(RxBurstFifo, 7, NumBytes, WIDTH) ;
```

Figure 33. GetBurst as used in TbStream_SendGetBurst1.vhd

### 8.3.6 Checking Bursts in the Receiver

When using CheckBurst, first items must be pushed into the BurstFIFO with BurstFIFO.push, PushBurstIncrement, PushBurst, or PushBurstRandom, and then call CheckBurst or TryCheckBurst. Figure 34 shows three calls to CheckBurst done in the test TbStream_SendCheckBurst1.vhd.

```
AxiReceiverProc : process
  variable NumBytes : integer ;
begin
  WaitForClock(StreamRxRec, 2) ;

--    log("Transmit 32 Bytes -- word aligned") ;
  PushBurstIncrement(RxBurstFifo, 3, 32, WIDTH) ;
  CheckBurst(StreamRxRec, 32) ;

  WaitForClock(StreamRxRec, 4) ;

--    log("Transmit 30 Bytes -- unaligned") ;
  PushBurst(RxBurstFifo, (1,3,5,7,9,11,13,15,17,19,21,23,25,27,29), WIDTH);
  PushBurst(RxBurstFifo, (31,33,35,37,39,41,43,45,47,49,1,3,5,7,9), WIDTH);
  CheckBurst(StreamRxRec, 30) ;

  WaitForClock(StreamRxRec, 4) ;
```

```
--    log("Transmit 34 Bytes -- unaligned") ;
  PushBurstRandom(RxBurstFifo, 7, 34, WIDTH) ;
  CheckBurst(StreamRxRec, 34) ;
```

Figure 34. CheckBurst as used in TbStream_SendCheckBurst1.vhd

## 8.4 Interacting with the BurstFifo in the AxiStream VCs

### 8.4.1 Accessing the BurstFifo in AxiStream VC

The burst FIFOs are initialized in AxiStream VC as shown Figure 35. Note that AxiStream VC the BurstFifo is accessed as a record element of TransRec.

```
  TransactionDispatcher : process
   . . .
  begin
    wait for 0 ns ;
    TransRec.BurstFifo <= NewID("RxBurstFifo", ModelID) ;
    wait for 0 ns ;
```

Figure 35. BurstFifo Initialization

### 8.4.2 Packing and Unpacking the FIFO in the AxiStream VCs

The burst FIFOs can be configured to be either byte width or match the verification component interface width. The following procedures (from FifoFillPkg_slv.vhd) are used to transform byte width data in the burst FIFO to/from the verification component interface width.

```
PopWord  (TransRec.BurstFifo, Valid, Data, BytesToSend, [ByteAddress])
PushWord (TransRec.BurstFifo, Data, DropUndriven, [ByteAddress])
CheckWord(TransRec.BurstFifo, Data, DropUndriven, [ByteAddress])
```

# 9. About the OSVVM AxiStream VCs

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

Please support OSVVM by purchasing your VHDL training from SynthWorks.

# 10. 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.

## 11. References

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