Stream

Model Independent Transaction

User Guide

User Guide for Release 2020.10

By

Jim Lewis

SynthWorks VHDL Training

Jim@SynthWorks.com

http://www.SynthWorks.com

Table of Contents

1 Overview

The Stream Model Independent Transaction package (StreamTransactionTransactionPkg.vhd) defines a record for communication and transaction initiation procedures that are suitable for Stream Interfaces.

2 Stream Transaction Interface (StreamRecType)

The Stream Transaction Interface (StreamRecType) defines the transaction interface between the test sequencer and the verification component. As such, it is the primary channel for information exchange between the two. It is defined as follows.

The record element types, bit_max, std_logic_vector_max_c, integer_max, time_max, and boolean_max, are defined in the OSVVM package ResolutionPkg. These types allow the record to support multiple drivers and use resolution functions based on function maximum (return largest value).

```
type StreamRecType 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        : StreamOperationType ;
  -- Data and Transaction Parameter to and from verification component
  DataToModel      : std_logic_vector_max_c ;
  ParamToModel     : std_logic_vector_max_c ;
  DataFromModel    : std_logic_vector_max_c ;
  ParamFromModel   : std_logic_vector_max_c ;
  -- Verification Component 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 - currently aliased to type integer_max
  Options          : integer_max ;
end record StreamRecType ;
```

3 Usage of the Transaction Interface (StreamRecType)

The data and parameter fields of the record are unconstrained. Unconstrained objects may be used on component/entity interfaces. The record fields will be sized by the record signal that is mapped as the actual in the test harness of the testbench. Such a declaration is shown below.

```
signal AxiStreamTransmitterTransRec : StreamRecType(
    DataToModel  (AXI_DATA_WIDTH-1 downto 0),
    DataFromModel(AXI_DATA_WIDTH-1 downto 0),
    ParamToModel  (0 downto 1),                    -- Not Used for AXI Stream
    ParamFromModel(0 downto 1)                     -- Not Used for AXI Stream
```

```
) ;
```

4 Types of Transactions

A transaction may be either a directive or an interface transaction.

Directive transactions interact with the verification component without generating any transactions or interface waveforms.

An interface transaction results in interface signaling to the DUT. An interface transaction may be either blocking or non-blocking.

A blocking transaction is an interface transaction that does not does not return (complete) until the interface operation requested by the transaction has completed.

An asynchronous transaction is nonblocking interface transaction that returns before the transaction has completed - typically immediately and before the transaction has started. An asynchronous transaction has "Async" as part of its name.

A Try transaction is nonblocking interface transaction that checks to see if transaction information is available, such as read data, and if it is returns it. A Try transaction has "Try" as part of its name.

5 Directive Transactions

Directive transactions interact with the verification component without generating any transactions or interface waveforms. These transactions are supported by all verification components.

```
------------------------------------------------------------
procedure WaitForTransaction (
--  Wait until pending (transmit) or next (receive) transaction(s) complete
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType
) ;

------------------------------------------------------------
procedure WaitForClock (
-- Wait for NumberOfClocks number of clocks
-- relative to the verification component clock
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  constant  WaitCycles      : in    natural := 1
) ;

------------------------------------------------------------
procedure GetTransactionCount (
-- Get the number of transactions handled by the model.
------------------------------------------------------------
  signal    TransactionRec   : inout StreamRecType ;
  variable  TransactionCount : out   integer
) ;
```

```
------------------------------------------------------------
procedure GetAlertLogID (
-- Get the AlertLogID from the verification component.
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  variable  AlertLogID      : out   AlertLogIDType
) ;

------------------------------------------------------------
procedure GetErrorCount (
-- Error reporting for testbenches that do not use OSVVM AlertLogPkg
-- Returns error count.  If an error count /= 0, also print errors
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  variable  ErrorCount      : out   natural
) ;
```

6 Burst FIFOs and Burst Mode Controls

The burst FIFOs hold bursts of data that is to be sent to or received from the interface. The burst FIFO can be configured in the modes defined for StreamFifoBurstModeType. Currently these modes defined as a subtype of integer, shown below. The intention is to facilitate model specific extensions without the need to define separate transactions.

```
subtype StreamFifoBurstModeType is integer ;

-- Word mode indicates the burst FIFO contains interface words.
-- The size of the word is interface specific (UARTs support up
-- to 8 bits) and sometimes interface instance specific
-- (AxiStream supports interfaces sizes of 1, 2, 4, 8, 16, ... bytes)
constant STREAM_BURST_WORD_MODE       : StreamFifoBurstModeType := 0 ;

-- Word + Param mode indicates the burst FIFO contains interface
-- words plus a parameter.   The size of the parameter are interface
-- specific.   For example, for the OSVVM UART, the Param is 3 bits
-- that correspond to parity, stop, and break error injection and
-- the AxiStream uses the Param as the User field whose size is
-- interface instance specific.
constant STREAM_BURST_WORD_PARAM_MODE : StreamFifoBurstModeType := 1 ;

-- Byte mode is experimental and may be removed in a future revision.
-- Byte mode indicates that the burst FIFO contains bytes.
-- The verification component assembles interface words from the bytes.
-- This allows transfers to be conceptualized in an interface independent
-- manner.
constant STREAM_BURST_BYTE_MODE       : StreamFifoBurstModeType := 2 ;
```

SetBurstMode and GetBurstMode are directive transactions that configure the burst mode into one of the modes defined in for StreamFfifoBurstModeType.

```
------------------------------------------------------------
procedure SetBurstMode (
```

```
------------------------------------------------------------
    signal   TransRec    : InOut StreamRecType ;
    constant OptVal      : In    StreamFifoBurstModeType
  ) ;

  ------------------------------------------------------------
  procedure GetBurstMode (
  ------------------------------------------------------------
    signal   TransRec    : InOut StreamRecType ;
    variable OptVal      : Out   StreamFifoBurstModeType
  ) ;
```

7 Set and Get Model Options

Model operations are directive transactions that are used to configure the verification component. They can either be used directly or with a model specific wrapper around them - see AXI models for examples.

```
  ------------------------------------------------------------
  procedure SetModelOptions (
  ------------------------------------------------------------
    signal   TransRec    : InOut StreamRecType ;
    constant Option      : In    integer ;
    constant OptVal      : In    boolean
  ) ;

  ------------------------------------------------------------
  procedure SetModelOptions (
  ------------------------------------------------------------
    signal   TransRec    : InOut StreamRecType ;
    constant Option      : In    integer ;
    constant OptVal      : In    integer
  ) ;

  ------------------------------------------------------------
  procedure SetModelOptions (
  ------------------------------------------------------------
    signal   TransRec    : InOut StreamRecType ;
    constant Option      : In    integer ;
    constant OptVal      : In    std_logic_vector
  ) ;

  ------------------------------------------------------------
  procedure SetModelOptions (
  ------------------------------------------------------------
    signal   TransRec    : InOut StreamRecType ;
    constant Option      : In    integer ;
    constant OptVal      : In    time
  ) ;

  ------------------------------------------------------------
  procedure GetModelOptions (
  ------------------------------------------------------------
    signal   TransRec    : InOut StreamRecType ;
```

```
    constant Option      : In    integer ;
    variable OptVal      : Out   boolean
  ) ;

  ------------------------------------------------------------
  procedure GetModelOptions (
  ------------------------------------------------------------
    signal   TransRec    : InOut StreamRecType ;
    constant Option      : In    integer ;
    variable OptVal      : Out   integer
  ) ;

  ------------------------------------------------------------
  procedure GetModelOptions (
  ------------------------------------------------------------
    signal   TransRec    : InOut StreamRecType ;
    constant Option      : In    integer ;
    variable OptVal      : Out   std_logic_vector
  ) ;

  ------------------------------------------------------------
  procedure GetModelOptions (
  ------------------------------------------------------------
    signal   TransRec    : InOut StreamRecType ;
    constant Option      : In    integer ;
    variable OptVal      : Out   time
  ) ;
```

8 Transmitter Transactions

8.1 Send

Blocking Send Transaction. Param, when present, is an extra parameter used by the verification component. The UART verification component uses Param for error injection.

```
  ------------------------------------------------------------
  procedure Send (
  ------------------------------------------------------------
    signal    TransactionRec  : inout StreamRecType ;
    constant  Data            : in    std_logic_vector ;
    constant  Param           : in    std_logic_vector ;
    constant  StatusMsgOn     : in    boolean := FALSE
  ) ;

  ------------------------------------------------------------
  procedure Send (
  ------------------------------------------------------------
    signal    TransactionRec  : inout StreamRecType ;
    constant  Data            : in    std_logic_vector ;
    constant  StatusMsgOn     : in    boolean := FALSE
  ) ;
```

8.2 SendAsync

SendAsync is an asynchronous / non-blocking send transaction. Param, when present, is an extra parameter used by the verification component. The UART verification component uses Param for error injection.

```
------------------------------------------------------------
procedure SendAsync (
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  constant  Data            : in    std_logic_vector ;
  constant  Param           : in    std_logic_vector ;
  constant  StatusMsgOn     : in    boolean := FALSE
) ;

------------------------------------------------------------
procedure SendAsync (
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  constant  Data            : in    std_logic_vector ;
  constant  StatusMsgOn     : in    boolean := FALSE
) ;
```

8.3 SendBurst

SendBurst is a blocking send burst transaction. Param, when present, is an extra parameter used by the verification component. The UART verification component uses Param for error injection.

```
------------------------------------------------------------
procedure SendBurst (
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  constant  NumBytes        : In    integer ;
  constant  Param           : in    std_logic_vector ;
  constant  StatusMsgOn     : in    boolean := FALSE
) ;

------------------------------------------------------------
procedure SendBurst (
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  constant  NumBytes        : In    integer ;
  constant  StatusMsgOn     : in    boolean := FALSE
) ;
```

8.4 SendBurstAsync

SendBurstAsync is an asynchronous / non-blocking send burst transaction. Param, when present, is an extra parameter used by the verification component. The UART verification component uses Param for error injection.

```
------------------------------------------------------------
procedure SendBurstAsync (
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  constant  NumBytes        : In    integer ;
  constant  Param           : in    std_logic_vector ;
  constant  StatusMsgOn     : in    boolean := FALSE
) ;

------------------------------------------------------------
procedure SendBurstAsync (
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  constant  NumBytes        : In    integer ;
  constant  StatusMsgOn     : in    boolean := FALSE
) ;
```

9 Receiver Transactions

9.1 Get

Get is a blocking get transaction. Param, when present, is an extra parameter used by the verification component.

```
------------------------------------------------------------
procedure Get (
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  variable  Data            : out   std_logic_vector ;
  variable  Param           : out   std_logic_vector ;
  constant  StatusMsgOn     : in    boolean := FALSE
) ;

------------------------------------------------------------
procedure Get (
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  variable  Data            : out   std_logic_vector ;
  constant  StatusMsgOn     : in    boolean := FALSE
) ;
```

9.2 TryGet

TryGet is a non-blocking try get transaction. If Data is available, get it and return available TRUE, otherwise Return Available FALSE. Param, when present, is an extra parameter used by the verification component. The UART verification component uses Param for received error status.

```
------------------------------------------------------------
procedure TryGet (
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  variable  Data            : out   std_logic_vector ;
  variable  Available       : out   boolean ;
  constant  StatusMsgOn     : in    boolean := FALSE
) ;

------------------------------------------------------------
procedure TryGet (
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  variable  Data            : out   std_logic_vector ;
  variable  Param           : out   std_logic_vector ;
  variable  Available       : out   boolean ;
  constant  StatusMsgOn     : in    boolean := FALSE
) ;
```

9.3 GetBurst

GetBurst is a blocking get burst transaction. Param, when present, is an extra parameter used by the verification component. The UART verification component uses Param for checking error injection.

```
------------------------------------------------------------
procedure GetBurst (
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  variable  NumBytes        : inout integer ;
  constant  StatusMsgOn     : in    boolean := FALSE
) ;

------------------------------------------------------------
procedure GetBurst (
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  variable  NumBytes        : inout integer ;
  variable  Param           : out   std_logic_vector ;
  constant  StatusMsgOn     : in    boolean := FALSE
) ;
```

9.4 TryGetBurst

TryGetBurst is a non-blocking try get burst transaction. If Data is available, get it and return available TRUE, otherwise Return Available FALSE. Param, when present, is an extra parameter used by the verification component. The UART verification component uses Param for received error status.

```
------------------------------------------------------------
procedure TryGetBurst (
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  variable  NumBytes        : inout integer ;
  variable  Available       : out   boolean ;
  constant  StatusMsgOn     : in    boolean := FALSE
) ;

------------------------------------------------------------
procedure TryGetBurst (
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  variable  NumBytes        : inout integer ;
  variable  Param           : out   std_logic_vector ;
  variable  Available       : out   boolean ;
  constant  StatusMsgOn     : in    boolean := FALSE
) ;
```

9.5 Check

Check is a blocking check transaction. Data is the expected value to be received. Param, when present, is an extra parameter used by the verification component. The UART verification component uses Param for received error status.

```
------------------------------------------------------------
procedure Check (
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  constant  Data            : in    std_logic_vector ;
  constant  Param           : in    std_logic_vector ;
  constant  StatusMsgOn     : in    boolean := FALSE
) ;

------------------------------------------------------------
procedure Check (
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  constant  Data            : in    std_logic_vector ;
  constant  StatusMsgOn     : in    boolean := FALSE
) ;
```

9.6 TryCheck

TryCheck is a non-blocking try check transaction If Data is available, check it and return available TRUE, otherwise Return Available FALSE. Param, when present, is an extra parameter used by the verification component. The UART verification component uses Param for received error status.

```
------------------------------------------------------------
procedure TryCheck (
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  constant  Data            : in    std_logic_vector ;
  constant  Param           : in    std_logic_vector ;
  variable  Available       : out   boolean ;
  constant  StatusMsgOn     : in    boolean := FALSE
) ;

------------------------------------------------------------
procedure TryCheck (
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  constant  Data            : in    std_logic_vector ;
  variable  Available       : out   boolean ;
  constant  StatusMsgOn     : in    boolean := FALSE
) ;
```

9.7 CheckBurst

CheckBurst is a blocking check burst transaction. Param, when present, is an extra parameter used by the verification component. The UART verification component uses Param for checking error injection.

```
------------------------------------------------------------
procedure CheckBurst (
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  constant  NumBytes        : In    integer ;
  constant  Param           : in    std_logic_vector ;
  constant  StatusMsgOn     : in    boolean := FALSE
) ;

------------------------------------------------------------
procedure CheckBurst (
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  constant  NumBytes        : In    integer ;
  constant  StatusMsgOn     : in    boolean := FALSE
) ;
```

9.8 TryCheckBurst

TryCheckBurst is a non-blocking try check burst transaction. Param, when present, is an extra parameter used by the verification component. If BURST Data is available, check it and return available TRUE, otherwise Return Available FALSE. The UART verification component uses Param for checking error injection.

```
------------------------------------------------------------
procedure TryCheckBurst (
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  constant  NumBytes        : In    integer ;
  constant  Param           : in    std_logic_vector ;
  variable  Available       : out   boolean ;
  constant  StatusMsgOn     : in    boolean := FALSE
) ;

------------------------------------------------------------
procedure TryCheckBurst (
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  constant  NumBytes        : In    integer ;
  variable  Available       : out   boolean ;
  constant  StatusMsgOn     : in    boolean := FALSE
) ;
```

10 Verification Component Support Functions

Verification component support functions help decode the operation value (StreamOperationType) to determine properties about the operation.

```
------------------------------------------------------------
function IsTry (
-- True when this transaction is an asynchronous or try transaction.
-----------------------------------------------------------
  constant Operation     : in StreamOperationType
) return boolean ;

------------------------------------------------------------
function IsCheck (
-- True when this transaction is a check transaction.
-----------------------------------------------------------
  constant Operation     : in StreamOperationType
) return boolean ;
```

11 Burst FIFOs Initiator

11.1 Creating Burst FIFOs in a Verification Component

To support bursting, OSVVM verification components use the FIFOs that are part of the OSVVM generic scoreboard for bursting. This also allows the FIFO to work as a scoreboard for receiver models. The OSVVM std_logic_vector FIFO, from ScoreboardPkg_slv, can handle any size of std_logic_vector.

The FIFO is created as follows:

```
shared variable BurstFifo  : osvvm.ScoreboardPkg_slv.ScoreboardPType ;
```

11.2 Accessing Burst FIFOs from the Test Sequencer

In the test sequencer, the burst FIFO is made visible using an external name, such as the following.

```
alias TxBurstFifo is <<variable .TbAxiStream.AxiStreamTransmitter_1.BurstFifo :
                                osvvm.ScoreboardPkg_slv.ScoreboardPType>> ;
alias RxBurstFifo is <<variable .TbAxiStream.AxiStreamReceiver_1.BurstFifo :
                                osvvm.ScoreboardPkg_slv.ScoreboardPType>> ;
```

11.3 Filling the Burst FIFO from the Test Sequencer

In the test sequencer, to send a burst to the transmitter or check a burst in the transmitter, the following procedures from FifoFillPkg_slv.vhd (in osvvm_common library) may be used.

```
------------------------------------------------------------
procedure PushBurst (
-- Push each value in the Bytes parameter into the FIFO.
-- Only DataWidth bits of each value will be pushed.
------------------------------------------------------------
  variable Fifo      : inout ScoreboardPType ;
  constant Bytes     : in    integer_vector ;
  constant DataWidth : in    integer := 8
) ;

------------------------------------------------------------
procedure PushBurstIncrement (
-- Push ByteCount number of values into FIFO.  The first value
-- pushed will be Start and following values are one greater
-- than the previous one.
-- Only DataWidth bits of each value will be pushed.
------------------------------------------------------------
  variable Fifo      : inout ScoreboardPType ;
  constant Start     : in    integer ;
  constant ByteCount : in    integer ;
  constant DataWidth : in    integer := 8
) ;

------------------------------------------------------------
procedure PushBurstRandom (
-- Push ByteCount number of values into FIFO.  The first value
-- pushed will be Start and following values are randomly generated
```

```
-- using the first value as the randomization seed.
-- Only DataWidth bits of each value will be pushed.
------------------------------------------------------------
  variable Fifo      : inout ScoreboardPType ;
  constant Start     : in    integer ;
  constant ByteCount : in    integer ;
  constant DataWidth : in    integer := 8
) ;
```

11.4 Reading and/or Checking the Read Burst in the Test Sequencer

The following PopBurst and CheckBurst are used in the test sequencer to verify received burst values.

```
------------------------------------------------------------
procedure PopBurst (
-- Pop values from the FIFO into the Bytes parameter.
-- Each value popped will be DataWidth bits wide.
------------------------------------------------------------
  variable Fifo      : inout ScoreboardPType ;
  variable Bytes     : out   integer_vector ;
  constant DataWidth : in    integer := 8
) ;

------------------------------------------------------------
procedure CheckBurst (
-- Pop values from the FIFO and check them against each value
-- in the Bytes parameter.
-- Each value popped will be DataWidth bits wide.
------------------------------------------------------------
  variable Fifo      : inout ScoreboardPType ;
  constant Bytes     : in    integer_vector ;
  constant DataWidth : in    integer := 8
) ;

------------------------------------------------------------
procedure CheckBurstIncrement (
-- Pop values from the FIFO and check them against values determined
-- by an incrementing pattern.  The first check value will be Start
-- and the following check values are one greater than the previous one.
-- Each value popped will be DataWidth bits wide.
------------------------------------------------------------
  variable Fifo      : inout ScoreboardPType ;
  constant Start     : in    integer ;
  constant ByteCount : in    integer ;
  constant DataWidth : in    integer := 8
) ;

------------------------------------------------------------
procedure CheckBurstRandom (
-- Pop values from the FIFO and check them against values determined
-- by a random pattern.  The first check value will be Start and the
-- following check values are randomly generated using the first
-- value as the randomization seed.
```

```
  -- Each value popped will be DataWidth bits wide.
  ------------------------------------------------------------
    variable Fifo      : inout ScoreboardPType ;
    constant Start     : in    integer ;
    constant ByteCount : in    integer ;
    constant DataWidth : in    integer := 8
  ) ;
```

11.5 Packing and Unpacking the FIFO

A verification component can be configured to be interface width or byte width. The following procedures are used to reformat data going into or coming out of the Burst FIFO – either in the verification component or test sequencer.

```
------------------------------------------------------------
procedure PopWord (
-- Pop bytes from BurstFifo and form a word
-- Current implementation for now assumes it is assembling bytes.
--
------------------------------------------------------------
  variable Fifo              : inout ScoreboardPType ;
  variable Valid             : out   boolean ;
  variable Data              : out   std_logic_vector ;
  variable BytesToSend       : inout integer ;
  constant ByteAddress       : in    natural := 0
) ;

------------------------------------------------------------
procedure PushWord (
-- Push a word into the byte oriented BurstFifo
-- Current implementation for now assumes it is assembling bytes.
--
------------------------------------------------------------
  variable Fifo              : inout ScoreboardPType ;
  variable Data              : in    std_logic_vector ;
  constant DropUndriven      : in    boolean := FALSE ;
  constant ByteAddress       : in    natural := 0
) ;

------------------------------------------------------------
procedure CheckWord (
-- Check a word using the byte oriented BurstFifo
-- Current implementation for now assumes it is assembling bytes.
--
------------------------------------------------------------
  variable Fifo              : inout ScoreboardPType ;
  variable Data              : in    std_logic_vector ;
  constant DropUndriven      : in    boolean := FALSE ;
  constant ByteAddress       : in    natural := 0
) ;
```

11.6 Examples

The test, TbStream_SendGetBurst1.vhd, interacts with an AxiStreamTransmitter and AxiStreamReceiver.

11.6.1 Sending Bursts via the Transmitter

The following are transactions initiated by the AxiStreamTransmitter verification component (see TbStream_SendGetBurst1.vhd). .

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

AxiTransmitterProc : process
begin
  . . .
  log("Transmit 32 Bytes -- word aligned") ;
  PushBurstIncrement(TxBurstFifo, 3, 32, DATA_WIDTH) ;
  SendBurst(StreamTransmitterTransRec, 32) ;

  WaitForClock(StreamTransmitterTransRec, 4) ;

  log("Transmit 30 Bytes -- unaligned") ;
  PushBurst(TxBurstFifo, (1,3,5,7,9,11,13,15,17,19,21,23,25,27,29), DATA_WIDTH) ;
  PushBurst(TxBurstFifo, (31,33,35,37,39,41,43,45,47,49,51,53,55,57,59), DATA_WIDTH) ;
  SendBurst(StreamTransmitterTransRec, 30) ;

  WaitForClock(StreamTransmitterTransRec, 4) ;

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

11.6.2 Getting Bursts via the Receiver

The following are transactions initiated by the AxiStreamReceiver verification component (see TbStream_SendGetBurst1.vhd).

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

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

--    log("Transmit 30 Bytes -- unaligned") ;
  GetBurst (StreamReceiverTransRec, NumBytes) ;
```

```
  AffirmIfEqual(NumBytes, 30, "Receiver: NumBytes Received") ;
  CheckBurst(RxBurstFifo, (1,3,5,7,9,11,13,15,17,19,21,23,25,27,29), DATA_WIDTH) ;
  CheckBurst(RxBurstFifo, (31,33,35,37,39,41,43,45,47,49,51,53,55,57,59), DATA_WIDTH);

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

11.6.3 Checking Bursts in the Receiver

The same bursts that were read in the receiver can also be checked in the receiver (see TbStream_SendCheckBurst1.vhd).

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

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

  WaitForClock(StreamReceiverTransRec, 4) ;

--    log("Transmit 30 Bytes -- unaligned") ;
  PushBurst(RxBurstFifo, (1,3,5,7,9,11,13,15,17,19,21,23,25,27,29), FIFO_WIDTH) ;
  PushBurst(RxBurstFifo, (31,33,35,37,39,41,43,45,47,49,51,53,55,57,59), FIFO_WIDTH) ;
  CheckBurst(StreamReceiverTransRec, 30) ;

  WaitForClock(StreamReceiverTransRec, 4) ;

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

12 About the OSVVM Model Independent Transactions

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

Please support OSVVM by purchasing your VHDL training from SynthWorks.

13 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.

14 References

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