Address Bus Model Independent Transaction User Guide

User Guide for Release 2020.07

By

Jim Lewis

SynthWorks VHDL Training

Jim@SynthWorks.com

http://www.SynthWorks.com

Table of Contents

1 Overview 3
2 Address Bus Transaction Record 3
3 Usage of the Record Interface 4
4 Types of Transactions 4
5 Directive Transactions 4
6 Set and Get Model Options 5
7 Master / Initiator Transactions 6
7.1 Interface Independent Transactions 6
7.1.1 Write Transactions 6
7.1.2 Read Transactions 7
7.2 Burst Transactions 8
7.3 Interface Specific Transactions 9
7.3.1 Interface Specific Write Transactions 9
7.3.2 Interface Specific Read Transactions 9
8 Responder Transactions 11
8.1 Interface Independent Transactions 11
8.1.1 Write Transactions 11
8.1.2 Read Transactions 11
8.2 Interface Specific Transactions 12
8.2.1 Write Transactions 12
8.2.2 Read Transactions 13
9 Burst FIFOs Initiator 14
9.1 Creating Burst FIFOs in a Verification Component 14
9.2 Accessing Burst FIFOs from the Test Sequencer 15
9.3 Filling the Write Burst from the Test Sequencer 15
9.4 Reading and/or Checking the Read Burst from the Test Sequencer 16
9.5 Examples 17
10 Verification Component Support Functions 17
11 About the OSVVM Model Independent Transactions 19
12 About the Author - Jim Lewis 19
13 References 19

1 Overview

The Address Bus Model Independent Transaction package (AddressBusTransactionTransactionPkg.vhd) defines transaction interface (a record for communication between the test sequencer and verification component) and transaction initiation procedures that are suitable for Address Bus Interfaces.

2 Address Bus Transaction Record

The Address Bus Transaction Record (AddressBusTransactionRecType) 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.

```
type AddressBusTransactionRecType 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 ;
  -- 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 - currently aliased to type integer_max
  Options            : ModelOptionsType ;
end record AddressBusTransactionRecType ;
```

The record element types, bit_max, std_logic_vector_max_c, integer_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).

3 Usage of the Record Interface

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

```
signal AxiInitiatorTransRec : AddressBusTransactionRecType(
          Address      (27 downto 0),
          DataToModel  (31 downto 0),
          DataFromModel(31 downto 0)
        ) ;
```

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.

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 a non-blocking interface transaction that returns before the transaction has completed - typically immediately and before the transaction has started.

A Try transaction is non blocking interface transaction that checks to see if transaction information is available, such as read data, and if it is returns it.

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 WaitForClock (
-- Directive:  Wait for NumberOfClocks number of clocks
--             relative to the verification component clock
------------------------------------------------------------
  signal   TransRec        : InOut AddressBusTransactionRecType ;
           NumberOfClocks  : In    natural := 1
) ;

------------------------------------------------------------
procedure GetAlertLogID (
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
  variable AlertLogID  : Out   AlertLogIDType
) ;

------------------------------------------------------------
procedure GetErrors (
-- Error reporting for testbenches that do not use AlertLogPkg
-- Returns error count.  If an error count /= 0, also print errors
```

```
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
  variable ErrCnt      : Out   natural
) ;

------------------------------------------------------------
procedure GetTransactionCount (
-- Get the number of transactions handled by the model.
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
  variable Count       : Out   integer
) ;

------------------------------------------------------------
procedure GetWriteTransactionCount (
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
  variable Count       : Out   integer
) ;

------------------------------------------------------------
procedure GetReadTransactionCount (
-- Get the number of read transactions handled by the model.
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
  variable Count       : Out   integer
) ;
```

6 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 AddressBusTransactionRecType ;
  constant Option      : In    ModelOptionsType ;
  constant OptVal      : In    boolean
) ;

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

------------------------------------------------------------
procedure SetModelOptions (
------------------------------------------------------------
```

```
  signal   TransRec    : InOut AddressBusTransactionRecType ;
  constant Option      : In    ModelOptionsType ;
  constant OptVal      : In    std_logic_vector
) ;

------------------------------------------------------------
procedure GetModelOptions (
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
  constant Option      : In    ModelOptionsType ;
  variable OptVal      : Out   boolean
) ;

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

------------------------------------------------------------
procedure GetModelOptions (
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
  constant Option      : In    ModelOptionsType ;
  variable OptVal      : Out   std_logic_vector
) ;
```

7 Master / Initiator Transactions

7.1 Interface Independent Transactions

Interface Independent transactions are required to be supported by all verification components. These are recommended for all tests that verify internal design functionality.

Many are blocking transactions which do not return (complete) until the interface operation requested by the transaction has completed. Some are asynchronous, which means they return before the transaction is complete - typically even before it starts.

These transactions are supported by all verification components.

7.1.1 Write Transactions

```
------------------------------------------------------------
procedure Write (
-- Blocking Write Transaction.
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
           iAddr       : In    std_logic_vector ;
           iData       : In    std_logic_vector ;
```

```
            StatusMsgOn : In    boolean := false
    ) ;

    ------------------------------------------------------------
    procedure WriteAsync (
    -- Asynchronous / Non-Blocking Write Transaction
    ------------------------------------------------------------
      signal    TransRec    : InOut AddressBusTransactionRecType ;
                iAddr       : In    std_logic_vector ;
                iData       : In    std_logic_vector ;
                StatusMsgOn : In    boolean := false
    ) ;
```

7.1.2 Read Transactions

```
    ------------------------------------------------------------
    procedure Read (
    -- Blocking Read Transaction.
    ------------------------------------------------------------
      signal    TransRec    : InOut AddressBusTransactionRecType ;
                iAddr       : In    std_logic_vector ;
      variable  oData       : Out   std_logic_vector ;
                StatusMsgOn : In    boolean := false
    ) ;

    ------------------------------------------------------------
    procedure ReadCheck (
    -- Blocking Read Transaction and check iData, rather than returning a value.
    ------------------------------------------------------------
      signal    TransRec    : InOut AddressBusTransactionRecType ;
                iAddr       : In    std_logic_vector ;
                iData       : In    std_logic_vector ;
                StatusMsgOn : In    boolean := false
    ) ;

    ------------------------------------------------------------
    procedure ReadPoll (
    -- Read location (iAddr) until Data(IndexI) = ValueI
    -- WaitTime is the number of clocks to wait between reads.
    -- oData is the value read.
    ------------------------------------------------------------
      signal    TransRec    : InOut AddressBusTransactionRecType ;
                iAddr       : In    std_logic_vector ;
      variable  oData       : Out   std_logic_vector ;
                Index       : In    Integer ;
                BitValue    : In    std_logic ;
                StatusMsgOn : In    boolean := false ;
                WaitTime    : In    natural := 10
    ) ;

    ------------------------------------------------------------
    procedure ReadPoll (
    -- Read location (iAddr) until Data(IndexI) = ValueI
```

```
-- WaitTime is the number of clocks to wait between reads.
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
           iAddr       : In    std_logic_vector ;
           Index       : In    Integer ;
           BitValue    : In    std_logic ;
           StatusMsgOn : In    boolean := false ;
           WaitTime    : In    natural := 10
) ;
```

7.2 Burst Transactions

Some interfaces support bursting, and some do not. Hence, support for burst transactions is optional. However, for an interface that does not support bursting, it is appropriate to implement a burst as multiple single cycle operations.

```
------------------------------------------------------------
procedure WriteBurst (
-- Blocking Write Burst.
-- Data is provided separately via a WriteBurstFifo.
-- NumBytes specifies the number of bytes to be transferred.
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
           iAddr       : In    std_logic_vector ;
           NumBytes    : In    integer ;
           StatusMsgOn : In    boolean := false
) ;

------------------------------------------------------------
procedure WriteBurstAsync (
-- Asynchronous / Non-Blocking Write Burst.
-- Data is provided separately via a WriteBurstFifo.
-- NumBytes specifies the number of bytes to be transferred.
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
           iAddr       : In    std_logic_vector ;
           NumBytes    : In    integer ;
           StatusMsgOn : In    boolean := false
) ;

------------------------------------------------------------
procedure ReadBurst (
-- Blocking Read Burst.
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
           iAddr       : In    std_logic_vector ;
           NumBytes    : In    integer ;
           StatusMsgOn : In    boolean := false
) ;
```

7.3 Interface Specific Transactions

Interface specific transactions support split transaction interfaces - such as AXI which independently operates the write address, write data, write response, read address, and read data interfaces. For split transaction interfaces, these transactions are required to fully test the interface characteristics. Most of these transactions are asynchronous.

7.3.1 Interface Specific Write Transactions

```
------------------------------------------------------------
procedure WriteAddressAsync (
-- Non-blocking Write Address
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
           iAddr       : In    std_logic_vector ;
           StatusMsgOn : In    boolean := false
) ;

------------------------------------------------------------
procedure WriteDataAsync (
-- Non-blocking Write Data
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
           iAddr       : In    std_logic_vector ;
           iData       : In    std_logic_vector ;
           StatusMsgOn : In    boolean := false
) ;

------------------------------------------------------------
procedure WriteDataAsync (
-- Non-blocking Write Data.  iAddr = 0.
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
           iData       : In    std_logic_vector ;
           StatusMsgOn : In    boolean := false
) ;
```

7.3.2 Interface Specific Read Transactions

```
------------------------------------------------------------
procedure ReadAddressAsync (
-- Non-blocking Read Address
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
           iAddr       : In    std_logic_vector ;
           StatusMsgOn : In    boolean := false
) ;

------------------------------------------------------------
procedure ReadData (
-- Blocking Read Data
------------------------------------------------------------
```

```
  signal   TransRec    : InOut AddressBusTransactionRecType ;
  variable oData       : Out   std_logic_vector ;
           StatusMsgOn : In    boolean := false
) ;

------------------------------------------------------------
procedure ReadCheckData (
-- Blocking Read data and check iData, rather than returning a value.
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
           iData       : In    std_logic_vector ;
           StatusMsgOn : In    boolean := false
) ;

------------------------------------------------------------
procedure TryReadData (
-- Try (non-blocking) read data attempt.
-- If data is available, get it and return available TRUE.
-- Otherwise Return Available FALSE.
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
  variable oData       : Out   std_logic_vector ;
  variable Available   : Out   boolean ;
           StatusMsgOn : In    boolean := false
) ;

------------------------------------------------------------
procedure TryReadCheckData (
-- Try (non-blocking) read data and check attempt.
-- If data is available, check it and return available TRUE.
-- Otherwise Return Available FALSE.
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
           iData       : In    std_logic_vector ;
  variable Available   : Out   boolean ;
           StatusMsgOn : In    boolean := false
) ;
```

8 Responder Transactions

A transaction based responder verification component primarily implement register addressable devices. As such, at this time, they do not support bursting. OSVVM also provides, Memory Responder verification components, which do support burst operations to or from the internal memory. Hence, the transactions here, at this time, do not support bursting.

8.1 Interface Independent Transactions

Interface Independent transactions are required to be supported by all verification components. Interface independent transactions are intended to support testing of model internal functionality.

8.1.1 Write Transactions

```
------------------------------------------------------------
procedure GetWrite (
-- Blocking write transaction.
-- Block until the write address and data are available.
-- oData variable should be sized to match the size of the data
-- being transferred.
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
  variable oAddr       : Out   std_logic_vector ;
  variable oData       : Out   std_logic_vector ;
  constant StatusMsgOn : In    boolean := false
) ;

------------------------------------------------------------
procedure TryGetWrite (
-- Try write transaction.
-- If a write cycle has already completed return Address and Data,
-- and return Available as TRUE, otherwise, return Available as FALSE.
-- oData variable should be sized to match the size of the data
-- being transferred.
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
  variable oAddr       : Out   std_logic_vector ;
  variable oData       : Out   std_logic_vector ;
  variable Available   : Out   boolean ;
  constant StatusMsgOn : In    boolean := false
) ;
```

8.1.2 Read Transactions

```
------------------------------------------------------------
procedure SendRead (
-- Blocking Read transaction.
-- Block until address is available and data is sent.
-- iData variable should be sized to match the size of the data
-- being transferred.
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
```

```
  variable oAddr       : Out   std_logic_vector ;
  constant iData       : In    std_logic_vector ;
  constant StatusMsgOn : In    boolean := false
) ;

------------------------------------------------------------
procedure TrySendRead (
-- Try Read transaction.
-- If a read address already been received return Address,
-- send iData as the read data, and return Available as TRUE,
-- otherwise return Available as FALSE.
-- iData variable should be sized to match the size of the data
-- being transferred.
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
  variable oAddr       : Out   std_logic_vector ;
  constant iData       : In    std_logic_vector ;
  variable Available   : Out   boolean ;
  constant StatusMsgOn : In    boolean := false
) ;
```

8.2 Interface Specific Transactions

Interface specific transactions are for supporting interfaces that can dispatch independent address and data transactions.

8.2.1 Write Transactions

```
------------------------------------------------------------
procedure GetWriteAddress (
-- Blocking write address transaction.
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
  variable oAddr       : Out   std_logic_vector ;
  constant StatusMsgOn : In    boolean := false
) ;

------------------------------------------------------------
procedure TryGetWriteAddress (
-- Try write address transaction.
-- If a write address cycle has already completed return oAddr and
-- return Available as TRUE, otherwise, return Available as FALSE.
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
  variable oAddr       : Out   std_logic_vector ;
  variable Available   : Out   boolean ;
  constant StatusMsgOn : In    boolean := false
) ;

------------------------------------------------------------
procedure GetWriteData (
-- Blocking write data transaction.
```

```
-- oData should be sized to match the size of the data
-- being transferred.
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
  constant iAddr       : In    std_logic_vector ;
  variable oData       : Out   std_logic_vector ;
  constant StatusMsgOn : In    boolean := false
) ;

------------------------------------------------------------
procedure TryGetWriteData (
-- Try write data transaction.
-- If a write data cycle has already completed return oData and
-- return Available as TRUE, otherwise, return Available as FALSE.
-- oData should be sized to match the size of the data
-- being transferred.
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
  constant oAddr       : In    std_logic_vector ;
  variable oData       : Out   std_logic_vector ;
  variable Available   : Out   boolean ;
  constant StatusMsgOn : In    boolean := false
) ;

------------------------------------------------------------
procedure GetWriteData (
-- Blocking write data transaction.
-- oData should be sized to match the size of the data
-- being transferred.  iAddr = 0
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
  variable oData       : Out   std_logic_vector ;
  constant StatusMsgOn : In    boolean := false
) ;

------------------------------------------------------------
procedure TryGetWriteData (
-- Try write data transaction.
-- If a write data cycle has already completed return oData and
-- return Available as TRUE, otherwise, return Available as FALSE.
-- oData should be sized to match the size of the data
-- being transferred.  iAddr = 0
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
  variable oData       : Out   std_logic_vector ;
  variable Available   : Out   boolean ;
  constant StatusMsgOn : In    boolean := false
) ;
```

8.2.2 Read Transactions

```
------------------------------------------------------------
procedure GetReadAddress (
```

```
-- Blocking Read address transaction.
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
  variable oAddr       : Out   std_logic_vector ;
  constant StatusMsgOn : In    boolean := false
) ;

------------------------------------------------------------
procedure TryGetReadAddress (
-- Try read address transaction.
-- If a read address cycle has already completed return oAddr and
-- return Available as TRUE, otherwise, return Available as FALSE.
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
  variable oAddr       : Out   std_logic_vector ;
  variable Available   : Out   boolean ;
  constant StatusMsgOn : In    boolean := false
) ;

------------------------------------------------------------
procedure SendReadData (
-- Blocking Send Read Data transaction.
-- iData should be sized to match the size of the data
-- being transferred.
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
  constant iData       : In    std_logic_vector ;
  constant StatusMsgOn : In    boolean := false
) ;

------------------------------------------------------------
procedure AsyncSendReadData (
-- Asynchronous Send Read Data transaction.
-- iData should be sized to match the size of the data
-- being transferred.
------------------------------------------------------------
  signal   TransRec    : InOut AddressBusTransactionRecType ;
  constant iData       : In    std_logic_vector ;
  constant StatusMsgOn : In    boolean := false
) ;
```

9 Burst FIFOs Initiator

9.1 Creating Burst FIFOs in a Verification Component

To support bursting, OSVVM verification components include FIFOs for bursting. For byte oriented interfaces, the FIFOs are byte oriented. For the Axi4 full master verification component, the write and read burst FIFOs are created as follows.

```
shared variable WriteBurstFifo : osvvm.ScoreboardPkg_slv.ScoreboardPType ;
shared variable ReadBurstFifo  : osvvm.ScoreboardPkg_slv.ScoreboardPType ;
```

9.2 Accessing Burst FIFOs from the Test Sequencer

In the test sequencer, these are made visible using an external name, such as the following.

```
alias WriteBurstFifo is <<variable .TbAxi4.Axi4Initiator_1.WriteBurstFifo :
                          osvvm.ScoreboardPkg_slv.ScoreboardPType>> ;
alias ReadBurstFifo  is <<variable .TbAxi4.Axi4Initiator_1.ReadBurstFifo :
                          osvvm.ScoreboardPkg_slv.ScoreboardPType>> ;
```

9.3 Filling the Write Burst from the Test Sequencer

In the test sequencer, the WriteBurstFIFO is filled using one of the PushBurst procedures in FifoFillPkg_slv.vhd (in osvvm_common library). To keep independent of interface widths, the OSVVM AXI models use an 8 bit wide FIFO and then assemble these into the data word.

```
  ------------------------------------------------------------
  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
  ) ;
```

9.4 Reading and/or Checking the Read Burst from 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
) ;
```

9.5 Examples

The test, TbAxi4_MemoryBurst.vhd, interacts with a AXI Memory Responder. The following are transactions initiated by the AxiMaster verification component.

```
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) ;
```

10 Verification Component Support Functions

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

```
------------------------------------------------------------
function IsWriteAddress (
-- TRUE for a transaction includes write address
------------------------------------------------------------
  constant Operation     : in AddressBusOperationType
) return boolean ;

------------------------------------------------------------
function IsBlockOnWriteAddress (
-- TRUE for blocking transactions that include write address
------------------------------------------------------------
  constant Operation     : in AddressBusOperationType
) return boolean ;

------------------------------------------------------------
function IsTryWriteAddress (
-- TRUE for asynchronous or try transactions that include write address
------------------------------------------------------------
  constant Operation     : in AddressBusOperationType
) return boolean ;

------------------------------------------------------------
```

```
function IsWriteData (
-- TRUE for a transaction includes write data
-------------------------------------------------------------
  constant Operation     : in AddressBusOperationType
) return boolean ;

-------------------------------------------------------------
function IsBlockOnWriteData (
-- TRUE for a blocking transactions that include write data
-------------------------------------------------------------
  constant Operation     : in AddressBusOperationType
) return boolean ;

-------------------------------------------------------------
function IsTryWriteData (
-- TRUE for asynchronous or try transactions that include write data
-------------------------------------------------------------
  constant Operation     : in AddressBusOperationType
) return boolean ;

-------------------------------------------------------------
function IsReadAddress (
-- TRUE for a transaction includes read address
-------------------------------------------------------------
  constant Operation     : in AddressBusOperationType
) return boolean ;

-------------------------------------------------------------
function IsTryReadAddress (
-- TRUE for an asynchronous or try transactions that include read address
-------------------------------------------------------------
  constant Operation     : in AddressBusOperationType
) return boolean ;

-------------------------------------------------------------
function IsReadData (
-- TRUE for a transaction includes read data
-------------------------------------------------------------
  constant Operation     : in AddressBusOperationType
) return boolean ;

-------------------------------------------------------------
function IsBlockOnReadData (
-- TRUE for a blocking transactions that include read data
-------------------------------------------------------------
  constant Operation     : in AddressBusOperationType
) return boolean ;

-------------------------------------------------------------
function IsTryReadData (
-- TRUE for asynchronous or try transactions that include read data
-------------------------------------------------------------
  constant Operation     : in AddressBusOperationType
```

```
) return boolean ;

------------------------------------------------------------
function IsReadCheck (
-- TRUE for a transaction includes check information for read data
------------------------------------------------------------
  constant Operation     : in AddressBusOperationType
) return boolean ;

------------------------------------------------------------
function IsBurst (
-- TRUE for a transaction includes read or write burst information
------------------------------------------------------------
  constant Operation     : in AddressBusOperationType
) return boolean ;
```

11 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.

12 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.

13 References

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