Address Bus
Model Independent Transaction
User Guide

User Guide for Release 2021.08

By

Jim Lewis

SynthWorks VHDL Training

Jim@SynthWorks.com

http://www.SynthWorks.com

Table of Contents

1. Overview

The Address Bus Model Independent Transaction package (AddressBusTransactionPkg.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. Managers and Subordinates

A Manager is the agent that initiates transactions, and a Subordinate is the agent that receives and responds to requests.

3. Address Bus Transaction Record

The Address Bus Transaction Record (AddressBusRecType) 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 AddressBusRecType is record
  -- Handshaking controls
  --   Used by RequestTransaction in the Transaction Procedures
  --   Used by WaitForTransaction in the Verification Component
  --   RequestTransaction and WaitForTransaction are in osvvm.TbUtilPkg
  Rdy               : bit_max ;
  Ack               : bit_max ;
  -- Transaction Type
  Operation         : AddressBusOperationType ;
  -- Address to verification component and its width
  -- Width may be smaller than Address
  Address           : std_logic_vector_max_c ;
  AddrWidth         : integer_max ;
  -- Data to and from the verification component and its width.
  -- Width will be smaller than Data for byte operations
  -- Width size requirements are enforced in the verification component
  DataToModel       : std_logic_vector_max_c ;
  DataFromModel     : std_logic_vector_max_c ;
  DataWidth         : integer_max ;
  -- Burst FIFOs
  WriteBurstFifo    : ScoreboardIdType ;
  ReadBurstFifo     : ScoreboardIdType ;
  -- StatusMsgOn provides transaction messaging override.
  -- When true, print transaction messaging independent of
  -- other verification based based controls.
  StatusMsgOn       : boolean_max ;
  -- Verification Component Options Parameters - used by SetModelOptions
  IntToModel        : integer_max ;
  BoolToModel       : boolean_max ;
  IntFromModel      : integer_max ;
  BoolFromModel     : boolean_max ;
  -- Verification Component Options Type - currently aliased to type integer_max
  Options           : integer_max  ;
end record AddressBusRecType ;
```

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).

4. AddressBusOperationType

AddressBusOperationType is an enumerated type that indicates to the verification component type of transaction that is being dispatched. Being an enumerated type, it allows the determination of the operation in the simulator's waveform window. Table 1 shows the correlation between AddressBusOperationType values and the transaction name.

AddressBusOperationType Value	Manager Transaction Name	Subordinate Transaction Name
WAIT_FOR_CLOCK	WaitForClock	WaitForClock
WAIT_FOR_TRANSACTION	WaitForTransaction	WaitForTransaction
WAIT_FOR_WRITE_TRANSACTION	WaitForWriteTransaction	WaitForWriteTransaction
WAIT_FOR_READ_TRANSACTION	WaitForReadTransaction	WaitForReadTransaction
GET_TRANSACTION_COUNT	GetTransactionCount	GetTransactionCount
GET_WRITE_TRANSACTION_COUNT	GetWriteTransactionCount	GetWriteTransactionCount
GET_READ_TRANSACTION_COUNT	GetReadTransactionCount	GetReadTransactionCount
GET_ALERTLOG_ID	GetAlertLogID	GetAlertLogID
SET_BURST_MODE	SetBurstMode	
GET_BURST_MODE	GetBurstMode	
SET_MODEL_OPTIONS	SetModelOptions	SetModelOptions
GET_MODEL_OPTIONS	GetModelOptions	GetModelOptions
WRITE_OP	Write	GetWrite
WRITE_ADDRESS		GetWriteAddress
ASYNC_WRITE	WriteAsync	TryGetWrite
ASYNC_WRITE_ADDRESS	WriteAddressAsync	TryGetWriteAddress
WRITE_DATA		GetWriteData
ASYNC_WRITE_DATA	WriteDataAsync	TryGetWriteData
WRITE_BURST	WriteBurst	
ASYNC_WRITE_BURST	WriteBurstAsync	
READ_OP	Read	TryGetWriteData
ASYNC_READ		TrySendRead
READ_CHECK	ReadCheck	

AddressBusOperationType Value	Manager Transaction Name	Subordinate Transaction Name
READ_ADDRESS		GetReadAddress
ASYNC_READ_ADDRESS	ReadAddressAsync	TryGetReadAddress
READ_DATA	ReadData	SendReadData
READ_DATA_CHECK	ReadCheckData	
ASYNC_READ_DATA	TryReadData	SendReadDataAsync
ASYNC_READ_DATA_CHECK	TryReadCheckData	
READ_BURST	ReadBurst	

Figure 1. Correlation between AddressBusOperationType and the transaction name

5. 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 AxiManagerRec : AddressBusRecType(
          Address      (27 downto 0),
          DataToModel  (31 downto 0),
          DataFromModel(31 downto 0)
        ) ;
```

6. 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.

7. 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 transaction completes
------------------------------------------------------------
  signal    TransactionRec  : inout AddressBusRecType
) ;

------------------------------------------------------------
procedure WaitForWriteTransaction (
--  Wait until pending transaction completes
------------------------------------------------------------
  signal    TransactionRec  : inout AddressBusRecType
) ;

------------------------------------------------------------
procedure WaitForReadTransaction (
--  Wait until pending transaction completes
------------------------------------------------------------
  signal    TransactionRec  : inout AddressBusRecType
) ;

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

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

------------------------------------------------------------
procedure GetWriteTransactionCount (
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
  variable Count          : Out   integer
) ;

------------------------------------------------------------
procedure GetReadTransactionCount (
-- Get the number of read transactions handled by the model.
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
```

```
  variable Count          : Out   integer
) ;

------------------------------------------------------------
procedure GetAlertLogID (
-- Get the AlertLogID from the verification component.
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
  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 AddressBusRecType ;
  variable ErrorCount     : Out   natural
) ;
```

8. BurstMode Control Directives

The burst FIFOs hold bursts of data that is to be sent to or was 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. The intention of using integers is to facilitate model specific extensions without the need to define separate transactions.

```
subtype AddressBusFifoBurstModeType is integer ;

-- Word mode indicates the burst FIFO contains interface words.
-- The size of the word may either be interface specific (such as
-- a UART which supports up to 8 bits) or be interface instance specific
-- (such as AxiStream which supports interfaces sizes of 1, 2, 4, 8,
-- 16, ... bytes)
constant ADDRESS_BUS_BURST_WORD_MODE       : AddressBusFifoBurstModeType  := 0 ;

-- 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 ADDRESS_BUS_BURST_BYTE_MODE       : AddressBusFifoBurstModeType  := 1 ;

-- ============================================================
--  Set and Get Burst Mode
--  Set Burst Mode for models that do bursting.
-- ============================================================
------------------------------------------------------------
procedure SetBurstMode (
------------------------------------------------------------
  signal   TransactionRec  : InOut AddressBusRecType ;
  constant OptVal          : In    AddressBusFifoBurstModeType
) ;
```

```
------------------------------------------------------------
procedure GetBurstMode (
------------------------------------------------------------
  signal   TransactionRec  : InOut AddressBusRecType ;
  variable OptVal          : Out   AddressBusFifoBurstModeType
) ;

------------------------------------------------------------
function IsAddressBusBurstMode (
------------------------------------------------------------
  constant AddressBusFifoBurstMode : in AddressBusFifoBurstModeType
) return boolean ;
```

9. 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   TransactionRec : InOut AddressBusRecType ;
  constant Option         : In    Integer  ;
  constant OptVal         : In    boolean
) ;

------------------------------------------------------------
procedure SetModelOptions (
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
  constant Option         : In    Integer  ;
  constant OptVal         : In    integer
) ;

------------------------------------------------------------
procedure SetModelOptions (
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
  constant Option         : In    Integer  ;
  constant OptVal         : In    std_logic_vector
) ;

------------------------------------------------------------
procedure GetModelOptions (
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
  constant Option         : In    Integer  ;
  variable OptVal         : Out   boolean
) ;

------------------------------------------------------------
procedure GetModelOptions (
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
  constant Option         : In    Integer  ;
```

```
  variable OptVal         : Out   integer
) ;

------------------------------------------------------------
procedure GetModelOptions (
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
  constant Option         : In    Integer  ;
  variable OptVal         : Out   std_logic_vector
) ;
```

10. Manager Transactions

10.1 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.

10.1.1 Write Transactions

```
------------------------------------------------------------
procedure Write (
-- Blocking Write Transaction.
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
           iAddr          : In    std_logic_vector ;
           iData          : In    std_logic_vector ;
           StatusMsgOn    : In    boolean := false
) ;

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

10.1.2 Read Transactions

```
------------------------------------------------------------
procedure Read (
-- Blocking Read Transaction.
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
           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   TransactionRec : InOut AddressBusRecType ;
           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   TransactionRec : InOut AddressBusRecType ;
           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   TransactionRec : InOut AddressBusRecType ;
           iAddr          : In    std_logic_vector ;
           Index          : In    Integer ;
           BitValue       : In    std_logic ;
           StatusMsgOn    : In    boolean := false ;
           WaitTime       : In    natural := 10
) ;
```

10.2 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.
-- NumFifoWords specifies the number of items from the FIFO to be transferred.
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
           iAddr          : In    std_logic_vector ;
           NumFifoWords   : In    integer ;
```

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

    ------------------------------------------------------------
    procedure WriteBurstAsync (
    -- Asynchronous / Non-Blocking Write Burst.
    -- Data is provided separately via a WriteBurstFifo.
    -- NumFifoWords specifies the number of items from the FIFO to be transferred.
    ------------------------------------------------------------
      signal   TransactionRec : InOut AddressBusRecType ;
               iAddr          : In    std_logic_vector ;
               NumFifoWords   : In    integer ;
               StatusMsgOn    : In    boolean := false
    ) ;

    ------------------------------------------------------------
    procedure ReadBurst (
    -- Blocking Read Burst.
    -- NumFifoWords specifies the number of items from the FIFO to be transferred.
    ------------------------------------------------------------
      signal   TransactionRec : InOut AddressBusRecType ;
               iAddr          : In    std_logic_vector ;
               NumFifoWords   : In    integer ;
               StatusMsgOn    : In    boolean := false
    ) ;
```

10.3 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.

10.3.1 Interface Specific Write Transactions

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

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

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

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

10.3.2 Interface Specific Read Transactions

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

------------------------------------------------------------
procedure ReadData (
-- Blocking Read Data
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
  variable oData          : Out   std_logic_vector ;
           StatusMsgOn    : In    boolean := false
) ;

------------------------------------------------------------
procedure ReadCheckData (
-- Blocking Read data and check iData, rather than returning a value.
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
           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   TransactionRec : InOut AddressBusRecType ;
  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   TransactionRec : InOut AddressBusRecType ;
           iData          : In    std_logic_vector ;
  variable Available      : Out   boolean ;
           StatusMsgOn    : In    boolean := false
) ;
```

11.Subordinate Transactions

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

11.1 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.

11.1.1 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   TransactionRec : InOut AddressBusRecType ;
  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   TransactionRec : InOut AddressBusRecType ;
  variable oAddr          : Out   std_logic_vector ;
  variable oData          : Out   std_logic_vector ;
  variable Available      : Out   boolean ;
  constant StatusMsgOn    : In    boolean := false
) ;
```

11.1.2 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   TransactionRec : InOut AddressBusRecType ;
  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   TransactionRec : InOut AddressBusRecType ;
  variable oAddr          : Out   std_logic_vector ;
  constant iData          : In    std_logic_vector ;
  variable Available      : Out   boolean ;
  constant StatusMsgOn    : In    boolean := false
) ;
```

11.2 Interface Specific Transactions

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

11.2.1 Write Transactions

```
------------------------------------------------------------
procedure GetWriteAddress (
-- Blocking write address transaction.
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
  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   TransactionRec : InOut AddressBusRecType ;
  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   TransactionRec : InOut AddressBusRecType ;
    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   TransactionRec : InOut AddressBusRecType ;
    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   TransactionRec : InOut AddressBusRecType ;
    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   TransactionRec : InOut AddressBusRecType ;
    variable oData          : Out   std_logic_vector ;
    variable Available      : Out   boolean ;
    constant StatusMsgOn    : In    boolean := false
  ) ;
```

11.2.2 Read Transactions

```
  ------------------------------------------------------------
  procedure GetReadAddress (
  -- Blocking Read address transaction.
  ------------------------------------------------------------
    signal   TransactionRec : InOut AddressBusRecType ;
    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   TransactionRec : InOut AddressBusRecType ;
  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   TransactionRec : InOut AddressBusRecType ;
  constant iData          : In    std_logic_vector ;
  constant StatusMsgOn    : In    boolean := false
) ;

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

12. Burst FIFOs Manager

12.1 BurstFifo is in the Interface

The WriteBurstFifo and ReadBurstFifo is inside AddressBusRecType, see Figure 2. This makes the BurstFifo easily accessible to both the Verification component as well as the Test Sequencer (TestCtrl). The BurstFifo is implemented using a ScoreboardID from the scoreboard package. This allows a VC to either use it as a FIFO or as a scoreboard. The FIFO is std_logic_vector based and uses the OSVVM library ScoreboardGenericPkg instance defined in ScoreboardPkg_slv.vhd (OsvvmLibraries/osvvm).

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

Figure 2. BurstFifo In AddressBusRecType

12.2 Using the Burst FIFO in the Manager VC

12.2.1 Initializing Burst FIFOs

The burst FIFOs need to be initialized. A good place to do this is in the transaction dispatcher of the verification components. Figure 3 shows the declaration of a BurstFifo.

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

Figure 3. BurstFifo Initialization

12.2.2 Accessing Burst FIFOs

The Burst Fifos support basic FIFO operations. These are shown in Figure 4.

```
Push(TransRec.ReadBurstFifo, Data) ;
Check(TransRec.ReadBurstFifo, Data) ;
Data := Pop(TransRec.WriteBurstFifo) ;
```

Figure 4. Making the BurstFifos visible in the test sequencer (TestCtrl)

12.2.3 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.
------------------------------------------------------------
  constant Fifo              : in    ScoreboardIDType ;
  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.
------------------------------------------------------------
  constant Fifo              : in    ScoreboardIDType ;
  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.
------------------------------------------------------------
  constant Fifo              : in    ScoreboardIDType ;
  variable Data              : in    std_logic_vector ;
  constant DropUndriven      : in    boolean := FALSE ;
  constant ByteAddress       : in    natural := 0
) ;
```

12.3 Using the Burst FIFO in the Test Sequencer

12.3.1 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 VectorOfWords parameter into the FIFO.
-- Only FifoWidth bits of each value will be pushed.
------------------------------------------------------------
  constant Fifo           : in    ScoreboardIDType ;
  constant VectorOfWords  : in    integer_vector ;
  constant FifoWidth      : in    integer := 8
) ;

------------------------------------------------------------
procedure PushBurstIncrement (
-- Push Count number of values into FIFO.  The first value
```

```
-- pushed will be FirstWord and following values are one greater
-- than the previous one.
-- Only FifoWidth bits of each value will be pushed.
------------------------------------------------------------
  constant Fifo      : in    ScoreboardIDType ;
  constant FirstWord : in    integer ;
  constant Count     : in    integer ;
  constant FifoWidth : in    integer := 8
) ;

------------------------------------------------------------
procedure PushBurstRandom (
-- Push Count number of values into FIFO.  The first value
-- pushed will be FirstWord and following values are randomly generated
-- using the first value as the randomization seed.
-- Only FifoWidth bits of each value will be pushed.
------------------------------------------------------------
  constant Fifo      : in    ScoreboardIDType ;
  constant FirstWord : in    integer ;
  constant Count     : in    integer ;
  constant FifoWidth : in    integer := 8
) ;
```

12.3.2 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 VectorOfWords parameter.
-- Each value popped will be FifoWidth bits wide.
------------------------------------------------------------
  constant Fifo           : in    ScoreboardIDType ;
  variable VectorOfWords  : out   integer_vector ;
  constant FifoWidth      : in    integer := 8
) ;

------------------------------------------------------------
procedure CheckBurst (
-- Pop values from the FIFO and check them against each value
-- in the VectorOfWords parameter.
-- Each value popped will be FifoWidth bits wide.
------------------------------------------------------------
  constant Fifo           : in    ScoreboardIDType ;
  constant VectorOfWords  : in    integer_vector ;
  constant FifoWidth      : 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 FirstWord
-- and the following check values are one greater than the previous one.
-- Each value popped will be FifoWidth bits wide.
```

```
  ------------------------------------------------------------
    constant Fifo      : in    ScoreboardIDType ;
    constant FirstWord : in    integer ;
    constant Count     : in    integer ;
    constant FifoWidth : 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 FirstWord and the
  -- following check values are randomly generated using the first
  -- value as the randomization seed.
  -- Each value popped will be FifoWidth bits wide.
  ------------------------------------------------------------
    constant Fifo      : in    ScoreboardIDType ;
    constant FirstWord : in    integer ;
    constant Count     : in    integer ;
    constant FifoWidth : in    integer := 8
  ) ;
```

12.3.3 Examples

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

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

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

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

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

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

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

13. 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 ;
```

14. 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.

15. 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.

16. References

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