Address Bus Model Independent Transaction User Guide

User Guide for Release 2022.01

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 a transaction interface (a record for communication between the test sequencer and the verification component) and transaction initiation procedures that are suitable for Address Bus Interfaces.

All verification components (VCs) that use this interface, support a common set of transactions (or API). Hence, a test writer who understands the transactions for one verification component will also understand the transactions of another verification component that also uses the same package. This makes the job of a verification engineer easier.

The main difference between verification components that support this package is the configuration of verification component interface specific features – such as settings for AWUser and AWProt for AXI4.

Having a shared set of transactions improves test case reuse between different verification components – a case where reuse is rarely possible with other verification methodologies.

2. Managers and Subordinates

A Manager is the agent that initiates transactions, and a Subordinate is the agent that receives and responds to requests. On some address bus interfaces the Manager is also called an initiator and the subordinate is called the target.

3. Address Bus Transaction Interface

The Address Bus Transaction interface connects the test sequencer to a verification component. As such, it is the primary channel for information exchange between the two. This is done with the record type AddressBusRecType.

3.1 AddressBusRecType

```
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                : RdyType ;
  Ack                : AckType ;
  -- 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.
  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.
  StatusMsgOn        : boolean_max ;
  -- Verification Component Options Parameters - used by SetModelOptions
  IntToModel         : integer_max ;
  IntFromModel       : integer_max ;
  BoolToModel        : boolean_max ;
  BoolFromModel      : boolean_max ;
  -- Verification Component Options Type
  Options            : integer_max ;
end record AddressBusRecType ;
```

One of the challenges of using a single record, such as AddressBusRecType, as an interface is dealing with multiple drivers on each record element. OSVVM does this giving each element a resolved type, such as bit_max, std_logic_vector_max_c, integer_max, time_max, and boolean_max. These 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).

The type AddressBusOperationType is discussed in VC section of this document.

3.2 BurstFifo is in the Interface

The WriteBurstFifo and ReadBurstFifo are inside AddressBusRecType. This makes the burst FIFOs easily accessible to both the verification component and the Test Sequencer (TestCtrl). The type, ScoreboardIdType, is a reference (though not an access type) to the scoreboard singleton data structure in ScoreboardGenericPkg. Using ScoreboardIdType allows the structure to be used as either a FIFO (by SendBurst or GetBurst) or a Scoreboard (by CheckBurst). The FIFO is std_logic_vector based and uses the ScoreboardPkg_slv instance from OsvvmLibraries/osvvm.

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

4. Running the Address Bus Demo

The best way to learn is by trying things out as you go. In this step you will download OSVVM, build the libraries, and then run the demo. Code from the demo is shown in examples in the respective sections.

OSVVM is available on GitHub at https://github.com/OSVVM as a git repository or at https://osvvm.org/downloads as a ZIP file. Retrieve OSVVM from GitHub using git as shown in Figure 1. Note that the "—recursive" option is required since the OSVVM repositories are submodules of OsvvmLibraries. Submodules greatly simplify development and deployment of the libraries.

```
git clone --recursive https://github.com/OSVVM/OsvvmLibraries.git
```

Figure 1. Retrieving OSVVM from GitHub

Prior to starting the OSVVM scripting environment, create a directory named sim in which to run your simulations. Start your simulator and go to the sim directory. Once there, use the steps in Figure 2 to build the OSVVM Libraries (utility and verification component). These directions are supported in Mentor QuestaSim/ModelSim or Aldec RivieraPRO. Aldec's ActiveHDL, Synopsys' VCS, and Cadence's Xcelium are also supported but require a few extra steps. For these steps and additional details of the OSVVM scripting environment see Script_user_guide.pdf (in OsvvmLibraries/Documentation).

```
cd sim
source ../OsvvmLibraries/Scripts/StartUp.tcl
build  ../OsvvmLibraries
build  ../OsvvmLibraries/AXI4/Axi4/RunDemoTests.pro
```

Figure 2. Building OSVVM and running the Demo

The intent of the OSVVM scripting is to make compiling and running your simulations independent of the simulator you are using.

GHDL can be run using tclsh. In windows, using MSYS2/MinGW64 start tclsh using "winpty tclsh".

5. 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. Examples of these are WaitForClock and GetAlertLogID.

An interface transaction results in interface signaling to the DUT. An interface transaction may be either blocking (such as Write or Read) or non-blocking (such as WriteAsync and TryReadData).

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.

An interface independent transaction can be implemented by any interface. An interface independent transaction contains all the information needed for an interface operation.

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. An interface specific transaction contains part of the information for an interface operation and is combined with other interface specific transactions to create an interface operation. For a split transaction interface, this allows different interface timing relationships to be created.

Interface specific transactions are only appropriate for split transaction interfaces and should not be implemented for interfaces that do not support this capability. Hence, they are not portable between different address bus verification components.

For general testing be sure to use interface independent transactions as they can be supported by all address bus interfaces. Use interface specific transactions to test interface characteristics.

6. Manager Transactions

Manager transactions supply the necessary transaction information so a manager verification component can initiate an address bus cycle.

6.1 Basic Interface Independent Transactions

6.1.1 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
) ;
```

6.1.2 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
  ) ;
```

6.1.3 Examples

The following code does 16 Write operations followed by 16 Read operations.

```
  ManagerProc : process
    variable RxData    : std_logic_vector(31 downto 0) ;
  begin
    . . .
    for I in 1 to 16 loop
      Write( ManagerRec, X"0000_0000" + 16*I, X"0000_0000" + I ) ;
    end loop ;

    for I in 1 to 16 loop
      Read ( ManagerRec, X"0000_0000" + 16*I, RxData) ;
      AffirmIfEqual(RxData, X"0000_0000" + I, "Read Data " ) ;
    end loop ;
```

The following code does 16 Write operations followed by 16 ReadCheck operations.

```
    for I in 1 to 16 loop
      Write( ManagerRec, X"0000_1000" + 16*I, X"0000_1000" + I ) ;
    end loop ;

    for I in 1 to 16 loop
      ReadCheck ( ManagerRec, X"0000_1000" + 16*I, X"0000_1000" + I ) ;
    end loop ;
```

6.2 Interface Independent Burst Transactions

Some interfaces support bursting, and some do not. Hence, support for burst transactions is optional. For an interface that does not support bursting, a burst may logically supported and implemented as multiple single cycle operations.

6.2.1 WriteBurst

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

6.2.2 ReadBurst

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

6.2.3 Example

The following code does 16-word bursts. FIFO interaction is done separately with push and CheckExpected.

```
ManagerProc : process
begin
  . . .
  for I in 1 to 16 loop
    Push( ManagerRec.WriteBurstFifo, X"0000_2000" + I  ) ;
  end loop ;
  WriteBurst(ManagerRec, X"0000_2000", 16) ;

  ReadBurst(ManagerRec, X"0000_2000", 16) ;
  for I in 1 to 16 loop
    CheckExpected( ManagerRec.ReadBurstFifo, X"0000_2000" + I  ) ;
  end loop ;
```

6.3 Patterns for Interface Independent Burst Transactions

6.3.1 Burst with Vector of Words

Burst transactions that accept an array of std_logic_vector.

```
------------------------------------------------------------
procedure WriteBurstVector (
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
           iAddr          : In    std_logic_vector ;
           VectorOfWords  : In    slv_vector ;
           StatusMsgOn    : In    boolean := false
) ;

------------------------------------------------------------
procedure WriteBurstVectorAsync (
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
           iAddr          : In    std_logic_vector ;
           VectorOfWords  : In    slv_vector ;
           StatusMsgOn    : In    boolean := false
) ;
```

```
------------------------------------------------------------
procedure ReadCheckBurstVector (
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
           iAddr          : In    std_logic_vector ;
           VectorOfWords  : In    slv_vector ;
           StatusMsgOn    : In    boolean := false
) ;
```

The following code does 13-word bursts using a vector pattern.

```
ManagerProc : process
  constant DATA_ZERO : std_logic_vector := (31 downto 0 => '0') ;
begin
  . . .
  WriteBurstVector(ManagerRec, X"0000_3000",
      (X"0001_UUUU", DATA_ZERO+3,  DATA_ZERO+5,  DATA_ZERO+7,  DATA_ZERO+9,
      DATA_ZERO+11,  DATA_ZERO+13, DATA_ZERO+15, DATA_ZERO+17, DATA_ZERO+19,
      DATA_ZERO+21,  DATA_ZERO+23, DATA_ZERO+25) ) ;

  ReadCheckBurstVector(ManagerRec, X"0000_3000",
      (X"0001_----", DATA_ZERO+3,  DATA_ZERO+5,  DATA_ZERO+7,  DATA_ZERO+9,
      DATA_ZERO+11,  DATA_ZERO+13, DATA_ZERO+15, DATA_ZERO+17, DATA_ZERO+19,
      DATA_ZERO+21,  DATA_ZERO+23, DATA_ZERO+25) ) ;
```

6.3.2 **Burst with an Incrementing Pattern**

Burst transactions that the first word as a std_logic_vector and additional words that are one larger in value.

```
------------------------------------------------------------
procedure WriteBurstIncrement (
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
           iAddr          : In    std_logic_vector ;
           FirstWord      : In    std_logic_vector ;
           NumFifoWords   : In    integer ;
           StatusMsgOn    : In    boolean := false
) ;

------------------------------------------------------------
procedure WriteBurstIncrementAsync (
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
           iAddr          : In    std_logic_vector ;
           FirstWord      : In    std_logic_vector ;
           NumFifoWords   : In    integer ;
           StatusMsgOn    : In    boolean := false
) ;

------------------------------------------------------------
procedure ReadCheckBurstIncrement (
------------------------------------------------------------
```

```
    signal   TransactionRec : InOut AddressBusRecType ;
             iAddr          : In    std_logic_vector ;
             FirstWord      : In    std_logic_vector ;
             NumFifoWords   : In    integer ;
             StatusMsgOn    : In    boolean := false
  ) ;
```

The following code does 12-word bursts with an incrementing pattern.

```
  ManagerProc : process
  begin
    . . .
    WriteBurstIncrement    (ManagerRec, X"0000_4000", X"0000_4000"+3, 12) ;

    ReadCheckBurstIncrement(ManagerRec, X"0000_4000", X"0000_4000"+3, 12) ;
```

6.3.3 **Burst with a Random Pattern**

Burst transactions that the first word as a std_logic_vector and additional words that are one larger in value.

```
  ------------------------------------------------------------
  procedure WriteBurstRandom (
  ------------------------------------------------------------
    signal   TransactionRec : InOut AddressBusRecType ;
             iAddr          : In    std_logic_vector ;
             FirstWord      : In    std_logic_vector ;
             NumFifoWords   : In    integer ;
             StatusMsgOn    : In    boolean := false
  ) ;
  ------------------------------------------------------------
  procedure WriteBurstRandomAsync (
  ------------------------------------------------------------
    signal   TransactionRec : InOut AddressBusRecType ;
             iAddr          : In    std_logic_vector ;
             FirstWord      : In    std_logic_vector ;
             NumFifoWords   : In    integer ;
             StatusMsgOn    : In    boolean := false
  ) ;

  ------------------------------------------------------------
  procedure ReadCheckBurstRandom (
  ------------------------------------------------------------
    signal   TransactionRec : InOut AddressBusRecType ;
             iAddr          : In    std_logic_vector ;
             FirstWord      : In    std_logic_vector ;
             NumFifoWords   : In    integer ;
             StatusMsgOn    : In    boolean := false
  ) ;
```

The following code does 13-word bursts with a random pattern.

```
  ManagerProc : process
```

```
begin
  . . .
  WriteBurstRandom    (ManagerRec, X"0000_5001", X"A015_2800", 13) ;

  ReadCheckBurstRandom(ManagerRec, X"0000_5001", X"A015_28UU", 13) ;
```

6.3.4 Burst with an Intelligent Coverage Random Pattern

Burst transactions that the first word as a std_logic_vector and additional words that are one larger in value.

```
------------------------------------------------------------
procedure WriteBurstRandom (
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
           iAddr          : In    std_logic_vector ;
           CoverID        : In    CoverageIDType ;
           NumFifoWords   : In    integer ;
           FifoWidth      : In    integer ;
           StatusMsgOn    : In    boolean := false
) ;
------------------------------------------------------------
procedure WriteBurstRandomAsync (
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
           iAddr          : In    std_logic_vector ;
           CoverID        : In    CoverageIDType ;
           NumFifoWords   : In    integer ;
           FifoWidth      : In    integer ;
           StatusMsgOn    : In    boolean := false
) ;
------------------------------------------------------------
procedure ReadCheckBurstRandom (
------------------------------------------------------------
  signal   TransactionRec : InOut AddressBusRecType ;
           iAddr          : In    std_logic_vector ;
           CoverID        : In    CoverageIDType ;
           NumFifoWords   : In    integer ;
           FifoWidth      : In    integer ;
           StatusMsgOn    : In    boolean := false
) ;
```

The following does 12-word bursts with an Intelligent Coverage Random pattern.

```
ManagerProc : process
  variable CoverID1, CoverID2 : CoverageIdType ;
begin
  . . .
  CoverID1 := NewID("Cov1") ;
  InitSeed(CoverID1, 5) ; -- Start Write and Read with the same seed value
  AddBins (CoverID1, 1,
      GenBin(0,7) & GenBin(32,39) & GenBin(64,71) & GenBin(96,103)) ;
  WriteBurstRandom(    ManagerRec, X"0000_6000", CoverID1, 12, DATA_WIDTH) ;
```

```
    CoverID2 := NewID("Cov2") ;
    InitSeed(CoverID2, 5) ; -- Start Write and Read with the same seed value
    AddBins (CoverID2, 1,
        GenBin(0,7) & GenBin(32,39) & GenBin(64,71) & GenBin(96,103)) ;
    ReadCheckBurstRandom(ManagerRec, X"0000_6000", CoverID2, 12, DATA_WIDTH) ;
```

6.4 FIFO Fill and Check Patterns and Composite Bursts

FIFO fill patterns can be used in conjunction with WriteBurst to build burst transactions that are a composite of Vector, Increment, and Random patterns. Likewise FIFO check patterns can be used with GetBurst in a similar fashion.

FIFO fill and check patterns are implemented in FifoFillPkg_slv.vhd (in the osvvm_common library).

6.4.1 Basic FIFO Operations

The Burst Fifos support basic FIFO operations. These are shown in Figure 3.

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

Figure 3. Basic FIFO Operations

6.4.2 FIFO Fill Patterns

6.4.2.1 Fill with a Vector of Words

```
  ------------------------------------------------------------
  procedure PushBurstVector (
  -- Push each value in the VectorOfWords parameter into the FIFO.
  -- FifoWidth must match the std_logic_vector parameter.
  ------------------------------------------------------------
    constant Fifo          : In    ScoreboardIdType ;
    constant VectorOfWords : In    slv_vector
  ) ;
  ------------------------------------------------------------
  procedure PushBurstVector (
  -- 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
  ) ;
```

6.4.2.2 Fill with an Incrementing Pattern

```
  ------------------------------------------------------------
  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.
-- FifoWidth must match the std_logic_vector parameter.
------------------------------------------------------------
  constant Fifo           : In    ScoreboardIdType ;
  constant FirstWord      : In    std_logic_vector ;
  constant Count          : In    integer
) ;
------------------------------------------------------------
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
) ;
```

6.4.2.3 Fill with a Random Pattern

```
------------------------------------------------------------
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.
-- FifoWidth must match the std_logic_vector parameter.
------------------------------------------------------------
  constant Fifo           : In    ScoreboardIdType ;
  constant FirstWord      : In    std_logic_vector ;
  constant Count          : In    integer
) ;
------------------------------------------------------------
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
) ;
```

6.4.2.4 Fill with an Intelligent Coverage Random Pattern

```
------------------------------------------------------------
-- Experimental and Provisional
procedure PushBurstRandom (
-- Push Count number of values into FIFO.  Values are
```

```
  -- randomly generated using the coverage model.
  -- Only FifoWidth bits of each value will be pushed.
  ------------------------------------------------------------
    constant Fifo         : in    ScoreboardIdType ;
    constant CoverID      : in    CoverageIdType ;
    constant Count        : in    integer ;
    constant FifoWidth    : in    integer := 8
  ) ;
```

6.4.3 FIFO Check Patterns

6.4.3.1 Check with a Vector of Words

```
    ------------------------------------------------------------
  procedure CheckBurstVector (
  -- Check values from the FIFO against the values
  -- in the VectorOfWords parameter.
  -- Width of VectorOfWords(i) shall match the width of the Fifo
  ------------------------------------------------------------
    constant Fifo           : in    ScoreboardIdType ;
    constant VectorOfWords  : in    slv_vector
  ) ;

  ------------------------------------------------------------
  procedure CheckBurstVector (
  -- Check values from the FIFO against the values
  -- in the VectorOfWords parameter.
  -- Each value of VectorOfWords shall be converted to FifoWidth bits wide.
  ------------------------------------------------------------
    constant Fifo           : in    ScoreboardIdType ;
    constant VectorOfWords  : in    integer_vector ;
    constant FifoWidth      : in    integer
  ) ;
```

6.4.3.2 Check with an Incrementing Pattern

```
  ------------------------------------------------------------
  procedure CheckBurstIncrement (
  -- Check values from the FIFO against the incrementing values
  -- that start with the value of the FirstWord.
  -- Width of FirstWord shall match the width of the Fifo
  ------------------------------------------------------------
    constant Fifo         : in    ScoreboardIdType ;
    constant FirstWord    : in    std_logic_vector ;
    constant Count        : in    integer
  ) ;

  ------------------------------------------------------------
  procedure CheckBurstIncrement (
  -- Check values from the FIFO against the incrementing values
  -- that start with the value of the FirstWord.
  -- Each value of VectorOfWords shall be converted to FifoWidth bits wide.
```

```
------------------------------------------------------------
    constant Fifo         : in    ScoreboardIdType ;
    constant FirstWord    : in    integer ;
    constant Count        : in    integer ;
    constant FifoWidth    : in    integer := 8
  ) ;
```

6.4.3.3 **Check with a Random Pattern**

```
------------------------------------------------------------
  procedure CheckBurstRandom (
  -- Check values from the FIFO against the random values
  -- that are generated using the value of the FirstWord and
  -- NumFifoWords as the randomization seeds.
  -- Width of FirstWord shall match the width of the Fifo
------------------------------------------------------------
    constant Fifo         : in    ScoreboardIdType ;
    constant FirstWord    : in    std_logic_vector ;
    constant Count        : in    integer
  ) ;

------------------------------------------------------------
  procedure CheckBurstRandom (
  -- Check values from the FIFO against the random values
  -- that are generated using the value of the FirstWord and
  -- Count as the randomization seeds.
  -- Each value of VectorOfWords shall be converted to FifoWidth bits wide.
------------------------------------------------------------
    constant Fifo         : in    ScoreboardIdType ;
    constant FirstWord    : in    integer ;
    constant Count        : in    integer ;
    constant FifoWidth    : in    integer := 8
  ) ;
```

6.4.3.4 **Check with an Intelligent Coverage Random Pattern**

```
------------------------------------------------------------
  -- Experimental and Provisional
  procedure CheckBurstRandom (
------------------------------------------------------------
    constant Fifo         : in    ScoreboardIdType ;
    constant CoverID      : in    CoverageIdType ;
    constant Count        : in    integer ;
    constant FifoWidth    : in    integer := 8
  ) ;
```

6.4.4 **Example: PushBurst and CheckBurst**

The following does 42-word bursts using a composite of Vector, Increment, Random, and Intelligent Coverage Random patterns.

```
    PushBurstVector(ManagerRec.WriteBurstFifo,
        (X"0000_B001", X"0000_B003", X"0000_B005", X"0000_B007", X"0000_B009",
         X"0000_B011", X"0000_B013", X"0000_B015", X"0000_B017", X"0000_B019") ) ;
    PushBurstIncrement(ManagerRec.WriteBurstFifo, X"0000_B100", 10) ;
    PushBurstRandom(ManagerRec.WriteBurstFifo, X"0000_B200", 6) ;
    CoverID1 := NewID("Cov1b") ;
    InitSeed(CoverID1, 5) ; -- Start Write and Read with the same seed value
    AddBins(CoverID1, 1,
        GenBin(16#B300#, 16#B307#) & GenBin(16#B310#, 16#B317#) &
        GenBin(16#B320#, 16#B327#) & GenBin(16#B330#, 16#B337#)) ;
    PushBurstRandom(ManagerRec.WriteBurstFifo, CoverID1, 16, 32) ;
    WriteBurst(ManagerRec, X"0000_B000", 42) ;

    ReadBurst (ManagerRec, X"0000_B000", 42) ;
    CheckBurstVector(ManagerRec.ReadBurstFifo,
        (X"0000_B001", X"0000_B003", X"0000_B005", X"0000_B007", X"0000_B009",
         X"0000_B011", X"0000_B013", X"0000_B015", X"0000_B017", X"0000_B019") ) ;
    CheckBurstIncrement(ManagerRec.ReadBurstFifo, X"0000_B100", 10) ;
    CheckBurstRandom(ManagerRec.ReadBurstFifo, X"0000_B200", 6) ;
    CoverID2 := NewID("Cov2b") ;
    InitSeed(CoverID2, 5) ; -- Start Write and Read with the same seed value
    AddBins(CoverID2, 1,
        GenBin(16#B300#, 16#B307#) & GenBin(16#B310#, 16#B317#) &
        GenBin(16#B320#, 16#B327#) & GenBin(16#B330#, 16#B337#)) ;
    CheckBurstRandom(ManagerRec.ReadBurstFifo, CoverID2, 16, 32) ;
```

6.4.5 FIFO Pop Burst

6.4.5.1 PopBurstVector

PopBurstVector returns the contents of the FIFO as a vector.   The vector can either be an array of std_logic_vector (slv_vector) or integer_vector.

```
  ------------------------------------------------------------
  procedure PopBurstVector (
  -- Pop values from the FIFO into the VectorOfWords parameter.
  -- Width of VectorOfWords(i) shall match the width of the Fifo
  ------------------------------------------------------------
    constant Fifo           : in    ScoreboardIdType ;
    variable VectorOfWords  : out   slv_vector
  ) ;

  ------------------------------------------------------------
  procedure PopBurstVector (
  -- 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
  ) ;
```

6.4.5.2 Example: PopBurstVector

The following does 5-word bursts using PopBurstVector and slv_vector.

```
    WriteBurstVector(ManagerRec, X"0000_C000",
        (X"0000_C001", X"0000_C003", X"0000_C005", X"0000_C007", X"0000_C009") ) ;

    ReadBurst(ManagerRec, X"0000_C000", 5) ;
    PopBurstVector(ManagerRec.ReadBurstFifo, slvBurstVector) ;
    AffirmIf(slvBurstVector =
        (X"0000_C001", X"0000_C003", X"0000_C005", X"0000_C007", X"0000_C009"),
        "slvBurstVector = C001, C003, C005, C007, C009") ;
```

The following does 5-word bursts using PopBurstVector and integer_vector.

```
    PushBurstVector(ManagerRec.WriteBurstFifo,
        (16#D001#, 16#D003#, 16#D005#, 16#D007#, 16#D009#), 32 ) ;
    WriteBurst(ManagerRec, X"0000_D000", 5) ;

    ReadBurst(ManagerRec, X"0000_D000", 5) ;
    PopBurstVector(ManagerRec.ReadBurstFifo, intBurstVector) ;
    AffirmIf(intBurstVector =
        (16#D001#, 16#D003#, 16#D005#, 16#D007#, 16#D009#),
        "slvBurstVector = D001, D003, D005, D007, D009") ;
```

6.5 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. Interface specific transactions provide a transaction for each independent aspect of the split transaction interface that can be called independently. This allows a WriteDataAsync transaction to be dispatched and then perhaps 4 clocks later a WriteAddressAsync transaction to be dispatched. This gives the test writer direct control of all aspects of the interface and facilitates testing of all interface characteristics. Most of these transactions are asynchronous.

6.5.1 Interface Specific Write Transactions

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

6.5.2 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
) ;
```

7. Subordinate Transactions

Subordinate transactions supply the necessary transaction information so a subordinate verification component can respond an address bus cycle. These verification components primarily implement register addressable devices. As a result, they do not support bursting.

OSVVM supports bursting in memory-based verification components. These verification components allow a manager to interact with the memory using either single word or burst cycles. The Axi4Memory VC described in the AXI4 VC User Guide is an example of this.

7.1 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.

7.1.1 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
) ;
```

7.1.2 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
  ) ;
```

7.2 Interface Specific Transactions

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

7.2.1 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
  ) ;
```

7.2.2 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
  ) ;
```

8. 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
  ) ;
```

9. 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 ;
```

10. 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
  ) ;
```

11. Verification Components and Stream Model Independent Transactions

11.1 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 4 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

AddressBusOperationType Value	Manager Transaction Name	Subordinate Transaction Name
ASYNC_WRITE_DATA	WriteDataAsync	TryGetWriteData
WRITE_BURST	WriteBurst	
ASYNC_WRITE_BURST	WriteBurstAsync	
READ_OP	Read	TryGetWriteData
ASYNC_READ		TrySendRead
READ_CHECK	ReadCheck	
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 4. Correlation between AddressBusOperationType and the transaction name

11.2 **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 ;
```

11.3 Using the Burst FIFO in the Manager VC

11.3.1 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 5 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 5. BurstFifo Initialization

11.3.2 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. 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.