OSVVM
Verification Component
Developer's Guide

-- Preliminary --

Release 2022.03

By

Jim Lewis

SynthWorks VHDL Training

Jim@SynthWorks.com

http://www.SynthWorks.com

Table of Contents

1 Overview

In this guide, we explore writing a verification component (VC) for a DpRam controller interface. The signaling on the DpRam controller is quite simple. This allows us to focus on the aspects of creating an OSVVM style VC.

Be sure to read the OSVVM's Structured Testbench Framework Guide first. This guide provides you with the background to the overall structure of OSVVM's testbench and where verification components fit in.

2 Step 1: Identify the Transactions Supported by the Interface

Writing an OSVVM VC starts by looking at the interface we want to drive. In this guide, we will test the OSVVM DpRam block. This DpRam is similar to the memory models provided by FPGAs – such as Xilinx's BRAM.

The DpRam has two identical interfaces (hence Dual Port) that can both write or read from the interface. A block diagram for the DpRam is shown in Figure 1. It has registers on all input signals (AddrX, DataX, and WriteX) and optionally has a register on the output DataOutX.

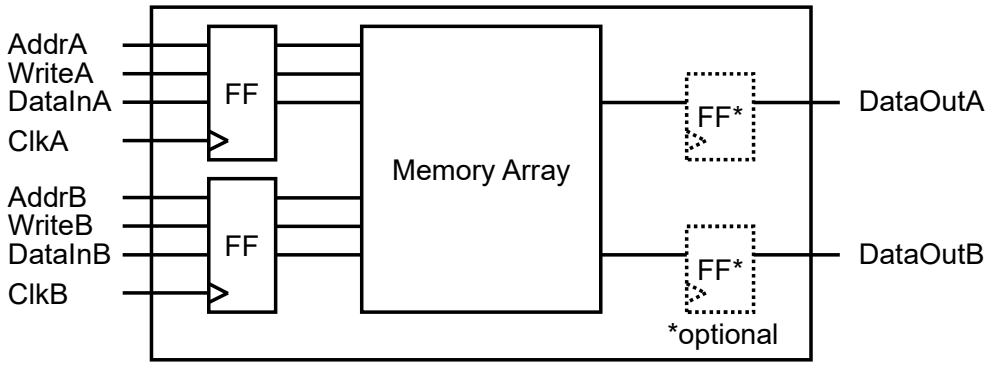

Figure 1. DpRam structure

A timing diagram for the interface is shown in Figure 2. A write operation starts by presenting the Address (AddrX), Data (DataInX), and the Write indicator (WriteX). On the next rising edge of clock, the write operation is completed. A read operation starts by presenting Address (AddrX). On the next rising edge of clock the Address is accepted and read data (DataOutX) is available.

It has registers on all input signals (AddrX, DataX, and WriteX) and optionally has a register on the output DataOutX.

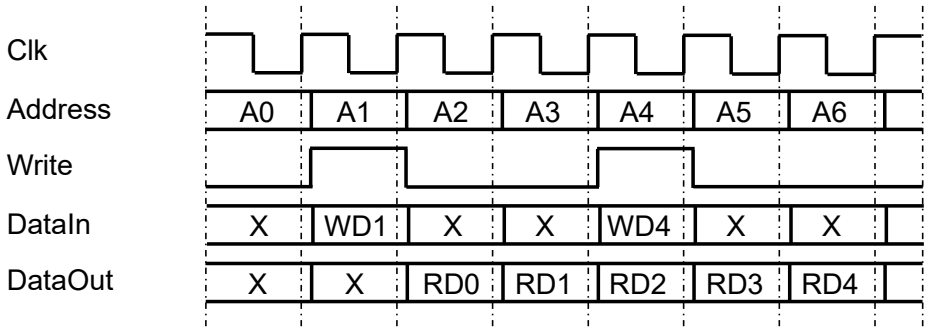

Figure 2. DpRam Timing Diagram

From a transaction point of view, OSVVM classifies interfaces that support address and data as an Address Bus interface. This interface supports a Write and a Read operation. Depending on the FPGA, it can also support a Write and Read of the same address. What you get on that read will depend on the FPGA target and configuration of the DPRAM for that target. For our purposes, the DPRAM provides the value that was in the RAM before the write operation started.

3 Step 2: Map the VC transactions to OSVVM MIT

To simplify VC development, OSVVM provides model independent transactions for Address Bus and Streaming interfaces that are intended to support a superset of the transactions that you need to implement in your VC.

We are in luck, the OSVVM Address Bus Model Independent Transactions support all the transactions needed in our interface. They are:

Write(TRec, iAddr, iData, [MsgOn])
Read(TransactionRec, iAddr, oData, [StatusMsgOn])
ReadCheck(TransactionRec, iAddr, iData, [StatusMsgOn])

4 Step 3: Create a simple blocking VC

If OSVVM were to have a "Lite" approach, this would be it.

In this blog, we will create a simple blocking VC. Blocking means that the test sequencer pauses (stops) while the transaction is executing in the VC. This is the same sort of capability we would get if we were to use a subprogram to implement the VC behavior - but this is simpler than that.

The structure for a blocking VC is shown in see Figure 3.

```
entity DpRamManager is
  port (. . .) ;
end entity DpRamManager ;
architecture blocking of DpRamManager is

TransctionHandler : process
begin

   wait for Transaction(. . . ) ;
   case TransRec.Operation is
      when WRITE_OP =>        -- Do Write things
      when READ_OP =>         -- Do Read Things
      when …  =>              -- Do Model Directives
  end case
end process TransactionHandler ;
end architecture blocking ;
```

Figure 3. Blocking VC Structure

Let us build the DpRamManager VC piece by piece.

4.1 Package References

To use the Address Bus Model Independent Transactions, we need to include the library osvvm_common and the context OsvvmCommonContext. To use the OSVVM utility library, we need to include the library osvvm and the context OsvvmContext. For interfaces that support bursting, we will need to include the ScorebaordPkg_slv.

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

library osvvm ;
  context osvvm.OsvvmContext ;
  use osvvm.ScoreboardPkg_slv.all ;

library osvvm_common ;
  context osvvm_common.OsvvmCommonContext ;
```

4.2 DpRamManager Interface

The DpRamManager interface has the DUT interface and a transaction record interface. The transaction record tells the VC what to do. For clocked interfaces, each output changes after a propagation delay after clock rises. There is one generic of the form tpd_Clk_<output> for each output.

```
entity DpRamManager is
generic (
  MODEL_ID_NAME    : string := "" ;
  DEFAULT_DELAY    : time   := 1 ns ;
  tpd_Clk_Address  : time   := DEFAULT_DELAY ;
  tpd_Clk_Write    : time   := DEFAULT_DELAY ;
  tpd_Clk_oData    : time   := DEFAULT_DELAY
) ;
port (
  Clk         : In   std_logic ;
  nReset      : In   std_logic ;

  -- DpRam Functional Interface
  Address     : Out  std_logic_vector ;
  Write       : Out std_logic ;
  oData       : Out std_logic_vector ;
  iData       : In  std_logic_vector ;

  -- Address Bus Transaction Interface
  TransRec    : InOut AddressBusRecType
) ;

  -- Name for OSVVM Alerts
  constant MODEL_INSTANCE_NAME : string :=
    IfElse(MODEL_ID_NAME /= "",
        MODEL_ID_NAME, PathTail(to_lower(DpRamManager'PATH_NAME))) ;

end entity DpRamManager ;  context osvvm_common.OsvvmCommonContext ;
```

4.3 Declare one or more AlertLogIDs

Declare the ModelID as a signal of AlertLogIDType. Most simulators allow you to initialize the AlertLogID as an initializer. Formally the language does not.

As a convention, we name the ID for the VC, ModelID. By following a convention, it simplifies cut and paste between different VC.

```
architecture SimpleBlocking of DpRamManager is
  signal ModelID : AlertLogIDType ;
begin
  Initialize : process
    variable ID : AlertLogIDType ;
  begin
    ID                      := NewID(MODEL_INSTANCE_NAME) ;
    ModelID                 <= ID ;
    wait ;
  end process Initialize ;
```

4.4 Anatomy of the TransactionHandler

The TransactionHandler process decodes commands from the test sequencer and executes them. The basic structure has a call to WaitForTransaction and then a case statement to decode the operation. In the body of the case target are interface actions that are specific to the interface.

The WaitForTransaction does the handshaking with the test sequencer and causes the process to wait until a transaction is available. It requires Clk and the two handshaking signals Rdy (driven by the sequencer) and Ack (driven by the VC). Both Rdy and Ack are in the transaction record.

The case statement decodes the operation field that is in the transaction record.

```
TransctionHandler : process
begin
  WaitForTransaction(
    Clk      => Clk,
    Rdy      => TransRec.Rdy,
    Ack      => TransRec.Ack
  ) ;
  Operation := TransRec.Operation ;
  case TransRec.Operation is
    when WRITE_OP =>        -- Do Write things
    when READ_OP =>         -- Do Read Things
    when …  =>              -- Do Model Directives
  end case ;
end process TransactionHandler ;
```

To initialize outputs, add an assignment to the outputs at the beginning of the process and put the repetitive part of the process into a "loop – end loop;".

```
TransctionHandler : process
begin
  -- Initialize Outputs
  Address     <= (others => 'X') ;
  Write       <= 'X' ;
  oData       <= (others => 'X') ;

  loop
    wait for Transaction(. . . ) ;
    case TransRec.Operation is
      when WRITE_OP =>        -- Do Write things
      when READ_OP =>         -- Do Read Things
      when …  =>              -- Do Model Directives
    end case ;
  end loop ;
end process TransactionHandler ;
```

4.5 Creating the Write Operation

The Write Operation implements the interface behavior for a write cycle. When WaitForTransaction exits, the VC will be aligned to the rising edge of clock. Address and oData will be driven aligned to clock with a propagation delay specified by the output timing generics (tpd_Clk_<output>).

The Address for the write operation is in the Address field of the transaction record. SafeResize is used to optionally resize the value as well as convert from the type used in the record (osvvm.ResolutionPkg.std_logic_vector_max) to std_logic_vector. Write data is in the DataToModel field of the transaction record.

```
when WRITE_OP =>
  Address <= SafeResize(TransRec.Address, Address'length) after tpd_Clk_Address ;
  oData   <= SafeResize(TransRec.DataToModel, oData'length) after tpd_Clk_oData ;
  Write   <= '1' after tpd_Clk_Write ;

  WaitForClock(Clk) ;
  Log( . . . ) ; -- shown below
  Address <= not Address after tpd_Clk_Address ;
  oData   <= not oData    after tpd_Clk_oData ;
  Write   <= '0' after tpd_Clk_Write ;
```

If you were to using procedures to implement the interface behavior, the above code is the same code that would be in the procedure. Hence, writing a simple VC is no more complex than writing a subprogram.

To provide feedback about what is happening in the VC, we need to log the write transaction. Above the call to log happens at the point the write operation is acted upon by the DpRam. The details of the log are shown below. For details on Alert, Affirm, and Log please see the AlertLogPkg Users Guide. The final parameter to Log below is the transaction enable for the log. This allows the Log to be enabled for a single transaction without having to enable and then disable it.

```
Log( ModelID,
  "Write Operation, Address: " & to_hxstring(Address) &
  "  Data: " & to_hxstring(oData) &
  "  Operation# " & to_string (TransRec.Rdy)
  INFO,
  TransRec.StatusMsgOn
) ;
```

4.6 Creating the Read Operation

The read operation is similar to the write operation except we have to wait an extra clock cycle for the data to be available. Currently this code does not account for the extra optional register on the output of the DpRam. Read data is returned to the test sequencer in the DataFromModel field of the transaction record. This version of SafeResize optionally resizes the data value as well as converts from std_logic_vector to the type used in the record (osvvm.ResolutionPkg.std_logic_vector_max).

```
when READ_OP  | READ_CHECK =>
  Address <= SafeResize(TransRec.Address, Address'length) after tpd_Clk_Address ;
  Write   <= '0' after tpd_Clk_Write ;

  WaitForClock(Clk) ;
  Address <= not Address after tpd_Clk_Address ;

  WaitForClock(Clk) ;

  TransRec.DataFromModel <= SafeResize(iData, TransRec.DataFromModel'length) ;
  Log( . . . ) ;
```

The Log for a simple read transaction is the same as the Log for the write operation shown previously.

4.7 Creating the Read Check Operation

```
when READ_OP  | READ_CHECK =>
  Address <= SafeResize(TransRec.Address, Address'length) after tpd_Clk_Address ;
  Write   <= '0' after tpd_Clk_Write ;

  WaitForClock(Clk) ;
  Address <= not Address after tpd_Clk_Address ;

  WaitForClock(Clk) ;

  TransRec.DataFromModel <= SafeResize(iData, TransRec.DataFromModel'length) ;
```

4.8 Creating Directive Transactions

A directive transaction communicates with the verification component, but does not result in any actions on the DUT interface. Common ones for OSVVM include, WaitForClock, GetAlertLogID, and WaitForTransaction. WaitForClock stops the VC for the specified number of clocks. GetAlertLogID looks up the AlertLogID of the VC and returns it. WaitForTransaction waits for any pending transactions to complete and returns – in a blocking model, this does nothing because there all transactions complete.

```
when WAIT_FOR_CLOCK =>
  WaitForClock(Clk, TransRec.IntToModel) ;

when GET_ALERTLOG_ID =>
  TransRec.IntFromModel <= integer(ModelID) ;
  wait for 0 ns ;

when WAIT_FOR_TRANSACTION =>
  wait for 0 ns ;
```

4.9 Detecting Multiple Drivers

When writing tests, it is natural to copy a piece of code from one process to another. It is common to forget to change the transaction record name to the one for the current process. As a result, the record for the other process ends up with two drivers on it (one for each process). When the resolution function used for the operation field detects more than one operation being driven, it changes the operation to MULTIPLE_DRIVER_DETECT.

It is best practice for the model to decode the MULTIPLE_DRIVER_DETECT and produce an appropriate FAILURE message.

```
when MULTIPLE_DRIVER_DETECT =>
  Alert(ModelID, MODEL_NAME & ": Multiple Drivers on Transaction Record." &
                 "  Transaction # " & to_string(TransRec.Rdy), FAILURE) ;
```

4.10 Catching Unused Commands

A VC is not required to implement all transactions that are supported by OSVVM Address Bus Model Independent Transactions. It is best practice to produce an appropriate FAILURE message when an unsupported transaction is received.

```
when others =>
  Alert(ModelID, "Transaction " & to_string(Operation) &
                 " is not implemented by the DpRamManager VC", FAILURE) ;
```

4.11 The Final Design

The final design can be found …

5 Going Further

Some interfaces, such as this one, allow multiple things to be going on at a time. For example, what happens when we do a read address 1 followed by a write address 2. The interface allows the write address 2 to start on the clock immediately after starting the read address 1 operation, such that due to latencies in the system, both will finish at the same time. This is shown in figure … With blocking transactions, the read address 1 is required to finish before the write address 2 can start. This leaves a dead cycle on the interface.

Part 2 of this blog will show how to implement asynchronous transactions in the VC, so that we can start a read operation and then do the write operation on the next cycle.

For more information on Writing Tests with OSVVM, see the OSVVM Test Writer's User Guide in the OSVVM documentation repository.

6 Building the OSVVM Libraries and Running the Testbenches

OSVVM is available on GitHub at https://github.com/OSVVM. The OSVVM repositories are all submodules of the OsvvmLibraries repository. Retrieve it using git as shown in Figure 4.

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

Figure 4. Retrieving OSVVM from GitHub

The AXI4, Axi4Lite, AxiStream, and UART verification components come with OSVVM testbenches. Prior to starting the OSVVM scripting environment, create a directory named sim to run your simulations from. Start your simulator in the sim directory. Figure 5 shows the steps to run Mentor QuestaSim/ModelSim or Aldec RivieraPRO to build the OSVVM utility and verification component libraries and then run the verification component regression suite.

```
cd sim
source ../OsvvmLibraries/Scripts/StartUp.tcl
build  ../OsvvmLibraries/OsvvmLibraries.pro
build  ../OsvvmLibraries/RunAllTests.pro
```

Figure 5. Compiling and Running OSVVM

For more details on the OSVVM scripting environment see Script_user_guide.pdf. In particular, details to run Aldec's ActiveHDL, GHDL, Synopsys VCS and Cadence Xcelium are in the Script user guide.

7 About the OSVVM

The OSVVM utility and verification component libraries were developed and are maintained by Jim Lewis of SynthWorks VHDL Training. These libraries evolved from methodology and packages developed for SynthWorks' VHDL Testbenches and verification class.

Please support OSVVM by purchasing your VHDL training from SynthWorks.

8 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.

9 References

[1] https://blogs.sw.siemens.com/verificationhorizons/2020/12/16/part-6-the-2020-wilson-research-group-functional-verification-study/

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