OSVVM's Structured Testbench Framework

User Guide for Release 2022.03

By

Jim Lewis

SynthWorks VHDL Training

Jim@SynthWorks.com

http://www.SynthWorks.com

Table of Contents

1 Overview

Writing tests is all about creating waveforms at an interface. In a basic test approach, each test directly drives and wiggles interface waveforms. This is tedious and error prone.

In OSVVM, signal wiggling is replaced by transactions. A transaction is an abstract representation of either an interface waveform (such as Send for a UART transmit) or a directive to the verification component (such as wait for clock). In OSVVM, a transaction is initiated using a procedure call. In a verification component (VC) based approach, the procedure call collects the transaction information and passes it to the VC via a transaction interface (a record). The VC then decodes this information and creates the corresponding interface stimulus to the device under test (DUT).

Figure 1 shows two calls to a send procedure and the corresponding waveforms produced by the UartTx verification component.

```
UartTbTxProc : process
begin
  WaitForBarrier(StartTest) ;
  Send(UartTxRec, X"4A") ;
  Send(UartTxRec, X"4B") ;
```

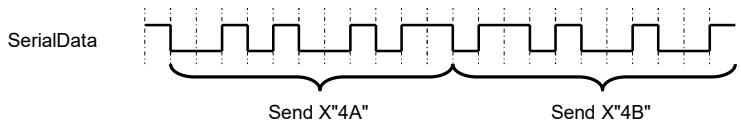

Figure 1. Two Calls to Send transaction and the resulting waveform

Writing tests using transactions makes our tests easy to read by anyone who has basic programming experience, including verification, hardware, software, and system engineers.

2 Framework / TestHarness

The objective of any verification framework is to make the Device Under Test (DUT) "feel like" it has been plugged into the board. Hence, the framework must be able to produce the same waveforms and sequence of waveforms that the DUT will see on the board.

The OSVVM testbench framework looks identical to other frameworks, including SystemVerilog. It includes the DUT (DpRam), verification components (DpRamController), and test sequencer (TestCtrl) as shown in Figure 2. The top level of the testbench connects the components together and is often called a test harness. Connections between the verification components and TestCtrl use VHDL records as an interface (aka transaction interface). Connections between the verification components and the DUT are the DUT interfaces (such as UART, AxiStream, AXI4, SPI, and I2C).

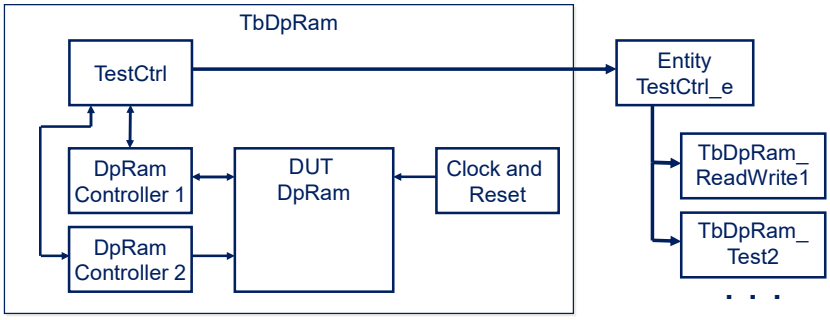

Figure 2. OSVVM Testbench Framework

The test harness is coded using structural code – just like connectivity in RTL design. Figure 3 shows a sketch of the code for TbDpRam. OSVVM verification components put component declarations in packages and reference the packages in the context declaration (DPRamContext). This allows us to use the simpler component instances without having a component declaration. The complete code is in TbDpRam.vhd in the directory OsvvmLibraries/DpRam/testbench.

```
library osvvm ;
  context osvvm.OsvvmContext ;
library OSVVM_DPRAM ;
  context OSVVM_DpRam.DpRamContext ;
. . .
entity  TbDpRam is
end entity TbDpRam ;
architecture TestHarness of TbDpRam is
  constant ADDR_WIDTH : integer := 24 ;
  constant DATA_WIDTH : integer := 16 ;

  signal Clk         : std_logic ;
  signal nReset      : std_logic ;
  signal AddrA       : std_logic_vector(ADDR_WIDTH-1 downto 0) ;
  signal WriteA      : std_logic ;
  signal DataInA     : std_logic_vector(DATA_WIDTH-1 downto 0) ;
  . . .
  signal Manager1Rec, Manager2Rec  : AddressBusRecType (
          Address      (AXI_ADDR_WIDTH-1 downto 0),
          DataToModel  (AXI_DATA_WIDTH-1 downto 0),
          DataFromModel(AXI_DATA_WIDTH-1 downto 0)
        ) ;

  component TestCtrl is
    port ( . . . ) ;
  end component TestCtrl ;  . . .
begin

  -- procedures
  osvvm.TbUtilPkg.CreateClock(Clk, tperiod_Clk) ;
  osvvm.TbUtilPkg.CreateReset(nReset, . . .) ;

  -- instances
  DpRam_1 : DpRam
    generic map (ADDR_WIDTH, DATA_WIDTH, FALSE, FALSE, "DpRam_1")
    port map (Clk, AddrA, WriteA, DataInA, DataOutA, AddrB, …) ;

  Manager_1 : DpRamManager port map (…, AddrA, WriteA, …, Manager1Rec);

  Manager_2 : DpRamManager port map (…, AddrB, WriteB, …, Manager2Rec);

  TestCtrl_1 :  TestCtrl port map   (nReset, Manager1Rec, Manager2Rec) ;
end TestHarness ;
```

Figure 3. TestCtrl Architecture

3 The Test Sequencer (TestCtrl)

Tests (or test cases) are written in the test sequencer (TestCtrl). A sketch of a TestCtrl architecture is shown in Figure 4. Key features of OSVVM test cases are:

- Transaction procedure calls (Write, Send, …) are used to create the sequence of interface waveforms that make up a test.
- A separate process is used to create transactions for each independent interface as this is the easiest way to get independent behavior that is present in a hardware system.
- A control process is used to coordinate test initiation and completion.
- Separate tests are written in separate architectures of TestCtrl
- Coordinated activities on independent interfaces are implemented using synchronization primitives (such as WaitForBarrier). More details are in TbUtilPkg User Guide and Quick Reference.

Details of writing tests are in the OSVVM Test Writers User Guide.

```
architecture BasicReadWrite1  of TestCtrl is
  . . .
begin
  ControlProc : process
  begin
    . . .
    WaitForBarrier(TestInit) ;
    . . .
    WaitForBarrier(TestDone, 5 ms) ;
    ReportAlerts ;
    std.env.stop;
  end process ;
  Axi4MProc : process
  begin
    WaitForBarrier(TestInit) ;
    Write(. . .) ;
    Read(. . .) ;
    WaitForBarrier(DutInit);
    . . .
    WaitForBarrier(TestDone) ;
  end process Axi4MProc ;
  TxProc : process
  begin
    WaitForBarrier(DutInit);
    Send(. . .) ;
    Send(. . .) ;
    . . .
    WaitForBarrier(TestDone) ;
  end process TxProc ;
  . . .
```

Figure 4. TestCtrl Architecture

4 The Verification Component (VC)

The verification component (VC) translates the transaction information into an interface waveform. It receives the transaction information on a record (see next section). A template for a simple OSVVM verification component is shown in Figure 5.

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

library osvvm_common ;
  context osvvm_common.OsvvmCommonContext ;

entity DpRamController is
generic ( . . . ) ;
port (
  -- DUT Interface
  . . .
  -- Testbench Transaction Interface
  TransRec    : InOut AddressBusRecType
) ;
end entity DpRamController ;
architecture SimpleBlocking of DpRamController is
begin

  TransactionHandler : process
  begin

      WaitForTransaction(Clk, TransRec.Rdy, TransRec.Ack) ;

      case TransRec.Operation is
        -- Model Transaction Dispatch
        when WRITE_OP =>
          DpRamWrite(TransRec, Clk, Address, oData, Write) ;

        when READ_OP =>
          DpRamRead(TransRec, Clk, Address, Write, iData) ;

        when WRITE_AND_READ =>
          DpRamWriteAndRead(TransRec, Clk, Address, oData, Write, iData) ;

        when others =>
          Alert(ModelID, "Unimplemented Transaction", FAILURE) ;

      end case ;
  end process TransactionHandler ;
end architecture SimpleBlocking ;
```

Figure 5. Verification Component Structure

OSVVM VC commonly start with libraries ieee, osvvm, and osvvm_common. They typically include the ieee packages, the OsvvmContext and OsvvmCommonContext context references, and the package ScoreboardPkg_slv.

The VC interface has the DUT interface signals and a TransRec of either AddressBusRecType or StreamRecType.

For simple VC, there is a transaction handler process that waits until the VC receives a transaction (WaitForTransaction) and then decodes the transaction and does appropriate interface signaling (here represented by procedure calls to DpRamWrite, DpRamRead, and DpRamWriteAndRead).

Details of writing VC are in the OSVVM Verification Component Developers Guide.

5 Record Based Transaction Interface

The transaction interface is a record that is used to communicate information between the test sequencer (TestCtrl) and the verification component (VC). As such, it is an "InOut" of both TestCtrl and the VC. To properly handle drivers, the record elements are a resolved type from the OSVVM package ResolutionPkg. We call such a record an OSVVM interface. An example an OSVVM interface is shown in Figure 6.

```
type AddressBusMasterTransactionRecType is record
  Rdy              : bit_max ;
  Ack              : bit_max ;
  Address          : std_logic_vector_max_c ;
  AddrWidth        : integer_max ;
  DataToModel      : std_logic_vector_max_c ;
  DataFromModel    : std_logic_vector_max_c ;
  . . .
end record AddressBusMasterTransactionRecType ;
```

Figure 6. An OSVVM Interface

Types in ResolutionPkg use "maximum" as a resolution function and add the suffix "_max" or "_max_c" to the type names. Maximum resolution picks the largest value driven on each element of signal object that has multiple drivers. The largest value is the right most value of a type (‘-‘ for std_ulogic). This works in conjunction with the left most value being the smallest and the default value for an object of that type.

ResolutionPkg provides the following (as subtypes): std_logic_max, std_logic_vector_max, unsigned_max, signed_max, bit_max, bit_vector_max, integer_max, integer_vector_max, time_max, time_vector_max, real_max, real_vector_max, character_max, string_max, boolean_max, and boolean_vector_max. More details on ResolutionPkg are in ResolutionPkg Users Guide.

For OSVVM VC, the OSVVM Model Independent Transaction Interfaces define two such records that are suitable for most VC.

6 Transaction Procedures

Calls to the transaction procedures tell a verification component what to do on its interface. The interface of the verification component determines how to do it.

Figure 7 shows an example of a transaction procedure. These procedures are nothing more than an abstraction layer that puts the transaction information into the record, signals a transaction is ready (RequestTransaction), and gathers results when the transaction completes.

```
procedure Read (
  signal   TransactionRec : InOut AddressBusRecType ;
           iAddr          : In    std_logic_vector ;
  variable oData          : Out   std_logic_vector
) is
begin
  -- Put values in record
  TransactionRec.Operation   <= READ_OP ;
  TransactionRec.Address     <= SafeResize(iAddr, TransactionRec.Address'length);
  -- Start Transaction
  RequestTransaction(Rdy => TransactionRec.Rdy, Ack => TransactionRec.Ack) ;
  -- Return Results
  oData  := SafeResize(TransactionRec.DataFromModel, oData'length) ;
end procedure Read ;
```

Figure 7. An OSVVM Interface

Historically we wrote transaction procedures for each verificaiton component. After doing this for long enough, we realized that from a transaction perspective, many interfaces do the same thing and could be handled by a common set of transaction procedures.

For OSVVM VC, the OSVVM Model Independent Transaction Interfaces define two sets of procedures that are suitable for most VC.

7 OSVVM Model Independent Transactions

OSVVM Model Independent Transactions evolved from the observation that some interfaces do the same transactions. Address bus interfaces (such as AXI, Avalon, Wishbone, …) all do read and write transactions. Streaming interfaces (such as AxiStream, UART, …) all do send and get transactions.

For these interfaces, OSVVM Model Independent Transactions define

- The Transaction Interfaces: AddressBusRecType and StreamRecType.
- The Transaction initiation procedures: Read, Write (address bus) vs Send, Get (stream).

With OSVVM Model Independent Transactions, it reduces the development of verification components to just write the model functionality/behavior.

8 Summary

Some methodologies (or frameworks) are so complex that you need an automation/build script to create initial starting point for verification components and/or the test harness. SystemVerilog + UVM is certainly like this. There are even several organizations that propose that you use their "Lite" or "Easy" approach.

OSVVM is simple enough that you don't need a "Lite" version or an automation to properly setup your test environment. This user guide will walk you the steps of how to write a verification component that effectively utilizes OSVVM capabilities. It will introduce you to OSVVM's existing verification components and how to use them as a template to create your own verification components. It will show you how to use OSVVM's model independent transactions to simplify and minimize the amount of work you need to develop your verification components.

At the end of the day, OSVVM does not need a "Lite" version because we make writing verification components as simple as writing a procedure.

At the heart of OSVVM's testbench framework is writing tests based on transactions, so we start our journey there.

9 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 8.

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

Figure 8. 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 9 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 9. 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.

10 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.

11 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.

12 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.