OSVVM's Structured Testbench Framework

User Guide for Release 2023.01

By

Jim Lewis

SynthWorks VHDL Training

Jim@SynthWorks.com

http://www.SynthWorks.com

Table of Contents

1 Overview

OSVVM's structured testbench framework is simple enough to use on small blocks. So simple in fact that a "Lite" or "Easy" approach is not needed. It is powerful enough to use on large, complex ASICs. This allows the same testbench architecture to be used for RTL, CORE, and System tests – this in turn facilitates reuse/pre-use of both verification components and test sequences.

SynthWorks has been using this framework for 25+ years in our training classes and consulting work. During that time, we have innovated new capabilities and evolved our approach to increase re-use and reduce effort and time spent.

In this document we examine OSVVM's framework in detail and see that it has many similar elements to SystemVerilog + UVM. However, one thing not present is OO language constructs. Instead OSVVM uses ordinary VHDL constructs, such as structural and behavioral code. This makes it readily accessible to both verification and RTL engineers.

2 Transactions

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 transmitter or Read for an AXI4 Bus) or a directive (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") ;
```

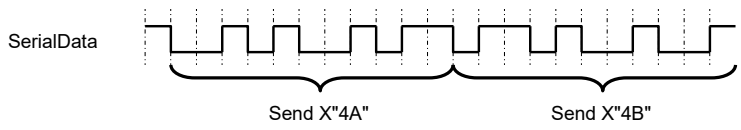

Figure 1. Two Calls to Send transaction and the resulting waveform

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

3 Framework / Test Harness

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 – this includes the simultaneous behavior of independent interfaces.

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 test sequencer calls a sequence of transactions. The transactions in turn instruct the verification components to create a sequence of waveforms on the DUT interface.

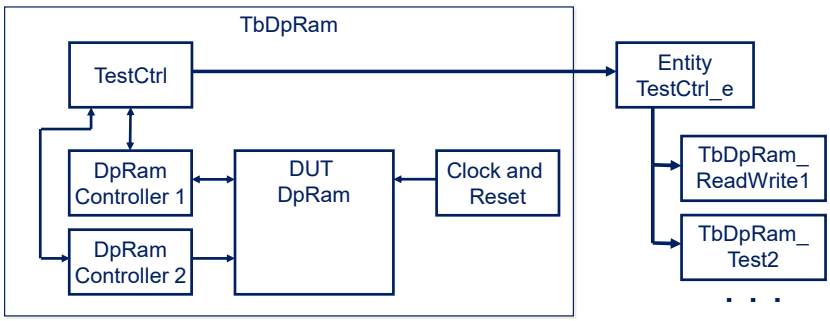

Figure 2. OSVVM Testbench Framework

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).

Do not despair about the amount of work here – OSVVM has done some of it for you. For example, our Model Independent Transactions define both the transaction interfaces and transaction call API for you.

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 the use of 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

4 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, Read, ReadCheck, Send, Get, Check, …) are used to create the sequence of interface waveforms that make up a test.
- Independent / concurrent behavior that is present in a hardware system is created by using a separate process for each independent interface.
- A control process is used to coordinate test initiation and completion.
- Separate tests are written in separate architectures of TestCtrl
- Synchronization primitives (such as WaitForBarrier) coordinate activities on independent interfaces. More details are in TbUtilPkg User Guide and Quick Reference.

Details of writing tests are in the OSVVM Test Writers User Guide. The complete code is in TbDpRam_BasicReadWrite.vhd in the directory OsvvmLibraries/DpRam/testbench.

```
architecture BasicReadWrite of TestCtrl is
  . . .
begin
  ControlProc : process
  begin
    SetAlertLogName("TbDpRam_BasicReadWrite") ;
    . . .
    WaitForBarrier(TestDone, 5 ms) ;
    EndOfTestReports ;
    std.env.stop;
  end process ;
  Manager1Proc : process
  begin
    . . .
    for i in 1 to 10 loop
      Write(Manager1Rec, X"01_0000" + i, X"1000" + i ) ;
    end loop ;
    . . .
    WaitForBarrier(TestDone) ;
  end process Manager1Proc;
  Manager2Proc : process
  begin
    . . .
    for i in 1 to 10 loop
      ReadCheck(Manager2Rec, X"01_0000" + i, X"1000" + i) ;
    end loop ;
    . . .
    WaitForBarrier(TestDone) ;
  end process Manager2Proc ;
```

Figure 4. TestCtrl Architecture

5 Selecting Test Cases

An OSVVM test suite consists of multiple architectures of the TestCtrl, each architecture contains a single test case. Organizationally the TestCtrl entity and its multiple architectures go in separate files.

Using multiple architectures does not change the test harness – other than we need to use component instantiation for TestCtrl rather than entity instantiation. Component instantiation allows late binding of the architecture and optionally the use of configuration declarations.

Compilation dependencies are on primary units (entity) and not the architectures. Hence, the design hierarchy and testbench can be compiled before the test architectures are compiled.

The architecture for a given test is selected by either default binding or a configuration declaration. With default binding, the most recent successfully analyzed (compiled) architecture is selected for simulation. Figure 5 shows the OSVVM script sequence to run a test using default binding.

```
TestName TbAxi4_MemoryReadWriteDemo1       ;# Set TestName for scripts
analyze  TbAxi4_MemoryReadWriteDemo1.vhd   ;# Analyze test case architecture
simulate TbAxi4Memory                      ;# Load top level.
```

Figure 5. OSVVM Sequence to Run a test using Default Binding

OSVVM examples use configuration declarations to select the architecture to run. Figure 6 shows the configuration to select test architecture MemoryReadWriteDemo1. To simplify the entire process, we give the configuration the same name as the test case name. The architecture name is a short version of the test case name. OSVVM examples put the configuration declaration at the bottom of the file containing the test architecture.

```
Configuration TbAxi4_MemoryReadWriteDemo1 of TbAxi4Memory is
  for TestHarness
    for TestCtrl_1 : TestCtrl
      use entity work.TestCtrl(MemoryReadWriteDemo1) ;
    end for ;
  end for ;
end TbAxi4_MemoryReadWriteDemo1 ;
```

Figure 6. Configuration for Test Architecture MemoryReadWriteDemo1

To run the test case using a configuration use RunTest as shown in Figure 7. This is an abbreviation for calling TestName, analyze, and simulate where they all use the same root name.

```
RunTest  TbAxi4_MemoryReadWriteDemo1.vhd
```

Figure 7. OSVVM Sequence to Run a test using Configurations

6 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 8.

```
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 8. 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). Often it is better practice to write code here rather than calling a subprogram – one more layer of abstraction introduced by the subprograms can lead to obfusciation.

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

7 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 9.

```
type AddressBusRecType 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 AddressBusRecType ;
```

Figure 9. 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, AddressBusRecType and StreamRecType, that are suitable for most VC.

8 Transaction API (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 10 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 10. 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.

9 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.

More details are provided in the Address Bus Model Independent Transactions User Guide and the Stream Model Independent Transactions User Guide.

10 Running the Examples

See the steps in the OSVVM Overview guide for running the OSVVM demos. Do the steps shown in Figure 11 in your simulator. StartUp.tcl needs to be sourced each time you start the simulator. See the Script User Guide for additional details for Aldec's ActiveHDL, GHDL, Synopsys VCS, and Cadence Xcelium.

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

Figure 11. Compiling and Running DPRAM tests

11 Summary

This document starts your journey into OSVVM's Structured Testbench Framework.

For creating the test harness, what you have here and in the example code (TbDpRam.vhd) which is in the directory (OsvvmLibraries/DpRam/testbench) should be sufficient. The code is a basically a structural netlist which is created in a 'Lego' like fashion – just like RTL code.

In OSVVM, creating test cases is nothing more than calling transactions in the test sequencer (TestCtrl). See the OSVVM Test Writers User Guide for more details. From there more details are in each of OSVVM's user guides for each package in the OSVVM Utility Library. The key benefit to organizing test cases in this fashion is that the test cases are readable by all – where all is RTL, verification, software, and system engineers.

In OSVVM, a verification component is nothing more than behavioral code. See the OSVVM Verification Component Developers Guide for more details. One benefit to this approach is that it is similar enough to RTL that an engineer can easily do both RTL and verification tasks.

Connections between the test sequencer and the verification component are handled by OSVVM's Model Independent Transactions. These define the interface records and abstract transaction API that can be used by verification components. For more details see the Address Bus Model Independent Transactions Users' Guide and the Stream Model Independent Transactions Users's Guide. This approach benefits verification component developers by removing the need to create transaction records and the transaction API for each verification component. This approach benefits test case developers since there are only two transaction APIs to learn rather than one per verification component. It also promotes reuse of test sequences between similar verification components.

12 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.

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, Advanced VHDL Testbenches and Verification, student manual for SynthWorks' class.