Writing Tests using OSVVM

Jim Lewis, SynthWorks Design Inc, Jim@SynthWorks.com

1. Introduction

Most people don't think of VHDL as a verification language. However, with the Open Source VHDL Verification Methodology (OSVVM) utility and verification component libraries it is. Using OSVVM we can create readable, powerful, and concise VHDL verification environments (testbenches) whose capabilities are similar to other verification languages, such as SystemVerilog and UVM.

This paper covers the basics of using OSVVM's transaction-based test approach to write directed tests, write constrained random tests, use OSVVM’s generic scoreboard, add functional coverage, add protocol and parameter checks, add message filtering, and add test wide reporting.

2. Why VHDL? Why OSVVM?

According the 2020 Wilson Research Group Functional Verification Study,

- 64% of FPGA design worldwide uses VHDL
- 50% of FPGA verification worldwide uses VHDL
- 18% of FPGA verification projects worldwide use OSVVM (or 36% of VHDL FPGA verification projects)
- For Europe, 34% of FPGA verification projects use OSVVM while only 26% use UVM

Clearly VHDL is a dominant language in the FPGA market and OSVVM is a leading VHDL Verification Methodology.

3. Benefits of OSVVM

For the VHDL community, OSVVM is a clear win. We can write tests in the same language we already know and re-use components, tests, and testbenches from other projects. More importantly OSVVM’s transaction-based approach simplifies creating readable and reviewable tests (an important metric in the safety critical community). In addition, OSVVM uses a component/entity-based approach just like RTL design. Hence, not only can RTL designers read tests and verification components, they can write them. While having independent design and verification teams is important, it is also important to be able to deploy engineers to either a design or verification role on a project by project basis.

4. What are Transactions?

A transaction is an abstract representation of an interface operation (such as UART transmit) or directive (such as get transaction count). In OSVVM, a transaction is initiated with a procedure call. In the OSVVM verification component approach, the procedure places the transaction information into a record and passes it to the verification component. The component in turn executes the transaction and provides 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") ;
```

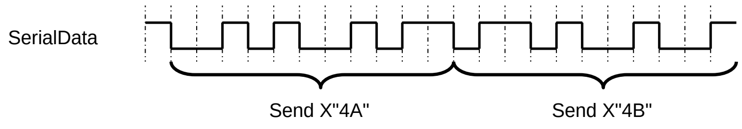

Figure 1. Two Calls to Send transaction and the resulting waveform

Each verification component in the OSVVM library implements a set of model independent transactions. The table below gives a brief summary

Table 1. OSVVM Standard Transactions

Bus Transactions

`Write(TransactionRec, X"AAAA", X"DDDD") ;`
`Read (TransactionRec, X"AAAA", DataOut) ;`
`ReadCheck(TransactionRec, X"AAAA", X"DDDD") ;`

Streaming or Serial Transactions (AxiStream, UART, …)

`Send(TransactionRec, X"DDDD") ;`
`Get(Transactionrec, DataOut) ;`
`Check(TransactionRec, X"DDDD") ;`

Common Directive Transactions
`GetTransactionCount(TransactionRec, Count) ;`

For more information about OSVVM's Model Independent transaction capability see, Address_Bus_Model_Independent_Transactions_user_guide.pdf and Stream_Model_Independent_Transactions_user_guide.pdf.

5. The OSVVM Testbench Framework

The OSVVM testbench framework looks identical to other frameworks, including SystemVerilog. It includes verification components (AxiMaster, UartRx, and UartTx) and TestCtrl (the test sequencer) as shown in figure 2. The top level of the testbench connects the components together (using the same methods as in RTL design) and is often called a test harness. Connections between the verification components and TestCtrl use VHDL records as an interface. Connections between the verification components and the DUT are the DUT interfaces. Tests are written by calling transactions in TestCtrl. Separate tests are separate architectures of TestCtrl.

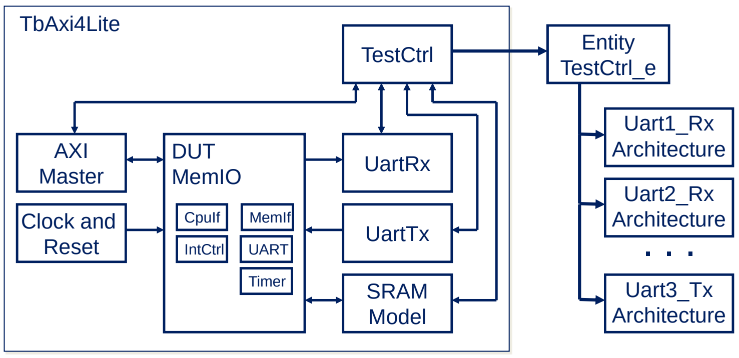

Figure 2. OSVVM Testbench Framework

The rest of paper focuses on writing tests in TestCtrl.

6. TestCtrl, The OSVVM Test Sequencer

The TestCtrl architecture consists of a control process plus one process per independent interface, see figure 3. The control process is used for test initialization and finalization. Each test process creates interface waveform sequences by calling the transaction procedures (Write, Send, …).

Each architecture of TestCtrl creates a separate test in the test suite. Hence, a single test is visible in a single file, improving readability.

Since the processes are independent of each other, synchronization is required to create coordinated events on the different interfaces. This is accomplished by using synchronization primitives, such as WaitForBarrier (from TbUtilPkg in the OSVVM library).

```
 architecture UartTx1  of TestCtrl is
   . . .
begin
  ControlProc : process
  begin
    . . .
    WaitForBarrier(TestDone, 5 ms) ;
    ReportAlerts ;
    std.env.stop;
  end process ;
  CpuTestProc : process
  begin
    WaitForBarrier(TestInit) ;
    Write(. . .) ;
    WaitForBarrier(DutInit);
    . . .
    WaitForBarrier(TestDone) ;
  end process ;
  TxProc : process
  begin
    WaitForBarrier(DutInit);
    Send(. . .) ;
    . . .
    WaitForBarrier(TestDone) ;
  end process ;
  . . .
```

Figure 3. TestCtrl Architecture

7. Test Initialization

The ControlProc both initializes a test and finalizes a test. Test initialization is shown in figure 4. SetAlertLogName sets the test name. Each verification component calls GetAlertLogID to allocate an ID that allows it to accumulate errors separately within the AlertLog data structure. Accessing the IDs here allows the message filtering of a verification component to be controlled by the test. WaitForBarrier stops ControlProc until the test is complete.

```
ControlProc : process
begin
  SetAlertLogName("Test_UartRx_1");
  TBID <= GetAlertLogID("TB");
  RxID <= GetAlertLogID("UartRx_1");
  SB.SetAlertLogId("UART_SB") ;
  SetLogEnable(PASSED, TRUE) ;
  SetLogEnable(RxID, INFO, TRUE) ;
  WaitForBarrier(TestDone, 5 ms) ;
  . . .
```

Figure 4. Test Initialization

8. A Simple Directed Test

A simple test can be created by transmitting (send) a value on one interface and receiving (Get) and checking (AffirmIfEqual) it on another interface. This is shown in figure 5.

```
TxProc : process

begin
  Send (TRec, X"10") ;


  Send (TRec, X"11") ;
  . . .

end process TxProc
```

```
RxProc : process
  variable RxD : ByteType;
begin
  Get(RRec, RxD) ;
  AffirmIfEqual(TBID, RxD, X"10");

  Get(RRec, RxD) ;
  AffirmIfEqual(TBID, RxD, X"11");
  . . .
end process RxProc ;
```

Figure 5. A Simple Directed Test

The AffirmIfEqual checks its two parameters. It produces a log "PASSED" message if they are equal and alert "ERROR" message otherwise. These are shown in figure 6.

```
%% Alert ERROR  In TB, Received: 08 /= Expected: 10 at 2150 ns
%% Log   PASSED In TB, Received: 11 at 3150 ns
```

Figure 6. Messaging from AffirmIfEqual

See the AlertLogPkg_user_guide.pdf and AlertLogPkg_quickref.pdf for more information about OSVVM's affirmation capability (AffirmIf, AffirmIfEqual, AffirmIfDiff, …).

9. Using Randomization

Constrained random randomly selects test values, modes, operations, and sequences of transactions. In general, randomization works well when there are a large variety of similar items to test.

The OSVVM package, RandomPkg, provides a library of randomization utilities. A subset of these is shown in figure 7.

```
-- Random Range:  randomly pick a value within a range
Data_slv8 := RV.RandSlv(Min => 0, Max => 15, 8) ;

-- Random Set: randomly pick a value within a set
Data1 := RV.RandInt( (1,2,3,5,7,11) ) ;

-- Weighted distribution:  randomly pick a value between 0 and N-1
-- where N is number of values in the argument
-- the likelyhood of each value = value / (sum of all values)
Data2 := RV.DistInt( (70, 10, 10, 5, 5) ;
```

Figure 7. Subset OSVVM's Random library

An OSVVM constrained random test consists of randomization plus code patterns plus transaction calls. For example, the code in figure 8 generates a UART test with normal transactions 70% of the time, parity errors 10% of the time, stop errors 10% of the time, parity and stop errors 5% of the time, and break errors 5% of the time.

```
TxProc : process
  variable RV : RandomPType ;
  . . .
for I in 1 to 10000 loop
  case RV.DistInt( (70, 10, 10, 5, 5) ) is
    when 0  =>    -- Nominal case   70%
      ErrorMode := UARTTB_NO_ERROR ;
      TxD:= RV.RandSlv(0, 255, Data'length) ;
    when 1  =>    -- Parity Error   10%
      ErrorMode:= UARTTB_PARITY_ERROR ;
      TxD:= RV.RandSlv(0, 255, Data'length) ;
    when  . . .  -- (2, 3, and 4)

  end case ;

  Send(UartTxRec, Data, ErrorMode) ;
end loop ;
```

Figure 8. An OSVVM Constrained Random Test

Hence, creating constrained random tests in OSVVM is simply a matter of learning the patterns. All of the pattern is written directly in the code, and hence, visible to review.

Constrained random introduces two issues to our testing. First, how do we self-check the test? Previously we recreated the transmit pattern on the receive side. Due to the complexity, this would be tedious and error prone. In the next section, we solve this problem by using OSVVM's generic scoreboards.

Second, how do we prove the test actually did something useful? We solve this problem by using OSVVM's functional coverage.

See the RandomPkg_user_guide.pdf and RandomPkg_quickref.pdf for more information about OSVVM's randomization capability.

10. OSVVM's Scoreboards

A scoreboard facilitates checking data when there is latency in the system. A scoreboard receives the expected value from the stimulus generation process and checks the value when it is received by the check process, as shown in figure 9.

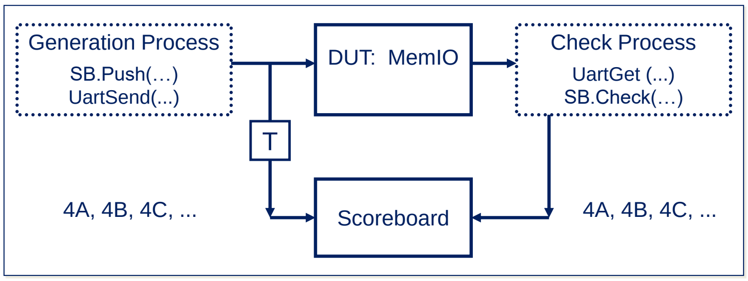

Figure 9. Scoreboard Block Diagram

The OSVVM scoreboard supports small data transformations, out of order execution, and dropped values. It uses package generics to allow the expected type and actual type to differ. The "match" function that determines if the expected and actual values match is also a package generic. The FIFO-like data structure of the scoreboard is created internal to a protected type.

The use model for OSVVM's scoreboard is shown in figure 10. The scoreboard instance is created using a shared variable declaration. On the transmit side (TxProc), the expected value is pushed into the scoreboard (SB.Push), and then a transaction is transmitted (Send). On the receive side (RxProc), the transaction is received (Get), and then the received value is checked in the scoreboard (SB.Check). This greatly simplifies RxProc since it no longer reproduces what the transmit side did. Scoreboards can also be used to simplify checking in directed tests.

```
use work.ScoreboardPkg_uart.all ;
shared variable SB : work.Scoreboardpkg_uart.ScoreboardPType;
```

```
TxProc : process
  . . .
begin
  for I in 1 to 10000 loop
    case RV.DistInt((. . .)) is
       . . .
     end case ;
    SB.Push((TxD, ErrorMode));
    Send(TRec, TxD, ErrorMode);
  end loop ;
  . . .
```

```
RxProc : process
  variable RxD   : ByteType;
  variable RV    : RandomPType;
begin
  for I in 1 to 10000 loop


    Get(RRec, RxD, RxErrorMode);
    SB.Check((RxD, RxErrorMode));
  end loop ;
  . . .
```

Figure 10. OSVVM Scoreboard Use Model

See the ScoreboardPkg_user_guide.pdf and Scoreboard_quickref.pdf for more information about OSVVM's scoreboard capability.

11. Adding Functional Coverage

Functional coverage is code that tracks items in the test plan. As such it tracks requirements, features, and boundary conditions.

If a test uses constrained random, functional coverage is needed to determine if the test did something useful. Going further as design complexity increases, functional coverage is recommended to assure that a directed test actually did everything that was intended.

There are two catagories of functional coverage: item (aka Point) coverage and cross coverage. Item coverage tracks relationships within a single object. For a UART, were transfers with no errors, parity errors, stop bit errors, parity and stop bit errors, and break errors seen?

Cross coverage tracks relationships between multiple objects. For a simple ALU, has each set of registers for input 1 been used with each set of registers for input 2?

Why not just use code coverage that is provided with a simulator? Code coverage only tracks code execution. Hence, code coverage cannot track the examples above since the information is not in the code. On the other hand, if a design's code coverage does not reach 100% then there are untested items and testing is not done. Hence, both code coverage and functional coverage are needed to determine when testing is done.

Functional coverage in OSVVM is implemented as a data structure within a protected type.

Figure 11 continues with RxProc from the constrained random test and adds functional coverage. First an instance of the coverage object (RxCov) is created using a shared variable. Next "RxCov.AddBins (GenBin(N))" is called to construct the functional coverage model. The value "N" corresponds to the integer representation of the UART status bits for Break, Stop, Parity, and Data Available. The calls to AddBins all complete at time 0, before any stimulus is generated or checked. Next, after the received stimulus has been retrieved (using Get), RxCov.Icover(RxErrorMode) is called to record the coverage. At the end of the test, RxCov.WriteBin prints the coverage results.

```
architecture CR_1 of TestCtrl is
  shared variable RxCov : CovPType ;  -- define coverage object
  . . .
begin
. . .
RxProc : process
  . . .
Begin
  -- Define coverage model
  RxCov.AddBins( GenBin(1) ) ;        -- Normal
  RxCov.AddBins( GenBin(3) ) ;        -- Parity Error
  RxCov.AddBins( GenBin(5) ) ;        -- Stop Error
  RxCov.AddBins( GenBin(7) ) ;        -- Parity + Stop
  RxCov.AddBins( GenBin(9, 15, 1) ) ; -- Break
  for I in 1 to 10000 loop
    Get(RRec, RxD, RxErrorMode);
    SB.Check((RxD, RxErrorMode));
    RxCov.ICover(RxErrorMode) ;       -- Collect functional coverage
  end loop ;
  . . .
  RxCov.WriteBin ;                    -- Print coverage results
```

Figure 11. UART RxProc with functional coverage added

See CoveragePkg_user_guide.pdf and CoveragePkg_quickref.pdf for more information about OSVVM's functional coverage capability.

12. Adding Protocol and Parameter Checkers

OSVVM alerts are used to check for invalid conditions on an interface or library subprogram. Alerts both report and count errors. Alerts have the levels FAILURE, ERROR (default), and WARNING. By default, FAILURE level alerts cause a simulation to stop. By default, ERROR and WARNING do not cause a simulation to stop. When a test completes, all errors reported by Alert (and AffirmIf) can be reported using ReportAlerts.

Figure 12 shows a protocol checker used in a memory model to detect if a write enable (iWE) and read enable (iOE) occur simultaneously while the memory is addressed (iCE). Parameter checkers are similar to protocol checkers and check for invalid parameters to library programs.

```
 SimultaneousAccessCheck: process
begin
  wait on iCE, iWE, iOE ;
  AlertIf(SramAlertID, (iCE and iWE and iOE) = '1',
      "nCE, nWE, and nOE are all active") ;
end process SimultaneousAccessCheck ;
```

Figure 12. Memory Model Protocol Checker

Alerts can be enabled (default) or disabled via a call to SetAlertEnable. The stopping behavior of Alert levels can be changed with SetAlertStopCount. Figure 13 shows the usage of both of these.

```
-- Turn off Warnings for a verification component
SetAlertEnable(UartRxAlertLogID, WARNING, FALSE) ;

-- If get 20 ERRORs stop the test
SetAlertStopCount(ERROR, 20) ;
```

Figure 13. Usage of SetAlertEnable and SetAlertStopCount

See the AlertLogPkg_user_guide.pdf and AlertLogPkg_quickref.pdf for more information about OSVVM's Alert capability (Alert, AlertIf, AlertIfEqual, SetAlertEnable, SetAlertStopCount, …).

13. Adding Message Filtering

OSVVM logs allow messaging to be turned on or off based on settings in the test – either globally or for a specific verification component. Logs have the levels ALWAYS, DEBUG, INFO, and PASSED. Logs print when enabled. Log ALWAYS is always enabled. The other logs are disabled by default. Figure 14 shows the usage of log and its output.

```
Log(TbID, "Test 1 Starting") ;
```

```
%% Log   ALWAYS   In Testbench, Test 1 Starting at 1770 ns
```

Figure 14. Log and its output

Logs are enabled or disabled using SetLogEnable. Figure 15 shows "PASSED" being enabled for the enitre testbench and "INFO" being enabled only for UartRx. Generally this is done in ControlProc at test initialization.

```
SetLogEnable(PASSED, TRUE) ;         -- Turn on PASSED for all models
SetLogEnable(RxID, INFO, TRUE) :    -- Turn on INFO for CpuID
```

Figure 15. SetLogEnable Usage

See the AlertLogPkg_user_guide.pdf and AlertLogPkg_quickref.pdf for more information about OSVVM's Alert capability (Log, SetLogEnable).

14. Test Finalization

Test finalization is error checking and reporting that is done in ControlProc after test completion. This is shown in figure 16. Finalization starts when "WaitForBarrier(TestDone, 5 ms)" resumes. This happens either when all of the test processes have called their corresponding WaitForBarrier (normal completion) or 5 ms passes. The 5 ms is a test timeout (watch dog) that activates if one of the test processes did not complete properly. The sequence of calls to AlertIf check for proper test finish conditions. ReportAlerts prints test results (see Test Wide Reporting).

```
ControlProc : process
begin
  . . .
  WaitForBarrier(TestDone, 5 ms) ;
  AlertIf(TBID, NOW >= 5 ms, "Test timed out") ;
  AlertIf(TBID, not SB.Empty, "Scoreboard not empty") ;
  AlertIf(TBID, GetAffirmCount < 1, "Checked < 1 items") ;
  ReportAlerts ;
  wait ;
end process ControlProc ;
```

Figure 16. Test Finalization

See the TbUtilPkg_user_guide.pdf and TbUtilPkg_quickref.pdf for more information about OSVVM's process synchronization capability (WaitForBarrier, WaitForClock, …).

15. Test Wide Reporting

The AlertLog data structure tracks FAILURE, ERROR, and WARNING for the entire test as well as for each AlertLogID (see GetAlertLogID). ReportAlerts prints a test completion message using this information. If GetAlertLogID was not called during the test, ReportAlerts prints either the simple PASSED or FAILED message shown in figure 17.

```
%% DONE  PASSED  Test_UartRx_1  Passed: 48  Affirmations Checked: 48  at 100000
ns
```

```
%% DONE  FAILED  Test_UartRx_1  Total Error(s) = 10  Failures: 0  Errors: 1
Warnings: 1  Passed: 48  Affirmations Checked: 48  at 100000 ns
```

Figure 17. Simple ReportAlerts

If GetAlertLogID was called during the test, ReportAlerts will include errors and passed for each AlertLogID as shown in figure 18.

```
%% DONE  PASSED  Test_UartRx_1  Passed: 48  Affirmations Checked: 48  at 100000
ns
```

```
%% DONE  FAILED  Test_UartRx_1  Total Error(s) = 7  Failures: 0  Errors: 7
Warnings: 0  Passed: 48  Affirmations Checked: 48  at 100000 ns
%%    Default                 Failures: 0  Errors: 0  Warnings: 0  Passed: 0
%%    OSVVM                   Failures: 0  Errors: 0  Warnings: 0  Passed: 0
%%    TB                      Failures: 0  Errors: 0  Warnings: 0  Passed: 0
%%    UART_SB                 Failures: 0  Errors: 0  Warnings: 0  Passed: 0
%%    Cpu_1                   Failures: 0  Errors: 7  Warnings: 0  Passed: 41
%%      Cpu_1 Data Error      Failures: 0  Errors: 7  Warnings: 0  Passed: 0
%%      Cpu_1 Protocol Error  Failures: 0  Errors: 0  Warnings: 0  Passed: 0
%%    UartRx_1                Failures: 0  Errors: 0  Warnings: 0  Passed: 0
```

Figure 18. ReportAlerts for each AlertLogID

See the AlertLogPkg_user_guide.pdf and AlertLogPkg_quickref.pdf for more information about OSVVM's ReportAlerts capability – this includes our capability to track requirements.

16. Including OSVVM Library

OSVVM provides context declarations (VHDL-2008) to allow the utility library and each verification component to be referenced with a single context reference, rather than multiple use clauses. This is shown in Figure 19.

```
library osvvm ;                      -- Utility Library
  context osvvm.OsvvmContext ;
library osvvm_axi4 ;                 -- AXI4 and AXI Stream
  context osvvm_axi4.Axi4LiteContext ;
  context osvvm_axi4.AxiStreamContext ;
Library osvvm_uart ;                 -- UART
  context osvvm_uart.UartContext ;
```

Figure 19. OSVVM Context References

17. Getting and Running OSVVM

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 20.

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

Figure 20. 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 21 shows the steps to do in your simulator to compile and run the tests for the AXI4 verification component. The tests for the AxiStream, AXI4Lite, and UART verification components are run in the same manner.

```
cd sim
do ../OsvvmLibraries/startup.tcl
build ../OsvvmLibraries
build ../OsvvmLibraries/AXI4/Axi4/testbench.pro
```

Figure 21. Compiling and Running OSVVM

18. Summary

OSVVM goes well beyond the basics shown in this article. To learn more, see the documentation on the GitHub site, or take SynthWorks' Advanced VHDL Testbenches and Verification class.

19. References

[1] https://blogs.mentor.com/verificationhorizons/blog/2019/01/15/part-6-the-2018-wilson-research-group-functional-verification-study/