OSVVM

Release 2020.05

Jim Lewis

Jul 19, 2020

MAIN DOCUMENTATION

Open Source VHDL Verification Methodology (OSVVM) provides utility and model libraries that simplify your FPGA and ASIC verification tasks. Using these libraries you can create a simple, readable, and powerful testbench that is suitable for either a simple FPGA block or a complex ASIC.

CHAPTER

ONE

OSVVM IS THE #1 VHDL VERIFICATION METHODOLOGY

According to the 2018 Wilson Verification Survey, OSVVM is the:

- #1 VHDL Verification Methodology
- #1 FPGA Verification Methodology in Europe (ahead of SystemVerilog + UVM)

CHAPTER

TWO

THE OSVVM UTILITY LIBRARY

The OSVVM utility library offers the same capabilities as those provided by other verification languages (such as SystemVerilog and UVM). For more see: OSVVM Utility Library

CHAPTER

THREE

THE OSVVM VERIFICATION IP LIBRARY (AKA VERIFICATION COMPONENTS)

The OSVVM model library is a growing set of models commonly used for FPGA and ASIC verification. For information more see: Documentation Comming Soon

3.1 OSVVM Utility Library Overview

3.1.1 Simplify your testbench development

OSVVM can be used in your current VHDL testbench, in part or in whole as needed. It allows mixing of our signature "Intelligent Coverage" methodology with other verification methodologies, such as directed, algorithmic, file based, and constrained random. Don't throw out your existing VHDL testbench or testbench models, re-use them.

There is no new language to learn. There are no specialized "OO" approaches – just plain old VHDL entities and architectures. As a result, it is accessible to RTL designers. In fact, it is our goal to make our testbenches readable to verification (testbench), design (RTL), system, and software engineers.

3.1.2 OSVVM Features and Capabilities

For me talking about OSVVM is like an Arlo Guthrie song – it is very easy to go on and on. Below is an overview of the various OSVVM packages. Details are filled in with links to blog posts. Additional details will be written in the future. To keep up, follow the RSS of the blog, follow SynthWorks on Twitter, or check back.

Transaction-Level Modeling

In OSVVM, we implement transaction-level models (what SystemVerilog calls a verification component), as an entity and architecture. To connect the transaction-level model to the testbench, we use records. To allow a single record to implement the transaction interface, we use the resoluton functions in OSVVM's ResolutionPkg. ResolutionPkg was first released in the 2016.11 OSVVM release and more blogs will be provided on it in the future.

Further information about OSVVM's Transaction-Level modeling capability:

> For more, see ResolutionPkg_user_guide.pdf.

Error logging and reporting – Alerts and Affirmations

OSVVM's AlertLogPkg simplifies and formalizes the signaling errors (at run time), counting errors, and reporting errors (summary at test completion). Indication of an error is done via a call to one of the Alert procedures (Alert, AlertIf, AlertIfNot, AlertIfEqual, AlertIfNotEqual, or AlertIfDiff). Alerts have the levels FAILURE, ERROR, or Warning. Each level is counted and tracked in an internal data structure. Within the data structure, each of these can be enabled or disabled. A test can be stopped if an alert value has been signaled too many times. Stop values for each counter can be set. At the end of a test, the procedure ReportAlerts prints a report that provides pass/fail and a count of the different alert values.

Further information about OSVVM's Alerts:

> Using Alerts For more, see AlertLogPkg_User_Guide.pdf.

Message filtering – Logs

Logs provide a mechanism to conditionally print information. Verbosity control allows messages that are too detailed for normal testing to be printed when specifically enabled. Logs have the levels ALWAYS, DEBUG, FINAL, and INFO. Through simulator settings, assert has this capability to a limited degree.

Further information about OSVVM's Logs:

> Using Logs For more, see AlertLogPkg_User_Guide.pdf.

Transcript files

OSVVM's transcript capability simplifies having different parts of a testbench print to a common transcript file. It does this by providing an internal file identifier (TranscriptFile), and subprograms for opening (TranscriptOpen) files, closing (TranscriptClose) files, printing (print and writeline), and checking if the file is open (IsTranscriptOpen).

Further information about OSVVM's Transcript Files:

> Testbench Transcripting using OSVVM. For more, see TranscriptPkg_user_guide.pdf.

Constrained Random test generation

OSVVM's RandomPkg provides a set of utilities for randomizing a value in a range, a value in a set, or a value with a weighted distribution. By itself this is not constrained random testing. OSVVM's constrained random capability uses these utilities plus code patterns to randomize a value, operation, or sequence that is valid in a particular test environment.

All constrained random testing is based on uniform randomization. Uniform randomization repeats values at a rate of log N, where N is the number of values generated. As a result, constrained random tests result in repeated stimulus – we have seen 5X or more for small problems.

In OSVVM, we use Intelligent Coverage randomization (see below) as our primary randomization methodology to avoid repeated stimulus, and constrained random as a refinement methodology in our tests.

Further information about OSVVM's randomization capability:

- RandomPkg Usage Basics: Protected Types, Seeds, and Randomization
- Basic Randomization Overloading
- Weighted Randomization
- Normal, Poisson, FavorBig, FavorSmall distributions
- Creating constrained random tests

- For more, see RandomPkg_user_guide.pdf.

Functional Coverage

Functional coverage is code that observes execution of a test plan. As such, it is code you write to track whether important values, sets of values, or sequences of values that correspond to design or interface requirements, features, or boundary conditions have been exercised.

Functional coverage is important for any randomized test generation approach since it is the only way to determine what the test has done. As the complexity of a design increases, 100% functional coverage assures us that all items in the test plan have been tested. Combine this with 100% code coverage and it indicates that testing is done.

Further information about OSVVM's Functional Coverage:

> The Basics of OSVVM's Point and Cross Functional Coverage: AKA, Functional Coverage Made Easy with VHDL's OSVVM Why you need functional coverage VHDL Functional Coverage is more capable than SystemVerilog For more, see CoveragePkg_User_Guide.pdf

Intelligent Coverage™ Randomization Methodology

Verification starts with a test plan that identifies all items in a design that need to be tested. OSVVM, like other advanced methodologies, uses functional coverage to observe conditions on interfaces and within the design to validate that the items identified in the test plan have occurred. As such, functional coverage helps determine when testing is done.

Unlike other methodologies, in OSVVM's Intelligent Coverage methodology, functional coverage is the prime directive – it is where we start our process. Intelligent Coverage is done in the following steps.

> Write a high fidelity functional coverage (FC) model Randomly select a hole in the functional coverage Refine the initial randomization with sequential code Apply the refined sequence (one or more transactions) Observe Coverage

The key point of Intelligent Coverage is that we randomize using the functional coverage. Then, if necessary, we refine the randomization using sequential code and any sequence generation method, including constrained random, algorithmic, directed, or file reading methods.

Further information about OSVVM's Intelligent Coverage Randomization:

> Intelligent Coverage Basics. AKA: Intelligent Coverage is 5X or More Faster than Constrained Random: . Weighted Intelligent Coverage. AKA: Functional Coverage Goals and Randomization Weights For more, see CoveragePkg_User_Guide.pdf

Utilities for testbench process synchronization

The OSVVM package, TbUtilPkg, provides testbench utilities for synchronizing processes, as well as, utilities for clock and reset generation.

Further information about OSVVM's Scoreboard and FIFO capability:

> For more, see TbUtilPkg_user_guide.pdf.

TbUtilPkg was first released in the 2016.11 OSVVM release and more blogs will be provided on it in the future.

Scoreboards and FIFOs (data structures for verification)

Scoreboards and FIFOs simplify test data checking when information flows from one part of a test to another with very little transformation.

Further information about OSVVM's Scoreboard and FIFO capability:

> For more, see ScoreboardPkg_user_guide.pdf.

ScoreboardPkg was first released in the 2016.11 OSVVM release and more blogs will be provided on it in the future.

Memory models

MemoryPkg simplifies the process of creating efficient data structures for memory models.

Further information about OSVVM's Scoreboard and FIFO capability:

> For more, see MemoryPkg_user_guide.pdf.

MemoryPkg was first released in the 2016.11 OSVVM release and more blogs will be provided on it in the future.

3.2 Index