World Class SystemVerilog & UVM Training

Sunburst Design - SystemVerilog UVM Verification Training

by Recognized Verilog & SystemVerilog Guru, Cliff Cummings of Sunburst Design, Inc.

Cliff Cummings is the only Verilog & SystemVerilog Trainer who helped develop every IEEE & Accellera Verilog, Verilog Synthesis and SystemVerilog Standard.

3 Days
70% Lecture, 30% Lab
Advanced Level

UVM is the unified future of SystemVerilog Verification

The good news is that the ***Universal Verification Methodology*** (**UVM**) is largely the same thing as the ***Open Verification Methodology*** (**OVM**) with a different first letter and a few enhancements including capabilities donated from VMM.

Course Objective

Make verification engineers knowledgeable, proficient and productive at UVM verification, using training materials developed by renowned Verilog & SystemVerilog Guru, Cliff Cummings.

Upon completion of this course, students will understand:

- SystemVerilog-verification language features
  - includes SystemVerilog classes & methods
  - includes SystemVerilog virtual classes & virtual methods
  - includes SystemVerilog interfaces and virtual interfaces
  - includes SystemVerilog constrained random testing
  - includes SystemVerilog functional coverage
  - includes SystemVerilog stimulus driving and verification sampling strategies
- UVM-verification language capabilities
  - includes UVM fundamentals and running tests
  - includes UVM base classes and reporting
  - includes UVM creating and properly starting tests
  - includes UVM testbench components and their usage
  - includes UVM transactions, sequences and virtual sequences
  - includes UVM override techniques and strategies

For more information, contact:
Cliff Cummings - cliffc@sunburst-design.com - Sunburst Design, Inc. - 801-960-1996

Course Overview

Sunburst Design - SystemVerilog UVM Verification Training is a 3-day, fast-paced intensive course that focuses advanced verification features using SystemVerilog and the UVM base class libraries.

Why is UVM hard to learn?

Many engineers believe they can learn UVM by picking up and reading a book and the UVM User Guide. They quickly discover this is exceptionally difficult to do. To learn why it is so hard to learn UVM from existing materials, see the Appendix notes at the end of this syllabus.

Good UVM training should address each of the issues that make UVM materials difficult to understand (as described in the Appendix notes).

Target Audience

Sunburst Design - SystemVerilog UVM Verification Training is intended for design & verification engineers who require UVM verification methodology training.

Prerequisites (mandatory)

This is a very advanced SystemVerilog design class that assumes engineers already have a good working knowledge of both Verilog and SystemVerilog. Engineers with no prior HDL training or experience will struggle in this class.

Classroom Details

Training is generally conducted at customer facilities and is sometimes offered as an open-enrollment training class. For maximum effectiveness, it is recommended to have one workstation or PC for every two students, with your preferred SystemVerilog simulator licenses (we often can help provide the simulator and temporary training licenses).

Please contact Cliff Cummings to customize the training materials to meet the needs of your engineering team.

For more information, contact:
Cliff Cummings - cliffc@sunburst-design.com - Sunburst Design, Inc. - 801-960-1996

Course Syllabus

Day One

OVM/UVM Resources & Introduction
Section Objective: Share OVM/UVM resources - There are conflicting guidelines from multiple resources regarding OVM/UVM methodologies. When one understands why there are differences, it is easier to learn from the divergent resources. This section explains the rationale behind the differing resources.

- OVM resources
- UVM resources
- UVM introduction
- UVM conflicting recommendations - why?

Classes & Class Variables
Section Objective: Learn class basics - OVM and UVM are class libraries used to construct powerful verification environments. Class fundamentals are described in this section.

- SystemVerilog class basics
- Traditional Object Oriented (OO) programming -vs- SystemVerilog Classes
- Class definition & declaration
- Class members (data) & methods (tasks & functions)
- Class handles & using class handles
- Built-in class object constructor - new()
- super & this keywords
- Assigning object handles
- User-defined constructors
- Class extension & inheritance
- Class extension - adding properties & methods
- Class extension - overriding base class methods
- Assigning class handles
- Assigning extended handles to base handles
- Casting base handles to extended handles (technique used in UVM)
- Chaining new() constructors - illegal new() constructors
- Overriding class methods
- Extending class methods
- Extern methods
- local & protected keywords
- LAB - Classes & Methods

For more information, contact:
Cliff Cummings - cliffc@sunburst-design.com - Sunburst Design, Inc. - 801-960-1996

UVM Overview – First Pass
Section Objective: Learn fundamentals of UVM testbench development and execution. This section briefly introduces important UVM fundamentals followed by a lab to help students become familiar with UVM concepts. Students will not fully understand all of the concepts in this section, but it is important that students do the lab to build a foundation for later learning. Each of the concepts in this section will be taught with greater detail in later sections. Engineers will learn more quickly after they have experienced the lab techniques at least once before exploring advanced UVM concepts.

- UVM transactions (data)
- Components (testbench components)
- Display command
- Top module DUT, interface, interface wrapper
- Testbench classes: environment, sequencer, driver, monitor, virtual interface
- Test classes
- Running tests using +UVM_TESTNAME command line switch
- LAB - UVM Setup (optional)
- LAB - UVM Common Errors
- LAB - First UVM Lab

Virtual Classes, Virtual Methods and Virtual Interfaces
Section Objective: Learn fundamentals of virtual classes/methods/interfaces - Virtual classes enable the creation of a set of base classes that provide a template for advanced verification environments. UVM is a base class library made up of mostly virtual classes that the user extends to create a reusable testbench environment. Virtual methods allow run-time base-method replacement that is a vital part of the UVM strategy (polymorphism).

- Introduction to Virtual - three types of "virtual"
- Virtual/abstract classes
- Legal & illegal virtual class usage
- Virtual class methods & restrictions
- Virtual Methods and rules
- Virtual -vs- non-virtual method override rules
- Why use virtual methods?
- Polymorphism using virtual methods
- Pure virtual methods (SystemVerilog-2009 update - used by UVM)
- Pure constraints (SystemVerilog-2009 update)
- Passing type parameters

Random & Constrained Random Class Variables
Section Objective: Learn about class variable randomization and setting constraints on that randomization - OVM and UVM use classes and constrained random variables for the construction of constrained random testing environments. Randomization and constraint fundamentals are described in this section.

- Directed -vs- random testing
- rand & randc class variables
- randomize() method - Randomizing class variables
- pre_randomize()/post_randomize() methods
- randomize ... with
- rand_mode()
- Randomization constraints
- Simple constraints
- Constraints blocks
- Important constraint rules
- Constraint distribution & set membership - dist & inside
- Constraint distribution operators
- External constraints & usage rules
- LAB - Random Variables & Randomization
- LAB - Constrained Random Stimulus

UVM Base Classes & Reporting (standard UVM print/display commands)
Section Objective: Learn about UVM base classes and basic display and reporting commands.

- UVM Base Classes
- Introduction to UVM core base classes, `include files and macros
- Block diagram of DUT-testbench structure
- UVM verification components
- UVM components and objects
- UVM transactions (passing UVM data & methods - dynamic class objects)
- UVM factory basics
- Reporting methods & arguments
- LAB - UVM Messaging

For more information, contact:
Cliff Cummings - cliffc@sunburst-design.com - Sunburst Design, Inc. - 801-960-1996

Day Two

UVM Transaction Base Classes
Section Objective: Learn to use and manipulate UVM transactions - This section answers important questions related to transactions including the basic question, why do transactions have to be classes?

- Why classes -vs- structs?
- Dynamic transaction classes
- `uvm_object_utils macro
- uvm_sequence_item -vs- uvm_transaction
- Field macros
- Randomizable data members
- Randomizable knobs
- Randomization constraints
- uvm_object constructors
- UVM sequence body task
- start_item(tx) - finish_item(tx)
- randomize() the transaction
- randomize() the transaction with inline constraints
- UVM sequences of uvm_sequence_item and uvm_sequence

Top Module & DUT
Section Objective: Learn how to connect an UVM class-based testbench to an actual Design Under Test (DUT) - This section explains the role that interfaces, virtual interfaces and configuration tables play in a testbench environment.

- Top module
- DUT (Design Under Test)
- DUT Interface
- Connecting DUT to DUT interface
- DUT interface handle
- DUT interface wrapper (class-based wrapper)
- Configuration tables
- set/get_config_object (old method to store the DUT interface handle)
- uvm_config_db#(type) set/get (new/easier method to store the DUT interface handle)
- Virtual interfaces for verification

UVM Testbench Driver-Components
Section Objective: Learn to use UVM drivers, sequencers, agents and environments - Setting up the driver is a critical step. The class-based driver must drive the module-based DUT through a virtual interface that drives a real interface.

- UVM components to build the testbench structure
- UVM testbench structure (quasi-static class objects)
- `uvm_component_utils macros
- uvm_component constructors
- UVM components connected through ports & exports
- Testbench driver (get-port configuration)
- Managing the virtual interface - config table - required dynamic casting
- Testbench sequencer (get-export configuration)
- Testbench agent & environment
- User-defined testbench package
- LAB - UVM Sequencer-Driver

UVM Testbench Monitor, Analysis Ports & Checkers
Section Objective: Learn to use UVM monitors, analysis ports and checkers - UVM uses monitors, analysis ports and checkers (and scoreboards) to capture DUT outputs and analyze the outputs using functional coverage for correctness and completeness.

- UVM testbench components to capture and examine outputs
- UVM checkers and functional coverage collection through analysis ports
- Testbench monitor in the testbench agent
- Why do we need copy and compare methods?
- Checkers and scoreboards
- Functional coverage collector basics
- LAB - UVM Agent & Virtual Interface
- LAB - UVM Analysis Ports & Coverage
- LAB - UVM Scoreboard Connections

For more information, contact:
Cliff Cummings - cliffc@sunburst-design.com - Sunburst Design, Inc. - 801-960-1996

UVM Scoreboards

Section Objective: Learn two techniques for creating self-checking scoreborads. The first technique is commonly shown in literature and uses 2 uvm_analysis_imp_ports and 2 uvm_tlm_fifos, which requires the use of special macros. The second scoreboard technique uses pre-coded scoreboard wrapper, predictor with extern calc-expected function, and pre-coded comparator with 2 uvm_tlm_analysis_fifos. The second technique only requires completion of the extern calc_expected function.

- What is the job of the scoreboard
- Scoreboard architecture #1
- Multiple analysis implementation ports
- `uvm_analysis_imp_decl macros
- Scoreboard architecture #2
- Pre-coded scoreboard wrapper and predictor
- Extern calc_exp function - requires user to complete this function
- Pre-coded comparator with 2 uvm_tlm_analysis_fifos
- Functional coverage collector basics
- LAB - DUT Test & UVM Scoreboard
- LAB - UVM Scoreboard - Predictor/Comparator

For more information, contact:
Cliff Cummings - cliffc@sunburst-design.com - Sunburst Design, Inc. - 801-960-1996

Day Three

Fundamentals of Running & Stopping UVM Tests
Section Objective: Learn proper methods to start and gracefully terminate UVM tests. This is a poorly documented topic in existing reference materials. Guidelines are presented to help properly stop tests using standard UVM techniques.

- Including UVM source files, base classes and macros
- Importing ovm_pkg/uvm_pkg
- run_test() fundamentals
- UVM phase basics
- Recommended uvm.f and run.f command files (Compilation)
- Selecting tests using +UVM_TESTNAME command line switch (Simulation)
- Declaring environments in tests
- uvm_component (test) constructors
- Test - run task & sequencer startup
- global_stop_request
- uvm_test_done objection
- LAB - Stopping OVM & UVM Tests

UVM Factory, Constructors & Transaction Level Modeling (TLM) Basics
Section Objective: Learn the basics of UVM factories, registration, class construction and introduce the concept of factory overrides. This section will show why factories are important to UVM testbenches and discuss new() -vs- type_id::create() methods. This section also details how transactions are passed between classes through the use of ports, exports, put-configurations, get-configurations and transport configurations

- UVM factory basics
- Why is a factory used in UVM
- What is needed to use the factory
- new() -vs- type_id::create() construction
- Component and data lookup from the factory
- Running without re-compilation
- Tests can make substitutions without changing the testbench source code
- Introduction to factory overrides
- TLM ports & exports
- Why "ports" and "exports"
- TLM put, get and transport configurations
- Transaction-level control flow
- Transaction-level data flow
- Transaction-level transaction type
- Put configurations
- Get configurations
- Transport configurations

Fork-Join Enhancements and Advanced UVM Sequence Generation
Section Objective: Learn advanced sequence generation techniques - New fork-join capabilities were added to SystemVerilog and they are commonly used by advanced UVM sequence generation environments.

- New SystemVerilog fork-join processes
- UVM virtual sequences
- Virtual sequencers & virtual sequences requirements
- m_sequencer, p_sequencer, `uvm_declare_p_sequencer
- Virtual sequence base class details
- Common test_base
- Starting virtual sequences
- Multi-bus virtual sequencer example
- LAB - Virtual Sequencer & Sequences

Clocking Blocks and Verification Timing
Section Objective: Learn important stimulus and verification timing issues and techniques - SystemVerilog clocking blocks help control timing for advanced UVM verification environments.

- Testbench stimulus/verification vector timing strategies
- #1step sampling
- Clocking blocks
- Clocking skews
- Default clocking block cycles
- Clocking block scheduling
- UVM usage of clocking blocks in an interface
- UVM driver timing using clocking blocks
- UVM signal sampling using clocking blocks

Functional Coverage
Section Objective: Learn functional coverage fundamentals - Functional coverage is used to track what has been tested. Functional coverage is used to help answer the question, "are we done testing?"

- Code coverage -vs- functional coverage
- Covergroups & coverpoints
- Auto-bins & user-named bins
- User-named array of bins
- Cross coverage
- Covergroup.sample() method
- Transition bins
- Coverage options & coverage capabilities
- LABS - Continue working on incomplete labs

For more information, contact:
Cliff Cummings - cliffc@sunburst-design.com - Sunburst Design, Inc. - 801-960-1996

APPENDIX

Why are OVM & UVM hard to learn?

Many engineers believe they can learn UVM by picking up and reading a book and the OVM or UVM User Guide. They quickly discover this is exceptionally difficult to do. Why is it so hard to learn UVM from existing materials?

Through years of experience, Sunburst Design has identified the following reasons why engineers struggle with existing UVM tutorial materials:

1) The OVM User Guide was written by Cadence and teaches Cadence recommended methods, which includes the use of a large number of OVM macros.
2) The OVM tutorials on VerificationAcademy.org are shown using Mentor recommended methods, which includes the use of fewer OVM macros and more OVM method calls.
3) The OVM Cookbook was written by Mentor employees and is based on an earlier version of OVM (the latest techniques are not shown in the book).
4) The above User Guide, tutorials and Cookbook do not acknowledge or explain the alternate methods, so users are left to draw erroneous conclusions that some of the methods shown are flawed, which is not true. Learners need to be taught the pros and cons of the alternate methods so that they understand why there are differences in the various methods presented.
5) All the people who have written OVM materials are *really*, *really* smart software engineers who assume that engineers already understand SystemVerilog syntax and semantics, object oriented programming and polymorphism semantics, and they don't know how teach these concepts to beginners.
6) Many of those who have written OVM materials are software engineers who do not have a strong grasp of good hardware design practices, and it shows in many of the examples.
7) The OVM User Guide (chapter 2) and the OVM Cookbook (chapter 3) introduce Transaction Level Modeling (TLM) concepts, including put, get and transport communication, but do a poor job of tying the concepts into the rest of the OVM materials. Engineers often wonder why TLM was introduced in these texts.
8) All OVM materials show the driver on the right and the monitor on the left (right to left data-flow inside of the agent). This contradicts known good hardware block diagramming methods (data should flow from left to right in block diagrams) and adds an unnecessary level of confusion to the learning process for those who are familiar with good block diagramming techniques.
9) There is a huge shortage of complete simple examples. Most of the publicly available example code is in abbreviated code-snippet form, leaving the new user to guess what is missing. Finding full examples in the materials is rare. One notable example shows OVM used on a large VHDL design, which introduces yet another unknown to the learning process.
10) Of course, you must understand classes, class-extension, virtual classes, virtual methods, dynamic casting, polymorphism, randomization, constraints, covergroups, coverpoints, interfaces and virtual interfaces before you can learn OVM. Too many engineers try to learn OVM without a full understanding of these SystemVerilog fundamentals (this is not the fault of OVM authors).
11) Classes are applied as stimulus and sampled for verification. Existing materials do not explain why classes are used instead of structs?
12) Interfaces, virtual interfaces and their recommend usage-models are somewhat buried in the materials and are poorly explained (most authors assume you understand these concepts without much explanation - they are wrong).
13) There are a significant number of typos and mistakes sprinkled throughout the materials and examples. The mistakes leave the learner to try to figure out which coding styles are correct and which have typos.

Sunburst Design UVM training addresses each of these issues.

For more information, contact:
Cliff Cummings - cliffc@sunburst-design.com - Sunburst Design, Inc. - 801-960-1996