World Class SystemVerilog & UVM Training

Sunburst Design - UVM Training

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

NOTE from Cliff Cummings:

UVM is **VERY DIFFICULT** to learn on your own with self-paced training. I know, I tried!!

Some engineers are tempted to skip this course and go straight to the Advanced UVM Training course offering. For many engineers, this is a mistake! Engineers who have taken SystemVerilog for Verification training are prepared for UVM training but not Advanced UVM training.

I have not found any good UVM book that properly and adequately teaches UVM for beginners. Most books assume that the reader is already an expert at SystemVerilog, Object Orient Programing (OOP) and Polymorphism before they start to read the books. Most books do not explain why UVM works the way it does. They just teach engineers to copy a pattern, which is a recipe for failure.

The title of this course is deceiving. Although the course title frequently includes either "Standard" or "Fundamentals," this UVM training is fast paced and advanced. Just a few weeks ago I had a student take this training who had already taken other UVM training. At the end of the course she commented that she now understood how UVM works and how to use it properly.

To take Advanced UVM training, engineers should either first take this training course, or should have two years of UVM Verification Experience. I have taught this UVM training course to engineers who have done UVM verification for multiple projects and claimed to have understood SystemVerilog classes and Object Oriented Programming. These are the engineers that ask the most questions in this training class because they can finally get answers to how and why SystemVerilog and UVM behave the way they do.

UVM topics need to be taught multiple times to understand how they work. That is why this course requires students to start doing full, self-checking UVM testbenches on the first day of class and includes 10 full self-checking labs among the 13 labs included in the course. The goal is to gain experience building and simulating multiple full, self-checking UVM testbenches.

If you are new to UVM or have less than 2 years of UVM experience with no formal UVM training, this is the course you should take.

This is the course I wish I could have taken when I was learning UVM!

Course Syllabus

All scheduled times are estimates only and adjusted for Irish Training sessions
~10 minute breaks near the top of each hour
(Lab time is scheduled for "Lunch & Lab" and near the end of the day)

Day One

UVM Resources & Introduction (Includes class introductions – 11:00-11:15 am)
Section Objective: Share UVM resources - There are conflicting guidelines from multiple resources regarding 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.

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

(1) Classes & Class Variables (11:15 am – 12:15 pm - part of this section will be moved to after lunch)
Section Objective: Learn class basics - UVM is a class library 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 by UVM)
- Chaining new() constructors - illegal new() constructors
- Overriding class methods
- Extending class methods
- Extern methods
- Static methods
- local & protected keywords

(2) UVM Overview First Pass & uvmtb_template files (12:15 pm – 1:00 pm)
Must complete this section before lunch to build the first full UVM testbench during lunch
Section Objective: Learn fundamentals of UVM testbench development and execution. This section briefly introduces important UVM fundamentals followed by a lab to help students with first-pass familiarity and introduction to UVM testbench development. Students will not fully understand all of the section concepts while creating the first UVM testbench, 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 a second time 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. This section also introduces the uvmtb_template files for rapid UVM testbench development.

- 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
- Stopping tests using raised & dropped objections
- uvmtb_template files
- The 8 template files that require modification for simple block-level verification
- LAB - UVM Common Errors
- LAB - UVM First Testbench - Testing a Counter *(Full UVM self-checking testbench #1)*

(LUNCH & LAB: 1:00 – 2:00 pm)

(1 - cont.) Complete section 1. (~2:00 – 2:30 pm)
(3) Virtual Classes, Virtual Methods and Virtual Interfaces (~2:30 – 3:30 pm)
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)
- Interfaces and virtual interfaces for UVM testbench development
- Passing type parameters

(4) Constrained Random Testing and Functional Coverage Part I (~3:30 – 4:30 pm)
Section Objective: Introduction to class variable randomization and setting randomization constraints - UVM uses classes and constrained random variables for the construction of constrained random testing environments. Introduction to functional coverage including covergroups and coverpoints. An introduction to constrained random testing and functional coverage 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 *(Full UVM self-checking testbenches #2-3)*
- LAB - Constrained Random Stimulus *(Full UVM self-checking testbench #4)*

(5) UVM Base Classes & Reporting (UVM print/display commands) (~4:30 – 5:30 pm)
Section Objective: Learn about UVM base classes and basic display and reporting commands. 3-day class includes introduction to SystemVerilog dynamic & associative arrays.

- SystemVerilog dynamic arrays
- SystemVerilog associative arrays
- 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
- Reporting macros and why they are preferred
- UVM_VERBOSITY explained
- Why the UVM User Guide, Reference Manual and Books get VERBOSITY wrong!
- LAB - UVM Messaging

Day Two

(6) UVM Transaction Base Classes, Sequences & Tests (11:00 am – 1:00 pm)
Section Objective: Learn to use and manipulate UVM transactions. This section shows why transactions are classes and not structs. ***This section also shows the two common techniques to define standard transaction methods, as well as the two common techniques to execute transactions and sequences, along with pros, cons and benchmarks of each method.*** This section then shows techniques to define and run sequences and tests.

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

(LUNCH & LAB: 1:00 – 2:00 pm)

(7) Top Module & DUT (2:00 – 3:00 pm)
Section Objective: Learn how to connect a UVM class-based testbench to an actual Design Under Test (DUT) - This section explains the role that interfaces, virtual interfaces, configuration tables and the UVM configuration database play in a testbench environment.

- Top module
- DUT (Design Under Test)
- DUT Interface
- Connecting DUT to DUT interface
- DUT interface handle
- uvm_config_db#(type) set/get (new/easier method to store the DUT interface handle)
- Configuration tables
- set/get_config_object (old method to store the DUT interface handle)
- Virtual interfaces for verification
- LAB - UVM Agent (Sqr-Drv-Mon) *(Full UVM self-checking testbench #5)*

(8) UVM Testbench Agent – Sequencer / Driver / Monitor (3:00 – 4:30 pm)
Section Objective: 3/4-day class includes introduction to SystemVerilog queues. Learn to use UVM environments, agents, sequencers, drivers, and monitors - 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 uses monitors to sample DUT signals through the virtual interface and captures the transaction that is then broadcast through an analysis port to a scoreboard and coverage collector.

- 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
- UVM analysis ports
- Analysis port broadcast command
- UVM monitors with analysis ports
- UVM agents with analysis ports
- Active and passive agents
- uvm_subscriber with analysis export
- Connecting a coverage collector using an analysis export
- LAB – FIFO Gray Code Pointer - *(Full UVM self-checking testbench #6)*

(9) UVM Scoreboards - Part I (4:30 – 5:45 pm)
Section Objective: This section starts off with a tutorial about SystemVerilog queues & mailboxes, then describes the uvm_tlm_fifos and how they are used. Next, learn the first scoreboard technique uses pre-coded scoreboard wrapper, predictor with extern calc-expected function, and pre-coded comparator with 2 uvm_tlm_analysis_fifos. The first technique only requires completion of the extern calc_expected function.

- SystemVerilog queues & mailboxes
- uvm_tlm_fifo & uvm_tlm_analysis_fifo
- What is the job of the scoreboard
- Scoreboard architecture #1
- 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
- LAB – UVM Scoreboard Style #1 - Barrel Shifter - *(Full UVM testbench lab #7)*
- LAB – UVM Scoreboard Style #1 - Pipeline Design - *(Full UVM testbench lab #8)*

Day Three

(10) UVM Scoreboards - Part II (11:00 - 11;45 am)
Section Objective: Learn the second scoreboard technique, which is commonly shown in literature and uses 2 uvm_analysis_imp_ports and 2 uvm_tlm_fifos. the second technique requires the use of special macros to allow using two analysis imp ports on a common component.

- Scoreboard architecture #2
- Multiple analysis implementation ports
- `uvm_analysis_imp_decl macros
- Full scoreboard style #2 code and description.
- LAB – UVM Scoreboard Style #2 - 2 Analysis Imp Ports - *(Full UVM testbench lab #9)*

(11) Fork-Join Enhancements & UVM Virtual Sequence Generation (11:45 am – 1:00 pm)
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

(LUNCH & LAB: 1:00 – 2:00 pm)

(12) Clocking Blocks and Verification Timing (2:00 – 3:00 pm)
Section Objective: Learn important stimulus and verification timing issues and techniques - SystemVerilog clocking blocks help control timing for 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

(13) Transaction Level Modeling (TLM) Basics & UVM Factory & Constructors (3:00 – 3:45 pm)
Section Objective: Transaction Level Modeling (TLM) is taught after it has been used the first two days of UVM training. This section shows how transactions are passed between classes through ports, exports, put-configurations, get-configurations and transport configurations.

- 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

(14) UVM Factory, Constructors & Factory Overrides (3:45 – 4:30 pm)
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 describe differences between new() -vs- type_id::create() methods.

- 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

(15) Constrained Random Testing and Functional Coverage Part II (4:30 – 5:30 pm)
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?" This section includes cover statements & compares them to covergroup coverage.

- 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
- Comparing cover to covergroup coverage
- LAB – UVM Functional Coverage - *(Full UVM testbench lab #10)*

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.