SystemVerilog UVM Verification Training & UVM RAL

by

Recognized Verilog & SystemVerilog Guru, Cliff Cummings of Paradigm Works, Inc.

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

Course Summary

Course Description

The #1 priority in this course is for engineers to complete and understand as many full UVM self-checking testbenches as time permits.

SystemVerilog UVM Verification Training is a 3-day, fast-paced, intensive course that focuses on advanced verification using UVM, and is intended for design & verification engineers who require UVM verification methodology training.

To become proficient at UVM verification, engineers need to gain experience coding multiple full, self-checking UVM testbenches. Engineers will use the Sunburst Design uvmtb_template files to rapidly develop 10 full, self-checking block-level UVM testbenches and other labs. This UVM training is unique because it explains the powerful, simple uvm_resource_db API and why engineers have been using the ***wrong*** uvm_config_db API for more than 10 years

Students will learn, use, and understand the following:

- SystemVerilog language verification language features
  - Includes SystemVerilog classes & methods
  - includes SystemVerilog classes & methods
  - includes SystemVerilog virtual classes & virtual methods
  - includes SystemVerilog interfaces and virtual interfaces
  - includes SystemVerilog constrained random testing
  - Students will have a good understanding of SystemVerilog RTL design features.
  - UVM – verification language capabilities
  - includes UVM fundamentals and running/stopping tests
  - includes UVM resources and the simple and powerful uvm_resource_db API
  - includes UVM correct messaging techniques and recommendations
  - includes UVM transactions, sequences, and virtual sequences
  - includes UVM testbench components and their usage
  - includes UVM scoreboard with predictor, comparator & reporting
  - includes UVM template files for rapid UVM testbench development
  - includes UVM factory, factory override, and usage

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 issue that makes UVM materials difficult to understand (as described in the Appendix notes)

Duration	4 days
Breakdown	70% Lecture, 30% Lab
Level	Beginner to Advanced
Prerequisites	**This is a very advanced SystemVerilog training 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.
Online Details	The course delivery can be in-person or virtual (virtual courses are convenient for both U.S. and non-U.S. engineers).

Course Syllabus

Half Day #1

(1) SystemVerilog Enhancements & Methodology Overview *(Not lectured)*

- Includes a quick review of SystemVerilog resources available to design & verification engineers.
  - UVM resources
  - UVM introduction
  - UVM conflicting recommendations - why?

(2) Classes, Virtual Classes, Virtual Methods

- Section Objective: Learn class basics - UVM is a class library used to construct powerful verification environments. Classes, virtual classes and virtual method 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
  - 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)
  - 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

(3) UVM Overview First Pass & uvmtb_template files

- The Section Objective: Learn fundamentals of UVM testbench development and execution. This section briefly introduces key UVM fundamentals, followed by a lab to help students gain first-pass familiarity and an introduction to UVM testbench development. Students will not fully understand all the section concepts while creating the first UVM testbench, but it is important that they complete 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)*

Half Day #2

(4) Advanced Class Topics and Virtual Interfaces

- Section Objective: Additional advanced class topics, including extern methods and singleton patterns - Virtual interfaces are used to connect a class-based testbench to a static design.
  - local & protected keywords
  - Static methods
  - Extern class methods
  - Singleton pattern used in UVM
  - The need for virtual interfaces
  - Virtual interface styles
  - Interface/DUT-port connections
  - Interfaces and virtual interfaces for UVM testbench development
  - Passing type parameters

(5) Constrained Random Testing and Functional Coverage Part I

- 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. This section introduces constrained random testing and functional coverage.
  - 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
  - Introduction to covergroups and coverpoints
  - Basic covergroup and coverpoint usage
  - LAB - Random Variables & Randomization *(Full UVM self-checking testbenches #2-3)*

(6) UVM Base Classes & Reporting (UVM print/display commands)

- Section Objective: Learn to use and manipulate UVM transactions. This section explains why transactions are classes, 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, knobs & 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

Half Day #3

(7) UVM Transaction Base Classes, Sequence & Tests

- Section Objective: Learn to use and manipulate UVM transactions. This section explains why transactions are classes, 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, knobs & 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

(8) Top Module, DUT & uvm_resource_db / uvm_config_db

- 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 resource database APIs play in a testbench environment.
  - Top module
  - DUT (Design Under Test)
  - DUT Interface
  - Connecting DUT to DUT interface
  - DUT interface handle

- 90%+ of UVM verification engineers are using the wrong database API
- uvm_resource_db#(type) set/read_by_name API
- uvm_config_db#(type) set/get API
- Why uvm_resource_db API is easier and more powerful than uvm_config_db API
- 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)*

Half Day #4

(9) UVM Testbench Agent – Sequencer / Driver / Monitor

- Section Objective: includes an 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 via a virtual interface that connects to a real interface. UVM uses monitors to sample DUT signals via the virtual interface, capture transactions, and then broadcast them through an analysis port to a scoreboard and a 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)*

(10) UVM Scoreboards – Part I

- Section Objective: 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 sb_calc_expected function.
  - SystemVerilog queues
  - SystemVerilog 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 sb_calc_exp function - requires the 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)*

Half Day #5

(11) Clocking Blocks and Verification Timing

- 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

(12) 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

(13) Fork-Join & UVM Scoreboards – Part II

- Section Objective: The second scoreboard technique is commonly shown in online examples and uses 2 uvm_analysis_imp_ports and 2 uvm_tlm_fifos, which requires the use of special macros. This section starts with a tutorial on SystemVerilog queues & mailboxes, then describes uvm_tlm_fifos and how they are used. The second scoreboard technique is then described.
  - Scoreboard architecture #2
  - Multiple analysis implementation ports
  - `uvm_analysis_imp_decl macros
  - LAB – UVM Scoreboard Style #2 - 2 Analysis Imp Ports *(Full UVM testbench lab #9)*

Half Day #6

(14) Transaction Level Modeling (TLM) Basics

- Section Objective: Transaction-Level Modeling (TLM) is taught after it is used for 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

(15) UVM Factory & Constructors

- Section Objective: Learn the basics of UVM factories, registration, class construction, and introduce the concept of factory overrides. This section explains why factories are important to UVM testbenches and describes the differences between the new() and type_id::create() methods.
  - UVM factory basics
  - Why is a factory used in UVM
  - 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

(16) Constrained Random Testing and Functional Coverage Part II

- Section Objective: Learn functional coverage fundamentals - Functional coverage is used to track what has been tested. Functional coverage helps answer the question, "Are we done testing?" This section includes cover statements & compares them to the 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)*

Half Day #7

(17) UVM Register Abstraction Layer (RAL)

- UVM package to describe and access registers by name instead of by address
  - Register package overview
  - Register models & memories
  - Register fields definition: uvm_reg_field
  - Register full definition: uvm_reg
  - Register memories definition: uvm_reg_mem
  - Collection of registers definition: uvm_reg_block
  - Register map definition: uvm_map
  - UVM register base classes
  - Register definition – configure() method
  - Register backdoor access
  - Tool-generated register packages and classes
  - LABS – UVM Register Design (with & without RAL) - *(Full UVM testbench labs #11 & 12)*

Half Day #8

(18) UVM Register Abstraction Layer (RAL) – More Detail

- UVM package adapter, predictor, and model as used by the UVM register package
  - Register adapter - derivative of uvm_object
  - Register predictor - derivative of uvm_component
  - Register model - derivative of uvm_component
  - The RAL API
  - Built-In register sequences
  - LABS – UVM Register Design (with & without RAL) - *(Continue working UVM testbench labs #11 & 12)*

Appendix – Why are OVM & UVM Hard to Learn?

Many engineers believe they can learn UVM by picking up a book and reading the OVM or UVM User Guides. 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 UVM User Guide was written by Cadence and teaches Cadence-recommended methods, which include the use of a large number of UVM macros.
2) The UVM tutorials on VerificationAcademy.org are shown using Siemens / Mentor recommended methods, which include the use of fewer UVM macros and more UVM 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 alternative methods, so they understand why there are differences among the methods presented.
5) All the UVM material authors 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 to teach these concepts to beginners.
6) Many of those who have written UVM 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 UVM 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/UVM materials. Engineers often wonder why TLM was introduced in these texts.
8) Most UVM materials show the driver on the right and the monitor on the left (right-to-left data flow inside the agent). This contradicts established good block diagramming methods (data should flow from left to right) and adds unnecessary confusion to the learning process for those 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 online examples 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 UVM. Too many engineers try to learn UVM without a full understanding of these SystemVerilog fundamentals (this is not the fault of UVM authors).
11) Classes are applied as stimulus and sampled for verification. Existing materials do not explain why classes are preferred over structs.
12) Interfaces, virtual interfaces, and their recommended 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.