World Class SystemVerilog & UVM Training

Sunburst Design - Advanced UVM Training

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

NOTE from Cliff Cummings:

If you are new to UVM or have less than 2 years of UVM experience with no formal UVM training, you should take UVM training and ***you should NOT take this Advanced UVM training course.***

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

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

When it comes to UVM, engineers tend to believe that they can pick-up UVM basic skills without taking formal UVM training. This is wrong and engineers that skip formal UVM training will likely experience confusion and frustration in this Advanced UVM training course.

I had one customer with engineers who had two years of UVM project experience. They asked for a 3-day Advanced UVM training course. I sent them the UVM training syllabus showing topics that I planned to eliminate from the UVM course and included a list of advanced topics for their consideration. I wanted to make sure that I had deleted the appropriate beginner topics from the scheduled 3-day training. The engineers came back and asked me to teach all four days! Even though the engineers had existing UVM experience on multiple projects, they all agreed that the full four days had helped them to understand how and why SystemVerilog and UVM worked the way they did in their testbenches.

To take Advanced UVM training, engineers should either first take the UVM training course, or should have two years of UVM Verification Experience.

UVM topics need to be taught multiple times to understand how they work. That course includes some of the fundamental UVM topics that I find most UVM-trained engineers do not understand or have missed. This course will include multiple full UVM testbenches including some very advanced UVM testbench techniques.

Engineers with little UVM experience or training will likely be lost and frustrated for most of the time in this training course. This is not an Advanced Beginner course!

NOTE: Sunburst Design merged with expert services provider, Paradigm Works. Some of the advanced topic content will be provided by Cliff's new colleagues at Paradigm Works (abbreviated PW).

Proposed Advanced UVM syllabus below. Advanced services companies have great large-project experience that should be considered some of the current Best Known Methods (BKMs).

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

(1) Advanced Classes Usage & Polymorphism (~11:00 – 11:45 am)

Section Objective: Learn advanced class topics - Upcasting & downcasting are used frequently in UVM. Declaring & using extern and static methods.
It is assumed that engineers already understand basic classes, class extension, class methods and Object Oriented Programming (OOP).

- Assigning class handles
- Assigning extended handles to base handles
- Casting base handles to extended handles (technique used by UVM)
- Upcasting & downcasting - used frequently in UVM
- Extern methods
- Static methods
- local & protected keywords

(2) Pure Virtual & Virtual Interfaces (~11:45 am - 12:30 pm)

Section Objective: Learn pure virtual method usage (used by subscribers) and virtual interface techniques.
It is assumed that engineers already understand virtual classes and methods.

- Pure virtual methods (SystemVerilog-2009 update - used by UVM)
- Interfaces and virtual interfaces for UVM testbench development
- Passing type parameters - class specializations

(3) Starting & Stopping UVM Tests (~12:30 - 1:00 pm)

Section Objective: Learn techniques to start and stop tests. Emphasis on the efficiency of using raised & dropped objections. This section introduces the uvmtb_template files that are used in the class labs for rapid UVM testbench development.

- Running tests using +UVM_TESTNAME command line switch
- Stopping tests using raised & dropped objections - emphasis on simulation efficiency.
- Objections - debugging trick

PW content - Coordinating Shutdown

  - UVM phases and UVM objections
  - UVM Heartbeat
  - UVM global watchdog
  - Using UVM runtime phases (or not) - pros/cons.

My take is that if I do not use the runtime phases I need to implement my own esoteric style.
PW Approach - *UVM Heartbeat - we configure the UVM Heartbeat with the UVM Objections to keep alive based on activity. I am working on a project now that does not have this feature and we have to deal with the runaway tests.*

- LAB - UVM Common Errors
- LAB - UVM First Testbench - Testing a Counter *(Full UVM self-checking testbench)*

(LUNCH & LAB: 1:00 – 2:00 pm)

(4) Proper Use of UVM print/display commands (~2:00 - 3:00 pm)

Section Objective: Proper printing techniques in UVM. This is a widely misunderstood concept and poorly documented.
It is assumed that engineers already understand SystemVerilog dynamic & associative arrays. Engineers should also already be familiar with UVM base classes including basic transactions, sequences and components.

- Reporting methods & arguments
- Reporting macros and why they are preferred
- UVM_LOW verbosity abuse - Performance penalty
- UVM_VERBOSITY explained
- Why the UVM User Guide, Reference Manual and Books get VERBOSITY wrong!
- Catch & throw message techniques.

(5) UVM Transaction Base Classes, Sequences & Tests (~3:00 - 5:00 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 performance benchmarks of each method (this is detailed in a 72-page SNUG-2014 paper on the Sunburst Design web page because most UVM engineers do not understand this topic).*** This section then shows techniques interesting techniques to run sequences and tests.
It is assumed that engineers already understand SystemVerilog constrained random testing and functional coverage.

- 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 - performance issues
- Randomizable data members
- Randomizable knobs
- Randomization constraints
- uvm_object constructors
- UVM sequence body task
- start_item(tx) - finish_item(tx)
- `uvm_do macros - performance issues
- Performance - driver creating a new object every time
- randomize() the transaction
- randomize() the transaction with inline constraints
- UVM sequences of uvm_sequence_item and uvm_sequence
- randomize ... with / rand_mode()
- Running UVM tests with inline constraints

Day Two

(6) Transaction Level Model (TLM) Basics & Analysis Port Details (~11:00 am - 12:00 pm)

Section Objective: Understanding how TLM ports, exports and imps work. A detailed discussion of analysis (broadcast) ports and how it works. Implementing more than one uvm_analysis_imp_port on a component requires special attention, which requires the use of special macros. Examination of performance issues.
It is assumed that engineers already understand construction of analysis paths. This section will give the details and background.

- TLM ports & exports
- Why "ports" and "exports"
- TLM put, get and transport configurations
- Get configurations
- Put configurations ***(these slides are included but typically skipped)***
- Transport configurations ***(these slides are included but typically skipped)***
- Analysis ports compared to satellite broadcasts
- Importance of copy command with broadcast analysis transactions
- How analysis paths work
- Legal and illegal analysis paths
- Performance - detecting analysis port connections
- uvm_subscriber with analysis export
- Connecting a coverage collector using an analysis export
- Multiple uvm_analysis_imp ports on a component
- `uvm_analysis_imp_decl macros
- LAB - UVM Scoreboard Style #2 - 2 Analysis Imp Ports - *(Full UVM testbench lab)*

(7) uvm_config_db and Configuration objects (~12:00 - 12:30 pm)

Section Objective: Use and abuse of the uvm_config_db. uvm_config_db wildcard performance issues. Defining and using configuration objects.
It is assumed that engineers already understand the basic UVM testbench structure including tests, environments, agents, sequencers, drivers, monitors and coverage collectors.

- Storage and retrieval techniques for the DUT interface handle
- uvm_config_db#(type) set/get (new/easier method to store the DUT interface handle)
- Performance and debug issues related to the uvm_config_db
- Defining configuration objects.
- Using configuration objects.
- Active and passive agents
- Virtual interfaces for verification
- LAB - UVM Testbench with configuration objects *(Full UVM self-checking testbench)*
- LAB – FIFO Gray Code Pointer - *(Full UVM self-checking testbench)*

(8a) SystemVerilog Mailboxes & TLM FIFOs (~12:30 - 1:00 pm)

Section Objective: SystemVerilog mailboxes and usage compared to queues. Two types of uvm_tlm_fifos and how they are used, especially in scoreboards.
It is assumed that engineers already understand SystemVerilog queues.

- SystemVerilog mailboxes
- uvm_tlm_fifo
- uvm_tlm_analysis_fifo

(LUNCH & LAB: 1:00 – 2:00 pm)

(8b) UVM Scoreboards (~2:00 - 2:45 pm)

Section Objective: Learn two techniques for creating self-checking scoreboards. 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.
It is assumed that engineers already understand the basic UVM testbench structure including tests, environments, agents, sequencers, drivers, monitors and coverage collectors.

- SystemVerilog queues
- 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)*
- LAB – UVM Scoreboard Style #1 - Pipeline Design - *(Full UVM testbench lab)*

(9) UVM Virtual Sequence Generation (~2:45 - 3:45 pm)

Section Objective: Learn advanced sequence generation techniques using virtual sequences. Understanding the role of the m_sequencer and p_sequencer handles in UVM.
It is assumed that engineers already understand the basic UVM testbench structure including tests, environments, agents, sequencers, drivers, monitors and coverage collectors.

- 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

(10) Clocking Blocks and Verification Timing (~3:45 - 4:45 pm)

Section Objective: Learn important stimulus and verification timing issues and techniques - SystemVerilog clocking blocks help control timing for UVM verification environments.
This section briefly shows programs -vs- modules and why programs should not be used. If you have a module-based testbench with race conditions that are fixed by programs, you are probably using the wrong timing to drive stimulus. This is explained with event scheduling.
It is assumed that engineers already understand interfaces and virtual interfaces.

- SystemVerilog event scheduling - modules -vs- programs
- Why programs should be avoided
- 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

(11) UVM Factory, Constructors & Factory Overrides (~4:45 - 5:30 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.
It is assumed that engineers already understand factory basics - this section will give details.

- 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

Day Three

Advanced Topics (These topics are constantly updated so scheduling is not firm - corresponding advanced labs are also under development)

(12) Reactive Tests/Drivers and Sequence Layering (~11:00 am - 12:00 pm)

Section Objective: Much testing is done using tests that drive constrained random tests. There are times when it is desirable to have the test react to feedback from the DUT after driving sequences. Assembling layered sequences and using them for multi-interface designs are described. These topics are described in this section.
It is assumed that engineers fully understand standard sequence/sequencer/driver stimulus generation.

- REQ and RSP transactions and how they are used
- Handshaking between sequences, sequencers and drivers
- Modifications to the driving tasks to enable bidirectional communication with the test
- Sequence layering techniques
- Using virtual sequences with layered sequences
- FIFO example that drives stimulus until full or reads stimulus until empty
- LAB – Reactive Stimulus FIFO testbench - *(Full UVM testbench lab)*

(13a) Advanced Topic - Large Project Parameterized Classes - *New Material*

- (Full disclosure - techniques from DVCon by Verilab engineers) (~12:00 - 1:00 pm)
It is assumed that engineers fully understand standard interface/DUT connections in the top module.

- Parameterized Classes
- Parameterized Interfaces
- Parameterized Registers
- Traditional DUT to Testbench Connection
- UVM Harness
- Traditional interface placement
- SystemVerilog port coercion - inputs
- Extracting RTL Parameters
- UVM Harness interface placement
- Tuning testbenches with RTL parameters
- Parameter Selection Optimization

Additional Advanced Topics - Large Project Topics

- (Full disclosure - Some Advanced UVM testbench techniques to be shared with Cliff from Paradigm Works engineers - time permitting)

PW content - Testbench Architecture

Basic testbench architecture - how to create vertically/horizontally reusable testbench.

- 1-A) Low level - top level TB, interface styles (use clocking blocks), interface connections to testbench and RTL (use RTL wrapper).
- 1-B) UVCs
  - Configuration classes and connections - subject described in DVCon 2020 paper.
  - Emulator ready UVCs Cook Book.
- 1-C) Predictors/scoreboards/checkers.

- 2) Configuration management & Vertical Integration.

Approaches used to coordinate and randomize all UVC configurations
Mentor Methodology (UVF) - *Uses a configuration approach that is base test centric. All the configuration & UVCs are setup/randomized/distributed in the base test.*
PW Approach - *Encapsulates configuration & UVCs within the ENV. This approach allows for easier vertical integration.*

(13b) Multi-block Environments Using UVM & Parameterized UVM Components & Tests (~2:00 - 3:00 pm)

Section Objective: Learn how to take separate block-level UVM testbenches and merge them into a larger UVM environment. Use of active and passive agents are described. Use of virtual sequences are also described for multi-interface environments. Also learn how to use parameterized UVM transactions and components with different parameters.
It is assumed that engineers already understand basic UVM block-level testbench structure.

- Active & passive agents
- Using one block as a new stimulus source for another block
- Using layered sequences and virtual sequences in a multi-block environment
- `uvm_object_param_utils
- `uvm_component_param_utils
- Class handle-based factory access & string-based factory access
- How to pass param-version tests from the command line
- Simple parameterized models with parameterized transactions & components
- UVM harness testbench techniques
- UVM parameterized harness techniques
- LAB - UVM harness FIFO lab *(Full UVM testbench lab)*
- LAB - UVM harness parameterized FIFO lab *(Full UVM testbench lab)*

(14) Advanced Topic - Dynamic Resetting

- (Full disclosure - Advanced UVM testbench techniques from Paradigm Works engineers) (~3:00 - 3:00 pm)

PW content - Dynamic Resetting
Multiple resets is a huge challenge with UVM. The challenges include outstanding sequence transactions (transactions that are waiting for item_done()), sequence killing (maybe SV fine gran threads), multiple reset domains, and perhaps the dreaded/contentious UVM phase jumping.

Some companies like Intel uses their own register package. Paradigm Works engineers (U.S. based expert verification contractors) report that using register package is often problematic.
The register package is the last section in case engineers do not want to stay for this section or want to work labs instead.

NOTE from Paradigm Works Engineers
PW content - UVM Register Model
*UVM Reg is probably the biggest time sink. Specifically, modeling DUT driven status is challenging. Keeping the UVM REG concurrent with the RTL is a challenge and does not work for up to 80% or more cases using the default UVM REG. At **<current customer>** I attempted to keep track of my status using UVM REG and thought all was okay but then realized it was broken.*

***PW content -** Additionally, I have used UVM REG callbacks to update the predictor code. This technique is somewhat valuable. It allows the predictor to reach into register model for updates.*
Finally, handling multiple reset domains is another topic worth discussing especially modeling multiple resets and reset domains (see this topic included in the syllabus).

(15) UVM Register Abstraction Layer (~3:30 - 5:30 pm)

Section Objective: Introduction to the UVM Register Package. Shows the register definition classes, which is the easy part of the register package. The hard part is understanding how the map, sequencer, adapter and predictor are correctly setup. The hard part is shown in detail.

- UVM register package resources
- Motivation for the register package
- Front door / back door DUT access
- Register definition hierarchy & requirements
- Register definition examples
- Register block requirements
- Register adapter requirements and implementation
- uvm_reg_bus_op usage
- Register adapter: reg2bus & bus2reg methods
- Register access commands
- Register sequences & Built-in register sequences
- Environment with register model, adapter and predictor
- Environment build_phase() requirements
- Transaction register sequences
- Register tests
- Try to add discussion of 20-30K registers and how this impacts factory performance (or objects in general)
- ***To include PW advanced content as described above - time permitting***
- LAB – UVM 4-register design
- LAB – UVM Register Package 4-register design

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.