UVM Fundamentals

Course Description

This 5-days course designed for ASIC & FPGA verification engineers that would like to use the SystemVerilog language and UVM methodology to verify deeply digital designs.

SystemVerilog is a significant new enhancement to Verilog and includes major extensions into abstract design, testbench, formal, and C-based APIs.

SystemVerilog also defines new layers in the Verilog simulation strata. These extensions provide significant new capabilities to the verification engineer, such as Object-Oriented Programming (OOP), randomization, assertions, packages, queues, dynamic & associative arrays, interfaces and functional coverage.
These new features allow better teamwork and co-ordination between different project members.

Universal Verification Methodology (UVM) is a standardized methodology for verifying digital designs and SoC. It is built on top of SV language and consists of set of standards, tools, and APIs for design verification.

UVM helps companies develop modular, reusable, and scalable test benches that can be deployed across multiple projects.

The first day introduces the essential SystemVerilog concepts & constructs that are critical for the UVM part and it includes hands-on labs to practice these concepts.

The first part covers OOP, functional code coverage, assertion and randomization.

The next 4-days introduce the UVM and its structure, then covers the UVM library, reporting mechanism, factory, TLM, configuration database, phases, hierarchy, test and testbench top.

Then the training covers how to generate stimulus with sequences and virtual sequences as well as RAL.

Extensive practical labs are integrated during the training to make sure that the participant understand the flow, structure and concept of each verification building block.

Course Duration

5 days

Goals

1. Use Object Oriented Programming
2. Apply SystemVerilog Assertions
3. Apply constrained random stimulus
4. Perform functional coverage
5. Become familiar with UVM structure
6. Use reporting macros, inline code command line to manage message verbosity
7. Become familiar with UVM library basics
8. Create objects. Components and use the factory and factory overrides
9. Analyze and debug the design with TLM elements and scoreboards
10. Use the configuration database macros
11. Build hierarchical testbenches and configure various components
12. Create top level testbench and connect it to virtual or physical interfaces
13. Create driver, sequencer and connect them
14. Create collector and monitor
15. Generate stimulus via virtual or physical sequences and run them
16. Implement RAL and access it through frontdoor and backdoor

Intended Users

Hardware/Software engineers who would like to verify ASIC/FPGA designs with SystemVerilog and the UVM

Prerequisites

1. Verilog language
2. SystemVerilog language
3. Verification guidelines
4. Experience with simulator

Course Material

1. Course book
2. Lab handbook (Phyton notebooks)
3. Virtual Machine with all necessary tools
4. Trainer solutions to all labs

Table of Contents

Day #1

- **SystemVerilog Essentials for UVM**
  - SystemVerilog threads
  - OOP in details
  - Randomization
  - **Lab #1: Classes and Randomization**
    - Define your own class
    - Create an instance of a class
    - Use virtual methods
    - Specify the protection level of a class fields (protected, local properties)
    - Declare class fields as rand and randomize their values
    - Use the static qualifier for class fields and methods
    - Add simulation logic and execute the simulation
  - Assertions
  - **Lab #2: Concurrent Assertions**
    - Create a concurrent assertion in SVA language
    - Create the assertion module, then bind it to the DUT
    - Access internal signals of the DUT within assertion module
    - Execute the simulation
  - Functional Coverage
  - **Lab #3: Functional Coverage**
    - Create functional coverage models using covergroup and coverpoint
    - Specify the values that must be covered using bins
    - Analyze results using your specific simulator tools (for example Cadence IMC)

Day #2

❖ Introduction to UVM

- Where UVM came from?
- What is UVM?
- The whys of UVM
- What does UCM include?

❖ UVM Testbench Basics

- What is UVM testbench?
- UVM testbench architecture
  - UVM testbench
  - UVM test
  - UVM environment
  - UVM sequence
  - UVM agent
  - UVM sequencer
  - UVM driver
  - UVM monitor
  - UVM scoreboard
  - UVM functional coverage

❖ UVM Library Basics

- UVM class library
- Classes & utilities categories
- The uvm_object class
- The uvm_pkg
- UVM core base classes derived from uvm_object class and their role

❖ UVM Reporting

- Testbench reporting
- UVM reporting structure
- Severity, verbosity and simulation handling behavior
- UVM reporting classes
- UVM messaging system
- UVM message types and APIs
- UVM message verbosity in details

- UVM message ID field
- Setting component specific verbosity threshold
- UVM reporting APIs structure
- UVM reporting guidelines
- **Lab #1: UVM Messaging System**
  - Use uvm_info/warning/error/fatal macros for presenting messages
  - Manage the verbosity both from the macro call and from the command line
  - Control the simulator's behavior in response to a called macro using the UVM_ACTIONS mechanism

❖ UVM Object

- What is UVM object?
- Class hierarchy and definition
- Create and get_type_name methods
- Utility macros for factory registration
- Creation of class object
- UVM field macros
- Field automation flags
- Radix control
- UVM object print
- Using the do_print method
- What is uvm_printer?
- Printer types and methods
- Using uvm_printer
- Using the sprint method
- Using convert2string method
- UVM copy and do_copy methods
- UVM clone and do_clone methods
- UVM compare and do_compare methods
- What is uvm_comparer?

❖ UVM Factory

- What is UVM factory?
- Factory registration
- Coding convention
- What is factory override?
- Factory override methods

- Type and instance override examples
- **Lab #2: Objects, Components and Factory Facility**
  - Create your own class inheriting from the uvm_object or uvm_component classes
  - Implement functions such as do_copy, do_compare, etc.
  - Use field macros
  - Dynamically replace instances of selected classes using the factory override mechanism

Day #3

❖ UVM Transaction Level Modeling (TLM)

- Why TLM?
- What is a transaction?
- UVM transfer methods
- TLM put port
- Blocking and non-blocking put port
- Sending transaction to higher hierarchy level
- TLM get port
- Blocking and non-blocking get port
- Blocking vs non-blocking considerations
- TLM FIFO
- TLM FIFO in multiple levels
- UVM TLM analysis port
- The write() method
- What is UVM subscriber?
- TLM connectivity debugging
- Connecting multiple ports to a single component
- **Lab #3: TLM**
  - Add a port of type uvm_analysis_* to a component
  - Connects the ports together
  - Use UVM functions to locate the cause of errors related to components TLM connections
  - Extend the component to include more than one port of type uvm_analysis_imp

❖ UVM Configuration Database

- What is the resource database?
- The UVM config_db
- Resource fields
- How the global databease behaves?
- Storing and retrieving methods introduction
- Usage rules
- Using set() method
- Putting virtual interfaces into database example
- Configuring agents inside an environment example
- Using get() method
- Calling get and set methods from different hierarchies
- Recommended practice
- Quiz: set vs get
- Database access debug

❖ UVM Phases

- What UVM phases are?
- UVM phases advantage
- UVM phases structure
- How to start a UVM phase execution?
- Build phases in details
- Run phases in details
- Cleanup phases in details
- **Lab #4: Configuration and Resources**
  - Store and retrieve configuration objects from the uvm_config_db
  - Understand the importance of super.build_phase() for components with/without fields covered by field macros

Day #4

❖ UVM Hierarchy

- UVM environment in details
- UVM driver in details
- UVM sequencer in details (including virtual sequencer)
- UVM monitor in details
- UVM agent in details

❖ Using Packages

- Packages review
- UVM package coding guidelines
- Package organization
- Package scope
- **Lab #5: Hierarchy**
  - Build testbench hierarchy using the factory
  - Configure components of type uvm_agent using the uvm_config_db mechanism
  - Organize the verification environment files by grouping them into packages
  - Manage the passing of configuration objects between components

❖ Testbench Top

- APB DPRAM testbench block diagram example
- What is the testbench top module?
- Testbench top code example
- Interface review
- Simple clock generation
- Complex clock generation (multiple parameters)

❖ UVM Test

- What is UVM test?
- How to write a test?
- Test in details
- How to run a UVM test?
- Derivative tests
- Apply different configuration

- **Lab #6: Connecting the DUT**
  - Instantiate a physical interface
  - Connect the physical interface with the DUT
  - Pass the virtual interface to the environment
  - Avoid parameterization hell by using the maximum footprint concept
  - Apply assertions and coverage

Day #5

❖ Stimulus Generation

- What is a sequence?
- What is UVM sequence?
- Sequence operations
- How to create a UVM sequence?
- UVM sequence items
- Sequence items randomization
- Connect sequence to sequencer
- Using `uvm_do_* sequence macros
- Macros interaction
- What is UVM virtual sequencer?
- Virtual sequence code example
- **Lab #7: Stimulus Generation**
  - Create a class representing the data model
  - Develop a sequence API
  - Run a sequence and manage the end of test using objection mechanism
  - Record the transactions

❖ UVM RAL

- Problem statement
- What is the register layer?
- RAL components
- Register model in details
- Fields and registers
- Memory

- Address map
- Register block
- Access policies
- Mirrored and desired values
- Fron and backdoor accesses
- Hierarchical HDL paths
- Backdoor access methods
- Backdoor configuration code example
- The complete picture
- Adapter in details
- Predictor
- F-coverage model and types
- F-coverage code example
- Creating the register block
- Connecting the register block
- Access methods
- **Lab #8: Register Abstraction Layer**
  - Build a simple register model
  - Configure backdoor access in the environment
  - Create a register sequence using the frontdoor access
  - Check register values using the backdoor access
  - Analyze the functional coverage results of a register