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 training introduces 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 using 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. Become familiar with UVM structure
2. Use reporting macros, inline code command line to manage message verbosity
3. Become familiar with UVM library basics
4. Create objects, components and use the factory and factory overrides
5. Analyze and debug the design with TLM elements and scoreboards
6. Use the configuration database macros
7. Build hierarchical testbenches and configure various components
8. Create top level testbench and connect it to virtual or physical interfaces
9. Create driver, sequencer and connect them
10. Create collector and monitor
11. Generate stimulus via virtual or physical sequences and run them
12. 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 for all labs

Table of Contents

Day #1

- Basic reporting macros
- Changing the verbosity level
- Setting component verbosity
- Setting severity actions
- Setting ID actions
- Reports from sequences
- Max quit count

- **Lab #1: UVM Messaging System**
  - Use uvm_info/warning/error/fatal macros for presenting messages
  - Manage verbosity both from the macro call and the command line
  - Control the simulator's behavior using the `UVM_ACTIONS mechanism

Day #2

❖ 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
- UVM Field macro flags
- Radix control
- Using the do_print() method
- The uvm_printer class
- Printer types and methods
- Using the uvm_printer class
- 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
- UVM uvm_comparer class

❖ UVM Factory Facility

- What is UVM factory?
- Factory registration
- Coding convention
- What is factory override?
- Factory override methods
- Type and instance override examples
- UVM uvm_component_param_utils

- **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 use 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 analysis port
- TLM ports, exports and imps
- The write() method
- The uvm_subscriber class

- Debugging TLM connectivity
- Multiple analysis ports

- **Lab #3: TLM**
  - Add a port of type uvm_analysis_* to a component
  - Connects the ports together
  - Use UVM functions to debug component TLM connectivity
  - Extend the component to include multiple uvm_analysis_imp ports

❖ UVM Configuration Database

- Overview of the configuration database
- Database resources
- Storing and retrieving methods
- Wildcard paths
- Using set() method
- Using get() method
- Configuring the component hierarchy
- Agent configuration
- Sequence configuration
- Debugging configuration settings

- **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 Phases

- What are UVM phases?
- Three groups of phases
- Breakdown of UVM phases
- Starting UVM phase execution

- Build phases in detail
- Run phases in detail
- Cleanup phases in detail

❖ UVM Hierarchy

- UVM environment class in detail
- UVM driver class in detail
- UVM sequencer class in detail (including virtual sequencer)
- UVM monitor class in detail
- UVM agent class in detail
- SystemVerilog Packages overview
- 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
  - Organize the verification environment files by grouping them into packages
  - Manage configuration object passing between components
  - Avoid parameterization hell by using the maximum footprint concept
  - Apply assertions and coverage

❖ 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

Day #5

❖ Stimulus Generation

- What is a sequence?
- Sequence operations
- How to create a UVM sequence?
- UVM sequence items
- Sequence items randomization
- Connecting a sequence to a sequencer
- Using `uvm_do_* sequence macros
- Macros interaction

- **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

❖ Virtual Sequences & Sequencers

- Problem statement: sequences in a multi-interface environment
- Physical vs. virtual sequences
- Virtual sequence overview
- Virtual sequence base class
- Virtual sequencer

- Virtual sequencer modes
- Locking or grabbing a sequencer
- Connecting a virtual sequencer to subsequencers
- m_sequencer and p_sequencer handles
- The uvm_declare_p_sequencer macro
- Starting virtual sequences

- **Lab #8: Stimulus Generation**
  - Create a virtual sequence
  - Create a virtual sequencer
  - Build a sequence hierarchy
  - Use the sequencer locking mechanism

❖ 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 #9: 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