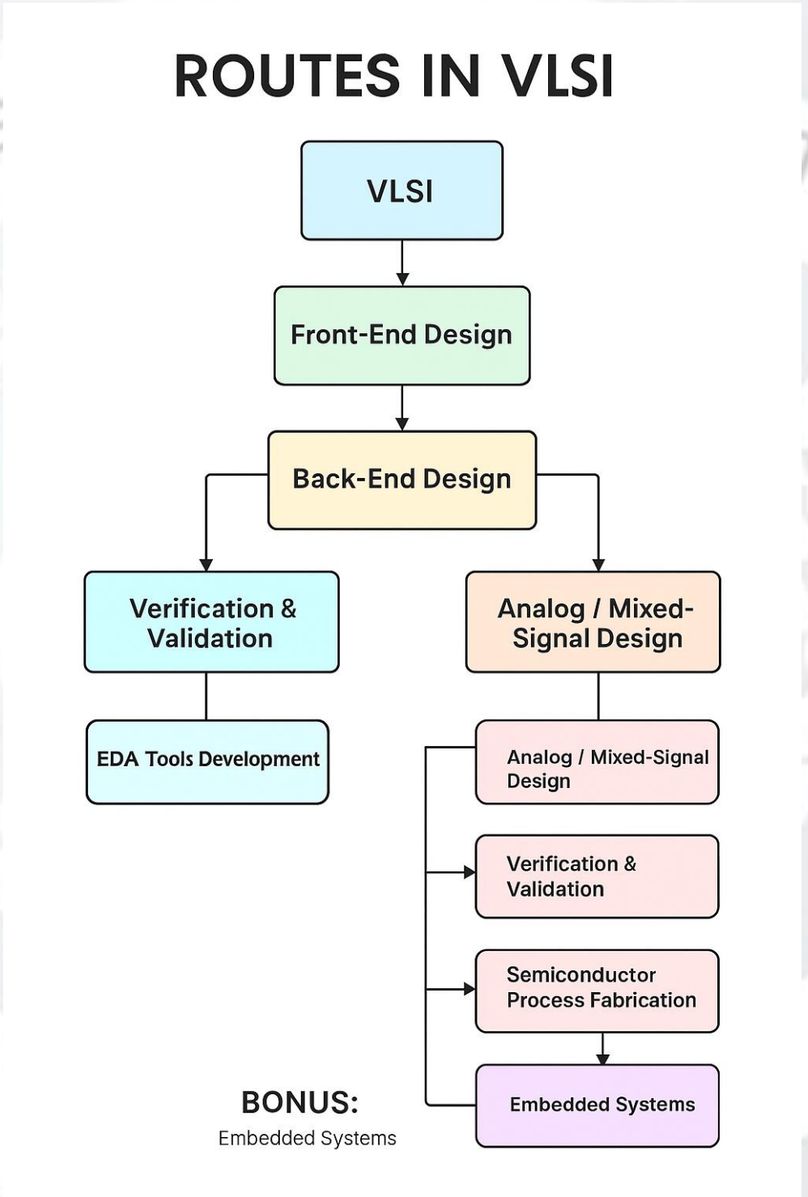
# SkSemAutom

OUTCOME BASED CURRICULUM

ML-Driven Verification for Next-Gen SoCs

## 🚀 Exploring the Many Routes in VLSI!

VLSI isn’t just about chip design — it’s a universe of opportunities. From Front-End Design to Physical Design, Verification, EDA Tool Development, and even Semiconductor Fabrication, every route plays a key role in bringing technology to life. This roadmap helps students and engineers discover where their interests truly lie — whether it’s logic, layout, coding, or circuits.



Master the Art of Semiconductor ML-Verification with Industry-Relevant Skills

Course Overview: This outcome-based course is designed to equip learners with essential ML-Driven SoC verification techniques, focusing on SystemVerilog, Universal Verification Methodology (UVM), and automation scripting (TCL). The program blends theoretical concepts with practical hands-on assignments, real-world design projects, and advanced debugging techniques, preparing learners for industry-ready roles in semiconductor technology.

Core Technical Skills:

- Verilog Proficiency: Deep dive into Verilog, including RTL design, procedural blocks, and data types.
- SystemVerilog Proficiency: Deep dive into SystemVerilog, including RTL design, verification constructs, procedural blocks, and data types.
- Universal Verification Methodology (UVM): Industry-standard methodology to improve the efficiency and reusability of verification environments by Integrating with Machine Learning Algorithms.
- Verification Techniques: Covers Black Box, White Box, and Gray Box verification approaches.
- DFV (Design for Verification): Optimized SoC design strategies that streamline verification.
- Hardware Simulation Tools: Exposure to Verilator for SoC simulation.
- Automation with TCL: TCL scripting to automate verification processes, reducing repetitive workload.
- Complete RISC-V Design and Verification using SV and Integration with ML
- Python-UVM-RTL: Deep dive into python-UVM based Verification leads to ML Integration
- UVM-ML: The UVM-ML framework that enables multi-language verification by allowing components written in different verification languages—like SystemVerilog (UVM-SV)—to work together in a unified testbench environment.

Practical Implementation & Assignments:

- Industry-based Verification Projects: Includes tasks such as Decade Counter Design, Scrambler/Descrambler Verification, FIFO design, and FSM Optimization.
- Testbench Development: Building efficient RTL test environments using SystemVerilog and verification tools.
- UVM-Based Testbenches: Developing self-checking and object-oriented verification models.
- Simulation & Debugging Techniques: Includes coverage analysis, lint tools, and debugging strategies.

**Additional Benefits & Opportunities:**

- Hands-on LMS Access: Full access to learning materials, recorded classes, and assignments via SkSemAutom LMS.
- Internship Opportunity: Students can gain practical experience alongside learning.
- Placement Assistance: Guidance provided for securing SoC Verification Engineer roles.
- EDA Tool Access: Learners get permanent access to necessary design tools for hands-on practice.
- Interview Preparation: Includes SoC verification interview questions for job readiness.

Course Structure & Duration:

- Format: 6-months online learning with daily live sessions.
- Total Learning Hours: 320 hours of intensive training. For ₹12,000 only! and early bird fees ₹10,000
- Complete Axis will be given to LMS (www.sksemautom.com) for Course Material, Recorded Classes, and Assignments
- Eligibility: M.Tech/ME/B.Tech/BE/MSc/BSc (Completed/Pursuing) (ECE/EEE/Electronics/Computer Science)

| Section | Section Name | Section Description | Lectures (Each Lecture is more than 30 mints) | Learning Objectives (After Completion of each lecture the Learner will be able to work on):- |
|---|---|---|---|---|
| Section-I | SoC Design Verification | ➢ System-on-Chip (SoC) Design Verification is a crucial process in ensuring the functionality, performance, and reliability of complex integrated circuits before they are manufactured. It involves testing and validating the design to catch potential bugs and ensure compliance with specifications. | ➢ Soc Design Verification<br>➢ Assignment | ➢ Importance of Verification<br>➢ Verification Plan and Strategies<br>➢ Verification Plan<br>➢ Functional Verification |

| Section | Section Name | Section Description | Lectures (Each Lecture is more than 30 mints) | Learning Objectives (After Completion of each lecture the Learner will be able to work on):- |
|---|---|---|---|---|
| Section-II | Verification Methods | ➢ Verification methods in SoC design ensure that the design functions a intended before fabrication. Here ar the main verification approaches:<br>➢ Simulation-Based Verification Uses testbenches to simulate the design behavior under various conditions. Popular methodologies include:<br>➢ Register Transfer Level (RTL Simulation: Verifies logic design.<br>➢ Gate-Level Simulation: Checks timing accuracy after synthesis.<br>➢ **Formal Verification:** Mathematica methods prove design correctness without exhaustive simulations.<br>➢ *Model Checking:* Analyzes state transitions.<br>➢ *Equivalence Checking:* Confirm that RTL code matches the synthesized netlist. | ➢ Verification Methods<br>➢ Assignment | ➢ Black Box Verification<br>➢ White Box Verification<br>➢ Gray Box Verification |

| Section | Section Name | Section Description | Lectures (Each Lecture is more than 30 mints) | Learning Objectives (After Completion of each lecture the Learner will be able to work on):- |
|---|---|---|---|---|
| Section-III | Design for Verification | ➢ **Design for Verification (DFV) in SoC Technology** refers to designing integrated circuits with verification efficiency in mind. Since verification consumes a significan portion of the development cycle DFV techniques help streamline this process, making it easier to identify and fix bugs. | ➢ Design for Verification<br>➢ Assignment | ➢ Introduction<br>➢ RTL Test bench internal modules to simulate use case scenario of VLSI SoCs<br>➢ Automated test environment<br>➢ Design and Verification Assertions |

| Section | Section Name | Section Description | Lectures (Each Lecture is more than 30 mints) | Learning Objectives (After Completion of each lecture the Learner will be able to work on):- |
|---|---|---|---|---|
| Section-IV | Verification | ➢ This section Describes how to write Verilog Code for verification with Industry based Examples. It also covers different verification Tools and brief explanation of Verification Language. | ➢ Verification Examples<br>➢ Verification Tools<br>➢ Verification Language Introduction<br>➢ Assignment | ➢ Decade counter Design and Verification<br>➢ self-synchronizing scrambler Design and Verification<br>➢ Descrambler Design and Verification<br>➢ Simulators.<br>➢ Coverage tools.<br>➢ Lint tools.<br>➢ Verification Language brief explanation |

| | | | | |
|---|---|---|---|---|
| Section-V | Verification Language Continued | ➢ **SystemVerilog (SV)** is a powerful hardware description and verification language (HDVL) used extensively in digital design and SoC verification. It extends **Verilog** with advanced features that enhance design modeling, simulation, and verification. | ➢ Introduction<br>➢ Assignment | ➢ ASIC Design Flow<br>➢ ASIC Verification .<br>➢ Strategies for SOC Verification<br>➢ Verilog Constructs<br>➢ Concurrent Assignments<br>➢ Procedural Block<br>➢ Introduction to SystemVerilog<br>➢ SystemVerilog for Hardware Description and Verification<br>➢ Summary and Future Discussions |

| | | | | |
|---|---|---|---|---|
| | | | ➢ SystemVerilog Literal Valuesand Data Types | ➢ Predefined Gates<br>➢ Structural Modelling<br>➢ SystemVerilog Format Specifier<br>➢ Multi-bit Constants and Concatenation<br>➢ Literals<br>➢ Data Types |
| | | | | ➢ Summary |
| | | | ➢ Hardware Description Using SystemVerilog<br>➢ Assignment | ➢ Introduction<br>➢ The Net Data Type<br>➢ Combinational Elements<br>➢ always_comb to Implement the Code Converters<br>➢ Understanding of Concurrency<br>➢ Procedural Block always_latch<br>➢ Procedural Block always_ff<br>➢ Use the always_ff to Implement the Sequential Design<br>➢ Instantiation Using Named Port Connections<br>➢ Instantiation Using Mixed Port Connectivity<br>➢ Summery |
| | | | ➢ SystemVerilog and OOPS Support<br>➢ Assignment | ➢ Enumerated Data Types<br>➢ Structures<br>➢ Unions<br>➢ Arrays<br>➢ Summary |
| | | | ➢ Important SystemVerilog Enhancements<br>➢ Assignment | ➢ Verilog Procedural Block<br>➢ SystemVerilog Procedural Blocks<br>➢ Block Label<br>➢ Statement Label<br>➢ Module Label<br>➢ Task and Function Enhancements<br>➢ Void Function<br>➢ Loops<br>➢ Guidelines<br>➢ Summary |
| | | | ➢ Combinational Design Using SystemVerilog<br>➢ Assignment | ➢ Role of always_comb Procedural Block<br>➢ Nested if-else and Priority Logic<br>➢ Parameter and Its Use in Design<br>➢ Conditional Operator and Use to Infer the Mux Logic<br>➢ Decoders<br>➢ Priority Encoder<br>➢ Summery |

| | | | | |
|---|---|---|---|---|
| | | | ➢ Sequential Design Using SystemVerilog<br>➢ Assignment | ➢ Intentional Latches Using always_latch<br>➢ PIPO Register Using always_ff<br>➢ Using Asynchronous Reset<br>➢ Using Synchronous Reset<br>➢ Up-Down Counter<br>➢ Shift Register<br>➢ Ring Counter<br>➢ Johnson Counter<br>➢ Implement RTL for Clocked Arithmetic Unit<br>➢ Implement RTL for Clocked Logic Unit<br>➢ Summery |

| Section | Section Name | Section Description | Lectures (Each Lecture is more than 30 mints) | Learning Objectives (After Completion of each lecture the Learner will be able to work on):- |
|---|---|---|---|---|
| Section-VI | RISC-V | ➢ RISC-V (pronounced "risk-five") is an open-source instruction set architecture (ISA) based on the principles of Reduced Instruction Set Computing (RISC). It was developed at the University of California, Berkeley, and is now maintained by RISC-V International. | ➢ The RISC-V Architecture and Instruction Set<br>➢ Assignment | ➢ Introduction to RISC-V<br>➢ Technical requirements<br>➢ The RISC-V architecture and applications<br>➢ The RISC-V base instruction set<br>➢ RISC-V extensions<br>➢ RISC-V variants<br>➢ 64-bit RISC-V<br>➢ Standard RISC-V configurations<br>➢ RISC-V assembly language<br>➢ Implementing RISC-V in an FPGA |
| | | | ➢ simple RISC-V processor using SystemVerilog<br>➢ Assignment | ➢ RISC-V Design simulation using Verilator |
| | | | ➢ IP-XACT & Automation | ➢ Create **XML templates** for RISC-V IP blocks<br>➢ Demonstrate **automated integration** of RISC-V cores into SoC verification environments<br>➢ Align with your IP-XACT module goals for scalable workflows |
| | | | ➢ RISC-V-AI | ➢ Explore AI acceleration on RISC-V as a capstone project<br>➢ Include compiler design, OS porting, and performance optimization |

| | | | | |
|---|---|---|---|---|
| | | | | |

| Section | Section Name | Section Description | Lectures (Each Lecture is more than 30 mints) | Learning Objectives (After Completion of each lecture the Learner will be able to work on):- |
|---|---|---|---|---|
| Section-VII | Universal Verification Methodology (UVM) | ➢ Universal Verification Methodology (UVM) is an industry standard verification methodology to define, reuse, and improve the verification environment and to reduce the cost of verification.<br>➢ It provides certain application programming interfaces (APIs) for the use of base class library (BSL) components in the verification environment making them reusable and tool independent. | ➢ Introduction to UVM<br>➢ Assignment | ➢ Introduction<br>➢ Importance of UVM<br>➢ Verification Planning & Coverage Diven Verification in UVM |
| | | | ➢ UVM-Overview<br>➢ Assignment | ➢ UVM Testbench and Environments<br>➢ Interface UVCs<br>➢ System and Module UVCs<br>➢ System Verilog UVM Class Library<br>➢ UVM Utilities |
| | | | ➢ SystemVerilog Interfaces and Bus Functional Models<br>➢ Assignment | ➢ Introduction<br>➢ The TinyALU BFM (Bus Functional Model)<br>➢ Creating a Modular Testbench<br>➢ Summary of Bus Functional Models |
| | | | ➢ Object-Oriented Programming (OOP)<br>➢ Assignment | ➢ Introduction<br>➢ Importance of OOP<br>➢ Code Reuse<br>➢ Code Maintainability<br>➢ Memory Management<br>➢ Summary of Object-Oriented Programming |

| | | | | |
|---|---|---|---|---|
| | | | ➢ Coroutines<br>➢ Assignment | ➢ Coroutine Definition<br>➢ Coroutine-based Co-simulation Testbench (Cocotb)<br>➢ Role of Coroutines in Cocotb<br>➢ Time-Consuming Functions Across Languages<br>➢ Summary of Classes and Extension |
| | | | ➢ Cocotb Queue<br>➢ Assignment | ➢ Introduction<br>➢ Task Communication<br>➢ Blocking Communication<br>➢ Nonblocking Communication |
| | | | ➢ Simulating with cocotb<br>➢ Assignment | ➢ Introduction<br>➢ Verifying a counter<br>➢ cocotb triggers<br>➢ Testing reset_n<br>➢ Checking that the counter counts |
| | | | ➢ Basic testbench: 1.0<br>➢ Assignment | ➢ Introduction<br>➢ The Tiny ALU<br>➢ A cocotb testbench<br>➢ Importing modules<br>➢ The Ops enumeration<br>➢ The alu_prediction () function<br>➢ Setting up the cocotb Tiny ALU test<br>➢ Sending commands<br>➢ Sending a command and waiting for it to complete<br>➢ Checking the result<br>➢ Finishing the test |
| | | | ➢ Tiny Alu Bfm<br>➢ Assignment | ➢ Introduction<br>➢ The Tiny ALU BFM coroutines<br>➢ The tiny alu_utils module<br>➢ Living on the clock edge<br>➢ The Tiny Alu Bfm singleton<br>➢ Initializing the Tiny Alu Bfm object<br>➢ The reset() coroutine<br>➢ The communication coroutines<br>➢ Launching the coroutines using start soon ()<br>➢ Interacting with the bfm loops<br>➢ The cocotb test |

| | | | | |
|---|---|---|---|---|
| | | | ➢ Class-based testbench: 2.0<br>➢ Assignment | ➢ Introduction<br>➢ The class structure<br>➢ The BaseTester class<br>➢ The RandomTester<br>➢ The MaxTester<br>➢ The Scoreboard class<br>➢ Initialize the scoreboard<br>➢ Define the data-gathering tasks<br>➢ The Scoreboard's start_tasks() function<br>➢ The Scoreboard's check_results() function<br>➢ The execute_test() coroutine<br>➢ The cocotb tests<br>➢ |
| | | | ➢ Why UVM<br>➢ Assignment | ➢ Introduction<br>➢ How do we define tests?<br>➢ How do we build testbenches?<br>➢ How do we reuse testbench components?<br>➢ How do we create verification IP?<br>➢ How do multiple components monitor the DUT?<br>➢ How do we create stimulus?<br>➢ How do we share common data?<br>➢ How do we modify our testbench's structure in each test?<br>➢ How do we log messages?<br>➢ How do we pass data around the testbench?<br>➢ |

| | | | | |
|---|---|---|---|---|
| | | | ➢ uvm_test testbench: 3.0<br>➢ Assignment | ➢ Introduction<br>➢ The HelloWorldTest class<br>➢ Refactoring testbench 2.0 into the UVM<br>➢ The BaseTest class<br>➢ The Random Test and MaxTest classes |
| | | | ➢ uvm_component<br>➢ Assignment | ➢ Introduction<br>➢ build_phase(self)<br>➢ connect_phase(self)<br>➢ end_of_elaboration_phase(self)<br>➢ start_of_simulation_phase(self)<br>➢ run_phase(self)<br>➢ extract_phase(self)<br>➢ check_phase(self)<br>➢ report_phase(self)<br>➢ final_phase(self)<br>➢ Running the phases<br>➢ Building the testbench hierarchy<br>➢ The uvm_component instantiation arguments.<br>➢ TestTop (uvm_test_top) |

| | | | | |
|---|---|---|---|---|
| | | | | ➢ MiddleComp (uvm_test_top.mc)<br>➢ BottomComp (uvm_test_top.mc.bc)<br>➢ Running the simulation |
| | | | ➢ uvm_env testbench: 4.0<br>➢ Assignment | ➢ Introduction<br>➢ Converting the testers to UVM components<br>➢ BaseTester<br>➢ RandomTester and MaxTester<br>➢ Scoreboard<br>➢ Using an environment<br>➢ Creating RandomTest and MaxTest |

| | | | | |
|---|---|---|---|---|
| | | | ➢ Logging<br>➢ Assignment | ➢ Introduction<br>➢ Creating log messages<br>➢ Logging Levels<br>➢ Setting Logging Levels<br>➢ Logging Handlers<br>➢ Adding a handler<br>➢ Removing a handler<br>➢ Removing the default Stream Handler<br>➢ Disabling logging<br>➢ Changing the log message format |
| | | | ➢ ConfigDB ()<br>➢ Assignment | ➢ Introduction<br>➢ A hierarchy-aware dictionary<br>➢ The ConfigDB().get() method<br>➢ The ConfigDB().set() method<br>➢ Wildcards<br>➢ Global data<br>➢ Parent/child conflicts |
| | | | ➢ Debugging the ConfigDB ()<br>➢ Assignment | ➢ Introduction<br>➢ Missing data<br>➢ Catching exceptions<br>➢ Printing the ConfigDB<br>➢ Tracing ConfigDB () operations |
| | | | ➢ UVM Factory<br>➢ Assignment | ➢ Introduction<br>➢ The create() method<br>➢ uvm_factory()<br>➢ Factory overrides by instance<br>➢ Using the create() method carefully<br>➢ Debugging uvm_factory() |
| | | | ➢ UVM factory testbench: 5.0<br>➢ Assignment | ➢ Introduction<br>➢ AluEnv<br>➢ RandomTest<br>➢ MaxTest |

| | | |
|---|---|---|
| | ➢ Component communications<br>➢ Assignment | ➢ Introduction<br>➢ Why use TLM 1.0?<br>➢ Ports<br>➢ Exports<br>➢ Nonblocking communication in pyuvm<br>➢ Debugging uvm_tlm_fifo |
| | ➢ Analysis ports<br>➢ Assignment | ➢ Introduction<br>➢ The uvm_analysis_port<br>➢ Extend the uvm_analysis_export class<br>➢ Extend the uvm_subscriber class<br>➢ Instantiate a uvm_tlm_analysis_fifo |
| | ➢ Components in testbench 6.0<br>➢ Assignment | ➢ Introduction<br>➢ The testers<br>➢ Driver<br>➢ Monitor<br>➢ Coverage<br>➢ The Scoreboard |

| | | |
|---|---|---|
| | ➢ Connections in testbench 6.0<br>➢ Assignment | ➢ The AluEnv TLM diagram<br>➢ AluEnv.build_phase()<br>➢ AluEnv.connect_phase()<br>➢ RandomTest and MaxTest |
| | ➢ uvm_object in Python<br>➢ Assignment | ➢ Introduction<br>➢ Creating a string from an object<br>➢ Comparing objects<br>➢ Copying and cloning |
| | ➢ Sequence testbench: 7.0<br>➢ Assignment | ➢ Introduction<br>➢ UVM Sequence Overview<br>➢ Driver<br>➢ AluEnv<br>➢ AluSeqItem<br>➢ Creating sequences<br>➢ Starting a sequence in a test |
| | ➢ Fibonacci testbench: 7.1<br>➢ Assignment | ➢ Introduction<br>➢ Fibonacci numbers<br>➢ FibonacciSeq<br>➢ Driver<br>➢ Sequence timing<br>➢ AluEnv |
| | ➢ get_response() testbench: 7.2<br>➢ Assignment | ➢ Introduction<br>➢ AluResultItem<br>➢ Driver<br>➢ get_response() pitfalls |

| | | |
|---|---|---|
| | ➢ Virtual sequence testbench: 8.0<br>➢ Assignment | ➢ Introduction<br>➢ Launching sequences from a virtual sequence<br>➢ Running sequences in parallel<br>➢ Creating a programming interface |
| | ➢ UVM-ML Frame work<br>➢ UVM-based verification of RISC-V cores<br>➢ ML-driven test generation for instruction coverage<br>➢ Assignment | ➢ Introduction<br>➢ Platforms and Simulators<br>➢ UVM-ML Open Architecture: Status, Use<br>➢ Backplane Architecture<br>➢ Adapters<br>➢ TLM 1.0 & 2.0 Support<br>➢ Real time Use case studies with UVM-ML framework |

# Thank You