Week 9

Codefest: Test, verify, and benchmark

ECE 410/510
Spring 2025

Challenge #27

Overview and context:
Verification and validation are often confused, but they answer fundamentally different questions about a design:

- Verification: "Are we building the product right?" Verification checks whether the design correctly implements the specified requirements. It's about correctness of implementation.
- Validation: "Are we building the right product?" Validation ensures the design meets the actual user needs and intended purpose. It's about fitness for purpose.

Modern testing philosophy: "Verification is not about proving there are no bugs—it's about gaining sufficient confidence that the design will work correctly in its intended application." The goal isn't perfection (which is impossible) but rather achieving an acceptable level of quality and reliability for the specific use case. Critical systems require more rigorous testing than consumer products, but all designs benefit from systematic verification. In essence, testing and verification transform designs from "it might work" to "we're confident it will work," which is essential for any serious engineering project.

Best practices for testing:

1. Constrained Random Testing: Use Python's random libraries with constraints
2. Coverage Tracking: Monitor which design features have been tested
3. Assertion Checking: Implement both Python and SystemVerilog assertions
4. Performance Metrics: Measure simulation speed and optimize bottlenecks
5. Regression Testing: Build a test suite that can be run automatically

Learning goals:

- Test, verify, and benchmark your chiplet design in a high-/low-level co-simulation (e.g., by using cocotb, see challenge #25)
- Compare it with your software implementation.

10-Step Summary: Testing Python-Verilog Designs with cocotb:
Example code available at https://drive.google.com/file/d/104PIB59HMKCW1M5zlg_SZw-xQWTdnVWe/view?usp=sharing

Step 1: Set Up the Cocotb Environment

- Install cocotb and required simulators (Icarus, Verilator, etc.)
- Create Makefile with simulation parameters
- Define testbench structure and clock generation
- Establish reset sequences and initial test connectivity
- Configure timing units and precision

Step 2: Create Functional Tests

- Define transaction classes for structured stimulus
- Test basic input/output functionality
- Verify core features work as specified
- Implement assertion-based checking
- Test each module feature systematically

Step 3: Implement Constrained Random Verification

- Create randomized transaction generators
- Define constraints for valid input ranges

- Build drivers to send transactions to *Design under Test* (DUT)
- Implement monitors to observe outputs
- Generate diverse test scenarios automatically

Step 4: Add Coverage Metrics
- Track which features have been tested
- Define coverage points for critical signals
- Monitor state machine transitions
- Create cross-coverage between related features
- Generate coverage reports and identify gaps

Step 5: Create Python Reference Model
- Build golden model of expected behavior
- Implement same algorithms in Python
- Compare DUT outputs against reference
- Detect functional mismatches automatically
- Maintain model synchronization with DUT

Step 6: Benchmark Performance
- Measure transaction throughput
- Calculate processing latency
- Track simulation speed metrics
- Identify performance bottlenecks
- Compare against design requirements

Step 7: Set Up Regression Testing
- Create parameterized test suites
- Automate test execution across configurations
- Track test results over time
- Generate regression reports
- Detect functionality degradation

Step 8: Add Debug and Visualization
- Capture signal traces during simulation
- Generate waveform visualizations
- Export VCD files for waveform viewers
- Create custom debug monitors
- Plot performance metrics graphically

Step 9: Test Corner Cases
- Test overflow/underflow conditions
- Verify reset behavior during operations
- Test simultaneous operations conflicts
- Check clock domain crossing safety
- Validate error handling mechanisms

Step 10: Final Validation and Report Generation
- Run complete test suite systematically
- Compile all test results
- Analyze coverage completeness
- Document performance measurements
- Generate comprehensive validation reports
- Provide pass/fail recommendations
- Archive results for future reference