IC Lab Formal Verification Bonus Report 2024 Fall

Name:_____ **Student ID:_** **Account:_____**

(a) What is Formal verification?

Formal verification is a mathematical approach used to ensure that a hardware design functions as intended. Unlike traditional simulation, which tests the design with selected input scenarios, formal verification systematically explores all possible combinations of inputs and states, leaving no corner case unchecked.

The process involves:
1. Exhaustively testing all input port values and undriven wire combinations for each cycle.
2. For the initial cycle, verifying all possible values of uninitialized registers.
3. Generating a detailed report that highlights any violated assertions and identifies covered properties.

This method provides a higher level of confidence in the correctness of the design, particularly during the front-end development stage.

What's the difference between Formal and Pattern based verification?

And list the pros and cons for each.

Formal Verification

- Description:
  Formal verification uses mathematical techniques to exhaustively analyze all possible states and input scenarios of a design. It aims to ensure correctness by thoroughly checking every potential case, including corner cases.
- Pros:
  Guarantees 100% coverage, including both common and corner cases.
  Provides a higher degree of confidence in the design's correctness.
- Cons:
  Computationally intensive and time-consuming.
  Scalability can become a bottleneck for highly complex designs, as traversing all possibilities may be infeasible within reasonable time.

Pattern-Based Verification

- Description:

  Pattern-based verification relies on generating specific input patterns and comparing the design's outputs to pre-determined golden results. It is primarily simulation-based and tests a subset of possible scenarios.
- Pros:

  Faster and more efficient for large, computation-heavy designs.

  Provides flexibility in focusing on specific test cases or scenarios.
- Cons:

  Limited coverage, making it prone to missing corner cases.

  Heavily dependent on the designer's ability to craft comprehensive and effective test patterns.

Key Differences

- Scope:

  Formal verification ensures exhaustive coverage of all scenarios.

  Pattern-based verification tests only selected scenarios based on designed patterns.
- Use Cases:

  Formal verification is ideal for verifying protocol adherence and critical correctness.

  Pattern-based verification is more suitable for designs involving extensive computations or high throughput.

(b) Explain SVA (SystemVerilog Assertions) and the roles of Assertion, Cover, and Assumption.

- SVA is a feature of SystemVerilog used to specify and verify the expected behavior of hardware designs. It provides a way to check correctness and validate designs by embedding properties directly in the HDL code.

- Assertion: Specifies that certain conditions must hold true during simulation or formal verification. Assertions are used to detect design errors by ensuring that specific behaviors or states are always satisfied.

- Cover: Captures and tracks scenarios or events of interest to verify that all critical parts of the design are exercised during testing. Coverage helps ensure completeness in validation.

- Assumption: Specifies constraints on the design's environment or inputs during formal verification. Assumptions define conditions that are expected to hold true, guiding the tool to focus on valid scenarios.

What is glue logic?

Glue logic refers to auxiliary or intermediate logic signals added to simplify complex relationships or conditions in a design, particularly when used in assertions or property descriptions.

- ➢ For example, instead of directly describing a complex behavior within every assertion, a simpler signal (like in_packet) can be defined to encapsulate the repeated behavior. This signal can then be referenced in multiple assertions, making the SVA code more concise and easier to maintain.
- ➢ Glue logic is typically used only for validation purposes and is not synthesized into the final hardware, so it does not impact the actual design.

Why will we use glue logic to simplify our SVA expression?

Glue logic is used to reduce the complexity of SVA expressions by breaking down intricate conditions into manageable components:

- ➢ It improves readability by encapsulating frequently used or complex behaviors into dedicated logic signals.
- ➢ It enhances reusability, as the same glue logic signal can be used across multiple assertions, reducing redundancy.
- ➢ It helps minimize errors in SVA expressions by isolating and abstracting complex logic into simpler units, making debugging and validation more straightforward.
  For instance, instead of repeating a long logical expression in several assertions, glue logic can encapsulate that behavior into a single signal, referenced throughout the SVA code.

(c) What is the difference between Functional coverage and Code coverage?

Functional Coverage

- ➢ Definition: Functional coverage involves verifying that specific behaviors or scenarios defined in the design specifications are exercised during simulation or testing. This is typically implemented using user-defined covergroups and assertions.
- ➢ Pros:

  Ensures that critical functional requirements are thoroughly tested.

  Provides targeted feedback on whether specific design features are adequately validated.
- ➢ Cons:

  Time-consuming to create and maintain, as it requires manual effort.

  Prone to human error, such as missing or incorrectly defined coverage points.

Code Coverage

- Definition: Code coverage checks whether all parts of the HDL code—such as branches, statements, or conditions—are executed during testing. It is automatically generated by coverage tools.
- Pros:

  Automated and quick to generate.

  Provides an overview of untested parts of the code.
- Cons:

  Does not ensure that the design's intended functionality is fully tested.

  Can give a false sense of completeness if code is executed but does not perform correctly (e.g., unmeaningful coding paths).

What's the meaning of 100% code coverage, could we claim that our assertion is well enough for verification? Why?

No, achieving 100% code coverage does not necessarily mean that the assertions or tests are sufficient for thorough verification.

Reason:

1. Code Execution ≠ Functional Validation:

- 100% code coverage only confirms that every line, branch, or condition in the code has been executed during simulation.
- It does not verify whether the executed code behaves correctly or adheres to the intended design specifications.

2. Missed Functional Scenarios:

- Some critical functional behaviors or corner cases might not be explicitly tested, even if all the code paths are executed.
- Functional coverage or checker-based validation is necessary to confirm that specific requirements or scenarios have been exercised and validated.

3. Unmeaningful Execution:

- A line of code could be executed without meaningful interaction or impact on the functionality, leading to false confidence in the verification quality.

(d) What is the difference between COI coverage and proof coverage for realizing checker's

completeness? Try to explain from the meaning, relationship, and tool effort perspective.

Meaning:

- COI Coverage: Tracks all signals that influence an assertion, whether directly or indirectly, during simulation.
- Proof Coverage: Verifies only the signals explicitly referenced in the assertion using formal methods for all possible scenarios.

Relationship:

- Proof coverage is a subset of COI coverage. COI includes all influencing signals, while proof focuses on the explicitly referenced ones.

Tool Effort:

- COI coverage is faster and uses fewer resources as it relies on simulation.
- Proof coverage is more resource-intensive, requiring formal tools to perform exhaustive analysis.

Summary: COI ensures complete influence tracking, while proof ensures correctness through formal verification. Both together achieve comprehensive checker coverage.

(e) What are the roles of ABVIP and scoreboard separately?
Try to explain the definition, objective, and the benefit.

Definition:

- ABVIP: Pre-designed IP with assertions for protocol compliance.
- Scoreboard: Compares DUV outputs with expected outputs to verify correctness.

Objective:

- ABVIP: Simplifies protocol testing and speeds up verification.
- Scoreboard: Ensures the DUV functions correctly by checking input-output consistency.

Benefits:

- ABVIP: Saves time, ensures reliability, and provides reusable tools.
- Scoreboard: Reduces errors, enhances coverage, and scales to complex designs.

(f) Among the JasperGold tools (Formal Verification, SuperLint, Jasper CDC, IMC

Coverage), which one do you think is the most effective based on its functionality and typical application scenarios? Please explain your reasoning by describing a hypothetical scenario where this tool would be particularly beneficial, and discuss any potential challenges or limitations that might arise when using it.

I believe Formal Verification is the most effective JasperGold tool. Formal Verification is particularly beneficial in scenarios involving large-scale circuit designs, where ensuring correctness across all states is critical but can be computationally expensive. For example, in a complex multi-core processor design, verifying the entire system at once using Formal Verification might be infeasible due to the overwhelming state space and the long convergence time.

➢ Hypothetical Scenario

Imagine designing a cache controller for a multi-core processor. The system involves multiple smaller modules, such as arbitration, coherency checks, and data routing. Instead of applying Formal Verification to the entire cache controller, we adopt a hierarchical modular design:

1. Module-level Verification:

- Apply Formal Verification to smaller modules, such as verifying that the arbitration module handles requests correctly without deadlocks or ensuring that the coherency module adheres to protocol rules.
- This significantly reduces the state space, as each module is smaller and less complex.

2. Integration-level Verification:

- After verifying the individual modules, integrate them into the larger design.
- At this level, perform simulation-based verification to check interactions between modules, complemented by targeted Formal Verification for critical paths or corner cases.

➢ Benefits

- Efficiency: Verifying smaller modules reduces runtime and computational resource requirements while maintaining the thoroughness of Formal Verification.
- Scalability: By breaking down the design into manageable pieces, we ensure that the tool can handle the state space effectively.
- Robustness: Combining module-level verification with integration testing ensures that the entire system is validated without overwhelming verification tools.

This hierarchical approach allows designers to leverage the strengths of Formal Verification while mitigating its runtime limitations in large-scale designs.