IC Lab Formal Verification Bonus Report 2024 Fall

Name: 鄭喆嚴 **Student ID:** 313510184 **Account:** iclab051

(a) What is Formal verification?

Formal verification is a mathematical approach to verify the correctness of systems or designs by proving that they meet specified properties, typically using techniques like model checking or theorem proving.

Pattern-based verification, on the other hand, involves using pre-defined patterns or templates to verify properties of the system, relying on heuristics and statistical methods rather than rigorous mathematical proofs.

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

Formal Verification is rigorous and guarantees correctness, while Pattern-based Verification is more heuristic and doesn't offer absolute correctness.

And list the pros and cons for each.

Formal Verification:

Pros: Provides absolute correctness guarantees, detects subtle bugs, and ensures no corner cases are missed.

Cons: Computationally expensive, time-consuming, and complex to implement.

Pattern-based Verification:

Pros: Faster, easier to implement, and can cover a large design space.

Cons: May miss corner cases and lacks the guarantee of absolute correctness.

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

SystemVerilog Assertions (SVA) are a set of features in the SystemVerilog language designed to help verify and validate digital designs. SVA allows you to specify properties or behaviors of a design that can be checked automatically during simulation. These assertions help ensure that the design behaves as expected under various conditions and constraints.

Assertion:

Assertions are used to specify expected behaviors or properties that the design should always satisfy. An assertion typically checks whether a certain condition holds true during simulation, and if it doesn't, an error is reported.

Cover:

A cover is used to track whether certain conditions or state transitions are being exercised during simulation. It doesn't trigger errors but logs coverage information, helping you understand which parts of the design have been tested.

Assumption:

Assumptions define conditions that must be assumed to be true for a property to be checked. They are typically used to model the environment in which the design operates, and they tell the simulator that certain conditions are valid during verification.

What is glue logic?

Glue logic refers to small pieces of logic that are used to connect different parts of a design, typically between subsystems that may have incompatible signals or interfaces. It helps in adapting or combining different signal types or protocols so that they can work together seamlessly.

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

We use glue logic to simplify SystemVerilog Assertions (SVA) expressions because it helps break down complex conditions into more manageable parts. By adding intermediary logic, we can isolate specific signal behaviors or transitions, making the SVA expression clearer and easier to debug. Glue logic acts as a bridge to reduce the complexity of the property being asserted, ensuring that each part of the design is tested independently before combining them into more comprehensive assertions. This also improves simulation efficiency and readability by minimizing the number of nested or convoluted conditions in a single assertion.

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

Functional Coverage refers to measuring whether the specific functional behaviors or requirements of the design have been fully exercised during simulation. It ensures that all the specified scenarios, edge cases, and state transitions defined in the design specifications are verified. For example, it checks if all states in a state machine are visited or if certain input combinations lead to the expected outputs.

Code Coverage, on the other hand, measures how much of the design's code (e.g., lines, branches, conditions) has been executed during simulation. It is concerned with the structural aspects of the design and ensures that all the lines of code are run at least once, helping to identify dead code or unreachable paths.

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

100% code coverage means that every part of the code (lines, branches, conditions) has been executed during the simulation. This indicates that the code has been fully exercised, but it does not guarantee that the design is functionally correct. Code coverage only ensures that the code is being run, not that the behavior of the design meets the required specifications.

No, 100% code coverage alone does not guarantee that the assertion is sufficient for verification. While it ensures that all parts of the code have been executed, it does not verify that the design behaves correctly under all conditions. The design could still have functional bugs or unmet requirements even if all the code is covered. Assertions need to be carefully defined to capture the intended functionality, and functional coverage must also be considered to ensure all desired behaviors are tested. Code coverage focuses on structural testing but does not fully address functional correctness.

(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.

COI Coverage (Check-Only Items Coverage) and Proof Coverage are both used to assess the completeness of verification checkers and the design's verification process. However, they differ in their focus, relationship, and the level of effort required by the verification tools.

Meaning:

COI Coverage:

COI coverage is focused on ensuring that all the relevant check-only items, which refer to specific functional behaviors or conditions that the checker is supposed to monitor, are being checked. These items are typically related to specific assertions, functional checks, or scenarios that need to be verified within the design.

The goal is to track whether all the conditions that are required to be checked during simulation have been covered by the existing checkers, making sure that the checker is sufficiently verifying the design's behavior.

Proof Coverage:

Proof coverage refers to the completeness of formal verification tools in proving that certain properties or assertions hold true for all possible input conditions in the design. It involves checking whether all the properties (typically safety or liveness properties) have been proven to hold, often through formal methods such as model checking or theorem proving.

The goal of proof coverage is to ensure that the formal verification has exhaustively proven the correctness of the design under all possible conditions, guaranteeing no counterexamples or errors remain.

Relationship:

COI Coverage:

COI Coverage is more focused on ensuring that functional checks and assertions are adequately exercised during simulation. It provides a qualitative measure of whether the verification environment is sufficiently covering all required functional conditions.

Proof Coverage:

Proof Coverage, on the other hand, is linked to the formal verification process and ensures that all properties of the design have been proven to hold, often over all possible inputs or design states.

These two coverage types complement each other: COI coverage ensures the functional aspects are verified through simulations, while proof coverage ensures correctness via formal verification.

Tool Effort:

COI Coverage:

Tools that measure COI coverage (e.g., simulation-based tools, functional verification tools) focus on tracking which check-only items have been covered during simulation. The effort is focused on writing the functional checkers, assertions, and coverage items.

The effort is generally moderate because it involves simulation runs to cover the designed check-only items. However, complete coverage depends on the quality and depth of the checkers and the range of test cases used.

Proof Coverage:

Formal verification tools that handle proof coverage (e.g., model checkers, theorem provers) require significantly more computational effort. They attempt to prove that the design satisfies certain properties exhaustively.

The effort for proof coverage can be high, as it involves exhaustive verification across all design states and inputs. Proof tools need to analyze all possible behaviors, which can be computationally expensive, especially for complex designs with many states.

(e) What are the roles of **ABVIP** and **scoreboard** separately?

Try to explain the definition, objective, and the benefit.

ABVIP (Automated Bus Functional Interface Protocol):

Definition:

ABVIP refers to a set of tools and methods used to model, monitor, and verify the behavior of bus protocols in a verification environment. It is part of a testbench framework that automates the verification of the communication between different blocks in a system, ensuring that the transactions between modules on a bus follow the correct protocol and adhere to the expected timing and data integrity rules.

Objective:

The main objective of ABVIP is to ensure that the bus protocol used for communication between components (e.g., memory, processors, or peripherals) is properly tested. It automates the task of checking that all read/write transactions, arbitration, handshaking, and other protocol-specific behaviors occur correctly and in the right order.

Benefit:

Efficiency: ABVIP automates the protocol verification process, reducing the need for manually writing verification code for each transaction, and increasing verification coverage.

Reusability: It allows the verification environment to be reused across different projects or versions of the design.

Accuracy: Ensures that the bus protocol is correctly implemented by checking various corner cases and edge conditions.

Automation: Simplifies and accelerates the verification process by automating the generation of test cases, monitoring transactions, and checking protocol compliance.

Scoreboard

Definition:

A scoreboard is a functional verification tool used in testbenches to track and compare expected and actual results during simulation. It serves as a reference model to ensure that the design's output matches the expected values after each transaction, and it checks the correctness of the design’s functionality by tracking data flow and comparing it to the expected behavior.

Objective:

The primary objective of the scoreboard is to track and compare the outputs generated by the design under test (DUT) with the expected outputs. It keeps track of the data or transactions sent through the DUT, and after processing, compares these results against the expected results based on the inputs, ensuring that the system operates correctly.

Benefit:

Verification Accuracy: The scoreboard provides a mechanism to catch discrepancies between expected and actual results, thus helping identify functional errors or bugs in the DUT.

Complexity Management: It helps manage and track complex data flows or sequences over time, providing a high-level overview of what has been processed and whether it conforms to expectations.

Post-Simulation Analysis: After the simulation runs, the scoreboard provides a detailed report of the transaction results, making it easier to debug and analyze issues.

Decoupling Verification from Implementation: The scoreboard allows designers to verify functional correctness without being tightly coupled to the design implementation details, making it more flexible and reusable across different 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.

Formal Verification stands out as the most powerful and comprehensive tool based on its functionality and typical application scenarios. Let's break down why this is the case.

Functionality and Application:

Formal Verification uses mathematical techniques to prove the correctness of a design by exhaustively exploring all possible states and inputs, ensuring that certain properties or behaviors hold true for the entire system. This makes it highly effective for verifying safety properties, liveness conditions, and functional correctness.

Typical Application Scenarios:

Critical Systems:

When designing safety-critical or mission-critical systems (e.g., aerospace, automotive, medical devices), formal verification can guarantee that the system behaves correctly in all possible scenarios, including edge cases that are difficult to simulate.

Complex Designs:

In complex digital systems like multi-core processors, SoCs (System on Chips), or highly concurrent systems, formal verification ensures that all design specifications and corner cases are rigorously checked.

Ensuring Property Safety:

It is particularly useful when you need to verify specific properties like deadlock-freedom, correctness of handshakes in communication protocols, or absence of race conditions in concurrent systems.

Hypothetical Scenario:

Imagine a high-performance, multi-core processor design that incorporates complex inter-core communication, cache coherency protocols, and intricate power management. Verifying the functional correctness of such a design purely through simulation would be time-consuming and potentially inadequate to cover all corner cases. In this scenario, Formal Verification would be invaluable because:

It can exhaustively explore all states, ensuring that complex scenarios like inter-core data consistency and cache invalidation are correctly handled across all possible states.

It can prove the absence of deadlocks, race conditions, and other hard-to-detect issues that could cause the system to behave incorrectly in edge cases.

It can provide a formal guarantee that the system meets its specifications in terms of functionality, correctness, and reliability.

Challenges and Limitations:

Scalability:

Formal verification is computationally expensive and can be difficult to scale to very large designs. The state space explosion problem may occur if the design has a large number of variables or a complex control flow.

Initial Setup Complexity:

Defining the properties to be verified and setting up the formal verification environment can be challenging. It requires careful modeling of the design and its behaviors.

Limited Property Coverage:

While formal verification can prove the correctness of certain properties, it might not be able to verify all aspects of a design (e.g., time-dependent behaviors or certain complex interactions), so combining it with other verification methods may still be necessary.