IC Lab Formal Verification
Lab11 Quick Test
2023 Fall

Name: 李松磊 **Student ID:** 312540001 **Account:** iclab129

(a) What is Formal verification and Pattern based verification?

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

And list the pros and cons for each.

Ans:

Formal verification and pattern-based verification are two approaches used to ensure the correctness of digital circuits and systems.

Formal Verification:

Formal verification involves mathematically proving that a system satisfies certain properties or specifications. It uses formal methods such as mathematical modeling, theorem proving, and logic to analyze the system. It encompasses: Model checking, Theorem proving and Equivalence checking.

Pros:

Accuracy: Provides rigorous mathematical proofs, ensuring correctness within specified properties.

Completeness: Can explore the entire state space, verifying all possible scenarios.

Bug Detection: Can uncover subtle errors or corner cases that might be missed in other methods.

Guaranteed Correctness: When a property is proven, it guarantees correctness under specified conditions.

Model Checking: Once the system model and property specifications are provided to the model checker, the verification is fully automated.

Theorem Proving: Because the mathematically way is employed therefore, we can handle the verification seamlessly for relatively complex systems.

Equivalence Checking: This process allows for ensuring that two designs functionality are same. Given the inputs are identical thus the expected output should be same.

Cons:

Complexity: Formal verification can be complex and resource-intensive, especially for large-scale systems. Given for example the states of a system increase with complexity. The number of possibilities therefore can become very large. Thus, it might not be feasible for very large designs due to computational constraints.

Expertise Required: Requires specialized skills in formal methods, making it inaccessible to some engineers. The reason is because theorem proving is not fully automated. It requires a certain amount of human intervention to complete. No counterexamples would be generated in case of failure to that proof. Thus, making the probing for such problem even more troublesome.

False Negatives: It's possible for a property to hold true but not be provable due to the complexity of the system or limitations in the verification tools.

Pattern-Based Verification:

Methodology: Pattern-based verification, also known as simulation-based verification, relies on the execution of predefined test patterns or scenarios to validate the system's behavior against expected outcomes.

Pros:

Ease of Use: Relatively simpler to implement and understand compared to formal methods. Also does not require detailed knowledge of the internal working circuitry of the Design under test.

Scalability: Suitable for large designs and can be applied to a wide range of systems. Which makes it more versatile in many cases.

Early Bug Detection: Can quickly uncover obvious bugs or errors during simulation runs. Errors can be found early in the design cycle, which can save lot of time and resources.

Tool Availability: Many simulation tools and environments support pattern-based verification.

Coverage: Pattern-based can provide a high level of test coverage since it can verify many different input patterns.

Cons:

Incomplete Coverage: Relies on predefined test patterns, leaving gaps in coverage that might miss certain corner cases. Furthermore, since it verifies the output patterns of the circuit, without supervision to the internal operation or behavior, this method may not detect certain types of errors such as timing violations or functional errors.

Limited Guarantee: Cannot guarantee the absence of bugs beyond the scope of tested patterns. Since we are dependent on test patterns, the effectiveness is also dependent on the quality and number of input test patterns used. If the test patterns are not robustly represented, some errors may still go undetected.

False Sense of Security: Passing tests don't ensure absence of bugs; some critical issues might

remain undetected.

In conclusion, formal verification offers a higher level of assurance and correctness but is more complex and resource intensive while pattern-based verification is simpler and more scalable but might not provide the same level of rigorous verification and can miss subtle errors. Often, a combination of both methods is used to complement each other's strengths and weaknesses in the verification process.

(b) What is glue logic?

Glue logic refers to the circuitry or logic elements used to connect different components or modules within a larger system ("glue"), especially when they have different interfaces, operating voltages, or signal types. It acts as an intermediary, facilitating communication and ensuring compatibility between these components. . It must be carefully designed to ensure that all the components are properly connected and communicating with each other, and that the signals are properly synchronized to prevent timing issues or data corruption.

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

When it comes to SystemVerilog Assertions (SVA), which are used for formal verification the use of glue logic can simplify SVA expressions in a few ways:

Interface Adaptation: Glue logic can be employed to adapt or transform signals between different interfaces. For instance, if one module communicates using certain signal conventions and another module expects different conventions, glue logic can bridge this gap, allowing the SVA to focus on a standardized interface.

Signal Conditioning: Sometimes, signals might need conditioning or manipulation before they can be used in assertions. Glue logic can perform these operations, ensuring that the signals fed into the SVA meet the required conditions or formats.

Abstraction: In complex systems, using glue logic to abstract certain details can make the SVA expression simpler and more readable. By handling low-level interactions or conversions, the SVA can focus on higher-level properties, leading to a more concise and manageable assertion.

Modularization: Glue logic can encapsulate the intricacies of interfacing between modules. By creating well-defined interfaces and using glue logic to manage these interfaces, the SVA expressions can concentrate on the behavior and properties of individual modules rather than the intricacies of their connections.

Employing glue logic in a hardware design context can simplify SVA expressions by handling signal interfacing, conditioning, and abstraction. It allows designers to focus the assertions on the essential properties of the system or modules under verification, making the verification process more manageable and effective. Glue logic can be used to break down complex components into smaller and simpler subcomponents, which can be easier to verify using SVA

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

Functional coverage and code coverage are both metrics used in verification to evaluate the thoroughness and completeness of testing, but they differ in what aspects of the design they assess.

Functional Coverage: Tracks the functionality of the design that has been exercised during simulation or testing. It measures how well the design's functional requirements or properties have been tested. It helps ensure that different scenarios or functionalities of the design have been adequately tested, even if the code paths have been covered. It focuses on what the design is supposed to do, checking whether all important behaviors and corner cases have been tested.

Code Coverage: on the other hand, measures the extent to which the code itself has been exercised during testing. It evaluates which parts of the code (lines, branches, statements, etc.) have been executed at least once during simulation or testing. It primarily focuses on ensuring that every line of code or code construct has been executed at least once, aiming to identify untested or dead code.

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

100% Code Coverage:

Achieving 100% code coverage means that every line of code in the design has been executed at least once during simulation or testing process. While achieving this metric is essential to ensure that all parts of the code have been exercised, it doesn't guarantee the correctness of the design, or the effectiveness of the assertions used for verification.

Assertion and Verification:

Even if 100% code coverage is achieved, it doesn't necessarily imply that the assertions used for verification are sufficient or that all possible scenarios have been thoroughly tested. Assertions are specific statements that check certain properties or behaviors of the design, and achieving code coverage does not confirm that all relevant design properties have been verified. Assertions need to be designed intelligently to cover various scenarios, edge cases, and functional requirements. They should be crafted based on the functional specifications and critical design properties. So, while code coverage is essential, having 100% code coverage doesn't guarantee that all aspects of the design have been adequately verified.

Therefore, a comprehensive verification strategy should include both effective assertions targeting specific design properties and achieving high code coverage to ensure a robust and thoroughly tested design.

Indeed, while code coverage gauges the code's execution extent, it doesn't assure its correct functionality. The presence of undetected errors in the design despite full code execution is possible. Assertions play a crucial role in validating the intended behavior and ensuring

alignment with specified requirements. Prioritizing comprehensive functional coverage and validating the design's intended behavior holds greater significance than merely aiming for 100% code coverage.

(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.

Ans:

COI coverage (Cover on Invalid) and proof coverage are metrics used in formal verification to evaluate the completeness of checkers or assertions, but they focus on different aspects of verification.

COI Coverage (Cover on Invalid):

Meaning: COI coverage measures how well a checker or assertion detects invalid scenarios or violations. It indicates the extent to which the checker activates or "covers" when the property it's monitoring becomes invalid or violated.

Relationship: It's closely related to identifying cases where the property being checked is not holding or where the design violates the specified properties. Achieving high COI coverage ensures that the checker adequately flags violations or discrepancies from expected behavior.

Tool Effort Perspective: Evaluating COI coverage often requires analyzing simulation traces or counterexamples generated by the formal verification tool. It involves determining how effectively the checker responds to invalid conditions in the design.

Proof Coverage: focuses on how thoroughly the formal verification tool explores the design space and proves the properties specified by the assertions or checkers.

Relationship: It's related to demonstrating the absence of violations or proving the correctness of the design with respect to the specified properties. High proof coverage indicates that the formal tool has successfully explored and verified the design space against the properties.

Tool Effort Perspective: Achieving high proof coverage involves exhaustive exploration of the design's state space using mathematical and logical techniques to demonstrate that the properties hold under all conditions.

Relationship and Tool Effort Perspective: COI coverage is more concerned with the effectiveness of the checker in identifying violations or invalid scenarios during simulation or analysis. It's more focused on the behavior of the checker when the property is violated.

- Proof coverage, on the other hand, is focused on the completeness of the formal verification process itself. It's about ensuring that the verification tool exhaustively explores and confirms the absence of violations or that the properties hold across the entire design space.
- Achieving high COI coverage might not necessarily guarantee comprehensive proof coverage, as it's possible for a checker to respond well to invalid scenarios but for the formal tool to miss certain valid scenarios that require deeper exploration.
- From a tool effort perspective, COI coverage analysis might be relatively easier and involve

examining specific cases of violation, while proof coverage requires more extensive computational effort to explore the entire design space and prove correctness across all scenarios.

- Both COI coverage and proof coverage are essential for ensuring the robustness and completeness of formal verification, but they address different aspects of the verification process and contribute to overall confidence in the correctness of the design.

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

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

ABVIP (Assertion-Based Verification IP) and scoreboards are crucial components in the realm of hardware verification, each serving distinct yet complementary roles in the verification process.

Assertion-Based Verification IP (ABVIP):

ABVIP refers to pre-designed verification IP modules integrated into the verification environment that contain assertion-based checks to validate the behavior of a design under test (DUT). These checks are created using assertion languages like System Verilog Assertions (SVA) or Property Specification Language (PSL).

Objective: The primary objective of ABVIP is to define and implement assertions that capture the intended behavior, constraints, and properties of the DUT. These assertions act as monitors that continuously observe the behavior of the DUT during simulation or formal verification runs.

Benefit: ABVIP plays a pivotal role in the verification process by providing a systematic and formal way to specify and check the correctness of the DUT. By expressing design properties in a formal language, ABVIP helps to detect violations early in the verification cycle, enabling quicker identification and resolution of issues. It enhances verification completeness by ensuring that the DUT operates according to specified requirements and constraints.

Scoreboard:

A scoreboard is a verification component used to cross-check and compare the outputs or behavior of the DUT against expected or reference outputs. It captures and evaluates the responses generated by the DUT based on predefined reference models or golden reference data.

Objective: The primary objective of a scoreboard is to verify the correctness of the DUT's outputs or behavior by comparing them with expected results. It performs this comparison by maintaining and updating reference models that represent the expected behavior of the DUT.

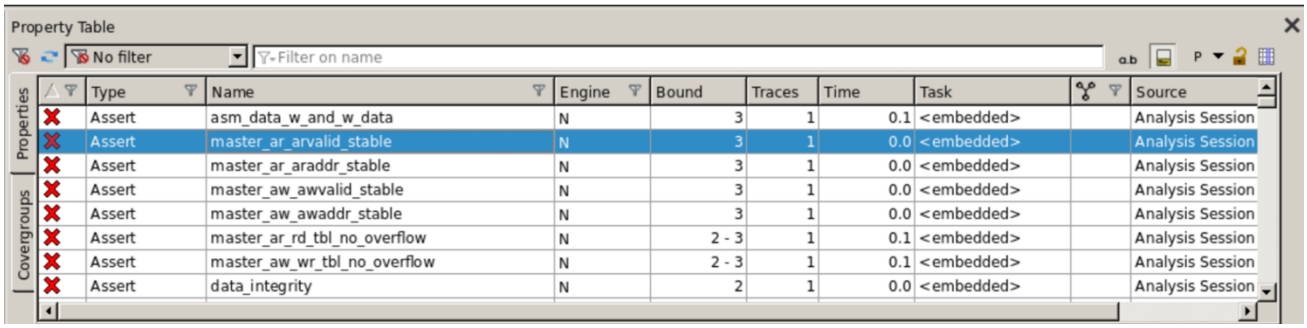
Benefit: Scoreboards are valuable for validating the functional correctness of the DUT by verifying its outputs against expected values or behaviors. They enhance the verification process by providing a means to identify discrepancies or deviations between the actual behavior of the DUT and the expected behavior defined by the reference models. This facilitates the detection of bugs or inconsistencies in the design.

Relationship and Integration:

- ABVIP focuses on specifying and checking the correctness of the design's behavior through assertions, while scoreboards focus on comparing the DUT's outputs against expected results.
- Both ABVIP and scoreboards contribute to comprehensive verification by complementing each other: ABVIP checks the internal behavior of the DUT, ensuring it adheres to defined properties, while scoreboards validate the external behavior by comparing outputs against expected values.
- Integrating ABVIP with scoreboards creates a robust verification environment that combines formal property checking with output validation, significantly enhancing the confidence in the correctness and functionality of the DUT.

(f) List four bugs in Lab Exercise, What is the answer of the Lab Exercise?

Property table Before debug:



Type	Name	Engine	Bound	Traces	Time	Task	Source
Assert	asm_data_w_and_w_data	N	3	1	0.1	<embedded>	Analysis Session
Assert	master_ar_arvalid_stable	N	3	1	0.0	<embedded>	Analysis Session
Assert	master_ar_araddr_stable	N	3	1	0.0	<embedded>	Analysis Session
Assert	master_aw_awvalid_stable	N	3	1	0.0	<embedded>	Analysis Session
Assert	master_aw_awaddr_stable	N	3	1	0.0	<embedded>	Analysis Session
Assert	master_ar_rd_tbl_no_overflow	N	2 - 3	1	0.1	<embedded>	Analysis Session
Assert	master_aw_wr_tbl_no_overflow	N	2 - 3	1	0.1	<embedded>	Analysis Session
Assert	data_integrity	N	2	1	0.0	<embedded>	Analysis Session

Ans:

(1)- inf.AW_VALID should be pulled high if(n_state == AXI_AW) instead of if(inf.AW_READY)

Before fix:

```
119  always_ff@(posedge clk or negedge inf.rst_n) begin
120      if(!inf.rst_n)begin
121          inf.AW_VALID <= 'b0;
122      end
123      else begin
124          if(inf.AW_READY)    inf.AW_VALID <=  1'b1;
125          else                    inf.AW_VALID <=  1'b0;
126      end
127  end
```

After fix:

```
129  always_ff@(posedge clk or negedge inf.rst_n) begin
130      if(!inf.rst_n)begin
131          inf.AW_VALID <= 'b0;
132      end
133      else begin
134          if(n_state==AXI_AW)     inf.AW_VALID <=  1'b1;
135          else                    inf.AW_VALID <=  1'b0;
136      end
137  end
138
```

(2)- inf.AR_VALID should be pulled high if (n_state == AXI_AR) instead of if(inf.AR_READY)

Before fix:

```
85   always_ff@(posedge clk or negedge inf.rst_n) begin
86       if(!inf.rst_n)begin
87           inf.AR_VALID <= 'b0;
88       end
89       else begin
90           if(inf.AR_READY)  inf.AR_VALID <=  1'b1;
91           else                  inf.AR_VALID <=  1'b0;
92       end
93   end
```

After fix:

```
95   always_ff@(posedge clk or negedge inf.rst_n) begin
96       if(!inf.rst_n)begin
97           inf.AR_VALID <= 'b0;
98       end
99       else begin
100          if(n_state==AXI_AR)  inf.AR_VALID <=  1'b1;
101          else                   inf.AR_VALID <=  1'b0;
102      end
103  end
```

(3)- Wrong format for inf.AW_ADDR (b instead of h)

Before fix:

```
148
149  always_ff@(posedge clk or negedge inf.rst_n) begin
150      if(!inf.rst_n)begin
151          inf.AW_ADDR <= 'b0;
152      end
153      else begin
154          if(n_state == AXI_AW && c_state != AXI_AW)   inf.AW_ADDR <= {8'h1000_0000, inf.C_addr, 2'b0};
155          else                                        inf.AW_ADDR <= inf.AW_ADDR ;
156      end
157  end
```

After fix:

```
158
159  always_ff@(posedge clk or negedge inf.rst_n) begin
160      if(!inf.rst_n)begin
161          inf.AW_ADDR <= 'b0;
162      end
163      else begin
164          if(n_state == AXI_AW && c_state != AXI_AW)   inf.AW_ADDR <= {8'b10000000, inf.C_addr, 2'b0};
165          else                                        inf.AW_ADDR <= inf.AW_ADDR ;
166      end
167  end
```

(4)- . inf.W_DATA should be given inf.C_data_w if(inf.C_in_valid&!inf.C_r_wb) (inf.C_r_wb = 0 : Write, inf.C_r_wb = 1 : Read)

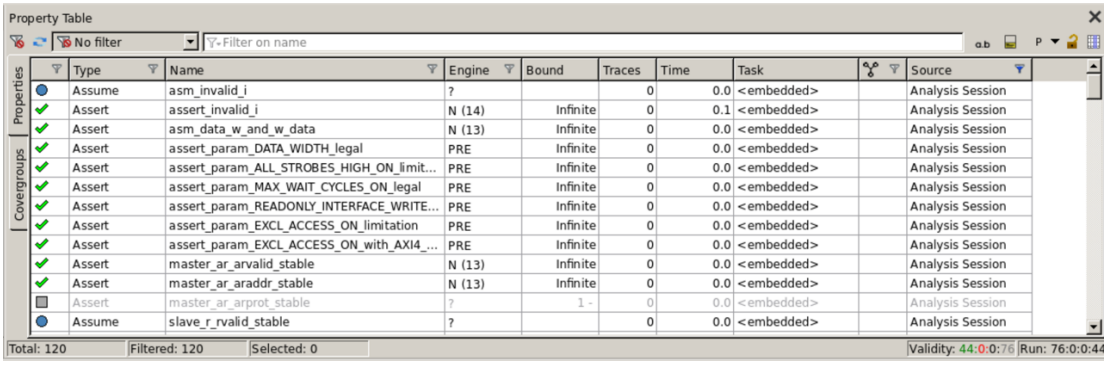
Before fix:

```
170  always_ff@(posedge clk or negedge inf.rst_n) begin
171      if(!inf.rst_n)begin
172          inf.W_DATA  <= 'b0;
173      end
174      else begin
175          if(inf.C_in_valid && inf.C_r_wb)      inf.W_DATA  <= inf.C_data_w;
176          else                                  inf.W_DATA  <= inf.W_DATA  ;
177      end
178  end
```

After fix:

```
180  always_ff@(posedge clk or negedge inf.rst_n) begin
181      if(!inf.rst_n)begin
182          inf.W_DATA  <= 'b0;
183      end
184      else begin
185          if(inf.C_in_valid && !inf.C_r_wb)      inf.W_DATA  <= inf.C_data_w;
186          else                                  inf.W_DATA  <= inf.W_DATA  ;
187      end
188  end
```

Property table After debug:



(g) Among the JasperGold tools (Formal Verification, SuperLint, Jasper CDC, IMC Coverage), which one have you found to be the most effective in your verification process? Please describe a specific scenario where you applied this tool, detailing how it benefited your workflow and any challenge you encountered while using it.

I used the formal verification tool in Jasper Gold. This functionality is very well thought after and easy to use while being such a powerful tool. In using assertions/assume and coverage expressions to detect the various bugs in the design, the tool created a platform on the form of property table for the answer to emerge along with details of each occurrence. It also allows independently browsing the different expressions of such assertions/assume/or covers

for more details including whether it is a proven property or not. The tool thus allows a seamless view of all the tasks in the design hierarchy that allows us as designers a convenient view of the source of any anomalies that could be uncovered.

I used this tool extensively to solve the bugs in this assignment namely finding whether valid_write-data assertion is violated where inf.AW_VALID should be pulled high after one cycle when we get to the next state (AXI_AW) instead of the one pulse if(inf.AW_READY) because it needs to be high until the state is invalid i.e. !(inf.AR_VALID & inf.AR_READY) we can clearly see in the before debugging property table that assertions are showed as red which mean not pass.