IC Lab Formal Verification  
Lab11 Quick Test

2023 Fall

**Name: Student ID: Account: iclab122 y**

1. What is Formal verification?

|  |
| --- |
| Formal verification is the act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics. Usually, formal verification is done before using simulation (for software) and emulation (for hardware) to verify the functionality of the design.  **Mathematical Model:** Formal verification relies on mathematical models to represent the system's design and behavior, using mathematical logic to prove that the hardware design satisfies the specified properties.  **Formal Specification:** Designer formulates a formal specification, including properties that the hardware design must satisfy. These properties can relate to correctness, safety, liveness, or other critical aspects of the system.  **Model Checking:** It visits all possible values of all registers/inputs (DUT states) as it walks through stimulus. It may reach states that hit properties, explore all reachable states, or keep exploring new states until timeout. |

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

And list the pros and cons for each.

|  |
| --- |
| **Formal verification:**  It tests all possible stimulus, one cycle at the time. It is used to test and try to visit all the states in every cycle.  **Pros:** 1. Systematic method  -None or very little randomization  -More deterministic  2. Less testbench effort required  -Formal testbench tends to be much simpler than sim testbench  3. Leads to higher quality  -Find bugs from a different angle: find with breadth-first search (BFS)  -Often reveals bugs that simulation would never catch  4. Improve productivity and schedule  -Can replace some block-level testbenches  -Verification can begin prior to testbench creation and simulation  **Cons:** If the design is too complicated or large, it may take a long time to test. Thus, it is usually used in smaller designs.  **Pattern based verification:**  Only input some random values into the system when the system is able to get data from outside. It always only test and visit a state every time. That behavior is similar to do a depth-first search (DFS) in the testing system.  **Pros:** 1. Directly test the functionality of the design  2. The testing input may be more meaningful than formal verification.  **Cons:** 1. Usually use randomization to create testing input, and that may cause some state  cannot be visited and tested during verification. (Undetermined states)  2. Complicated Testbench  3. Need to implement the whole algorithm of the target function precisely. Not just describe some I/O behavior. |

1. What is glue logic?

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

|  |
| --- |
| While modeling complex behaviors, SVA (System Verilog Assertion) may be complicated. Thus, we can use some “auxiliary logic” to observe and track events. That’s called glue logic. After using this trick, we can greatly simplify codes, and expressing SVA properties may be trivial. Also, glue logic comes at no extra price and JasperGold does not care whether property is all SVA or SVA + glue logic. Therefore, it is recommended to choose it based on clarity. |

1. What is the difference between **Functional coverage** and **Code coverage**?

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

|  |
| --- |
| **Functional coverage:**  It is used to test the behavior of the design. The targets are specific states, conditions, or sequences. It is property related covers, which needs to realize by properties written by designers. Coverage information is more meaningful and realizable, but it is time consuming and may not cover corner case.  **Code coverage:**  It is used to test the code can be executed(visit) when the system is working. The targets are the branch, statement, and expression in the code. Cover items are automatically generated by EDA tools from RTL code and can be fully covered. However, it may cause some false alarm issue.  100% code coverage indicates that every single line of the codes can be visited. We cannot claim that our assertion is well enough for verification because it may not capture all meaningful design functionality. On the other words, the functionality may exist some mistakes. |

1. 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:**  **Meaning:** COI coverage is Cone-Of-Influence Coverage. With COI coverage, we can find the union of all assertion COIs. The remaining out of COI cover items indicate holes in the assertion set. These holes stand for some code that is not checked by any asserts.  **Relationship:** COI represents the maximum potential of proof coverage.  **Tool effort perspective:** It is a fast measurement because no formal engines are run. COI doesn’t require a proof to take place.  **Proof Coverage:**  **Meaning:** It represents the portion of the design verified any formal engines. It will find the region cannot truly influence assertion status.  **Relationship:** It is a subset of the COI.  **Tool effort perspective:** It requires a proof, needs greater tool effort and runs slower than COI. |

1. What are the roles of **ABVIP** and **scoreboard** separately?

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

|  |
| --- |
| **ABVIP:**  **Definition:** The Assertion Based Verification Intellectual Properties are a comprehensive set of checkers and RTL that check for protocol compliance.  **Objective:** To verify a protocol and analyze its completeness.  **Benefit:** It can help the designer to verify new system within some protocol easier and more quickly.  **Scoreboard:**  **Definition:** Scoreboard behaves like a monitor to observe input data and output data of DUV.  **Objective:** To check (1) Data packet dropped (2) Duplicated Data Packets  (3) Order of Data Packets (4) Corrupted Data Packets  **Benefit:** It is formal optimized to reduce state-space complexity and reduce barrier for adoption. |

1. List four **bugs** in Lab Exercise

What is the answer of the Lab Exercise?

|  |
| --- |
| **Bug#1:** inf.AR\_VALID/inf.AR\_ADDR should be stable if(inf.AR\_VALID && !inf.AR\_READY)      Solution: line90, if(inf.AR\_READY) 🡪 if(n\_state == AXI\_AR)  **Bug#2:** inf.AW\_VALID/inf.AW\_ADDR should be stable if(inf.AW\_VALID && !inf.AW\_READY)      Solution: line124, if(inf.AW\_READY) 🡪 if(n\_state == AXI\_AW)  **Bug#3:** Data integrity – inf.AW\_ADDR translation is failed.      Solution: line134, inf.AW\_ADDR <= {**8'h1000\_0000**, inf.C\_addr, 2'b0} 🡪 inf.AW\_ADDR <= {**1'b1, 7'b0**, inf.C\_addr, 2'b0}  **Bug#4:** inf.W\_DATA is not equal to temp\_w when (inf.W\_VALID && inf.W\_READY)      Solution: line145, if(inf.C\_in\_valid && **inf.C\_r\_wb**) 🡪 if(inf.C\_in\_valid && **!inf.C\_r\_wb**) |

1. 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.

|  |
| --- |
| In Lab07, we used JasperGold to ensure that our design with crossing domain modules does not have any convergence or functional issues. Since CDC designs are difficult to find cross domain issues in finite pattern based verification, JG is very useful to find our blind spot effectively before synthesizing, which saves debugging time and avoid unexpected errors. The “tracing why” function and showing relevant waveforms/codes makes it easier to find our problems.  Most challenge I encountered is about the convergence issues. For example, since clk1 input will be send only after clk3 finish outputting all values in pattern, I thought that it is unnecessary to create a flag from handshake synchronizer to tell clk1 module whether it can receive next input initially. But in JG, it checked all possibilities, so it pasted violation message, and I have to modify my design to make it more reliable.  Another feedback is that sometimes JG tells many violation messages at the same time, but they are caused by the same reason. Therefore, I have to distinguish which issue I should solve in priority by myself.  Overall, JG is a strong tool to help designers finding possible problems. After being familiar to its GUI, it must be very convenient to find problems. |