IC Lab Formal Verification
Lab10 Quick Test
2024 Spring

Name : 王語 **Student ID:** 312605003 **Account:** iclab065

(a) What is Formal verification?

Hardware designs are complex. Traditional simulation techniques can't guarantee catching all errors, especially in intricate designs. Formal verification provides a more rigorous approach.

It is a way to mathematically prove that the hardware design meets its intended purpose. It's like an extremely thorough inspection that can identify any bugs or errors during the front-end design stage.

Formal verification employs an exhaustive approach to test all possible input stimuli:

1. At each cycle, it tests all combinations of values for input ports and undriven wires.

2. In the first cycle, it tests all combinations of values for uninitialized registers.

3. Finally, it outputs a report summarizing violated assertion properties and hit cover properties.

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

Formal verification and pattern based verification are two different verification methods, each with its own advantages and disadvantages, and different attributes of the circuit have their own suitable verification methods.

Unlike Formal verification, pattern based verification is used to check for errors by simulating the results of a large amount of test data.

In general, circuit behavior that requires adherence to protocols is suitable for Formal verification, while circuits that involve large amounts of computation are suitable for pattern based verification.

And list the pros and cons for each.

Formal pros :

Formal verification uses mathematical methods to prove that all potential states are traversed in all possible input scenarios, thus providing a higher degree of certainty.

Formal cons :

Formal verification is time-consuming, and sometimes it is not possible to go through all the possibilities in a limited amount of time.

Pattern pros :

Pattern verification is an efficient way to verify the functionality of a circuit, especially one with an end-to-end nature.

By generating a large amount of test data and golden answers, the correctness of the design can be initially confirmed.

Pattern cons :

Pattern verification design depends on the designer's meticulousness, and the drawback is very obvious, that is, the corner case may be missed, resulting in the inability to achieve bug-free, such as in the first few labs, design inputs are random generation, it may occur in their own test circuit function is normal, but the assistant teacher demo will be a bug situation. This can happen when you test the circuit yourself and everything works fine, but when the TA is demoing the circuit, there is an error. Speaking of which, I'm in tears and snot, with a cigarette in my left hand and a drink in my right, and my scores are gone forever. Iclab makes me self-doubt.

Supplementary:

The two verification methods can be used interchangeably to achieve better results.

(b) What is glue logic?

It means using auxiliary logic to observe and track events.

For some validation scenarios, the logic written inside each assert property is quite complex if it is written in the normal SVA way. If you use glue logic, you can simplify the logic or write the assert property in a more concise way.

In most cases, this auxiliary logic is not synthesized by the compiler, so it is not a burden to the original design.

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

Glue logic simplifies complex signal relationships into a new logic, so what would otherwise be required to write SVA expressions for each signal can be achieved with glue logic's SVA expressions, simplifying the complexity of SVAs, improving readability.

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

Functional coverage is developed by the designer based on the circuit specification, according to the corresponding functional description of the covergroup. Every requirement of functional coverage is meaningful, but it takes a lot of time to generate manually and can be inadvertent.

Code Coverage is the process of breaking down the code into branches, statements, and expressions, and checking to see if each of these has been executed. Code Coverage can be generated automatically by software.

The two coverages are defined by different facets, each with its own advantages and disadvantages.

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

No.

100% code coverage only means that all parts of the code have been executed, but it lacks functional validation and testing, and cannot guarantee the correctness and coverage of the functionality, so only 100% code coverage is not enough for validation.

(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 verifies the combination of cover items for the logic examined by the assertion by tracing back all the variables affecting the logic. on the other hand, proof coverage transforms the circuit into a simplified mathematical logic model, verifying only the cover items directly affecting the logic examined by a formal engine that verifies all possible cases for each cycle. The formal engine verifies all possible scenarios for each cycle. Usually proof coverage is one of the branches of COI.

Relationship

Checker coverage is a combination of COI coverage and Proof coverage, and is mainly used to compensate for Stimuli Coverage's inability to ensure 100% correctness of a design.

COI coverage is related to simulation verification, where COI is checked against test cases, and Proof coverage is related to formal verification, where all possible cases are verified against a formal engine.

Tool effort

Because running Proof Coverage involves formal proof, it requires more and longer runtime and resources than COI coverage.

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

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

Definition

ABVIP is a pre-written verification IP with detailed test and check assertions, usually used to verify a specific function or protocol of a design, and is a high-reliability verification tool that improves verification quality and reduces development time.

Scoreboard is a component of the verification environment that acts like a monitor, comparing the actual output of a design with the expected output and reporting error information back to the designer to ensure the correct functionality of the design.

Objective

The goal of ABVIP is to provide a well-developed tool that accelerates the testing and verification process, while the goal of Scoreboard is to check that the design is functionally correct.

Benefit

For well-defined and standardized functions or protocols, ABVIP eliminates the need for users to develop additional validation programs, and the results are more rigorous and reliable.

Scoreboard automatically checks for errors and reduces manual errors. It ensures that circuits function properly in all possible situations.

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

The following errors have not been detected by JasperGold.

✖	Assert	top.slave.genStableChks.genStableChks...	N	3	1	0.0	<embedded>		Analysis Session
✖	Assert	top.slave.genStableChks.genStableChks...	N	3	1	0.0	<embedded>		Analysis Session
✖	Assert	top.slave.genStableChks.genStableChks...	N	3	1	0.0	<embedded>		Analysis Session
✖	Assert	top.slave.genStableChks.genStableChks...	N	3	1	0.0	<embedded>		Analysis Session
✖	Assert	top.slave.genPropChksRDInf.genNoRdTbl...	N	3	1	0.0	<embedded>		Analysis Session
✖	Assert	top.slave.genPropChksWRInf.genNoWrTbl...	Hp	3	1	0.2	<embedded>		Analysis Session

Here are the four bugs in the program and the corresponding fixes.

Original:

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

Fixed :

```
always_ff@(posedge clk or negedge inf.rst_n) begin
    if(!inf.rst_n)begin
        inf.AR_VALID <= 'b0;
    end
    else begin
        if(n_state == AXI_AR)  inf.AR_VALID <=  1'b1;
        else                   inf.AR_VALID <=  1'b0;
    end
end
```

Reason : AR_VALID follows the change of the state machine to ensure that it goes back to 0 at the same time as AR_READY.

Original:

```
always_ff@(posedge clk or negedge inf.rst_n) begin
    if(!inf.rst_n)begin
        inf.AW_VALID <= 'b0;
    end
    else begin
        if(inf.AW_READY)    inf.AW_VALID <=  1'b1;
        else                inf.AW_VALID <=  1'b0;
    end
end
```

Fixed :

```
always_ff@(posedge clk or negedge inf.rst_n) begin
    if(!inf.rst_n)begin
        inf.AW_VALID <= 'b0;
    end
    else begin
        if(n_state == AXI_AW)   inf.AW_VALID <=  1'b1;
        else                    inf.AW_VALID <=  1'b0;
    end
end
```

Reason : AW_VALID follows the change of the state machine to ensure that it goes back to 0 at the same time as AW_READY.

Original:

```
always_ff@(posedge clk or negedge inf.rst_n) begin
    if(!inf.rst_n)begin
        inf.AW_ADDR <= 'b0;
    end
    else begin
        if(n_state == AXI_AW && c_state != AXI_AW)   inf.AW_ADDR <= {8'h1000_0000, inf.C_addr, 2'b0};
        else                                        inf.AW_ADDR <= inf.AW_ADDR ;
    end
end
```

Fixed :

```
always_ff@(posedge clk or negedge inf.rst_n) begin
    if(!inf.rst_n)begin
        inf.AW_ADDR <= 'b0;
    end
    else begin
        if(n_state == AXI_AW && c_state != AXI_AW)   inf.AW_ADDR <= {1'b1, 7'b0, inf.C_addr, 2'b0};
        else                                        inf.AW_ADDR <= inf.AW_ADDR ;
    end
end
```

Reason : 8'h1000_0000 is equal to 8'b0000_0000

Original:

```
always_ff@(posedge clk or negedge inf.rst_n) begin
    if(!inf.rst_n)begin
        inf.W_DATA  <= 'b0;
    end
    else begin
        if(inf.C_in_valid && inf.C_r_wb)       inf.W_DATA  <= inf.C_data_w;
        else                                   inf.W_DATA  <= inf.W_DATA  ;
    end
end
```

Fixed :

```
always_ff@(posedge clk or negedge inf.rst_n) begin
    if(!inf.rst_n)begin
        inf.W_DATA  <= 'b0;
    end
    else begin
        if(inf.C_in_valid && !inf.C_r_wb)       inf.W_DATA  <= inf.C_data_w;
        else                                   inf.W_DATA  <= inf.W_DATA  ;
    end
end
```

Reason : inf.C_r_wb is 0 for "write". While Inf.C_r_wb is 1 for "read".

(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.

A good Digital design Engineer is expected to design bug-free circuits with good PPA.

When it comes up with Accurately described protocols, formal verification provides a different approach to verification.

It is no need to write additional testbench or checker for verification, saving a lot of time and making the verification results more rigorous and reliable.

Most importantly, it can verify bugs that cannot be detected by millions of test data.

As long as I can guarantee that I pass the 1st demo of the tool is the best tool!