IC Lab Formal Verification
Lab11 Quick Test
2024 Spring

Name: 林柏佑 **Student ID:** 109651066 **Account:** iclab049

(a) What is Formal verification?

Formal verification is a way to use mathematical methods to verify the correctness of logic without timing checks, and it will test all possible input stimuli. In every cycle, it will test all value combinations on inputs and undriven wires. At first cycle, it will test all value combinations on uninitialized registers. Finally, it will output the violated assertion properties and hit cover properties into a report. Due to no need for stimulus, it can be verified after the RTL is written. Problems also can be discovered in the earliest design flow and repaired more efficiently.

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

Formal uses mathematical methods to verify all possible paths under a given constraint. It is often used in protocols or smaller designs. While pattern verify the certain path given by designer, and only test one input combination (the internal signal combination is also one type) at every time.

And list the pros and cons for each.

Formal verification

Pros：

It can cover all common or corner cases. As a result, there is a higher chance to detect the problem of circuits. Basically, the design pass formal verification can be said to be 100% correct.

Cons：

Due to verifying all possible paths, if the system is too complicated, the verification time will increase exponentially. Besides, it may also consume lots of resources, and it is easy to cover the path that might never occur.

Pattern verification

Pros：

Due to only checking the specific behavior, it consumes less resources and time to quickly improve the coverage of cases.

Cons：

If the user does not consider the complete situation, it is easy to cause incomplete coverage and difficult to detect the corner case. For example, in the first few labs, the input of the design was generated randomly. It may happen that the function is normal when we test the circuit by ourself, but it will get error when testing by TA.

(b) What is glue logic?

Glue logic is the additional logic used to reduce the assertion complexity in SVA. By using these additional logic, it can make the representation of SVA simpler and easier to understand.

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

As mentioned above, if you simply use SVA, when you encounter complex asertion, you need to write a long and complex logical expression. But if you use Glue Logic, you can paraphrase some reusable behavior descriptions with a variable, which can also make the behavior description in the assertion more concise.

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

Functional coverage：

It need designer to manual design the coverage points. Thus, these coverage points are usually more meaningful, but it is also easy to miss some corner cases, and more time-comsuming.

Code coverage：

EDA tool will automatically generate cover items based on the given RTL code. It will directly check whether each line of code in the design is executed correctly. Thus, there is no need for designer to write the file by himself.

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

No. Code verification only checks whether each line of code has been executed, it cannot guarantee functional correctness. Therefore, 100% code coverage can only mean that every line of code in deisgn has been executed. As for whether the function of the design is correct, it still need the help of functional or checker 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.

Meaning：

COI coverage is the region that all the signals in the code that will affect the assertion. Proof coverage is the region that only the signals included in the assertion. The combination of COI coverage and Proof coverage is checker

coverage, which is mainly used to make up for the inability of Stimuli Coverage to ensure correctness of the design.

Relationship：

Proof coverage is a subset of COI coverage.

Tool effort：

Because running Proof Coverage requires formal verification, it requires more time 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：

The Assertion Based Verification Intellectual Properties (ABVIPs) are a comprehensive set of checkers and RTL that check for protocol compliance.

Scoreboard：

Scoreboard behaves like a monitor used to observe input data and output data of DUV.

Objective

ABVIP：

Provide a well-developed tool for designers to check those existed protocol.

Scoreboard：

Check the consistency of desgin’s input and output at high level.

Benefit

ABVIP：

Let designer verify new design more quickly and easily.

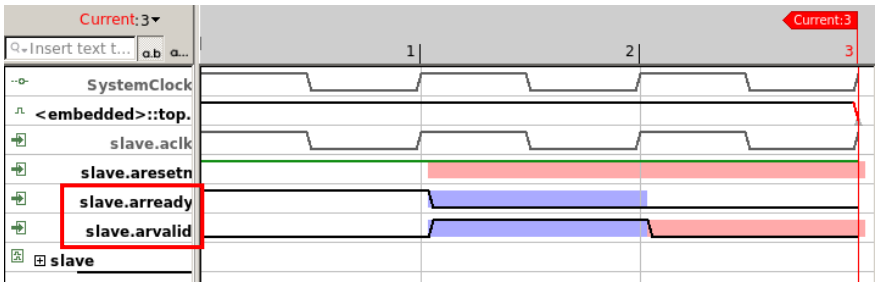
Scoreboard：

Formal optimized to reduce the state-space complexity and also reduce barrier for adoption.

(f) List four **bugs** in Lab Exercise

What is the answer of the Lab Exercise?

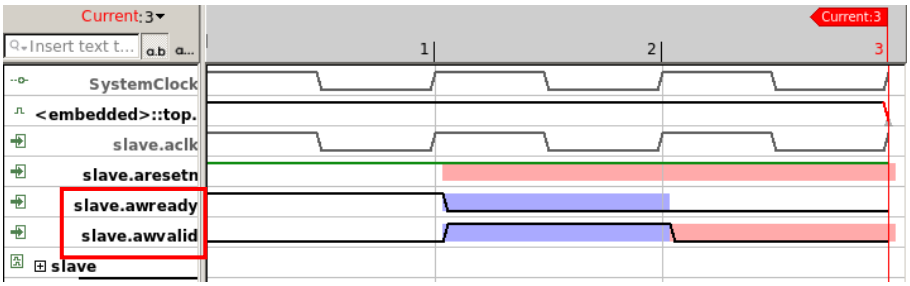
1. inf.arvalid & inf.arready should be 0 when inf.arvalid become 0



```
// * Bug1
always_ff @(posedge clk or negedge inf.rst_n)
begin
    if(!inf.rst_n) inf.AR_VALID <= 'b0;
    else
    begin
        if(inf.AR_READY) inf.AR_VALID <= 1'b1;
        else             inf.AR_VALID <= 1'b0;
    end
end
```

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

2. inf.awvalid & inf.awready should be 0 when inf.awvalid become 0




```
// * Bug2
always_ff @(posedge clk or negedge inf.rst_n)
begin
    if(!inf.rst_n) inf.AW_VALID <= 'b0;
    else
    begin
        if(inf.AW_READY) inf.AW_VALID <= 1'b1;
        else             inf.AW_VALID <= 1'b0;
    end
end
```

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

3. When C_r_wb is 1, it is read mode; 0 is write mode, so the following writing method will cause W_DATA to never get C_data_w, and can keep writing 0 into it.

```
// * Bug3
always_ff @(posedge clk or negedge inf.rst_n)
begin
    if(!inf.rst_n) inf.W_DATA <= 'b0;
    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

// * Correct
always_ff @(posedge clk or negedge inf.rst_n)
begin
    if(!inf.rst_n) inf.W_DATA <= 'b0;
    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
```

4. The highest bit of AW_ADDR is 8. Obviously, 8'h1000_0000 exceeds this limit. The synthesizer will report a warning, and the actual value considered will be the smallest 8 bits of 8'h1000_0000, resulting in 8'h00. This mismatch leads to W_DATA being written to an incorrect address. As a result, we need to change 8'h1000_0000 to 8'b1000_0000.

```
// * Bug4
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

// * Correct
always_ff @(posedge clk or negedge inf.rst_n)
begin
    if(!inf.rst_n) inf.AW_ADDR <= 'b0;
    else
    begin
        if(n_state == AXI_AW && c_state != AXI_AW) inf.AW_ADDR <= {8'b1000_0000, inf.C_addr, 2'b0};
        else                                       inf.AW_ADDR <= inf.AW_ADDR ;
    end
end
```

(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.

From the Jasper CDC used in Lab07, and the IMC Coverage used in Lab9. These tools indeed play a crucial role in completing the design. But if I had to choose the most effective JasperGold tools for me, IMC Coverage would be my top choice.

With the IMC Coverage Tool, I can clearly know how many percent of the coverage I have obtained so far. Because I only use the ./02_detail given by the TA, and the displayed bin will not be displayed if it is not 100%. But sometimes it may only be 1 or 2 more times to get 100%, and you need to use the display method to calculate how many hits you have, but if you use IMC Coverage to check, the whole process will become quite faster.

Next, let's talk about the challenges I encountered. Surprisingly, using this tool will not encounter the convergence issue that had encountered in CDC. The only problem is that IMC Coverage can only display the number of times a bin has been hit, and cannot detail the input given by each pattern and the transition in between. Thus, I still need to use dislpay function to debug. But overall, this tool is really useful for designing 100% coverage patterns. As a result, it impressed me deeply.