IC Lab Formal Verification
Lab11 Quick Test
2024 Spring

Name: 蔡宗儒 **Student ID:** 109511207 **Account:** iclab059

(a) What is Formal verification?

A: Formal verification uses mathematical or logical methods to verify the correctness of a system. It employs brute force to test all inputs and undriven wires at each cycle, checking for violations of specifications without the need for timing checks

What's the difference between **Formal** and **Pattern** based verification? And list the pros and cons for each.

A: Formal verification uses mathematical methods to trace all possible paths under given constraints for validation(as breadth-first search), while pattern based verification(as depth-first search) employs a random approach, validating only specific paths each time.

Formal verification:

Pros:

- Being able to list out all combinations within the circuit results in higher coverage.
- More deterministic.
- Detecting circuit issues in early stage.
- Often reveals bugs that simulation would not catch.
- No need to write testbench.
- Assertions and covers are reusable.

Cons:

- Verification time increases exponentially with design complexity.
- Easier to cover paths that will never happen.

Pattern base verification:

Pros:

- The verification time is shorter.

Cons:

- Lower coverage due to unconsidered cases.
- Hard to detect corner cases.

(b) What is glue logic?

A: Glue logic is the auxiliary logic added to simplify complex logic or assertion during verification.

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

A: Because glue logic can simplify verbose and lengthy logic expressions, and usually would not be synthesized. It can make the code cleaner and enhance the readability.

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

A: Functional coverage examines the functionality of the design. It ensures that design operates correctly under various conditions. Designers need to create coverpoints themselves, which can be time-consuming and prone to oversight, leading to situations where corner cases are not adequately considered.

Code coverage checks whether each line of the RTL code in the design has been executed. It is automatically generated by EDA tools and is able to prevent human errors. However, it does not guarantee functional correctness and may also encounter false alarm issues.

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

A: 100% code coverage means that every line of code in the design has been executed. However, it does not verify whether the design meets specifications or whether its functionality is correct. It is possible that we may have forgotten to include certain functionalities in the RTL code from the beginning, which would require Functional coverage to verify.

(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.

A:

Meaning:

COI coverage: COI stands for Cone-Of-Influence, which refers to the region encompassing all cover items that influence an assertion.

Proof coverage: Proof coverage represents the actual region that the assertion will check.

Relationship:

Proof coverage is a subset of COI coverage.

Tool effort:

COI coverage: COI coverage only requires continuous back tracing to identify all cover items, without the need for a formal engine. Therefore, it saves time and resources, with lower tool effort.

Proof coverage: Proof coverage requires formal proof to find the actual region checked by the assertion, leading to more time and resource consumption, with higher tool effort.

(e) What are the roles of **ABVIP** and **scoreboard** separately? Try to explain the definition, objective, and the benefit.

A:

Definition:

AVBIP: short for Assertion Based Verification Intellectual Properties, refers to IPs composed of assertions, typically employed to verify protocols.

Scoreboard: Scoreboard is used to compare the output of DUV with expected or reference results. It is commonly used in simulation environments to validate the functionality of a DUV.

Objective:

AVBIP: For a complex protocol, writing a significant number of assertions for verification may be necessary. Moreover, for the same protocol, the same assertions can be reused. Therefore, the objective of AVBIP is to rapidly provide verification assertions and accelerate the entire verification process.

Scoreboard: Scoreboard behaves like a monitor, observing input data and output data of DUV

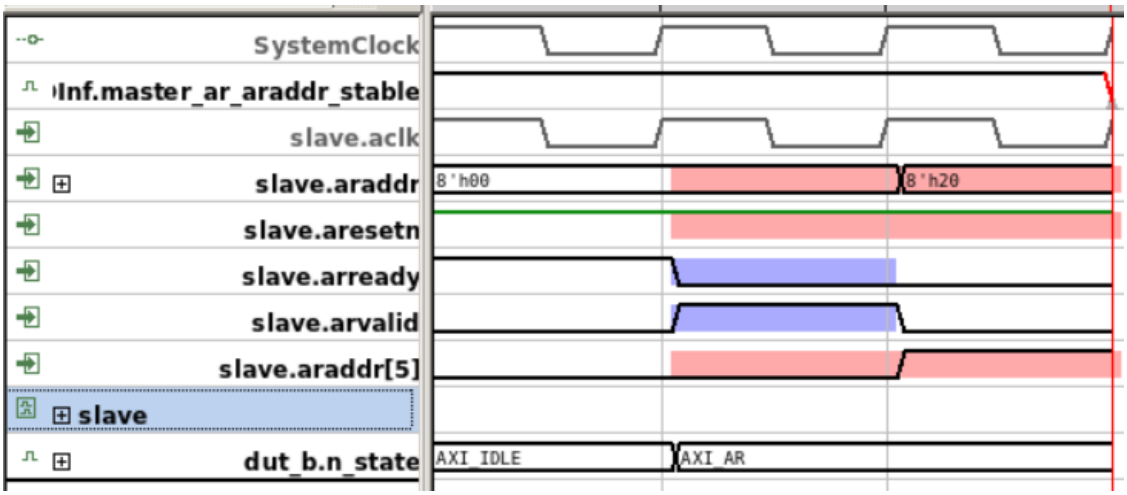
Benefit:

AVBIP: AVBIP eliminates the need for verification engineers to write a significant number of assertions themselves. In addition to accelerating the entire verification process, it significantly reduces the errors that may occur when writing numerous assertions manually. This ensures that the verification process is more rigorous and reliable.

Scoreboard: Formal optimized to reduce state-space complexity, which means to reduce barrier for adoption.

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

Bug1:



The original approach might fail to perform a handshake. The correct approach would be to modify the condition to be judged by an FSM, specifically (n_state == AXI_AR).

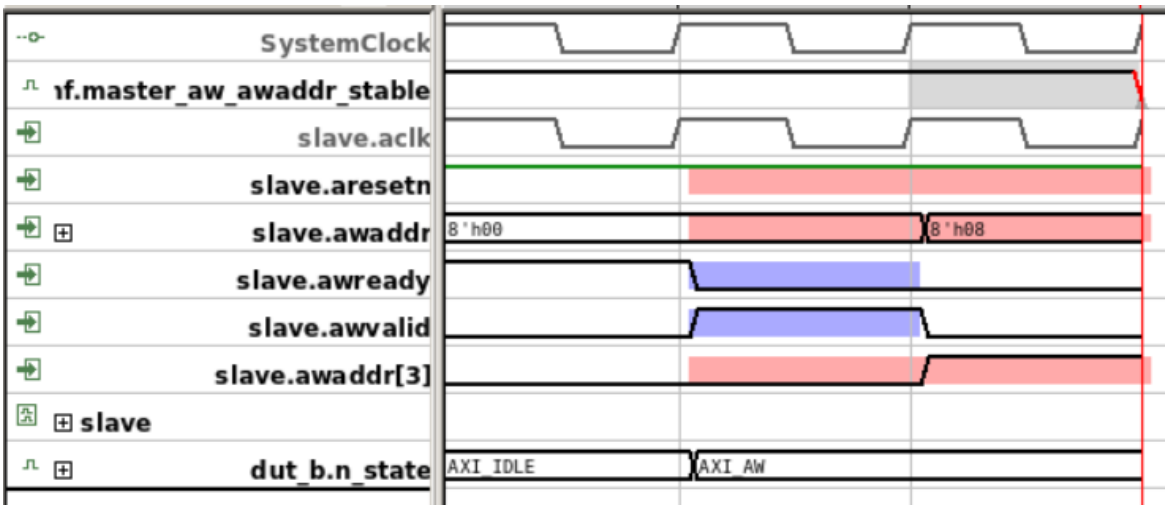
Error:

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

Bug2:



The original approach might fail to perform a handshake. The correct approach would be to modify the condition to be judged by an FSM, specifically (n_state == AXI_AW).

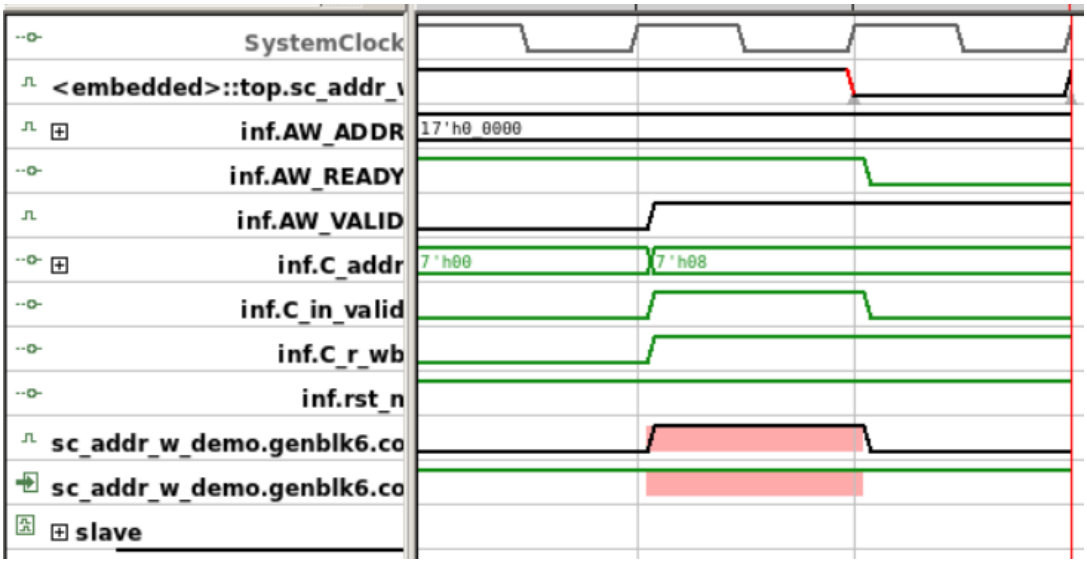
Error:

```
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
```

Correct:

```
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
```

Bug3:



The original approach would cause the leftmost 8 bits of AW_ADDR to be set to 8'h00, resulting in the written address being {8'h00, inf.C_addr, 2'b0}, which is incorrect.

It should be changed to inf.AW_ADDR <= {1'b0, 7'b0, inf.C_addr, 2'b0}.

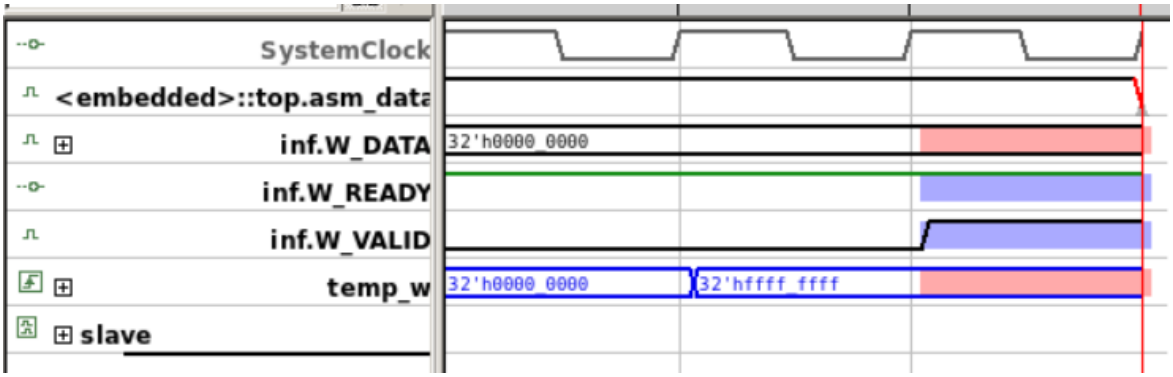
Error:

```
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)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
```

Bug4:



When C_r_wb is high, it is in read mode; if it is low, it is in write mode. Therefore, the original approach would incorrectly write C_data_w into AW_ADDR during a read operation for W_DATA. Hence, the condition should be changed to (inf.C_in_valid && !inf.C_r_wb).

Error:

```
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
```

Correct:

```
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
```

(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.

Among these four, I personally find IMC coverage the most user-friendly and effective. This is because it provides clear visibility into the current coverage percentage and allows direct examination of how many times each bin has been hit. During Lab09, I discovered that latency was higher in a particular case. So, I adjusted the number of this case to reduce simulation time and achieve better performance. I utilized IMC coverage to minimize the number of hits for this case.

However, I encountered a challenge regarding transitions due to insufficient consideration. This resulted in one transition bin being hit one time less than the SPEC. Additionally, since Coverage_Detail.log does not display if a bin isn't hit 100%, I initially assumed that one transition bin wasn't hit at all. It was only after opening IMC Coverage that I realized it was hit one less time, indicating a mistake in my value assignment.