# IC Lab Formal Verification Bonus Quick Test 2023 Spring

**Name:** 顏子芸 **Student ID:** 0812005 **Account:** iclab100

1. Bonus:

(a) What is Formal verification?

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

And list the pros and cons for each.

- **Formal verification** is a technique that uses mathematical methods to test all possible values of inputs and registers. We can use formal checks properties such as assert, cover, and assume to specify the signal we want to verify.
- **Formal verification**, like WFS, aims to visit all reachable states until the desired property is hit.
  - pros – (1) guarantee 100% coverage. (2) enable systematic verification of consistency between the specification and inputs.
  - cons – (1) complex and time-consuming. (2) have scalability challenges for large systems since the number of states to be verified increases exponentially with the number of bits of states.
- **Pattern based verification**, like DFS, is primarily focus on verifying the correctness of system outputs. However, it may miss or cover multiple reachable states, leading to incomplete coverage.
  - pros – (1) simple. (2) can quickly generate patterns to identify common errors.
  - cons – (1) hard to exhaustively simulate all possible states.

(b) What is glue logic?

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

- **Glue logic** refers to auxiliary logic used for observing and tracking events. It can be utilized to simplify the SVA expression.
- Once glue logic is in place, SVA expression can be simplified, making the code easier to read and understand.

(c) 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** is used to verify if the functionalities of a design have been executed. Users can verify specific functionalities by using **coverpoint**s.

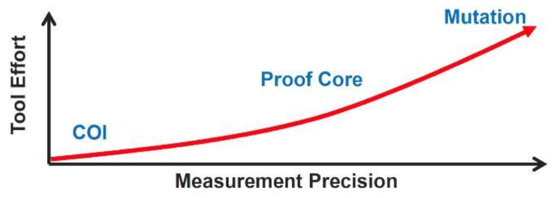
  **Code coverage** is used to verify the completeness of code execution, including statement(line) coverage, branch coverage, FSM coverage, and so on. This coverage is collected by the simulation tools.

- No, 100% code coverage only means all statements in our code has been executed. However, it cannot guarantee the correctness of functionalities implemented by those statements.

(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.

- **COI(cone of influence) coverage** identifies which cover items can potentially affect the assertion results based on code tracing. Since there are no formal engines running, it can provide a fast measurement.

- **Proof coverage** identifies the cover items that truly make assertion correct. The cover items in proof coverage are called proof cores, which are a subset of cover items of COI. In addition, since proof coverage requires verification through running formal engines, it takes more time to measure.



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

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

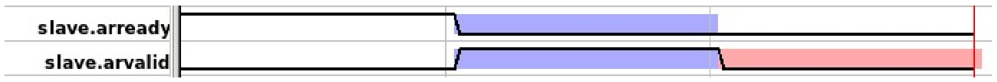
- **ABVIP**(Assertion Based Verification Intellectual Property) is a comprehensive set of checkers and RTL that check for protocol compliance (e.g. AXI4). It contains a lot of assertions that allow designers to easily verify a protocol and analyzing its completeness.

- **Scoreboard** behaves like a monitor and is responsible for verifying the similarity of input and output data from the DUV(Design Under Verification). A scoreboard can help reduce the complexity of the DUV.

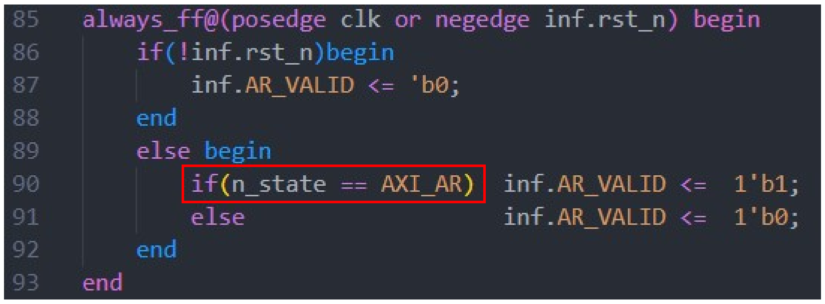
(f) List four bugs in Lab Exercise

What is the answer of the Lab Exercise?

1. Assert “master_ar_arvalid_stable” is not proven.



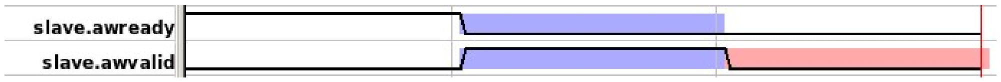
Solution:



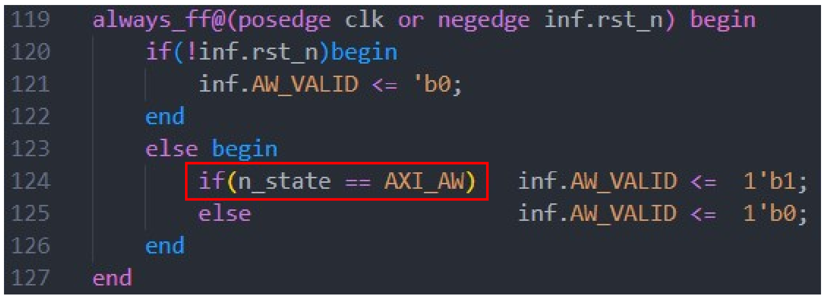
```
85  always_ff@(posedge clk or negedge inf.rst_n) begin
86      if(!inf.rst_n)begin
87          inf.AR_VALID <= 'b0;
88      end
89      else begin
90          if(n_state == AXI_AR)  inf.AR_VALID <=  1'b1;
91          else                   inf.AR_VALID <=  1'b0;
92      end
93  end
```

Modify the condition in line 90 of Bridge.sv to **n_state == AXI_AR**.
(i.e. let inf.AR_VALID remain at 1'b1 until inf.AR_VALID & inf.AR_READY == 1'b1)

2. Assert “master_aw_awvalid_stable” is not proven.



Solution:

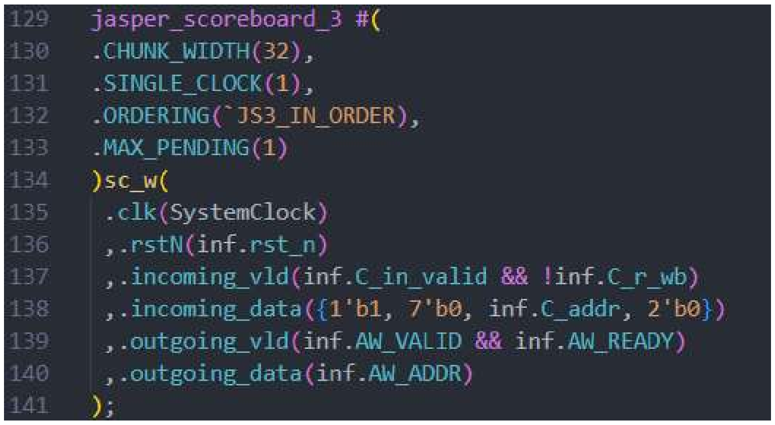


```
119  always_ff@(posedge clk or negedge inf.rst_n) begin
120      if(!inf.rst_n)begin
121          inf.AW_VALID <= 'b0;
122      end
123      else begin
124          if(n_state == AXI_AW)  inf.AW_VALID <=  1'b1;
125          else                   inf.AW_VALID <=  1'b0;
126      end
127  end
```

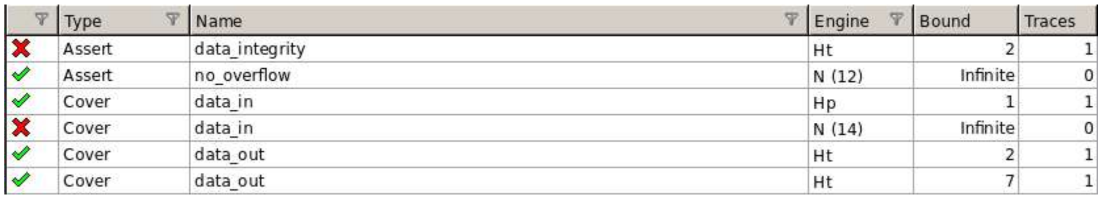
Modify the condition in line 124 of Bridge.sv to **n_state == AXI_AW**.
(i.e. let inf.AW_VALID remain at 1'b1 until inf.AW_VALID & inf.AW_READY == 1'b1)

3. After adding JasperGold Build-In ScoreBoard IP to verify the correctness of the "inf.AW_ADDR" address translation, we can see the data integrity of inf.AW_ADDR is not proven.

<u>JasperGold Build-In ScoreBoard IP (in top.sv):</u>

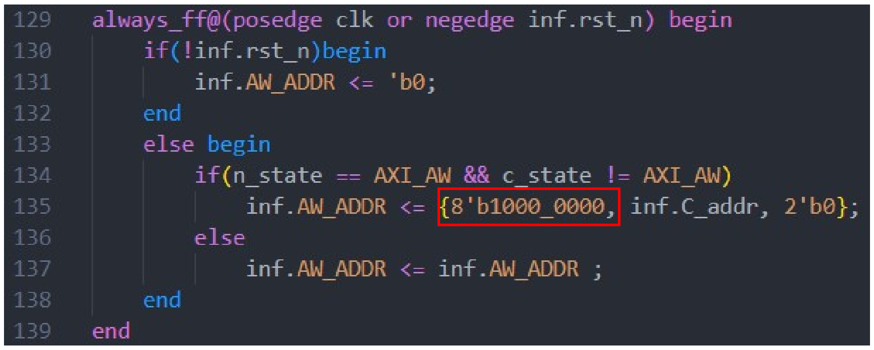


```
jasper_scoreboard_3 #(
.CHUNK_WIDTH(32),
.SINGLE_CLOCK(1),
.ORDERING(`JS3_IN_ORDER),
.MAX_PENDING(1)
)sc_w(
 .clk(SystemClock)
 ,.rstN(inf.rst_n)
 ,.incoming_vld(inf.C_in_valid && !inf.C_r_wb)
 ,.incoming_data({1'b1, 7'b0, inf.C_addr, 2'b0})
 ,.outgoing_vld(inf.AW_VALID && inf.AW_READY)
 ,.outgoing_data(inf.AW_ADDR)
);
```



| | Type | Name | Engine | Bound | Traces |
|---|---|---|---|---|---|
| ✗ | Assert | data_integrity | Ht | 2 | 1 |
| ✓ | Assert | no_overflow | N (12) | Infinite | 0 |
| ✓ | Cover | data_in | Hp | 1 | 1 |
| ✗ | Cover | data_in | N (14) | Infinite | 0 |
| ✓ | Cover | data_out | Ht | 2 | 1 |
| ✓ | Cover | data_out | Ht | 7 | 1 |

<u>Solution:</u>

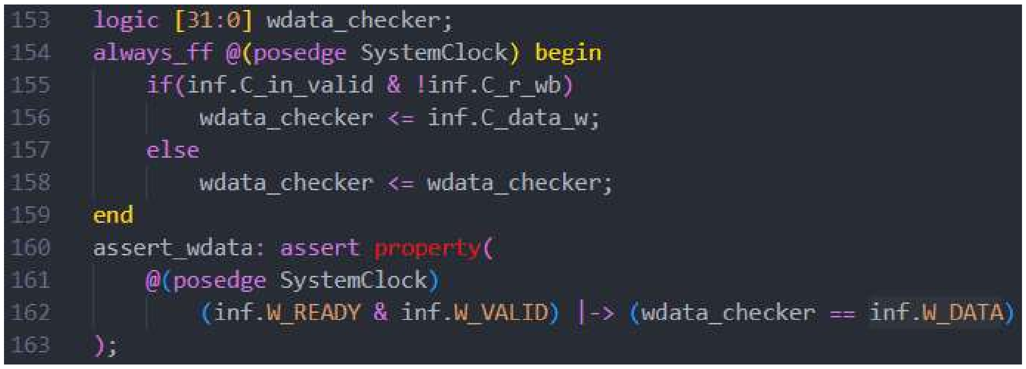


```
always_ff@(posedge clk or negedge inf.rst_n) begin
    if(!inf.rst_n)begin
        inf.AW_ADDR <= 'b0;
    end
    else begin
        if(n_state == AXI_AW && c_state != AXI_AW)
            inf.AW_ADDR <= {8'b1000_0000, inf.C_addr, 2'b0};
        else
            inf.AW_ADDR <= inf.AW_ADDR ;
    end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

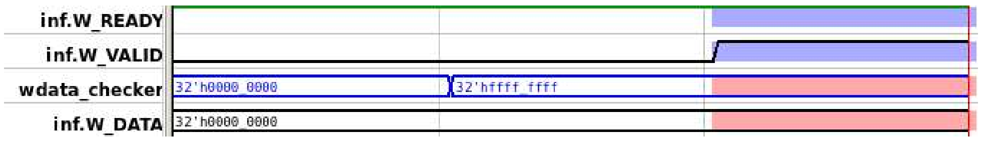
As shown at the red box in the above figure, replace the original 'h' with 'b'.

4. After adding the glue logic and assertion to verify the correctness of the “inf.W_DATA”, we can see the data integrity of inf.W_DATA is not proven.

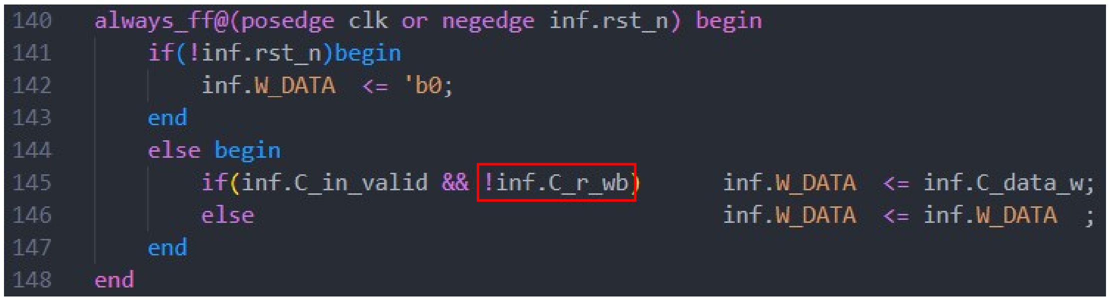
Glue logic and assertion (in top.sv):



```
logic [31:0] wdata_checker;
always_ff @(posedge SystemClock) begin
    if(inf.C_in_valid & !inf.C_r_wb)
        wdata_checker <= inf.C_data_w;
    else
        wdata_checker <= wdata_checker;
end
assert_wdata: assert property(
    @(posedge SystemClock)
        (inf.W_READY & inf.W_VALID) |-> (wdata_checker == inf.W_DATA)
);
```



Solution:



```
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
```

As shown at the red box in the above figure, add an '!' before inf.C_r_wb.