IC Lab Formal Verification
Lab11 Quick Test
2024 Spring

Name: 邱博謙 **Student ID:** 109511094 **Account:** iclab104

(a) What is Formal verification?

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

And list the pros and cons for each.

i. Formal verification is a method that uses **mathematical techniques** to prove the correctness of a design or system.

It is mainly used to compare two different representations of a design to determine whether they have the **same input-to-output functionality**. Moreover, it is used to explore the **state-space** of a system to test whether certain **properties**, typically specified in the form of assertions, are true.

ii. **Formal** based verification is based on various mathematical algorithms without any time checking(**state-based**). In this way, designers avoid spending extra time to consider verification plans, code and functionality coverages.

On the other hand, **Pattern** based verification is **time-based**, so designers have to generate stimulus, which inevitably increase work for designers.

iii. Pros & Cons:

	Formal	**Pattern**
Pros	● Higher quality. ● Independent to testbench. ● Reusable	● Focus on the scenarios we care ● Suitable for large modules. ● Used with other verification
Cons	● Don’t care cases may be cover. ● High resource consumption	● Corner case may be missed . ● More testbench effort.

(b) What is glue logic?

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

i. Glue logic is the custom logic circuitry used to interface several off-the-shelf integrated circuits. It acts as auxiliary logic to observe and track events.

ii. Because of modeling complex behaviors, SVA may be complicated. That is, it essential to simplify SVA expression. The advantage of using **glue logic** to simplify is that it comes at **no extra price**. Moreover, it is much more readable.

(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?

i. **Functional coverage** is a **user-defined** metric that measures how much of the design specification, as enumerated by features in the test plan, has been exercised.

However, **Code coverage** is auto-generated **by EDA tools**. In general, its target is to check the implementation of all statements/branches/expressions of the code.

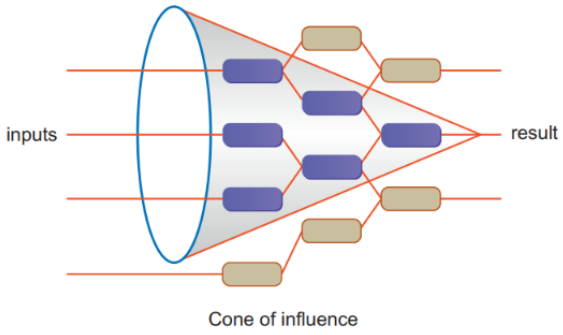
ii. 100% code coverage means that every line in the code has been executed.

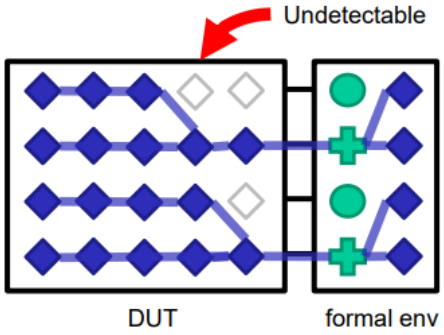
When it comes to a well constrained verification, we should ensure the environment can recreate all the scenarios in which DUT should operate. That is, there is no any false failure(100% code coverage), and all possible scenarios could be tested(100% functional coverage)

Therefore, we can't claim that our assertion is well enough for verification under the circumstance of only 100% code coverage. Because we merely guarantee the correctness of the code, but there may be some omissions of specifications, which are possible to result in critical fatal.

(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.

i. **COI(Cone-Of-Influence) coverage** is the percentage of the union of all cover items(inputs, register value) in DUT which may affect any assertion(result). In this way, we find all related signals through their connection. It is similar to find all leaf nodes in a tree. In the case, the root node is assertion, and the leaf nodes are cover items affecting assertion.





COI coverage helps us find all related cover items fast, but not all of them can truly influenced assertion status. Therefore, we adopt **proof percentage** to verify whether each item of above can truly affect assertion status, which causes more tool effort.

ii. Summary:

	COI coverage	**Proof coverage**
Meaning	The region which each assertion affected by some cover items.	The region can truly influence assertion status.
Relationship	Max potential of proof coverage. Simulation verification.	Subset of COI. Formal verification.
Tool effort	Less(no running formal engine).	More(verified by formal engine).

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

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

i. **ABVIPs(Assertion Based Verification Intellectual Properties)** are comprehensive sets of checkers and RTL that check for protocol compliance. Its objective is to make sure the design satisfies protocol specification. Just like other IPs, designers can save time on the functions offered by ABVIPs. Moreover, because IP is developed by professional verification team, it is much more reliable and accurate.

Scoreboard behaves like a monitor. It observe input data and output data of DUV, and check whether the input(initiator -> DUV) is same as the output(DUV -> target). The error messages are helpful for designer to debug.

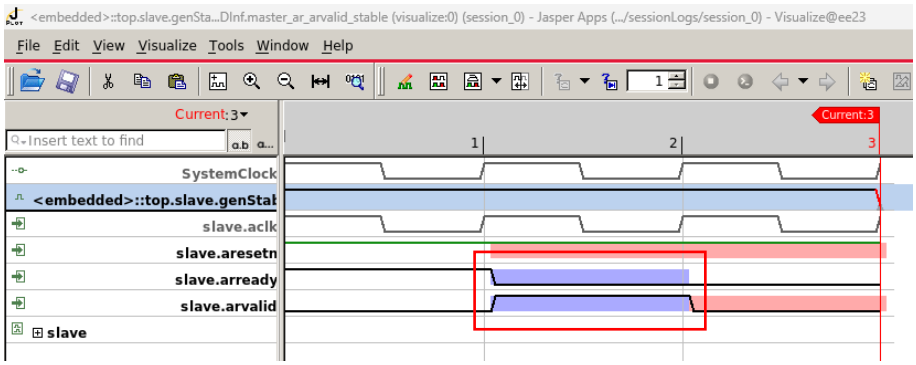
ii. Summary:

	ABVIP	**Scoreboard**
Definition	A comprehensive set of checkers and RTL that check protocol compliance.	A monitor-like verification environment
Objective	Provide solutions for verification. Speedup and verify testing process.	Check functionality of design.
Benefit	Avoid designing additional checkers. Higher reliable.	Easier to debug Automatic operation.

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

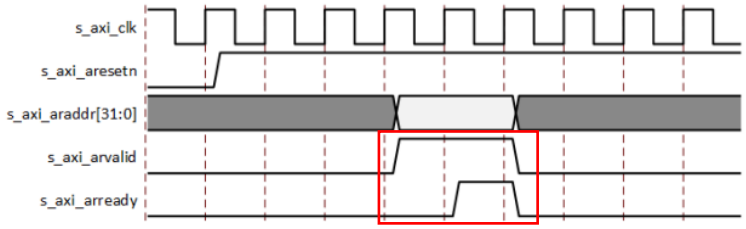
What is the answer of the Lab Exercise?

i. **Tool Report:**



Analysis:

Obviously, arready and arvalid signals don't perform a handshake(normal case is shown below). Additionally, we can observe arvalid signal is only raised 1 cycle after arready is pulled down. Therefore, I think the error is that the judgement of arvalid uses arready instead of an FSM.



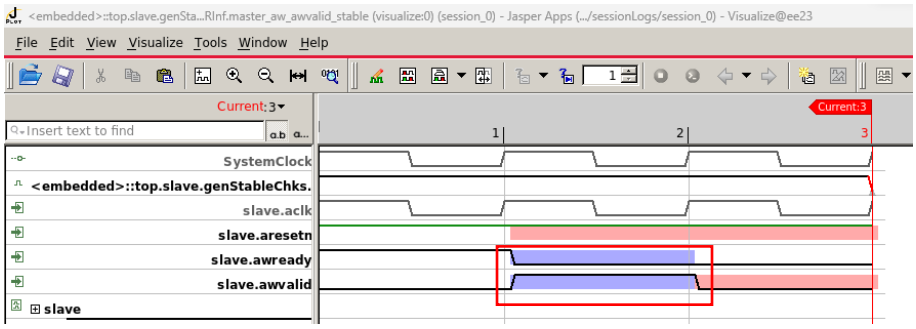
Error code:

```
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(inf.AR_READY)  inf.AR_VALID <=  1'b1;
91          else              inf.AR_VALID <=  1'b0;
92      end
93  end
```

Correct code:

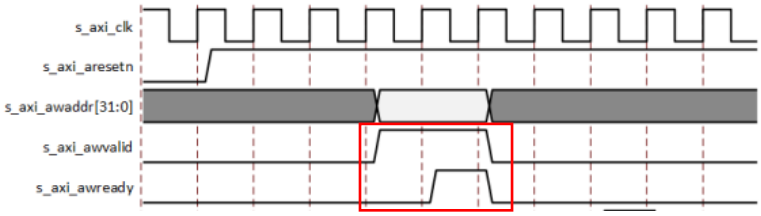
```
85  always_ff@(posedge clk or negedge inf.rst_n) begin
86      if(!inf.rst_n)begin
87          inf.AR_VALID <= 'b0;
88      end
89      else begin
90          if(n_state == AXI_AR)  inf.AR_VALID <=  1'b1;
91          else                   inf.AR_VALID <=  1'b0;
92      end
93  end
```

ii. **Tool Report:**



Analysis:

Obviously, awready and awvalid signals don't perform a handshake(normal case is shown below). Additionally, we can observe awvalid signal is only raised 1 cycle after awready is pulled down. Therefore, I think the error is that the judgement of awvalid uses awready instead of an FSM.



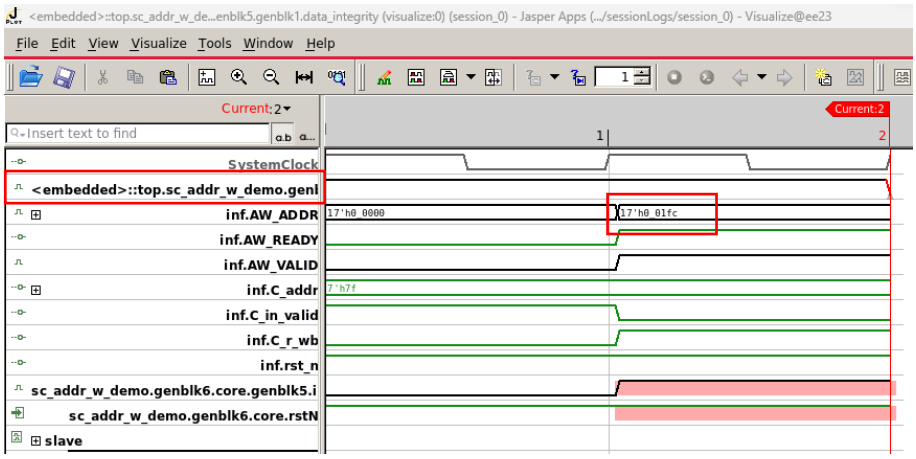
Error code:

```
129  always_ff@(posedge clk or negedge inf.rst_n) begin
130      if(!inf.rst_n)begin
131          inf.AW_VALID <= 'b0;
132      end
133      else begin
134          if(inf.AW_READY)    inf.AW_VALID <=  1'b1;
135          else                inf.AW_VALID <=  1'b0;
136      end
137  end
```

Correct code:

```
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
```

iii. **Tool Report:**



Analysis:

The waveform looks fine at first glance. Different from 2 former case, handshake has been performed. However, we can observe that value of AW_ADDR is strange with the data integrity checker of addr_w. The address of a DRAM should start from 17'h1,0000 rather than 17'h0,0000. Therefore, there must be some wrong in the assignment for AW_ADDR.

Error code:

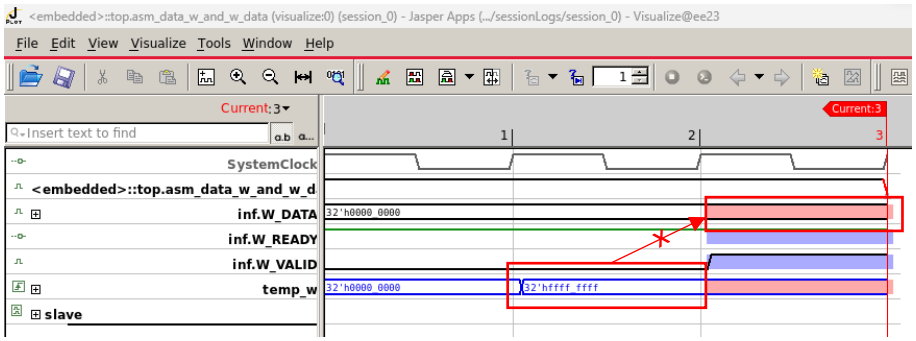
```
149 always_ff@(posedge clk or negedge inf.rst_n) begin
150     if(!inf.rst_n)begin
151         inf.AW_ADDR <= 'b0;
152     end
153     else begin
154         if(n_state == AXI_AW && c_state != AXI_AW)   inf.AW_ADDR <= {8'h1000_0000, inf.C_addr, 2'b0};
155         else                                        inf.AW_ADDR <= inf.AW_ADDR ;
156     end
157 end
```

Only these bits would be assigned

Correct code:

```
129 always_ff@(posedge clk or negedge inf.rst_n) begin
130     if(!inf.rst_n)begin
131         inf.AW_ADDR <= 'b0;
132     end
133     else begin
134         if(n_state == AXI_AW && c_state != AXI_AW)   inf.AW_ADDR <= {1'b1, 7'b0, inf.C_addr, 2'b0};
135         else                                        inf.AW_ADDR <= inf.AW_ADDR ;
136     end
137 end
```

iv. **Tool Report:**



Analysis:

When W_VALID is high, temp_w(C_data_w) isn't transferred to W_DATA. Thus, we can know the assignment of W_DATA is wrong.

Error code:

```
170 always_ff@(posedge clk or negedge inf.rst_n) begin
171     if(!inf.rst_n)begin
172         inf.W_DATA  <= 'b0;
173     end
174     else begin
175         if(inf.C_in_valid && inf.C_r_wb)      inf.W_DATA  <= inf.C_data_w;
176         else                read mode         inf.W_DATA  <= inf.W_DATA  ;
177     end
178 end
```

Correct code:

```
140 always_ff@(posedge clk or negedge inf.rst_n) begin
141     if(!inf.rst_n)begin
142         inf.W_DATA  <= 'b0;
143     end
144     else begin
145         if(inf.C_in_valid && !inf.C_r_wb)      inf.W_DATA  <= inf.C_data_w;
146         else                write mode         inf.W_DATA  <= inf.W_DATA  ;
147     end
148 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.

i. I thought that **Formal Verification** is the most effective among the 4 tools. Actually, if the name of checker is well-defined, I can notice where the bugs may exist without waveform. With the prove function, the results of checkers can be clearly represented on a table, which helps me find errors faster.

Reference: Functional Formal Verification with Cadence JasperGold.pdf

https://blog.csdn.net/Holden_Liu/article/details/123147983

Lab03_Exercise_Note_protected.pdf