IC Lab Formal Verification
Lab11 Quick Test
2023 Fall

Name: 李品賢 **Student ID:** 312510151 **Account:** iclab021

(a) What is Formal verification?

Ans:

Formal verification is a procedure employed to prove the correctness of the design. This typically includes creating a mathematical model of the design and utilizing formal language, along with automated tools and formal methods, to validate whether the design aligns with the specified requirements.

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

Ans:

In contrast to Pattern-based verification, where stimulus is randomly generated to assess design functionality using methods like DFS, Formal verification provides a comprehensive approach to code testing. It systematically tests all potential stimulus one cycle at a time, reaching every possible stimulus similar to BFS. Consequently, formal verification is more likely to identify errors that may have been overlooked by Pattern-based verification.

And list the pros and cons for each.

Ans:

Formal verification:

Pros:

1. Accelerate IC design flow, verify begin prior to testbench creation and stimulation
2. Little randomization, more deterministic
3. Less testbench effort required
4. Reveals some bugs that simulation would never catch

Cons:

1. Time-consuming (especially for large designs)

✓

2. Less flexible for directed test

Pattern based verification:

Pros:

1. Quick verification of designs at early stage
2. It can ignore some state that won't happened
3. More readable, easier to understand and apply

Cons:

1. More testbench effort required
2. May miss some cases and states, cannot cover all possible errors

(b) What is glue logic?

Ans:

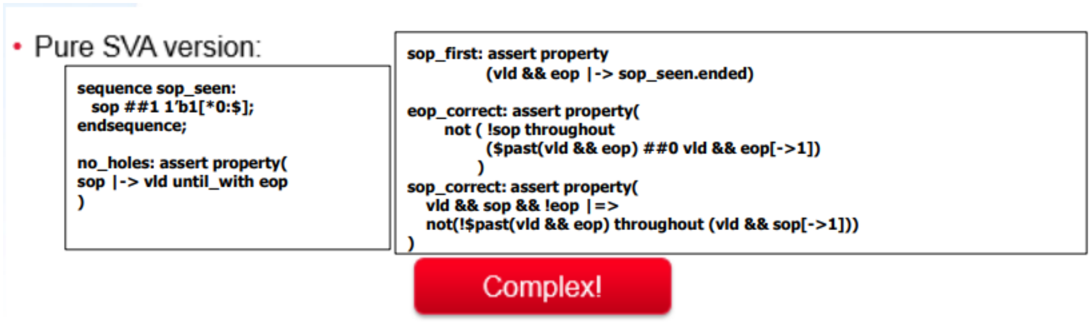
When modeling complex behaviors, SVA expressions can become overly complex. In such cases, some auxiliary logic can be used to observe and track events, helping us simplify the coding. This auxiliary logic is commonly referred to as "glue logic".

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

Ans:

Because SVA expressions can be complex, involving multiple conditions and logical operations, using glue logic can help transform these complex SVA expressions into clearer and more identifiable forms.

SVA could be complicated, like the figure shown below. The assertion is very complicate without glue logic.



- Pure SVA version:

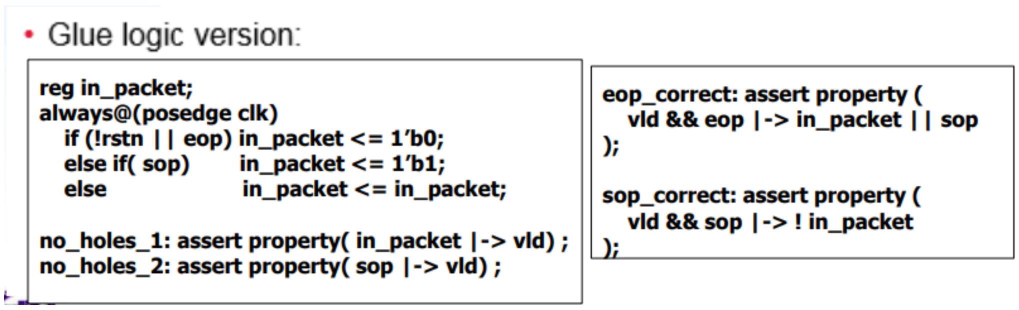
```
sequence sop_seen:
  sop ##1 1'b1[*0:$];
endsequence;

no_holes: assert property(
sop |-> vld until_with eop
)
```

```
sop_first: assert property
           (vld && eop |-> sop_seen.ended)

eop_correct: assert property(
     not ( !sop throughout
           ($past(vld && eop) ##0 vld && eop[->1])
         )
sop_correct: assert property(
   vld && sop && !eop |=>
   not(!$past(vld && eop) throughout (vld && sop[->1]))
)
```

The whole SVA will be simpler than the one without glue logic, If we use glue logic to create additional signal in figure below.



• Glue logic version:

```
reg in_packet;
always@(posedge clk)
   if (!rstn || eop) in_packet <= 1'b0;
   else if( sop)      in_packet <= 1'b1;
   else               in_packet <= in_packet;

no_holes_1: assert property( in_packet |-> vld) ;
no_holes_2: assert property( sop |-> vld) ;
```

```
eop_correct: assert property (
   vld && eop |-> in_packet || sop
);

sop_correct: assert property (
   vld && sop |-> ! in_packet
);
```

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

Ans:

Functional coverage assesses whether all essential functionalities of your design have been tested It includes property's related covers and covergroups. Also, it aims at checking specific states, conditions or sequences to be verified, focusing on verifying the completeness of a design's functionality. It is possible to represent all meaningful design functionality, and implements the verification plan, that is, what needs to be verified.

Pros:

1. Possible to represent all meaningful design functionality
2. Implements the verification plan.
3. Nothing is don't-care

Cons:

1. Requires planning, coding, and debug.
2. May be incomplete due to human error.

Code coverage will check all the possible cases written in the code, for example, branches, statements and expressions. Unlike functional coverage, code coverage focuses on the extent of code coverage and structural integrity during testing. Therefore, code coverage may not capture all meaningful design functionality, and can be noisy. It only tell you whether you have execute this code or not. In conclusion, it is a necessary but insufficient verification method.

Pros:

1. Easy to generate automatically ✓

2. guaranteed to be structurally complete

Cons:

1. May not capture all meaningful design functionality
2. Can be noisy – “don’t care” or duplicate covers

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

Ans:

Achieving 100% code coverage doesn't guarantee the adequacy of our verification, as code coverage solely verifies whether the code is "executed" or not, rather than assessing the correctness of the code. This implies that despite achieving 100% code coverage, functional coverage may not reach 100% due to the lack of coverage for certain corner cases.

(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.

Ans:

COI Coverage:

Each assertion is affected by some cover items, and the region will be called Cone Of Influence (COI). It finds the union of the all assertion COIs. The tool analyzes this checker and determines all the inputs, outputs and internal variables of the DUT that influence this checker. The result of this analysis is called the COI. Once the COI is determined, the tool can now determine the full state space of possible values the logic within the cone can take. It then parallelly explores every path in the state space and checks if the assertion can be proven wrong.

Proof Coverage:

Proof coverage is a subset of COI, representing the region that genuinely affects the assertion status. COI is the upper limit of potential proof coverage. While COI does not require an actual proof, proof coverage requires one. Identifying more unchecked code than COI measurement demands increased tool effort, making it a slower measurement than COI.

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

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

Ans:

ABVIP:

Definition: ABVIP stands for "Assertion-Based Verification Intellectual Properties." It comprises a comprehensive set of checkers and RTL that verify protocol compliance, such as ARM, AMBA, AXI, and others.

Objective: The primary aim of ABVIP is to elevate the accuracy and efficiency of the verification process by offering a comprehensive set of assertions based on protocols.

Benefits: ABVIP accelerates the verification process, improves debugging capabilities, and ultimately saves time for designers.

Scoreboard:

Definition: A scoreboard is designed to monitor both the input and output data of the Design Under Verification (DUV). It can identify issues like data packets dropped, duplicated data packets, and the order of data packets, while also detecting corrupted data packets.

Objective: The overarching goal of a scoreboard is to track and observe the data of the design being tested, ensuring its correct operation. Furthermore, it contributes to reducing the state-space complexity and the barriers to adoption.

Benefits: A scoreboard facilitates error detection and enhances the ease of debugging for designers.

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

What is the answer of the Lab Exercise?

1. Problem: arvalid should remain stable if arvalid and !arready;

   Line 90:

   Wrong: if(inf.AR_READY)

   Correct: if(n_state == AXI_AR)

✓

2. Problem: awvalid should remain stable if awvalid and !awready

   Line 124:

   Wrong: if(inf.AW_READY)

   Correct: if(n_state == AXI_AW)

3. Problem: Incoming data is binary; however, outgoing data is hexadecimal.

   Line 134:

   Wrong: inf.AW_ADDR <= {8'h1000_0000, inf.C_addr, 2'b0};

   Correct: inf.AW_ADDR <= {1'b1, 7'b0, inf.C_addr, 2'b0};

4. Problem: inf.C_r_wb is 0 in writing mode, not 1

   Line 145:

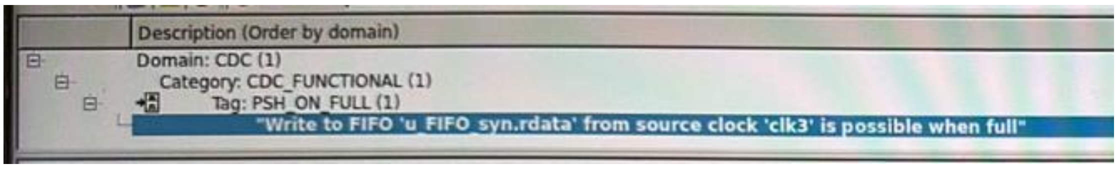
   Wrong: if(inf.C_in_valid && inf.C_r_wb)

   Correct: if(inf.C_in_valid && !inf.C_r_wb)

(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.

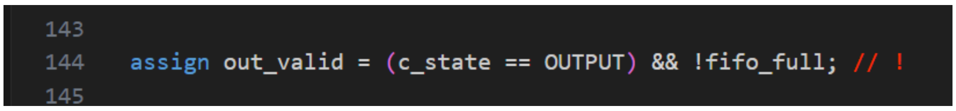
使用最有感的是 CDC 驗證，驗下去才發現 design 很多問題，許多 clock synchronization 的問題沒有考慮到。

在 Lab7 時，使用 JasperGold 去做 CDC 驗證時，遇到一個情況一值沒辦法通過驗證，波型已經正確了，但是還是會出現以下錯誤。

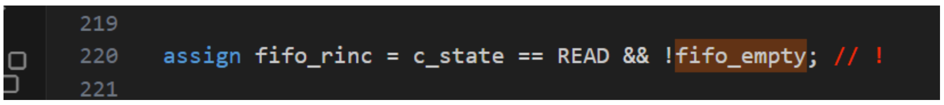


```
Description (Order by domain)
Domain: CDC (1)
    Category: CDC_FUNCTIONAL (1)
        Tag: PSH_ON_FULL (1)
            "Write to FIFO 'u_FIFO_syn.rdata' from source clock 'clk3' is possible when full"
```

後來找到原因，送進 FIFO module 的 winc 和 rinc，必須要使用 fifo_full 及 fifl_empty 當成判斷條件才可以，如下圖。



```
assign out_valid = (c_state == OUTPUT) && !fifo_full; // !
```



```
assign fifo_rinc = c_state == READ && !fifo_empty; // !
```

推斷可能是 tool 沒辦法直接找到 fifo_full 和 winc 或 fifl_empty 和 rinc 的關聯性，導致

沒辦法通過驗證。找到這個問題花了不少時間，因此印象深刻。