IC Lab Formal Verification
Lab11 Quick Test
2023 Fall

Name: 蘇柏叡 **Student ID:** 109511055 **Account:** iclab103

(a) What is Formal verification?

Formal verification 是利用數學方法驗證邏輯正確性的方式，一次驗證一個 cycle 去測試所有可能的 input 有沒有違反給定的 spec，且不會進行 timing 的檢查。因為不用有 stimulus，所以可以在 rtl 撰寫完成後就進行驗證，可以在最早期的 design flow 就發現問題，更有效率的修復。

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

Formal 是利用數學方法，將給定的 constraint 下的所有可能路徑 trace 過一遍做驗證，常用於 protocol 或較小的 design 當中；而 pattern 則是 based on DUT 的特定行為或模式，在每個 pattern 針對某條路徑去做驗證。

And list the pros and cons for each.

Formal：

Pros：因為是將所有可能展開檢查，所以覆蓋度會更高，有較高的機會檢測到電路的問題；且自動化程度較高、時間花費較少。

Cons：若系統過於複雜，會導致檢查數量呈指數上升，驗證時可能會耗費大量資源，且容易覆蓋到不會發生的路徑。

Pattern：

Pros：因只檢查特定模式下的行為，所以消耗的資源較少，適用於更廣泛的應用。

Cons：若使用者沒有考慮到完善的情況，容易造成覆蓋不完全，降低正確性；且自動化程度較低、時間花費較多。

(b) What is glue logic?

Glue logic 是指在 SVA 中用來減少 assertion 複雜度的額外邏輯，利用這些額外的邏輯，可以讓 SVA 的表示更加簡單好懂。

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

在合成時因為 glue logic 不影響整體電路 output 邏輯，所以 design compiler 會把

它合掉，不會對整體面積、timing 等問題產生影響。

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

Functional coverage：需要人工規劃設計 coverpoint，因此通常這些 coverpoint 會較有意義，但同時也容易遺漏某些 corner case，且設計時間會較長。

Code coverage：EDA tool 照給定的 rtl code 自動生成 cover item，希望每一行 code 都有被觸發到，但可能因為某些 case 本來就打不到，所以需要手動 waive 掉這些 alarm。

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

Code coverage 100%代表目前的 rtl 中每行都有被打到，但這並不代表你的 rtl code 就是對的，可能 design 需要某些功能、運算，但你的 rtl 裡面沒有，所以雖然 code coverage 100%，但 functionality 還是有可能是錯的。

(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：所有會影響到 formal environment 檢查時的 cover item 的區域，就是 COI。

Proof coverage：assertion 真正會檢查到的地方形成的區域。

Relationship：

Proof coverage 是 COI coverage 的一個子集合。

Tool Effort：

COI coverage：因為 COI 是所有會影響 output 的 cover item 形成的區域，所以只需要一直往前 trace 就可以，不需要用 formal engine，因此比較快。

Proof coverage：因為要找到真正 assertion 會檢查到的地方，所以會花比較多時間。

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

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

ABVIP：

Definition：全名為 assertion based verification intellectual properties，是由一堆 assertion 組成的 IP，用來檢查 protocol 等 design。

Objective：利用 assertion IP 來檢查 design，能夠讓驗證更嚴謹，更能系統性的檢查設計是否符合預期的功能及行為。

Benefit：像是這次 lab 使用的 AXI4 lite protocol，若要自己寫 assertion 檢查，會需要好幾千行，且容易出錯，使用 ABVIP 則可以大大降低寫 checker 的時間及出錯的機率。

Scoreboard：

Definition：指在 digital design 中驗證的組件、方法，用來監視系統在模擬中的進展和正確性。

Objective：像是監視器一樣，用來比較預期結果跟實際模擬結果是否一致，有助於檢查設計中的錯誤。

Benefit：通過持續監測和比較結果，有助於整體驗證流程，提供信心，確保設計在不同條件下的行為符合預期；且簡化了驗證流程，使測試更加高效，減少人為錯誤的可能性；並有助於在設計早期發現錯誤，使設計師能夠在問題變得更加複雜和昂貴之前解決問題。

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

What is the answer of the Lab Exercise?

1. AR_VALID 如果是用下圖的寫法，會有 handshake 多個 cycle 的情況，因為雖然上面的 state 控制是用 handshake 完成後就跳走，但這邊 AR_VALID 只要 AR_READY 是 1 就會一直拉著，根據 bridge 不同的寫法，可能會造成 AR_ADDR 吃到預期外的位置，或是 R_VALID 如果 delay 比較短，可能會跟 AR_VALID 重疊。

   正確的寫法應該用 n_state == AXI_AR 作為判斷式。

```
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
```

2. AW_VALID 如果是用下圖的寫法，就會跟 AR_VALID 發生類似的情況，AW_ADDR 可能吃到預期外的位置，或是 W_READY 如果 delay 比較短，可能會跟 AW_VALID 重疊。

   正確的寫法應該用 n_state == AXI_AW 作為判斷式。

```
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
```

3. AW_ADDR 的最高位 8bit 如果是下面的寫法，8’h1000_0000 超出了前面給定的 8bit，因此合成器會報 warning，且實際上被吃到的值會是 8’h1000_0000 的最小 8 位，也就是 8’h00，導致 W_DATA 寫進錯誤的地址中。

   正確的寫法應該是 inf.AW_ADDR <= {1’b0, 7’b0, inf.C_addr, 2’b0}。

```
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
```

4. C_r_wb 為 1 時是 read mode；0 是 write mode，因此下面的寫法會導致 W_DATA 可能永遠都吃不到 C_data_w，只能一直寫 0 進去。

   正確的寫法應該是 n_state == AXI_W。

```
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.

我覺得 IMC coverage 對我的驗證流程最有幫助，因為可以根據 cover hit 的數量，很快地找到是我的 checker 寫錯還是 pattern 有問題，像在 lab10 中測 coverage 時，一直跑不到 100%，去開 IMC 後發現某個 SPEC 的 hit 數量跟我預期的不一樣，去看 checker 後才發現是有條件沒有寫好，確定 checker 沒有問題之後，就可以確保 pattern 在給

定的條件下是100%有打到，讓測資的完整度更高。

上面的 debug 過程中，我原本一直以為條件沒有給會讓 hit 的數量暴增，因此當初看到 hit 數量有多有少時，就覺得是 pattern 寫錯，一直在找 pattern 哪裡有問題，後來才知道 cover 條件沒有寫 hit 數也有可能變少，讓我多花了不少時間 debug。