

Project2 RV12 ISA Formal

組別: Group23

組員: M16131111 童品綸、N26134992 賴郁明

內容

1.	Design Architecture	2
2.	Datapath	3
2.1	SRAI.....	3
2.2	BLTU.....	4
2.3	JAL	4
2.4	LW	5
2.5	AUIPC	6
3.	Pipeline Follower	6
4.	SVA	16
	SRAI assertion:	16
	LW assertion:.....	19
	AUIPC assertion:	20
	JAL assertion:	22
	BLTU assertion:	24
5.	Bugs Detection	27
6.	Steps to Resolve the Bug	29
7.	Result	30

1. Design Architecture

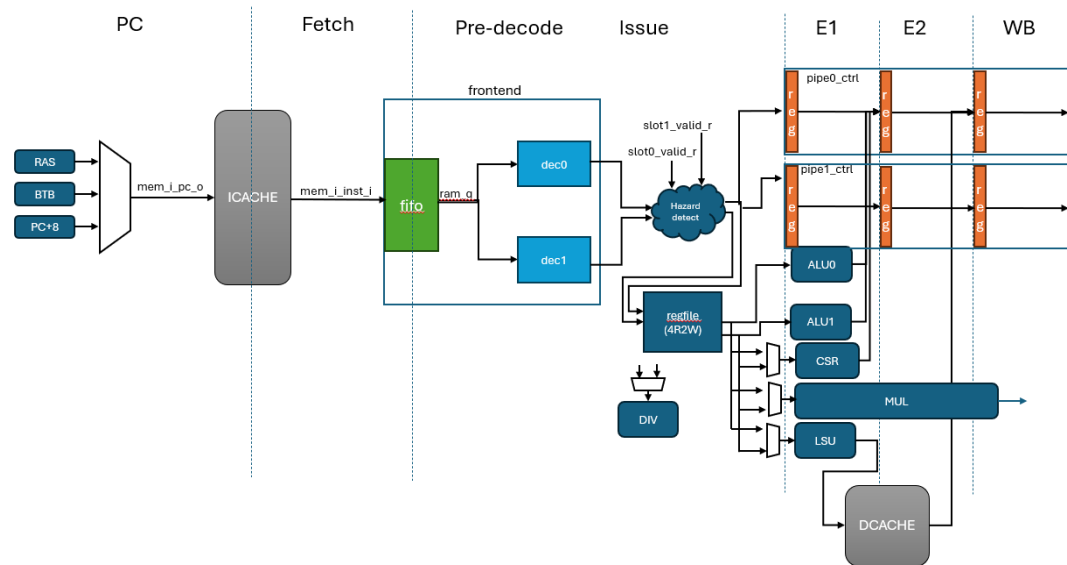


圖 1、CPU core architecture

這次驗證的 CPU 架構為 6 級 pipeline，支援 RV32I 指令集，且支援分支預測功能。其中的 PC stage，負責進行下一個要抓取指令之 PC 值計算，並在發生分支或跳躍時正確的切換下個 PC 位址。pre-decode stage + issue 負責對指令進行預解碼，還有拿解碼指令之地址去暫存器檔案抓地址，還有接著是判斷兩個指令的資料相依性來判斷是否可以同時執行指令，以及判斷前面 branch 預測結果跟實際結果是否相同，不相同則會清除以及加載本來該執行之指令 PC 到 PC stage，在 E1 stage 進行指令運算如 ALU、CSR 等等，在 E2 stage 進行的是去記憶體讀取資料以及寫入資料的動作，WB stage 則是將前面不論是載入記憶體資料或者運算結果等等存回目標暫存器的階段。

2. Datapath

2.1 SRAI

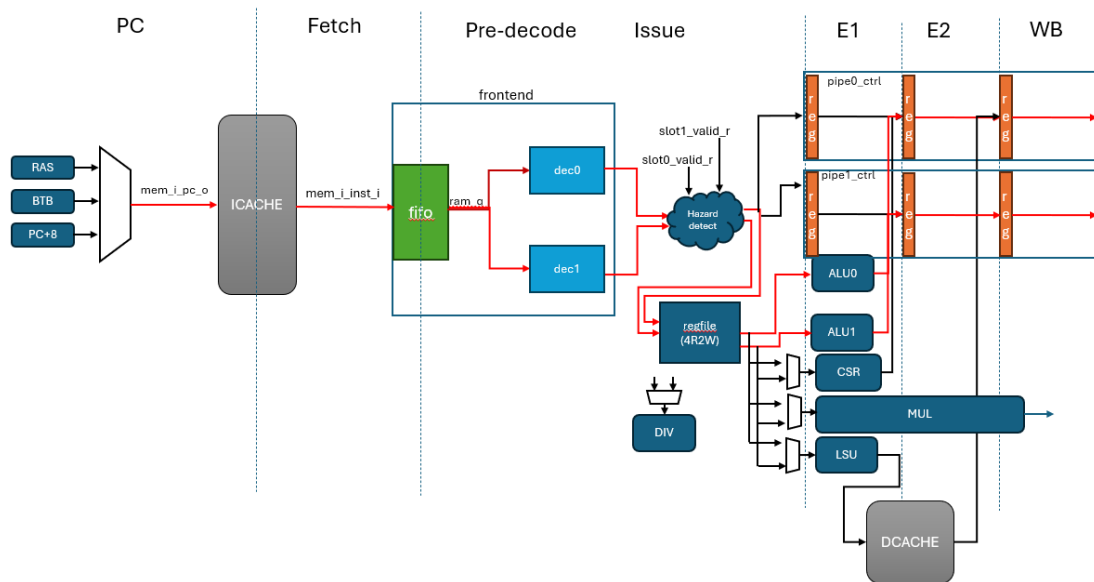


圖 2、SRAI datapath

SRAI 指令經過 pre-decode 以及 issue 解碼該指令獲得立即值，在 E1 時會透過算術邏輯單元進行位移，在 wb 時在寫回目標暫存器。

2.2 BLTU

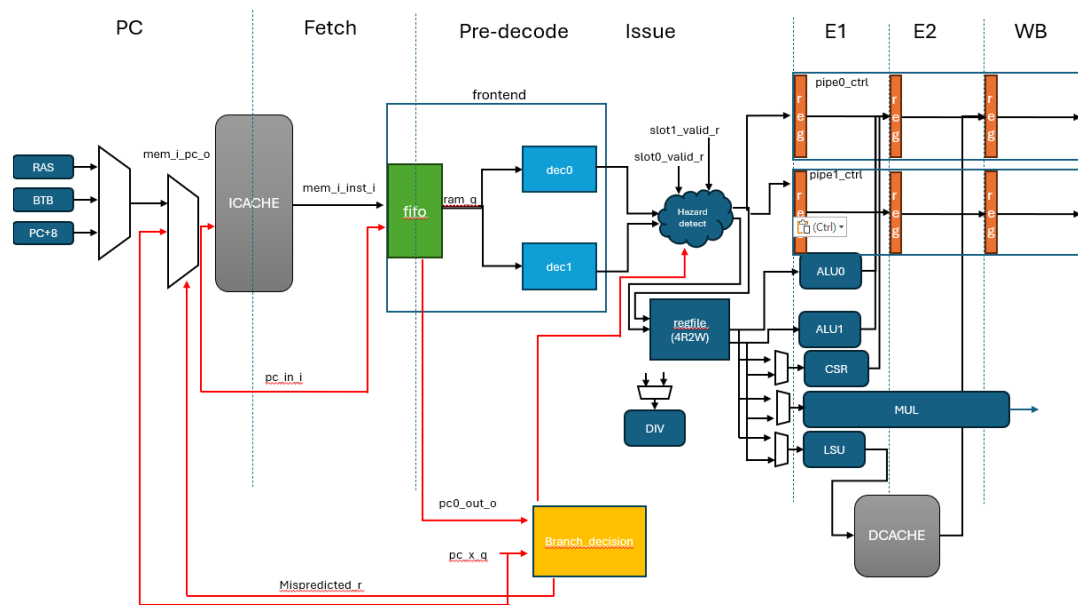


圖 3、BLTU datapath

BLTU 指令在一開始會先由於分支預測先猜測出可能結果，在 Pre-decode 以及 issue 時在看預測跟實際是否相符，相符則沒問題，不符則要清洗過去已經進入流水線之指令，並且載入實際該被執行之指令。

2.3 JAL

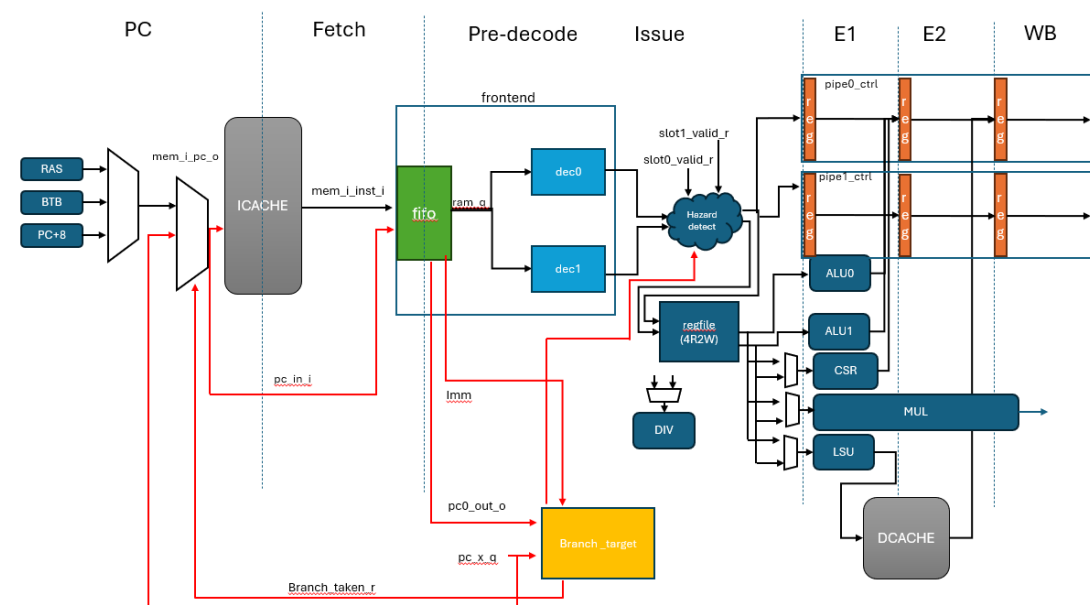


圖 4、JAL datapath

JAL 指令會在 pre-decode 算出跳躍之指令，並且進行跳躍，在 wb 時會將原本 pc 值寫入目的暫存器。

2.4 LW

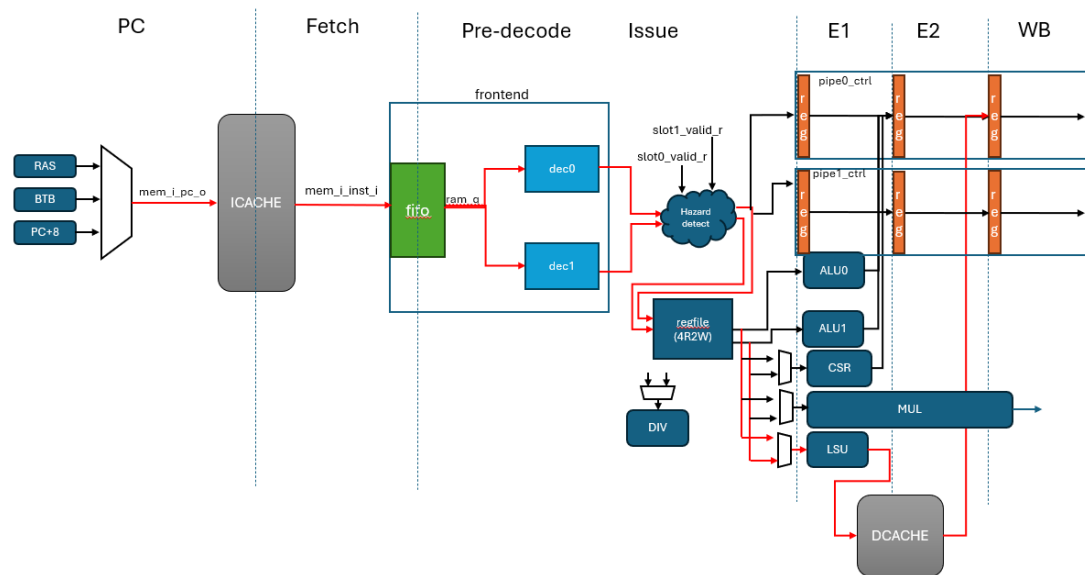


圖 5、LW datapath

LW 指令會 Pre-decode+Issue 算出應該去哪個記憶體位置載入資料，並且在 E2 時將要載入資料寫回暫存器檔案。

2.5 AUIPC

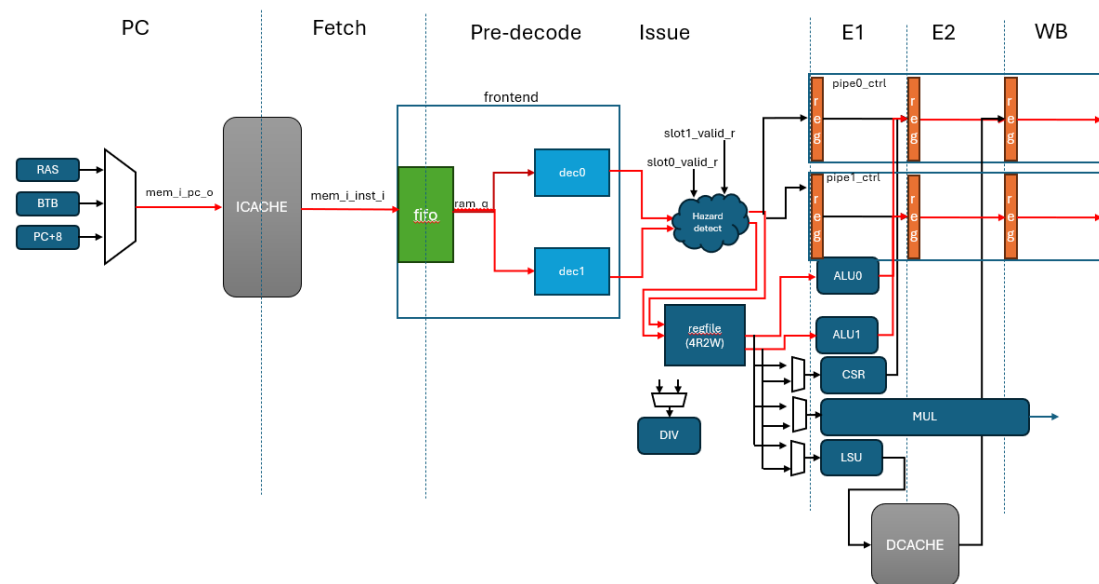


圖 6、AUIPC datapath

AUIPC 指令會在經過 pre-code 以及 issue 解碼該指令獲得立即值，在 E1 時會透過算術邏輯單元跟 PC 進行相加，在 wb 時在寫回目標暫存器。

3. Pipeline Follower

Scoreboard3

引導 GPT 使用 scoreboard3 驗證指令路徑上之 FIFO，第一步先給 GPT 說明文件以及 fetch_fifo module，使其掛上 scoreboard3 進行驗證，初始版本會抓到錯誤的輸出訊號如 result1。

接著 prompt2 告訴 GPT 要針對其中的 pc,insn 進行驗證，它可以找到更好的對應關係。接著 prompt3 使其按照 module 內之訊號名稱修正錯誤得命名。

Prompt4 後則指引 GPT 修改各式錯誤的部分，成功生出 SB3 驗證之 program。

Prompt1:



JG_FPV_COV_Basic_Training_2023_Y...
PDF



請你根據該文件93頁後的說明使用scoreboard3驗證該fetch_fifo

```
//-----  
//          biRISC-V CPU  
//          V0.8.1  
//          Ultra-Embedded.com  
//          Copyright 2019-2020  
//  
//          admin@ultra-embedded.com  
//  
//          License: Apache 2.0  
//-----  
// Copyright 2020 Ultra-Embedded.com  
//  
// Licensed under the Apache License, Version 2.0 (the "License"):
```

Result1:

verilog



複製程式碼

```
jasper_scoreboard_3 #(  
    .CHUNK_WIDTH(32),  
    .IN_CHUNKS(1),  
    .OUT_CHUNKS(1),  
    .ORDERING(`JS3_IN_ORDER)  
) u_scoreboard (  
    .rstN(rst_i),  
    .clk(clk_i),  
    .incoming_vld(push_i),  
    .incoming_data(data_in_i),  
    .outgoing_vld(pop0_i),  
    .outgoing_data(data0_out_o)  
);
```

Prompt2:

你好我想對該fifo module中的insn與pc進行 scoreboard3的驗證



對於 `fetch_fifo` 模組中的 `insn` (指令數據) 和 `pc` (程序計數器)，您可以利用 `jasper_scoreboard_3` 進行數據完整性和順序驗證。以下是詳細的步驟和信號映射建議。

Result2:

示例配置

```
verilog

jasper_scoreboard_3 #(
    .CHUNK_WIDTH(96),           // 64 bits for insn + 32 bits for PC
    .IN_CHUNKS(1),             // 單次輸入數據塊
    .OUT_CHUNKS(1),            // 單次輸出數據塊
    .ORDERING(`JS3_IN_ORDER)    // 驗證輸出是否按順序
) u_scoreboard (
    .rstN(rst_i),
    .clk(clk_i),
    .incoming_vld(push_i),      // FIFO 寫入有效信號
    .incoming_data({data_in_i, pc_in_i}), // 輸入數據組合 (insn + PC)
    .outgoing_vld(valid0_o),    // FIFO 輸出有效信號
    .outgoing_data({data0_out_o, pc0_out_o}) // 輸出數據組合 (insn + PC)
);
```

複製程式碼

Prompt3:

訊號名稱請按照fifo module中的命名

Result3:


```

) \
input logic          clk_i,
input logic          rst_i,
input logic          push_i,
input logic [WIDTH-1:0] data_in_i,
input logic [PC_WIDTH-1:0] pc_in_i,
input logic          valid0_o,
input logic [WIDTH-1:0] data0_out_o,
input logic [PC_WIDTH-1:0] pc0_out_o
);

// 實例化 jasper_scoreboard_3
jasper_scoreboard_3 #(
    .CHUNK_WIDTH(WIDTH + PC_WIDTH), // 組合數據寬度 (data + pc)
    .IN_CHUNKS(1),                  // 單次輸入塊數量
    .OUT_CHUNKS(1),                  // 單次輸出塊數量
    .ORDERING(`JS3_IN_ORDER)        // 按順序驗證
) u_scoreboard (
    .rstN(~rst_i),                  // Reset 信號
    .clk(clk_i),                    // Clock 信號
    .incoming_vld(push_i),          // 輸入有效信號
    .incoming_data({data_in_i, pc_in_i}), // 輸入數據 (指令 + 程序計數器)
    .outgoing_vld(valid0_o),         // 輸出有效信號
    .outgoing_data({data0_out_o, pc0_out_o}) // 輸出數據 (指令 + 程序計數器)
);

endmodule

```

Prompt4:

你的data_out指令沒有抓完全，bit數對不上data_in

感謝您的提醒！我再次檢查了 `fetch_fifo` 模組的代碼，發現確實需要對 `data0_out_o` 和 `data1_out_o` 的信號位數進行仔細匹配和組合，因為這些信號是 `data_in_i` 的分段結果。以下是修正後的 `fetch_fifo_checker` 和綁定實現：

嘿~你現在已經生的大致正確了，但是有一個問題你似乎注意到，就是他的PC因為會有對齊的問題，所以只要驗上29bit

< 2/2 >

outgoing_vld訊號抓到錯誤的了，請抓取會將資料pop出來的信號

Result4:

```
parameter MAX_PENDING = 2 // 最大掛起數據量
) (
  input logic          clk_i,
  input logic          rst_i,
  input logic          push_w,
  input logic [WIDTH-1:0] data_in_i,
  input logic [PC_WIDTH-1:0] pc_in_i,
  input logic          pop_complete_w,
  input logic [WIDTH/2-1:0] data0_out_o, // 下半段指令數據
  input logic [WIDTH/2-1:0] data1_out_o, // 上半段指令數據
  input logic [PC_WIDTH-1:0] pc0_out_o
);

// 截取需要驗證的 PC 高 29 位
wire [CHECK_PC_WIDTH-1:0] pc_in_check = pc_in_i[PC_WIDTH-1:PC_WIDTH-CHECK_PC_WIDTH];
wire [CHECK_PC_WIDTH-1:0] pc0_out_check = pc0_out_o[PC_WIDTH-1:PC_WIDTH-CHECK_PC_WIDTH];

// 實例化 jasper_scoreboard_3
jasper_scoreboard_3 #(
  .CHUNK_WIDTH(WIDTH + CHECK_PC_WIDTH), // 組合數據寬度 (完整指令 + 檢查用 PC)
  .IN_CHUNKS(1),                        // 單次輸入塊數量
  .OUT_CHUNKS(1),                        // 單次輸出塊數量
  .ORDERING(`JS3_IN_ORDER),             // 按順序驗證
  .MAX_PENDING(MAX_PENDING),             // 最大掛起數據量
  .SINGLE_CLOCK(1)                       // 單時鐘域設計
) u_scoreboard (
  .rstN(~rst_i),                        // Reset 信號
  .clk(clk_i),                          // Clock 信號
  .incoming_vld(push_w),                 // 輸入有效信號
  .incoming_data({data_in_i, pc_in_check}), // 輸入數據 (完整指令 + 截取後 PC)
  .outgoing_vld(pop_complete_w),          // 使用 pop_complete_w 作為輸出有效信號
  .outgoing_data({data1_out_o, data0_out_o}, pc0_out_check) // 拼接輸出指令和截取後
);

endmodule
```

信號由給 CLK、CST

Property table						
No filter						
Filter on name						
Type	Name	Engine	Bound	Traces	Time	
✓ Assert	fetch_fifo.fetch_fifo_checker_i.u_scoreboard.genblk6.core.genblk5.genblk1.data_integrity	Mpcusto...	Infinite	0		
✓ Assert	fetch_fifo.fetch_fifo_checker_i.u_scoreboard.genblk6.core.genblk5.genblk2.no_overflow	Mpcusto...	Infinite	0		
✓ Cover	fetch_fifo.fetch_fifo_checker_i.u_scoreboard.genblk6.core.genblk7.COVER[0].data_in	PRE	1	1		
✓ Cover	fetch_fifo.fetch_fifo_checker_i.u_scoreboard.genblk6.core.genblk7.COVER[0].data_out	Hp	2	1		
✓ Cover	fetch_fifo.fetch_fifo_checker_i.u_scoreboard.genblk6.core.genblk7.COVER[1].data_in	Hp	2	1		
✓ Cover	fetch_fifo.fetch_fifo_checker_i.u_scoreboard.genblk6.core.genblk7.COVER[1].data_out	Hp	3	1		
● Assume	assume:0	?		0		

Scoreboard3 驗證通過

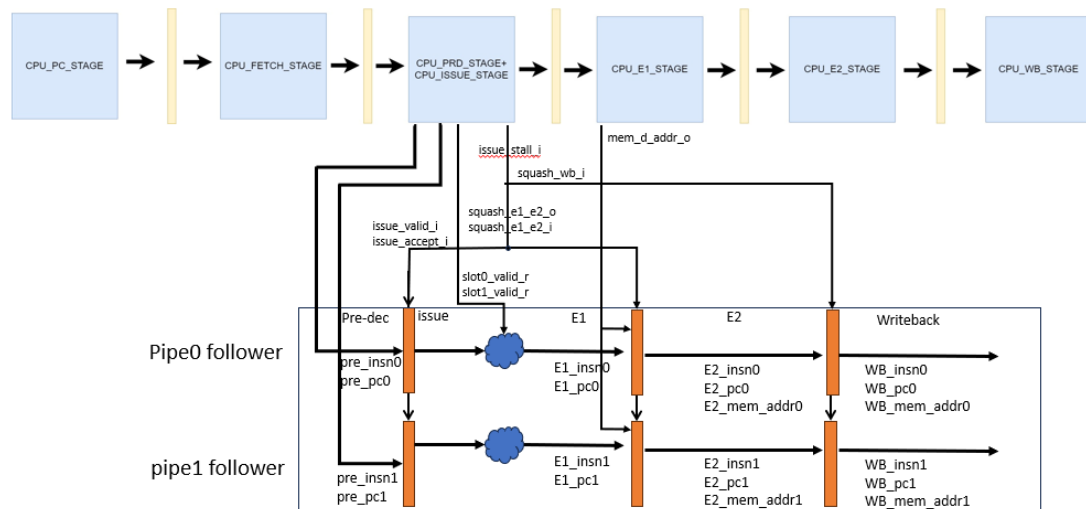


圖 7、pipeline follower

在進行 ISA formal 驗證時，我們會引導 GPT 在驗證環境中建立一個 pipeline follower 來記錄 CPU pipeline 中各個 stage 在執行的指令，透過引入 CPU 中的各種控制信號，使 pipeline follower 各個 stage 中的指令與 CPU 同步。且須注意因為這是 dual issue，所以在 pipeline 上一次是處理兩條指令時由於有些指令會有相依性，有些是沒有，所以他內部控制訊號會決定那些可以同時做，那些沒辦法同時做，來去控制後面指令能不能在流水線中繼續傳遞下去，由於兩條指令控制訊號不同，因此要分兩個 pipeline follower 來傳遞指令，之後跟之前要 single issue 的 CPU 一樣只要在 wb stage 驗證其指令解碼之正確性即可。因為在驗證的指令到達 wb stage 時，該指令之前所完成的指令皆以執

行完畢並寫入 register file，讓我們在計算 golden 值時可以避免 data dependency 的問題，因此可以根據解碼該指令的內容計算出該指令正確執行的 golden 值，並在該階段進行 assertion 的驗證。

在 pipeline follower 中除了有指令進行傳遞外，我們也引導 GPT 將 program counter 與指令一同傳下去，在指令傳到 wb stage 時，該指令對應的 pc 也會傳到，因此在進行 branch 驗證時可以使用該 pc 計算 golden 值。

引導 GPT 時生出 pipeline follower code 時，我們一開始是給他第一次作業的 sva 當範本，以及整個 ISA formal 的概念，還有這次驗證的目標以及 RTL 等等，但他生出來的東西不盡理想，如同底下 Prompt1 以及 Result1 所示，他的 pipeline follower 只生出二級，而且控制訊號還自己捏造，所以後來我們發現他沒辦法一次 trace 太多 code，所以我們採用一級一級讓他 trace 的方式，並且有提醒他要在 always block 裡正緣觸發才有可能為控制信號，然後也會將與語法錯或是創造不存在的訊號跟他講，並且時刻提醒他不要只看名字要看真正指令跟 PC 經過的地方，如同 Prompt2、Prompt3、Prompt4、Prompt5 那樣，最後成功把每級的控制訊號都找到，如同 Result2 那樣，最後就把每級都找出來就可以完成，然後中間訊號有找到接近的但沒有完成抓對時，我們也不會直接修改或者跟她說某某訊號有錯，而是會把背後原因跟他講讓他自己去發現並改善，如同 Prompt6 以及 Result3 所示。

Prompt 1:

我想要你生成驗證一個biriscv CPU的SVA，具體步驟如下：

1.我會先給你我先使用isa formal 驗證另一顆RV12 CPU我所寫出的isa.sva，這是一顆single issue、6 stage的CPU，SVA主要的組成有

(1) pipeline follower: 目的是要追蹤在各個stage中目前在執行什麼指令

你要先依照指令的傳遞找出這顆CPU的各個stage，並找出控制指令在stage間傳輸的控制信號，仿照其邏輯來創建pipeline follower

(2) assertions: 寫出驗證 SRAI、BLTU、JAL、LW、AUIPC五個指令的assertions

SRAI、AUIPC的驗證方法為: 等該指令到達WB stage時，在下個cycle驗證past golden跟 result是否相同

LW的驗證方法為: 等該指令到達WB stage時，在該cycle驗證golden跟 result是否相同

JAL的驗證方法為: 等該指令到達WB stage時，觸發JAL_trigger，並將golden存起來，等待下一個有效的指令到達WB stage在驗證該指令PC與golden是否相同

BLTU的驗證方法為: 等該指令到達WB stage時，觸發BLTU_trigger，並將golden存起來，等待下一個有效的指令到達WB stage在驗證該指令PC與golden是否相同，由於branch的golden有兩種可能，有進行分支與否，因此會分成兩種情況

2.我想要你根據範例isa.sva來學習，並寫出驗證biriscv CPU的SVA，biriscv CPU是一個6級、dual issue、in-order的CPU，在module名稱與stage分類及stall、flush訊號皆與RV12不同，因此你必須先trace該biriscv CPU的source code 找出對應的指令輸入，stage及控制信號來建立pipeline follower

，在引用信號時命名方式須按照模組的階層關係如在pipe1_ctrl module裡的squash_wb_i名為

(u_core.u_issue.u_pipe1_ctrl.squash_wb_i)

Result1:

```

// Fetch stage
always_ff @(posedge clk_i or negedge rst_i) begin
    if (!rst_i) begin
        fetch0_instr <= 32'b0;
        fetch1_instr <= 32'b0;
    end else if (!issue0_stall) begin
        fetch0_instr <= u_core.u_fetch.fetch0_instr_i;
        fetch1_instr <= u_core.u_fetch.fetch1_instr_i;
    end
end

// Decode stage
always_ff @(posedge clk_i or negedge rst_i) begin
    if (!rst_i) begin
        decode0_instr <= 32'b0;
        decode1_instr <= 32'b0;
    end else if (!flush) begin
        decode0_instr <= fetch0_instr;
        decode1_instr <= fetch1_instr;
    end
end
end

```



```

// Assertions for instructions
property check_SRAI;
    @(posedge clk_i)
    disable iff (!rst_i)
    (wb0_instr[6:0] == 7'b0010011 && wb0_instr[14:12] == 3'b101) |->
    (##1 $past(golden_SRAI) == result_SRAI);
endproperty

property check_AUIPC;
    @(posedge clk_i)
    disable iff (!rst_i)
    (wb0_instr[6:0] == 7'b0010111) |->
    (##1 $past(golden_AUIPC) == result_AUIPC);
endproperty

```

Prompt 2:



在這個版本中你還是遇到的一樣德問題:

```
WARNING (WNL021): ../property/isa3.sva(14): external reference
"riscv_core.u_fetch.fetch_instr" remains unresolved and will be
treated as a free net.
WARNING (WNL021): ../property/isa3.sva(16): external reference
"riscv_core.u_fetch.fetch_pc_0" remains unresolved and will be
treated as a free net.
WARNING (WNL021): ../property/isa3.sva(17): external reference
"riscv_core.u_fetch.fetch_pc_1" remains unresolved and will be
treated as a free net.
WARNING (WNL021): ../property/isa3.sva(57): external reference
"riscv_core.u_decode.decode_stall" remains unresolved and will be
treated as a free net.
WARNING (WNL021): ../property/isa3.sva(72): external reference
"riscv_core.u_exec.exec_stall" remains unresolved and will be
treated as a free net.
WARNING (WNL021): ../property/isa3.sva(87): external reference
"riscv_core.u_lsu.lsu_stall" remains unresolved and will be treated
as a free net.
```

你創造了不存在的stall訊號用來控制，你現在應該做的是去分析我給你的biriscv的各個module，並找出其中有在將指令進行傳遞的always blocks 並根據其是否將指令及PC傳遞到下個stage的條件作為控制信號

Prompt 3:

你的分法似乎是根據訊號的或模組的名稱來進行分類，我想教你一個更正確的方法，首先我們要關注在CPU中指令的傳遞，要先找到進入CPU的指令信號，並沿著指令經過的路徑，只要有經過正緣觸發的always block就會傳到下個 stage，請根據此方法告訴我你分析出的pipeline stage如何分配

Prompt 4:

你的分法似乎是根據訊號的或模組的名稱來進行分類，我想教你一個更正確的方法，首先我們要關注在CPU中指令的傳遞，要先找到進入CPU的指令信號，並沿著指令經過的路徑，只要有經過正緣觸發的always block就會傳到下個 stage，請根據此方法告訴我你分析出的pipeline stage如何分配

Prompt5:

還是有一點問題，我們先從E1 stage開始修改，你現在的重點是要找到 pipe ctrl中哪一個always posedge block有在進行指令與PC的傳遞並將其仿造到pipeline follower中

Result2:

```
else if (riscv_core.u_pipe0_ctrl.valid_e1_q &&
        ~(riscv_core.u_pipe0_ctrl.squash_e1_e2_o || riscv_core.u_pipe0_ctrl.squash_e1_e
        exec_instr_0 <= riscv_core.u_pipe0_ctrl.opcode_e1_q;
        exec_instr_1 <= 32'd0; // 因為是單發射的 E1 階段
        exec_pc_0    <= riscv_core.u_pipe0_ctrl.pc_e1_q;
        exec_pc_1    <= 32'd0;
    end
```

Prompt6:

從 fetch_fifo 模組的 data0_out_o, data1_out_o 取得指令。你現在取的是從fifo取出後又經過了一些邏輯，因此我想要你直接取fifo 出來的結果

Result3:

```
// Fetch stage: 從 fetch_fifo 的原始輸出直接取得指令與PC
assign fetch_instr_0 = riscv_core.u_decode.u_fifo.ram_q[riscv_core.u_decode.u_fifo.rd_
assign fetch_instr_1 = riscv_core.u_decode.u_fifo.ram_q[riscv_core.u_decode.u_fifo.rd_
assign fetch_pc_0    = riscv_core.u_decode.u_fifo.pc_q[riscv_core.u_decode.u_fifo.rd_p
assign fetch_pc_1    = riscv_core.u_decode.u_fifo.pc_q[riscv_core.u_decode.u_fifo.rd_p
```

4. SVA

SRAI assertion:

第一步先給 GPT Project1 的 isa.sva 引導它生出 biriscv 的 SRAI assertions

但此時會有明顯錯誤，如在 assertion 中 past 的使用會有錯誤，因此必須繼續引導它修正，告訴它在驗證時是使用過去的 golden，與現在的 result 相比。

最後在模擬時使用 counterexample waveform 及一些提示讓 GPT 找到錯誤的

地方並進行修正。

Prompt1:



現在我要你生成驗證SRAI指令的assertions 請根據RV12的範例，
isa.sva，生成驗證biriscv SRAI指令的assertions

Result1:

```
// 定義 SRAI 指令條件
assign is_SRAI_insn = (wb_insn_opcode == 7'b0010011) &&
                      (wb_insn_funct3 == 3'b101) &&
                      (wb_insn_funct7 == 7'b0100000);

// 計算黃金值 (golden value)
assign golden_SRAI = $signed(wb_rs1_data) >>> wb_insn_shamt;

// 從寄存器檔案讀取結果
assign result_SRAI = rf[wb_insn_rd_addr];

// SRAI 驗證的 property
SRAI_check: assert property(@(posedge clk_i) disable iff(!rst_i)
    (is_SRAI_insn ==> (golden_SRAI == $past(result_SRAI))))
    );
```

Prompt2:

assertion有錯誤，在驗證時應該是使用過去的golden比較從reghfile
取出的result

Result2:

```

assign RESULT_insn0_SRAI = {W0_insn0_u_data};

// SRAI 驗證的 property
SRAI_insn0_check: assert property(@(posedge clk_i) disable iff(!rst_i)
    (is_SRAI_insn0 |> ($past(golden_insn0_SRAI) == result_insn0_SRAI))
);

SRAI_insn1_check: assert property(@(posedge clk_i) disable iff(!rst_i)
    (is_SRAI_insn1 |> ($past(golden_insn1_SRAI) == result_insn1_SRAI))
);

endmodule

```

Prompt3:



請你根據波型找到SVA可能錯誤的地方，似乎在stall的時候不該驗證該指令

Result3:

根據波形圖和描述，SVA 的問題可能出現在 stall 條件下仍然進行了 SRAI 指令的驗證。這種情況下，因為 pipeline 被 stall，SRAI 指令不應該觸發驗證。

以下是修正後的 SVA 寫法，考慮到 `issue_stall_i` 信號的影響：

修正的 SRAI Assertions

```

systemverilog
// SRAI 驗證的 property
SRAI_insn0_check: assert property(@(posedge clk_i) disable iff(!rst_i)
    (is_SRAI_insn0 && !u_pipe1_ctrl.issue_stall_i |> $past(golden_insn0_SRAI) == result_i
);

SRAI_insn1_check: assert property(@(posedge clk_i) disable iff(!rst_i)
    (is_SRAI_insn1 && !u_pipe1_ctrl.issue_stall_i |> $past(golden_insn1_SRAI) == result_i
);

```

LW assertion:

第一步先給 GPT Project1 的 isa.sva 引導它生出 biriscv 的 LW assertions，但此時會有明顯錯誤，如在 assertion 中是比較當前 cycle 而非下一個 cycle，也不必使用到 \$past 函數，因此使用 counterexample waveform 及一些提示讓 GPT 找到錯誤的地方並進行修正。

Prompt1:



現在我要你生成驗證LW指令的assertion請根據RV12的範例，isa.sva，生成驗證biriscv LW指令的assertions

Result1:

LW 指令驗證 Assertions

以下是 LW 指令的 `insn0` 和 `insn1` 的 assertion 生成：

```
systemverilog 複製程式碼

// LW 指令的 Assertions
LW_insn0_check: assert property(@(posedge clk_i) disable iff(!rst_i)
    (is_LW_insn0 && !u_pipe0_ctrl.issue_stall_i | =>
        $past(golden_insn0_LW) == result_insn0_LW)
);

LW_insn1_check: assert property(@(posedge clk_i) disable iff(!rst_i)
    (is_LW_insn1 && !u_pipe1_ctrl.issue_stall_i | =>
        $past(golden_insn1_LW) == result_insn1_LW)
);
```

Prompt2:



先前生出的SVA有錯誤，counter example 波型如圖所示，LW應該只需在指令到達WB stage時，取得該cycle時之golden與result驗證是否相同，

Result2:

驗證當前 golden_LW 與 result_LW 是否相符：

更新後的 LW 指令 Assertion

```
systemverilog
// 定義 LW 指令判斷
assign is_LW_insn = (wb_insn_opcode == 7'b0000011) && (wb_insn_func3 == 3'b010) && (!core

// 計算 golden 值
assign golden_LW = wb_rs1_data + wb_I_imm;

// 驗證 Assertion
LW_check: assert property(
    @(posedge HCLK) disable iff (!HRESETn)
    (is_LW_insn && !wb_stall_o_delay) |-> (golden_LW == result_LW)
);
```

AUIPC assertion:

第一步先給 GPT Project1 的 isa.sva 引導它生出 biriscv 的 AUIPC assertions
但此時會有明顯錯誤，如在 assertion 中 past 的使用會有錯誤，因此必須繼續引導它修正，告訴它在驗證時是使用過去的 golden，與現在的 result 相比。

Prompt1:



isa.sva

檔案

現在我要你生成驗證AUIPC指令的assertion請根據RV12的範例，isa.sva，生成驗證biriscv AUIPC指令的assertions

Result1:

systemverilog

複製程式碼

```
// 判斷 AUIPC 指令
assign is_AUIPC_insn = (wb_insn_opcode == 7'b0010111) && (!core.wb_unit.wb_insn_o.bubble)

// 計算 AUIPC 的 golden 值
assign golden_AUIPC = wb_pc + wb_U_imm;

// 寫回結果
assign result_AUIPC = core.int_rf.rf[wb_insn_rd_addr_reg];

// AUIPC Assertion
AUIPC_check: assert property(
    @(posedge HCLK) disable iff(!HRESETn)
    (is_AUIPC_insn && !wb_stall_o_delay) |-> (golden_AUIPC == result_AUIPC)
);
```

Prompt2:

assertion有錯誤，在驗證時應該是使用過去的golden比較從reghfile取出的result

Result2:

修訂後的 AUIPC Assertion

```
systemverilog 複製程式碼

// 定義 AUIPC 指令觸發條件
assign is_AUIPC_insn = (wb_insn_opcode == 7'b0010111) && (!core.wb_unit.wb_insn_o.bubble);

// 計算 golden 值 (PC + imm)
assign golden_AUIPC = wb_pc + wb_U_imm;

// 從寄存器文件讀取的結果值
assign result_AUIPC = core.int_rf.rf[wb_insn_rd_addr_reg];

// Assertion for AUIPC 指令
AUIPC_check: assert property(
    @(posedge HCLK) disable iff(!HRESETn)
    (is_AUIPC_insn && !wb_stall_o_delay) | => ($past(golden_AUIPC) == result_AUIPC)
);
```

```
assign result_insn0_SRAI = rf[wb_insn0_rd_addr];

// SRAI 驗證的 property
SRAI_insn0_check: assert property(@(posedge clk_i) disable iff(!rst_i)
    (is_SRAI_insn0 | => ($past(golden_insn0_SRAI) == result_insn0_SRAI))
);

SRAI_insn1_check: assert property(@(posedge clk_i) disable iff(!rst_i)
    (is_SRAI_insn1 | => ($past(golden_insn1_SRAI) == result_insn1_SRAI))
);

endmodule
```

JAL assertion:

第一步先給 GPT Project1 的 isa.sva 引導它生出 biriscv 的 JAL assertions

可以看出 GPT 生出的 SVA 不管是 Implication 前的條件到取得來比較的 pc 值訊號都是有問題的，因此必須繼續引導他修正。

在這種程式架構上不符合 isa formal 行為的明顯錯誤，我們修正的過程透過下 prompt 告知 GPT 在 implication 的條件上與我給的範例不符，並告知缺失的部分及程式正確運作的方式，他就會生出比較正確的版本，經過幾次的修正，我們透過觀察比較範例 sva 跟 GPT 生出來的版本的差異，直到程式的架構符合 isa formal 驗證的行為。

Prompt1:



- 9 現在我要你生成驗證JAL指令的assertion請根據RV12的範例，isa.sva，生成驗證bircsv JAL指令的assertions

Result1:

JAL Assertion Code

```
systemverilog 複製程式碼

// 定義 JAL 指令觸發條件
assign is_JAL_insn = (wb_insn_opcode == 7'b1101111) && (!core.wb_unit.wb_insn_o.bubble) &&

// 計算 golden 值
assign golden_PC_JAL = (wb_pc + wb_insn_immJ) & 32'hffff_fffc; // 目標地址 (對齊)
assign golden_rd_JAL = wb_pc + 32'h4; // 返回地址 (PC + 4)

// 從寄存器文件讀取的結果值
assign result_rd_JAL = core.int_rf.rf[wb_insn_rd_addr_reg]; // 寄存器寫入的值
assign result_PC_JAL = core.wb_unit.wb_pc_o; // 核心當前 PC

// JAL assertion: 驗證目標 PC 和寄存器寫回數據
JAL_PC_check: assert property(
    @(posedge HCLK) disable iff(!HRESETn)
    (is_JAL_insn && !wb_stall_o_delay) | => ($past(golden_PC_JAL) == result_PC_JAL)
);

JAL_rd_check: assert property(
    @(posedge HCLK) disable iff(!HRESETn)
    (is_JAL_insn && !wb_stall_o_delay) | => $past(golden_rd_JAL) == result_rd_JAL
);
```

Prompt2:

根據範例SVA的JAL驗證方式，須符合三個條件，第一個是JAL_trigger會在第一個JAL指令進入WB stage時trigger並直到下個合法指令進入WB stage才會放下，第二個是要確認當前指令是否合法，以及要確定被驗證的PC不等於第一個進入WB stage的JAL指令

Result2:

```

// 定義 JAL trigger
always_ff @(posedge HCLK or negedge HRESETn) begin
    if (!HRESETn) begin
        JAL_trigger <= 1'b0;
        JAL_original_pc <= 32'd0;
        JAL_golden_pc_reg <= 32'd0;
    end else if (is_JAL_insn && wb_no_bubble) begin
        JAL_trigger <= 1'b1; // 第一個 JAL 指令進入 WB
        JAL_original_pc <= wb_pc; // 保存 JAL 指令的 PC
        JAL_golden_pc_reg <= golden_PC_JAL; // 保存 JAL 的目標地址
    end else if (wb_no_bubble && is_valid_insn) begin
        JAL_trigger <= 1'b0; // 下一個合法指令進入 WB 後釋放
    end
end

// 排除與其他指令的干擾 (例如 BLTU)
assign JAL_same_pc = (JAL_original_pc == wb_pc);

// Assertion for PC 校驗
JAL_PC_check: assert property (
    @(posedge HCLK) disable iff (!HRESETn)
    (JAL_trigger && wb_no_bubble && !JAL_same_pc) |-> (JAL_golden_pc_reg == core.wb_unit.wb_
);

// Assertion for 寄存器寫回數據校驗
JAL_rd_check: assert property (
    @(posedge HCLK) disable iff (!HRESETn)
    (is_JAL_insn && !wb_stall_o_delay) |=> ($past(golden_rd_JAL) == result_rd_JAL)
);

```

BLTU assertion:

第一步也是給 GPTproject1 的 isa.sva 叫他生出 biriscv 的 BLTU 的 assertion

可以看出 GPT 生出的 SVA 不管是 Implication 前的條件到取得來比較的 pc 值訊號都是有問題的，因此必須繼續引導他修正在這種程式架構上不符合 isa formal 行為的明顯錯誤，我們修正的過程透過下 prompt 告知 GPT 在 implication 的條件上與我給的範例不符，並告知缺失的部分及程式正確運作的方式，他就會生出比較正確的版本，經過幾次的修正，我們透過觀察比較範例 sva 跟 GPT 生出來的版本的差異，直到程式的架構符合 isa formal 驗證的行為

Prompt1:



isa.sva
檔案

現在我要你生成驗證BLTU指令的assertions 請根據RV12的範例 isa.sva，生成驗證bircsv BLTU指令的assertions

Result1:

```
// Assertions for BLTU Pipe 0
BLTU_taken_pipe0: assert property (@(posedge clk_i) disable iff (!rst_i)
    (is_BLTU_insn0 && (wb_rs1_data0 < wb_rs2_data0)) | =>
    (golden_taken_pc0 == riscv_core.u_frontend.u_pc.pc_reg)
) else $error("BLTU Pipe0 taken PC mismatch");

BLTU_no_taken_pipe0: assert property (@(posedge clk_i) disable iff (!rst_i)
    (is_BLTU_insn0 && !(wb_rs1_data0 < wb_rs2_data0)) | =>
    (golden_no_taken_pc0 == riscv_core.u_frontend.u_pc.pc_reg)
) else $error("BLTU Pipe0 not taken PC mismatch");

// Assertions for BLTU Pipe 1
BLTU_taken_pipe1: assert property (@(posedge clk_i) disable iff (!rst_i)
    (is_BLTU_insn1 && (wb_rs1_data1 < wb_rs2_data1)) | =>
    (golden_taken_pc1 == riscv_core.u_frontend.u_pc.pc_reg)
) else $error("BLTU Pipe1 taken PC mismatch");

BLTU_no_taken_pipe1: assert property (@(posedge clk_i) disable iff (!rst_i)
    (is_BLTU_insn1 && !(wb_rs1_data1 < wb_rs2_data1)) | =>
    (golden_no_taken_pc1 == riscv_core.u_frontend.u_pc.pc_reg)
) else $error("BLTU Pipe1 not taken PC mismatch");
```

Prompt2:

根據範例SVA的BLTU implication條件，應該要符合三個條件，第一個是BLTU_trigger在第一個BLTU指令進入WB stage時要trigger並直到下個合法指令進入wb stage才會放下。第二個要確認當前指令是否合法，判斷方式是當前wb stage指令不能全0，第三個是為了不要驗證到當前的BLTU指令因此被驗證指令的PC不能等於第一個進入的BLTU指令的PC，此外要使用BLTU_state來分有跳跟沒跳兩種BLTU assertion，請根據isa.sva修正

Result2:

```

// BLTU assertions for pipe0
BLTU_taken_pipe0: assert property (@(posedge clk_i) disable iff (!rst_i)
    (BLTU_trigger_pipe0 && (BLTU_state_pipe0 == 2'b01) && wb_no_bubble_pipe0 && !same_pc_p
    (golden_pc_pipe0 == riscv_core.u_issue.u_pipe0_ctrl.wb_pc_o)
);

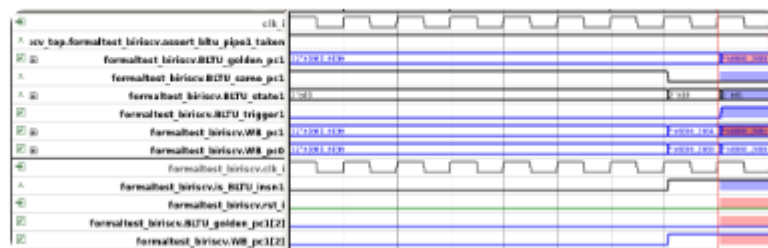
BLTU_no_taken_pipe0: assert property (@(posedge clk_i) disable iff (!rst_i)
    (BLTU_trigger_pipe0 && (BLTU_state_pipe0 == 2'b10) && wb_no_bubble_pipe0 && !same_pc_p
    (golden_pc_pipe0 == riscv_core.u_issue.u_pipe0_ctrl.wb_pc_o)
);

// BLTU assertions for pipe1
BLTU_taken_pipe1: assert property (@(posedge clk_i) disable iff (!rst_i)
    (BLTU_trigger_pipe1 && (BLTU_state_pipe1 == 2'b01) && wb_no_bubble_pipe1 && !same_pc_p
    (golden_pc_pipe1 == riscv_core.u_issue.u_pipe1_ctrl.wb_pc_o)
);

BLTU_no_taken_pipe1: assert property (@(posedge clk_i) disable iff (!rst_i)
    (BLTU_trigger_pipe1 && (BLTU_state_pipe1 == 2'b10) && wb_no_bubble_pipe1 && !same_pc_p
    (golden_pc_pipe1 == riscv_core.u_issue.u_pipe1_ctrl.wb_pc_o)
);

```

Prompt3:



根據jg產生的counter example波型有以下錯誤，golden會與WB_pc1不符，會不會比較的對象不會是WB_pc1，因為這是個dual issue CPU，所以下個合法指令一定會是2個被讀取的指令中先被讀取到的，你覺得是哪個呢？

Result3:

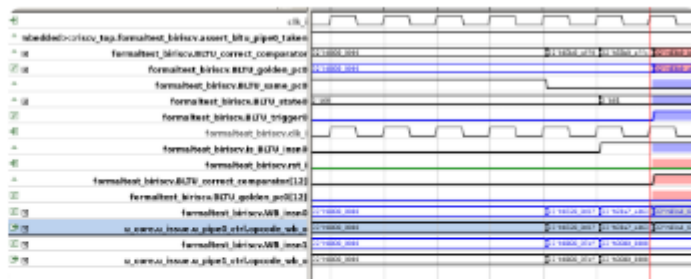
```
// BLTU assertions for pipe0
BLTU_taken_pipe0: assert property (@(posedge clk_i) disable iff (!rst_i)
    (BLTU_trigger_pipe0 && (BLTU_state_pipe0 == 2'b01) && wb_no_bubble_pipe0 && !same_pc_p
    (golden_pc_pipe0 == riscv_core.u_issue.u_pipe0_ctrl.wb_pc_o)
);

BLTU_no_taken_pipe0: assert property (@(posedge clk_i) disable iff (!rst_i)
    (BLTU_trigger_pipe0 && (BLTU_state_pipe0 == 2'b10) && wb_no_bubble_pipe0 && !same_pc_p
    (golden_pc_pipe0 == riscv_core.u_issue.u_pipe0_ctrl.wb_pc_o)
);

// BLTU assertions for pipe1
BLTU_taken_pipe1: assert property (@(posedge clk_i) disable iff (!rst_i)
    (BLTU_trigger_pipe1 && (BLTU_state_pipe1 == 2'b01) && wb_no_bubble_pipe1 && !same_pc_p
    (golden_pc_pipe1 == riscv_core.u_issue.u_pipe1_ctrl.wb_pc_o)
);

BLTU_no_taken_pipe1: assert property (@(posedge clk_i) disable iff (!rst_i)
    (BLTU_trigger_pipe1 && (BLTU_state_pipe1 == 2'b10) && wb_no_bubble_pipe1 && !same_pc_p
    (golden_pc_pipe1 == riscv_core.u_issue.u_pipe1_ctrl.wb_pc_o)
);
```

5. Bugs Detection



嗨~我們現在要你找出這個CPU可能存在的bug，我們透過波形可以看到這個CPU的指令1之指令資料在pipeline follower跟在CPU抓取出來的有落差，所以代表指令在CPU傳遞的過程中可能遭受一些不對的邏輯來造成資料損壞，你可以把每個module看一遍，找出可能的問題嗎？

圖 8、A bug waveform and a prompt to guide GPT in determining and fixing

指令在進來後會跟 pipeline follower 裡的指令最前面一個 bit 有差，經過觀察，從 CPU 裡抓出來的指令訊號，第一個 bit 永遠都會是 0，代表 CPU 硬體在

問題分析

1. 資料傳遞錯誤：

- 波形顯示指令資料在 pipeline 追蹤的過程中第一位bit發生變化，可能是資料處理或傳遞中的操作導致資料被覆蓋或被修改。
- 在decode模組中，指令資料經過FIFO (fetch_fifo) 暫存並分別傳遞至 u_dec0 和 u_dec1，FIFO的輸出是否正確需進一步確認。

2. 資料對齊錯誤：

- FIFO的 data0_out_o 和 data1_out_o 是從 ram_q 中提取不同區段的資料，這可能導致對齊錯誤。
- 特別是 data1_out_o 的分配方式 {1'd0, ram_q[rd_ptr_q][WIDTH-2:(WIDTH/2)]}，在某些情況下可能引入不必要的位元或導致資料失真。

3. 初始化問題：

- rst_i 信號初始化時，FIFO中的資料是否正確清零，可能導致殘留數據對後續的指令資料造成干擾。

4. 無效資料覆蓋有效資料：

- 當 fetch_in_fault_fetch_i 或 fetch_in_fault_page_i 設定為1時，fetch_in_instr_w 被設為零 (64'b0)。如果此時有效的指令資料被錯誤標記為無效，會導致pipeline處理的指令資料異常。



```
assign data0_out_o = {ram_q[rd_ptr_q][(WIDTH/2)-1:0]};  
assign data1_out_o = {1'd0, ram_q[rd_ptr_q][WIDTH-2:(WIDTH/2)]};
```

圖 10、GPT response results

這是 GPT 生成的回應，照他的回應去看 design 確實有發現有一個 bit 在指令傳遞時被錯誤給予永遠為 0，而非指令原本的值。

6. Steps to Resolve the Bug

```
assign data0_out_o = {ram_q[rd_ptr_q][(WIDTH/2)-1:0]};  
assign data1_out_o = {1'd0, ram_q[rd_ptr_q][WIDTH-2:(WIDTH/2)]};
```

圖 11、修改前指令傳遞之訊號

```
assign data0_out_o = {ram_q[rd_ptr_q][(WIDTH/2)-1:0]};  
assign data1_out_o = {ram_q[rd_ptr_q][WIDTH-1:(WIDTH/2)]};
```

圖 12、修改後指令傳遞之訊號

因為在指令傳遞下去會發生上述的錯誤，因此我們的解法是把他給成正確

值，就沒有問題了。

7. Result

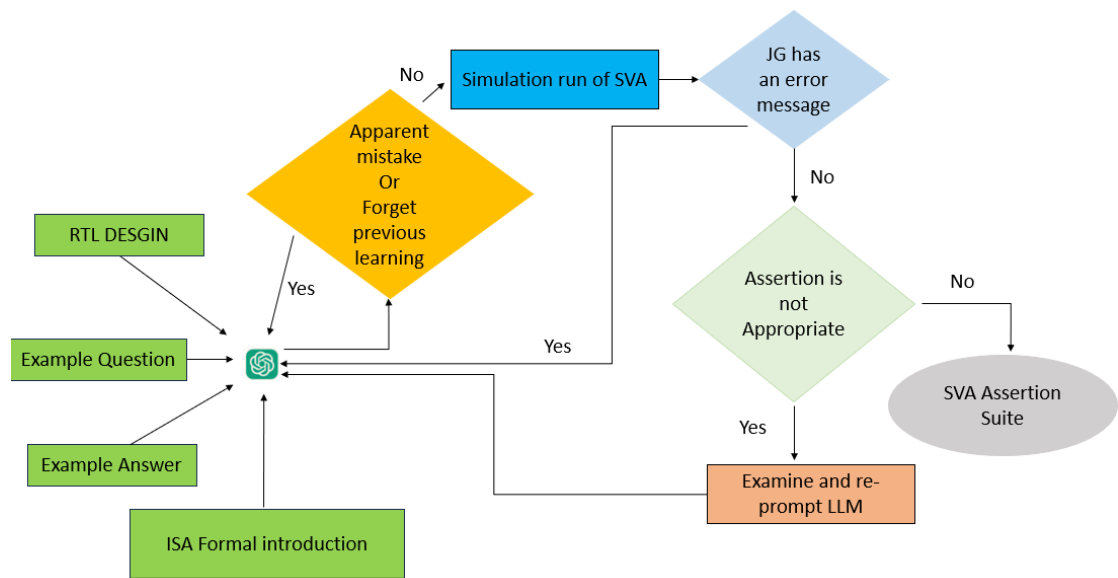


圖 13、Training an LLM to create SVA flowcharts

Prop	?	Assert	riscv_top.formaltest_biriscv.LW_insn0_check	Bm	12 -	0	279.5	<embedded>
	✓	Cover (related)	riscv_top.formaltest_biriscv.LW_insn0_check:precondition1	Ht	8	1	0.3	<embedded>
Covergroups	?	Assert	riscv_top.formaltest_biriscv.LW_insn1_check	Bm	12 -	0	343.1	<embedded>
	✓	Cover (related)	riscv_top.formaltest_biriscv.LW_insn1_check:precondition1	Ht	8	1	0.4	<embedded>
	?	Assert	riscv_top.formaltest_biriscv.AUIPC_insn0_check	B	12 -	0	9.2	<embedded>
	✓	Cover (related)	riscv_top.formaltest_biriscv.AUIPC_insn0_check:precondition1	Ht	8	1	0.4	<embedded>
	?	Assert	riscv_top.formaltest_biriscv.AUIPC_insn1_check	Bm	12 -	0	381.8	<embedded>
	✓	Cover (related)	riscv_top.formaltest_biriscv.AUIPC_insn1_check:precondition1	Ht	8	1	0.4	<embedded>
	?	Assert	riscv_top.formaltest_biriscv.SRAI_insn0_check	Ht	12 -	0	358.6	<embedded>
	✓	Cover (related)	riscv_top.formaltest_biriscv.SRAI_insn0_check:precondition1	Ht	8	1	0.4	<embedded>
	?	Assert	riscv_top.formaltest_biriscv.SRAI_insn1_check	Ht	12 -	0	567.0	<embedded>
	✓	Cover (related)	riscv_top.formaltest_biriscv.SRAI_insn1_check:precondition1	Ht	8	1	0.4	<embedded>
	?	Assert	riscv_top.formaltest_biriscv.JAL_PC_check0	K	15 -	0	43.1	<embedded>
	✓	Cover (related)	riscv_top.formaltest_biriscv.JAL_PC_check0:precondition1	L	9 - 13	1	1.4	<embedded>
	?	Assert	riscv_top.formaltest_biriscv.JAL_PC_check1	B	13 -	0	8.4	<embedded>
	✓	Cover (related)	riscv_top.formaltest_biriscv.JAL_PC_check1:precondition1	B	9	1	0.9	<embedded>
	?	Assert	riscv_top.formaltest_biriscv.JAL_rd_check0	B	12 -	0	7.9	<embedded>
	✓	Cover (related)	riscv_top.formaltest_biriscv.JAL_rd_check0:precondition1	Ht	8	1	0.4	<embedded>
	?	Assert	riscv_top.formaltest_biriscv.JAL_rd_check1	Ht	12 -	0	571.1	<embedded>
	✓	Cover (related)	riscv_top.formaltest_biriscv.JAL_rd_check1:precondition1	Ht	8	1	0.4	<embedded>
	?	Assert	riscv_top.formaltest_biriscv.assert_bltu_pipe0_taken	Ht	12 -	0	805.1	<embedded>
	✓	Cover (related)	riscv_top.formaltest_biriscv.assert_bltu_pipe0_taken:precondition1	L	10 - 31	1	79.9	<embedded>
	?	Assert	riscv_top.formaltest_biriscv.assert_bltu_pipe0_not_taken	Ht	11 -	0	160.2	<embedded>
	✓	Cover (related)	riscv_top.formaltest_biriscv.assert_bltu_pipe0_not_taken:precondition1	Bm	9	1	12.8	<embedded>
	?	Assert	riscv_top.formaltest_biriscv.assert_bltu_pipe1_taken	Ht	11 -	0	179.6	<embedded>
	✓	Cover (related)	riscv_top.formaltest_biriscv.assert_bltu_pipe1_taken:precondition1	L	10 - 324	1	61.3	<embedded>
	?	Assert	riscv_top.formaltest_biriscv.assert_bltu_pipe1_not_taken	Ht	11 -	0	211.8	<embedded>
	✓	Cover (related)	riscv_top.formaltest_biriscv.assert_bltu_pipe1_not_taken:precondition1	Bm	9	1	12.9	<embedded>

圖 14、sva 驗證結果

經過本次 project2 的練習，我們整理出與 LLM 下 prompt 生成 SVA 的流程，一開始先輸入 RTL design, Example questions, Example answer，使其根據範例生成 SVA，接著觀察有無明顯錯誤進行修正，最後進行模擬，如有錯誤就將 JG 產生

的 error message 或波形回饋給 LLM 進行錯誤分析及更正。