Project1 RV12 ISA Formal

組別: Group23

組員: M16131111 童品綸、N26134992 賴郁明

內容

1. Design Architecture

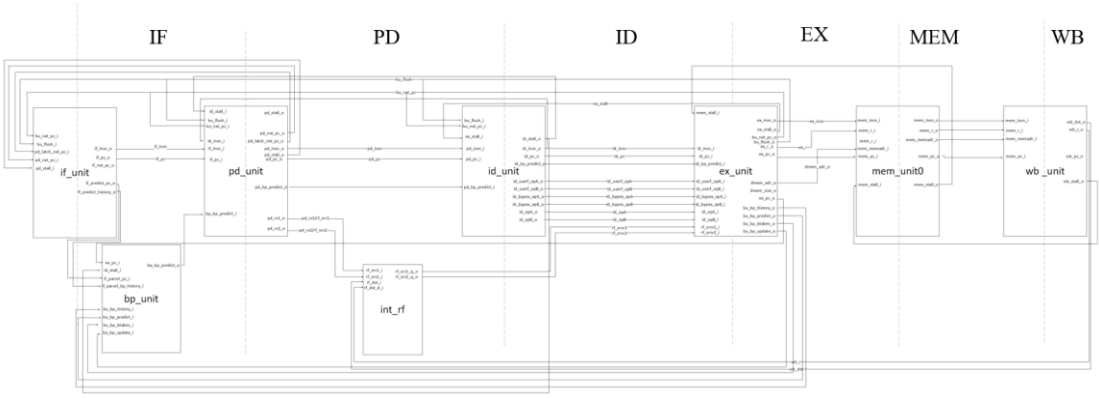

圖 1、CPU core architecture

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 if_unit 負責進行抓取指令，並在發生分支或跳躍時正確的切換下個 PC 位址。pd_unit 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，接著在 ID stage 進行指令之其他解碼並且產生控制訊號，在 EX stage 進行指令之算術邏輯運算以及判斷分支是否成立，如果分支成立結果不符合一開始的預測就會進行 flush，將前面的已經進入流水線之指令清空成 bubble 並且重新抓回原本該執行之指令，MEM 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

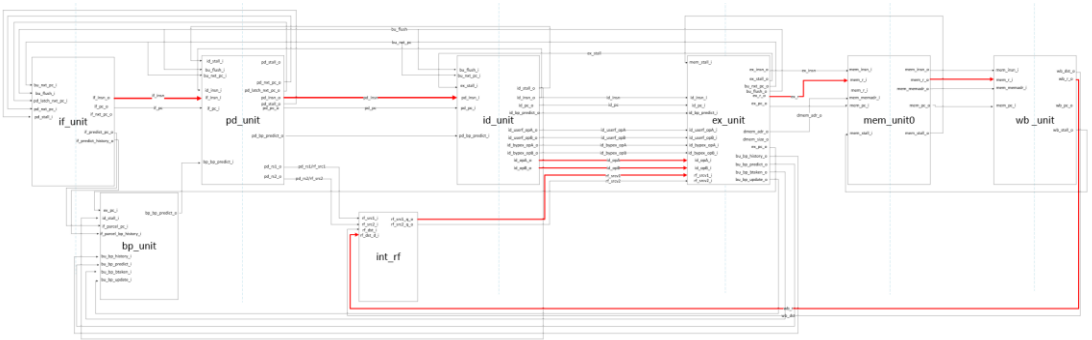

圖 2、SRAI datapath

SRAI 指令經過 if, pd, id 解碼該指令獲得立即值，在 ex 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

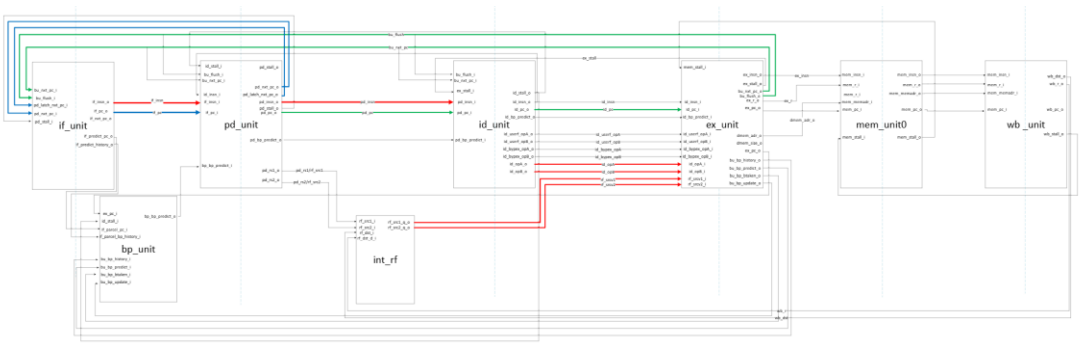

圖 3、BLTU datapath

BLTU 指令在一開始會先由於分支預測先猜測出可能結果，在 ex 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

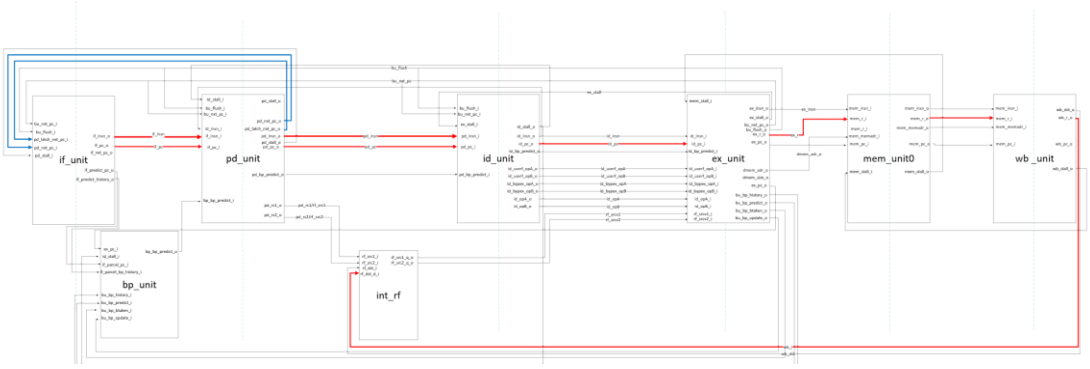

圖 4、JAL datapath

JAL 指令會在 pd 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

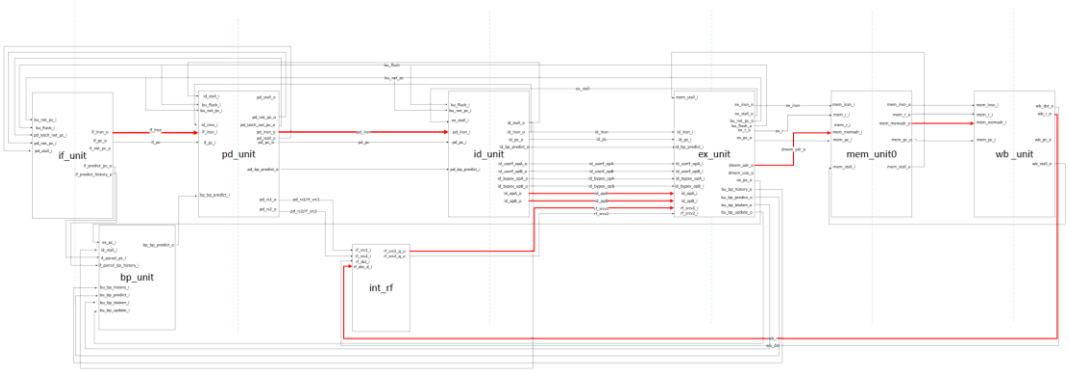

圖 5、LW datapath

LW 指令會在 ex 算出應該去哪個記憶體位置載入資料，並且在 wb 時將要載入資料寫回暫存器檔案。

2.5 AUIPC

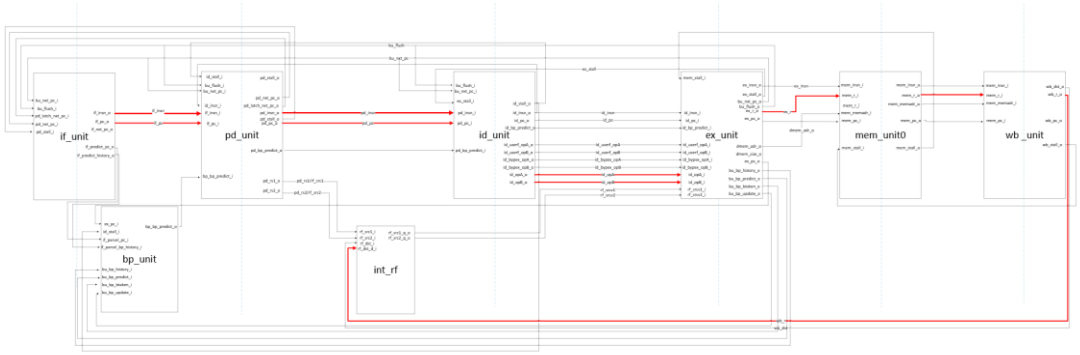

圖 6、AUIPC datapath

AUIPC 指令會在經過 if, pd, id 解碼該指令獲得立即值，在 ex 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

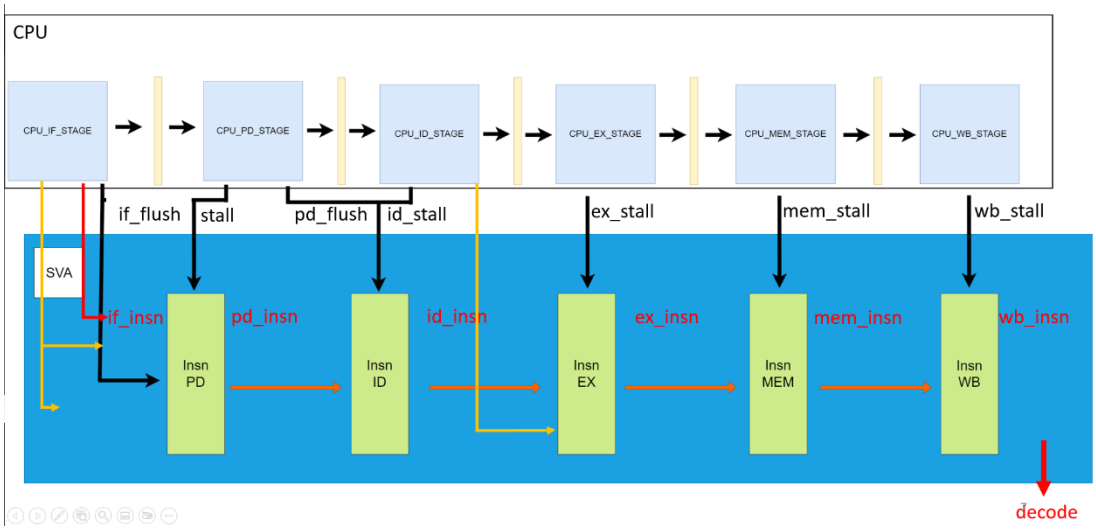

圖 7、pipeline follower

在進行 ISA formal 驗證時，我們會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 stall, flush 控制信號，使 pipeline follower 各個 stage 中的指令與 CPU 同步。

並在最後將指令傳到 wb stage 時對 wb stage 的指令進行解碼。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。

此外我們也將 branch_predict 及判斷指令是否為 bubble 的訊號傳到 wb stage 作為後續 branch 指令驗證的條件。

4. SVA

SRAI assertion:

```
355   // SRAI assertion
356   SRAI_check: assert property(@(posedge HCLK) disable iff(!HRESETn) (is_SRAI_insn |=> golden_SRAI_reg == result_SRAI));
```

SRAI 指令是否正確是透過 Write back stage 的下個 cycle 拿應該要寫進暫存器的資料跟已經寫入暫存器的資料作比對，看答案是否一樣來驗證指令是否被正確完成。

JAL assertion:

```
358 // JAL assertion
359   JAL_PC_check: assert property(@(posedge HCLK) disable iff(!HRESETn) (is_JAL_insn_PC |-> golden_PC_JAL == core.if_unit.pd_nxt_pc_i));
360   JAL_rd_check: assert property(@(posedge HCLK) disable iff(!HRESETn) (is_JAL_insn |=> golden_rd_JAL_reg == result_rd_JAL));
```

JAL 指令是否正確是透過判斷其跳躍後地址是否跟應該跳躍之地址相同，以及存回暫存器之原本地址跟已經寫入暫存器之地址是否相同來驗證指令是否被正確完成。

LW assertion:

```
362 // LW assertion
363   LW_check: assert property(@(posedge HCLK) disable iff(!HRESETn) (is_LW_insn |-> golden_LW == result_LW_delay));
```

LW 指令是否正確是透過判斷 MEM stage 去 memory load data 之 address 是否跟該指令應該要去 load data 之 address 相同。

AUIPC assertion:

```
358 // JAL assertion
359   JAL_PC_check: assert property(@(posedge HCLK) disable iff(!HRESETn) (is_JAL_insn_PC |-> golden_PC_JAL == core.if_unit.pd_nxt_pc_i));
360   JAL_rd_check: assert property(@(posedge HCLK) disable iff(!HRESETn) (is_JAL_insn |=> golden_rd_JAL_reg == result_rd_JAL));
```

AUIPC 指令是否正確是透過 Write back stage 的下個 cycle 拿應該要寫進暫存器的資料跟已經寫入暫存器的資料作比對，看答案是否一樣來驗證指令是否被正確完成。

BLTU assertion:

```
349   // BLTU assertion
350   BLTU_jump_prediction_success    : assert property ( @(posedge HCLK) disable iff(~HRESETn) ((is_BLTU_insn && BLTU_state==3'b001 && wb_no_bubble && ex_no_bubble) |-> (jump_prediction_success_pc == core.ex_units.ex_pc_o));
351   BLTU_no_jump_prediction_success : assert property ( @(posedge HCLK) disable iff(~HRESETn) ((is_BLTU_insn && BLTU_state==3'b010 && wb_no_bubble && mem_no_bubble) |-> (no_jump_prediction_success_pc==core.mem_unit0.mem_pc_o)));
352   BLTU_jump_prediction_fail       : assert property ( @(posedge HCLK) disable iff(~HRESETn) ((is_BLTU_insn && BLTU_state==3'b011 && wb_no_bubble && pd_no_bubble) |-> jump_prediction_fail_pc == core.pd_unit.pd_pc_o);
353   BLTU_no_jump_prediction_fail    : assert property ( @(posedge HCLK) disable iff(~HRESETn) ((is_BLTU_insn && BLTU_state==3'b100 && wb_no_bubble && mem_no_bubble) |-> no_jump_prediction_fail_pc==core.mem_unit0.mem_pc_o));
```

因為 branch 在執行時會因為預測是否成功以及是否有進行跳躍而產生不同結果，因此 BLTU SVA 在驗證時，我們把它分成 4 種清況來進行判斷，分別為

預測要進行分支跳躍且預測成功、預測不跳且預測成功，預測要跳且預測失敗，預測不跳且預測失敗，

先根據 pipeline follower 中 wb stage 指令解碼出的 opcode 及 funct3 來判斷是否為 BLTU 指令，再根據 rs1, rs2 data 來判斷 BLTU 是否有進行分支跳躍，並根據 wb_branch_prediction，來判斷是屬於哪種情況 BLTU 指令，接著分四種情況進行 assertion 驗證。

在 assertion 中先判斷何時要進行 BLTU 的驗證，透過不同的 BLTU_state 來觸發不同情況的 assertion，以及在驗證時要確定該指令不是 bubble 狀態，以及要進行比對的下一個指令(在 mem stage)也不是 bubble 狀態，因此在判斷的條件上要加上 wb_no_bubble 及 mem_no_bubble ，在 case2、4 時分別為沒有進行分支跳躍且預測成功及預測失敗，在這兩個情況下，BLTU 指令後會接著做，因此我們取該指令的下一個指令也就是在 mem stage 的 PC 來進行比對。

因為在 case1 發生時可以看到 mem stage 還會是原先的 pc+4 且此時的 mem bubble 為 1 此是跳躍後的 pc 會在 ex stage。在 assertion 中先判斷何時要進行 BLTU 的驗證，透過不同的 BLTU_state 來觸發不同情況的 assertion，以及在驗證時要確定該指令不是 bubble 狀態，以及要進行比對的下一個指令(在 mem stage)也不是 bubble 狀態，因此在判斷的條件上要加上 wb_no_bubble 及 mem_no_bubble ，在 case3 時分別為有進行分支跳躍且預測失敗，在這個情況下，由於 bu_flush 會將錯誤的跳躍位置洗掉，因此我們取該指令的下一個指令也就是在 mem stage 的 PC 來進行比對。由於預測結果錯誤，需要在 ex stage 時將 bu_flush 設為 1 此時會將 if_predict_pc_o 設為 bu_nxt_pc 也就是分支跳躍後的 PC，給到三個地方，其中一個是現在的 if_predicted_pc_o，我們只要在 wb 時找出當前 bu_nxt_pc_o 在哪找他來進行比較就能確認他預測失敗後的補救措施有沒有做到對，然後由於上面有說 bu_nxt_pc 在清洗當下會在 if_predicted_pc_o，所以理論上來說只要在後面兩個 cyclec 時抓當前 bu_nxt_pc 所在位置即可確認，但因為中間 mem 跟 pd 的 stage 會差一個 stall，所以實際上是要在三個 stage 後抓才會是當前 bu_nxt_pc 所在位置，也就是在指令在 WB 時要抓 pd 的 pc 來驗證。

5. Bugs Detection

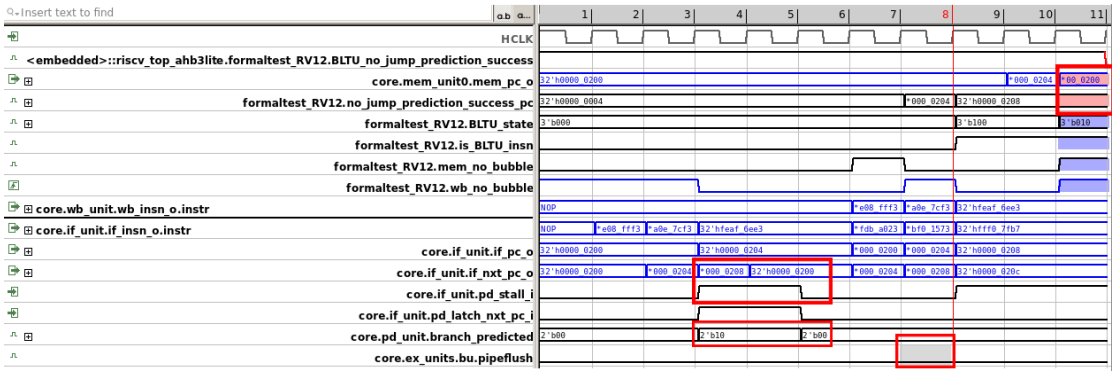

圖 8、bug waveform

branch 指令在 if stage 時，若發生 pd_stall，該 branch 指令停留在 if stage 多個週期，若在 pd_stall 期間 branch_predicted[1]為 1，代表預測分支跳躍，會將 if_nxt_pc 更新為跳躍後指令，但在 pd_stall 結束後，此時 branch 指令仍在 if stage 中，但 branch_predicted[1]會被設為 0。導致雖有發生跳躍行為，但 branch_predicted[1]被錯誤更新為 0，最終導致在 ex stage 的 pipeflush 信號會錯誤判斷在 pd stage 的跳躍狀態，而無法正確進行 branch 預測失敗的修正，導致我們在 wb stage 進行 branch 指令驗證時，goldenPC 為 pc+4，但 mem_pc 為跳躍後指令。因此導致 branch 指令的錯誤。

6. Steps to Resolve the Bug

original:

```
// Branch and Jump prediction
always_comb
  casex ( {du_mode_i, if_insn_i.bubble, decode_opcode(if_insn_i.instr)} )
    {1'b0,1'b0,OPC_JAL   } : begin
                                 branch_taken     = 1'b1;
                                 branch_predicted = 2'b10;
                                 rsb_push         = decode_rsb_push;
                                 rsb_pop          = decode_rsb_pop;
                                 pd_nxt_pc_o      = if_pc_i + ext_immUJ;
                             end

    {1'b0,1'b0,OPC_JALR  } : begin
                                 branch_taken     = has_rsb ? decode_rsb_pop : 1'b0;
                                 branch_predicted = 2'b00;
                                 rsb_push         = decode_rsb_push;
                                 rsb_pop          = decode_rsb_pop;
                                 pd_nxt_pc_o      = rsb_predict_pc;
                             end

    {1'b0,1'b0,OPC_BRANCH} : begin
                                 //if this CPU has a Branch Predict Unit, then use it's prediction
                                 //otherwise assume backwards jumps taken, forward jumps not taken
                                 branch_taken     = (HAS_BPU != 0) ? bp_bp_predict_i[1] : ext_immSB[31];
                                 branch_predicted = (HAS_BPU != 0) ? bp_bp_predict_i    : {ext_immSB[31], 1'b0};
                                 rsb_push         = 1'b0;
                                 rsb_pop          = 1'b0;
                                 pd_nxt_pc_o      = if_pc_i + ext_immSB;
                             end

    default                : begin
                                 branch_taken     = 1'b0;
                                 branch_predicted = 2'b00;
                                 rsb_push         = 1'b0;
                                 rsb_pop          = 1'b0;
                                 pd_nxt_pc_o      = 'hx;
                             end
  endcase
```

圖 9、修改前之分支預測結果連接訊號

- modified:

```
// Branch and Jump prediction
always_comb
  casex ( {du_mode_i, if_insn_i.bubble, decode_opcode(if_insn_i.instr)} )
    {1'b0,1'b0,OPC_JAL   } : begin
                                 branch_taken     = 1'b1;
                                 branch_predicted = 2'b10;
                                 rsb_push         = decode_rsb_push;
                                 rsb_pop          = decode_rsb_pop;
                                 pd_nxt_pc_o      = if_pc_i + ext_immUJ;
                             end

    {1'b0,1'b0,OPC_JALR  } : begin
                                 branch_taken     = has_rsb ? decode_rsb_pop : 1'b0;
                                 branch_predicted = 2'b00;
                                 rsb_push         = decode_rsb_push;
                                 rsb_pop          = decode_rsb_pop;
                                 pd_nxt_pc_o      = rsb_predict_pc;
                             end

    {1'b0,1'b0,OPC_BRANCH} : begin
                                 //if this CPU has a Branch Predict Unit, then use it's prediction
                                 //otherwise assume backwards jumps taken, forward jumps not taken
                                 branch_taken     = (pd_stall_o_delay)?  0: (HAS_BPU != 0) ? bp_bp_predict_i[1] : ext_immSB[31]; // modified
                                 branch_predicted = (pd_stall_o_delay)?  0: (HAS_BPU != 0) ? bp_bp_predict_i    : {ext_immSB[31], 1'b0};// modified
                                 rsb_push         = 1'b0;
                                 rsb_pop          = 1'b0;
                                 pd_nxt_pc_o      = if_pc_i + ext_immSB;
                             end

    default                : begin
                                 branch_taken     = 1'b0;
                                 branch_predicted = 2'b00;
                                 rsb_push         = 1'b0;
                                 rsb_pop          = 1'b0;
                                 pd_nxt_pc_o      = 'hx;
                             end
  endcase
```

圖 10、修改後之分支預測結果連接訊號

```
//Next Program Counter

//Combinatorial to Predictor predict_pc is registered by the memory address
//register. Then we can use registered output to reduce critical path
always_comb
  if      ( st_flush_i        ) if_predict_pc_o = st_nxt_pc_i    & ADR_MASK;
  else if ( du_we_pc_strb     ) if_predict_pc_o = du_dato_i      & ADR_MASK;
  else if ( bu_flush_i        ) if_predict_pc_o = bu_nxt_pc_i    & ADR_MASK;
  else if ( pd_latch_nxt_pc_i ) if_predict_pc_o = pd_nxt_pc_i    & ADR_MASK; //pd_flush absolutely breaks the CPU here
  else if (!pd_stall_i          &&
           !if_nxt_insn_o.bubble &&
           !du_stall_i)
    if (is_16bit_instruction)   if_predict_pc_o = if_nxt_pc_o +2 & ADR_MASK;
    else                        if_predict_pc_o = if_nxt_pc_o +4 & ADR_MASK;
  else                          if_predict_pc_o = if_nxt_pc_o    & ADR_MASK;



always_comb
  if      ( st_flush_i        ) if_predict_pc_o = st_nxt_pc_i    & ADR_MASK;
  else if ( du_we_pc_strb     ) if_predict_pc_o = du_dato_i      & ADR_MASK;
  else if ( bu_flush_i        ) if_predict_pc_o = bu_nxt_pc_i    & ADR_MASK;
  else if ( pd_latch_nxt_pc_i & !pd_stall_i) if_predict_pc_o = pd_nxt_pc_i    & ADR_MASK; //pd_flush absolutely breaks the CPU here
  else if (!pd_stall_i          &&
           !if_nxt_insn_o.bubble &&
           !du_stall_i)
    if (is_16bit_instruction)   if_predict_pc_o = if_nxt_pc_o +2 & ADR_MASK;
    else                        if_predict_pc_o = if_nxt_pc_o +4 & ADR_MASK;
  else                          if_predict_pc_o = if_nxt_pc_o    & ADR_MASK;
```

圖 9、修改前後之分支預測結果所控制之訊號

因為在 pd_stall 時跳躍會發生上述的錯誤，因此我們的解法是先在 pd_stall 期間將 branch 預測結果都改成預測分支不發生，但由於直接改 branch predicted 邏輯太複雜，不好修改，因此我們將分支預測器接出來的訊號都強制改為執行分支預測不發生的行為。

7. Result

	Assume	riscv_top_ahb3lite.formaltest_RV12.no_16bit_insn	?		0	0.0	<embedded>	Analysis Session
	Assert	riscv_top_ahb3lite.formaltest_RV12.BLTU_jump_prediction_success	Hp	15 -	0	5769.7	<embedded>	Analysis Session
	Cover (related)	riscv_top_ahb3lite.formaltest_RV12.BLTU_jump_prediction_success:precondition1	Ht	8	1	1.9	<embedded>	Analysis Session
	Assert	riscv_top_ahb3lite.formaltest_RV12.BLTU_no_jump_prediction_success	Ncustom6...	Infinite	0	206.7	<embedded>	Analysis Session
	Cover (related)	riscv_top_ahb3lite.formaltest_RV12.BLTU_no_jump_prediction_success:precondition1	Ht	8	1	2.0	<embedded>	Analysis Session
	Assert	riscv_top_ahb3lite.formaltest_RV12.BLTU_jump_prediction_fail	Ht	15 -	0	5147.9	<embedded>	Analysis Session
	Cover (related)	riscv_top_ahb3lite.formaltest_RV12.BLTU_jump_prediction_fail:precondition1	L	10 - 15	1	279.7	<embedded>	Analysis Session
	Assert	riscv_top_ahb3lite.formaltest_RV12.BLTU_no_jump_prediction_fail	Mpcusto...	Infinite	0	2556.6	<embedded>	Analysis Session
	Cover (related)	riscv_top_ahb3lite.formaltest_RV12.BLTU_no_jump_prediction_fail:precondition1	B	11	1	13.0	<embedded>	Analysis Session
	Assert	riscv_top_ahb3lite.formaltest_RV12.SRAI_check	Ht	15 -	0	2044.0	<embedded>	Analysis Session
	Cover (related)	riscv_top_ahb3lite.formaltest_RV12.SRAI_check:precondition1	Ht	8	1	1.6	<embedded>	Analysis Session
	Assert	riscv_top_ahb3lite.formaltest_RV12.JAL_PC_check	Cache	Infinite	0	0.0	<embedded>	Analysis Session
	Cover (related)	riscv_top_ahb3lite.formaltest_RV12.JAL_PC_check:precondition1	N	3	1	0.9	<embedded>	Analysis Session
	Assert	riscv_top_ahb3lite.formaltest_RV12.JAL_rd_check	Mpcusto...	Infinite	0	2659.0	<embedded>	Analysis Session
	Cover (related)	riscv_top_ahb3lite.formaltest_RV12.JAL_rd_check:precondition1	Ht	8	1	2.1	<embedded>	Analysis Session
	Assert	riscv_top_ahb3lite.formaltest_RV12.LW_check	Bm	19 -	0	4555.0	<embedded>	Analysis Session
	Cover (related)	riscv_top_ahb3lite.formaltest_RV12.LW_check:precondition1	Ht	12	1	6.8	<embedded>	Analysis Session
	Assert	riscv_top_ahb3lite.formaltest_RV12.AUIPC_check	Bm	21 -	0	2147.8	<embedded>	Analysis Session
	Cover (related)	riscv_top_ahb3lite.formaltest_RV12.AUIPC_check:precondition1	Ht	8	1	1.8	<embedded>	Analysis Session
	Assume	assume:0	?		0	0.0	<embedded>	Analysis Session

圖 12、sva 驗證結果

這次 CPU 的控制訊號及複雜度相比過去計組所學之 CPU 高很多，特別是在分支預測器控制訊號的部分，然後我們 Server 等級不夠，導致就算只限制合法指令輸入，都很容易造成指令驗到一半就爆記憶體導致無法驗完，這次最大的難題還有是 debug 的部分，特別是在時間有限而且 CPU 不是自己製作時，有

bug 往往很難分清楚其根本之錯誤邏輯，就算有 jaspergold 可以輔助，但要修改時也很難知道修改會不會在間接影響其他模塊，因此我們的修改方式最終採取間接修改，在其控制訊號有錯誤之地方不直接修改產生控制訊號之邏輯，而是只對他錯誤控制傳遞邏輯進行修改。