Project1_Group23
Report

Group Member : 陳語彤、陳俐蓉、吳念泓

Outline

I. Pipeline Follower

II. Assumption for All Instruction

III. Assertion Design for Verifying ISA

   a) ANDI

   b) AUIPC

   c) LBU

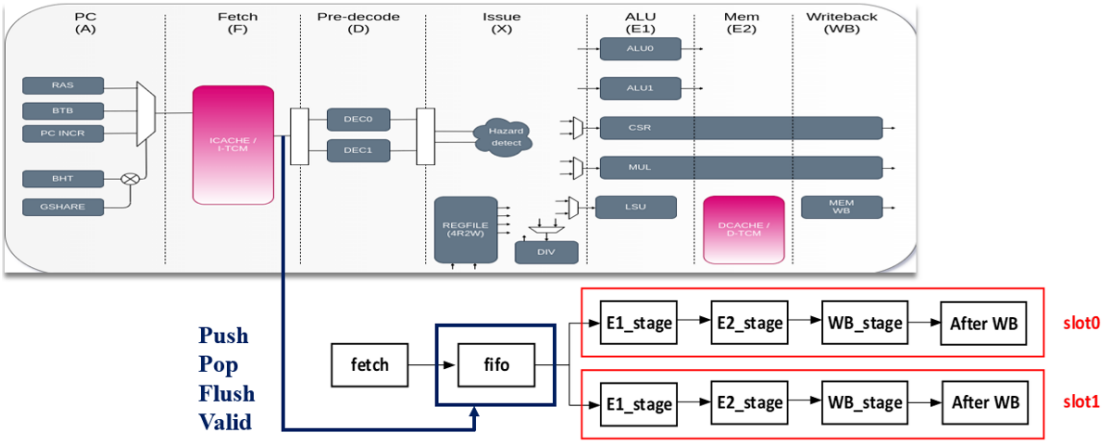
   d) BGE

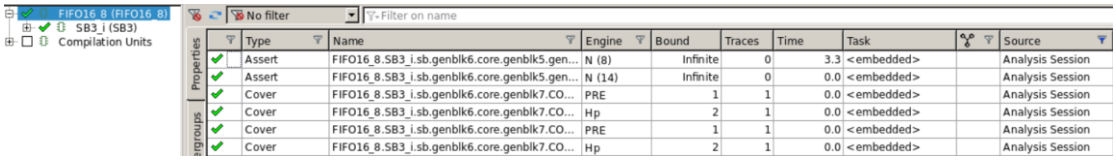
IV. Trojan

V. Other Trojan

VI. Work Distribution Chart

一、 Pipeline Follower





Type	Name	Engine	Bound	Traces	Time	Task	Source
Assert	FIFO16_8.SB3_i.sb.genblk6.core.genblk5.gen...	N (8)	Infinite	0	3.3	<embedded>	Analysis Session
Assert	FIFO16_8.SB3_i.sb.genblk6.core.genblk5.gen...	N (14)	Infinite	0	0.0	<embedded>	Analysis Session
Cover	FIFO16_8.SB3_i.sb.genblk6.core.genblk7.CO...	PRE	1	1	0.0	<embedded>	Analysis Session
Cover	FIFO16_8.SB3_i.sb.genblk6.core.genblk7.CO...	Hp	2	1	0.0	<embedded>	Analysis Session
Cover	FIFO16_8.SB3_i.sb.genblk6.core.genblk7.CO...	PRE	1	1	0.0	<embedded>	Analysis Session
Cover	FIFO16_8.SB3_i.sb.genblk6.core.genblk7.CO...	Hp	2	1	0.0	<embedded>	Analysis Session

```
always @ (posedge clk or posedge rst)
if (rst)
begin
    wr_ptr <= 1'd0;
    r_ptr <= 1'd0;
    count <= 2'd0;
    for (i = 0; i < 2; i = i + 1)
    begin
        ram[i] <= 64'd0;
    end
end
else if(u_core.u_frontend.u_decode.genblk1.u_fifo.flush_i)
begin
  wr_ptr <= 1'd0;
  r_ptr <= 1'd0;
  count <= 2'd0;
  for (i = 0; i < 2; i = i + 1)
  begin
      ram[i] <= 64'd0;
  end
end
```

首先要先說明在這個 CPU 中的 fifo 深度為 2，在驗證 fifo 的時候因為 CPU 會有 flush 訊號，將 fifo 中的值全部清為 0，導致我們如果將此 fifo 掛上 scoreboard 會出現錯誤，所以我就寫一個 fifo 並且通過 scoreboard 的驗證(結果在上圖)，而為了符合 CPU 的訊號我在這個驗證完後的 fifo 加上 flush 訊號，他做的事情其實就跟 reset 一樣，將所有的訊號做歸 0 的動作同時也將 fifo 中的指令全部清為 0。

```
else
begin
    if (write_valid)
    begin
        ram[wr_ptr] <= ins_64bit;
        wr_ptr <= (wr_ptr==1'd1) ? 1'd0 : wr_ptr + 1'd1;
    end
    else
    begin
      wr_ptr <= wr_ptr;
    end

    if(pop_finish)
    begin
        r_ptr <= (r_ptr==1'd1) ? 1'd0 : r_ptr + 1'd1;
    end
    else
    begin
        r_ptr <= r_ptr;
    end

    if(write_valid & ~pop_finish)
        count <= count + 2'd1;
    else if(~write_valid & pop_finish)
        count <= count -2'd1;
    else
        count <= count;
end
```

接下來是對 fifo 做 push(write valid)跟 pop 的動作，因為在驗證時我沒有多去判斷 fifo 是否空跟滿，所以到在 push 的時候也就是 write valid 我就會將資料寫入並且將 write pointer 加 1，我將原先 fifo 的一些判斷是否空跟滿的訊號拿掉，在下段會解釋我為什麼沒去判斷空跟滿我的結果是不會有問題的，接著 pop 的部分也一樣，所以可以看到在 pop 的時候我就會將資料讀出並且將 read pointer 加 1。

最後就是 counter 的部分，這個部分就是在算 fifo 裡面儲存了幾個的值，在要寫不讀的時候 count 會加 1，而在要讀不寫時 count 會減 1，其餘則維持原值。

```
always @(*)
begin
  if(rst)
  begin
      ins2 = 32'd0;
      ins3 = 32'd0;
  end
  else if(out0_valid & out1_valid)
  begin
      ins2 = ram[r_ptr][31:0];
      ins3 = ram[r_ptr][63:32];
  end
  else if(out0_valid)
  begin
      ins2 = ram[r_ptr][31:0];
      ins3 = 32'd0;
  end
  else if(out1_valid)
  begin
      ins2 = 32'd0;
      ins3 = ram[r_ptr][63:32];
  end
  else
  begin
    ins2 = 32'd0;
    ins3 = 32'd0;
  end
end
```

接者這是我將指令讀出的部分，在這裡我只將有效的指令讀出，因為在 CPU 執行的過程中只有有效的指令才會被執行，所以需要去驗證此條指令的正確性，反之 valid 為 0，指令不管為多少都不會影響 CPU 的執行，所以可以不用去驗證此條指令，在這邊我都將讀出指令給為 NOP。

```
fifo_data0 : assert property
(
  @(posedge clk)
  disable iff (rst)
  (out0_valid) |-> (ins2 == u_core.u_frontend.u_decode.fetch_out0_instr_o)
);

fifo_data1 : assert property
(
  @(posedge clk)
  disable iff (rst)
  (out1_valid) |-> (ins3 == u_core.u_frontend.u_decode.fetch_out1_instr_o)
);
```

接著就是我驗證所寫的 assertion，因為 CPU 的指令只有在 valid 時才是有效的，所以我在這邊只檢查會被執行的指令(也就是 valid 的指令)，首先我先驗證 Data Integrity，在 valid0 為 1 時的同個 cycle， ins2 的值要與 CPU fetch0 的值相等，這樣就能確定指令有被正確讀出，並沒有讀到錯誤的指令，或是指令在 fifo 中遺失的問題，而 valid1 也是一樣的操作。

```
fifo_w_ptr : assert property
(
  @(posedge clk)
  disable iff (rst)
  (write_valid) |=> (u_core.u_frontend.u_decode.genblk1.u_fifo.wr_ptr_q == w_idx)
);
```

```
always @(posedge clk)
begin
    w_idx <= u_core.u_frontend.u_decode.genblk1.u_fifo.wr_ptr_q + 1;
end
```

```
fifo_rd_ptr : assert property
(
  @(posedge clk)
  disable iff (rst)
  (pop_finish) |=> (u_core.u_frontend.u_decode.genblk1.u_fifo.rd_ptr_q == rd_idx)
);
```

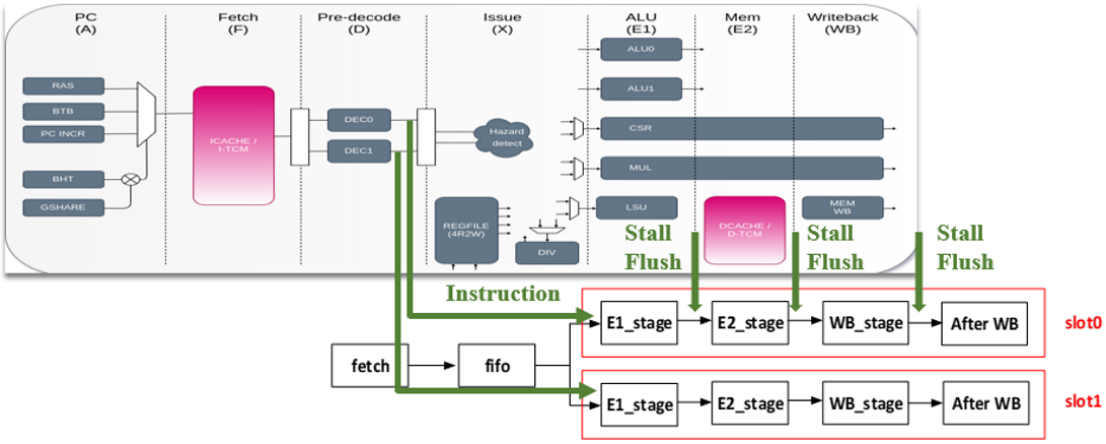
```
always @(posedge clk)
begin
    rd_idx <= u_core.u_frontend.u_decode.genblk1.u_fifo.rd_ptr_q + 1;
end
```

接著則是驗證 writer pointe 是否有在執行完 push 時的下一個 cycle 有正確的加 1，而其中 w_idx 是我用上一個 cycle 的 write pointer+1 的結果，在 write valid(也就是要 push)為 1 後，的下一個 cycle，CPU 的 write pointer 要與 w_idx 的值相同，代表 write pointer 有被正確加 1，而 read pointer 的做法與 write pointer 相同。

```
overflow : assert property
(
  @(posedge clk)
  disable iff (rst)
  (count==2'd2) |-> ~(write_valid)
);
```

```
underflow : assert property
(
  @(posedge clk)
  disable iff (rst)
  (count==2'd0) |-> ~(pop_finish)
);
```

接著是驗證 overflow 的部分，這裡可以確保我剛剛在 write data 時沒有判斷 fifo 是否滿了是沒問題的，因為在 count 為 2 時， write 就不會被拉起，所以能確保 fifo 不會 overflow(在滿的時候還要將資料寫入)，能夠放心在 write 拉起時就將資料寫入，最後則是驗證不會 underflow，這裡可以確保剛剛我在 pop 時沒有判斷 fifo 是否空的是沒問題的，因為在 count 為 0 時，pop 就不會被拉起，所以能確保 fifo 不會 underflow(在空的時候要去讀取資料) 。



在 Pipeline Follower 中，我們在 decode 後將指令與 pc 值存入 Pipeline Follower 中，並拉出每一級的 stall、flush 訊號，確保 Pipeline Follower 中的控制訊號與 CPU 執行是相同的。

但在 Pipeline Follower 中的 E1_stage，在這裡我們是抓出 stall 跟 no flush 的訊號，比較特別的地方是在 decode 後還未將指令分到 slot0 或 slot1，是在進入 E1_stage 才做分辨，而在 no flush 時才會進到這裡將指令分到 slot0 或 slot1 中。

接著是 E2_stage、WB stage 這兩個 stage 皆是抓出 stall、flush 訊號，如果有 stall 則指令就維持原值，而如果有 flush 則指另將會變成 NOP，最後就是 WB stage 的後一級，在這級中我們只傳了 rd 的值，因為我們要驗證結果是否寫回暫存器，只需要知道存入暫存器的 index，而指令中其他參數是沒有用到的，所以我們在這級當中只要 rd 是有效的，我們就會將 rd 的值傳送到下一級進行比對。

二、 Assumption for All Instruction

```
   (u_core.u_frontend.fetch_instr_w[14:12]==3'b000)) //jalr
||((u_core.u_frontend.fetch_instr_w[6:0]==7'b1100011) &&
```

☞ I-type

31 20	19 15	14 12	11 7	6 0		
imm[11:0]	rs1	funct3	rd	opcode	Mnemonic	Description
imm[11:0]	rs1	000	rd	1100111	JALR	rd = PC + 4 PC = imm + rs1 (Set LSB of PC to 0)

```
((u_core.u_frontend.fetch_instr_w[6:0]==7'b0000011) && //Btype
 ((u_core.u_frontend.fetch_instr_w[14:12]==3'b000) ||
  (u_core.u_frontend.fetch_instr_w[14:12]==3'b001) ||
  (u_core.u_frontend.fetch_instr_w[14:12]==3'b010)||
  (u_core.u_frontend.fetch_instr_w[14:12]==3'b100)||
  (u_core.u_frontend.fetch_instr_w[14:12]==3'b101)))
```

☞ B-type

31 25	24 20	19 15	14 12	11 7	6 0		
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	Mnemonic	Description
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ	PC = (rs1 == rs2)? PC + imm: PC + 4
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE	PC = (rs1 != rs2)? PC + imm: PC + 4
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT	PC = ($rs1_s$ < $rs2_s$)? PC + imm: PC + 4
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE	PC = ($rs1_s$ ≥ $rs2_s$)? PC + imm: PC + 4
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU	PC = ($rs1_u$ < $rs2_u$)? PC + imm: PC + 4
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU	PC = ($rs1_u$ ≥ $rs2_u$)? PC + imm: PC + 4

在 assumption 的部份我們只限制要輸入正確的指令，因為在沒有限制的情況下可能會輸入無效的指令，照成驗證的複雜性，接下來說明我們如何限制指令的輸入，我們用其中 2 個指令來說明，第 1 個是 jarl 指令,可以看到右邊 jarl 的指令架構，其中 instruction 的第 0 到第 6bit 為 opcode 與第 12 到第 14bit 是 funct3 這兩個皆是固定的，第 2 個是 b type 指令,可以看到右邊 b type 的指令架構，其中 instruction 的第 0 到第 6bit 為 opcode 與第 12 到第 14bit 是 funct3 這兩個皆是固定的，其中 funct3 又可以是 000 到 111 的值。

三、 Assertion Design for Verifying ISA

a) ANDI

```
logic anditri_pipe0;
assign anditri_pipe0 = (rvfi_ins4_pipe0[6:0] == 7'b0010011 &&
                        rvfi_ins4_pipe0[14:12] == 3'b111 &&
                        rvfi_ins4_pipe0[11:7] != 5'd0);
```

接著是 andi 指令，首先先說明指令的 trigger，其中 rvfi_ins4_pipeline 是 pipeline follower 傳到 wb stage 的指令，當這個指令的 opcode 為 0010011 及 funct3 為 111 時且 rd 不為 0，anditri 就會為 1。

```
andi_pipe0_data_reg : assert property
(
  @(posedge clk)
  disable iff (rst)
  (anditri_pipe0 & valid_rd_pipe0) |=> (gold_andi_pipe0_WB[0] == RF[rd_after_wb0][0])
);
```

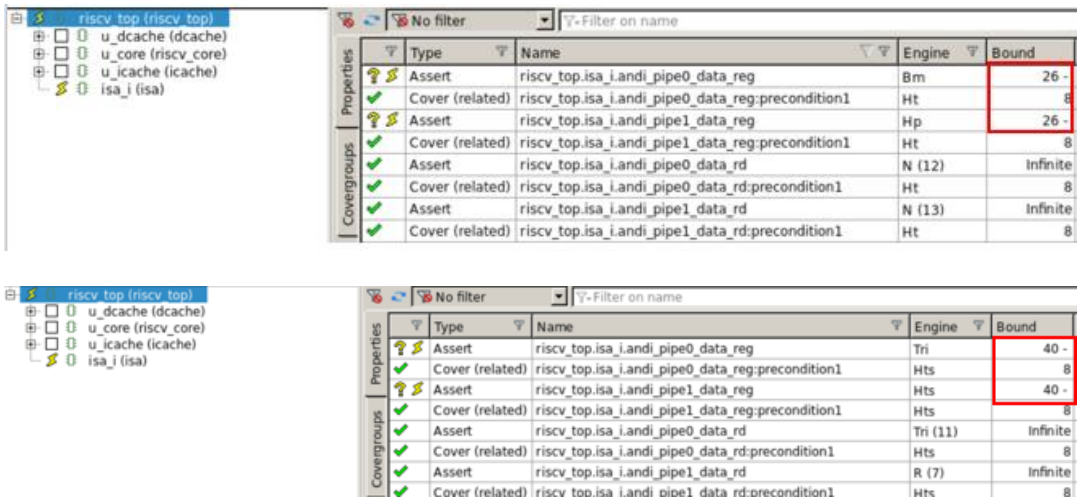
```
always @(posedge clk)
begin
    gold_andi_pipe0_WB <= RF[rvfi_ins4_pipe0[19:15]] & andi_pipe0_imm;
end
```

接下來說明我們如何驗證資料的正確性，在這次的驗證中我們選擇驗證 wb stage 的下一級，因為在此時資料已經被寫回暫存器了，所以我們只需驗證暫存器的值，就可以確保 alu 運算結果是正確的也可以得知結果正確寫回暫存器，那觸發這個 assertion 的前提是 andi 指令被 trigger 加上 rd 是有效的，因如果 rd 值為 0 就不需確認暫存器中的值了，那在被觸發後的下級暫存器中的值要與 golden 的值相同，Golden 值的算法在上圖 2。

```
andi_pipe0_data_rd : assert property
(
  @(posedge clk)
  disable iff (rst)
  (anditri_pipe0 & valid_rd_pipe0) |=> (rd_pipe0_ins5 == rd_after_wb0)
);
```

```
always @(posedge clk)
begin
    rd_after_wb0 <= u_core.u_issue.u_regfile.rd0_i;
end
```

接著則是驗證在 wb stage 的下一級的 rd 是否與 pipeline follower 中的 rd 是相同的，這個 assertion 在 andi 指令 trigger 與 rd 值是有效的時候才會被觸發(而會在 wb stage 被觸發)，所以在被觸發的下一個 cycle 驗證 rd 是否正確。



這裡說明我們如何推進驗證的 bound 數量，在這裡可以看到在 rd 的驗證已經跑完，但 data 的驗證只在 bound=26 的地方，所以我們將原先驗證 32bit 改成驗證 5bit，就是驗證暫存器中的最低 5bit 是否與 golden 最後 5bit 相同，這時 bound 就可以跑到 40 的地方，後來我們也有嘗試只驗證 1bit 但推進的效果沒有很明顯。

b) AUIPC

```
logic auipctri_pipe0;
assign auipctri_pipe0 = (rvfi_ins4_pipe0[6:0] == 7'b0010111 &&
                         rvfi_ins4_pipe0[11:7] != 5'd0);
```

接著是 auipc 指令，首先先說明指令的 trigger，其中 rvfi_ins4_pipeline 是 pipeline follower 傳到 wb stage 的指令，當這個指令的 opcode 為 0010111 及 rd 不為 0，auipctri 就會為 1。

```
auipc_pipe0_data_reg : assert property
(
  @(posedge clk)
  disable iff (rst)
  (auipctri_pipe0 & valid_rd_pipe0) |=> (gold_auipc_pipe0_WB[4:0] == RF[rd_after_wb0][4:0])
);
```

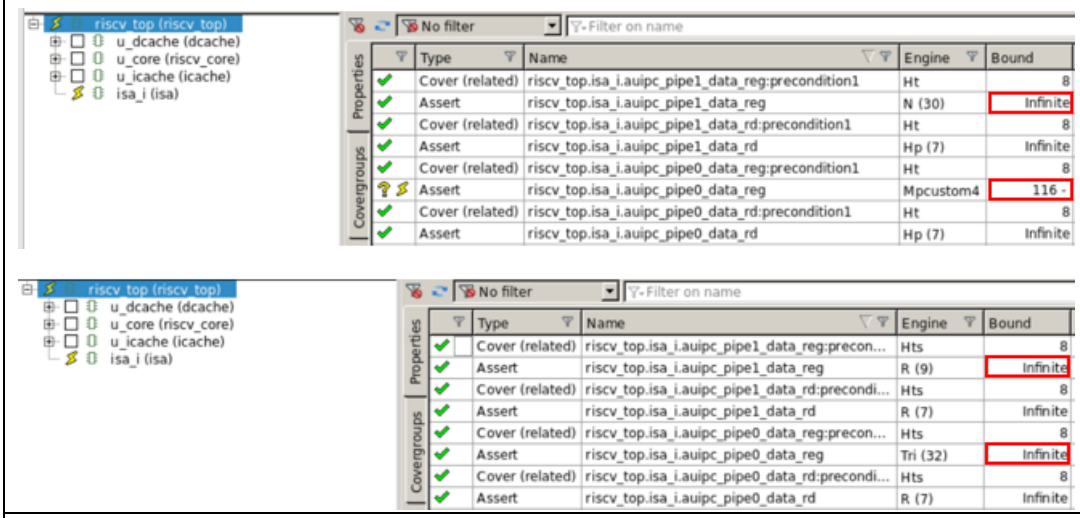
```
always @(posedge clk)
begin
    gold_auipc_pipe0_WB <= rvfi_pc4_pipe0 + auipc_pipe0_imm;
end
```

接下來說明我們如何驗證資料的正確性，在這次的驗證中我們選擇驗證 wb stage 的下一級，因為在此時資料已經被寫回暫存器了，所以我們只需驗證暫存器的值，就可以確保 alu 運算結果是正確的也可以得知結果正確寫回暫存器，那觸發這個 assertion 的前提是 auipc 指令被 trigger 加上 rd 是有效的，因如果 rd 值為 0 就不需確認暫存器中的值了，那在被觸發後的下級暫存器中的值要與 golden 的值相同，Golden 值的算法在上圖 2。

```
andi_pipe0_data_rd : assert property
(
  @(posedge clk)
  disable iff (rst)
  (anditri_pipe0 & valid_rd_pipe0) |=> (rd_pipe0_ins5 == rd_after_wb0)
);
```

```
always @(posedge clk)
begin
    rd_after_wb0 <= u_core.u_issue.u_regfile.rd0_i;
end
```

接著則是驗證在 wb stage 的下一級的 rd 是否與 pipeline follower 中的 rd 是相同的，這個 assertion 在 auipc 指令 trigger 與 rd 值是有效的時候才會被觸發(而會在 wb stage 被觸發)，所以在被觸發的下一個 cycle 驗證 rd 是否正確。



這裡說明我們如何推進驗證的 bound 數量，在這裡可以看到在 rd 的驗證已經跑完，但 data 的驗證只在 bound=116 的地方，所以我們將原先驗證 32bit 改成驗證 5bit，就是驗證暫存器中的最低 5bit 是否與 golden 最後 5bit 相同，這時 bound 就可以跑到無限的地方，雖然這樣無法保證整個值是對的，但至少可以確定值的後 5bit 在 CPU 執行過程中是不會出錯的。

c) LBU

Trigger

```
assign lbutri_pipe0 = (rvfi_ins4_pipe0[6:0] == 7'b0000011 &&
                       rvfi_ins4_pipe0[14:12] == 3'b100 &&
                       rvfi_ins4_pipe0[11:7] != 5'd0 &&
                       u_core.u_issue.u_pipe0_ctrl.valid_wb_o
                       );
```

Trigger 可用來來判斷是否需要檢查這條指令，需要符合 2 個條件 Trigger 才會拉起:

1. 在 WB stage 時會判斷 pipeline follow 裡的指令是不是 LBU，會檢查 opcode 和 function3 是不是 LBU 的
2. DUV 的 write back valid 拉起，代表此時的 data 是會被寫回 register file 的

Property1: 檢查送進 memory 的 address 是否正確

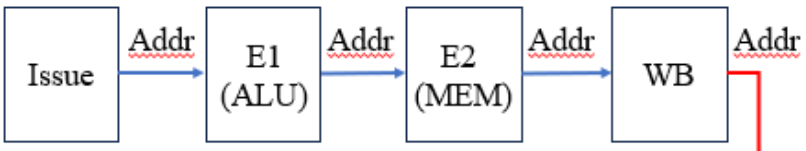
1. 在 sva 中計算 golden address

   會在 LBU 到達 WB stage 時把 register file 裡的值和 immediate 相加產生 golden address，因為此時前面指令的結果已經寫回 register file，這時候拿到的值一定是對的，不會有 forwarding 的問題。

```
assign  rvfi_rs1_pipe0 = rvfi_ins4_pipe0[19:15];
assign  rvfi_imm_pipe0 = (rvfi_ins4_pipe0[31])? {20'hfffff,rvfi_ins4_pipe0[31:20]} :
                                                {20'h00000,rvfi_ins4_pipe0[31:20]};
assign  gold_addr_pipe0 = RF[rvfi_rs1_pipe0] + rvfi_imm_pipe0;
```

2. 把 DUV 計算的 address 跟著 pipeline follow 傳到 WB，在 trigger 拉起時跟 golden address 做比對

   這裡有用到抽象化的概念，只取 2 個 bit 來檢查，加快驗證速度。




```
check_Addr_pipe0 : assert property
(
    @(posedge clk)
    disable iff (rst)
    lbutri_pipe0 |-> (reg_Addr_WB_pipe0[16:15] == gold_addr_pipe0[16:15])
);
```

3. 驗證結果

   雖然已經有做抽象化，但驗證速度還是偏慢，最後分別 bound 在 16 和 27。

	Type	Name	Engine	Bound
?	Assert	riscv_top.isa_i.check_Addr_pipe0	K	16 -
?	Assert	riscv_top.isa_i.check_Addr_pipe1	Hts	27 -

Property2: 檢查指令是否有把 memory 讀出來的資料處理成 load byte，並把剩餘位數補 0

1. 在 sva 中計算 golden data

   會先把 memory 讀出來的原始資料 pipe 到 WB stage，接著將這筆原始資料處理為 load byte data 作為 golden data。

```
//pipe0
always @ (posedge clk or posedge rst)
if (rst)
begin
    LW_WB_pipe0 <= 32'd0;
end
else if (pipe0_stall_WB)
begin
    LW_WB_pipe0 <= LW_WB_pipe0;
end
else if(pipe0_flush_WB)
begin
    LW_WB_pipe0 <= 32'd0;
end
else
begin
    LW_WB_pipe0 <= u_core.u_lsu.mem_data_rd_i;
end
```

```
always@(*)
case (gold_addr_pipe0[1:0])
    2'h3: golden_LBU_WB_pipe0 = {24'd0,LW_WB_pipe0[31:24]};
    2'h2: golden_LBU_WB_pipe0 = {24'd0,LW_WB_pipe0[23:16]};
    2'h1: golden_LBU_WB_pipe0 = {24'd0,LW_WB_pipe0[15:8]};
    2'h0: golden_LBU_WB_pipe0 = {24'd0,LW_WB_pipe0[7:0]};
endcase
```

2. trigger 拉起時比對 DUV 的 writeback data 和 golden data 是否相同

```
check_mem_LBU_data_pipe0: assert property
(
    @(posedge clk)
    disable iff (rst)
    lbutri_pipe0 |-> (golden_LBU_WB_pipe0[16:15] == WB_data_pipe0[16:15])
);
```

3. Assumption

   因為這裡只會考慮 memory 讀出來的資料是正確的狀況，所以這裡我們有多加了 2 個 assumption，都是為了避免 memory 讀錯資料。

```
no_access_fault: assume property( !((u_core.u_lsu.mem_ack_i && u_core.u_lsu.mem_error_i) || u_core.u_lsu.mem_unaligned_e2_q ));
mem_always_accept: assume property(u_core.u_lsu.mem_accept_i == 1'b1);
```

4. 驗證結果

	Type	Name	Engine	Bound
✔	Assert	riscv_top.isa_i.check_mem_LBU_data_pipe0	Ncustom1 (1	Infinite
✔	Assert	riscv_top.isa_i.check_mem_LBU_data_pipe1	Ncustom1 (1	Infinite

Property3: 檢查資料是否有被正確寫回 register file

這邊原理和 ANDI、AUIPC 指令一樣，在 trigger 拉起時(WB stage)的下個 cycle 把 register file 的值取出來和 writeback data 比對是否一樣

```
always@(posedge clk  or posedge rst)
if(rst) begin
    WBData_pipe0 <= 32'd0;
    WBData_pipe1 <= 32'd0;
end
else    begin
    WBData_pipe0 <= u_core.u_issue.pipe0_result_wb_w;
    WBData_pipe1 <= u_core.u_issue.pipe1_result_wb_w;
end
```

```
check_mem_wb_pipe0 : assert property
(
    @(posedge clk)
    disable iff (rst)
    lbutri_pipe0 |=> (WBData_pipe0 == RF[rd_pipe0_ins5])
);
```

d) BGE

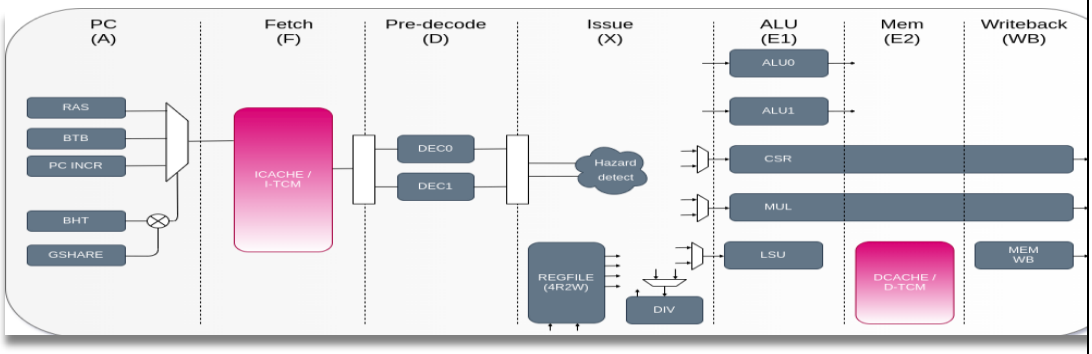
```
logic bgetri_pipe0;
assign bgetri_pipe0  = (rvfi_ins4_pipe0[6:0] == 7'b1100011 &&
                        rvfi_ins4_pipe0[14:12] == 3'b101 );

logic bgetri_pipe1;
assign bgetri_pipe1  = (rvfi_ins4_pipe1[6:0] == 7'b1100011 &&
                        rvfi_ins4_pipe1[14:12] == 3'b101 );
```

首先需要產生 BGE 的 trigger，來判斷是否需要檢查這條指令 trigger 需要符合條件才會拉起。 在 WB stage 時會判斷 pipeline follow 裡的指令是不是 BGE，BGE 的指令 opcode 為 1100011 funct3 為 101 。

```
always @(posedge clk) begin
    if (rst)
    branch_pc_3_pipe0 <= 32'd0;
    else if (pipe0_stall_E2)
    branch_pc_3_pipe0 <= branch_pc_3_pipe0;
    else
    branch_pc_3_pipe0 <= u_core.u_issue.branch_exec0_pc_i;
end

always @(posedge clk) begin
    if (rst)
    branch_pc_4_pipe0 <= 32'd0;
    else if (pipe0_stall_WB)
    branch_pc_4_pipe0 <= branch_pc_4_pipe0;
    else
    branch_pc_4_pipe0 <= branch_pc_3_pipe0;
end
```



再來因為 branch 計算完的 PC 位置為 E1 stage，由於我們要實現 end to end 的驗證方式，故我們須等到 rs 存完暫存器也就是在 WB stage 中
才能將這個值與 golden 的值做比較。

因此我將 E1 算出來的值 往下 pipe2 級。

```
assign    gold_pc_four_pipe0 =  (bgetri_pipe0 & valid_wb_pipe0) ? rvfi_pc4_pipe0 + 32'd4 : 32'd0;

assign    gold_pc_imm_pipe0 =  (bgetri_pipe0 & valid_wb_pipe0) ?
rvfi_pc4_pipe0 + {{19{rvfi_ins4_pipe0[31]}}, rvfi_ins4_pipe0[31], rvfi_ins4_pipe0[7], rvfi_ins4_pipe0[30:25], rvfi_ins4_pipe0[11:8], 1'b0} : 32'd0;
```

```
function [0:0] greater_than_signed;
    input  [31:0] x;
    input  [31:0] y;
    reg [31:0] v;
begin
    v = (y - x);
    if (x[31] != y[31])
        greater_than_signed = y[31];
    else
        greater_than_signed = v[31];
end
endfunction
```

這邊計算 golden PC+4 及 golden PC+立即值 以及比較的 function。

```
assign branch_taken_pipe0 = (bgetri_pipe0) ? greater_than_signed(gold_rs1_pipe0,gold_rs2_pipe0) | (gold_rs1_pipe0 == gold_rs2_pipe0) : 1'd0 ;
assign branch_taken_pipe1 = (bgetri_pipe1) ? greater_than_signed(gold_rs1_pipe1,gold_rs2_pipe1) | (gold_rs1_pipe1 == gold_rs2_pipe1) : 1'd0 ;

assign gold_bge_pc_pipe0  = (branch_taken_pipe0) ?  gold_pc_imm_pipe0 :  gold_pc_four_pipe0 ;
assign gold_bge_pc_pipe1  = (branch_taken_pipe1) ?  gold_pc_imm_pipe1 :  gold_pc_four_pipe1 ;
```

然後這邊是 BGE 決定要跳向哪個地址的 trigger，BGE 指令是看 rs1 有無大於或等於 rs2，所以可以看到透過一個比大小的 function 來計算出 rs1 是否大於 rs2， 後面這邊是做有沒有相等的判斷。

```
bge_pipe0 : assert property
    (@(posedge clk)disable iff (rst)
    (bgetri_pipe0 & valid_wb_pipe0 ) |-> (gold_bge_pc_pipe0 == branch_pc_4_pipe0));

bge_pipe1 : assert property
    (@(posedge clk)disable iff (rst)
    (bgetri_pipe1 & valid_wb_pipe1 ) |-> (gold_bge_pc_pipe1 == branch_pc_4_pipe1));
```

最後將算出來的值當作 golden 和上一個投影片中的 branch_pc4 做比較

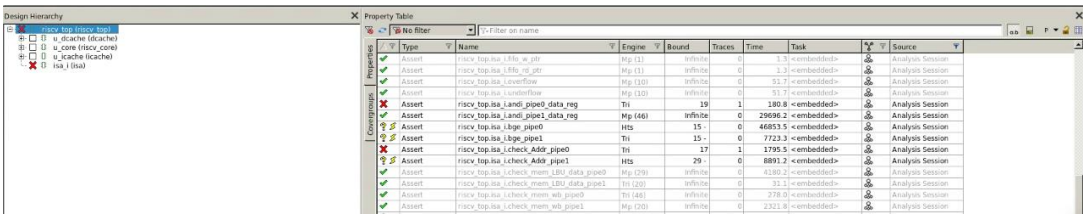
	Type	Name	Engine	Bound	Traces	Time	Task		Source
?	Assert	riscv_top.isa_i.bge_pipe0	Hp	13 -	0	9470.5	<embedded>		Analysis Session
✓	Cover (related)	riscv_top.isa_i.bge_pipe0:precondition1	Ht	8	1	0.4	<embedded>		Analysis Session
?	Assert	riscv_top.isa_i.bge_pipe1	Hp	12 -	0	4009.7	<embedded>		Analysis Session
✓	Cover (related)	riscv_top.isa_i.bge_pipe1:precondition1	Ht	8	1	0.5	<embedded>		Analysis Session
●	Assume	assume:0	?		0	0.0	<embedded>		Analysis Session
●	Assume	assume:1	?		0	0.0	<embedded>		Analysis Session

```
bge_pipe0 : assert property
    (@(posedge clk)disable iff (rst)
    (bgetri_pipe0 & valid_wb_pipe0 ) |-> (gold_bge_pc_pipe0[3] == branch_pc_4_pipe0[3]));
```

?	Assert	riscv_top.isa_i.bge_pipe0	Hts	20 -	0	55135.9	<embedded>		Analysis Session
✓	Cover (related)	riscv_top.isa_i.bge_pipe0:precondition1	Hts	8	1	0.8	<embedded>		Analysis Session
?	Assert	riscv_top.isa_i.bge_pipe1	Tri	14 -	0	5505.8	<embedded>		Analysis Session
✓	Cover (related)	riscv_top.isa_i.bge_pipe1:precondition1	Hts	8	1	0.5	<embedded>		Analysis Session

這裡為 BGE 的驗證結果，由於 BGE 是透過 2 個暫存器做比較，相對其他指令來說 bound 數偏低且跑得相當久，所以我們透過抽象化的方法加速驗證速度，因而提高 bound 數。數據為 13bound 提升至 20bound。

四、Trojan



```
wire    ctrl;
assign  ctrl = (reg_r10_q==32'd10 && reg_r9_q==32'd10 && reg_r8_q==32'd10 && reg_r7_q==32'd10 && reg_r6_q==32'd10 && reg_r5_q==32'd10  ) ? 1'd1 : 1'd0;
```

```
assign ra0_value_o = (ctrl) ? 32'd0 : ra0_value_r;
```

trojan 埋的位置為 regfile 中的暫存器回傳值， trigger 的條件為任意幾個暫存器等於隨意值的時候啟動，會讓 pipe0 的暫存器回傳值發生錯誤，指令只要會用到 rs1 的都會報錯(按)，可以看到 ANDI 和 LSU 的指令發生錯誤，Andi 是因為 rd 存回的值為 rs1+立即值，而 LSU 則是需要算出 rs1+立即值的地址所以兼會出錯，原本預計 BGE 指令也該出錯，但因為 BGE 指令的 bound 數推進太慢，故尚未抓出來。

```
                - unknown                       : 0
                - error                         : 0
no_properties
no_properties
[<embedded>] % get_design_info
Statistics [for instance "riscv_top"]
---------------------------
# Flops:          872 (10232) (0 property flop bits)
# Latches:        0 (0)
# Gates:          19678 (116040)
# Nets:           21608
# Ports:          58
# RTL Lines:      11376
# RTL Instances: 45
# Embedded Assumptions: 0
# Embedded Assertions:  0
# Embedded Covers:      0
10232
```

```
                  - unknown                     : 0
                  - error                       : 0
no_properties
no_properties
[<embedded>] % get_design_info
Statistics [for instance "riscv_top"]
---------------------------
# Flops:         872 (10232) (0 property flop bits)
# Latches:       0 (0)
# Gates:         19701 (116184)
# Nets:          21632
# Ports:         58
# RTL Lines:     11424
# RTL Instances: 45
# Embedded Assumptions: 0
# Embedded Assertions:  0
# Embedded Covers:      0
10232
```

Percentage of HT =(19701-19678)/19648*100= 0.11%

加入 trojan 之後所增加的 gate count 從 19678 提升至 19701 大概為 0.11% 。

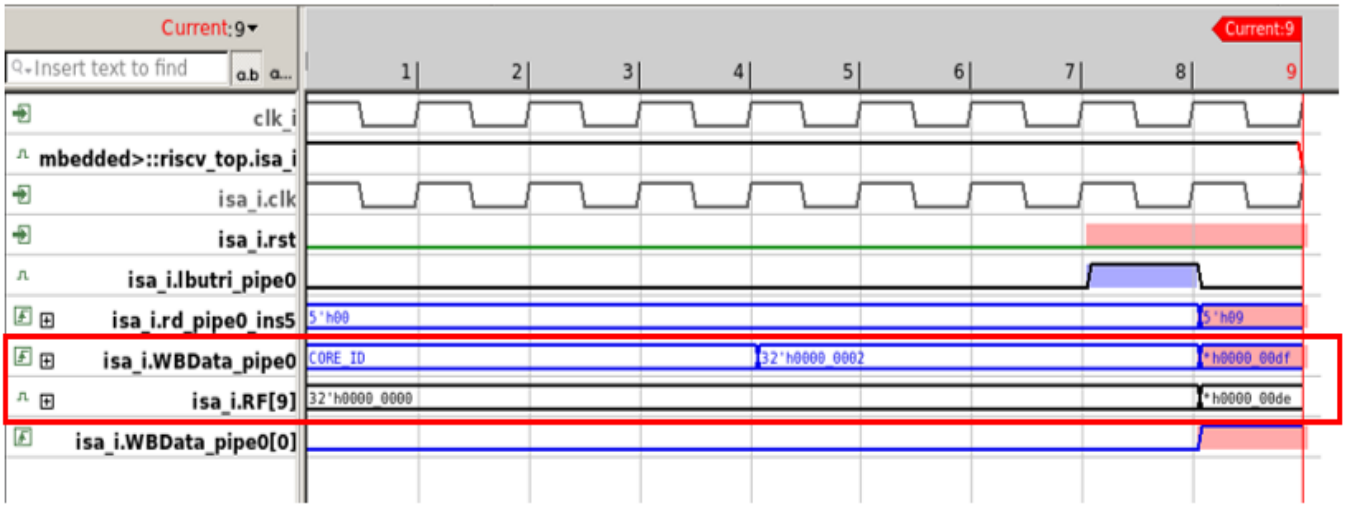
五、Other trojan

驗證結果

	Type	Name	Engine	Bound
✗	Assert	riscv_top.isa_i.andi_pipe0_data_reg	Hts	9
✗	Assert	riscv_top.isa_i.andi_pipe1_data_reg	Hts	9
✗	Assert	riscv_top.isa_i.auipc_pipe0_data_reg	Hts	9
✗	Assert	riscv_top.isa_i.auipc_pipe1_data_reg	Hts	9
✗	Assert	riscv_top.isa_i.bge_pipe0	Hts	9
✗	Assert	riscv_top.isa_i.bge_pipe1	Hts	9
✗	Assert	riscv_top.isa_i.check_Addr_pipe0	Hts	9
✗	Assert	riscv_top.isa_i.check_Addr_pipe1	Hts	9
✗	Assert	riscv_top.isa_i.check_mem_wb_pipe0	Hts	9
✗	Assert	riscv_top.isa_i.check_mem_wb_pipe1	Hts	9
✓	Cover (related)	riscv_top.isa_i.andi_pipe0_data_reg:precondi...	Hts	8
✓	Cover (related)	riscv_top.isa_i.andi_pipe1_data_reg:precondi...	Hts	8
✓	Cover (related)	riscv_top.isa_i.andi_pipe0_data_rd:preconditi...	Hts	8

我們驗 trojan 的 sva 和原本的相同，沒有增加其他 property，我們在第 9 個 bound 的時候抓到他們的 trojan。從驗證結果可以看到像是檢查 write back data 是否有正確寫回 register，以及檢查 memory address 是否正確等等和 register file 有相關的 property 都沒有通過。

波型



這邊用檢查 write back data 是否被正確寫回的 property 舉例

可以看到波型裡 write back data 和最後寫進 register 的數值不同，而我們去檢查 register file 的 code，確實有發現他們對 write back data 的最後一個 bit 做反向，導致 register 裡的數值和正確的

write back 數值有誤差。

六、組員分工

陳語彤	ANDI/AUIPC/FIFO 驗證
陳俐蓉	LBU/Other trojan 驗證
吳念泓	BGE 驗證/埋+驗證 trojan