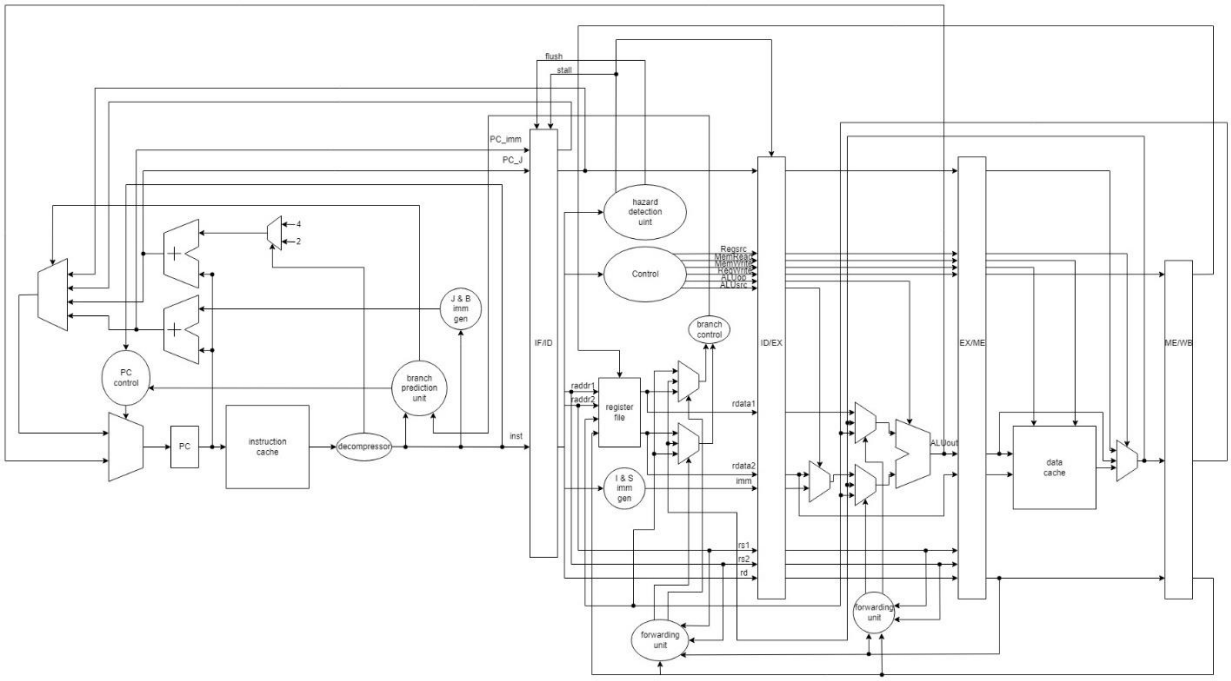
DSD Pipeline RISCV report

Group 2: b09901081 施伯儒、b09901178 廖昶翔



We finish:

1. Five stage pipeline structure
2. Hazard detection unit
3. Forwarding unit x2
4. Branch Prediction unit
5. Decompressor
6. Read-only cache
7. Pass bassline testbenches
8. Pass extension testbenches
9. Pass Q_sort testbench

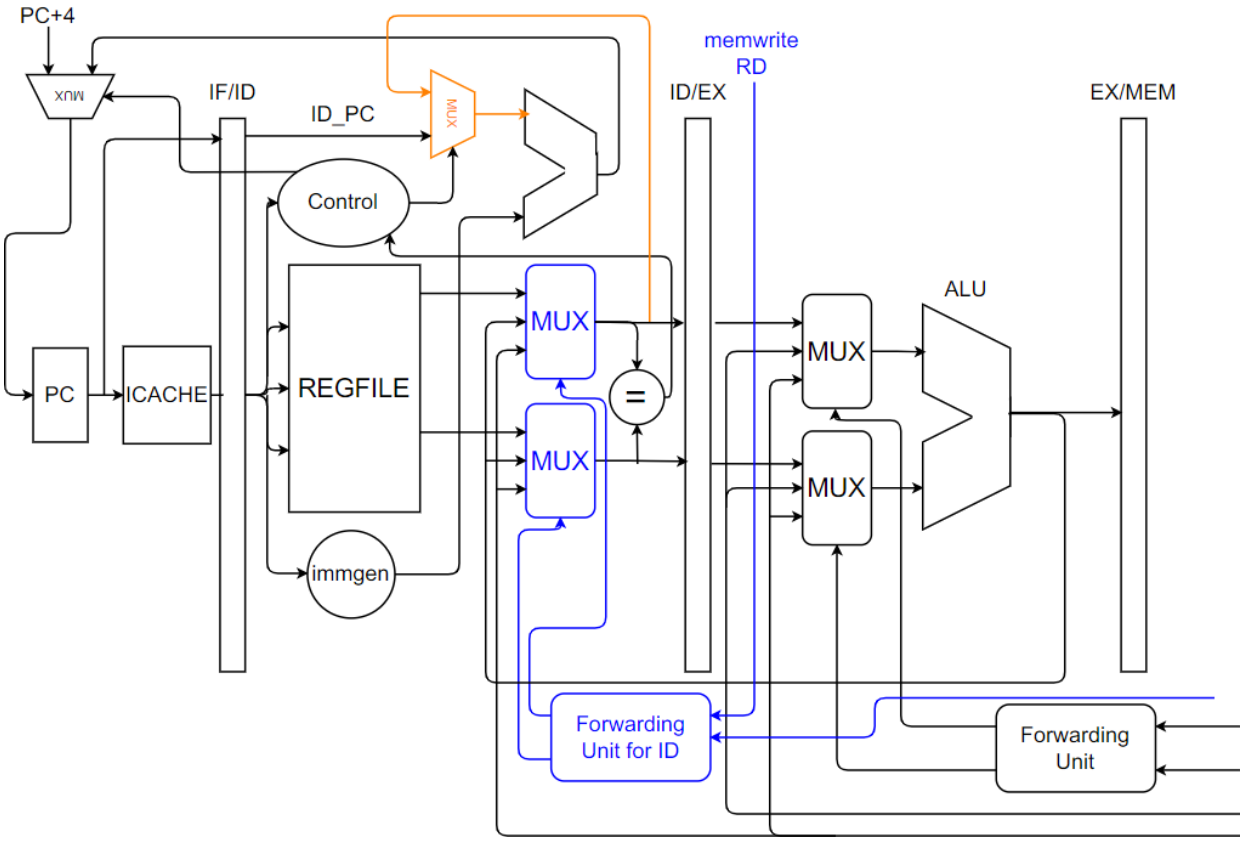
Baseline

Design details

從single-cycle的RISCV到pipelined-RISCV，雖然pipeline的設計能讓thorughput提高，但是相對應的會需要考慮許多的問題，包括forwarding、load-use hazard、branch hazard...等，由於補充影片中已經對此部份做過介紹，因此在這裡就不再贅述。在作業說明中已經有給pipelined-RISCV的block diagram，但其實對要做出能夠執行一整套的assembly，還欠缺了一些東西，除了作業說明提到的，我們還有:

新加入一個forwarding unit

在pipelined-RISCV裡面，我們會把Jalr Jal Bne Beq拉到ID stage來做，因為如果在EX stage來做的話，只要新的PC不是PC+4，就必須要stall兩個cycle，在ID stage判斷可以只stall一個cycle，但是必須考慮ID-stage的forwarding unit。原本的forwarding unit是放在EX-stage，來確保進去ALU的東西是最新的，同理，我們在ID-stage做上述四個指令的時候也有可能會遇到這種狀況，因此必須 在ID-stage也加入forwarding unit來確保資料的正確性(如下圖藍色部份)。Forwarding unit要考慮EX、MEM、WB stage的forward回來，之所以要考慮WB stage是因為有可能在同一個cycle內regfile讀出和寫入的地方是同一個，但是寫入的data會到下一個posedge才會把資料推進register中，所以這種情況也要考慮進去，直接把原本要寫入的data讀出來。



舉例來說，以下狀況如果不加forwarding unit就會導致出錯。

```
addi x11 x11 0x004
bne  x11 x12 loop
```

當addi前進到EX-stage的時候bne正在ID-stage中，因此這時候要把ALU計算出來的結果趕快送給bne的comparator做判斷，才會是邏輯正確的code。另外，由於Jalr跳轉的address是regfile[RS1]+immediate，因此在計算跳轉address的時候必須考　慮adder的第一個input有可能是PC或者是regfile[RS1](如上圖橘色部份)。

有做到以上事情就可以保證把Jalr Jal Bne Beq拉到ID-stage做的同時不會有任何錯誤了。

在這次的Project裡面，由於是用cache接到slow_memory，在cache miss的時 候會送出stall訊號讓電路稍等一下，因此處理cache的stall問題也是一個大關鍵。**我們的做法是一旦出現cache stall，便會先令所有stage都先停在原處**

避免出現錯誤。

實作上的小重點

1. 當出現sw指令是會把regfile[RD]根據指定的address寫到memory(cache)中，我們的做法是在WB-stage的時候接regfile[RD]，這樣要可以避免掉forward的問題，因為有可能要寫的data還在WB stage，還沒寫進去regfile。

2. 由於RISC-V processor對instruction只會讀取不會寫入，因此可以將I-Cache的write logic拿掉，可以省下約20%的面積。

3. 由於mem_ready在slow_mem中固定會在negative edge被拉起來，但當mem_ready被拉起來之後才會去做decode選擇memory讀出來的data要寫到哪一個block裡面，這些邏輯都需要時間來做，這樣一來很可能會造成timing violation。故比較好的作法是將mem_ready input擋一層FF， 這樣就可以確保有full cycle可以做剩下的運算。

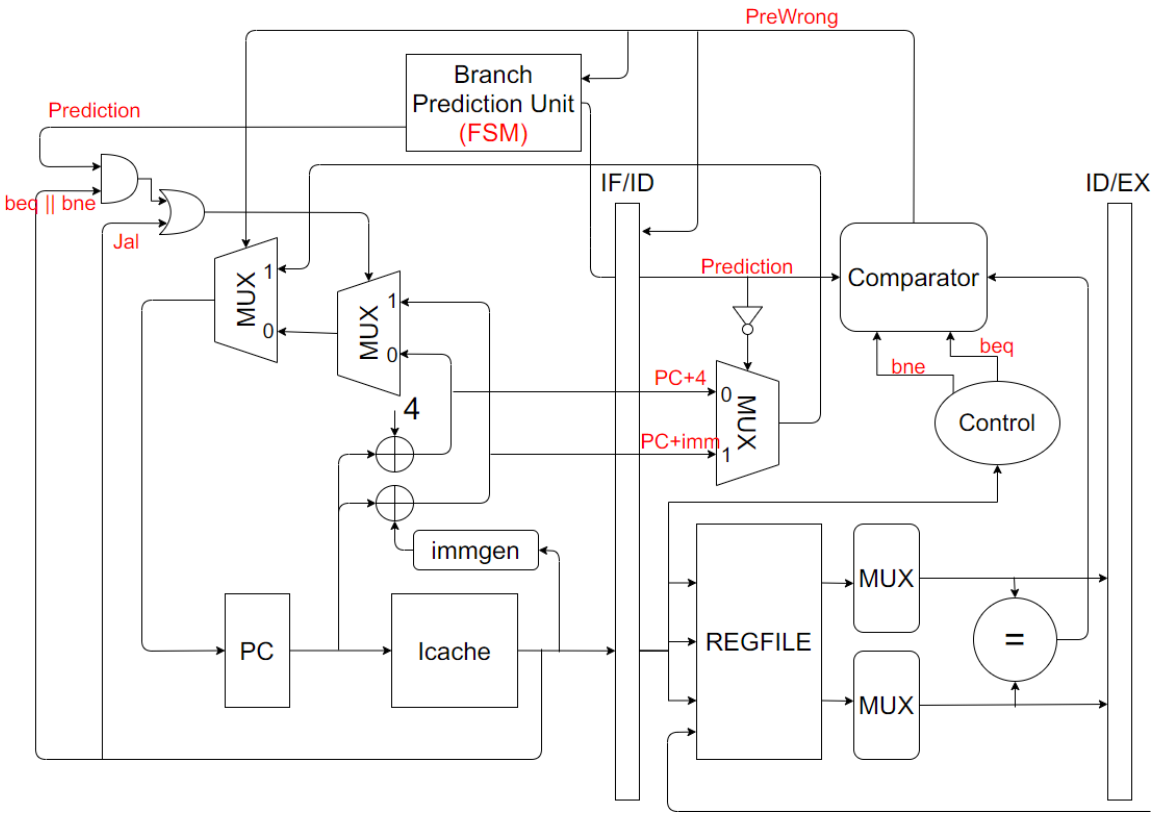
Critical Path

當碰到需要write register的指令後接branch或是jalr指令時會形成critical path。

Extension

Branch Prediction

(A) Design

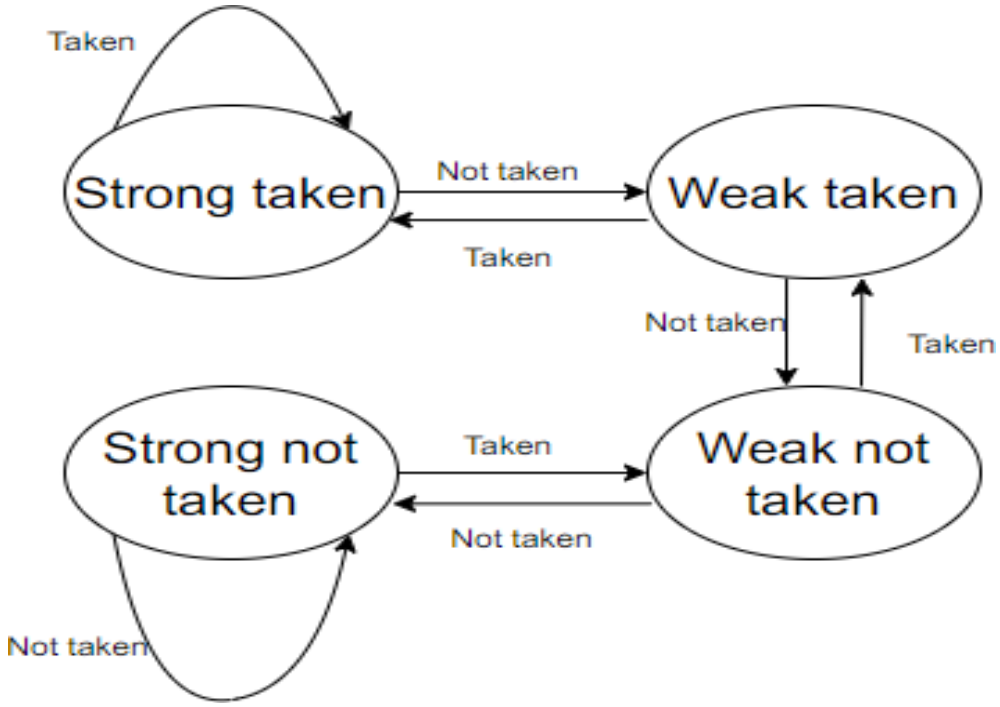


上圖是我加入Branch Prediction Unit 的電路簡圖(僅畫到ID stage)，基本上大部分的邏輯和設計都和pdf裡面要求的spec相同

(a) introduction

這次的Pipelined CPU支援的branch指令包含beq和bne，**BPU的功能主要是透過把Program Counter跳轉的時機從ID stage轉到IF stage來省下一個stalled cycle，就算預測失敗，也只要在ID stage把錯誤的instruction flush掉就好**，其主要功能就是當我一連串的instruction出現loop的時候，能透過prediction來省下cycle數。而當我們遇到loop的時候loop中的branch指令都會是同種，**因此在最後我的設計是將beq和bne的BPU合併，BPU預測的是branch指令會不會taken(e.g.會不會跳到PC+imm的address)**，而不是預測是否兩個待測的數值相同。這樣設計可以多省下一個BPU的面積(相較替兩種branch指令都設計一個BPU)，同時也能達到Branch prediction unit的功能。

下圖是BPU的state diagram，主要是用2-bit predictor來實作，這樣的predictor在預測迴圈中只有一個branch指令的時候、或者迴圈中除了控制跳轉的branch指令之外還有一個不一定發生的branch指令時，能夠達到良好的成效。



Some modifications:

```
assign PC_w = (J)           ? PCimm_ID  :
              (JALR_EX)     ? ALUout_EX :
              (PreWrong)    ? PC_correct:
              (BrPre_IF)    ? BPU_PC_IF :PC4_IF;
```

```
assign PC4_IF = PC_IF + 4;
assign inst_IF = {ICACHE_rdata[7-:8], ICACHE_rdata[15-:8], ICACHE_rdata[23-:8], ICACHE_rdata[31-:8]};
assign immB_IF = {{20{inst_IF[31]}}, inst_IF[7], inst_IF[30:25], inst_IF[11:8], 1'b0};
assign op_IF = inst_IF[6:0];
assign B_IF  = &{op_IF[6], op_IF[5], ~op_IF[4], ~op_IF[3], ~op_IF[2], op_IF[1], op_IF[0]};    // 1100011
assign imm_IF = {32{B_IF}} & immB_IF;
assign BPU_PC_IF = PC_IF + imm_IF;
```

```
assign flush = (branch&&~branch_hazard&&PreWrong) || J || JALR || JALR_EX || PreWrong;
```

1. **JAL & JALR**

   J type的指令在判斷PC下個位置的優先序比BPU更高，因為如果我要jump的話，我應該要立即jump，但是因為處理J type指令是在ID stage才處理，不可以把branch排在它之前。否則，PC可能會跳去branch而非jump。

2. **Move B type instruction decoder to IF stage**

   因為我們得在IF stage就判斷是否branch，那當然有關B type的運算得提前到IF stage去做。

3. **Flush or not**

當判斷錯誤時，我們得及時flush ID stage的資料(後面的stage繼續運作)，避免後面的stage使用到錯誤的ID stage資料。

(B) What you have learned?

2-bit predictor的幾個特點

1. 對於簡單的單層迴圈會有良好的預測效果，因為單層迴圈通常是在最後以bne或beq組成會是always taken或never taken的情況，state經過修正之後可以有效地預測。

2. 對單層迴圈內加入一個branch指令的預測效果大部分時間還不錯，2- bit predictor的特點就是"look before you leap"，在確定要切換到taken/not taken state的時候容許一個額外的指令，恰好可以處理中間這個不一定會發生的指令。實際上，能夠處理這種case也是2-bit predictor和1 bit predictor中最主要的分別，1 bit predictor只能用來處理簡單的單層迴圈。

3. 對2.的worst case來說就是中間額外的branch指令造成和in-loop的branch指令交錯的情形，對於這種interleaved的pattern，2-bit predictor最好也只能達到50%的預測率，最差則會預測全落空。

4. Branch Prediction Unit的缺點：乍看之下branch prediction unit確實可以對大部分的情況來做predict並有良好的成效，但是也會伴隨著幾項缺點，除了增加一些面積之外，最顯著的差異就是因為我們對icache讀出來的東西去做處理，會增加critical path。

Compression

Introduction

RISCV Compression將幾個比較常用，並且可以在條件下簡化的instruction從32bit壓縮成16bit。在達到壓縮的目的下，對於在16bit中放入足夠的資訊就變成設計RISCV Compression的重點。

Design Concept

為了交互處理16bit與32bit的指令，以及對於這兩種指令對於Instruction Cache的位置對齊以及讀取方式，需要稍微思考下面幾個設計的重點：

1. 設計decoder或是decompressor：由於decoder需要在ID stage額外設置一個對於指令是否為compressed instruction的判斷，而PC的計算又要在IF stage進行計算，所以我們認為利用decompressor，並且在IF stage完成設計，可以在解析兩種指令的情況下最大限度地不改變critical path的長度。

2. ICACHE memory配置：ICACHE可以設計成兩種方式，第一種是不改變ICACHE，利用PC以及stall 幾個cycle來完成分隔在不同word中的資訊，第二種是修改ICACHE，輸入32bit的address，利用額外設計的部分來讀取橫跨兩個word的資訊。這邊考慮使用第一種方法。

Design Details

首先介紹decompressor的設計。Decompressor是透過3 bit的function以及2 bit的op code來決定其指令，decompress方式如下:

```
case({C_inst[15:13],C_inst[1:0]})
```

根據這個分類，可以找出one to one的mapping，只有在其中兩個case時，我們才需要更進一步去細分:

```
5'b10001: begin
    case(C_inst[11:10])
```

```
5'b10010: begin
    case({C_inst[12], C_inst[6:2]!=5'd0})
```

再來，介紹在主設計中，針對decompressor所做出的調整:

下圖為在主設計中額外的調整。可以看到，i_inst_before_w表示輸入

```
// Long instuction but not aligned
assign i_inst = {inst[7:0], inst[15:8], inst[23:16], inst[31:24]};
assign i_inst_before_w = i_inst[31:16];
assign C_inst = PC[1] ? i_inst[31:16] : i_inst[15:0];
assign I_not_aligned_w = (!stall) & (!I_not_aligned_r) & (PC[1]) & (C_inst[1]) & (C_inst[0]);
assign o_stall  = stall || I_not_aligned_w;
assign o_addr = I_not_aligned_r ? (PC[31:2] + 30'd1) : PC[31:2];

always@(*) begin
    if (I_not_aligned_r) begin
        // PC + 4
        o_PCoff = 2'b10;
        o_inst = {i_inst[15:0],i_inst_before_r};
    end
    else if (C_inst[1:0] == 2'b11) begin
        // PC + 4
        o_PCoff = 2'b10;
        o_inst = i_inst;
    end
    else begin
        // PC + 2
        o_PCoff = 2'b01;
```

進來的指令前半段，而C_inst則依據PC[1]是否為0決定選取前半段或後半段(如果是compressed instruction，會造成PC+2，所以PC[1] = 1)，當發生alignment issue時，我們會設定I_not_alignment_w為1，同時輸出stall，停止一cycle，去蒐集另一半的資料，並且o_addr可以向icache告知下個要讀取的資料為何，藉由巧妙的控制read address和PC位置，可以達到正確選取資料的結果。另外因為我們有做instruction buffer，所以當另一半的32 bits資料進入，可以順利從buffer使用前一次得到的instruction資訊。最後，再依據目前狀況，決定PC_offset，解決+2 or +4的問題。