DSD Pipeline RISCV report

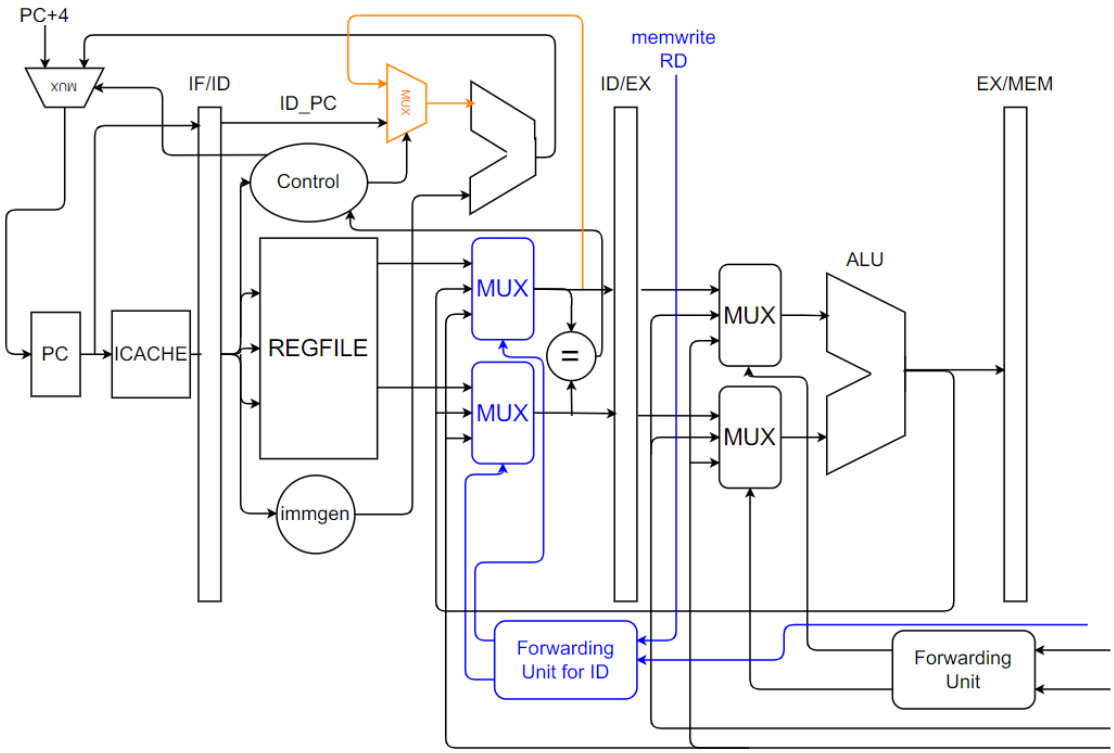
Group 3: b07901021 潘世軒、b07901036 陳俊廷、b07901137 吳陽禾

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，以下一一說明。

1. 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WBstage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。

2. 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。

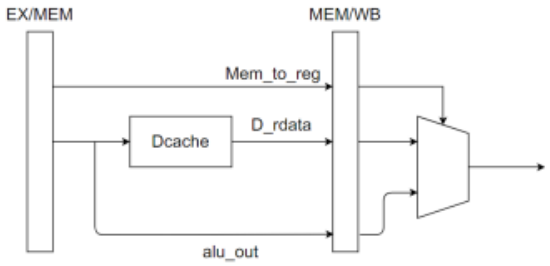
   Icache stall: 把原本在ID stage的指令保留，其他指令繼續往後做，這樣做的原因是因為若是icache stall的同時ID stage剛好是beq等指令的話，如果接下來都做nop，原本beq要跳到的address就會被洗掉，造成錯誤，因此要把在ID stage的指令保留。但為了避免不小心更改到memory或regfile的值，會把重複的指令的control signal都設為0來避免write的動作

   Dcache stall:將MEM stage以前的指令全部都留在原處，WB stage後由於不影響，可以繼續往後做

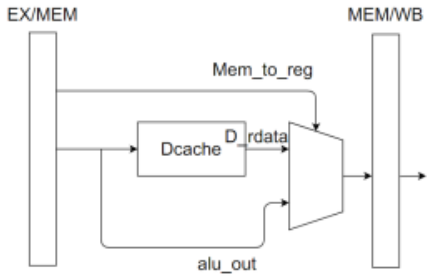
3. 當出現sw指令是會把regfile[RD]根據指定的address寫到memory(cache)中，我們的做法是在MEM-stage的時候直接接regfile[RD]，但如同前方所述，這邊一樣要注意forward的問題，因為有可能要寫的data還在WB stage，還沒寫進去regfile，因此這邊就需要增加邏輯來判斷是否需要把WBstage的data forward回來。

Optimization

1. 我們盡量把邏輯運算在越前面的pipeline stage做，這樣一來能節省FF的使用。下面為其中一個實例，我們將判斷write back的data的MUX放在MEM stage能使FF節省許多。當然，這樣做的前提是critical path並不會因此增加。

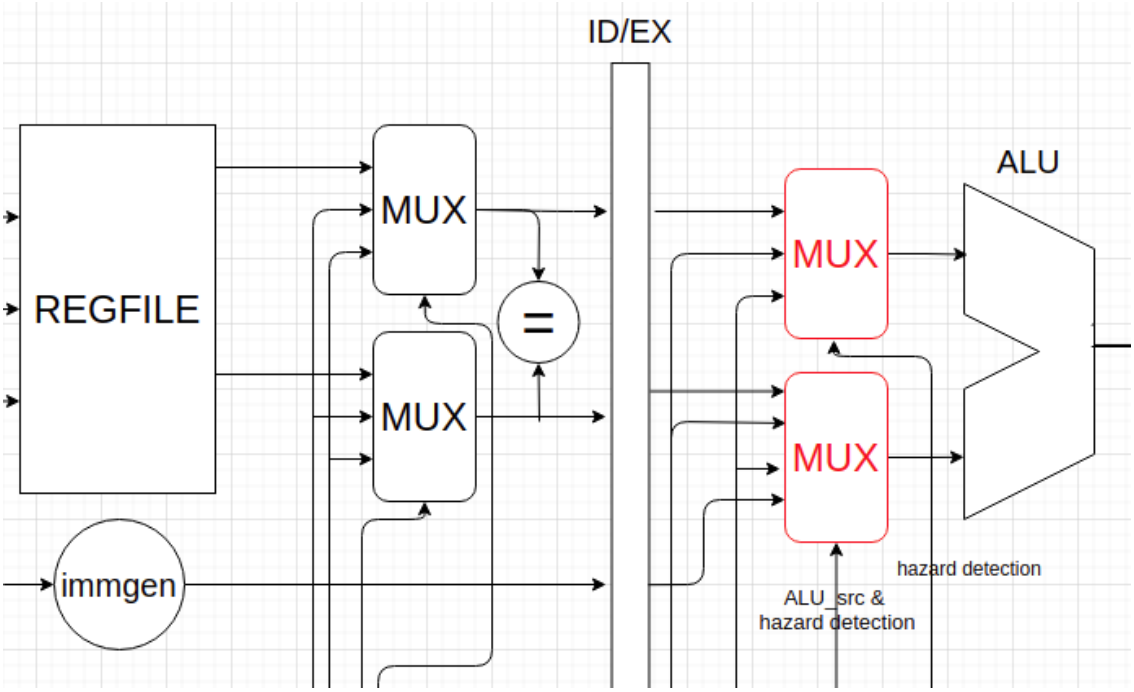


Before: FF 65 bits

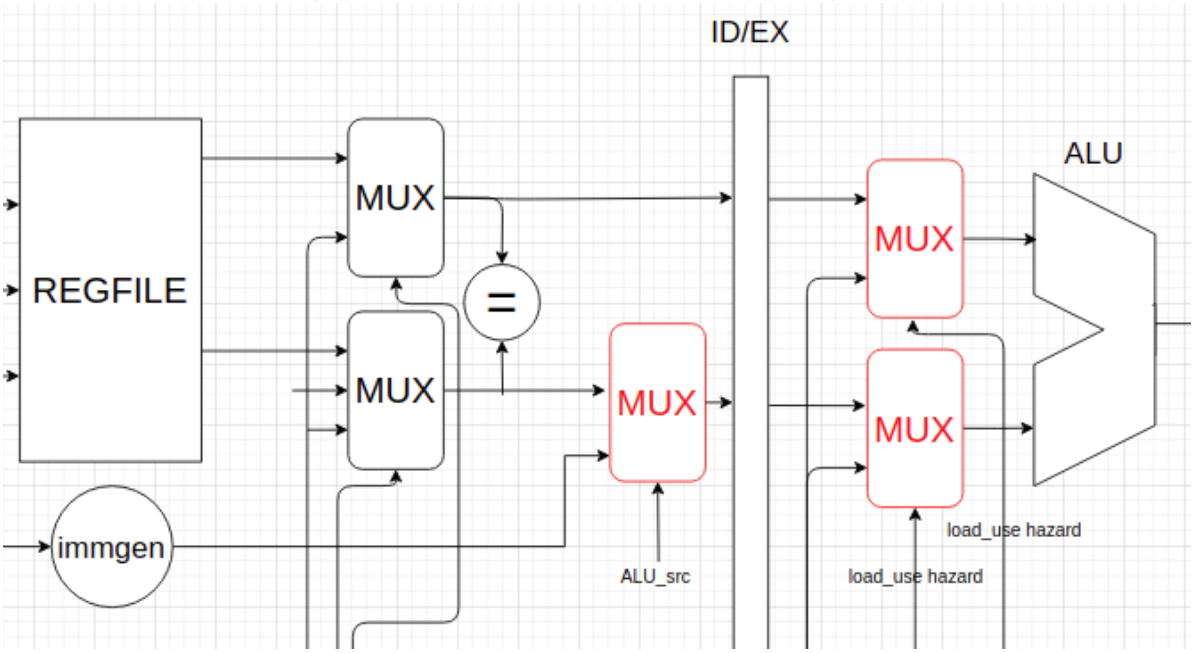


After: FF 32 bits

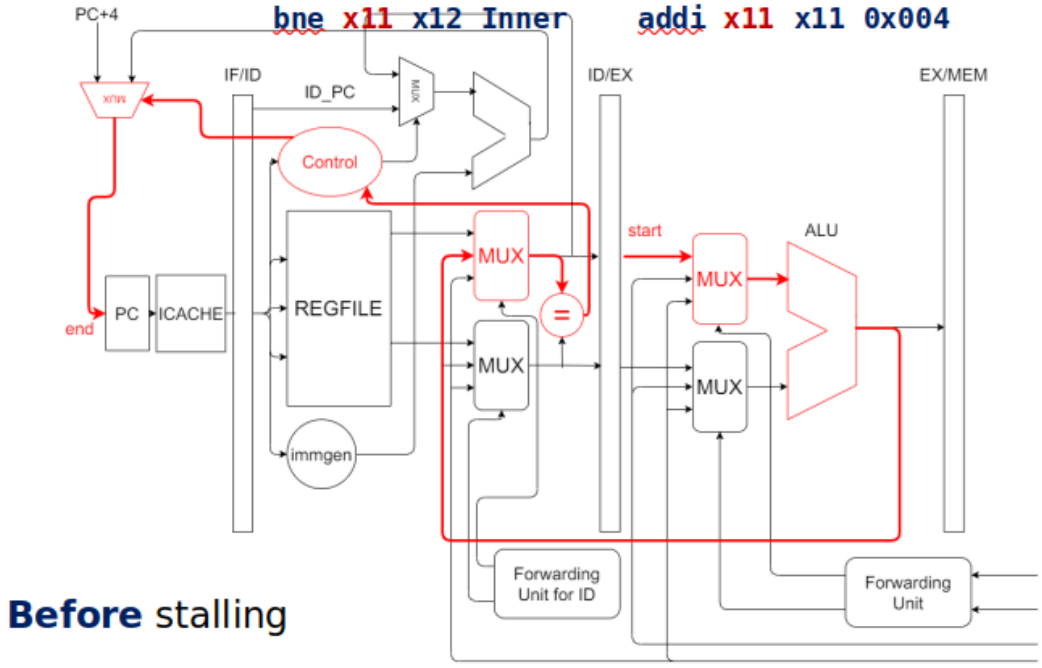
另一個實例如下。



一般情況下ID-EX的datapath是將Registerfile出來的資料經過hazard detection後，與immediate一起被送到EX stage並且在EX stage透過hazard detection與ALU_src來決定要拿哪兩個data來運算。然而其實可以將這個MUX移到ID stage，如此一來除了可以簡短EX stage的critical path，也可以節省FF的使用。除此之外，正如前面提到，我們會多一個ID stage的hazard detection，而在EX stage裡面的data hazard其實都可以在ID stage detect，所以也可以提前forward。唯一需要在EX stage考慮的是load use hazard，因為在ID stage時該資料還未被讀出，所以依然要在EX stage 去forward。

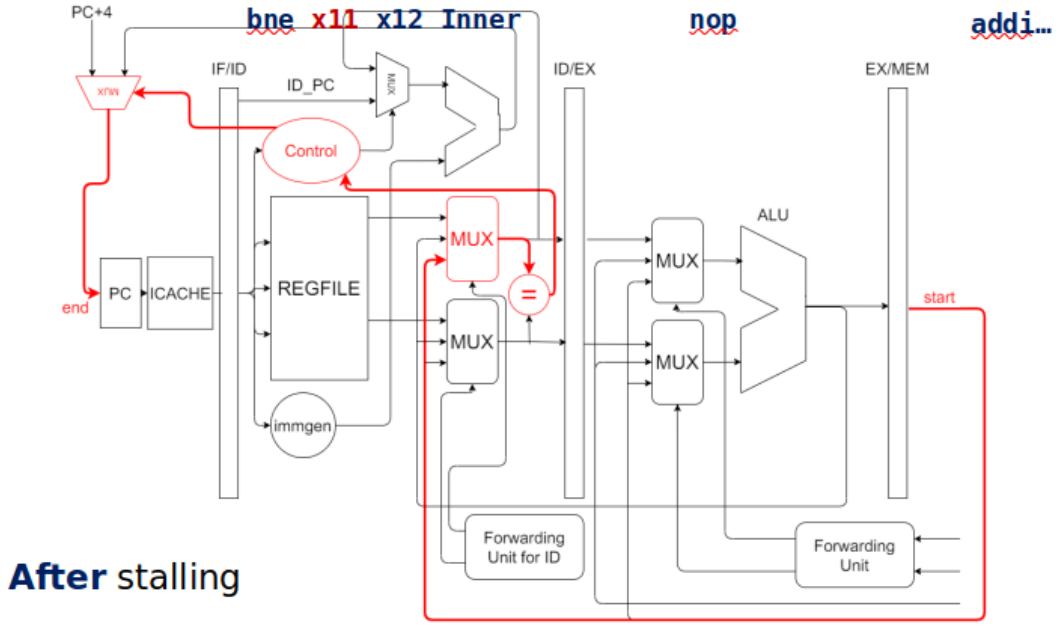


2. 在基本的設計中，當碰到需要write register的指令後接branch或是jalr指令時會形成critical path(如下圖)。

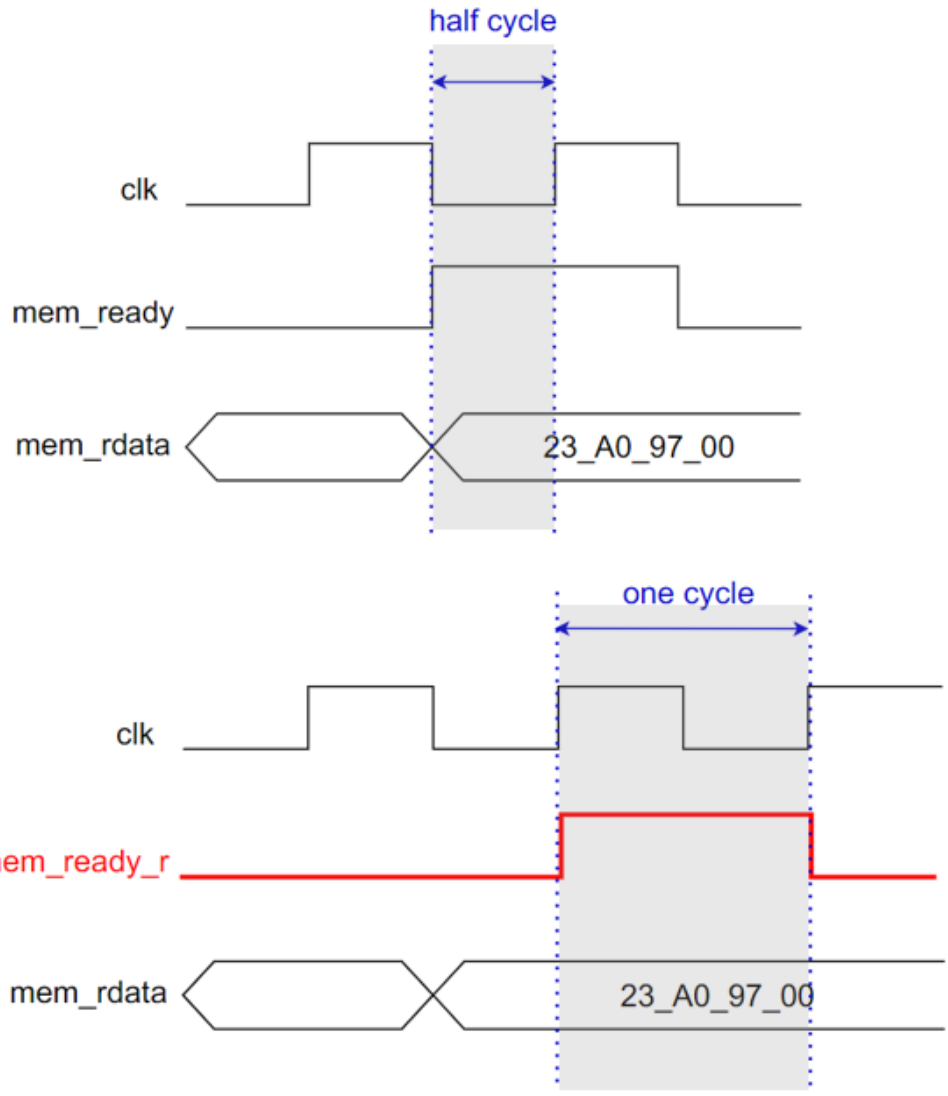


在這樣的情況下，critical path ~= 4ns。為了解決這個問題，我們將這樣的情況命名為jump hazard，並且在遇到這個hazard時我們就在中間insert一個nop以降低critical path(如下圖)。加了

nop以後這個case就不會是critical path，並且可以使得critical path降低到2.5ns左右。



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(如下圖)。



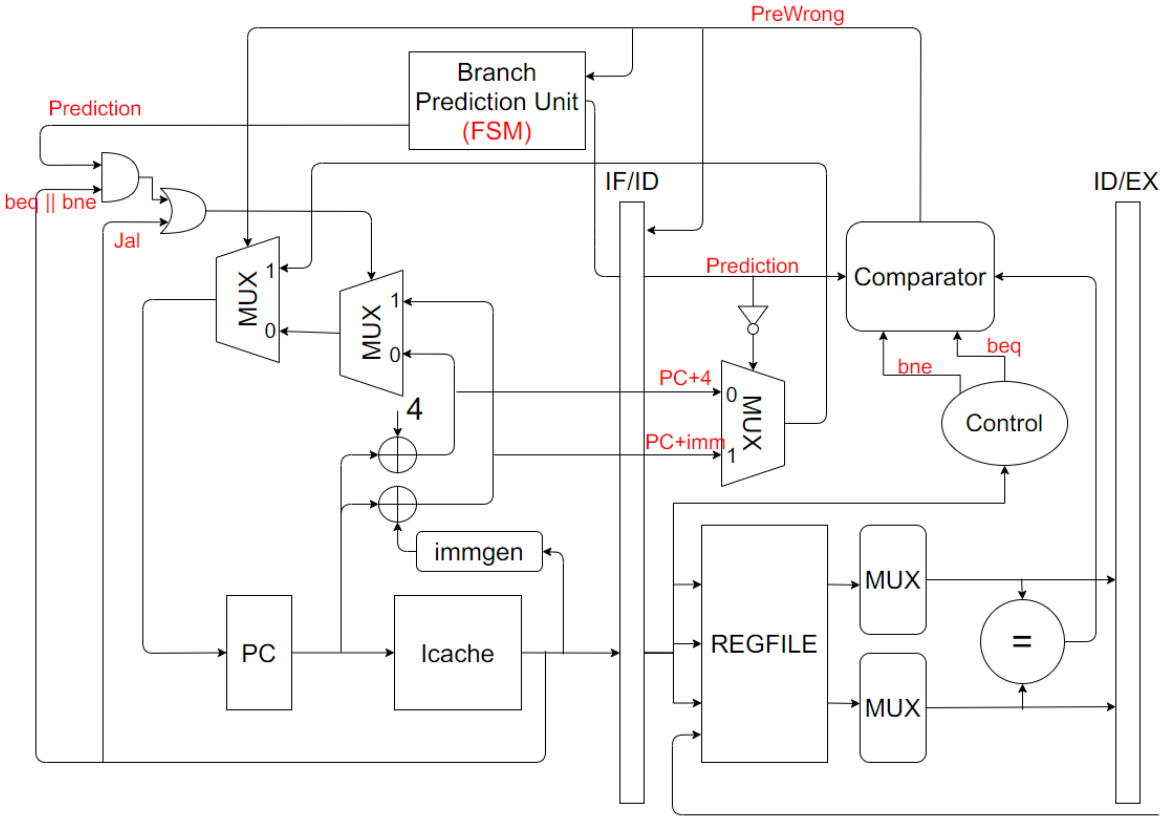
4. 由於instruction大多是連續的address，不太會有tag相同的問題，因此replace的情形幾乎不太會發生，並透過實驗驗證，我們發現使用direct-mapped的ICache效果會略勝associative ICache。

5. 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。
6. 我們在合成時發現，SLT的指令(<)會讓dc多map一個design ware的cmp6(6功能comparator)，但其實這個功能可以用減法器來達到一樣的效果，並且畢竟ALU同時只會執行一個指令，這樣並不會影響到SUB。因此我們用減法的方式來判斷SLT，讓SUB與SLT共用一組減法器以減少面積。
7. 與上一點相似，我們發現在處理BEQ與BNE中如果直接寫 == 與 != 會分別合成出一個cmp6，因此相同地我們也將兩個指令共用一個比較器以減少面積。

Extension

Branch Prediction

(A) Design methodology

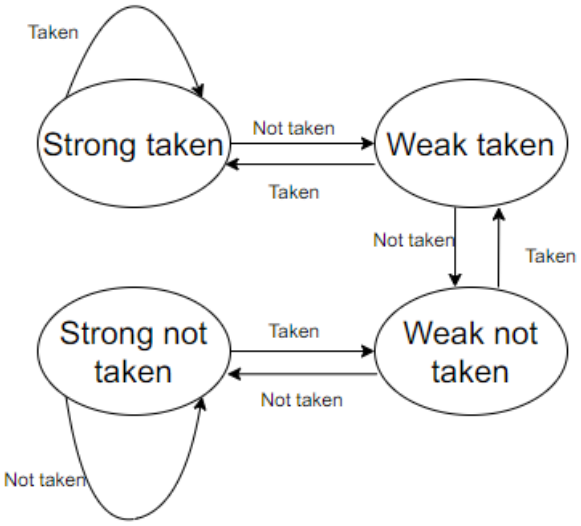


上圖是我加入Branch Prediction Unit 的電路簡圖(僅畫到ID stage)，基本上大部分的邏輯和設計都和pdf裡面要求的spec相同，以下就一些我有特別設計的巧思或是一些比較值得提的部分進行說明。

1. BPU

   這次的Pipelined CPU支援的branch指令包含beq和bne，當我在設計BPU的時候我就在想如果有兩種branch的指令，是要將兩者的BPU分開設計還是有什麼辦法可以合在一起做，後來我想了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指令能夠達到良好的成效。(詳細會在part(d)進行討論)



2. Jal

   Jal指令的功能是將next_PC設為PC+imm，並將PC+4存到指定的register中，有別於Jalr必須從Regfile中讀出RS1的值後才能計算，Jal只要PC和imm就能進行計算，其實有點像是一定會跳轉的branch指令，因此我選擇將Jal的計算也從ID移到IF stage來做計算，只要從Icache中讀出Jal指令，就會將next_PC設成PC+imm(如上方電路圖的左上角)。

3. Hazard Modification

   在做Baseline的時候，我們為了減少critical path 將下方情況做了中間stall一個cycle的改進，但在此處，由於crtical path會變成 Icache讀出來 (就算是hit，Icache讀出來也要一段不短的時間)之後做decode，算imm一直到決定next_PC這條，因此在Branch Prediction這部分，下方情況**並不會**stall一個cycle。

```
addi x11 x11 0x001
beq  x8  x11 0x004
```

(B) The relationship between BPU and parameter size for generating test program

在這部分，我透過調整BrPred_generate.py內的參數，分為三個部分來以RTL code測試設計出來的BPU

1. 固定part A, B為10, 調整part C 次數

C	10	20	30	40	50
Total cycle	174.5	194.5	214.5	234.5	254.5

   part C的類型為always branch，會執行的指令如下

```
0x50: addi x13 x13 0x001
0x54: bne  x10 x13 0xFFE (to 0x50)
```

在上方表格中可以發現，每增加10次會多20個cycle，而part C恰好包含兩個instruction，這告訴我們對於always branch的狀況，BPU可以完美預測(不需要stall cycle)。這是由於進入partC之後，最多只要stall 3個cycle，BPU的state就可以持續停留在**Strong Taken**，這樣一來無論我要在這邊loop多少次，都可以預測出是taken，只有在最後一次跳出去會需要stall一個cycle而已。

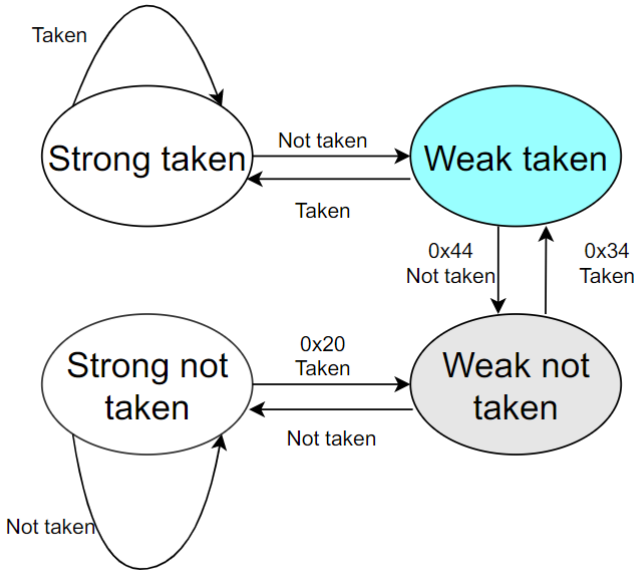
2. 固定part A, C為10, 調整part B 次數

B	10	20	30	40	50
Total cycle	174.5	234.5	294.5	354.5	414.5

Part B的類型為interleaved (taken和not taken交錯出現)，會執行的指令如下

```
0x30: addi x12 x12 0x001
0x34: bne  x9  x12 0x008 (taken, to 0x44)
...
0x44: bne  x8  x11 0x008 (not taken)
0x48: jal  x0  0xFFFF4   (to 0x30)
```

在上方表格中可以發現，每增加10次會多60個cycle，但一次loop終止會執行4個instruction，因此代表在每次loop中會stall兩個cycle，這代表說設計出來的BPU無法處理這種interleaved的pattern。參考下方的state diagram，當我們執行完part A之後的beq會是taken，因此將state從**Strong not Taken**移到**Weak Not Taken**，再來就會進入part B 的interleaved pattern，再來就會依照taken -> Not taken -> taken的順序持續在**Weak Not Taken**和**Weak Taken**切換，造成我們的預測永遠都會落空。
換個角度想，考慮一個比較好的case，即使我們現在變成在下圖的上面兩個state持續切換(也就是說一直都預測是taken，只是差別在於strong還是weak)，面對這種interleaved的pattern，也還是會有50%的失誤率。
總而言之，對於這種interleaved的pattern，2-bit predictor並不能很有效的預測，失誤率好則50%，差則100%，具體有什麼作法可以處理這種棘手的情況，將會在(F)有更詳細的討論。



3. 固定part B, C為10, 調整part A 次數

A	10	20	30	40	50
Total cycle	174.5	204.5	234.5	264.5	294.5

part A類型為never branch，會執行的指令如下

```
0x1c: addi x11 x11 0x001
...
0x20: beq  x8  x11 0x004 (not taken)
0x24: jal  x0  0xFFFFC   (to 0x1C)
```

在上方表格中可以發現，每增加10次會多30個cycle，而part A只包含3個instruction，這告訴我們對於never branch的狀況，BPU可以完美預測(不需要stall cycle)。這是由於進入partA之後，最多只要stall 3個cycle，BPU的state就可以持續停留在**Strong Not Taken**，這樣一來無論我要在這邊loop多少次，都可以預測出是Not taken，只有在最後一次跳出去會需要stall一個cycle而已。

(C) The relationship between BPU and baseline

除了(b)部分更改interleaved, always/never taken的數字之外，我想將尚未加入branch prediction unit的CPU拿來和加入BPU之後的做一個簡單的對照。因此我將baseline尚未經過optimization的code拿來和加入BPU的做比較。但在進行比較之前，先看看baseline，其實**並不是**沒有做branch prediction，baseline只是遇到branch的時候會先fetch PC+4而已，所以說，與其說baseline沒有做prediction，不如說baseline是predict never taken。

> 尚未經過optimization的代表說如果有addi接著beq這種模式的話，會將ALU出來的結果直接forward給ID stage的comparator做使用(沒有stall一個cycle)，同時為了公平起見，我將Baseline的Jal也移到IF stage做計算，這樣一來，差別只有在BPU的有無

第一部分我把兩個程式都拿來執行hasHazard的instruction，將累積至固定address的cycle紀錄如下(RTL, cycle = 10, 透過nWave紀錄)

> 0x60: return FibonacciSeries
> 0xA0: end of Bubblesort (not yet output)
> 0xB4: return BubbleSort (output)

	0x60	**0xA0**	**0xB4**	**Total**
Baseline	269.5	1563.5	1706.5	1720.5
with BPU	257.5	1580.5	1711.5	1725.5

1. Fibonacci部分
   截至fibonacci結束為止，有BPU的領先了12個cycle，這是由於fibonacci這邊是always taken，fibonacci的這個loop總共會跑14圈(計算1,2,3,5,...610)，對baseline來說，只有最後一個cycle不用stall，因此共需要stall 13個cycle。有BPU的intitial state是**Strong Taken**，只有最後一個cycle需要stall，共需要stall 1個cycle，因此領先了12個cycle，同時，在結束這段時，有BPU的會停在**Weak Taken**。
2. BubbleSort部分
   這部分有BPU的只領先10個cycle，這代表baseline追回了兩個cycle，疑？照理來說應該有predict會比較好啊？怎麼會這樣呢。這是因為我們在做sort的時候要把由小到大sort成由大到小，所以中間swap的beq會是never taken，inloop的最後一個bne則是always taken，outloop的最後一個bne也是always taken。
   baseline會在inloop和outloop的時候需要stall cycle，BPU在剛進來此部分的時候，遇到的pattern是前方所提到的interleaved的情況(swap not taken and bne taken)，就會預測全部落空，在結束第一個outloop後才會變成在**Not Taken**的兩個state中跳動，變成預測率50%的情形。因此在此部分中，baseline永遠預測Taken的成效反而是比有BPU的情況好上一些，因為2-bit predictor並不能有效的處理interleaved的情況(50%或0%)，若是sort的數列變成隨機排列的話，對2 bit predictor就會比較有利，因為當前幾個cycle把state調整好之後，就算需要swap或不需要

swap，都能維持在**Taken**的state中，對於inloop的bne就能預測成功，只會因為swap而需要stall cycle。

3. BubbleSort output 部分
   這邊就是比較簡單的單層迴圈，是always taken，所以說BPU在這部分所耗用的cycle數自然是勝過baseline(少用12個)

接下來，我把兩個程式都拿來執行BrPred的instruction (a10b20c30)，將累積至固定address的cycle紀錄如下(RTL, cycle = 10, 透過nWave紀錄)

> 0x2C: part A end
> 0x40: part B end (inst jump to part C)
> 0x58: part C end (next inst of bne)

	0x2C	**0x40**	**0x58**	**Total**
Baseline	59.5	174.5	269.5	279.5
with BPU	61.5	200.5	264.5	274.5

1. part A(never branch)
   第一部分是never branch，相比baseline只會在最後一個loop預測落空外，BPU會在先前2個loop才能把state轉到**Not Taken**的那兩個state，因此這部分Baseline小勝。
2. part B(interleaved)
   如同part(b)所提到，BPU在這邊會預測全部落空，而baseline可以達到50%的預測率(剛好是not taken, taken交錯)，因此截至這裡baseline領先蠻多的。
3. part C(always branch)
   這邊baseline就會預測全部落空，而BPU則能應付這種簡單的迴圈，但因為C執行的次數比較多，所以綜合起來BPU的performance還是會小贏一些。

(D) What you have learned?

由以上討論我們可以看出來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，比方說hasHazard instructions裡面的Bubble Innerloop中有一個swap的敘述，在大部分的程式中，中間這個不一定會發生，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(如swap)，因此2-bit predictor對這種case也能達到良好的成效。
   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inloop的branch指令交錯的情形(比方說hasHazard instructions裡面的Bubble Innerloop，要把順序直接相反，所以造成每次都要swap)，或是如BrPred裡面的partB，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。

總結，對於instruction中常出現的大部分類型，2 bit predictor已經有蠻好的成效，對於大魔王interleaved這種情況，除了從硬體改進，也可以在把Code編譯成assembly的時候就要注意如何compile，透過指令的改變，應該是可以避免這種情況的。這也證明我們設計一台CPU不只需要考慮硬體，如何將code變成assembly甚至是machine code也是一門很大的學問！

(E) Other detailed discussion

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。

為了比較這個差異，我利用hasHazard的instruction將optimized baseline和有BPU用gate-level simulation比較如下：

	sdc cycle	min tb cycle	Total time	Total cycle
optimized Baseline	2.6	2.51	5119.37	2039.5
with BPU	3.2	3.2	5604.8	1751.5

> 1. Total time 是利用min tb cycle去跑的
> 2. Total cycle的差距不光只有branch prediction的影響，還有因為clock cycle的不同，由於slow_memory裡面的latency固定是15ns，因此optimized baseline在和memory要東西的時候所需要的cycle數會比較多。還有在有BPU的設計中Jal會移到IF stage做，因此每遇到一次Jal，可以多省一個cycle。

可以看到有BPU可以省下一些cycle，但是由於critical path導致period的升高，再這個測試中總時間是比optimized baseline要多的，但是如同先前所討論過的，因為這邊的bubblesort是要把由小到大排列成由大到小，因此對2-bit predictor來說是不太公平的。為了消除這個差距，我用了修改過的hasHazard instructions來對其進行測試：

原本在執行完Fibonacci之後，Dcache(或者是memory)中儲存的有關這個數列的資料如下表。

address	**0x00**	**0x04**	**0x08**	**0x0c**	**0x10**	**0x14**	**0x18**	**0x1c**
content	0	1	1	2	3	5	8	13
address	**0x20**	**0x24**	**0x28**	**0x2c**	**0x30**	**0x34**	**0x38**	**0x3c**
content	21	34	55	89	144	233	377	610

為了讓後面的bubblesort比較有隨機性，在執行完Fibonacci之後我將上述數列中的幾個數值先交換，交換後如下表，由於數列中成員不變，理論上在做完bubblesort之後應該要是可以通過testbed的。

address	**0x00**	**0x04**	**0x08**	**0x0c**	**0x10**	**0x14**	**0x18**	**0x1c**
content	21	610	144	2	3	377	55	13
address	**0x20**	**0x24**	**0x28**	**0x2c**	**0x30**	**0x34**	**0x38**	**0x3c**
content	0	34	8	89	1	233	5	1

為了做出這種比較隨機的數列，我在原本的instructions後又加了一些instruction，如下，最主要是在Fibonacci執行完之後會跳到Random的地方將中間的某幾個順序調換，結束調換後再回到main來繼續執行bubble sort。

```
//0xC8// jalr x2  x16 0xF58 (to 0x10) // to FibonacciSeries
//0xCC// jalr x2  x16 0X02C (to 0xE4) // to Random           (NEW)
//0xD0// jalr x2  x17 0xFAC (to 0x64) // to BubbleSort       (CHANGED)
//0xD4// addi x8  x0  0xD5D
//0xD8// jal  0xFFF30 (to 0x08)       // to OutputTestPort   (CHANGED)
//0xDC// nop
//0xE0// nop
//0xE4// addi x18 x0  0x000           // Random: set the addr
//0xE8// addi x19 x0  0x020           // set the addr
//0xEC// jal  x3  0x03C (to 0x128)    // to Swap
//0xF0// addi x18 x0  0x004           // set the addr
//0xF4// addi x19 x0  0x03C           // set the addr
//0xF8// jal  x3  0x030 (to 0x128)    // to Swap
//0xFC// addi x18 x0  0x008           // set the addr
//0x100// addi x19 x0  0x030          // set the addr
//0x104// jal  x3  0x024 (to 0x128)   // to Swap
//0x108// addi x18 x0  0x014          // set the addr
//0x10C// addi x19 x0  0x038          // set the addr
//0x110// jal  x3  0x018 (to 0x128)   // to Swap
//0x114// addi x18 x0  0x018          // set the addr
//0x118// addi x19 x0  0x028          // set the addr
//0x11C// jal  x3  0x00C (to 0x128)   // to Swap
//0x120// nop
//0x124// jalr x0  x2  0x000          // return Random
//0x128// lw   x20 x18 0x000          // Swap
//0x12C// lw   x21 x19 0x000
//0x130// sw   x20 x19 0x000
//0x134// sw   x21 x18 0x000
//0x138// jalr x0  x3  0x000          // return Swap
```

採用這個策略之後，我又對gatelevel netlist進行了一次simulation，結果如下，可以發現由於bottleneck frequency的上升，有對critical path做處理的optimized baseline還是勝出。但現在的instructions的迴圈數量其實不多，如果說單個迴圈裡面的敘述不多(比方說三個instr就接beq/bne)，使用BPU會有更大的差距。
但實際上我們並不會把CPU永遠都用最快速度來跑，因為如果clock frequency極高的話會導致power的消耗很大，如果我們用一樣的frequency來考慮這兩個design，不論是省下來的cycle數還是time都是有做BPU的設計會勝出，並且差距蠻明顯的，這證明說在CPU中加入BPU實際上是對減少工作時間非常有效的，並且只需要用小小的一個2-bit predictor(以sdc=3.2合成面積約為26000 um^2^, 大約增加10%的面積)就可以達成此目的，實際上也有一些CPU中就是採用這種簡單的2bit -predictor，例如1993年的Intel Pentium。但是現在的CPU中為了達到更高命中率，幾乎都會採用更複雜的branch prediction unit。

	sdc cycle	tb cycle	Total time	Total cycle
optimized Baseline	2.6	2.61 (min)	5765.3	2208.9
optimized Baseline		3.31	7179.49	2169.0
with BPU	3.2	3.31	5924.53	1789.9

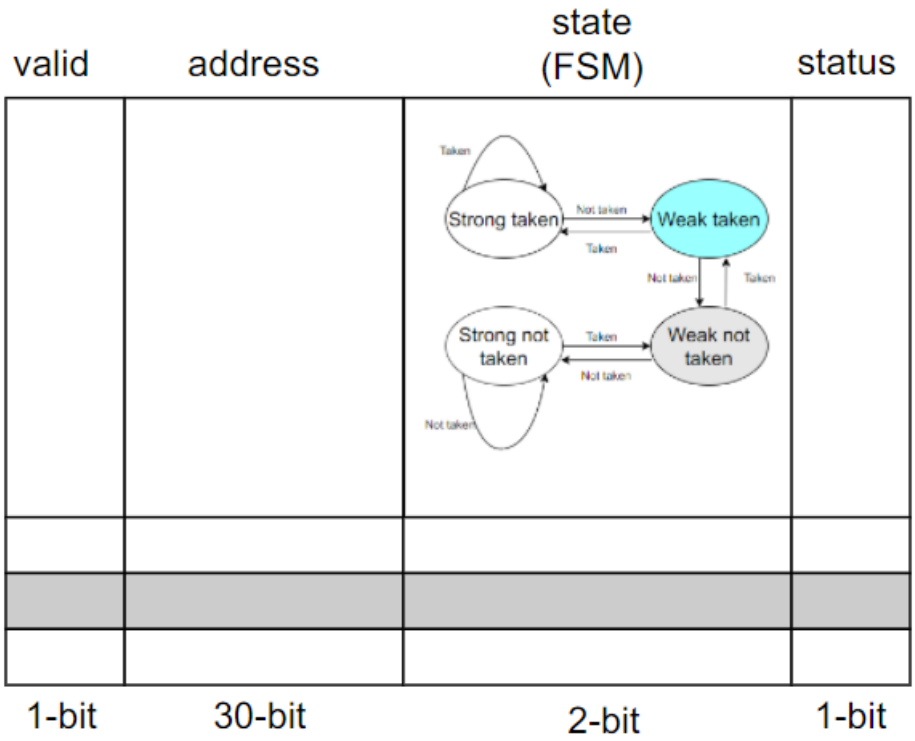
(F) Another architecture of BPU

在實際上跑instruction的時候，有可能遇到如上所述的interleaved的pattern，或是各種千奇百怪的pattern，原先設計的2-bit predictor只能用來應付幾種常出現的pattern，因此我對BPU又做了一個簡單的改進。我想要在BPU裡面加上有點像Buffer (BPB, Branch Prediction Buffer)的東西，用來記錄位址和預測結果，其中每一個位址都是用上面的2-bit predictor來預測，因此當我們第二次遇到這個addr的時候，我就可以去cache裡面找，並根據預測結果照著做，如果沒有的話(ReadMiss)就先猜測是Taken(因為對loop來說，比較常是Taken)，並把它加入這個buffer裡面。

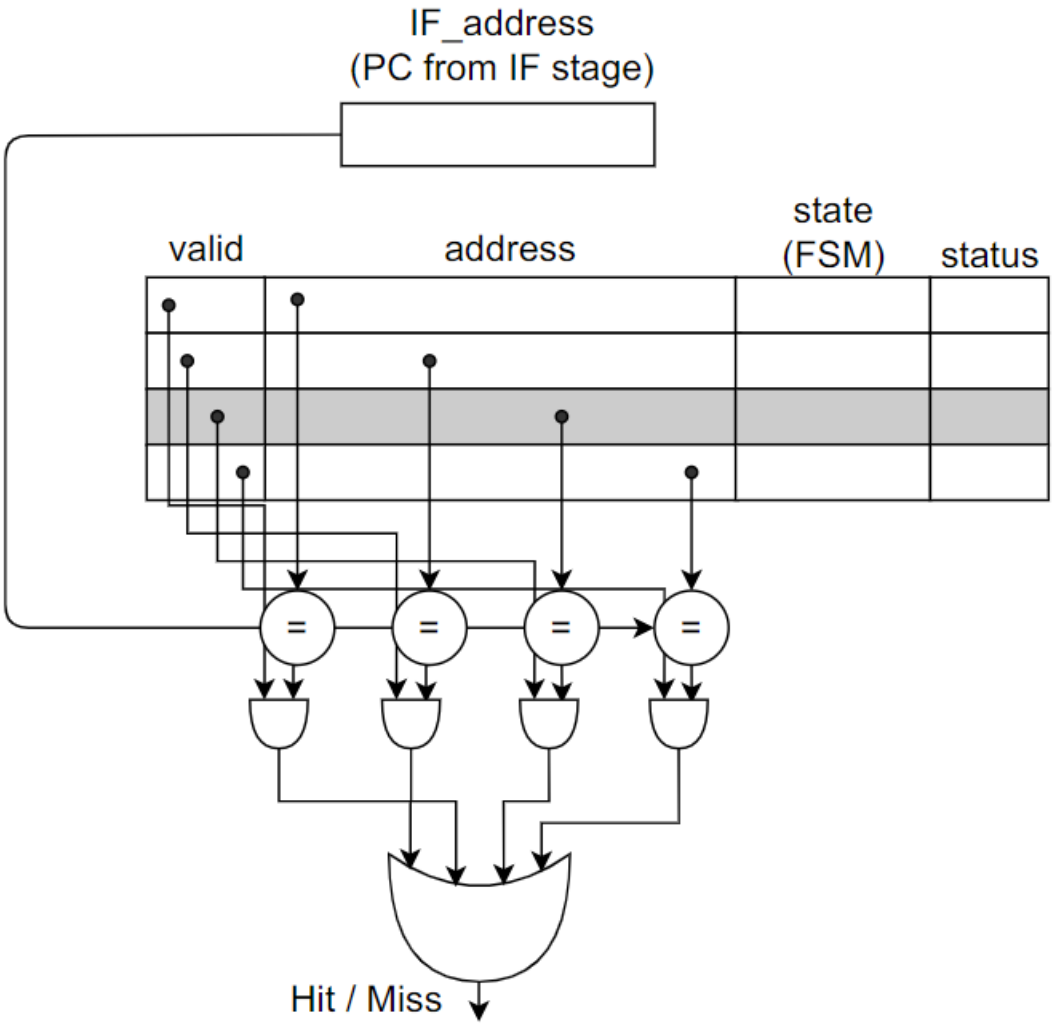
> 在面對單一addr要使用哪種預測法，考量到如果是1-bit predictor在離開loop和下次進來loop的時候都會各錯一次，對於迴圈次數小的迴圈，錯的比率會變得很高，因此我選擇替buffer裡面的predictor都各做一個2-bit predictor

這樣做的話對面對那些可能會讓2-bit predictor爛掉的case會有比較好的成效。比方是interleaved，由於interleaved是固定會不會Taken，因此用記錄位址的方法可以處理interleaved的case。

其實整個流程有點像設計cache給Branch prediction使用，Buffer Spec如下

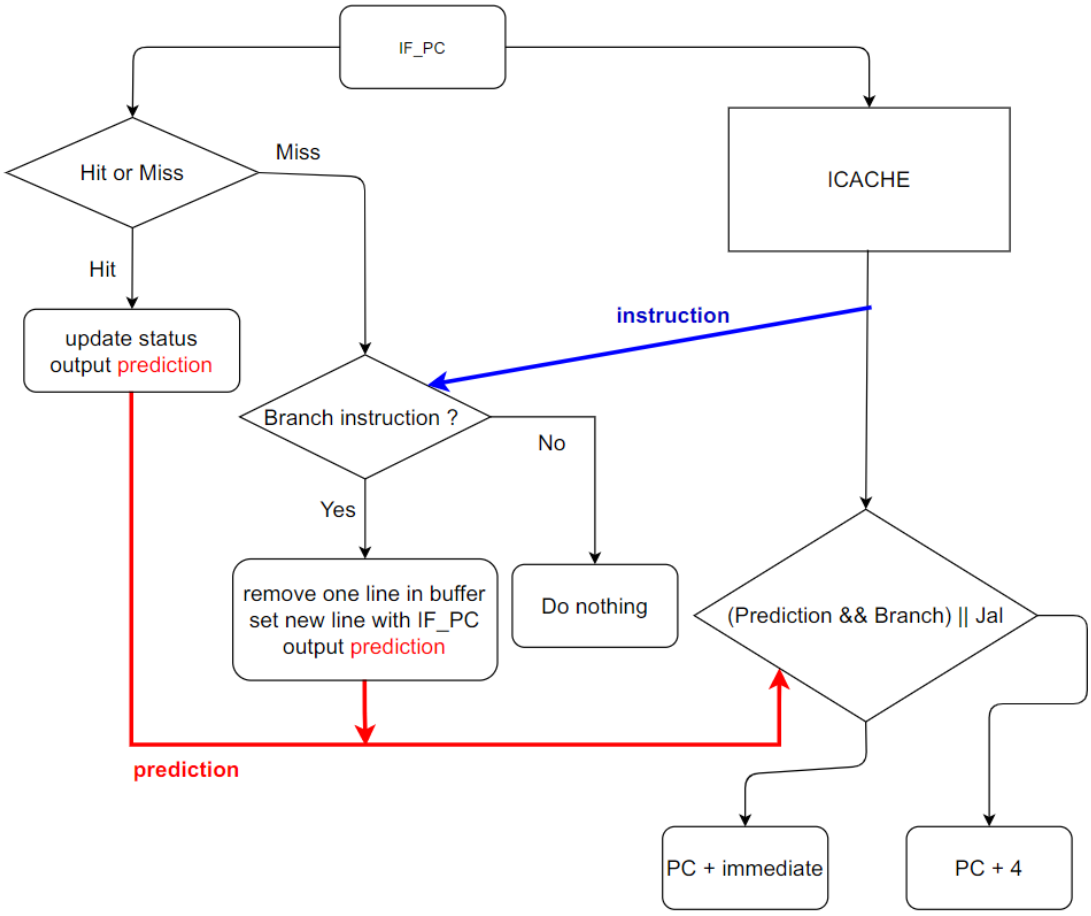


在決定現在的addr是否hit的時候，也和full-associative cache很像，我會檢查buffer中的4個addr有沒有和IF_addr相同，圖示如下。並根據hit或miss做出相對應的動作。



- ReadHit (PC Hit)
  - IF stage: 根據FSM的prediction來決定PC是否jump
  - ID stage: 根據預測正確或錯誤移動FSM中的state，決定要不要clear status
- ReadMiss (PC Miss)
  - IF stage: 先等到Icache讀出來，確定是否為beq/bne
  - IF stage: 把此PC加入buffer中，起始state為STRONG_TAKEN
  - ID stage: 根據預測正確或錯誤移動FSM中的state，決定要不要clear status
- Replacement Policy: Bit-Pseudo-LRU
  每一個cache line都有一個status bit來記錄使用狀況，當這個line被使用，會從0->1，同時看是否所有的status bit都是1，如果是的話就把所有的bit寫到0(clear)。
  當ReadMiss發生時，從status為0的cache line中取一個來取代(我是從line0照順序看到line3，有遇到status 0的就拿來用)。

- IF-stage flow chart



Preformance

將有BPB的設計和單純2-bit predictor的設計分別執行/a10b20c30的instructions以及I_mem_hasHazard的instructions後比較其cycle數(RTL, cycle = 10)，可以看到透過branch prediction buffer，可以更提高branch prediction的正確性，並減少執行所需cycle數，相對之下，由於需要decoder, multiplexer以及實際上有4個2bit-predictor在運作，合成出來的面積大約是2-bit predictor的三倍左右(理論上應該要4倍多一些...?，但他合成過程也不太確定發生了什麼事XD)。但是如果要使用這個BPB的結構其實還要考慮一個問題，就是branch 連續發生的數量，如果branch的數量和block數量相比差太多的話會造成裡面常常發生replace的情形，等於說我現在的prediction都還沒用到就會被取代掉，造成預測的效果不佳，因此實際上在電腦中設計時一樣必須配合compiler來設計，確保compiler編譯出來的assembly盡量避免太多同時會被用到的branch，並根據數值來設計branch prediction buffer的block數。

	a10b20c30	I_mem_hasHazard
BPB	230.5	1592.5
2-bit predictor	274.5	1725.5

Compression

(A) 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。例如在這次提供的Testbench中，指令長度就從11個block的instruction壓縮到8個block。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。

(B) Design Concept

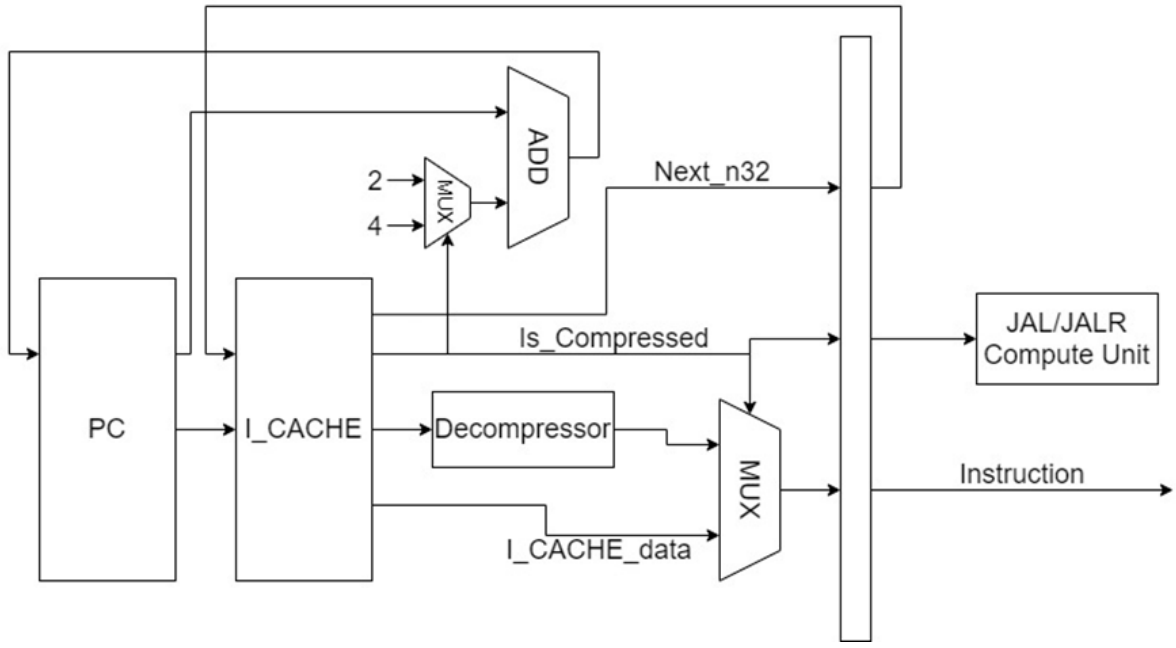
為了交互處理16bit與32bit的指令，以及對於這兩種指令對於Instruction Cache的位置對齊以及讀取方式，需要稍微思考下面幾個設計的重點：

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。這邊考慮使用修改ICACHE的部分來節省讀取時間。

(C) Design Details

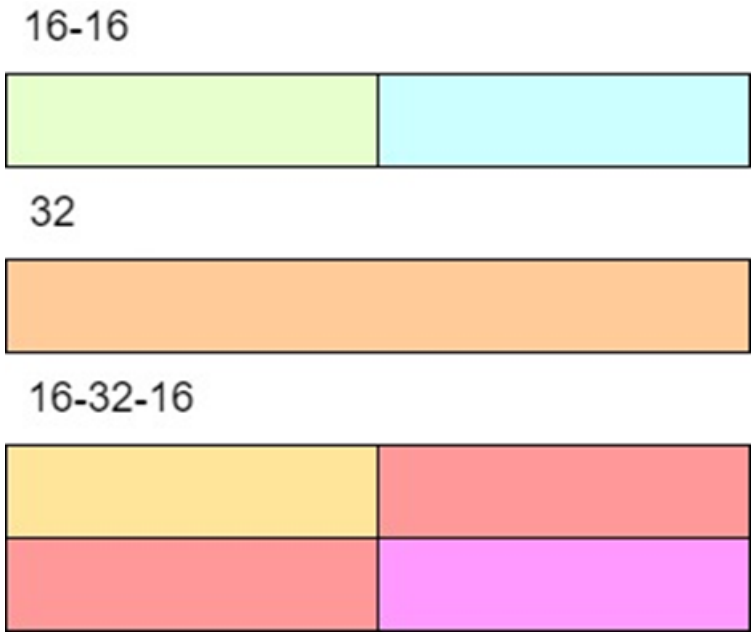
首先介紹decompressor的設計。
Decompressor是透過3 bit的function以及2 bit的op code來決定其指令，而在decompress的時候透過自行設計的測資發現到提供的decompressed instruction在處理c.mv時，與RISCV提供的add rd rs x0不同，decompressed instruction中使用的是addi rd rs 0x000，讓我在設計decompressor的測資時花了一點時間理解。

接下來介紹在主設計中，針對decompressor所做出的調整。

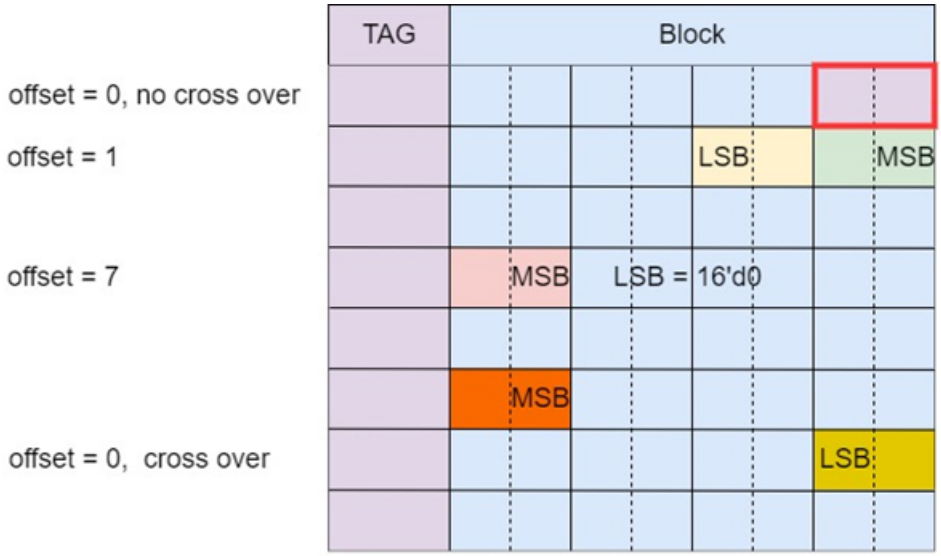


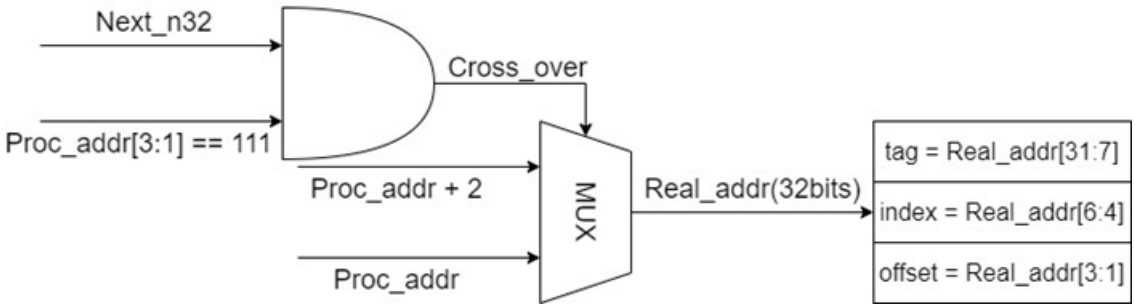
上圖為在主設計中額外的調整。可以看到，在ICACHE給出data之後，可以判斷出前面16 bits是否為compressed instruction，利用Is_Compressed這支訊號表示，而同時可以判斷後面16 bits是否為compressed instruction，如果否，就表示這16 bits為一個32 bits的instruction的部分，此時Next_n32就會為1，當然，在輸出為32 bits的non compressed指令時，無論後面16 bits為何，Next_n32會維持在0，因為無法判斷出下一個指令是否為32 bits。
在輸出data後，會將第31 bit至第16 bit送入decompressor，並且由Is_Compressed來決定使用的是decompressor的輸出的指令，或是ICACHE輸出的32 bits的指令。同時，會決定PC是+2還是+4。
Is_Compressed在經過IF/ID的Flip-Flop後，會讓JAL/JALR判斷要存入PC+2還是PC+4，而Next_n32則會輸入ICACHE作為其輸出data的評判標準。

接下來介紹ICACHE的調整。



首先要提到instruction的在一個word中幾種排列方式，前面的16-16以及32都是比較好處理的方法，最後的16-32-16或是16-32-32-16等等，則是需要特別處理的。尤其是當跨過word的insturction同時也跨過了ICACHE中的情況下，ICACHE需要同時處理兩個block。





為了處理不同的狀況，ICACHE被設計成輸出接下來32bit的資訊，無論下一個instruction為16 bit或是32 bit。如果offset，也就是address輸入時對齊的位置在7的位置，也就是address為0x?E，並且過了Flip-Flop的Next_n32輸入時，表示下一個instruction是會跨過block的32 bit instruction，此時由於這一條instruction的前半部分已經被存在CACHE裡面，所以需要去讀取的是下一個block的資料，所以設置一個Real_addr，應對需要讀取跨過block的資料，再根據Real_addr的tag、index以及offset決定讀取的資料方式。

當offset為偶數，且cross over不發生的情況下，讀取的即為block中完整的word。當offset為奇數，則需要讀取兩個word中的資料，如上圖所示，其中offset為7的時候，表示下一個指令為16 bit，此時僅需要讀出最後一個word的後16 bit，剩下的補0即可。當offset為偶數且cross over發生時，則會讀取下一個block，再將讀到的資料合併成32bit的instruction。

(D) Performance and Conclusion

	Value
Sdc Cycle(ns)	3
Design Area(μm^2)	18338.709635
Testbench Cycle(ns)	3
Decompression Execution Cycle	520.5
Compression Execution Cycle	431.5
Decompression ICACHE stall	14
Compression ICACHE stall	8

可以看到，設計上增加RISCV Compression的處理並沒有增加太多的面積，從原本的大約28萬增加到30萬，但是在表現上大約節省了20%的處理時間，原因可能是來自於ICACHE需要從memory讀取資料的次數變少了，所以使用Compression雖然會比較複雜，但是各種表現上都會有所進步。

L2 Cache

(A) Concept

在Baseline的設計中，我們使用了split L1 I-Cache D-Cache，但當Cache miss的時候會多5 cycles*的miss penalty。 在baseline的hasHazard測資中，D-Cache與I-Cache分別會有5, 19個miss，也就是說會多120個cycles的miss penalty，約佔整個process elapsed time的6%；而在L2 Cache的測資中(更大的fibonacci)分別miss了691, 20次，多了3555個cycles，佔了7.5%。
為了降低Cache的miss penalty，我們實做了L2-Cache。加了L2-Cache可以使得L1 miss但L2 hit的penalty減少*(80% better)，但同時也會使得L1 miss, L2 miss的penalty增加。故如果instruction太少或是data fetching的位置都一樣，使用L2-Cache反而會使得performance降低。

* 這邊只是以助教提供的slow mem為例，會隨著實際memory而有所差異，總之是個相較之下很大的penalty
* 由於memory fetching需要5 cycles, L2只需要1 cycle，故能優化80%

(B) Type of L2 Cache

在實做L2 Cache上有兩種作法，一種為unified Cache，也就是將D-Cache/I-Cache合併；另一種便是split Cache，將兩者分開。兩者分別的優勢大致上有以下(在總block數相同的前提下)：

- Unified:
  - **由於有較多的block數，故能有效降低miss rate**
  - 與下一級的儲存空間(L3 Cache or memory)的bandwidth較小
- Split:
  - 由於有較少的block數，在比較tag與address上的critical path較短
  - 由於同時能fetch D-Cache與I-Cache，故能有較大的bandwidth (僅在L1 D-Cache與I-Cache同時miss時發生)
  - **可以針對不用write的I-Cache進行優化**
  - 分開可以使得L2 Cache內邏輯變簡單，Combinational的邏輯部份面積變小

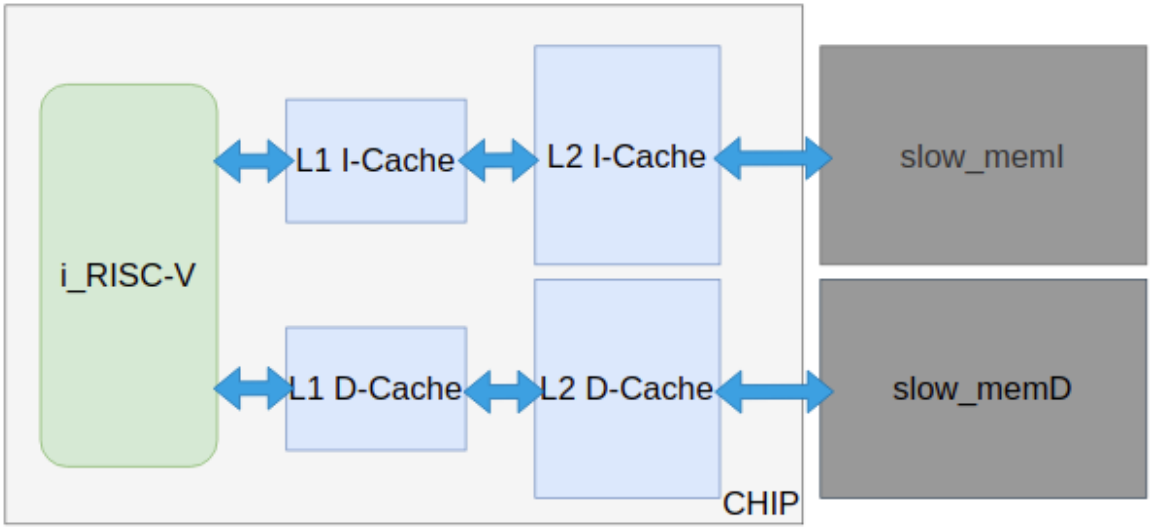
目前市面上的CPU都基於更高的hit rate而選擇unified L2 cache
(2002年intel Itanium 9000有曾經使用過split L2 cache)

> 事實上也有論文提出split的performance 較好，但似乎沒有很受重視 沒有citation QQ
> 或許這也代表研究與現實的差異
> Çano, Erion. "Simulation of L2 Cache Separation Impact in CPU Performance." International Conference on ICT Innovations. Springer, Cham, 2014.

(C) Implementation

(一) 基本實作 Split cache

由於split cache邏輯上比較直觀簡單，故我們首先以split cache的方式實作。
(I-Cache使用 directed map, D-Cache使用 2-way associative)



實做後我們發現，在助教提供的L2 Cache tb(instruction)和baseline的版本類似，也是做Fibonacci+bubblesort，只是L2的tb把數列的總數增加了，故在bubblesort時會存取到的資料變多。而正因為如此，事實上我們發現只有L2 D-Cache有實質幫助，**在I-Cache上實做L2 cache並不會提高performance**，因為迴圈中的instruction幾乎都是相鄰的addr，並且個數也不超過L1-Cache的capacity 32(L1 Cache的block數*4，一個block存4word)，故L1 miss的情況幾乎都是發生在fetch新的instruction，使用L2-Cache並不能增加performance。

相對的，實做D-Cache能提高約12%的performance，而L2 D-Cache的大小約在32個block(16 sets, 2-way associative)時達到最佳，繼續加大並不會有幫助，因此在這次project中L2 Cache大小以32 blocks為主。

面積的部份，我都使用clk=10去合成。
只有L1-Cache面積約240000(um^2)
加了L2 D-Cache則約540000(um^2)
>> 4KB L2 D-Cache約300000(um^2)

(二) 進階版tb

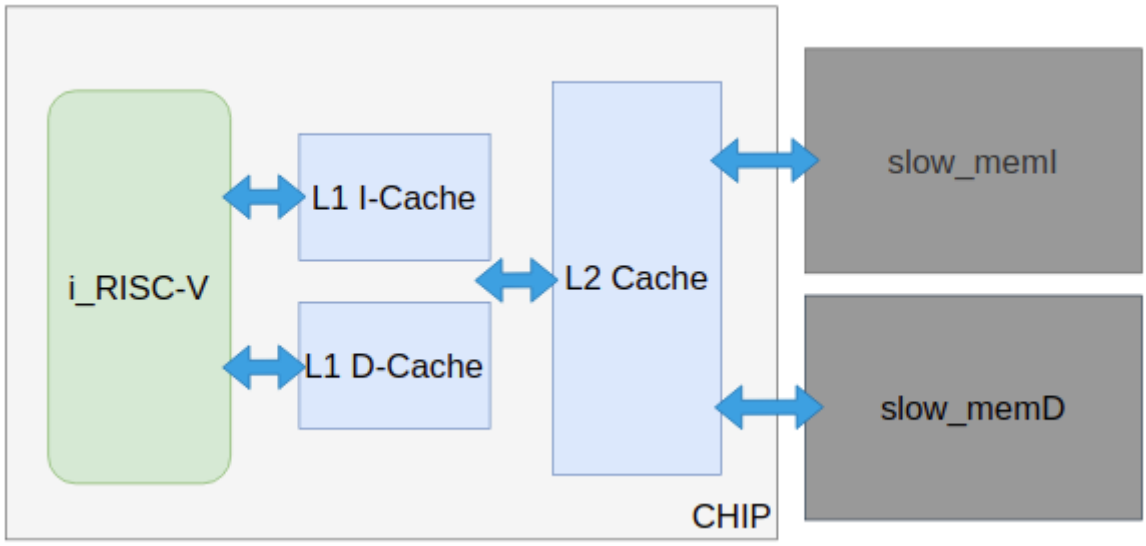
由於前面提到，使用L2 I-Cache不能使得performance增加。原因在於，在fibonacci與bubblesort的迴圈中instruction個數太少，導致迴圈內fetch instruction基本上都是hit。有鑑於此，我便在L2 Cache的tb(instruction)中，bubblesort的swap迴圈內增加了32個nop instruction，始得每次swap迴圈內的instruction個數超過L1的capacity。

在使用新的tb後(後面簡稱新的tb為modified L2)，在都使用L2 D-Cache的前提下，使用L2 I-Cache能使performance增加33%。

面積部份，相較只加了L2 D-Cache的540000(um^2)
再使用L2 I-Cache後面積為760000(um^2)
>> 4KB L2 I-Cache約220000(um^2)
(由於I-Cache可以省去寫入的邏輯，故較D-Cache小)

(三) 番外篇：Unified L2-Cache

由於市面上的CPU都選擇使用Unified的，故我們也implement了unified L2-Cache(total 64 blocks)，試著與split L2-Cache比較。



事實上，在現實的硬體架構中memory大多只有一個，不會像助教提供的分開成slow_mem_I, slow_mem_D，因此Data與Instruction使用的address不會重疊。然而在這個project中，address不共用，因此我們需要將存進L2-Cache的資料分別標注為I/D。

先前提到，Unified的優勢在於較多的block可以使得miss rate降低進而增加performance，缺點則在於面積可能會較大，並且tag matching的時間較長，bandwidth較小。

值得一提的是，在這兩個tb中也不會有I/D L1-Cache同時miss的情況(conflict)，所以unified bandwidth較小的缺點也沒有展現出來。事實上conflict的情況在L1常發生(所以L1 Cache才需要split)，但在L2發生的機率就很小了，所以L2-Cache才可以使用unified Cache。

經過實做與比較，我們發現在目前的兩個tb(L2 Cache, modified L2)中，unified與split在miss rate上並沒有差別，也就是說unified的優勢未能展現出來。(將split的I/D miss rate 加權平均後與unified比較)
因此我試著將L2的block size減少，變為split個別16 blocks與32 blocks的unified L2，由此一來便可以看出Unified hit rate較高的效益。

面積的部份，我們發現unified相較split大了許多，可能的原因有下：

1. split Cache可以化簡I-Cache 寫入的邏輯，但unified無法
2. 由於需要將資料分別標注I/D，導致邏輯更複雜(如下附圖)

儘管如此，市面上的cpu還是選擇使用unified L2-Cache，故可推斷實際使用上unified Cache 的performance有好到可以彌補area較大的問題。

unified 64 blocks

```
Number of ports:                         3208
Number of nets:                         64861
Number of cells:                        61944
Number of combinational cells:          47716
Number of sequential cells:             13882
Number of macros/black boxes:               0
Number of buf/inv:                       9774
Number of references:                       7

Combinational area:             463517.501331
Buf/Inv area:                    65950.779002
Noncombinational area:          377749.585773
Macro/Black Box area:                0.000000
Net Interconnect area:         9645349.345978

Total cell area:                841267.087105
Total area:                   10486616.433083
```

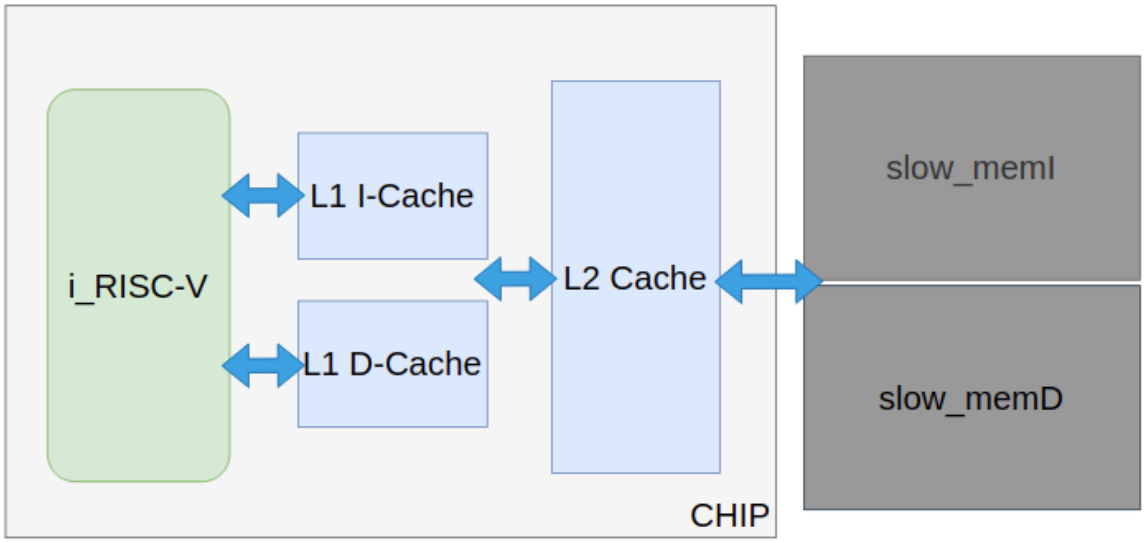
split 32 blocks

```
Number of ports:                         3209
Number of nets:                         56059
Number of cells:                        53286
Number of combinational cells:          38832
Number of sequential cells:             13850
Number of macros/black boxes:               0
Number of buf/inv:                       8168
Number of references:                      10

Combinational area:             376403.535350
Buf/Inv area:                    49144.821572
Noncombinational area:          384091.072567
Macro/Black Box area:                0.000000
Net Interconnect area:         8045794.260468

Total cell area:                760494.607917
Total area:                   8806288.868384
```

(由上圖可發現sequential部份面積差不多，但combinational部份unified明顯大了一些)

(四) 番外篇的番外篇： Unified L2-Cache 2.0

由於我們覺得unified L2-Cache的面積相較之下有點過大，並且實際上memory只會有一個，不像tb那樣分成slow_mem_I, slow_mem_D，因此我們將舊版的優化，並且把L2-Cache到mem的bandwidth變成一半。



結論，將L2-Cache到mem的bandwidth變小並不會對performance有很大的影響(因為I/D mem兩者conflict只在需要將dirty data寫回memory並接著讀出data時發生)，而面積也會因此縮小一點。

(D) 實驗數據

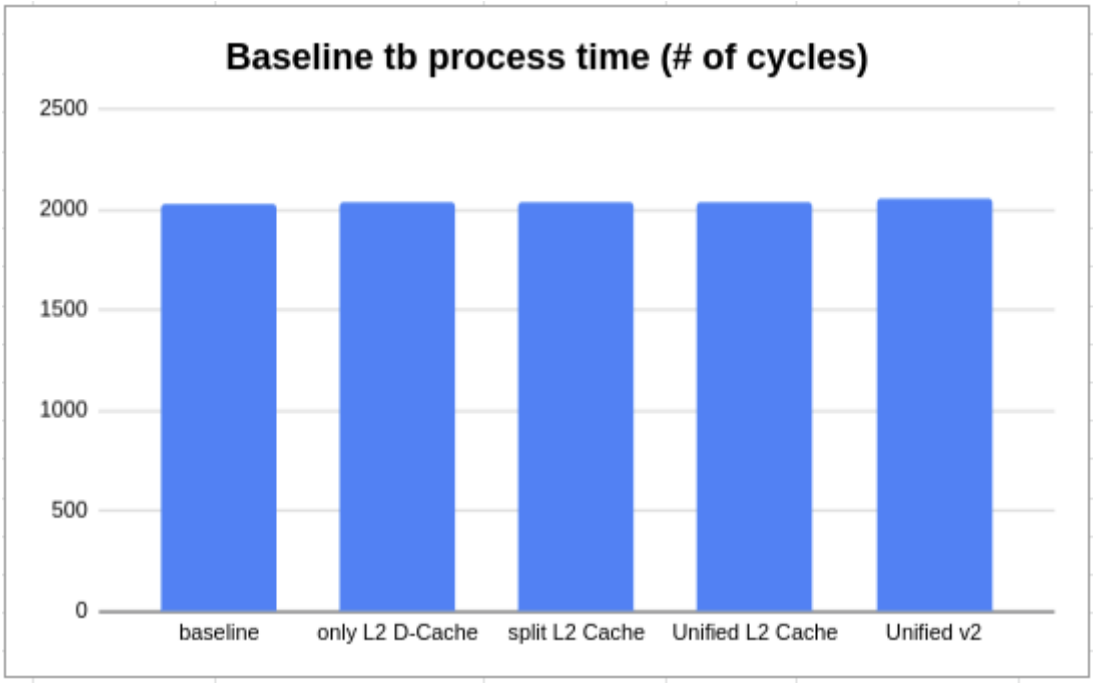
		Baseline			L2 Cache			modified L2		
SPEC	area (nm^2) (clk=10)	# of cycle	L1 miss rate	L2 miss rate	# of cycle	L1 miss rate	L2 miss rate	modified L2	L1 miss rate	L2 miss rate
baseline	238610.3076	2034	0.93%/0.92%	-/-	47230	0.04%/5.38%	-/-	240535	12.84%/5.38%	-/-
only L2 D-Cache	538856.6029	2039	0.93%/0.92%	-/100%	41863	0.05%/5.38%	-/1.54%	235168	13.33%/5.38%	-/1.54%
split L2 Cache	760494.6079	2036	0.93%/0.92%	77.8%/100%(82.6%)	41863	0.05%/5.38%	80%/1.54%(2.67%)	160052	13.33%/5.38%	0.13%/1.54%(0.2%)
Unified L2 Cache	897155.6825	2041	0.92%/0.92%	82.60%	41863	0.05%/5.38%	2.67%	160052	13.33%/5.38%	0.20%
Unified v2	841267.0871	2062	0.92%/0.92%	82.60%	41902	0.05%/5.38%	2.67%	160099	13.33%/5.38%	0.20%
split L2 Cache	519253.3305	2036	0.93%/0.92%	77.8%/100%(82.6%)	43243	0.05%/5.38%	80%/11.55%(12.54%)	161447	13.2%/5.38%	0.144%/11.55%(0.92%)
Unified v2	550779.1403	2062	0.92%/0.92%	82.60%	41944	0.05%/5.38%	2.95%	161185	13.2%/5.38%	1.10%
* miss rate: I-Cache/ D-Cache (weighted average) * 16 blocks respectively										

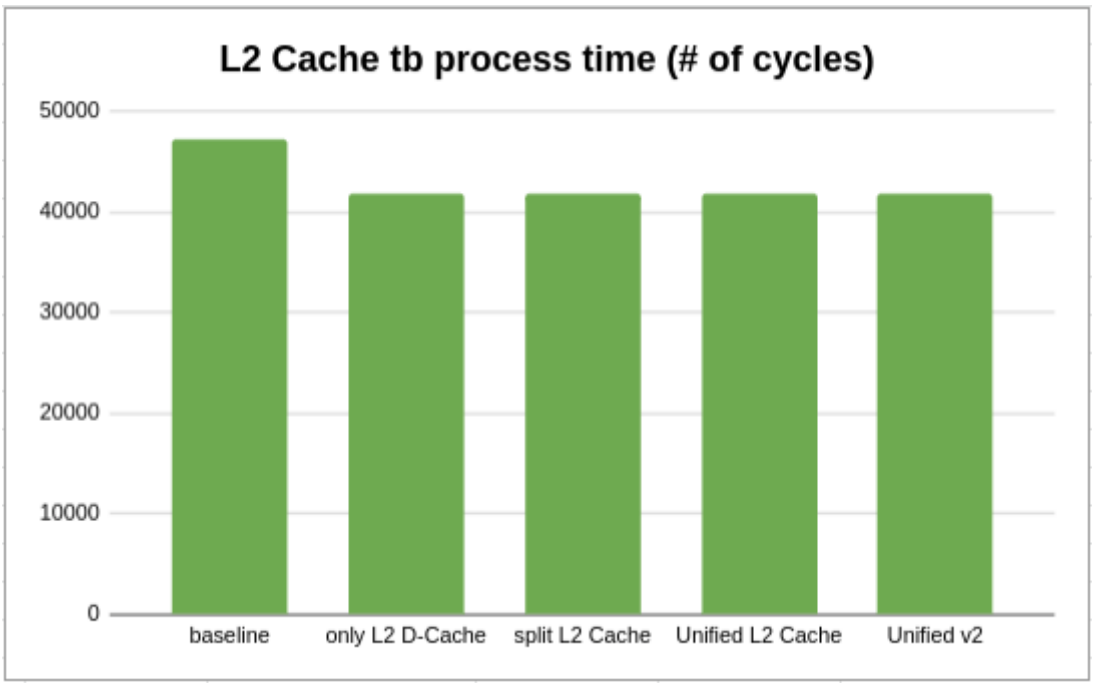
表格放大

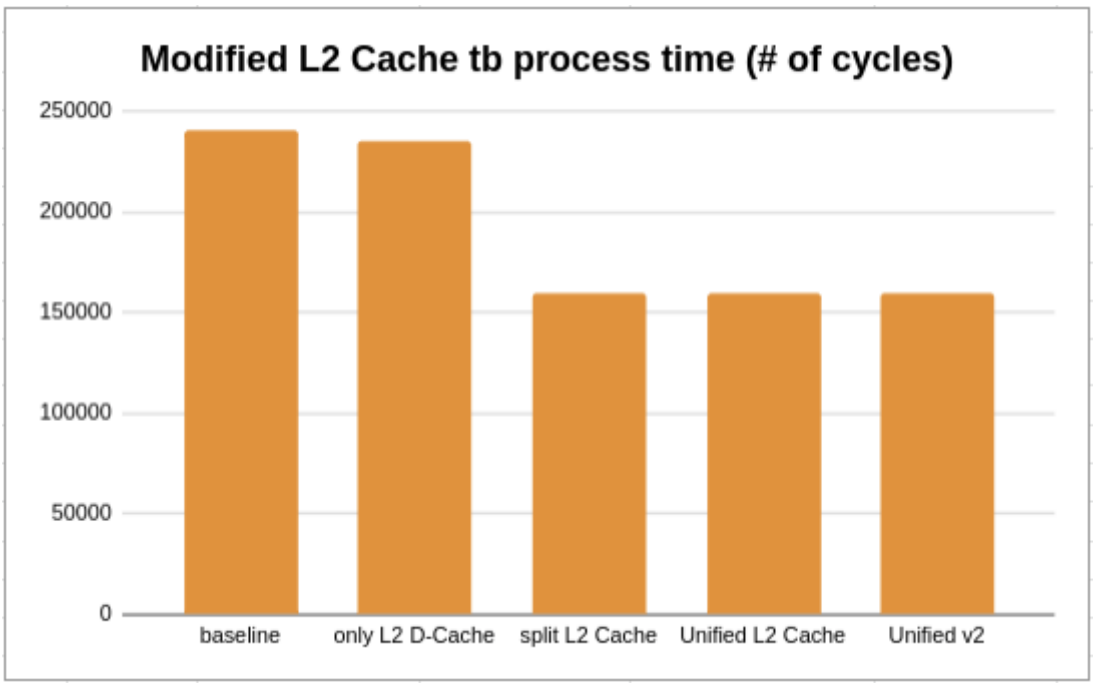
備註：

1. 每一列代表同一個spec下的實驗結果
2. 第一列中Baseline, L2 Cache, modified L2分別代表三個tb的instruction:
    - Baseline: Fibonacci+bubblesort (n=15)
    - L2 Cache: Fibonacci+bubblesort (n=19)
    - Modified L2: Fibonacci+bubblesort(nop inserted) (n=19)
3. 面積皆使用clk=10去合成
4. L1-Cache分別 8 blocks, L2-Cache分別 32 blocks (unified 64 blocks)，藍色部份則為分別16 blocks (unified 32 blocks)
5. I-Cache皆使用directed map, D-Cache皆使用2-way associative
6. Spec:
    - baseline: 只有L1 I/D Cache
    - only L2 D-Cache: baseline + L2 D-Cache
    - split L2 Cache: only L2 D-Cache + L2 I-Cache
    - Unified L2 Cache: 結合 L2 I/D Cache
    - Unified v2: 優化版本, 並將L2-Cache - memory間的bandwidth減半

以下長條圖是將上面表格中前六個spec做的整理(L2 Cache皆為total 64 blocks)


Baseline tb process time (# of cycles)
2500
2000
1500
1000
500
0
baseline
only L2 D-Cache
split L2 Cache
Unified L2 Cache
Unified v2


L2 Cache tb process time (# of cycles)
50000
40000
30000
20000
10000
0
baseline
only L2 D-Cache
split L2 Cache
Unified L2 Cache
Unified v2


Modified L2 Cache tb process time (# of cycles)
250000
200000
150000
100000
50000
0
baseline
only L2 D-Cache
split L2 Cache
Unified L2 Cache
Unified v2

(E) 數據分析

(一) L2 D-Cache的效益：baseline vs only L2 D-Cache

	Baseline tb			L2 Cache tb		
SPEC	# of cycle	L1 miss rate	L2 miss rate	# of cycle	L1 miss rate	L2 miss rate
baseline	2034	0.93%/0.92%	-/-	47230	0.04%/5.38%	-/-
only L2 D-Cache	2039	0.93%/0.92%	-/100%	41863	0.05%/5.38%	-/1.54%
* miss rate: I-Cache/ D-Cache						

- 這邊比較只有L1 I/D cache(baseline) 與增加了L2 D-Cache後的差異
- 從L2-Cache的tb中可看出，由於L1-Cache相同，故有相同的miss rate。但不同的是，L2 D-Cache的miss rate極低，代表我們可以有效降低L1 的miss penalty，如紅框所示，約優化12%。
- 跑Baseline的tb時，由於instruction過於簡單，L2-Cache並沒有幫助，反而增加miss penalty，故會花較多的cycle
- 在表中可發現有L2 D-Cache時L1 I-Cache的miss rate會略為增加，主要原因是：由於在D-Cache stall時，對RISC-V processor來說需要stall每級pipeline的processing，故在IF stage會重複fetch相同的instruction。而在增加了L2 D-Cache後，可以使得D-Cache stall的時間變短，因此重新fetch相同instruction的次數降低。在這樣的情況下，加了L2 D-Cache後由於miss的次數相同，而總次數減少，故miss rate較高，但其實總stall的cycle不變。

(二) 迴圈更長的instructions帶來的差異： L2 Cache tb vs modified L2 tb

	L2 Cache tb			modified L2		
SPEC	# of cycle	L1 miss rate	L2 miss rate	# of cycle	L1 miss rate	L2 miss rate
only L2 D-Cache	41863	0.05%/5.38%	-/1.54%	235168	13.33%/5.38%	-/1.54%
split L2 Cache	41863	0.05%/5.38%	80%/1.54%	160052	13.33%/5.38%	0.13%/1.54%
* miss rate: I-Cache/ D-Cache						

- 前面提到，由於助教提供的L2 Cache tb中instuction不夠複雜，L2 I-Cache沒有帶來效益，故我將原本的tb稍做修改，使得loop內的instruction數量變多
- 觀察split L2 Cache與only L2 D-Cache，發現在L2 Cache tb中L2 I-Cache的miss rate是80%，並且L2-Cache hit帶來的效益也是80%(前面有提到)，因此兩者抵銷後可以發現performance完全沒有變好(clk數相同)。
- 至於在modified L2 tb下，可以發現L2-Cache的效益提高了很多，相較沒有L2 I-Cache減少了33%的run time(如紅框所示)

(三) 兩種L2 Cache的比較： Split L2 Cache vs Unified L2 Cache

		L2 Cache tb			modified L2		
SPEC	area (nm^2) (clk=10)	# of cycle	L1 miss rate	L2 miss rate	modified L2	L1 miss rate	L2 miss rate
split	760494.6079	41863	0.05%/5.38%	80%/1.54%(2.67%)	160052	13.33%/5.38%	0.13%/1.54%(0.2%)
unified	897155.6825	41863	0.05%/5.38%	2.67%	160052	13.33%/5.38%	0.20%
* miss rate: I-Cache/ D-Cache (weighted average)							

- 這邊比較兩種implementation的差異
- 相較split，unified的面積較大
- 由於目前三種tb皆不夠複雜，故可發現miss rate(有將split中I/D兩者加權) 完全相同，故無法體現Unified帶來的好處

(四) L2-Mem bandwidth減少帶來的差異： Unified L2 Cache vs Unified v2

		L2 Cache tb			modified L2		
SPEC	area (nm^2) (clk=10)	# of cycle	L1 miss rate	L2 miss rate	modified L2	L1 miss rate	L2 miss rate
unified v1	897155.6825	41863	0.05%/5.38%	2.67%	160052	13.33%/5.38%	0.20%
unified v2	841267.0871	41902	0.05%/5.38%	2.67%	160099	13.33%/5.38%	0.20%

- 由於優化過，故v2面積略小

- 由於L2-memory間的bandwidth減少，所以當同時fetch instruction與data時會使得cycle變多，但其實影響並不大，幾乎可以忽略。
- 事實上本來就只有一個memory，所以理論上v2更貼近現實應用

(五) Unified 的強大之處： 16 blocks split vs 32 blocks unified

		L2 Cache tb			modified L2		
SPEC	area (nm^2) (clk=10)	# of cycle	L1 miss rate	L2 miss rate	modified L2	L1 miss rate	L2 miss rate
split	897155.6825	43243	0.05%/5.38%	80%/11.55%(12.54%)	161447	13.2%/5.38%	0.144%/11.55%(0.92%)
unified	841267.0871	41944	0.05%/5.38%	2.95%	161185	13.2%/5.38%	1.10%
* miss rate: I-Cache/ D-Cache (weighted average)							

- 由於前面沒能看出來Unified的優勢，無法說服我們目前市面上都使用Unified，所以我們將block size由32變成16(前面討論的皆是block size = 32的情況)
- 在L2 Cache的tb下，可以看到block size相同的情況下，Unified的hit rate明顯較高(紅框部份)
- 至於在modidfied L2 tb下，由於I-Cache使用率增加，導致在Unified中的資料都是instruction，因此Data的hit rate降低而造成overall的效益沒有split來的好。不過事實上也僅差一點點而已，平均下來Unified的效益還是較高。

(F) 結論

事實上我們都知道這三個tb很侷限，都不是general case，不夠貼近真實的使用狀況。但從中我們還是能得出與我們推論相符的結果。大致上得出的結論如下：

1. L2 Cache在instruction夠複雜的情況下效果明顯
2. L2 Cache的block size取決於instruction多複雜，針對目前的測資我們發現total 64 blocks就已經足夠，但主要還是要看使用者的情況
3. 在general case下，unified cache hit rate會比split cache低許多，我想這也是市面上大多使用unified L2 Cache的原因
4. 儘管L2 Cache的面積比起RISC-V processor更大，用AT值去衡量L2 Cache的效益一定是較低的。但對於使用者來說時間會比面積更加寶貴，並且實際上miss penalty可能會更高，並且L1 Cache因為需要即時給資料的原因不能做太大，故L2 Cache還是有存在的必要。