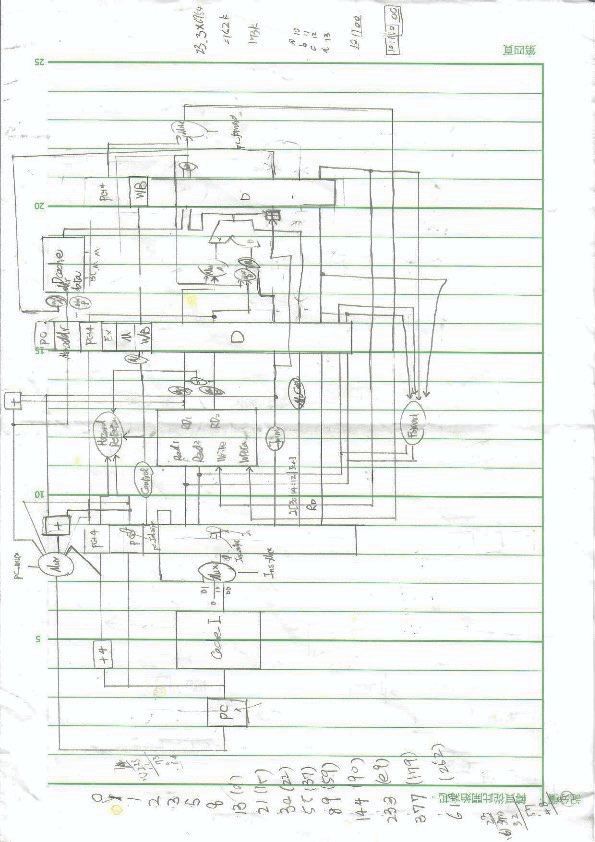
**108-2 Digital System Design Final Project Detailed Report**

RISC-V Group3

B07901100 林亮昕、B07901048 莊哲維、B07901056 張凱鈞

1. **Baseline**
2. **Design**

我們發現之前的5stage pipeline架構有許多不足的地方，於是我們提出了一些改善方法使得新的CPU架構的效能更好，以下是我們的設計藍圖以及實施方式，至於效能分析的部分將在Baseline的第二部分有更詳盡的說明。



1. Jalr 移至ID stage

由於branch的data path在ID stage，我們將Jalr 同樣移至ID stage，如此一來可發現兩邊都各有一個加法器處理jump（藍框）或branch（紅框）的PC，並不會造成critical path 增長太多。

1. 4 satge CPU

將Dcache（橘框）移至EX stage 且並排於ALU，原因在於5stage instruction 只有lw跟sw，而他們資料讀取的位置（Data\_address=Immediate+Regsiter\_data）並不需要ALU運算獲得，可以與先前提到Jalr data path （jump\_address= Immediate+Regsiter\_data）做資源共享，直接將Jalr中加法器的輸出端(jump\_address)經過pipeline接到Dcache的位址輸入端（Data\_address）。

1. 增加一條forward data path 給Data\_address

由於Data\_address同樣可能遇到hazard的問題，我們可以將上一個cycle的結果加上目前的immediate作為下次forward 位址的值（黃框），只需要在ALU後面加上一個小的adder即可（可以根據程式的大小以及資料量調整adder的大小，雖然critical path會增長但不至於太多，詳細可見第二部分Analysis）

1. Branch Acceleration

我們有想到一個方法可以不用Branch prediction但又可以不用flush 多餘的instruction，我們取名為Branch Acceleration。方法如下：首先，當第一次branch的時候先將此PC位址記下，方便以後認出這個有可能會要branch 的PC位址（Ａ），其次，等到下一個cycle來臨時，紀錄剛剛跳到的PC位址（Ｂ），以及其經過Icache得到Insstruction（Ｃ），還有再下一個PC的位址（Ｄ＝Ｂ＋4）。最後，只要之後再碰到這個可能要branch的PC位址（Ａ）而且確定要branch，那麼此時就不用flush多餘的instruction，只要將ID stage 的PC以及instruction換成（Ｂ）與（Ｃ），並且將IF stage 的PC換成（Ｄ）即可順利地執行之後的程式。此外，如果碰到一個不是Ａ卻要branch的地方時，我們將更新ＡＢＣＤ，以確保資料能夠預測最近一次要branch的狀況。以上的功能如果換成jal一樣也能運作，因此在CHIP裡我們會instantiate兩個Branch Acceleration的module 一個for branch另一個for jal。

1. Pipelining Forward Unit

如果ID階段與Ex階段寫入的register位址相同（有forward），那麼下一cycle時則會輪到Ex階段與WB階段寫入的reg位址相同，因此我們只需要一個比較ID和Ex階段的forward unit，再將forward訊號經過pipeline傳給需要forward的ALU和Dcache即可。此外，當write back位址跟register讀取位址相同時，forward unit 也會對register的output做forwarding。

1. 使用direct map cache
2. 由於2wat cache要比較兩個tag需要一點時間，我們選擇只需要比一個tag就好的 direct map cache。
3. 減少 PC 的 width

當程式的instruction數目不是很多時，我們可以紀錄PC前面幾個bit就好而不需要用到全部32 bit，這能夠同時減少面積以及cycle的時間

1. 減少cache 中tag的width 以及比較data時的width

同（六），在一個Instruction小於64或是128的程式中，我們可以減少資料需要比較的值已達到更好的performance。

1. **Analysis**

(一)目前效能最好的CHIP

|  |  |
| --- | --- |
| synthesis cycle(ns) | 2.5 |
| pass cycle(ns) | 2.63 |
| no hazard time(ns) | 603 |
| has hazard time(ns) | 5651 |
| Area(k) | 241 |
| AT(no)(k\*ns) | 145323 |
| AT(yes) (k\*ns) | 1361891 |

(二) 5stage CPU v.s 4stage CPU

|  |  |  |
| --- | --- | --- |
| 配置 | dm cache+5stage | dm cache+4stage+Branch Accerelation |
| synthesis cycle(ns) | 2.5 | 2.5 |
| pass cycle(ns) | 2.6 | 2.7 |
| has hazard time(ns) | 6584 | 5651 |
| Area(k) | 264 | 241 |
| AT(yes) (k\*ns) | 1738170 | 1361891 |

從以上可以看到，雖然4stage的critical path 因為forward Dcache address 而略微增加，但是減少的時間卻非常多，大約減少了6584/2.6-5651/2.7=440個cycle，可見解決掉load use hazard（4stage）以及不必要的flush pc（Branch Acceleration）後的確能幫助CPU的效能大幅提升（AT值大幅下降）。

1. **Extension ─ Branch Prediction**
   1. **Design**

在Pipelined RISCV Chip中，如果出現branch operatiion，勢必會空掉一個clock cycle來清除前一個錯誤的指令（IF/ID reg），為了優化CPU的效能，我們置入branch prediction model期望能夠減少flush的cycle數·。

如果期望能夠避免空掉的cycle，我們必須在IF stage就完成指令解碼，才能及時處理並獲得正確（預測應要獲得）的下一指令，在此架構下，除jalr因需要使用register file內資料而無法提前處理外，其他跳轉類指令都能夠提前解決或作出預測，藉以達成減少執行cycle數的目的。

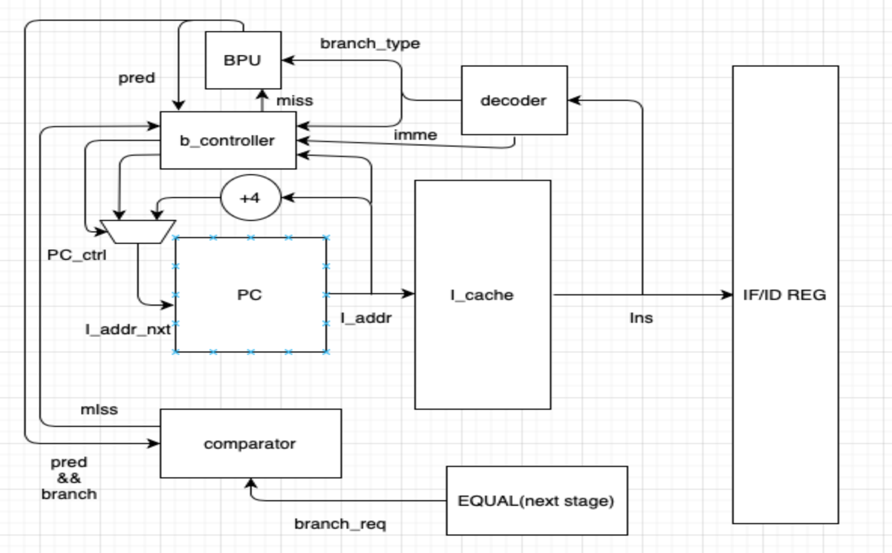
* 1. **Structure and Implementation**

1. 預測模型

我們本次所採用的預測模型是2 bits predictor，只有在連續兩次的錯誤預測後才會改變預測結果，此模型在一般而言有著不錯的穩定性，不會因為特定形式的程式表現有太劇烈的變化，並且有不算大的面積。



1. 電路架構

整體而言，branch predictor電路的目的是要正確地控制I\_addr\_nxt，為此我們設計了下圖的電路：

首先由decoder將指令解碼，傳送指令類型至BPU與branch\_controller，若指令為beq或bneq則訊號branch會等於1，啟動預測與回授控制，I\_addr\_nxt會等於branch\_controller傳出的Imm + PC。在啟動預測的狀態下，comparator會收到ID state的branch request信號，與預測比對能得出預測結果的正確性，決定是否要flush，並且改變BPU的state。

* 1. **Analysis**

|  |  |  |
| --- | --- | --- |
|  | Baseline | Branch pridiction |
| Area | 241977.948685 | 266145.529653 |
| Cycle number(hazard) | 2363 | 2051 |
| Cycle number(BrPred) | 350 | 264 |
| A\*T | 2287176056 | 2183461784 |
| A\*T | 338769200 | 281050176 |

從上表中可以發現，branch prediction的存在一定程度增加了面積，但是在執行cycle上卻也有不小的幫助，在面積允許的情況下是加速的好選擇。

然而在極限追求cycle time的狀況下，因為他增加了IF stage的critical path，合成出的最小cycle受限於IF stage，沒有辦法達成與baseline相同的執行速度。

1. **Extension ─ Compressed Instruction**
   1. **Design**

當我們有了可以通過baseline、並能應付Hazard的Pipelined RISCV Chip後，要接著實作出可以處理16-bits指令的Chip，必須解決三個問題：Extract information from encoded in 16-bits、PC Increment、Address alignment。以下將分別敘述我如何解決這些問題：

1. Extract information from encoded in 16-bits

我採用的是decompressor的方式，將16種可能的compressed instructions一一對應到32-bits的指令。每次從instruction cache讀出compressed instruction後，就會先經過decompressor轉換成32-bit的指令，再進行接下來的運算。

1. PC Increment

若讀到了compressed instruction，則接下來的要讀的instruction的address為PC+2(假設沒有跳轉)，而非一般的PC+4。我的實作方式是在下一個PC前加上一個multiplexer，控制訊號為「當前指令是否為compressed」，是的話則傳出PC+2，反之則為PC+4。

1. Address alignment

由於compressed instruction的存在，所以在instruction memory中，指令可能會從任何2的倍數的address開始。因此，一個32-bit的指令就可能會橫跨兩個word，甚至是橫跨兩個cache block。所以必須要特別處理這類指令(以下簡稱為CBI, cross-boundary instruction)的讀取。

我設計了三種不同的策略，以下分別敘述它們的設計架構及實作方式：

*Strategy 1、Bubble (CHIP\_Bubble.v)*

(1) 設計理念：

要讀取一個CBI，最直接的方法就是花兩個cycle的時間來讀取。這樣一來，包括chip傳送給cache的address、cache內部的設計等等，基本上都跟原先的baseline沒什麼差別，只需要在遇到CBI時，先用一個buffer把第一個cycle讀到的word的後16 bits (也就是這個CBI的後半段)存起來，下一個cycle再讀取前半段的16 bits並跟buffer存的16 bits連接成一條32-bit instruction。

(2) 實作方法：

　　在chip中加入一個判斷電路。當遇到CBI的時候(即當前PC為PC+2且ICACHE\_rdata末兩位為11)，則

* + - 1. 把“instruction\_state”設為1 (用來顯示目前是不是正在讀CBI)。
      2. 把當前的ICACHE\_rdata後半段存入buffer。
      3. 把下一個指令設為NOP，也就是在下一個cycle送出一個bubble，才能夠同時從cache中讀取剩下的16 bits。
      4. 暫停增加PC一次(因為下個cycle仍然是在讀同一個指令)。

到了下個cycle，當chip發現instruction\_state為1時，傳給I\_CACHE的address會改變為PC[31:2]+1(下一個word)，同時把buffer中的值拿出來與目前的ICACHE\_rdata組合成32-bit的指令。

*Strategy 2、Buffer (CHIP\_Buffer.v)*

1. 設計理念：

　　在bubble strategy中，每次遇到CBI時，都需要花費兩次cycle的時間，相當浪費。而在bubble strategy中，利用了一個buffer來暫存CBI的一半指令，但除此之外就沒有再利用到這個buffer了。

因此，在buffer strategy中，每當從PC+4的address開始讀值並且該address的前半為compressed instruction時，或是從PC+2開始讀一個CBI時，就會把剩下的16 bits存入buffer。在下一個cycle(假設無branch)，若buffer中的16 bits為一個CBI的後半時，就會到PC[31:2]+1(下一個word)的位置讀取這個CBI的前半。如此一來就不需要花費兩個cycle來讀取CBI了。

1. 實作方法：

與bubble類似，同樣是加上一個判斷電路；此外會記錄buffer中儲存的是哪個address的指令(buffer\_tag)：

1. 在PC+4位置且前半為compressed instruction時，分別儲存後半的指令及當前位置到buffer及buffer\_tag。
2. 在PC+2時，先檢查當前位置是否與buffer\_tag一致：
   1. 相同：若buffer儲存的是compressed instruction，則直接decompress並執行；若為CBI後半，則讀取下一個位置的前半指令來組合，並儲存下一個位置的後半指令及下一個位置到buffer及buffer\_tag。
   2. 不同：從cache讀取目前PC位置的inctruction，若為compressed instruction，則直接decompress並執行；若為CBI後半，則送出NOP指令、暫停增加PC，並把目前位置的16 bits存入buffer。(在下一個cycle就會變成當前位置與buffer\_tag一致的情形了)

*Strategy 3、Halfword Cache (CHIP\_Halfword.v)*

1. 設計理念：

　　Buffer strategy在絕大多數時候都不需要多花費cycle讀instruction，但有個例外：當有跳轉指令恰巧跳到某個PC+2位置，且該位置是CBI的後半時。由於是跳轉，所以此時buffer儲存的指令不會是該位置的指令，因此就必須花費兩個cycle來讀取該指令。

為了解決這個buffer strategy的弱點，我們決定從cache下手──把cache改成可以從任意2的倍數的address輸出(因此稱為Halfword Cache)。這樣一來，即使是CBI指令，也可以在一個cycle裡就一次輸出了。

1. 實作方法：
2. 首先，因為可能會從halfword的位置開始讀取，所以從chip傳到cache的address值就得多往下一位。然而，chip與cache間的address bits數是固定的，因此我就省略了PC的MSB(也就是從PC[31:2]變成傳PC[30:1])。由於instruction memory相較於PC的位數可以涵蓋的範圍小非常多，因此省略掉PC的MSB(始終為0)並不會造成任何影響。
3. 在cache中，就依照傳入的address，輸出該address後的32 bits。需要特別處理的是這32 bits會橫跨兩個cache的block時，如下圖：(每個格子都是16 bits，橫坐標軸為proc\_addr的末三位，縱坐標軸為cache index)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 000 | | 001 | | 010 | | 011 | | 100 | | 101 | | 110 | | 111 | |  |
| 1 | |  | |  | |  | |  | |  | |  | |  | |  | |
| 2 | |  | |  | |  | |  | |  | |  | |  | |  | |

當讀取index 1的111位置時，要輸出的就會是index 1的最後16 bits及index 2的最前面16 bits，即黃色的部分。這種情況下，就必須要先確認下一個index所儲存的是不是我們想要存取的address。

因此，我在原先的finite state machine多加入了一個state，當遇到上述的情形時，且下一個index所儲存的不是我們想要存取的address時，就會進入這個state、向memory讀取該address儲存的instruction。

* 1. **Analysis**

1. Result
   * + 1. 說明：
2. 以下結果中，decompression指的是baseline的版本(1.0版)、跑I\_mem\_decompression的結果。三種可以處理compression instruction的chip都是從這個版本的基礎上修改而成。而同時組員又對Baseline進行了數個版本的改進。因此，這裡decompression的結果與前面baseline的結果會有所差異。
3. 各圖表意義：

*Fig. 3-1*：在.sdc檔cycle固定為4.0 ns的情況下，所合成出來的各個chip的面積。

*Fig. 3-2*：Baseline跑I\_mem\_decompression及其他三種chip跑I\_mem\_compression所需的cycle數量。這裡沒有使用總時間是因為各種chip在post-synthesis simulation中能通過的cycle大小未必相同，為了屏除在合成方面的影響，因此使用cycle數量來進行比較。

*Fig. 3-3*：Fig. 3-1的area乘上Fig. 3-2的cycle number所得的結果。

*Fig. 3-4*：在I\_mem\_compression中，並沒有出現buffer strategy的弱點──也就是跳轉到PC+2位置、讀CBI的情況。因此，我稍微調整了I\_mem\_compression的指令順序，改寫成一份有這種情形的I\_mem\_compression\_2.txt。這張圖就是三種chip跑I\_mem\_compression\_2.txt所需的cycle數量。

*Fig. 3-5*：Fig. 3-1的area乘上Fig. 3-4的cycle number所得的結果。

* + - 1. 圖表結果：

|  |  |
| --- | --- |
|  |  |
| Fig. 3-1 | Fig. 3-2 |
|  |  |
| Fig. 3-3 | Fig. 3-4 |
|  |  |
| Fig. 3-5 |  |

1. Comparison
2. 三種strategy比較

在有buffer strategy弱點的情況下，如同前面的設計理念所述，需要的cycle數量確實是halfword cache < buffer < bubble。而同時他們的面積差異並不大，因此halfword cache有較好的A\*T值(Fig.3-4、3-5)。

不過，在沒有buffer strategy弱點的情況下，使用buffer會有最佳的A\*T值(Fig. 3-3)。因此，在實際運用上，仍要視真正的instruction memory情況來決定要採取何種策略。但整體來說，halfword cache strategy的效能最為全面。

1. 有compression及無compression比較

相對於沒有compression的指令集，有compression的好處在於可以減少memory size、減少存取數量進而減少power、還能提高效率(throughput)。

以我們的測試結果來說，可以發現在cycle number上三種策略的compression chip都遠小於decompression chip (Fig. 3-2)。可見確實能夠提升效率。而cycle變少的情況也意味著向memory存取的次數減少，因此可以降低power。

而觀察I\_mem\_compression及I\_mem\_decompression，也可以發現前者所需的memory size大約只有後者的2/3，確實也能夠大幅降低memory size。

**附錄、**

1. 測試說明
2. 在Compression的halfword cache strategy中，我改變了傳遞的PC位數，因此必須修改testbench中傳給testbed的address(從DCACHE\_addr[29:0]改成{1'b0, DCACHE\_addr[29:0]} )，才能達成testbed的終止條件(curaddr==`CheckNum)。因此，在測試halfword strategy時，需使用Final\_tb\_compression\_halfword.v這份檔案。
3. 在Compression中，若要測試含有buffer strategy的弱點的testbench，要改用I\_mem\_compression\_2.txt及Testbed\_compression\_2.v。
4. 參考資料
5. 心得