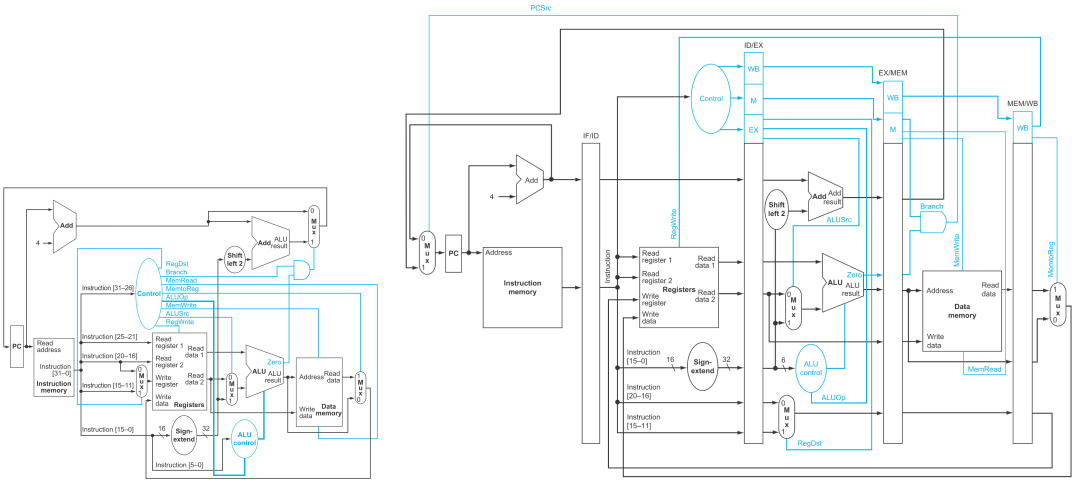
CA Final Project Report

組員：電機四 彭俊又 B01208026
組員：電機三 徐彥旻 B03901027
組員：資工二 齊煒禎 T05902124

Baseline Design

目標：將作業三的 Single Cycle MIPS 改成 Pipeline MIPS。



前置作業：

將本次 Pipeline MIPS 需要用到的指令先做成一張指令與機器碼的對照表格

Name	Type	OP [31:26]	rs [25:21]	rt [20:16]	rd [15:11]	[10:6]	function [5:0]	FunName	descryption
ADD	R-type	000000	rs	rt	rd	00000	100000	ADD	`rd <- rs + rt`
ADDI	I-type	001000	rs	rt	immediate			ADD	`rt <- rs + immediate`
SUB	R-type	000000	rs	rt	rd	00000	100010	SUB	`rd <- rs - rt`
AND	R-type	000000	rs	rt	rd	00000	100100	AND	`rd <- rs AND rt`
ANDI	I-type	001100	rs	rt	immediate			AND	`rt <- rs AND immediate`
OR	R-type	000000	rs	rt	rd	00000	100101	OR	`rd <- rs OR rt`
ORI	I-type	001101	rs	rt	immediate			OR	`rt <- rs OR immediate`
XOR	R-type	000000	rs	rt	rd	00000	100110	XOR	`rd <- rs XOR rt`
XORI	I-type	001110	rs	rt	immediate			XOR	`rt <- rs XOR immediate`
NOR	R-type	000000	rs	rt	rd	00000	100111	NOR	`rd <- rs NOR rt`
SLL	R-type	000000	00000	rt	rd	sa	000000	SLL	`rd <- rt << sa`
SRA	R-type	000000	00000	rt	rd	sa	000011	SRA	`rd <- rt >> sa (arithmetic)`
SRL	R-type	000000	00000	rt	rd	sa	000010	SRL	`rd <- rt >> sa (logical)`
SLT	R-type	000000	rs	rt	rd	00000	101010	SLT	`rd <- (rs < rt)`
SLTI	I-type	001010	rs	rt	immediate			SLT	`rt <- (rs < immediate)`
BEQ	I-type	000100	rs	rt	offset			NO ALU	`if(rs==rt) PC=PC+offset`
J	J-type	000010	target					NO ALU	`not PC-relative`
JAL	J-type	000011	target					NO ALU	`rs <- PC`
JR	R-type	000000	rs	000000000000000			001000	NO ALU	`PC <- rs`
JALR	R-type	000000	rs	00000	rd	00000	001001	NO ALU	`rd <- return_addr, PC <- rs`
LW	I-type	100011	base	rt	offset			ADD	`rt <- memory[base+offset]`
SW	I-type	101011	base	rt	offset			ADD	`memory[base+offset] <- rt`

其次，由於將指令轉換成不同控制訊號的控制單元(Control Unit)，其內容基本上可以視為一個解碼器。為了實作方便，我們也將不同輸入訊號對應的輸出控制訊號製作成一張表格。表格中不同背景顏色，代表著該控制訊號送到不同的 Pipeline stage 執行，越晚執行的訊號其顏色越深。舉例而言，Jump、Jr 在最早的 Instruction Decode 的階段執行，顏色最深的 MemtoReg、RegWrite、Jal 在最後一個階段(WriteBack)才執行。

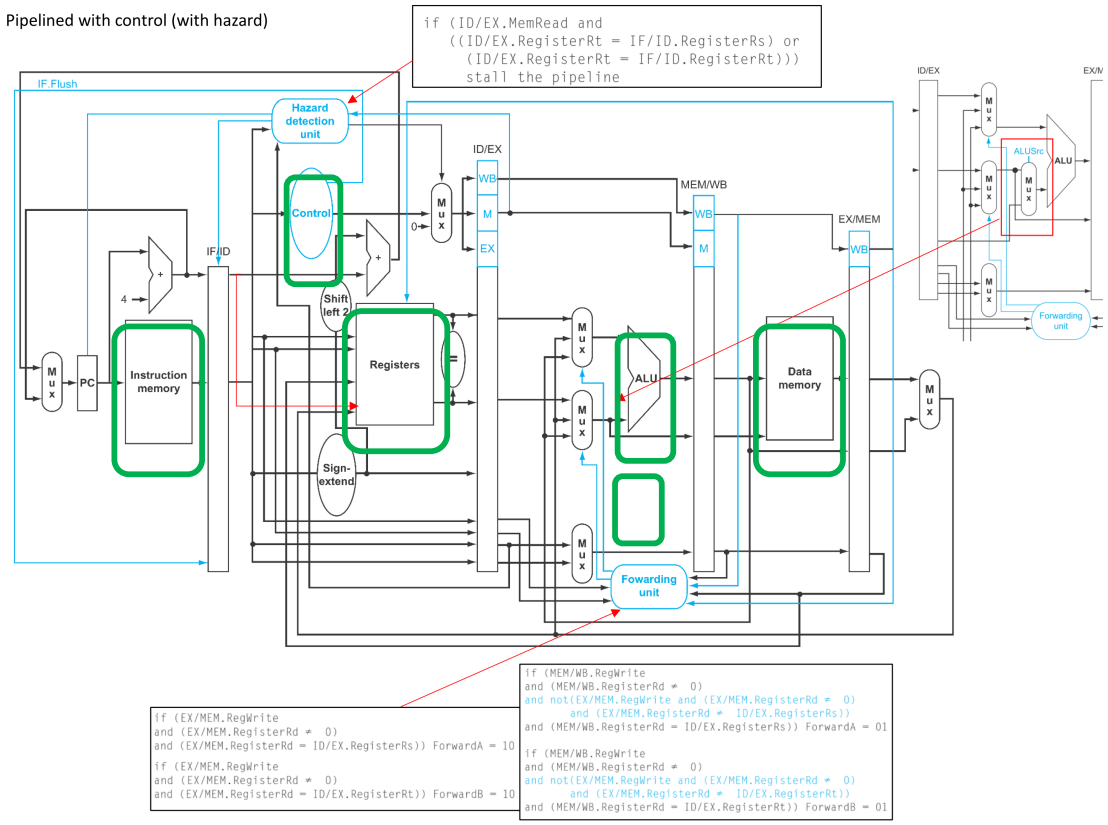
	Type	OP [31:26]	function [5:0]	Jump	Jr	RegDst	ALUsrc	MemRead	MemWrite	Branch	MemtoReg	RegWrite	Jal
ADD	R-type	000000		0	0	1	0	0	0	0	0	1	0
ADDI	I-type	001000		0	0	0	1	0	0	0	0	1	0
SUB	R-type	000000		0	0	1	0	0	0	0	0	1	0
AND	R-type	000000		0	0	1	0	0	0	0	0	1	0
ANDI	I-type	001100		0	0	0	1	0	0	0	0	1	0
OR	R-type	000000		0	0	1	0	0	0	0	0	1	0
ORI	I-type	001101		0	0	0	1	0	0	0	0	1	0
XOR	R-type	000000		0	0	1	0	0	0	0	0	1	0
XORI	I-type	001110		0	0	0	1	0	0	0	0	1	0
NOR	R-type	000000		0	0	1	0	0	0	0	0	1	0
SLL	R-type	000000		0	0	1	0	0	0	0	0	1	0
SRA	R-type	000000		0	0	1	0	0	0	0	0	1	0
SRL	R-type	000000		0	0	1	0	0	0	0	0	1	0
SLT	R-type	000000		0	0	1	0	0	0	0	0	1	0
SLTI	I-type	001010		0	0	0	1	0	0	0	0	1	0
BEQ	I-type	000100		0	0	0	0	0	0	1	0	0	0
J	J-type	000010		1	0	0	0	0	0	0	0	0	0
JAL	J-type	000011		1	0	0	0	0	0	0	0	1	1
JR	R-type	000000	001000	1	1	0	0	0	0	0	0	0	0
JALR	R-type	000000	001001	1	1	1	0	0	0	0	0	1	1
LW	I-type	100011		0	0	0	1	1	0	0	1	1	0
SW	I-type	101011		0	0	0	1	0	1	0	0	0	0
NOP				0	0	0	0	0	0	0	0	0	0

ALU 的控制單元(ALU Control Unit)同樣的也是一個解碼器的結構，因此也利用表格簡化輸入訊號與輸出訊號的對應關係。

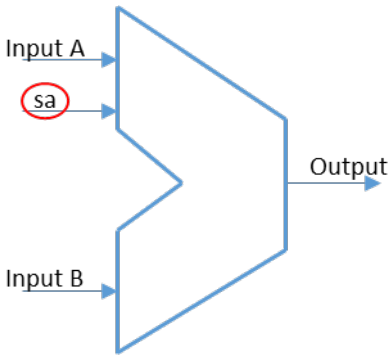
ALU Function	ALUControlInput [3:0]	OP [31:26]	FunctionCode [5:0]
ADD	0000	001000 100011 101011	100000
SUB BEQ (SUB)	0001	000100	100010
AND	0010	001100	100100
OR	0011	001101	100101
XOR	0101	001110	100110
NOR	0110		100111
SLL	0111		000000
SRL	1000		000011
SRA	1001		000010
SLT	1100	001010	101010

基本架構：

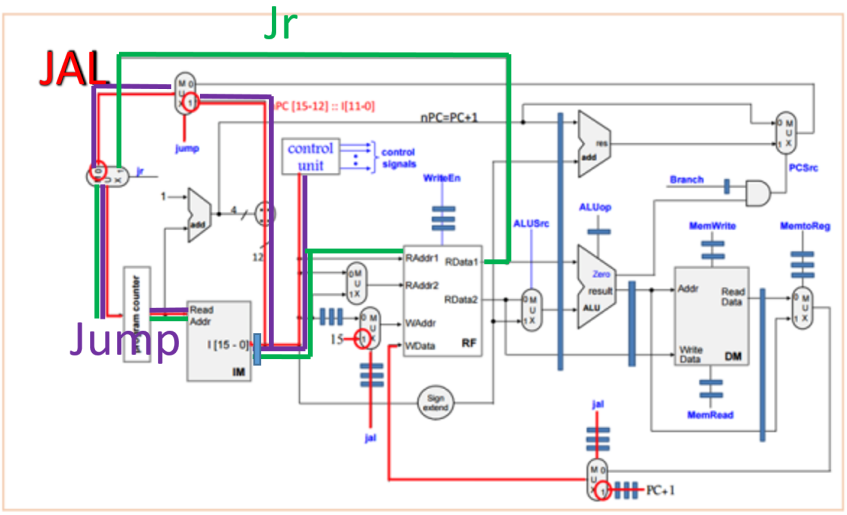
我們以課本中繪製的 Pipeline MIPS 作為我們最基本的結構，再從中修改加入本次專案所需的新的指令。



上圖中，以綠色線框起來的模組，是之前的作業中已實作過的部分，稍加修改即可直接拿來使用。包括作業三的 Registers，和作業五的 Cache 直接放在 Instruction Memory 和 Data Memory 的位置。Control Unit 和 ALU Control Unit 在作業三也實作過，次處利用一開始製作的表格，將不同輸入訊號相對應的輸出訊號時做出來。ALU 的部分，大致上與作業一的相同，有進行更改的部分，是在遇到 SLL, SRA, SRL 三個指令時，為了簡化整體結構，我們將 Instruction[10:6] (sa)的訊號直接輸入 ALU。Forwarding Unit 的部分，用課本中給的判斷式，很容易的就可以將它做出來。



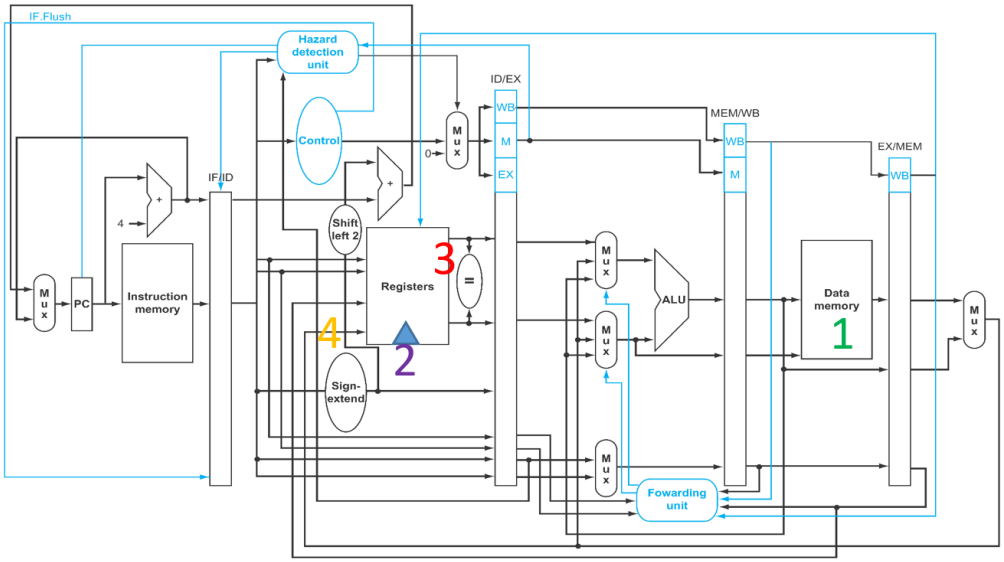
J, Jr, Jal 指令，從圖中可以看到三種指令的訊號路徑。



比較需要注意的是 Jal 時要儲存的(PC+4)需要經過 3 層的 Flip-Flop 之後再存進 registers。

Hazard Detection 的部分是實作上最麻煩的地方。經過多次的測試，最後整理出 4 種情況：

1. Memory Read: 如課本中提到的，當資料從 Data Cache 讀出後，需要 stall 一個 clock，才能透過 Forwarding Unit 送回前一級執行。
2. Register Write: 當寫入與讀取的資料為同一個 register 時，由於 Register 是在 Positive Edge 才將資料寫入，因此需要 stall 一個 clk 等資料寫入後才做讀取的動作。
3. Branch: 由於我們將 Branch Equal 的比較放在 Instruction Decode 的階段，若我們要進行比較的資料還沒被存回 Register 時，會有 Hazard 產生。
4. Jr & Jal: 由於 Jal 的資料要經過 3 個 stage 才會被存入 Register，因此若在那之前遇到 Jr 我們也需要透過 Stall 來避免 Hazard 產生。



實作結果：

Simulation:

模擬的結果，用 5ns Clock Cycle，總共需要 3181 Cycles

```
ncsim> run
-----------------------------------------------------

START!!! Simulation Start .....

-----------------------------------------------------

Duration:  3181
--------------------------- Simulation FINISH !!---------------------------
================================================================================

 \(^o^)/ CONGRATULATIONS!!  The simulation result is PASS!!!

================================================================================
Simulation complete via $finish(1) at time 16067500 PS + 0
./Final_tb.v:162              #(`CYCLE) $finish;
ncsim> exit
```

Synthesis:

Area Report:

Total Cell Area: 302339 (um^2) ; (D $, I $, MIPS) = (81977, 77288, 139646)

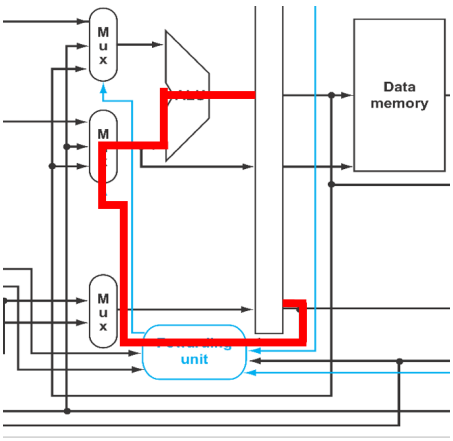
```
Combinational area:       173131.402813
Noncombinational area:    129207.781185
Net Interconnect area:    2279165.022034

Total cell area:          302339.183998
Total area:               2581504.206032
```

Cell	Reference	Library	Area	Attributes
D_cache	cache_0		81977.628368	h, n
I_cache	cache_1		77228.303009	h, n
i_MIPS	MIPS_Pipeline		139646.793025	h, n
Total 176 cells			302339.183998	

Timing Report:

```
clock CLK (rise edge)                                   2.60       2.60
clock network delay (ideal)                             0.50       3.10
clock uncertainty                                      -0.10       3.00
i_MIPS/ExMem_reg[39]/CK (DFFRX4)                        0.00       3.00 r
library setup time                                     -0.09       2.91
data required time                                                 2.91
--------------------------------------------------------------------------
data required time                                                 2.91
data arrival time                                                 -2.91
--------------------------------------------------------------------------
slack (MET)                                                        0.00
```

Critical Path



Performance:

```
ncsim> run
-----------------------------------------------------

START!!! Simulation Start .....

-----------------------------------------------------

duration:  3335
-------------------------- Simulation FINISH !!---------------------------
==============================================================================

 \(^o^)/ CONGRATULATIONS!!  The simulation result is PASS!!!

==============================================================================
Simulation complete via $finish(1) at time 8426250 PS + 0
./Final_tb.v:162              #(`CYCLE) $finish;
ncsim> exit
```

Simulation Clock Cycle: 2.50

```
6 `define CYCLE 2.50 // You can modify your clock frequency
```

AT value:

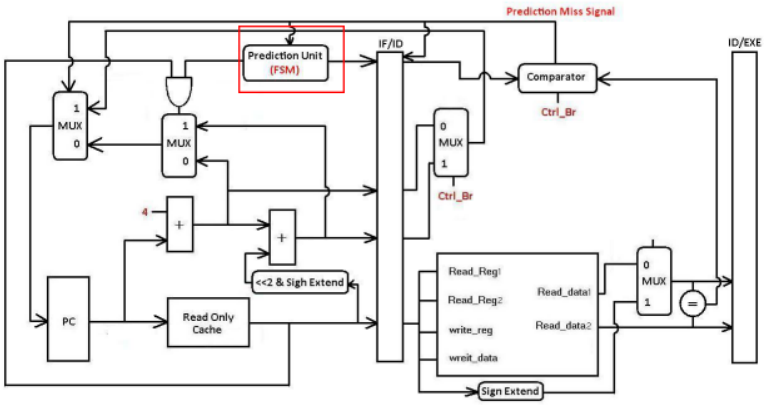
Area(um^2) * Total simulation time (ns) = 302339 * 8426.25 = $2.55*10^9$

Branch Prediction (2-bit Predictor)

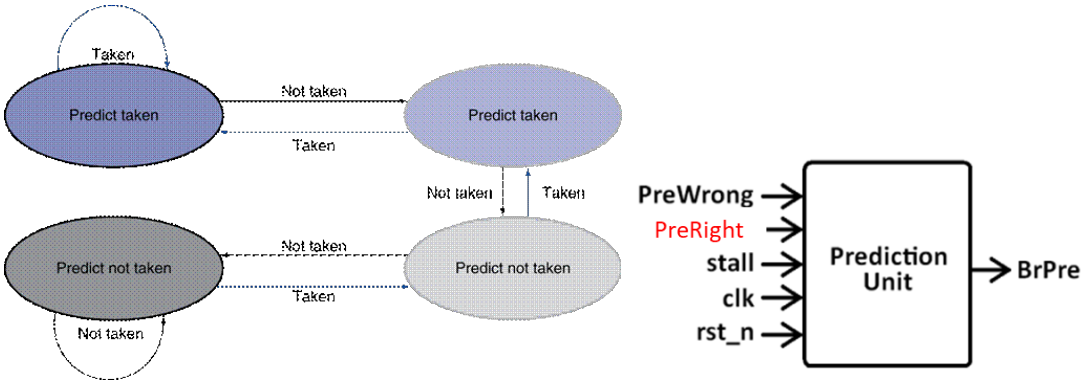
上課時有提到 1-bit, 2-bit 以及其他的預測方法，由於 2-bit Predictor 的效果較 1-bit 的好，加上複雜性遠低於其他的預測方法，在考量到不要 Over Design 的情況下，我們選用 2-bit 的預測單元。

設計架構：

以助教提供的電路圖實作。

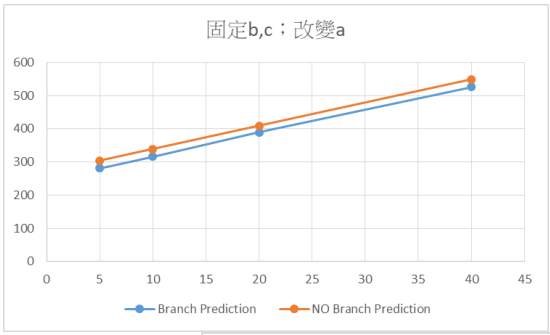


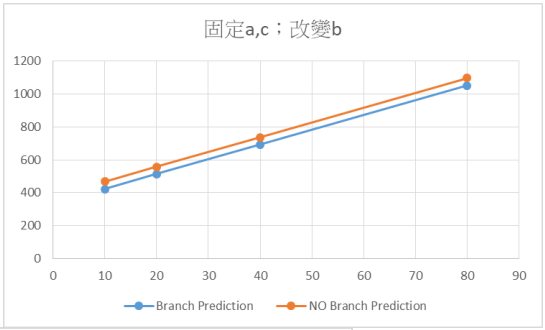
左下圖為 Prediction Unit FSM 的狀態圖。在 Prediction Unit 中，由於狀態的轉換是由預測的正確與否而決定，因此使用 Prediction Right 和 Prediction Wrong 這兩種訊號來做為 Prediction Unit 的輸入訊號，改變預測狀態。

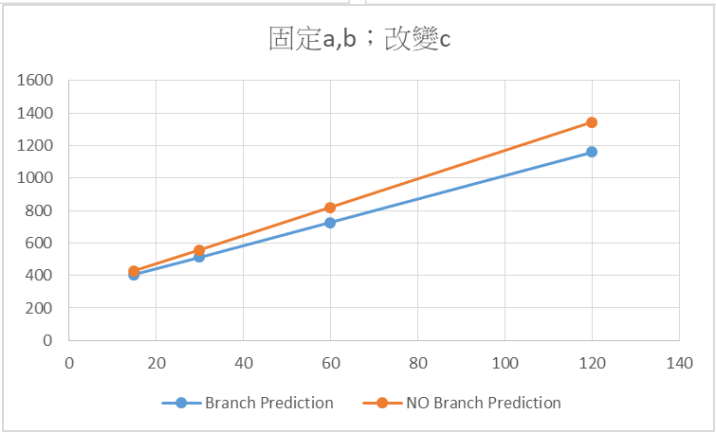


實作結果：

Brpred Testbed:







當我們改變 always NOT Branch 的情況時，Prediction Unit 會正確的預測結果，但是沒有 Prediction Unit 的 MIPS 預設情況也同樣是 Not Branch 因此兩者在結果上並無法看出差別。當 Branch 與 NOT Branch 交錯出現時，無論有沒有 Prediction Unit 都會出現一次正確一次錯誤的判斷。在 Always Branch 的情況時，我們可以很明顯的看出有 Prediction Unit 的 MIPS 會讓執行的 cycle 數下降。從實驗中我們可以看到加入 Prediction Unit 可以改善連續 Branch 的情形。

HasHazard Testbed:

	Branch Prediction	NO Branch Prediction
HasHazard	3180	3181

而當我們用有 prediction unit 的 MIPS 去執行 HasHazard 的測試時，我們會發現兩者的總 cycle 數差距幾乎沒有差別，當我們仔細的去分析時，發現兩者在 Prediction Wrong 的次數是相同的，由於 HasHazard 中執行迴圈的關係，在多數情況下 BEQ 的結果都是 Not Branch，因此 Prediction Unit 並無法發揮其效果。

Synthesis:

```
clock CLK (rise edge)                                  3.00       3.00
clock network delay (ideal)                            0.50       3.50
clock uncertainty                                     -0.10       3.40
i_MIPS/ExMem_reg[60]/CK (DFFRX4)                       0.00       3.40 r
library setup time                                    -0.09       3.31
data required time                                                3.31
--------------------------------------------------------------------------
data required time                                                3.31
data arrival time                                                -3.31
--------------------------------------------------------------------------
slack (MET)                                                       0.00
```

```
Combinational area:      160794.699664
Noncombinational area:   136783.277077
Net Interconnect area:   2243044.852203

Total cell area:         297577.976740
Total area:              2540622.828944
```

```
clock CLK (rise edge)                                  3.00       3.00
clock network delay (ideal)                            0.50       3.50
clock uncertainty                                     -0.10       3.40
D_cache/CacheMem_r_reg[7][44]/CK (DFFRX1)              0.00       3.40 r
library setup time                                    -0.14       3.26
data required time                                                3.26
--------------------------------------------------------------------------
data required time                                                3.26
data arrival time                                                -3.26
--------------------------------------------------------------------------
slack (MET)                                                       0.00
```

```
Combinational area:      168638.385072
Noncombinational area:   140424.200279
Net Interconnect area:   2285185.076447

Total cell area:         309062.585352
Total area:              2594247.661798
```

當我們用同樣 clock cycle 的條件合成時，加入 Branch Prediction Unit 的 MIPS 總面積為 309062 (um^2)。相比較之下，可以得出 Prediction Unit 的面積大約是 11458 (um^2)。

Assembly

目標：根據自己組寫好的 mips cpu 的特徵，修改和調整 assembly 代碼，或者調整 mips cpu 的元件，達到加速的效果。

1.翻譯成 mips 支持的 machine code

2.執行和測試

3.修改 assembly 加速

4.總結

1. 翻譯成 mips 支持的 machine code

(a)Assembly 中的硬替換：

比如：move $t1,$t2　改為 add $t1,$t2,$zero

Bge $t1,(0x0004)$t2　改為 slt $at,$t1,$t2　beq $at, (0x0004)$zero 等等。

然後注意 baseline 的 mips 中不支援 bne，addu，等等指令

(b)Assembly 中語句的名稱改為位置

因為步驟(a)中會改變語句之間的相對位置，所以(b)要在(a)之後。

把各個語句的名稱如 loop: 的對應 address 記錄，並在 j loop 的同類語句中替換為地址。

因為 mips 裡沒有初始的 stack，$sp 也沒有值，因此若 jal，需要初始化 $sp。

(c)翻譯成 machine code

利用助教給的網站，依次翻譯各個 instruction。

2.執行和修改：

需要做和修改的檔：D_mem、I_mem_Assembly、Testbend_Assembly

(a) Data

翻譯結束後只有 instruction 而沒有 data，所以需要生成 data。生成 data 的方式有兩種，一種是在 instruction 中生成 data，一種是 D_mem 中放好 data。為了更好地作類型間的比較，用 C++生成 best case（由小到大）、worst case（由大到小）、average case（隨機）的 D_mem

(b)I_mem_Assembly

在上一步中翻譯好。

(c)Testbend

Testbend 需要有兩個功能：統計運算所需要的 clk 數目和檢查結果的正確性。實現的方式為，約定好一個 address，testbend 一直監視這個位址的資料，Assembly 代碼中把控制信號和排好的資料搬過去檢驗。Testbend 在 reset 信號到來時開始統計 clk，Assembly 排序好後把開始搬運信號寫在特定 address，testbend 結束 clk 數目統計，進入 check 狀態，檢查 assembly 依次搬過來的各個資料是不是由小到大排列正確的資料。

3.修改 Assembly：

(a)減少 stall

Mips 中會帶來 hazard 的地方中 IdExMemRead && ((IdExRegRt == IfIdRegRs)和 WbRegWrite && ((IfIdRegRd == IfIdRegRs)帶來的 stall 是可以可以通過調整 assembly 的語句的位置來取消掉的。所以要調整以下語句：lw 得到的資料在下一個 instruction 中被用到；上一個 instruction 的要寫回 register 要在下一個 instruction 中用到。另外 stall 產生無法在 assembly 的部分解決。因此檢查 bubblesort 和 quicksort 的代碼中的以上兩種情況，通過移動其他寄存器的運算指令來減少 stall。

(b)減少 branch 錯誤的數目

在寫好的 MIPS 中，是默認 branch 不 take 的，因此在這個前提下，可以調整 assembly 的結構，使得最多的 branch 的位置在最多的情況下不 take，只在邊際的最後條件下 take。

```
bbst:   add $t1,$zero,$zero
biglp:  add $t0,$s1,$zero
        addi $t1,$t1,1
        bge $t1,$s0,endst
        add $t2,$zero,$zero
        sub $t5,$s0,$t1
smlp:   bge $t2,$t5,biglp
        lw $t3,0($t0)
        lw $t4,4($t0)
        bge $t3,$t4,swap
        addi $t0,$t0,4
        addi $t2,$t2,1
        j smlp
swap:   sw $t3,4($t0)
        sw $t4,0($t0)
        addi $t0,$t0,4
        addi $t2,$t2,1
        j smlp
endst:  addi $t2,$zero,0x0932
        add $t0,$s1,$zero
        add $t1,$zero,$zero
        sw $t2,0x0100($zero)
check:  beq $t1,$s0,over
        lw $t2,0($t0)
        addi $t0,$t0,4
        sw $t2,0x0100($zero)
        j check
over:   addi $t2,$zero,0x0D5D
        sw $t2,0x0100($zero)
trap:   j trap
```

```
bbst:   addi $t1,$zero,0
biglp:  add $t0,$s1,$zero
        addi $t1,$t1,1
        bge $t1,$s0,endst
        addi $t2,$zero,0
        sub $t5,$s0,$t1
smlp:   bge $t2,$t5,biglp
        lw $t3,0($t0)
        lw $t4,4($t0)
        addi $t0,$t0,4
        addi $t2,$t2,1
        bge $t3,$t4,swap
        j smlp
swap:   sw $t3,0($t0)
        sw $t4,-4($t0)
        j smlp
endst:  add $t0,$s1,$zero
        add $t1,$zero,$zero
        addi $t2,$zero,0x0932
        sw $t2,0x0100($zero)
check:  beq $t1,$s0,over
        lw $t2,0($t0)
        sw $t2,0x0100($zero)
        addi $t0,$t0,4
        j check
over:   addi $t2,$zero,0x0D5D
        sw $t2,0x0100($zero)
trap:   j trap
```

critical region dehazard brings us 10 percent speed up

```
qsort:  bge $a1,$a2,endqsort
        sw $a1,0($sp)
        addi $sp,$sp,-4
        sw $a2,0($sp)
        addi $sp,$sp,-4
        sw $ra,0($sp)
        addi $sp,$sp,-4
        jal partition
        sw $v0,0($sp)
        addi $sp,$sp,-4
        add $t1,$v0,$zero
        lw $a1,16($sp)
        addi $a2,$t1,-4
        jal qsort
        lw $a1,4($sp)
        addi $a1,$a1,4
        lw $a2,12($sp)
        jal qsort
        lw $ra,8($sp)
        addi $sp,$sp,16
endqsort:
        jr $ra
```

```
qsort:  bge $a1,$a2,endqsort
        sw $a1,0($sp)
        sw $a2,-4($sp)
        sw $ra,-8($sp)
        addi $sp,$sp,-12
        jal partition
        sw $v0,0($sp)
        addi $sp,$sp,-4
        add $t1,$v0,$zero
        lw $a1,16($sp)
        addi $a2,$t1,-4
        jal qsort
        lw $a1,4($sp)
        lw $a2,12($sp)
        addi $a1,$a1,4
        #lw $a2,12($sp)
        jal qsort
        lw $ra,8($sp)
        addi $sp,$sp,16
endqsort:
        jr $ra
```

improvement in not critical region gives us less benefit

Bubble sort

4.結果分析：

500 個數據	Best case	Average case	Worst case
Bubble sort	2721316	3141275	3314274
修改後	2347046	2839610	3064771
提升	13.8%	9.6%	7.5%

50 個數據	Best case	Average case	Worst case
Bubble sort	26058	28834	30433
修改後	22381	25862	27980
Quick sort	33353	9123	29652
修改後	32654	8630	29027

Clock Cycle 數目表

1. 關鍵位置的 stall 數目減少起到了非常大的速度提升效果。由於在原本的 bubblesort 中，每次判斷相鄰位置兩個資料的大小並決定是否要調換順序，都需要 lw 兩個資料然後 sll 兩個資料決定是否跳轉到 swap 函數，非常頻繁使用，處理掉了這個 stall 為 bubble sort 帶來了 10%左右的提速效果。Quicksort 中，有更多的 data hazard 和低效率語句，但是由於位置不關鍵，處理後效果提升不明顯。
2. memory 的大小一共 1024，stack 從大開始往小擴展，data 從小往大擴展。quicksort 中由於模組化更好，分了很多函數，在遞迴呼叫中耗費大量的 stack 位置用以保存資料，因此無法排序 500 個資料，memory 很快在遞迴中被耗盡所有位置。
3. quicksort 在遞增資料和遞減資料的情況下排序效果都非常差，因為 quicksort 通過以最後一個資料作為 pivot，把比 pivot 大或小的資料分別丟到 pivot 的兩側，並繼續調用 quicksort 去排序兩側的資料。因此，如果比 pivot 大和小的資料個數越接近，quicksort 就能越快地減小 sort 的規模，best case 和 worst case 都每次把規模 n 減少成了 n-1，而平均數據是以 n/2 的速度減小規模的。
4. quicksort 在極端資料的情況下提速方法思考：通過查閱演算法書籍，可以發現為了避免極端資料引來的時間複雜度非常高的情況，每次選取 pivot 時，先隨機調換一個資料和最後一個資料的位置，再把最後一個資料作為 pivot 進行 quicksort。但是在這個 assembly 中，指令是串列執行的，產生亂數的過程又會帶來很大的額外計算開銷。因此，針對這個特定的情況，可以在 mips 中通過添加特定部件來實現加速：由於 mips cpu 中，每個硬

體部件都是並行運行的，可以額外寫一個 module 存放一個資料，用以在每個 clk 進行一定的邏輯運算，從而得到一個偽亂數，添加新的指令，從這個 module 調出亂數存在特定的 register，便可以實現隨機選取 pivot 的 assembly 實現，實現在該情況下的提速效果。

Level 2 Cache:

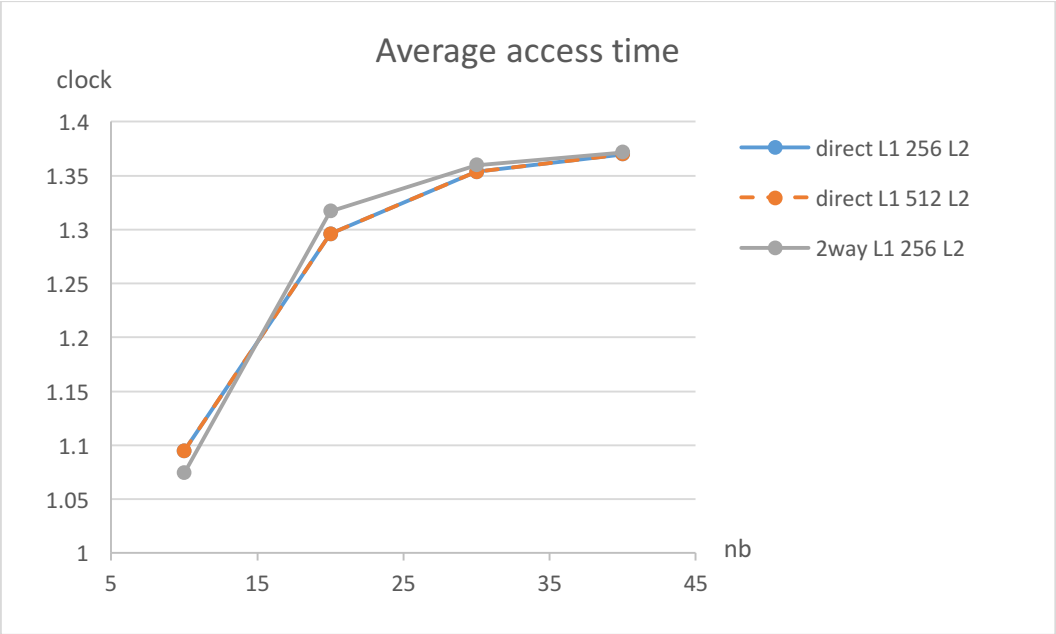
Implementation of L2 cache:

將 L1 cache 的大小改至 256 words ，並將 proc stall 改成 ready 的訊號，成為 L1 cache 與 memory 的中間層。另外也有實作 512 words 的版本。

Implementation of Two-Way L1 Cache:

對於每個 set ，增加 2-bit reference tag，若為 2’ b10 ，代表若需要換掉一個 block 時，將這個 set 的第一個 block 換掉；若為 2’ b01，則將這個 set 的第二個 block 換掉。

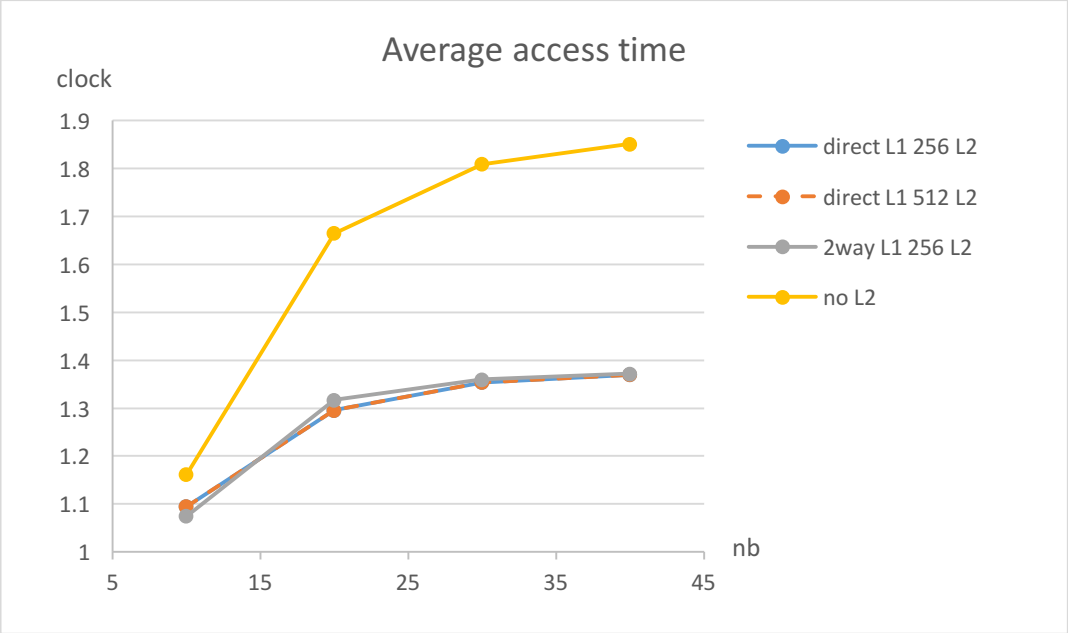
Analysis on Data Cache:



上圖是以助教提供的測資產生程式測驗的結果，對於不同的長度的 Fibonacci sequence 操作的表現，縱軸是平均每個 sw/lw 的指令所花去的 cycle 數，從圖中可以看得出來，數列的長度愈長，Average access time 就愈長。因為需要處理的資料變多，使得 L1 miss 的比例變高。

比較 256 words 跟 512 words 的 L2 cache (橘色虛線與藍色線)，可以發現表現一樣，是因為所處理的資料量都足夠全部放進 cache 當中，兩種大小的 cache 都只需要跟 memory 拿一次資料 ，若要看出差異，則必須要自行生成資料量更大的測試或是拿更小的 L2 cache 來比較。

比較 direct map L1 cache 與 two-way associative L1 cache，其表現也差不多，是因為這次的 Fibonacci sequence 與 bubble sort 皆為 sequential 的操作，故 way number 的改變對於效能的影響不大。由此推論，在這次的測試中，way number 的增加對於 instruction cache 的影響會比對 data cache 的影響要大。



上圖加上只有 L1 cache 的測試結果，呈現是否加上 L2 cache 對於效能的影響。加上了 L2 cache 後，在 nb =10, 20, 30, 40 時，Average access time 分別減少了 7.5%, 20.1%, 25%, 以及 26% 。下方的表格為 nb = 20 時，Score 1(Average access time)與 Score 2 (Total execution time) 在有無加入 L2 cache 時的比較。

	with L2 cache	without L2 cache
Average access time (clock)	1.29	1.66
Total execution time(ns) (clock set to 5 ns)	289047.5	264787.5