DSD Final report -- MIPS Group8

B06901071 何承叡
B06901058 陳聖佑

壹、Baseline design

1. Pipeline MIPS design

(a) ctrl unit:

在 ID stage 執行，接受 eq(判斷 beq 或 bne 的訊號)、op(instruction[31:26])、func(instruction[31:26])後，發出 9 個控制訊號，我們整理成以下的大表後再寫成 verilog:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1					ID				EX			MEM		WB	
2		form	op	func	PCDst(	WrDst	RegDst	ifjal	ALU	ALUop	ALUsrc(	MemWrite	MemRead	RegWrite	Mem2Reg
3	ADDI	I	001000		0	0	0	0	add	0000	0	0	0	1	0
4	ANDI	I	001100		0	0	0	0	and	0001	0	0	0	1	0
5	ORI	I	001101		0	0	0	0	or	0010	0	0	0	1	0
6	XORI	I	001110		0	0	0	0	xor	0011	0	0	0	1	0
7	SLTI	I	001010		0	0	0	0	<	0100	0	0	0	1	0
8	BEQ	I	000100		0/1	x(0)	x(0)	0		x(0000)	x(0)	0	0	0	x(0)
9	BNE	I	000101		0/1	x(0)	x(0)	0		x(0000)	x(0)	0	0	0	x(0)
10	LW	I	100011		0	0	0	0	add	0000	0	0	1	1	1
11	SW	I	101011		0	x(0)	x(0)	0	add	0000	0	1	0	0	x(0)
12	J	J	000010		2	x(0)	x(0)	0		x(0000)	x(0)	0	0	0	x(0)
13	JAL	J	000011		2	1	2	1	add	0000	x(0)	0	0	1	x(0)
14	ADD	R	0	100000	0	0	1	0	add	0000	1	0	0	1	0
15	SUB	R	0	100010	0	0	1	0	sub	0101	1	0	0	1	0
16	AND	R	0	100100	0	0	1	0	and	0001	1	0	0	1	0
17	OR	R	0	100101	0	0	1	0	or	0010	1	0	0	1	0
18	XOR	R	0	100110	0	0	1	0	xor	0011	1	0	0	1	0
19	NOR	R	0	100111	0	0	1	0	nor	0110	1	0	0	1	0
20	SLL	R	0	000000	0	0	1	0	sll	0111	1(shamt	0	0	1	0
21	SRA	R	0	000011	0	0	1	0	sra	1000	1(shamt	0	0	1	0
22	SRL	R	0	000010	0	0	1	0	srl	1001	1(shamt	0	0	1	0
23	SLT	R	0	101010	0	0	1	0	<	0100	1	0	0	1	0
24	JR	R	0	001000	3	x(0)	x(1)	0		x(0000)	x(1)	0	0	0	x(0)
25	JALR	R	0	001001	3	1	x(1)	0	add	0000	x(1)	0	0	1	x(0)
26	NOP	N	0	0	x(0)	x(0)	x(1)	0		x(0111)	x(1)	0	0	x(1) !!(因	x(0)

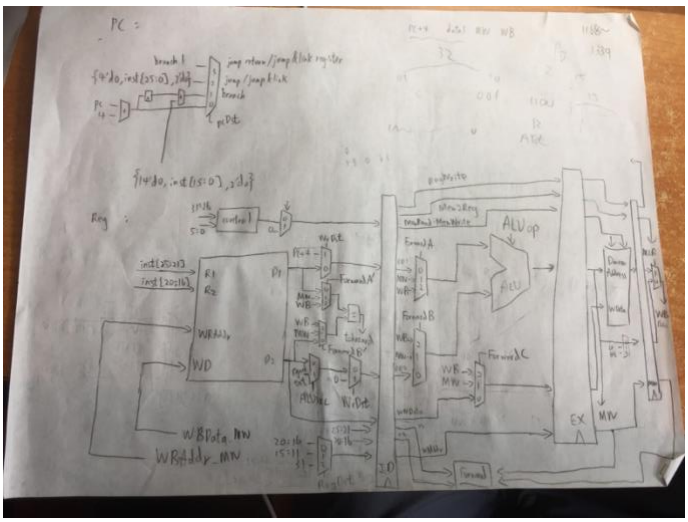
(其中 PCDst=0 表下個 PC 為 PC+4;1 表 branch 的 destination;2 表 j,jal 要跳到的地方;3 表 jr,jalr 要跳到的地方)

(b) stall、forwarding unit:

大致如課本所述，但 forwarding unit 因有在 ID stage 判斷 branch，我們除了原本給 EX stage 的訊號，需要另外拉一條訊號到 ID stage 去更新最新的值，以保證在 ID stage 判斷 branch 的值是正確的

(c) 架構搭建

基本的架構設計與教授講義提供的電路圖差異不多。其中，branch 的判斷機制因提前到 ID stage 判斷，所以當遇到所需資料在 EX stage 剛被 ALU 算出的話，我們會藉由 stall 一個 cycle 等到下個 cycle 再從 MEM stage forward 資料來減少 critical path。



2. Cache design

我們的 cache 為 direct-mapped，write policy 為 write back。設計的架構為每次資料進來後，經過是否 hit、是否有 read 或 write、是否 dirty、是否 mem_ready 一系列的判斷後，做出相對應的動作，我們試圖用以下幾種方法優化 cache:

(a) 寫 FSM，這樣就不用每次都要經過一連串判斷。但實作後發現合成完後跑 gate-level 所需的時間會增加，我們認為可能是因為原本 cache 要做的事情就不多，只有在 hit 時向 processor 給值或改值，miss 時向 memory 發出 read 或 write 而已，因此要再判斷 state 反而會要另外存 bit 且多花時間判斷。

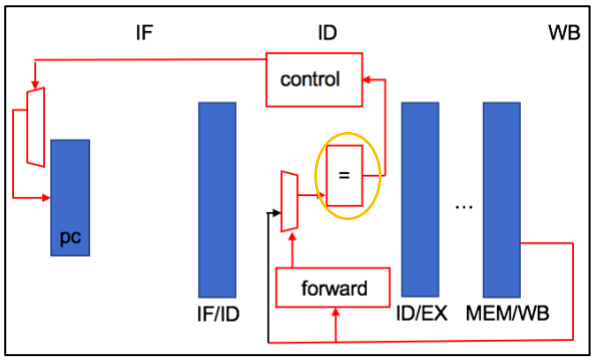
(b) 將 mem_ready 的訊號擋 FF，因為 mem_ready 是在負緣給值，如果不擋 FF 就只剩半個 cycle 可以做事，可能會增加所需的 cycle time。但真的擋了 FF 後，卻發現其實沒有影響 cycle time，反而因為增加了執行的 cycle 數而讓 total 執行時間增加，這可能也是因為比起 Pipeline MIPS 做的複雜的事情，其實 Pipeline MIPS 半個 cycle 的做事時間對 cache 來說就已經足夠了，擋 FF 反而讓她要在下個 cycle 才能做事，且浪費了半個 cycle 的時間。

(c) 在 write back 的 mem_ready 拉起來後，立刻將 mem_read 拉起來，因為我們的 cache 是用 write back 的方式，因此如果必是因為要 read memory 才會需要 write back 回 memory，表示 write back 回 memory 後必 read memory，如果可以立刻 mem_read 就不用再判斷再做，可以節省一個 cycle。但實作後發現反而會被 mem_ready 的訊號影響，因為立刻 mem_read 後，write back 的

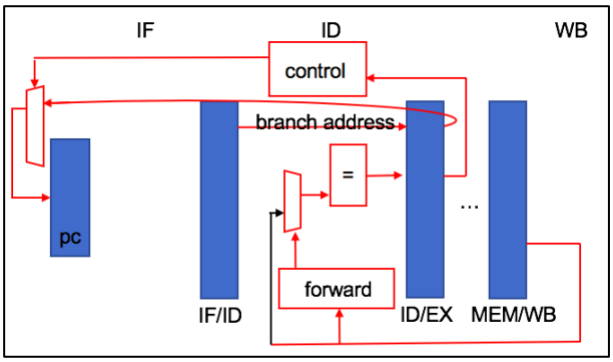
mem_ready 還在，會跟後面的 mem_read 完後的 mem_ready 搞混，但如果再寫判斷式讓他能區隔開來後，同樣會發現所需的 cycle time 又增加，最後仍沒有賺到。

我們試圖用上面三種方法優化，但發現還是原本的方式是最好的，最後透過調整判斷式的前後順序，例如 hit 需要經過很長的時間判斷才能得到結果，read、write 則是 clk 一拉起來就知道了，所以我們讓時間短的判斷先，時間長的判斷放在最後，可以讓 critical time 再往下降一些。

3. Optimization



圖一 優化前



圖二 優化後

如上圖一，可以看到原本電路設計的 critical path 會落在紅線段，此時的 cycle time 為 2.9ns。之後，我們透過 timing report 以及 CHIP_syn.v 去追訊號線，並判斷途中橘色圓圈處大約是 critical path 的一半。為了減少多出的 register 量，所以決定在判斷等式後的輸出切 pipeline，如圖二。如此一來，cycle time 減少至 2.5ns，增加的 register 為 32 bits 的 branch address 以及 1 bit 的等式結果。然而，此方法卻也會導致如果遇到需要採用 branch 跳 address 時，會從一個 stall cycle 變成兩個 stall cycle。

4. Baseline Result & Comparison

	Cycle time (ns)	Area (μm^2)	Execution Time (ns)	A*T value ($\mu m^2 \times ns$)
優化前	2.9ns	297,196.07	6,201.65	1,843,105,994.65
優化後	2.5ns	311,107.96	5,753.75	1,790,037,417.36

圖三 (hasHazard, Problem Size = 16)

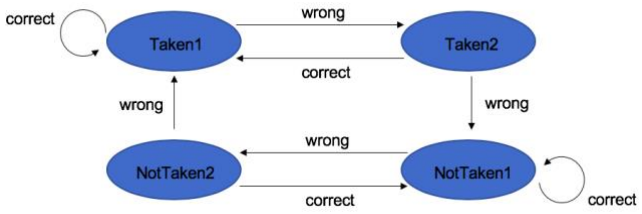
從圖三的比較中可以看出，優化後的 cycle time 減少，但也因如此以及 pipeline 多出的 register，造成 area 上的增加。除此之外，優化前的總執行 cycle 數是 2129 個，優化後的則增加至 2292 個，所以在 AT 上的減少並非十分顯著，減少了 2.8%的 AT。我們認為 AT 停留在 1.7×10^9量級左右，除了 stall cycle 數的問題之外，與 area 也有一定關係。三十萬左右的面積與週一各組的 area 結果相比，似乎有可以再

進步的地方。我們在 verilog 的書寫方式以及實作前的電路評估上仍有許多值得學習的空間。

貳、Extension

1. Branch prediction

(1) 1-level 2-bit branch prediction

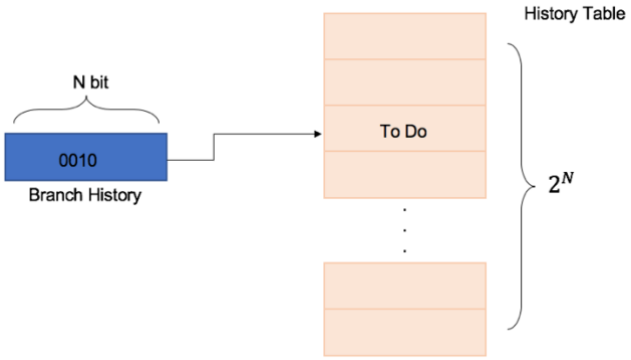


圖四

1-level 2-bit branch prediction 的設計採用講義上的方式設計，然而經過實驗結果會發現遇到 interleave branch 的時候，會在跳轉 state 上剛好差一個 cycle 而錯開，導致 10 的 pattern 被預測下個為 0；01 則被預測下個為 1。其中 1 代表採用 branch，0 代表不採用 branch。這使得不僅沒有減少 cycle 數，反而有增加的趨勢。

(2) 2-level 2-bit branch prediction with local history table

為了解決上述問題，我們至網路上查到 2-level branch prediction 的原理。如下圖五：



圖五

實作 local history table 時，會有一個 N bits 的 buffer(branch history)紀錄過去 N bits 的**實際 branch 結果**以及2^N bits 的 buffer(history table)記錄每個 pattern 下次應選擇的路徑。圖五即為一個 N=4 的範例，因為 branch history 記錄的是 0010，所以下次的選擇就是箭頭指到的位址(index=2)中所存的參數。

經過分析後，我們可以得出以下結論。如果 loop pattern 的長度有 P bits，則只要這 P bits 的 loop pattern 中任一連續 N bits 不重複，就可以用 2-level N-bit local-history-table branch predition 完整預測。其中，$N + 1 < P \leq 2^N$。

然而，此一設計遇到 0001 這種 pattern 就會失效，因為 00 會對應

到兩種可能。此外，history table 的硬體空間是隨著 N 做 exponential 增長，所以在 N 的抉擇上需要考量，而在這次的 final 中，我們選用 N=2 作為最後的實作方式。

(3) Branch Prediction Result & Comparison

1. RTL Cost Cycle Comparison

	hasHazard	I_mem_BrPred			
	Problem Size = 47	Never Branch	Interleave Branch	Always Branch	Total
No Branch Prediction	16466	50040	70016	40021	160077
1-bit	16491	50040	70007	**30023**	150070
2-bit	17458	50040	70019	**30023**	150082
History Table	**15352**	50040	**60021**	**30024**	**140085**

(nb_notBr, nb_interBr, nb_Br) = (10000, 10000, 10000)

圖六

我們將 3 種不同 branch prediction 以及沒有 branch predition 的電路結果做比較。如圖六，I_mem_BrPred 的三個參數都是調 10000。可以看出在 Never branch 上因為 no branch prediction 本身就是預測不會 branch，所以不會省到任何 cycle；always branch 則是三種 branch prediction 都可以成功預測，減少近 10000 個 cycle；interleave branch 則是只有 history table 能成功跳過。

值得注意的是，在 hasHazard 的測資中，1-level 2-bit branch prediciton 不減反增。推測是因為測資有許多 interleave branch 而造成 cycle 數的增加。

2. Gate Level Result

經過比較後，我們決定採用 history table 的版本進行合成。起初，有想再進一步採用 baseline 的方式優化；然而，這會導致 history table 以及 branch table 的更新落後一個 cycle。如果並沒有同時兩個 beq 或 bne 的指令一起出現，理當就可以採用此一方式進行優化。不過，因為不能保證編譯器編譯出的 instructions 能滿足此點，加上時間上的不足，所以就沒有做此一優化。

	Cycle time (ns)	Area of BPU (μm^2)	Execution Time (ns)	A*T value ($\mu m^2 \times ns$)
History Table	2.8ns	14,045.985096	5688.2	1,765,753,263.08

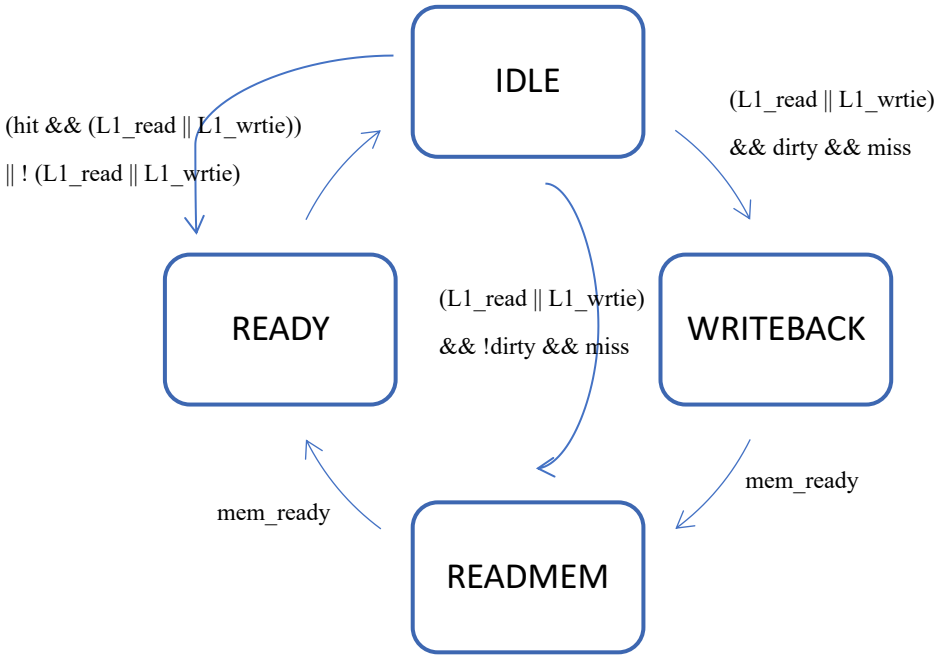
圖七 Gate Level Result (hasHazard, Problem Size = 16)

最後的結果如圖七，cycle time 為 2.8ns，area 增加 14000μm^2左右，約增加 4.74%。因為 total cycle 數的減少，所以連帶 AT 值也比 baseline 優化的版本少 1.35%。

2. L2 Cache

我們 L2 cache 為 separate、direct-mapped、write back 的方式，大小為 32blocks*4words，另外在 input 有擋 FF，且在 posedge 的 clk 才給 output。

FSM:



圖七

優化過程:

(a) posedge 的 clk 才給 output。一開始設計想說因為對 L1 來說，L2 的角色是原本的 memory，所以從 L1 的角度來看，L2 的行為應該跟 memory 一樣，所以設計 L2 會在 negedge 的 clk 給 L1 ready 的訊號，就像原本的 memory 一樣，但發現這樣合成後會只剩一半的 clk，所以後來改成統一在 posedge 的 clk 才給 output，讓合成的 clk 不會變一半。

(b) input 擋 FF。原本想說像 L1 的設計一樣不擋 FF，但發現還是要將 L1 與 memory 進來的 input 擋 FF，cycle time 才不會太長。

Result:

原本(只有 L1):AMAT = HT1+MR1*MP1

(a) HT1:1(cycle)

(b) MR1:有 WriteBack 與 ReadMem 兩種 miss，如下表

(c) MP1:8(cycles)

L1+L2:AMAT = HT1+MR1*HT2+MR1*MR2*MP2

(a) HT2:2(cycles)

(b) MR2:有 WriteBack 與 ReadMem 兩種 miss，如下表

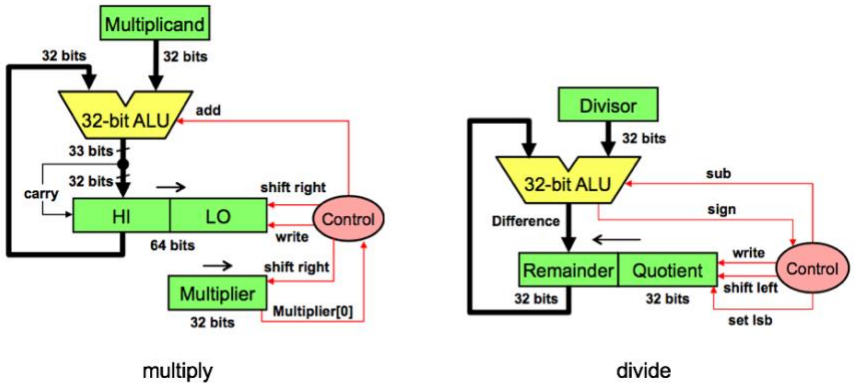
(c) MP2:8(cycles)

	I cache	D cache
MR1(WB)	0	4.9%(635/12855)
MR1(RM)	0.02%(10/47900)	5.1%(655/12855)
MR2(WB)	0	0
MR2(RM)	6/10(60%)	1.9%(22/1290)
AMAT only L1(cycles)	1.0016	1.803
AMAT L1+L2(cycles)	1.00136	1.314
improvement	0.02%	27%

可以發現 I cache 幾乎沒進步，因為本來 L1 就幾乎夠用了，只有 10 個漏掉需要向 L2 拿，可是可能因為是新的指令，L2 有 60%也需要向 memory 拿，所以幾乎沒賺到。而 D cache 進步了 27%，因為 L1 約有 10%的 miss，而這些 miss L2 幾乎都能抓到，只有 1.9%會漏掉需要再向 memory 拿，因此用了 L2 能進步非常多。

3. Multiplication and Division

(1) MultDiv Design



圖八

乘除器的設計我們使用 iterative approach 來實現。其中乘法記錄使用 HILO 的 register 記錄 partial sum，所以並沒有過多 overhead。除法則是額外使用兩個 32-bit register 記錄 remainder 以及 quotient，造成硬體上的 overhead 較大。這是一個可能可以再優化的部分，將 remainder、quotient 合併至 HILO 運算。

(2) MultDiv Result

	Cycle time (ns)	Area of BPU (μm^2)	Execution Time (ns)	A*T value ($\mu m^2 \times ns$)
MultDiv	2.8ns	22,838.516852	2,991.8	955,031,738.987

圖九 (MultDiv, Problem Size = 12)

MultDiv 最後的 cycle time 為 2.8ns，area 增加了 23000μm^2左右，約增加 7.71%。

參、Work Assignments Chart

何承叡	陳聖佑
MIPS 的 verilog 架構搭建 MIPS noHazard/hasHazard 過 baseline MIPS baseline 優化 Branch Prediction MultDiv	ctrl、forwarding、stall unit Cache L2Cache

肆、心得

我們覺得這次 Final Project 利用實作的方式，能讓我們比起只看課本，更加了解 Pipeline 的架構，且更加熟悉 Verilog 的語法，另外能理解原理後，真正實做出一個簡單功能的 CPU，且讓程式能在上面執行，也非常的有成就感。

經過這學期的鍛鍊，發現在設計電路上最重要的並非寫 verilog，而是如何設計且分析電路。做 final 時，因為有事先將電路架構畫到紙上，並在腦中模擬幾次才寫 verilog，使得後面要進行優化以及做 extension 時，都能知道應該在電路中如何增減修改，減少許多時間成本。此外，這堂課也將原本只會 RTL 的我們帶到會使用一些 design compiler。對實際的製作及電路合成多一份理解，而並非僅侷限在已抽象化過的 RTL 階層。

最後，十分感謝這學期教授與助教們的用心教學，學到十分多的 verilog coding 技巧以及硬體思維模式!