

DSD Final Report

Group 8

Group Members:

電機三 洪鈺萌 b04901111

電機三 陳柏帆 b04901002

電機三 林志皓 b04901069

Outlines:

MIPS pipeline basic structure

Cache Structure

Baseline performance

Extension 1 – Branch Prediction

Extension 2 – L2 Cache

Extension 3 – Multiplication & Division

一、MIPS pipeline basic structure

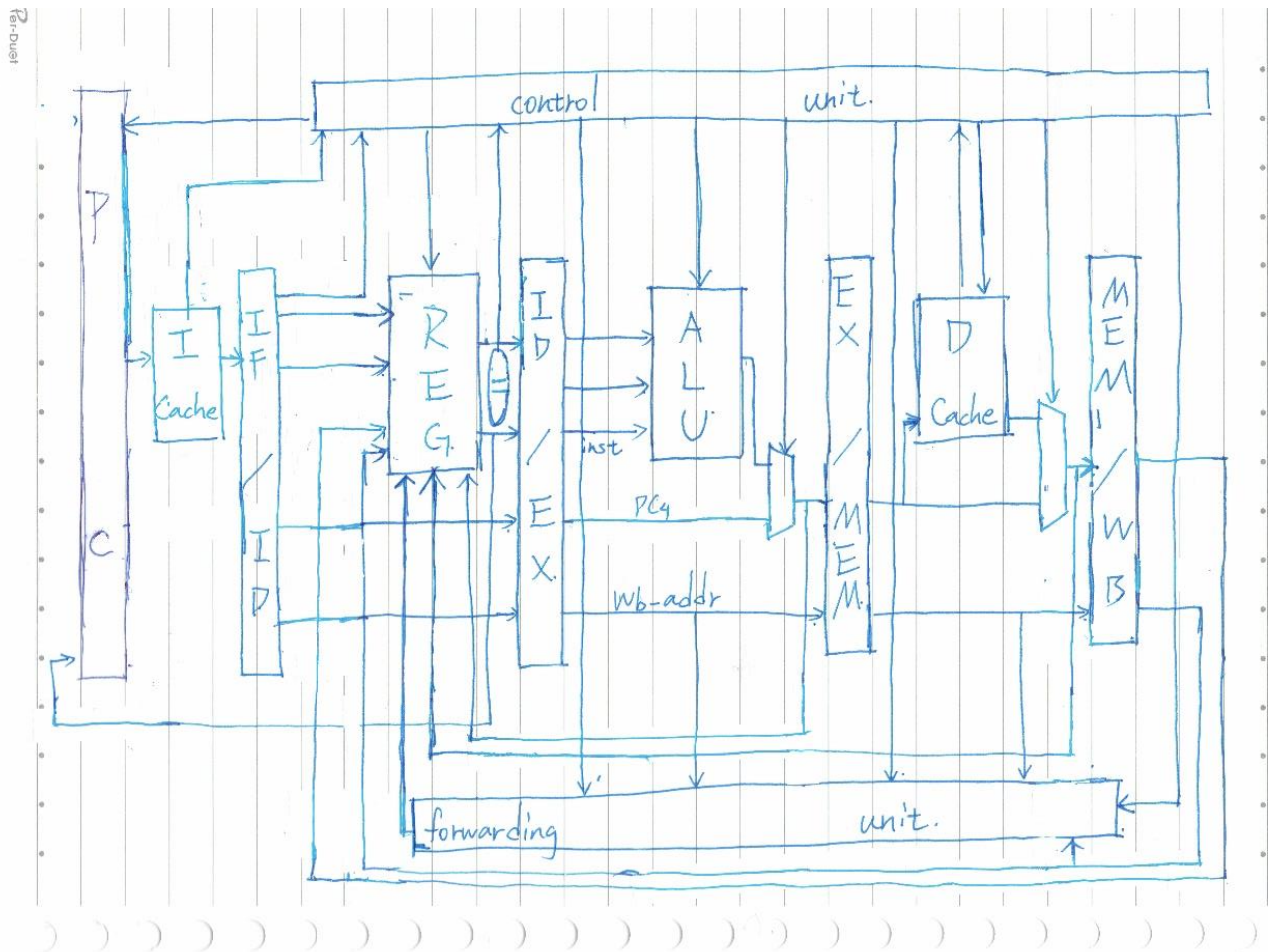


Fig 1. MIPS pipeline 架構圖

(1) **IF stage** : 這級主要是計算 PC 要從 instruction cache 拿指令所要輸出給 cache 的位址，而除了最基本的 PC+4 外，由於 MIPS 在設計上亦需要支援 JUMP、BEQ、JAL、JALR、JR 等等需要隨情況跳到不同位址的指令，因此需要一個 MUX 接收下一級 control unit 給的訊號來判斷下一個輸入 PC 的值為何。此外，比較不同的是，由於 J 與 JAL 只需要跳到一個指定的位址，而不用考慮 Register 裡存的資料是甚麼，因此我們直接將這 2 個指令拿到 IF stage 就進行判斷，並且把要跳到的位址也在這級就進行計算，且直接傳給 PC 作為輸入，這麼一來，就不需要額外多一個 flush 去處理下一個 PC 不是 PC+4 的情況，如此一來，能夠省下大幅時間。

(2) **ID stage** : 在這級基本上是執行 control unit 的判斷訊號，拿 register 裡儲存的值，計算 BEQ、JALR、JR 這些指令所要跳的位址，並且提早決定 BEQ 是否

成立等等。此外，比較特別的是，我們也把 Forwarding unit 往前移到這級作判斷，原因則是因為 BEQ 也相對提前了一級要在 ID 這級作判斷，因此需要確保拿來比較 BEQ 是否成立的資料是最新的。

(3)**EX stage**：這級除了先前提到的已經把 Forwarding unit 以及 BEQ 的判斷往前移到 ID stage 就先處理外，主要就只剩 ALU 的運算了，然而由於 MIPS 支援的指令中，包含了需要作 Sign extension 的 addi、sw、lw、slti 以及需要作 Zero extension 的 andi、ori、xori 這些 I-type 指令，因此必須區分不同情況作不同的處理，是比較麻煩的部分。

(4)**MEM stage**：這級就比較單純一點，就是處理 sw 與 lw 需要從 data cache 的某個位址拿資料或是將某資料存到 data cache 的某特定位址中這 2 個情況。此外，我們也提早在這級判斷要 forward 的資料究竟是從 data cache 拿的，或是 ALU 計算完的結果，如此一來，就能節省在 ID 那級，最新資料的選擇所用到的 MUX 大小。

(5)**WB stage**：基本上這級沒有比較特別的地方，就是執行是否要將某筆資料存到 Register 的某個位置，然而由於不同指令要存入的位置與值都不同，像是 JAL 要將 PC+4 的值存到 \$r31；而 JALR 則是將 PC+4 的值存入 \$rd，因此在處理值與位置這部分相對來說較為複雜。

二、Cache

Structure:

我們這組實行的是 Direct-mapped cache, writing strategy 是 write-back, 以下是架構：整個 cache 總共有 8 個 blocks, 而每個 block 儲存 4 words(32bits / word) 供 processor 取用，另外有 25 bit 作為 TAG 之用，還有 1 bit 作為 valid bit, 因此總共有 $32 \times 4 + 25 + 1 = 154$ bits/block

Valid bit	Tag	DATA 1	DATA 2	DATA 3	DATA 4
1 bit	25 bits	32 bits	32 bits	32 bits	32 bits

Fig 2. A block of cache

Finite State Machine:

總共只有 3 個 states，之間的關係如下：

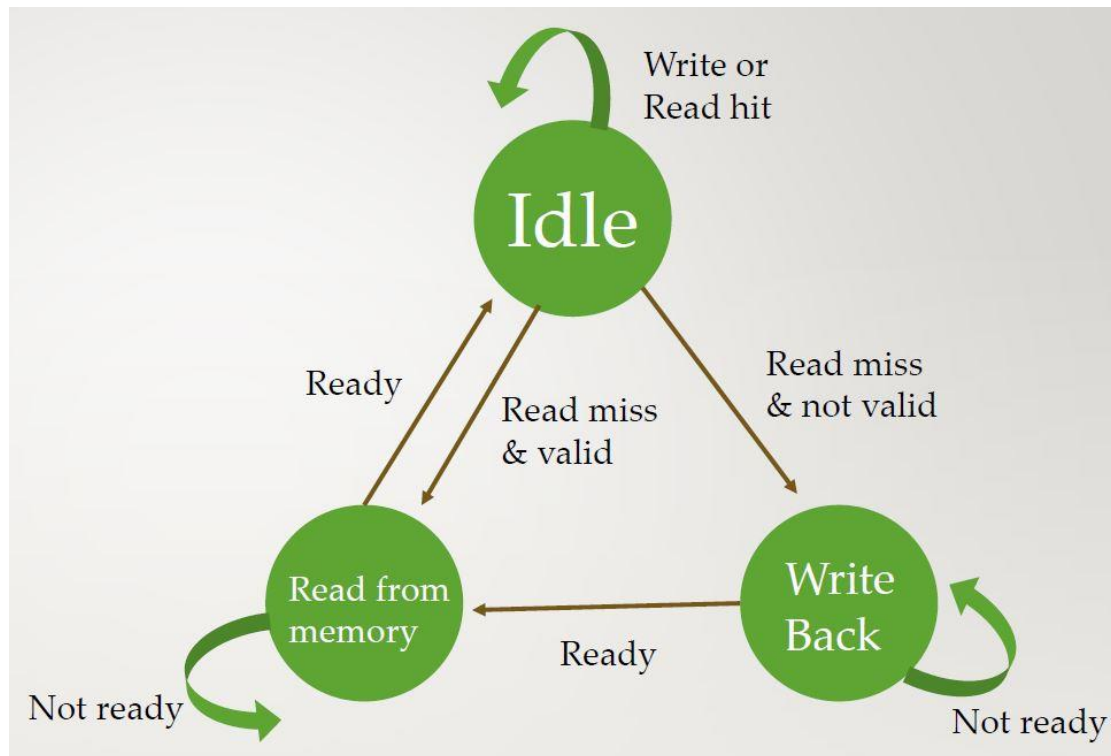


Fig 3. Finite State Machine of Cache

- (1) IDLE state : 在最初開始時處於這個 state，當收到來自 MIPS 的 read signal 時，如果 hit 的話，則會留在 IDLE，並在當下這個 cycle 回傳 data；如果收到 write signal 時且 hit，則也是在當下 cycle 改好 data，並將該 block 的 valid 設為 0。如果 read miss/write miss，則 stall，並依據 valid 的值決定下個 state 是 READ FROM MEMORY / WRITE BACK
- (2) READ FROM MEMORY state: 維持 stall signal = 1，從 slow memory 取 4 words 放進 cache，完成之後跳回 Idle state
- (3) WRITE BACK state: 維持 stall signal = 1，將 cache 中的 data 改回 slow memory 中相對應的位置，並在完成之後跳到 READ FROM MEMORY。

3 state 的優缺點:

優點: 因為 state 數少，所以就可以減少整個 MIPS 在執行上所需要轉換 state 或是 stall 的 cycle number，以達到縮短 latency 的目的。

缺點: 由於在 IDLE state 需要處理很多步驟(因為把原本別的 state 的工作包進來)，因此如果 critical path 在這邊的話，那在合成之後執行的 cycle length 就有可能無法繼續往下壓，以獲得更好的 AT 值。

三、Baseline performance

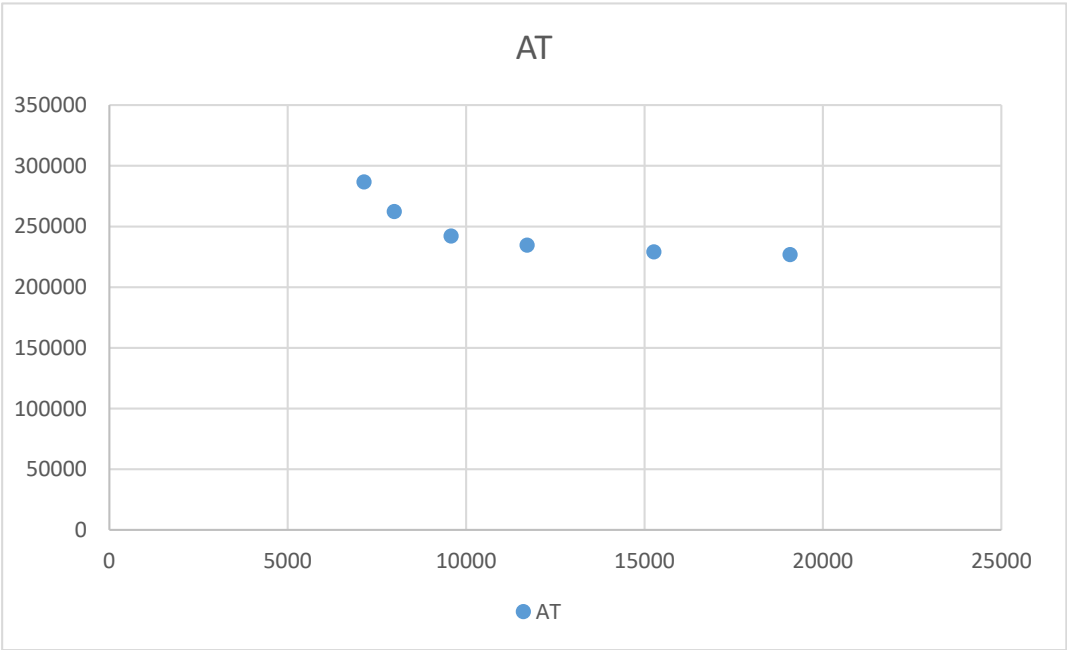


Fig 4 AT 值曲線圖
橫軸:時間(ns) 縱軸:面積(um²)

完成時間 (NS)	面積(UM²)
19085	226723
15268	228975
11715	234572
9583	241989
7145	286617
7990	262171

最好的 AT 值:我們最好的結果是面積 286617.867118 um²，時間是 7145.25 ns

AT 值 = 2.047878465×10^9

四、Extension – Branch Extension

Structure

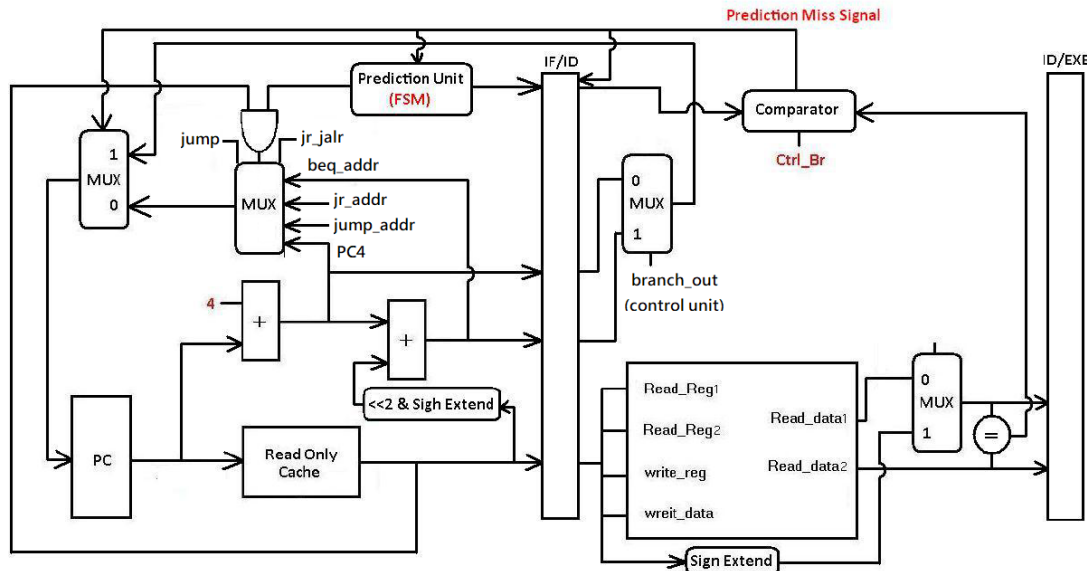


Fig 5. MIPS with Branch prediction unit

有別於先前設計，branch 指令是在 ID 那級才被判斷，這樣會使得每次執行 branch，如果有要跳到新的位址，就必須將下一個要輸入的指令 flush 為 NOP，接著才把正確的位址傳給 PC，讓 PC 去跟 instruction cache 去拿正確的下一個指令，可是這麼一來，就會浪費一個 clock cycle 的時間(NOP)。為了增進效能，因此改為把 branch 拿到 IF stage 就先做判斷，上圖的 prediction unit 就執行猜測 branch 是否有要跳位址的功能，會根據 finite state machine 當前的 state 輸出對應的結果，並且與 instruction cache 輸出的當前指令是否為 branch 去做 AND，假如 2 者都成立，才會讓 PC 下一個位址變成 branch address，否則仍維持 PC+4。但可能會有明明不該跳卻跳，或是該跳卻沒跳這樣猜測錯誤的情況，因此需要於下一級 ID stage 再重新確認一次，將 prediction unit 的猜測結果與 control unit 實際判斷出的結果去做比較，假如真的猜錯了，就需要執行 flush 功能，把猜錯的下一個指令整個清掉，再把正確的位址傳給 PC。

Finite state machine

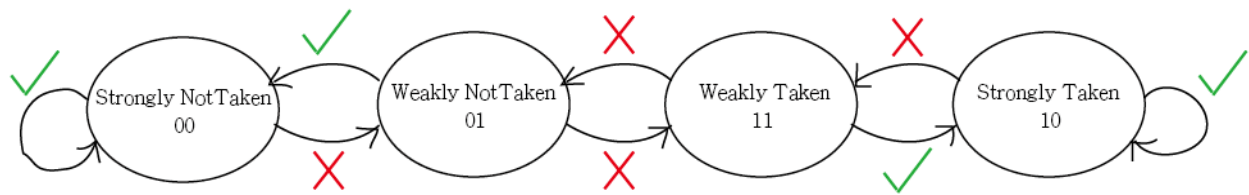


Fig 6. Finite State Machine of Branch prediction

上圖為 branch prediction unit 的 finite state machine 結構。一開始的 idle state 設在 Strongly NotTaken，表示假如有 branch 指令要執行，在尚未從 Register 拿資料前，一律先假定 \$rs 和 \$rt 裡存的資料不相等，所以 PC 不會跳到 branch 指令所指定的位置；如果猜測正確，則維持原 state，但如果猜測錯誤，則為讓下一個 state 跳到 Weakly NotTaken，一樣是假設不跳，但相對來說信心就比較沒那麼高；接著如果又判斷錯誤，那麼下一個 state 會變成 Weakly Taken，也就是改成猜測 branch 都會跳，以下依此類推。可以注意到必須連續判斷錯誤 2 次，才會從 NotTaken 變成 Taken。

Performance

實際跑 I_mem_BrPred 的測資(clock cycle 設 10)，用原本沒有 branch prediction unit 的 MIPS 跑，其執行時間為 4055ns；而用有 branch prediction unit 的 MIPS 跑，執行時間則為 3755ns，整個執行時間少了約 7.4%，可見在某些情況(如：用 branch 判斷某迴圈是否停止)用有 branch prediction unit 的效率較佳；然而跑 I_mem_hasHazard 的測資，反倒是沒有 branch prediction unit(19085ns)較有 branch prediction unit(19155ns)的執行時間略快，原因則在於 I_mem_hasHazard 裡的測資沒有涵蓋太多 branch 指令，所以導致用 branch prediction unit 猜錯而需要 flush 的次數較多，因此在小測資情況無法看出 branch prediction unit 的效能。

Synthesis Result

實際合成出來，clock cycle 可以壓到 3.5，總執行時間及合成面積如下：

	I_mem_BrPred	I_mem_hasHazard
Total cell area(um ²)	289315.035948	289315.035948
Total execution time(ns)	1643.5	7624.7
Total execution cycles	432	2006

五、Extension – L2 Cache

Structure:

我們的 L2 cache 架構類似於 L1 cache，同樣是 direct-mapped & write back 的設計，而總體的容量變成 256words，4 words/block，所以總共 64 個 entry，大致上如下圖：

Valid	Tag	Data
1 bit	22 bits	128 bits

Fig. A block of L2 cache

比起 L1 cache 不同的是，相對於 L1 cache，L2 cache 算是一個 memory(小一點也快一點)，所以在回傳 data 的時候都是一次傳 128bits，寫入的時候同理。

Performance Analysis:

以下將對於有無 L2 cache 進行比較，我們生成了不同長度的數列測資對兩者進行測試，當數列越大(數字越多)，則越可能看出 L2 cache 的影響。

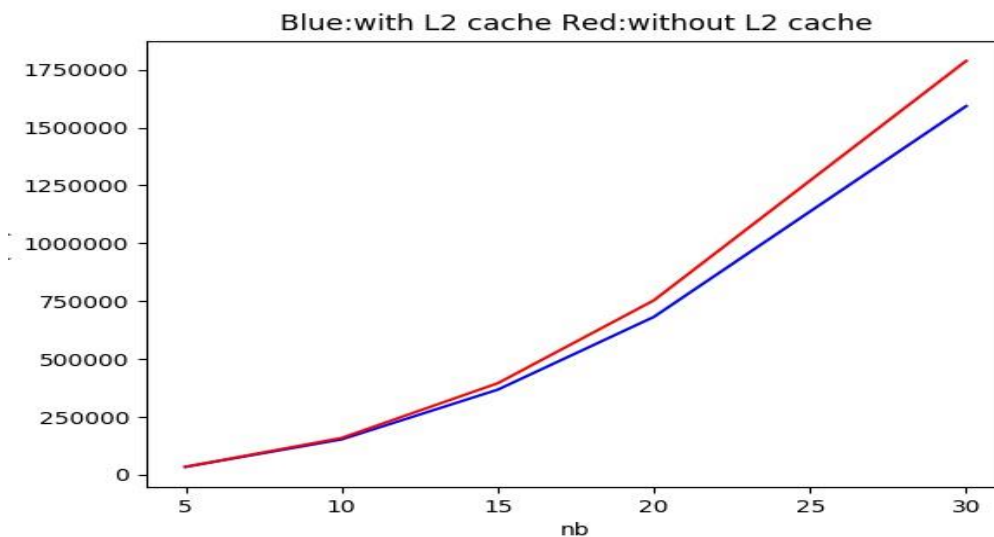


Fig. Simulation time of with and without L2 cache

上圖中橫軸代表 NB 值(越大表示數列越長)，而縱軸代表執行的時間，藍線是有 L2 cache 的，紅線則是沒有；可以看出隨著數列長度增加，有 L2cache 的效果越顯著，所花費的時間與沒有 L2 cache 的逐漸拉開。

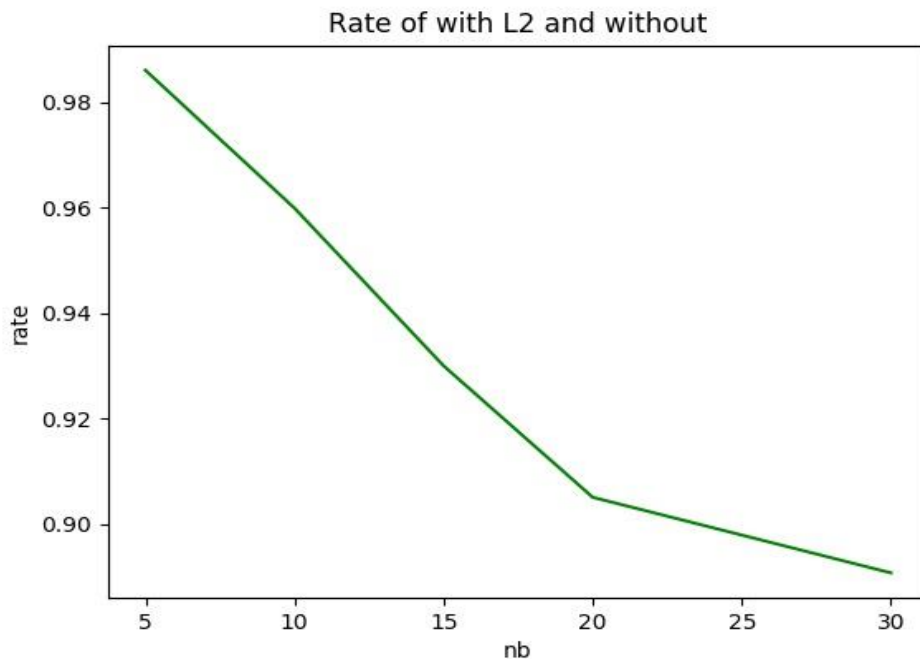


Fig. Ratio of simulation time of without and with L2 cache

上圖所表示的是有無 L2 cache 執行時間的比值隨著 NB 值的變化。可以看出，隨著數列變長，原在 NB = 5 時兩者幾乎相同(比值 0.98)，但到 NB = 30 時，比值來到 0.87，效果顯著。

Synthesis Result:

合成的面積是 1435504 (μm^2)，大約比起 baseline 的多出 1150000 左右，而跑 NB = 20 的測資的總共時間是 123122 ns，clock cycle = 4。

六、Extension – Multiplication & Division

Structure:

我們的乘除法沒有照著講義做，採用直觀的辦法把兩個運算子都先用 flip-flop 存起來，算的過程中會把結果存到另一組 flip-flop 中，一共三組。後來考量到乘法的其中一個運算子要做 sign-extension，而除法的運算子則是要視情況做 2 補數變換，決定把乘除法用來存運算子的位置分開，多開兩組 flip-flop，一共五組。除了用來儲存資料的 flip-flop，我們還開了兩組 5bit 的 counter 來記錄當前加減法的位置。

計算方法:

乘法時單純就按照 counter 的位置來進行加法，counter 算到 31 的時候要做特殊處理，看狀況把要加的數字做二補數變換，同時給存運算子的兩組 flip-flop 新的值，避免連續的乘除法產生 hazard。32 位都加完之後把結果傳出去。

除法一進來時就要先記錄商數跟餘數要不要變成負的傳出去，之後開始做減法，一邊記錄商數，counter 數到 31 的時候把餘數塞進存結果的位置。如果商數跟餘數有要做正負變換，在 flip-flop 的輸出端做。

測資:

原測資沒有測到負數的情況，所以我測試了不同正負組合的運算正確性。另外我也更改了 testbed，在正確執行完之後會跳出 congratulation。

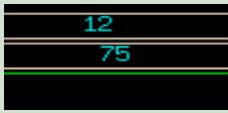
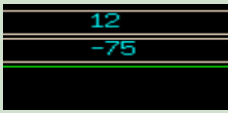

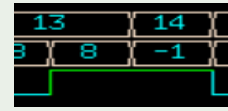
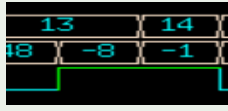
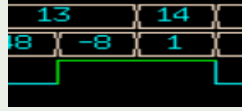
Numbers to be Multiplication/Division	-25 -3	-25 3	25 -3
Multiplication			
Division			

Fig Multiplication & Division result

performance:

合成之後的面積是 344517，跑 MultDiv 的測資通過時間是 1814.75ns。通過的 clock cycle 和沒有加乘除的 CHIP.v 去跑 hasHazard 一樣是 3.5，面積則多了六萬。