DSD final project - Pipelined MIPS

Group : M3
Group member :
b06901014 李筠婕
b06901063 黃士豪
b06901153 林瑩昇

Baseline

Result

1. Area : 300,870 (um^2)

```
Number of ports:                         2387
Number of nets:                         23077
Number of cells:                        20928
Number of combinational cells:          16527
Number of sequential cells:              4387
Number of macros/black boxes:               0
Number of buf/inv:                       5941
Number of references:                      19

Combinational area:             175429.683867
Buf/Inv area:                    44035.648060
Noncombinational area:          125444.651712
Macro/Black Box area:                0.000000
Net Interconnect area:         2739627.321136

Total cell area:                 300874.335580
Total area:                     3040501.656716
```

2. Total Simulation Time of given hasHazard testbench : 6616.5 (ns)

```
------------------------- Simulation FINISH !!-------------------------
================================================================================

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

================================================================================
Simulation complete via $finish(1) at time 6616500 PS + 0
./tb/Final_tb.v:158           #(`CYCLE) $finish;
ncsim> exit
```

3. Area*Total Simulation Time : 1,990,735,041 (um^2 * ns)
4. Clock cycle for post-syn simulation (cycle in sdc, not cycle in testbench) : 3 (ns)

Special Design

1. Cache Buffer

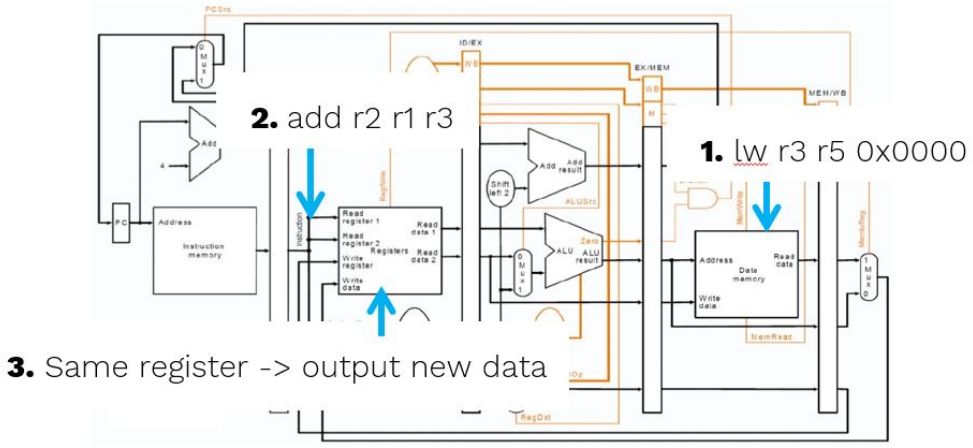
Memory 值的變化以及 mem_ready 的信號都是在 negedge 改變，因此若在 Combinational Part 中，直接在 mem_ready 拉起時使用 Memory 中的新值，將只剩半個 cycle 能夠以此值做運算。如下圖所示：

原先應該在左邊箭頭處讀值並計算，在這邊我們用一個 flipflop(data_ready) 擋住，因此就變成用下一個 cycle (完整的 cycle)去進行計算，雖然會多 delay 一個 cycle，但可以有效減少 critical path 的長度。

2. Lw Stall

當於 WB stage 要求寫入的 register 編號與 ID stage 要求讀取的編號相同時，將會產生 data hazard 的情形。一般的 forwading unit 沒有考慮到這個情形，因此只能在 ID stage stall 一個 cycle，而我們改寫了 register file，使得欲寫入與欲讀出的 register 相同時將會直接輸出要寫入的 data，如此一來就可以減少一個 cycle 的 stall。以下圖為例：

原先在 register[8] 中的 data 應為168，而將寫入 register[8] 的值為0，經過我們的修改，此時的 register file 將不會輸出168，而是比較輸入/輸出 register 編號後改為輸出0，達到減少 cycle 的效果。將此情形畫在 pipeline mips 的 block diagram 上，可以更清楚瞭解其運作：



在進行 lw r3 r5 0x0000 時，ID stage 同時要求讀取 r3 的值，這時 r3 的值將會發生 data hazard，而我們的設計便是避免此情形發生。

Debug Method

Python instruction traverse

每次在 debug 的時候都會遇到一個問題，我們必須把指令的 transition、哪個 register 在哪個指令之後更新、哪個 memory 在哪個指令被讀寫等資訊都寫下來，才能夠與 nWave 中的波型做對照。不過如果今天換一個 pattern 或是事先寫下來的紙不見了，又要在重複一次先前冗長的過程，所以我們直接寫了一個 python 檔，他可以 traverse 過 pattern 檔案來輸出對應的指令，並把更動的 register、memory 對應各個指令輸出。雖然在寫這份 python 檔的時候看似與 final project 毫無相干，但卻幫我們省下很多 debug 的時間。

```
[26742] addr:0x0C, instr:0x3e00008
jr r31
[0]0, [1]0, [2]0, [3]0, [4]0, [5]0, [6]0, [7]0, [8]3, [9]320, [10]280, [11]4, [12]4, [13]4183, [14]4184, [15]1,
[16]52, [17]120, [18]16, [19]0, [20]0, [21]0, [22]0, [23]0, [24]0, [25]0, [26]0, [27]0, [28]0, [29]108, [30]228, [31]192,


[26743] addr:0xC0, instr:0x214a0004
addi r10 r10 0x4
[0]0, [1]0, [2]0, [3]0, [4]0, [5]0, [6]0, [7]0, [8]3, [9]320, [10]284, [11]4, [12]4, [13]4183, [14]4184, [15]1,
[16]52, [17]120, [18]16, [19]0, [20]0, [21]0, [22]0, [23]0, [24]0, [25]0, [26]0, [27]0, [28]0, [29]108, [30]228, [31]192,
```

上圖其實就是一個範例，它可以知道下一個指令要跳轉至哪裡，也可以把運算後的值存入 register file 中。

Extension

Branch Prediction

Result

1. Total execution cycles of given I_mem_BrPred : 1053ns/6ns = 175 cycles

```
------------------------- Simulation FINISH !!-------------------------
================================================================================

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

================================================================================
Simulation complete via $finish(1) at time 1053 NS + 0
./tb/Final_tb.v:158           #(`CYCLE) $finish;
ncsim> exit
[b06014@cad30 ~/DSD_final_project]$
```

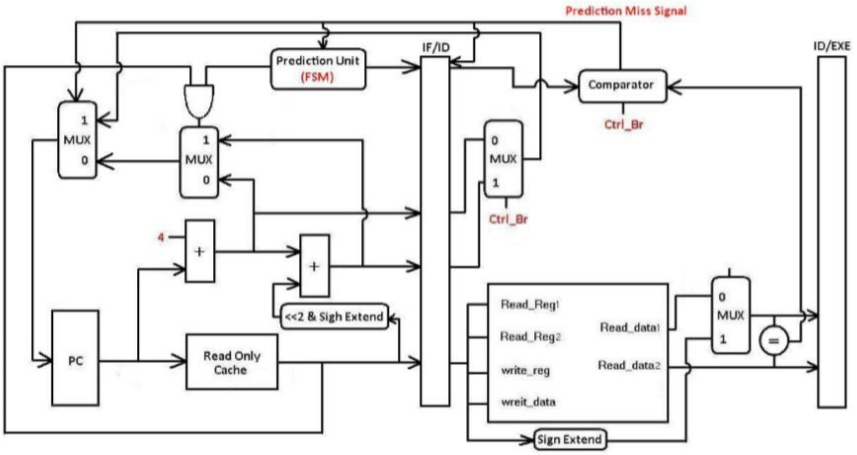
2. Total execution cycles of given I_mem_hasHazard : 10251ns/6ns = 1708 cycles

```
------------------------- Simulation FINISH !!-------------------------
================================================================================

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

================================================================================
Simulation complete via $finish(1) at time 10251 NS + 0
./tb/Final_tb.v:158           #(`CYCLE) $finish;
ncsim> exit
[b06014@cad30 ~/DSD_final_project]$
```

3. Synthesis area of BPU : 6809.968332 (um^2)

Design

原本 baseline 是在 EX stage 才判斷是否要 branch，導致如果實際上是要 branch 的話會有一個 cycle 的 penalty，此外，因為 branch instruction 是否要 branch 常常是連續的(ex. for loop)，因此可以某種程度 predict 是否要 branch。這裡我們實作 branch prediction，將是否要 branch 的決定提前到 IF stage，減少 miss prediction 的所增加的 cycle 數。設計如下圖：在 IF stage 根據 prediction unit 決定是否要 branch，並在 ID stage 判斷前面做的 prediction 是否正確的，若是正確的我們就有成功省下 cycle，若 predict 錯誤，則會浪費一個 cycle。

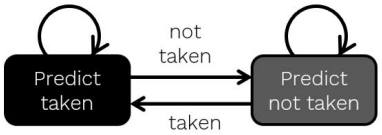


Experiment

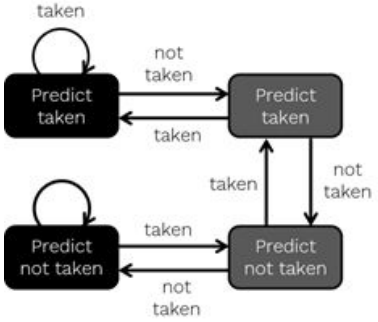
1. Different method

prediction unit 的設計我們實做了三種方法，以下會一一比較。結構如下所示：

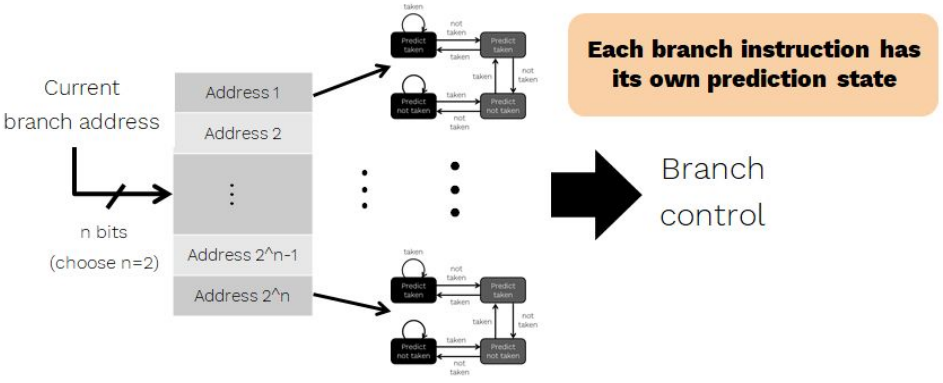
1) 1 bit predictor



2) 2 bit predictor



3) Branch cache

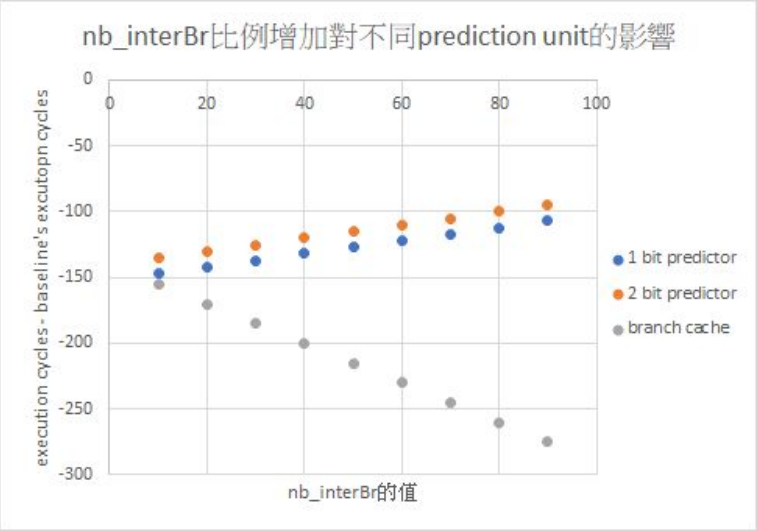


2. Different ratio of patterns

1) (nb_notBr, nb_interBr, nb_Br) 固定總數，比較 baseline (never predict branch)，以及 extension (1 bit predictor, 2 bit predictor, branch cache)，在不同 ratio 下，執行 I_mem_BrPred 所需要的 cycle 數。

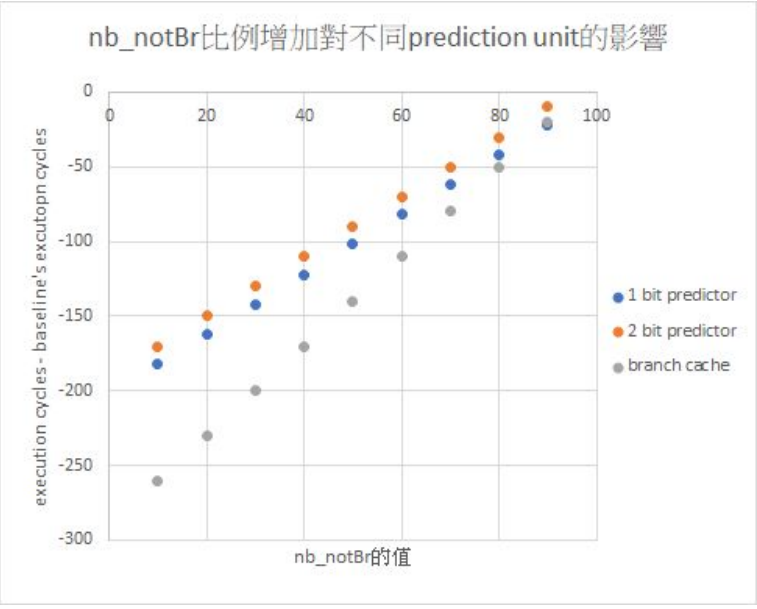
notBr	interBr	Br	Total cycle (Baseline)	Total cycle (1 bit predictor)	Total cycle (2 bit predictor)	Total cycle (branch cache)
10	10	80	580	328	340	320
10	80	10	790	678	690	530
80	10	10	510	468	480	460

2) (nb_notBr, nb_interBr, nb_Br) 固定總數，改變 nb_interBr 的比例，比較不同方法相對於 baseline 的優化程度。



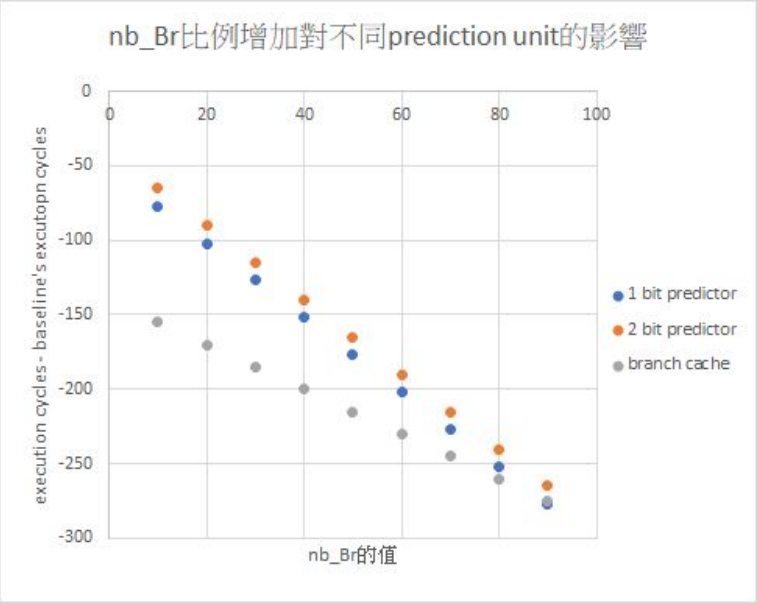
由上圖可以發現，隨著 nb_interBr 的增加，1 bit predictor 與 2 bit predictor 因為會在不同的 prediction state 之間震盪，導致很多 wasted cycles。因此，對 baseline 的優化程度會慢慢變差，而反觀 branch cache，優化程度愈來愈好。

3) (nb_notBr, nb_interBr, nb_Br) 固定總數，改變 nb_notBr 的比例，比較不同方法相對於 baseline 的優化程度。



由上圖可以發現當 nb_notBr 比例增加，不論是 1 bit predictor、2 bit predictor 或是 branch cache 相對於 baseline，saved cycles 變小，這是因為 branch prediction saved cycles 主要是在它正確預測是要 branch 的時候，若是一直 not branch，那 branch prediction 較不能發揮它的優勢。

4) (nb_notBr, nb_interBr, nb_Br) 固定總數，改變 nb_Br 的比例，比較不同方法相對於 baseline 的優化程度。



如上圖，可以發現當 nb_Br 比例增加，不論是 1 bit predictor、2 bit predictor 或是 branch cache 相對於 baseline，saved cycles 都逐漸增加，因為每正確 predict 一次 branch 相對於 baseline 就可以省3個 cycle，發揮 branch prediction 的優勢。

3. AT comparison

1) 加入 brach prediction unit 的 pipline MIPS 最小可以用 cycle time = 5ns 合成，可以過 gate level simulation。

Cycle	Area	Execution time (ns)	AT
10	263,253	2705	712,099,794
6	270,066	1053	284,380,047
5	282,710	1393	393,815,664
4.7	290,066	Violated	X
4.5	296,946	Violated	X

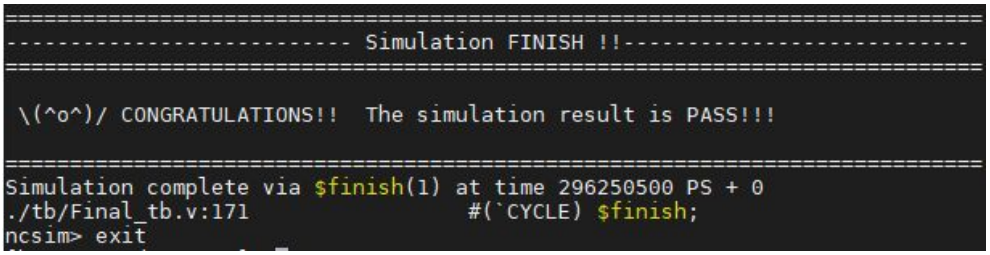
2) 從上表可以發現用 cycle time = 6ns 合成 AT 值是最小的，同樣用 cycle time = 6ns 合成沒有加 branch prediction unit 的 piplined MIPS，跑 BrPred (a=10, b=20, c=20) 的 testbench 可以發現，因為 execution time 幾乎變為一半，有加 branch prediction unit 比起沒有加，AT 值少了將近三分之二。

cycle = 6	A	T(ns)	AT
Baseline	263,256	2523	664,196,285
With BPU	270,066	1053	284,380,047

L2 Cache

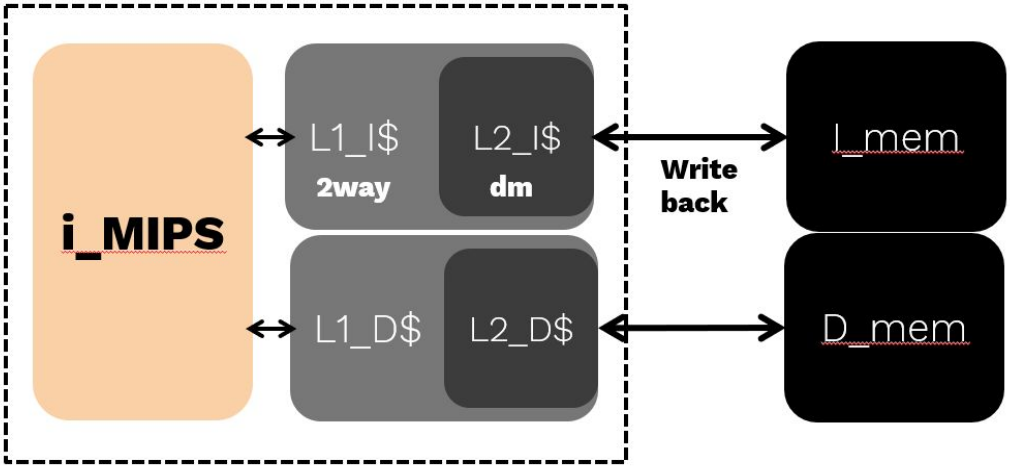
Result

1. Average memory access time : 0.8 (ns)
   我們計算的方法是去看 MIPS 與 D_cache 之間 stall 的 cyle 數，再去計算測資中與 D_memory 溝通的次數，所以 average memory access time = total stall time/total access 次數。我們發現算出來是0.8時難以置信，我們認為這是因為測資太小，所以我們就用 nb20incre30 下去測試，發現 average memory access time 就變成了6.6，也就符合我們的預期。
2. Total execution time of given I_mem_L2Cache : 296,250.5 (ns)



Design

在 L2 cache 的設計上，我們採用了 L1 2-way associative、L2 directed map 的設計，並都使用 Write back 處理 data 的更新。以下我們進行了不同參數下的 L2 cache 表現實驗。



Experiment

1. Different number of words in total

Cache size	Total memory access cycle	D_mem access	I_mem access	Execution time
64	7,370	1,455	19	509,135
128	1,545	294	15	439,235
256	180	21	15	422,855
baseline	6,365	1,253	20	476,655

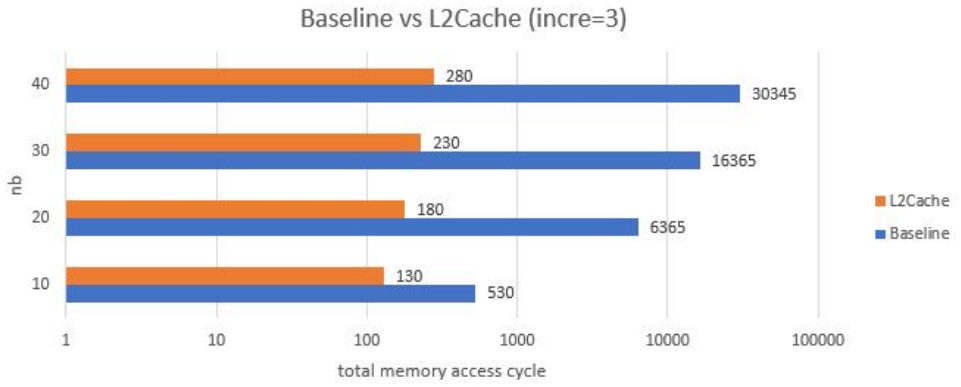
由此實驗可發現，L2 Cache 的 size 越大，能夠節省的 cycle 數越多，而在 cache size 達到256時達到極值。因為此 test case 的原因，size 再調大(大於256)就不會再額外節省 cycle 數。

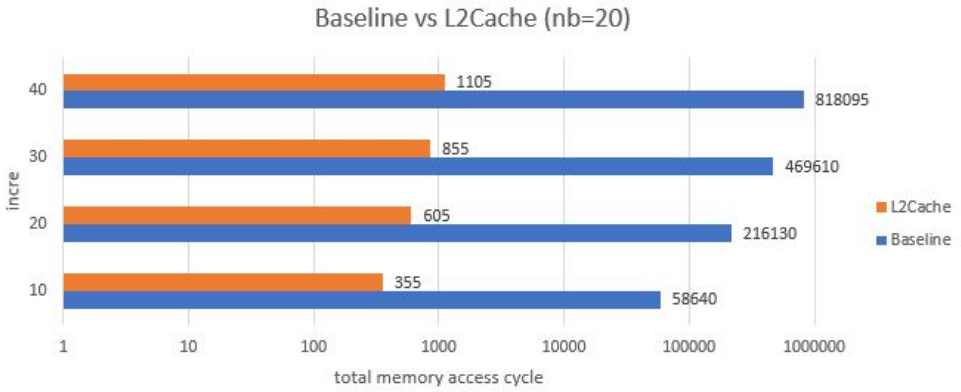
2. Different ratio of patterns

在實驗中，我們固定 nb、incre 兩個參數中其中一個，接著將進行讀寫的 data 量一步步增多去觀察 memory access cycle 的變化。
可以發現，隨者 data 量的增加，L2 Cache 所帶來的加速效果也越發明顯。在(nb=10, incre=3)時，L2 Cache 可以達到4x加速，而到了(nb=20, incre=40)更有740x加速，可見隨著 data 量的增加，L2 Cache 可以為 MIPS 帶來很好的加速。下面則是我們的實驗數據與說明：

		Baseline			Extension				
nb	incre	total mem access cycle	D_mem access	I_mem access	total mem access cycle	D_mem access	I_mem access	**Saved cycles**	**Faster**
10	3	530	86	20	130	11	15	**400**	**4x**
20	3	6365	1253	20	180	21	15	**6185**	**35x**
30	3	16365	3253	20	230	31	15	**16135**	**71x**
40	3	30345	6049	20	280	41	15	**30065**	**108x**
20	10	58640	11708	20	355	56	15	**58285**	**165x**
20	20	216130	43206	20	605	106	15	**215525**	**357x**
20	30	469610	93902	20	855	156	15	**468755**	**549x**
20	40	818095	163599	20	1105	206	15	**816990**	**740x**

上表中，total mem access cycle 是 CHIP 向 slow memory 要資料時所需要花費的總 cycle 數，D/I_mem access 則是 CHIP 與 Data/Instruction memory 要資料時分別需要的次數。由最右側的 saved cycles 可以看出 baseline 與 L2 Cache 執行時總消耗 cycle 數的差異。





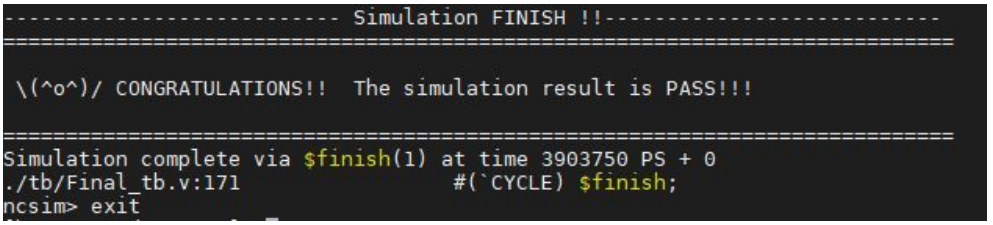
由上面兩張圖可以發現 L2 Cache 在各種情況下的表現都比 Baseline 優秀許多。

Multiplication & Division

乘除法器其實非常相似，基本的方法便是用 iterative 的方法來做運算，而不同 iteration 的設計可以帶來不同的優點及缺點，因此我們進行了兩種實驗：第一是直接利用乘法器進行乘法；另外一種是實作了 Booth Encoding，藉此來加速 iterative 乘法的運算。

Result

1. Area of MultDiv : 380,888 - 263,523 = 117,365 (um^2)
2. Total execution time of given I_mem_MultDiv : 3,903.75 (ns)



3. Clock cycle for post-syn simulation (cycle in sdc, not cycle in testbench) : 7 (ns)

Experiment

1. Different iteration module

直接用內建的乘法器不僅面積很大，因為需要在一個cycle內完成全部計算，需要大量的加法器，使得 critical path 十分的長。反之，如果將32 bit x 32 bit的乘法中的32個 partial product 的計算分成32個 cycle 進行計算、相加，可以大幅減少面積與 critical path 的 delay，為此付出的則是32個 iteration 才能完成一次運算。
在這個實驗中，我們了5種乘法器，分別需要2、4、8、16、32個 iteration 來完成一個乘法。表格中"x"代表沒有做實驗，"violated"代表有 timing violation，而 AT 值則是面積及總共所需的 execution time 的乘積。

Cycle	Iter2	Iter4	Iter8	Iter16	Iter32
8	X	72,747	X	X	X
7	177,661	81,885	X	X	X
6	204,696	96,144	X	X	X
5	Violated	Violated	56,681	27,151	15,120
4	X	X	Violated	32,413	18,370
3	X	X	X	Violated	26,613
AT	2,456,352	2,307,456	2,267,240	2,074,432	2,554,848

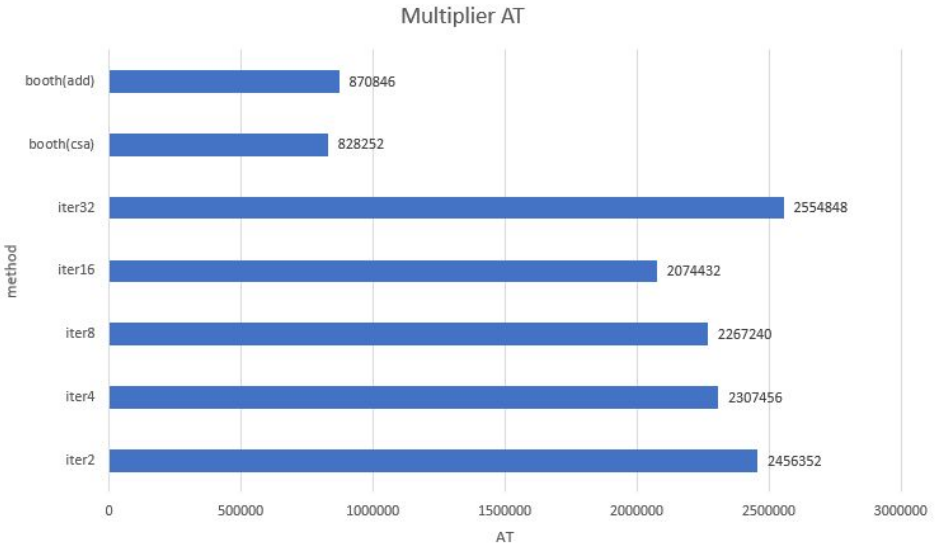
由上表可以發現，將乘法拆解成16個 iteration 可以得到最好的結果。

2. Different algorithm

我們實作了 Booth encoding 的演算法，這個演算法的原理是把 partial product 的加法數量減少，並用 encoding 的方式使得乘法能直接用shift實現，而非使用完整的乘法器。使用此演算法不僅面積能變小，crtical path 也能變短。
在實作的時候發現，運用 Booth encoding 後會產生16個 partial product，此16個 partial product 需要利用加法器相加。我們自己實作一個 Wallace tree adder 去處理這些 partial product，再與 EDA tool 合成出的加法器做比較，發現我們自己實作的成果 crtical path 與面積都較小，為較好的設計。如下表所示：

Cycle	Booth(+)	Booth(csa)
9	103,020	93,385
8	102,876	99,089
7	137,367	120,871
6	145,141	138,042
5	Violated	201,590
AT	870,846	828,252

自行實作的 Booth Algorithm 相較於 EDA tool 合成出的結果約進步了5%左右，criitical path 也比較小。



由上圖可以發現，在 iteration 為16時有最小的AT值；若實作 Booth Algorithm 則更能大幅減少 AT 值。