HW2_Branch Predictor

黃柏竣, 110550123

***Abstract*—此作業先對原先運行之 branch predictor 做分析，並在分析後，決定預計要實作的目標，使 branch predictor 預測成功率變高，進而使效能提升。**

I. INTRODUCTION

Branch 是在程式需要進行條件判斷，進而跳到不同執行處時，所需要的控制，使程式的靈活性、功能性增加。

而 Branch predictor 則是預測 branch 走向的一個元件，若預測失誤，會使 CPU 多幾個 stall cycle，降低執行效率，因此優化 branch predictor 很重要。

II. BRANCH PREDICTOR

A. *One-level branch predictor*

此種 branch predictor 僅考慮 local 的資訊，會以 bimodal assumption 的策略實作，表示 predictor 認為 branch 行為在一段時間內會保持不變，所以會用較簡單的策略預測，並會持續使用相同預測，例如在迴圈時，大多會預測 taken。

不過，因為 one-level branch predictor 無法區分不同分支指令之間的行為模式，因此對於複雜的分支模式可能預測效果較差。

Homework 2 中原先使用的 branch predictor 即為 one-level，使用 32 個 entry 的 branch history table，而 entry 是取 PC 的第 2 到第 6bits，判斷是否 hit 及決定跳到的 address，用另外 2bits 的來記錄該 entry 的 state 決定該 branch 是否 taken。

B. *Two-level branch predictor*

Two-level 則是會考慮 global 和 local 兩者的資訊，再去做判斷預測，通常會透過兩層的 table 去紀錄及索引，使其可記錄到不僅限於 local 的資訊，藉此得到更精準的預測，因此此種 branch predictor 相較於 one-level，更可以處理複雜的分支模式。

在 homework 2 首先實作出的 two-level branch predictor 是 Gshare，除了原本的 branch history table，它增加了紀錄距離此次 branch 前幾次 branch 的 taken 情況，也就是 global branch history，先指定要存的 bit 數，以 shift register 形式儲存，將最近 branch 的 pattern 記錄下來，而此儲存下來的資訊會被用來和 PC 的和前者相同的 bit 進行 XOR，再作為 entry 對另一層的 table 進行查詢，透過兩層的資訊來提高 prediction 的準確度。

之後在實作並測試 Gshare branch predictor 之後，發現表現沒有到很令人滿意，在 BHT 同樣 entry size 的情況下，比原本的分數還差，不過看到他成功預測率蠻高的，又想到上課有講到的 TAGE 概念，所以就打算將原本的跟 Gshare 組在一起，先從 Gshare 找，如果沒有 hit，再拿原本 branch predictor 的結果；多一個原本的 branch predictor 當後盾，若 hit 不到還能去找它，就能緩解 Gshare hit rate 很低的問題，。

III. IMPLEMENTATION

A. *Analysis of the original predictor*

先在 config 的地方將 branch predictor disable 掉，再去跑 CoreMark，觀察 PC 在 CoreMark text section 位址範圍內的總指令執行數、執行 branch 指令的次數、branch 指令 misprediction 的次數以及 branch history table 的 miss 次數。並且也觀察 enable branch predictor 後的這些資訊，並進行比較，除此之外，還對 branch history table 的 entry 數做調整、評估。

總指令執行的計算為判斷 writeback stage 的 PC 是否在 CoreMark 的 text section 位址範圍內，若是的話，就在對應 register 的值+1。

而執行 branch 指令次數則是去看 decode stage 的 output，若判定為 conditional or unconditional branch，便在對應 register 的值+1。

misprediction 則是看從 execute stage 來的判定 output，若 output 為 1 代表預測錯誤，會在對應 register 的值+1。

miss 次數是看，在執行 branch 指令的情況下，確認 PC 是否有 hit 到 table 中紀錄的值(即 tag)，若沒有 hit，就將 register 內的值+1，原本因為判斷是否執行 branch 指令是在 decode stage，但確認 hit 是在 fetch stage，因此在這部分有多用一個 register 存取 fetch stage miss 的情況，可是後來看 simulation 的波形圖，發現這樣反而會差 1 個 clock，所以就取消了這部份的設計。

最後，這些 register 的值是透過 Vivado ILA 得到，並在之後自行做比較和計算分析。

B. *Gshare branch predictor*

用一個 shift register 來存最近幾個 branch 指令的 taken 情況，至於記幾個指令，就跟打算給這個 register 的 bits 數相同，之後就是跟 PC 的同個數 bits 進行 XOR，當作 read 或 write address，對原本設計存在 distri_ram 裡的 BHT 做讀寫，比較需要注意的是處理 stall 的部分，若此 shift register 沒處理到這部分，同樣在 stall 時記錄不該紀錄的東西，雖然不會跑不起來，但得到的數據會蠻差的。

C. *Original one combined with Gshare branch predictor*

實作部分，基本上就是將兩者各有一份資源，有各自的 branch_likelihood、BHT 等等，再從它們的 output 去決定 branch predict unit 該輸出的結果，因為 Gshare 的預測成功率較高，因此以它的預測結果為優先，若 Gshare 沒有 hit，再使用原先 branch predictor 的結果；更新 table 的時機則和原本相同，都是 write enable 時才會同時對兩個 table 做寫入的動作。

IV. RESULTS

A. *Enable versus Disable*

從 CoreMark 自己的跑分得到資訊，disable branch predictor 後，跑出的分數從原本的約 82.86 變成約 74.18，分數降了蠻多。

而根據從 Vivado ILA 得到的值，disable branch predictor 相比於 enable 的總指令執行數則是上升了大約 12%。

B. *Commands ratio of branch predictor with number of entry=32*

透過 Vivado ILA 取得的值，並從其中整理出的資訊如表 TABLE I.。.

TABLE I. PERCENTAGE OF TOTAL COUNT IN COREMARK

Type	*Percentage*
Total commands count	100%
Branch commands count	13.48%
Branch miss count	11.19%
Branch misprediction count	1.35%

C. *Branch commands ratio and CoreMark score with different number of entry*

修改 branch history table 的 entry 數後，進行測試，branch 指令的 hit/miss 及預測正確與否，透過 Vivado ILA 取得 register 內紀錄的值，各值除以總 branch commands 數量；以及進行 CoreMark 的跑分，從各結果整理出的資訊如下表 TABLE II.。.

TABLE II. PERCENTAGE OF BRANCH COMMAND AND COREMARK SCORE WITH DIFFERENT NUMBER OF ENTRY IN COREMARK

Number of entries in BHT	*Hit rate*	*Miss rate*	*Prediction rate*	*Misprediction rate*	*CoreMark Score*
32	17.02%	82.98%	89.99%	10.01%	82.86
64	18.03%	81.97%	89.13%	10.87%	83.19
128	18.52%	81.48%	88.36%	11.64%	83.33
256	18.69%	81.31%	88.56%	11.84%	83.45
512	18.70%	81.30%	88.11%	11.89%	83.48
1024	18.70%	81.30%	88.10%	11.90%	83.49

D. *Branch commands ratio with different number of entry in Gshare*

修改 Gshare branch history table 的 entry 數後(也代表修改 global history register 的長度)，進行 CoreMark 的跑分，從結果整理出的資訊如表 TABLE III.。.

TABLE III. PERCENTAGE OF BRANCH COMMAND AND COREMARK SCORE WITH DIFFERENT BHT SIZE IN GSHARE

Number of entries in BHT	*Hit rate*	*Miss rate*	*Prediction rate*	*Misprediction rate*	*CoreMark Score*
32	0.34%	99.66%	93.78%	6.22%	81.26
64	0.04%	99.96%	93.17%	6.83%	81.61
128	0.04%	99.96%	92.64%	7.36%	81.85
256	0.28%	99.72%	93.33%	6.67%	82.29
512	0.26%	99.74%	94.96%	5.04%	83.00
1024	0.25%	99.75%	95.07%	4.93%	83.35

E. *Branch commands ratio with different number of entry in combined branch predictor*

修改原先 branch predictor 及 Gshare branch predictor 的 branch history table 的 entry 數後(兩者 entry 數相同)，進行測試，原本的 branch predictor 於 entry size=1024 時，LUT、LUTRAM 分別的使用量為 26%、24%；而 Combined branch predictor 於各 512 entry size 時也是相同的使用率；經過測試後，得出的結果整理出的資訊如表 TABLE IV.。.

TABLE IV. PERCENTAGE OF BRANCH COMMAND AND COREMARK SCORE WITH DIFFERENT BHT SIZE IN COMBINED ONE

Both number of entries in BHT	*Hit rate*	*Miss rate*	*Prediction rate*	*Misprediction rate*	*CoreMark Score*
32	17.23%	82.77%	89.43%	10.57%	83.05
64	18.11%	81.89%	88.57%	11.43%	83.32
128	18.58%	81.42%	87.90%	12.10%	83.37
256	18.71%	81.29%	88.58%	11.42%	83.58
512	18.74%	81.26%	90.38%	9.62%	83.97

V. DISCUSSION

A. *Enable or Disable branch predictor*

雖然看到原本的 one-level branch predictor，表現沒有到大多都 hit，可是在 disable branch predictor 後再去做同樣的 CoreMark 跑分，可以從分數看到相較於調整 BHT size 的提升，disable branch predictor 使其非常顯著地下降，可從中了解 branch predictor 的重要性，也看到 branch 預測失

誤帶來的 flush、stall 跑完整個 CoreMark 累積起來也是不容小覷。

B. *Hit rate is low*

在原本的 branch predictor 中，經過分析和測試，發現 hit rate 還蠻低的，即使將 branch history table 的 entry 數量往上調，上升的幅度也不大，其實令我蠻驚訝的，entry 數量變成了兩倍及四倍，碰撞的機會感覺是大大地減少，但可能數量級有點差距，所以即使調了兩倍四倍，還是沒有一個顯著的效果。

C. *Affect of BHT's size changes*

從調整 branch history table 的 entry 數量得到的數據，可以看到雖然在跑分及 hit rate 的部分有微幅上升，但在看預測 branch 後的下個 PC 的正確性，卻反而下降，代表一味地增加 entry 數量也不完全是好事，不只會使資源占用的部分變多，也會讓 misprediction rate 上升，且一直測試到 1024 個 entries，也發現成長幅度越來越小，主要在 entry 數為 256 時，有較大變化，我認為若變化這麼小，可以選擇少消耗一些資源。

在 Gshare branch predictor 中，分數的浮動比原本的還要大，表示 entry size 變動對 Gshare 的影響較大，可能是因為記錄到的 branch pattern 數量變多，使得 Gshare 有更多資訊來去做判斷。

D. *Gshare branch predictor*

Hit rate 低的情況，在 Gshare branch predictor 中更加嚴重，連 1%都達不到，沒想到經過 XOR 之後想要再找到對應的 address 這麼不容易。而因為 hit rate 低的關係，就算成功預測率很高也不夠有用，畢竟要先 hit 之後，才會有後面的預測行為，這也使得，在同樣 BHT entry size 的情況下，Gshare branch predictor 在 CoreMark 跑出來的分數沒有比原本的還高。

E. *Combined branch predictor*

從 Combined branch predictor 的實驗結果可看出，Gshare Hit rate 低的情況，可以透過另一個 hit rate 較高，預測正確性也不算太差的來平衡，達到使用相同 LUT、LUTRAM 的情況下，卻得到較佳的成績，雖然這一點要在較高 entry size 時才看得出來(實驗結果顯示各 256 size 之後)，不過這個成果仍能展示出這個嘗試是有效的。

F. *Reflection*

一開始看到 Gshare 的表現比原本的差，令我有些驚訝，也想要嘗試改做 TAGE，但看 paper 後，還是不清楚 Ctrl 及 useful bit 的部分，可是感覺還是有學到那種 multi-level、多次判斷的架構概念，加上又看到 Gshare 的預測正確性蠻高，是 hit rate 太低導致分數沒有比原本的高，所以就想要用那樣的概念去取長補短，幸好實作之後出來的成果有驗證我的想像，雖然分數沒有比原本的 branch predictor 高多少，但我認為將自己的想法實作並驗證成功，對我來說仍是一次很不錯的經驗。

APPENDIXES

Fig. 1. Line graph of CoreMark score of different branch predictors