

# Computer Organization 2019

## HOMEWORK 6

系級： 資訊工程系    學號： F74109016    姓名： 葉惟欣

### 問題(Question)

Q1. How do you know the number of block from input file?

因為第一行的輸入為 cache size(word)，第二行的輸入為每一個 block 的大小 (word)，因此可以得知 block 的數量就是所以的  $\text{cache size} / \text{block size} = \text{block number}$ .

Q2. How do you know how many set in this cache?

每個 set 會有一些 block 稱叫多少 way，由  $\text{cache size} / \text{block size} = \text{block number}$ ，之後的 block number 數量再去除以每個 set 有多少個 block 就可以得到有多少的 set.

Q3. How do you know the bits of the width of the Tag ?

因為cache 的size 是用 word size紀錄，且每個block 的size 也是用多少word記錄 (word addressable)，所以這裡就不考慮block offset 中的 byte offset (通常一個 word 有4個bytes，則byte offset 佔2個bits)。而每個block 有  $2^x$  個words，則我們會需要x個bits來表示word offset。如果1個block只有1個word  $x=0$ 。有4個words  $x=2$ 。

Block address		Block offset	
tag	index	Word offset	Byte offset
32-2-x-n個bits	N個bits	X個bits	
64-2-x-n個bits			

而index 需要n個bit取決於cache 會被分成多少個set，有可能一個set裡面就只有1個block(1 way = directed mapped)，有可能一個set裡面有4blocks(4 ways)。被分成  $2^n$ 個sets index 就要n個bits表示。

如果一個address 是32bits，(32位元的電腦)的話，且一個words為4 bytes，則32bits 分給byte offset 為4取底為2的log，byte offset 為2個bits.

剩30個bits分給index 為n個bits，給word offset 為 x 個bits。則留給tag 的就為  $30 - n - x$ 。也因此當我計算時我會先將word address 除以block size取商數，也就相當於把block offset 的部分先取掉，得到的商數為即為index + tag 所表示的部分，

再除block number 取餘數為index 的部分，商就為tag bits 所表示的部分。

```
if(Asso == 0){ //Directed Mapped
    BLOCK cache[blockNum];
    while(FILEIN>>wordAddr){
        int blcokAddr = wordAddr/blockSize;
        int index = blcokAddr % blockNum;
        int tag = blcokAddr/blockNum;
```

如果是64位元的電腦就是用64bits去算而不是再用32bit。

幾位元的電腦是在說word有多少個bits，也就是一次讀是讀多少個bits，而address跟data一樣都是需要去讀的，所以如果是32位元的電腦，如果要讀一個address(byte addressable)的話則同樣也是讀32個bits。可以想成位址也是資料的一種類型。64位元的也已次類推。

Q4. Briefly describe your data structure of your cache.

Cache 的資料結構為block 的陣列，最小的單位為block，而block 裡面有整數tag 與 bool valid。

如果為directed map的話，因為每個index只會對應到一個block所以一維的block (size 為blockNum)陣列。

而4 way set associative 相當於一個index 會對應到4個block，所以會是 (setNumber \* 4)的二維陣列，但程式碼的部份因為寫起來比較方便，所以我統一都將表達維度1的size 為blockNum。

而fully set associative則為一個index裡面有blockNum的block 數量，相當於blockNum way set associative。因此方便，我也同樣將它設定為一維的block陣列 (size 為blockNum)。

Q5. Briefly describe your algorithm of LRU.

LRU 為 least recently use algorithm，是一種選擇要哪個 block 被替換掉的演算法。因此當如果 set 裡面的 blocks 的 tag 都有 valid 的 value 的話，那就要選出一個 block 作為 victim，將新的 block 放在該 victim 原本的編號 way # 的位置。

下表為我的 LRU 演算法的演示，以測資 3 為例。

在表中行 way (0, 1, 2, 3)下面是代表 tag 的 value，而 tag 前面為 most recently use 的排名，意思是排名第一名是最新被用的，排名第 4 是最久以前備用的，真

正的實作過程我並沒有去紀錄排名，而是用 circular double linked list 來直接實現，等下會說明。

淺藍色的部分為在當讀第 12 個 word address 後，在 index 7 中最新的 block 也就是 most recently used 為 way 2，tag 為 2205，但當讀第 15 個 word address 後，同樣是在 index 7 的 block 而讀的 tag 剛好與 way3 的 tag 2207 對應，因此 way3 就變成最新被使用的排名變第一名，而剛剛原先的第一名為 way2 就變第二名了。

綠色的部分也是相同的概念因為在讀第 22 個 word address 後，在 index 7 中最新的 block 也就是 most recently used 為 way 2：1980，而之前的 way3 2207 就變成第二名。

在讀第 23 個 word address 後，在 index 7 中最新的 block 也就是 most recently used 又變成為 way 3：2207，而之前的 way2 1980 就變成第二名。

	cache number	8									
cache word size	256										
block word size	8										
4-way set associative	1	tag	cache index	way	way	way	way				
LRU	1 block address	商1	餘1	0	1	2	3				
1	126242	15780	1972	4	1-1972	2-0000	3-0000	4-0000	miss	-1	
2	129024	16128	2016	0	1-2016	2-0000	3-0000	4-0000	miss	-1	
3	126224	15778	1972	2	1-1972	2-0000	3-0000	4-0000	miss	-1	
4	125886	15735	1966	7	1-1966	2-0000	3-0000	4-0000	miss	-1	
5	141308	17663	2207	7	2-1966	3-0000	4-0000	1-2207	miss	-1	
6	126514	15814	1976	6	1-1976	2-0000	3-0000	4-0000	miss	-1	
7	141172	17646	2205	6	2-1976	3-0000	4-0000	1-2205	miss	-1	
8	141174	17646	2205	6	2-1976	3-0000	4-0000	1-2205	hit	-1	
9	141176	17647	2205	7	3-1966	4-0000	1-2205	2-2207	miss	-1	
10	141178	17647	2205	7	3-1966	4-0000	1-2205	2-2207	hit	-1	
11	141180	17647	2205	7	3-1966	4-0000	1-2205	2-2207	hit	-1	
12	141182	17647	2205	7	3-1966	4-0000	1-2205	2-2207	hit	-1	
13	141184	17648	2206	0	2-2016	3-0000	4-0000	1-2206	miss	-1	
14	141186	17648	2206	0	2-2016	3-0000	4-0000	1-2206	hit	-1	
15	141304	17663	2207	7	3-1966	4-0000	2-2205	1-2207	hit	-1	
16	141306	17663	2207	7	3-1966	4-0000	2-2205	1-2207	hit	-1	
17	126308	15788	1973	4	1-1972	3-0000	4-0000	1-1973	miss	-1	
18	126700	15837	1979	5	1-1979	2-0000	3-0000	4-0000	miss	-1	
19	125652	15706	1963	2	2-1972	3-0000	4-0000	1-1963	miss	-1	
20	126707	15838	1979	6	3-1976	4-0000	1-1979	2-2205	miss	-1	
21	125928	15741	1967	5	2-1979	3-0000	4-0000	1-1967	miss	-1	
22	126776	15847	1980	7	4-1966	1-1980	3-2205	2-2207	miss	-1	
23	141308	17663	2207	7	4-1966	2-1980	3-2205	1-2207	hit	-1	
24	125704	15713	1964	1	1-1964	2-0000	3-0000	4-0000	miss	-1	
25	126926	15865	1983	1	2-1964	3-0000	4-0000	1-1983	miss	-1	
26	129030	16128	2016	0	1-2016	3-0000	4-0000	2-2206	hit	-1	
27	126488	15811	1976	3	1-1976	2-0000	3-0000	4-0000	miss	-1	
28	129032	16129	2016	1	3-1964	4-0000	1-2016	2-1983	miss	-1	
29	126576	15822	1977	6	4-1976	1-1977	2-1979	3-2205	miss	-1	
30	126254	15781	1972	5	3-1979	4-0000	1-1972	2-1967	miss	-1	
31	126196	15774	1971	6	1-1971	2-1977	3-1979	4-2205	miss	victim-1976	
32	126454	15806	1975	6	2-1971	3-1977	4-1979	1-1975	miss	victim-2205	
33	129032	16129	2016	1	3-1964	4-0000	1-2016	2-1983	hit	-1	
34	129034	16129	2016	1	3-1964	4-0000	1-2016	2-1983	hit	-1	
35	126966	15870	1983	6	3-1971	4-1977	1-1983	2-1975	miss	victim-1979	

接下來是橘色的部分當要讀第 31 個 word address 發現不在 cache 中，因此要把 cache 中的某個 block 替換掉，因此就直接選當前替換時排名為 4 的 block，(也

就是距離該 block 最久以前被用的)。舉例:在讀完第 29 個 address, index 為 6 的第四名為 way1 1976, 所以第 31 個 word address 要選一個替換掉, 就選 way1, 且印出 victim 1976。

程式碼: Fully set associative

```
cache.cpp
148 int searchBlockLRU(NODEFULL** Algo, BLOCK cache[], int tag, int* numOfHit, int* numOfMiss, int blockNum){
149     NODEFULL* curr = *Algo;
150     int goal = -1, emptyNum = 0;
151     for(int i=0; i<blockNum; i++){
152         if(cache[curr->blockNum].tag == 0) emptyNum++;
153         if(cache[curr->blockNum].tag == tag){ //exist //update
154             goal = curr->blockNum;
155             //exist but is least recently use
156             if(((*Algo)->blockNum == goal) *Algo = ((*Algo)->front);
157             //exist but is not least recently use
158             else{
159                 NODEFULL* originNew = ((*Algo)->back);
160                 if((originNew->blockNum != curr->blockNum){
161                     //connect current's front and back
162                     ((*curr->back)->front = (*curr)->front;
163                     ((*curr->front)->back = (*curr)->back;
164                     //update new
165                     ((*Algo)->back = curr;
166                     (*curr)->front = *Algo;
167                     (*curr)->back = originNew;
168                     (*originNew)->front = curr;
169                 }
170             }
171             break;
172         }
173         curr = (*curr)->back;
174     }
175     if(goal == -1 && emptyNum==0){
176         int victim = cache[(*Algo)->blockNum].tag;
177         cache[(*Algo)->blockNum].valid = true;
178         cache[(*Algo)->blockNum].tag = tag;
179         *Algo = ((*Algo)->front);
180         (*numOfMiss)++;
181         return victim;
182     }
183     if(goal == -1 && emptyNum!=0){
184         cache[(*Algo)->blockNum].tag = tag;
185         cache[(*Algo)->blockNum].valid = true;
186         *Algo = ((*Algo)->front);
187         (*numOfMiss)++;
188         return -1;
189     }
190     if(goal != -1){
191         (*numOfHit)++;
192         return -1; //not replacement;
193     }
```

第 151 到 174 行 為尋找 set 中有沒有那個 block, 跟目前的 block 的 tag 比對, 且同時記錄 empty block 的數量。如果有找到該 tag, 則需要做排名的調整, 我是直接調整 circular double linked list 的 way 相對位址, LRU[index]永遠紀錄排名最後一名的 way number 的 node 當找到的相同的 tag 剛好是目前的最後一名, 則將他變成第一名, 也就是第 156 行, 因此就將 LRU 往前移一個移到原先的倒數第二名。讓倒數第二名變成最後一名, 而因為 circular 的關係最後一名的下一個就為第一名。

如果找到的 tag 為中間節點, 那就需要把該節點變成第一名, 也就是一到 LRU[index]指向的節點之後一個也就是第一名的位置, 將原本的第一名接在新的第一名之後, 且不改變原先其他節點的相對位址, (第 158-169)行做調整這些的



動作。

上述是有找到節點的奇況，如果沒有找到節點的話又分為兩個 case，一種是沒有空的 block(第 175 到 182 行)，第二種是還有空的 block(第 183-189 行)。

第一種就直接把現在 LRU 指向的最後一名當作 victim，用新的 tag 取代。取代後的 way # 為第一名(最新用的)，如此以來就要把 LRU[index] 往前移一個 node(第 179 行)移到原先的倒數第二名，讓倒數第二名變最後一名。

第二種因為還有空，有空的 block 一定包含 LRU[index] 目前所指向的 node，因為當這個操作做完，LRU[index] 會在往前移一個 node(第 186 行)，也就是他會一直記錄著距離被使用最久遠的 block，而空的 block 正好從未被使用，所以一定一直被 LRU[index] 紀錄。

4-way set associative(跟上面的一樣，只是排名從 1-4)

```
cache.cpp
195 int searchWayLRU(NODE* Algo[], BLOCK cache[][4], int index, int tag, int* numOfHit, int* numOfMiss){
196     NODE* curr = Algo[index];
197     int goal = -1, emptyNum = 0;
198     for(int i=0; i<4; i++){
199         if(cache[index][(*curr).wayNum].tag == 0) emptyNum++;
200         if(cache[index][(*curr).wayNum].tag == tag){ //exist update
201             goal = (*curr).wayNum;
202             //exist but is least recently use
203             if((*Algo[index]).wayNum == goal) Algo[index] = ((*Algo[index])).front;
204             //exist but is not least recently use
205             else{
206                 NODE* originNew = ((*Algo[index])).back;
207                 if((*originNew).wayNum != (*curr).wayNum){
208                     //connect current's front and back
209                     ((*curr).back).front = (*curr).front;
210                     ((*curr).front).back = (*curr).back;
211                     //update new
212                     ((*Algo[index])).back = curr;
213                     (*curr).front = Algo[index];
214                     (*curr).back = originNew;
215                     (*originNew).front = curr;
216                 }
217             }
218             break;
219         }
220         curr = (*curr).back;
221     }
222     if(goal == -1 && emptyNum==0){
223         int victim = cache[index][((*Algo[index])).wayNum].tag;
224         cache[index][(*Algo[index]).wayNum].valid = true;
225         cache[index][(*Algo[index]).wayNum].tag = tag;
226         Algo[index] = ((*Algo[index])).front;
227         (*numOfMiss)++;
228         return victim;
229     }
230     if(goal == -1 && emptyNum!=0){
231         cache[index][(*Algo[index]).wayNum].tag = tag;
232         cache[index][(*Algo[index]).wayNum].valid = true;
233         Algo[index] = ((*Algo[index])).front;
234         (*numOfMiss)++;
235         return -1;
236     }
237     if(goal != -1){
238         (*numOfHit)++;
239         return -1; //not replacement;
240     }
241 }
```

Q6. Run trace2.txt, trace3.txt and get the miss rate and put it in your report.

Trace1. Miss rate 為 1

Trace2. Miss rate 為 0.5

Trace3 Miss rate 為 0.216545

Trace4 Miss rate 為 0.000597

可以看到 fully associative 的 miss rate 很低，比 trace3 的 4-way set associative 低很多，但在找相對應的 index 卻會花較多的時間，但 CPU 真正在做的時候是 parallel，所以花的其實不是時間是硬體的成本，因為要與每個 block 的 tag 做比較，而 4-way set associative 的 Miss rate 已經比 directed mapped (trace1 trace2) 低了，且每次只有 4 個 block 要做比較，硬體成本也比 fully associative 低很多。因此在 cache size 較大的時候會傾向用 4-way associative，來減少硬體成本，而 cache size 較小，要求 miss rate 很低的話，則傾向用 fully associative，因為也不需要那麼多 block 平行比較。

trace2.out - 記事本	trace3.out - 記事本	trace4.out - 記事本
檔案(F) 編輯(E) 格式(O) 檢視(V)	檔案(F) 編輯(E) 格式(O) 檢視(V) 說明	檔案(F) 編輯(E) 格式(O) 檢視(V) 說明
-1	-1	-1
-1	1968	-1
-1	-1	-1
-1	-1	-1
-1	-1	-1
-1	-1	-1
-1	-1	-1
-1	-1	-1
3066	-1	-1
-1	1863	-1
3066	-1	-1
-1	-1	-1
-1	-1	-1
Miss rate = 0.500000	Miss rate = 0.216545	Miss rate = 0.000597

## 心得(Report)

(請寫下完成本次作業的心得、學到哪些東西、困難點的部分。)

(Please write your learned lesson and conclusion, and difficult point.)

這次的作業剛開始看完教授的講義跟課本，我以為我都懂了，所以就開始寫，但寫了之後才發現還有很多細節沒有注意到，像是測資給的是 word address，然後只要 tag 相同且 valid = true，就是 hit，不用去管讀的 word address 是否相同，即便不同只要比對成功就是 hit。而 word address 的部分，也上網看了很多影片，才慢慢摸索出不用看 byte offset 的部分，因為最小單位是 word，所以這部分可以說是省略掉了。在寫 LRU algorithm 的時候因為有一些指標要比較小心，這是比較困難的部分，至於 FIFO 因為就只是把 LRU，如果有讀到已經存在 cache 的相同 tag 的 block 要更新的部分，所以當寫完 LRU 後這邊都變得很容易了。只要觀念對了，其實後面都只是程式碼實作出來邏輯的問題。雖然在從起初到真正了解一堆名詞 word, block, block offset, index, tag, set number, block number, cache size, block size 花了很多時間，但因為有花這些時間真正去了解，之後寫程式碼就順利很多了。