Project 3：

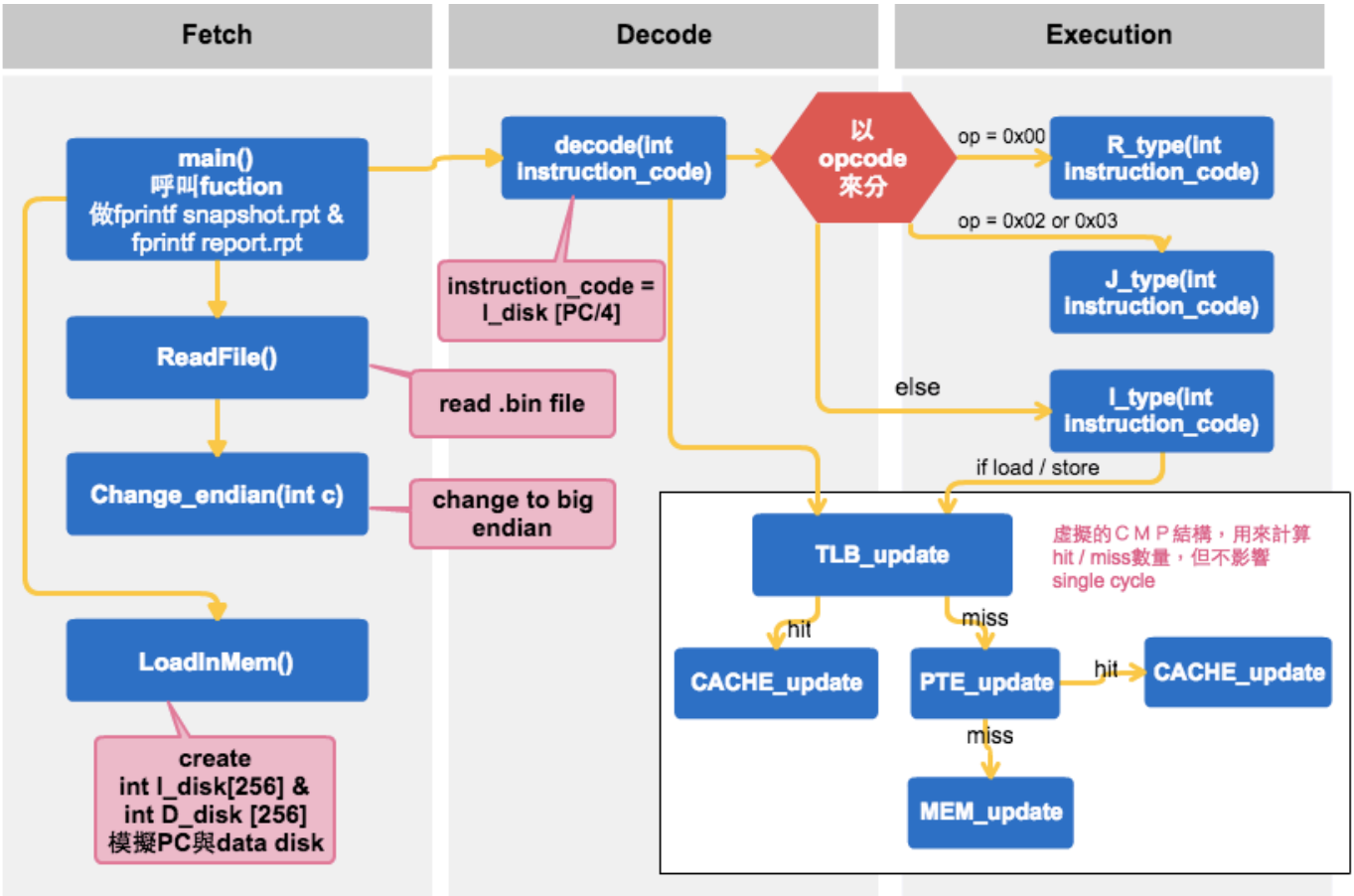
Cache_Memory_Pagetable(CMP)　Report

102062222 張心愉

● Simulator design & elaboration

Simulator 流程簡圖 ：

說明：最上面灰色方塊為 simulator 階段，藍色方塊為程式內 function 與簡略說明，粉紅色方塊為對該方塊的 comment，紅色方塊為如何 decode 出三種 type 指令，黃色線段為呼叫被指向的 function（以下會有詳細說明），線段上註解為呼叫該 function 條件。



Code design 設計說明：

這次的 project 是由 project 1 single cycle 延伸而成，所以保留了第一次所有的 function，除了拿掉 error detect 的部分。

而 project3 要設計出有 TLB,PTE,MEM,CACHE,DISK 等結構，因此我選擇用 struct 來區分與實作。

Structure 設計：

```
static struct CACHE{
    int *LRU_order;
    int *valid;
    int *data;
    int *tag;
} I_CACHE , D_CACHE;

static struct MEMORY{
    int *LRU_order;
    int *valid;
    int *data;
} I_MEM , D_MEM;

static struct PAGE_TABLE_ENTRY{
    int *content;
    int *valid;
} I_PTE , D_PTE;

static struct TLB{
    int *LRU_order;
    int *valid;
    int *content;
    int *index;
} I_TLB , D_TLB;
```

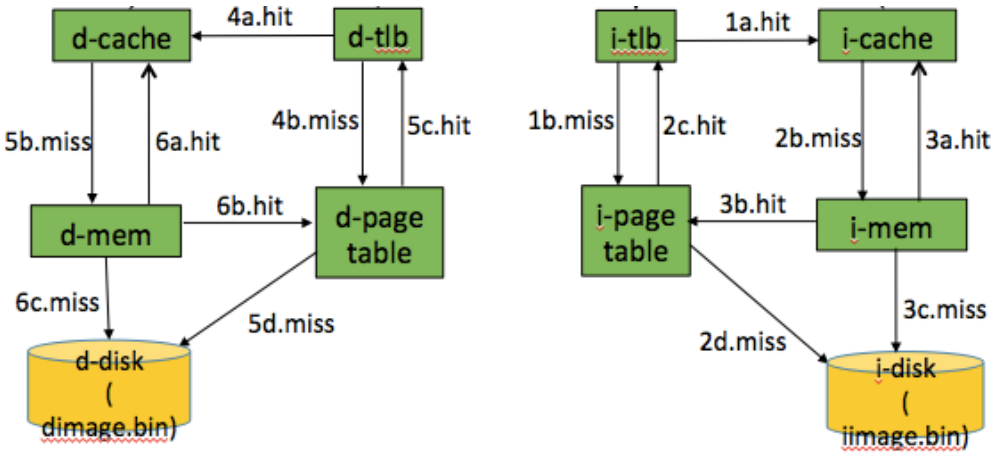
這邊設計的邏輯是根據 code 實作方便而設計的，由於此次 project 要求要用 LRU order 來判斷要取代掉哪一個 entry，因此在 CACHE , MEM , TLB 都加上 LRU_order 來儲存誰是最舊的資料。(PTE 不需加 LRU。在 PTE 中 address / pagesize 的位子就是直接存指到該 MEM 的 index。)

而 LRU_order，存放的是 index，我是將最新資料 index 放到最後面，將其餘往前移，所以最舊的 index 就是 LRU_order [0] ，要取代掉時也直接取這個就好。TLB 不需要 tag，因題目要求 fully-associative，我選擇用 index 來區分。

Data 欄位的存在沒有其必要性，因為此次 project 僅只是要求要實作 hit / miss 數，所以 data 的資料仍然可以從 disk 取就好。但為了實作的直觀性，我還是加上 data 欄位。

雖然此次題目要求「Both instruction memory and data memory adopt **write-back/allocate** policy 」，但因並沒有要實做出 memory hit / miss，所以並不用真的執行 write back policy，要資料時依然直接去 disk 抓即可。

如何更新 TLB，PTE，MEM，CACHE：



由助教給的 tutorial 可知，I TLB 與 D TLB 的查找方式其實是相同的(且因為不用做 WB policy)，以下分 TLB , PTE , CACHE 分別討論各種情況。

1. TLB hit：(TLB.valid == 1 && TLB.index == VA / pagesize)

表示要找的資料在 memory 內且這種情況下不用再去 PTE search，TLB content 所放的即是 physical address，並用此 PA 去 CACHE 查找資料。

2. TLB miss：先到 PTE 查找

a. PTE hit：表示資料在 memory 裡面但是 TLB 沒有記錄，先將 TLB content 更新（等於 PTE content），TLB 優先選擇 valid 為零的 entry，若無則選 LRU_order 最少用到的該 entry 取代。並以此 content 得到的 physical address 到 CACHE 找資料。

b. PTE miss：表示資料沒有存在 memory 中，(因為 PTE 對到的位子即是 memory 位子，如果 page fault 就表示該位子存的不是所要的 address)。此時需要回到 disk 抓 data，取代掉 memory 裡面 valid = 0 的 entry 或者是 LRU_order 最少用到的那個 entry。得到 content 後再去 CACHE 找資料。

＊＊＊這邊要注意！當 page fault 時，原本存在 memory 裡面的那個 page 會被替換掉，此時可能需更動 TLB , PTE , CACHE。若是 PTE , TLB 有記錄到該被

替換掉的 page，則 valid 需設為零。且 CACHE 中有碰到該 page 的 block 也必需將 valid 設為零。

3. CACHE hit：（CACHE.valid == 1 && CACHE.tag == PA / blocksize）

表示 data 已經在 CACHE 中。

4. CACHE miss：

表示 data 不再 CACHE 中而在 memory 內，將 valid 為零的 block 取代，或者是 LRU_order 最少用到的 block 取代掉，但不用更新 memory LRU。

以上四種情況無論 hit / miss 皆須做 LRU_order 順序更新，因為很多取代的地方都要考慮 LRU_order 的情況，並在每次更新後重新整理 LRU_order 順序。而前面有提過 I / D 的查找方法其實是相同的，因此整份 code 只要實做一次查找的方法，並複製貼上即可。

以下敘述程式主要 function 內容：

main()

這邊與 project 1 幾乎都一樣，我所加的部分為，在 ReadFile & LoadInMem 結束後，增加一個新 function –AllocateMemory，用來動態配置記憶體。

AllocateMemory (argc , argv)——題目要求能夠藉由 argv , argc 手動設定 TLB 等空間大小，先給定一個陣列值作為 default size：

```
/* I-MEM-size , D-MEM-size , I-pagesize , D-pagesize , I-CACHE-size
 I-blocksize , I-set.associativity , D-CACHE-size , D-blocksize , D-set.associativity */
int parameter[10] = {64, 32, 8, 16, 16, 4, 4, 16, 4, 1};
```

接著判斷 argc 是否等於 11，若是，就將 argv 的值轉成 int 並依序放入 parameter 中，並由 parameter 的值算出各空間總體大小，以此 calloc 記憶體，並在 calloc 結束後，給定所有空間初始值，valid = 0 , tag = -1 , index = -1 , content = -1。

以下是各空間整體大小算法：

```
I_MEM_block_num = parameter[0] / parameter[2];
D_MEM_block_num = parameter[1] / parameter[3];
I_CACHE_block_num = parameter[4] / parameter[5];
D_CACHE_block_num = parameter[7] / parameter[8];
I_PTE_entries = 1024 / parameter[2];
D_PTE_entries = 1024 / parameter[3];
I_TLB_entries = I_PTE_entries /4;
D_TLB_entries = D_PTE_entries /4;
```

Decode() & R-type , I-type , J-type——這邊基本上與 project 1 相同，只是在要去 disk 找資料的地方需先進 TLB 做查詢，開始計算 hit / miss。
I_TLBupdate 的呼叫在 decode 階段，將拿到的 PC (virtual address) 去查詢 I_TLB。
D_TLBupdate 的呼叫在 I_type 遇到 load / store 指令時，將 reg + imm 的值（也就是 virtual address）拿去查詢 D_TLB。

進入 I_TLBupdate，則按照上述（第三頁）的步驟進行 search & update，由於每個階段都有 LRUupdate，因此我將 LRU 獨立出一塊 function 來做，避免程式碼出現過多重複的地方。

LRUupdate(int* LRUorder , int entries , int init_value , int replace_new)——傳入的值為目前對哪一個空間的 LRU 更新，與區分 LRU 上下邊界，還有最新的那個 index 是誰。這個 function 的方法是，檢查原 LRU_order 裡面是否有 replace_new 的 index，若是有，就將從這個位子開始，後面的全部往前移一位，接著將最後一位的 index 值改為 replace_new，並且 return 這時候的位址，告訴原 function 是否有成功給值。

I_TLBupdate / D_TLBupdate ——檢查 Virtual address / pagesize 的 index 是否存在 TLB 裡面，再根據 hit or miss（第三頁的步驟）更新 TLB 的 tag , valid , LRU order，與對 CACHE , PTE update & search。若是 TLB miss，則要注意 PTE 回傳的值（被取代掉的 index）是否存在 TLB 中，若有，則將該 index 的 valid 更改為 0。

I_PTEupdate / D_PTEupdate ——檢查 Virtual address / pagesize 的 index valid 是否為 1，判斷 hit or miss。Hit 需更新 LRU order。Miss 的情況下，需更新 memory 裡面的 data，並重新設定 valid & LRU order。並檢查 memory 裡被取代掉的 page 是否存在 CACHE , PTE，若存在，需將那個 page or block 之 valid 改為零。注意 PTE update 會 return 被改過 memory 的 index 給 TLB，讓 TLB 更新。

I_CACHEupdate / D_CACHEupdate ——依 associativity 做區分，分成三種情況討論—fully associativity , direct map , n-ways associativity。首先檢查 physical address 的 tag 是否存在 CACHE 中，以此判斷 hit / miss，並更新 LRU order。Miss 則除了 LRU order 外，還需更新 data , valid 與 tag。

全部 table 更新完後，也執行完該 PC 指令，跳回 main()，回到 while loop 並印出該 cycle register 的值，接著繼續做下一個指令……，直到指令正常執行完，跳出 while loop，fclose 寫入的檔案，並做 fprintf report.rpt，整個程式結束。

Testcase design & elaboration

這次的 testcase 因為限制了 halt error的部分，加上若是同學都沒有改原本的 single cycle，就只能單純檢測他們的 hit / miss 數量是否有錯，因此我使用了前兩次的測資加上更多的 load / store 部分，並用 branch 指令增加 cycle 數看同學是否真的有做好 tlb 等結構設計，以此來設計我的 testcase。

1. D-Memory access

```
sb $2, -1($1)              # store to 1023
lw $4, 1020($0)    # $4 = 0x00000033
sh $2, -4($1)              # store to 1020
lw $5, 1020($0)            # $5 = 0xD4330033
sb $2, -2($1)              # store to 1022
lw $3, 1020($0)            # $03 = 0xD4333333
sh $2, -8($1)              # store to 1016
lw $7, 1016($0)            # $07 = 0xD4330000
sw $2, -8($1)              # store to 1016
lw $6, 1016($0)            # $06 = 0xFFFFD433
sb $2, -10($1)             # store to 1014
lw $9, -12($1)             # $9 = 0x00003300
sh $2, 1014($0)            # store to 1014
lw $8, -12($1)             # $8 = 0x0000D433
lbu $14, 1023($0)          # $14 = 0x00000033
addi $14, $14, 0x80      # $14 = 0x000000B3
lb $15, 1023($0)           # $15 = 0xFFFFFFB3
```

首先先在一開始的部分檢測 load /store 檢測，這時候會開始做 D tlb/pte/cache 的運算，這邊的測資會檢測邊界值的情況，並同時檢查若是 virtual address 並非四的倍數，傳進去會不會壞掉，運用多個 load / store 檢查 CMP 設計的完整性。

2. Jump and loop

```
        j Back
Back:
        and $10, $10, $11          #
        sll $11, $11, 2      #
        bne $10, $0, Back          # Back = -3
        # nor & bne & sra
```

```
        addi $10, $10, 7   #
        lui $11, -0x0fff     #
        jal Try
Try:
        nor $12, $10, $11         #
        sra $11, $11, 1     #
        bne $12, $0, Try   # Try = -3
```

這是檢查 bne 與 PC 計算是否正確，同時檢查 I instruction 跳來跳去實惠不會出錯，並增加 cycle 數量。

3. JR 跳躍測資

```
        addi $5, $30, 0xffff    #number overflow PC 15C
        addi $18, $18, 16    #PC = 160
        lw $2, 0($0)       #PC =164
        jr $18                 # PC=168
        add $0, $2, $2              # pc16C
        addi $18, $18, 12
        jr $18
```

這邊是檢測當 jr j jal 等指令不斷出現時 I instruction 是否會錯，因為平常的 I instruction 都是照著順序作，因此就算沒寫好可能也不會發現，這邊就是讓 I instruction 如同 D 一樣跳來跳去，檢測 CMP 是否會在 cycle 數量增加的時候出錯。

6. loads instruction!

```
lw          $1, 244($0)
        lw          $1, 168($0)
        lw          $1, 88($0)
        lw          $1, 152($0)
        lw          $1, 148($0)
        lw          $1, 80($0)
        lw          $1, 160($0)
        lw          $1, 100($0)
        lw          $1, 208($0)
        lw          $1, 48($0)
        lw          $1, 164($0)
         lw          $1, 8($0)
```

```
        lw      $1, 92($0)
        lw      $1, 28($0)
        lw      $1, 92($0)
        lw      $1, 24($0)
        lw      $1, 144($0)
        lw      $1, 200($0)
        lw      $1, 116($0)
        lw      $1, 100($0)
        lw      $1, 32($0)
        lw      $1, 136($0)
        lw      $1, 60($0)
        lw      $1, 80($0)
        lw      $1, 200($0)
        lw      $1, 8($0)
    j fin
fin:
        lh      $0,    0($29)          #two error - - Write $0 error and overflow
```

最後又再次跑過所有 load 指令。首先要先對所有 TLB PTE 更新再次檢查，這邊特別的是，跑了非常多的 load 之後，若是沒有將 PTE miss 對 TLB 更新放在前面，就會出現錯誤，也就是 TLB 更新順序的問題。需要很仔細地檢查 project3 才會過。我個人也是被這比這筆測爆，因此加入 testcase 中來測同學。

這邊比較心機的部分是對 write 0 做偵測，因為這次的 project3 可以說是沒有 error detect，所以 write 0 的同時依然要進 D tlb 查找，如果同學忘了將這部分加進原本的 single cycle 就會出錯，同時對所有指令的 write 0 都跑過一變，看同學對 CMP 設計的細心度。最後以一個 write 0 and number overflow 的測資為結尾。

測資總結：

因為助教限制了 D 的 error，整體來說這次測資的設計彈性變小，因為少了印出每個 cycle 的資料，加上可以用原本的 single cycle 來印 snapshot.rpt，要測爆別人也變得有點困難，如果同學都有把 CMP 設計好，基本上這次的測資不太會有什麼問題。

當然前提是助教給的 10 個變數測資不會太刁難，畢竟若是助教給了無法整除的 10 個變數….那我想除了助教之外八成的同學應該都會直接爆掉。結論是，

這次的 project3 雖然設計 CMP 的部分十分繁雜，但因為助教在題目上給了很多限制，因此只要好好跟著 tutorial 做並細心寫完，就不太會有問題。

Repo Conclusion :

這次的 project3，對我而言最困難的部分，就是看懂題目要求並實作 CMP 結構，雖然老師上課都有教過，可是因為這部分算是講得滿快的，加上老師在上這一段的時候都在弄 pipeline 的東西，因此說真的沒有學得很懂。反而是後來寫 project3 的時候要強迫自己弄懂全部，但因 data 欄位不需要真的實作，加上 write back 也不用，所以簡化很多。甚至與同學討論過後，得到結論是連 mem 都可以用 PPN 來代替就好，不用真的做出 mem。其實我的 mem 因為裡面並沒有存 data，所以也只是個給 PTE 來只位置的空間。

總體而言，這次的 project3 與前兩次不同，彈性很大，大家寫的方法也都不盡相同，這時候與同學討論不僅可以更了解 CMP 結構，也能學到其他人的想法與做法。但因只要印出 report.rpt，在 debug 的時候會變得十分痛苦，我的做法是將所有結構印出來，並一個一個對照看，非常花時間，但除了手算之外也只能這樣做。

共三次的 project，從中我學到非常多東西，設計 testcase 與 simulator 就不用說了。也發現這三次的 project 其實都有關聯性，並了解到當大家想法不同的時候，要如何說明自己的想法並聽懂別人的說明，甚至是教會其他同學。也因此整份 code 的架構設計十分重要。