Report for Architecture project 3

Group: archi01 Members:101021120 鄭宇翔, x1022108 曹士杰

Simulator Design

1. Data structure

我使用 C 語言實作本次 project，除了第一次 project 所使用的 data structure 外，我還使用了以下 struct:

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

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

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

static struct Translation_Lookaside_Buffer{
    int *content;
    int *valid;
    int *tag;
    int *LRUorder;
}I_TLB, D_TLB;
```

來做為各個 cache, TLB, PTE, memory(包括 instruction 與 data)。

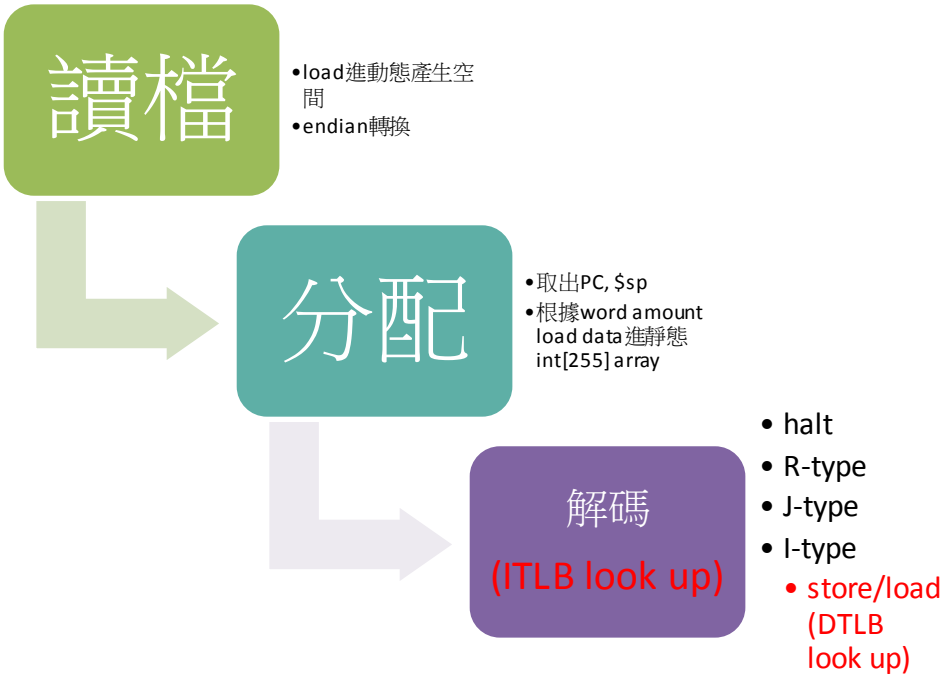
由於這次 project 需要以上結構的大小為 configurable，因此我用指

標再用 malloc 配置空間大小(下一節詳述)。

並且,由於本次的 replacement policy 為 LRU(least recently used) replacement,因此我在 cache, TLB, memory 裡都有 LRUorder 欄位,內存的是 block (or page)的 index,程式內會把剛用到的那個 block (or page) index 搬到後面去,因此需要 LRU replacement 時,LRUorder[0]會是優先須被替換的 block (or page) index。

實際上,data 欄位的有無並不會影響本次 project 的結果,因為本次 project 只需 count 出 cache/PTE/TLB 的 hit/miss 數,你的 data 還是可以只與 hard drive(iimage/dimage)做存取。

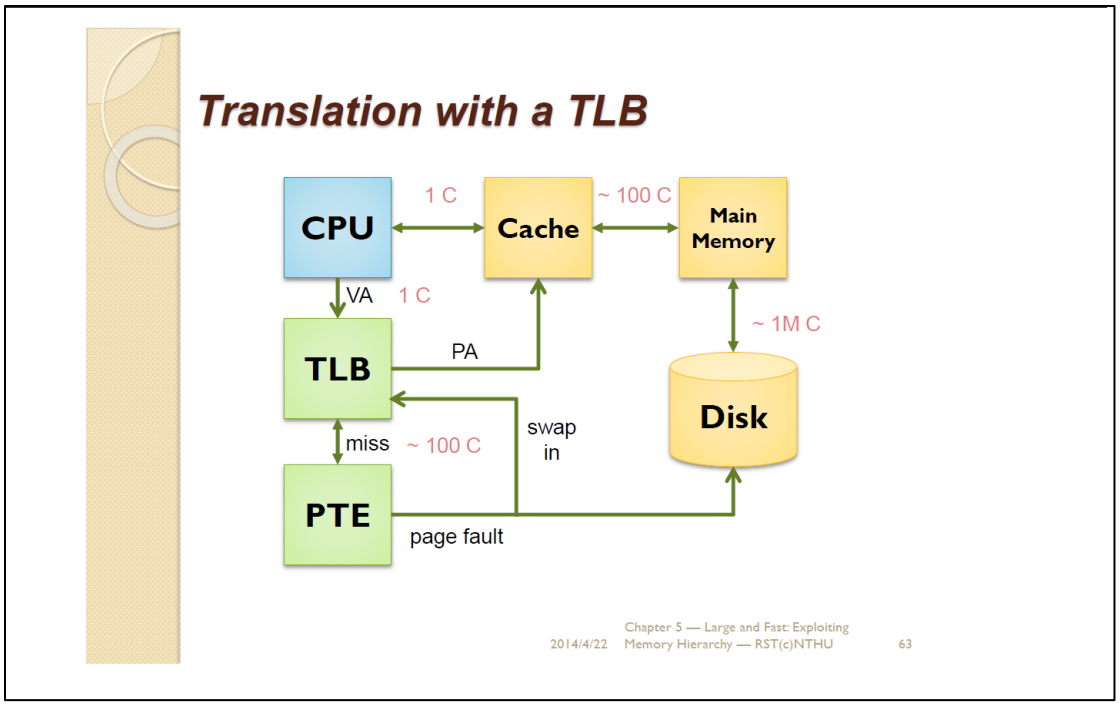
2. Program structure



以上是第三次 project 之流程圖。

結構上與第一次 project(single cycle)是一樣的。而與第一次 project 不一樣的地方是紅色部分,PC 與 store or load 指令的 base + immediate 都是 virtual 的 address,要透過查詢 TLB 的方式做 virtual 與 physical 間的轉換。

下頁圖顯示了各結構之間的 relation:



因此概念上，我們的 CPU 拿到 virtual address(VA)之後，去查詢 TLB，然後分兩種情形討論：

1. TLB hit(查 valid == 1, 並且 tag == VA 所在 PTE page 的 index): 代表資料在 memory 裡面，因此 TLB content 放的是 physical address(PA, 我定為所在的 memory byte index)，先做 LRU 紀錄，再拿這個 PA 去做 Cache look up。
2. TLB miss(上述反之)，去查 PTE：
   (1) PTE hit(查 index = address/pagesize 的 valid == 1): 代表資料在 memory 裡面但 TLB 未紀錄，因此先將 PTE 這部份放到 TLB(需做 LRU 紀錄，可能要 LRU replacement)，且 PTE content 放的是 physical address(PA, 我定為所在的 memory byte index)，接著再拿這個 PA 去做 Cache look up。
   (2) PTE miss (page fault)(查 index = address/pagesize 的 valid == 0): 代表資料在 hard drive (iimage/dimage)裡面，因此我們要把資料搬到 memory(需做 LRU 紀錄，可能要 LRU replacement)並且記錄 PA，且 valid 改為 1，並將 PTE 這部份放到 TLB(需做 LRU 紀錄，可能要 LRU replacement)，接著再拿這個 PA 去做 Cache look up。

注意當 memory 做 LRU replacement(置換 page)時，舊有的 page 已不存在於 memory 中，因此此時的 PTE 與 TLB 有記錄到的這個 page 的 valid 須改為 0(意義是已不在 memory 中，而是在 hard drive 裡)，且 cache 有碰到該 page 的 block 的 valid 也須改為 0(意義是這些 block 裡的 data 已

無意義，因為被換過了)。

接著查詢 Cache 的部分也分成兩種：

1. Cache hit(查 valid == 1, 並且 tag == PA): 代表 data 在 cache 裡，依照 associativity 去做 LRU 紀錄。(associativity == 1 為 direct map，沒有 LRU，若 associativity == block 數目為 full associativity，其餘為 n-ways)
2. Cache miss(上述反之)：代表 data 不在 cache 裡而是在 memory 裡，接著 index 從小到大看是否有 valid == 0 的 index。若有，valid 改成 1，tag 紀錄 data 在 memory 的 block index，LRU 紀錄；若無，LRU replacement。

以上為查詢的 SOP，不論為 instruction or load/store address，動作都是如此，只是 data 搬動方向不同。

並且，本次 project 雖然要求我們使用 write back/allocate policy(只存取這一層，需用到下一層時再存取下一層)，但實際上因為沒有 L2 cache(or upper level)也沒有要記錄 memory 的 hit/miss，因此實際上用 write through policy(一次一致全階層存取)實作的結果會是相同的，但若需顯示 memory 的 hit/miss 可能用此方式結果就會有差異。

以下細述函式內容：

main():

大致上與 project1 相同，且因為本次 project 不允許測資有任何 error ，因此我將 error handler 所需印在的 error_dump.rpt 拿掉改成 stderr (印在螢幕上)。

在 while 迴圈前 call AllocateMemory(argc ,argv) ，將手動輸入的參數送進 AllocateMemory() 函式，並在裡面做空間大小的配置與初始化。(包括 cache, TLB, PTE, memory)

並且在 while 迴圈結束後 call PrintReport() 函式將所有(I/D; cache, TLB, PTE)的 hit/miss 總數印到 report.rpt 裡。

AllocateMemory():

先預設一個

```
/* IMEM-size, DMEM-size, I-pagesize, D-pagesize,
ICACHE-size, I-blocksize, I-associativity, DCACHE-size,
D-blocksize, D-associativity*/
ArgArray[10] = {64, 32, 8, 16, 16, 4, 4, 16, 4, 1}
```

做為預設的空間大小與 associativity 配置。

再判斷 argc 之數目，若為 1(./CMP)則使用 ArgArray 的內容做 calloc(equiv. to malloc + memset to 0)；若為 11(./CMP + 10 個參數)則將 ArgArray 改成手動輸入之內容並做配置；若為其餘數字螢幕上顯示 error。

```
PTE entries 數目為 1024/pagesize。
TLB entries 數目為 PTE entries 數/4。
CACHE blocks 數目為 CACHE 總大小/ blocksize。
MEM blocks 數目為 MEM 總大小/ pagesize。
```

接著，依照以上資訊做 calloc。

注意！即便欄位數一樣，不可以做以下動作：

```
D_MEM.valid = D_MEM.LRUorder = (int*) calloc(DMEM_blocks, sizeof(int));
```

這樣會造成 valid 與 LRUorder 欄位指向同一塊空間，改 valid 等於改 LRUorder，這是錯的，要分開配置指向不同空間。

最後值得一提的是我將 tag 與 content 欄位預設內容為-1，以免與地址 0 搞混。

decode()--R_type()/I_type()/J_type()

大致上與 project1 相同，唯有在一開始 call ITLBlookup(PC) 將拿到的 PC 去查詢 ITLB。

並且 **I_type()**裡面所有與 DISC 做存取的 8 個 store/load 相關指令(sw, sh, sb, lw, lh, lhu, lb, lbu)在存取 data 前，call DTLBlookup(R[rs] + imm) 將拿到的 R[rs] + imm 去查詢 DTLB。

ITLBlookup()/DTLBlookup():

兩函式內容相同，唯所有 I 與 D 的不同，程式碼內容一步步實現第 3 頁之查詢 SOP，有許多地方都是在做 LRU 的 reorder，這部分寫起來蠻冗的，但我也沒有把它放到另一個小函式，幾乎都只是查詢上下界與空間之不同，程式碼式一樣的，我是做 copy & paste，目前沒有一個好的想法將其分開寫成獨立的 LRUreorder()函式。

IPTElookup()/DPTElookup():

同上，但這個函式有回傳值，回傳 PTE 因為 memory 之 LRU replacement 所置換掉的舊 page 的 PTE entry。給 ITLBlookup()/DTLBlookup()做查詢是否自己的某個 entry 的 valid 也需改成 0。

ICACHElooko ()/DCACHElookup():

同上，只是 associativity 不同會造成上下界與 LRU 有無的不同，因此我分成 direct map, fully associative, n-ways 三種來寫。direct map 無 LRU，無需做 LRU reorder/replace；而 n-ways 與 fully associative 在於迴圈上下界之不同。

Conclusion:

這次 project 我是照著 SOP 一步步實現，中間的邏輯看似複雜其實簡單，但程式碼因沒經過較好的分析、簡化與獨立，導致重複的部分相當多，只有迴圈上下界或'D'與'I'的不同。因此行數看起來相當多(約 1300 行)，但實際上只有約 200 ~ 300 行是要仔細想過，其餘複製貼上，將 D 改成 I。這是因為我目前沒有想到一個好的方式簡化它，才會導致冗餘的部分過多，而且也較難維護，我想這是我必須要努力改進的部分。

Testcase design

Preface

Testcase 設計考慮到主要是測試 load 及 store 指令時的 hit 和 miss 數量，並且不允許出現 error 的指令，因此我們組 testcase 不考慮 data 的具體數值，而是盡量多使 testcase 進行不同情況下的 load 和 store 操作。

同時我們提前預知每一個 testcase 對於各個 cache、TLB、PTE 是 direct map 或者 n-way associative 以及 block 和 page 的 size。因此在 testcase 設計是只能粗略的、預測性的進行 hit 和 miss 的測試。

Testcase for project3

具體設計如下：

一、連續的 load、store 指令，測試 data cache、TLB、PTE 的 hit miss 是否正確。並且講 data 的 address 距離隨機進行設置，這樣的目的是盡可能的測試到 hit 和 miss 的不同情況。

```
lw $1, 0($0)
lb $1, 4($0)
sw $1, 40($0)
lh $1, 8($0)
lw $1, 40($0)
lw $2, 48($0)
lh $1, 50($0)
lw $1, 0($0)
```

二、進行 j、beq 等指令使 instruction 進行不斷地跳轉，並且對跳轉的距離進行提前設置，既有近距離跳轉，也有遠距離跳轉，這樣的目的也是盡可能的測試到 hit 和 miss 的不同情況。

```
    lw $1, 0($0)
    j l1
l1:
    sw $1, 0($0) # 近距離跳轉，只跳轉的下一條指令，
                 # 其目的是測試 instruction hit 的情況。

    addi $2, $0, 20
l6:
    addi $1, $1, 1
    lw $3, 0($0)
    lb $3, 4($0)
    lh $3, 8($0)
    sw $1, 40($0)
    sw $1, 4($0)
    lw $3, 40($0)
    lw $4, 48($0)
    lh $3, 50($0)
    lw $5, 0($0)
    sw $3, 0($0)
    sw $5, 4($0)
    sw $4, 40($0)
    sw $3, 100($0)
    lw $3, 0($0)
    lw $5, 0($0)
    bne $1, $2, l6   #l6=-17，遠距離跳轉
                     #其目的在于盡可能的測試到
                     #instruction miss 的情況，
                     #同時在這期間加入我們第一條對於 data
                     #的 hit/miss 測試。
```

三、為測試其他組的 simulator 的穩定性，我們用迴圈的形式不斷執行類似指令以測試其 simulator 在指令過多時是否會出錯。

```
    addi $2, $0, 20
l5:
    addi $1, $1, 1
    bne $1, $2, l5   #l5=-2，第二條也有對這一項進行測試
```

Conclusion:

整體而言，這次 project 通常會錯只是錯在 hit/miss 算錯，若 data 內容只從 disc 存取，則 snapshot 會與 project1 相同，應該已經不會再有錯誤，除非是第一次 project 沒改正確或真的從 cache 存取 data 才有可能弄錯。並且因為不可有 error，因此 testcase 的彈性會大幅下降，只能測別人的 hit/miss 是否有算錯。