**計算機結構程式專題一報告**

學校: 國立臺南大學

學生: 蔡易霖

**如何執行**

此專案使用Java程式語言撰寫，提供三種執行方式:

1. 使用 jacac 編譯並執行
2. 使用包裝檔 jar 執行
3. 使用 .exe 執行

**使用 jacac 編譯並執行**

在 cacheAssociative.java 檔案目錄下執行以下指令來編譯 .java 檔案。

#編譯

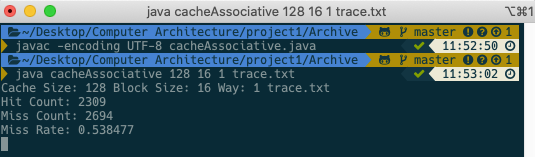
javac -encoding UTF-8 cacheAssociative.java

執行指令時給予參數，順序為 cache\_sizeCache的大小其單位為KByte、block\_size每個Cache Block的大小其單位為Byte、set\_degree一個set中的Cache Block個數、fileName檔名。

#執行

java cacheAssociative 128 16 1 trace.txt

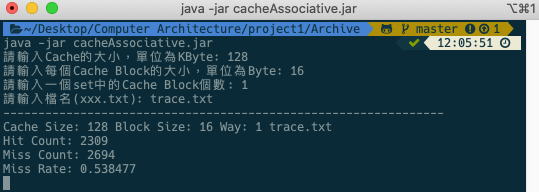
trace.txt 需與編譯出來的程式放置於同一個資料夾位置



**使用包裝檔 jar 執行**

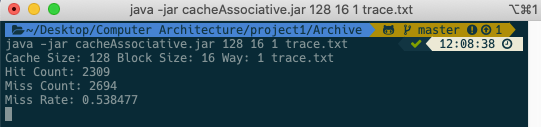
程式碼已經預先打包成 jar 檔案，可以直接執行指令，並依照提示給予參數。

java -jar cacheAssociative.jar



再者可以直接在執行指令時給予參數，順序一樣為cache\_size、block\_size、set\_degree、fileName。

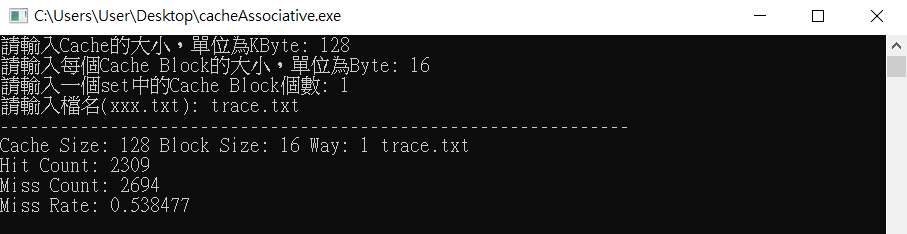
java -jar cacheAssociative.jar 128 16 1 trace.txt



trace.txt 需與 .jar 檔放置於同一個資料夾位置

**使用 .exe 執行**

在 windows 環境下打開執行檔，依據提示輸入指定參數。



trace.txt 需與 .exe 檔放置於同一個資料夾位置

## 程式架構與程式碼解說

### 自定義 Class

此專案定義了三種 Class 類別，分別為 Address、Block 與 CacheSet。



### 讀檔 Function

### 主程式 main()

程式進入點，流程為讀入參數→Cach大小轉換→計算set的個數→初始化 setArray→讀檔取得記憶體位置→計算Miss Rate。

計算Miss Rate的方式首先要得到Miss次數，為了方便計算在處理記憶體時已經先將16進位的位置轉換成10進位。接著再來來取得此記憶體是位於第幾個 Memory Block (position)。接著再將 position % setSize 取餘數就能得到該記憶體是位於哪一個 set。如果該 set 中的 valid 為 0 代表目前 Block 為空，因此就將此記憶體放入指定的位置(victim)並且更新orderList的順序，並將 missCount+1。若當同一個set中的所有 Cache Block 都有資料(即valid=1)且 Cache Block 的所有 tag 與從處理的記憶體位置中所獲得的 tag 不相等的話，代表 miss 並將 missCount+1。取代的機制為 LRU(Least Recently Used)，也就是將存在於 Cache 中最久沒用到的 Block 給取代掉，實作方式是使用 ArrayList 用來儲存過去每個 Cache Set 中的 Block 使用的順序。變數 orderList 為最近使用的排序，數字愈小表示愈久之前使用過，也就是將要被取代掉的候選位置。

### 

## 測試結果

|  |  |  |  |
| --- | --- | --- | --- |
| Cache Size (Byte) | Block Size (Byte) | n-way | Miss Rate |
| 128K | 16 | 1 | 0.538477 |
| 2 | 0.534879 |
| 4 | 0.533680 |
| 8 | 0.533280 |
| 256K | 16 | 1 | 0.535679 |
| 2 | 0.534479 |
| 4 | 0.533280 |
| 8 | 0.533280 |
| 512K | 16 | 1 | 0.534479 |
| 2 | 0.533480 |
| 4 | 0.533280 |
| 8 | 0.533280 |
| 1024K | 16 | 1 | 0.533880 |
| 2 | 0.533280 |
| 4 | 0.533280 |
| 8 | 0.533280 |
| 512K | 8 | 1 | 0.857885 |
| 2 | 0.856286 |
| 4 | 0.856086 |
| 8 | 0.856086 |
| 512K | 16 | 1 | 0.534479 |
| 2 | 0.533480 |
| 4 | 0.533280 |
| 8 | 0.533280 |
| 512K | 32 | 1 | 0.374775 |
| 2 | 0.373976 |
| 4 | 0.373776 |
| 8 | 0.373776 |
| 512K | 64 | 1 | 0.299220 |
| 2 | 0.297821 |
| 4 | 0.297621 |
| 8 | 0.297621 |
| 1024K | 8 | 1 | 0.857086 |
| 2 | 0.856086 |
| 4 | 0.856086 |
| 8 | 0.856086 |
| 1024K | 16 | 1 | 0.533880 |
| 2 | 0.533280 |
| 4 | 0.533280 |
| 8 | 0.533280 |
| 1024K | 32 | 1 | 0.374175 |
| 2 | 0.373776 |
| 4 | 0.373776 |
| 8 | 0.373776 |
| 1024K | 64 | 1 | 0.298021 |
| 2 | 0.297621 |
| 4 | 0.297621 |
| 8 | 0.297621 |

從上表中可以察覺到在同樣的cache size下，如果提升block size有逐漸降低miss rate的趨勢，這是因為當block size變大相對提升了空間區域性(Spatial Locality)，如果一個記憶體剛被使用過他附近的記憶體位置也很有可能被使使用到。但如果不斷的提高block size，反而會導致cache內的總block數太少導致miss rate提升。此外我們也能發現相同block size在不同的cache size狀況下其miss rate變化不大。而且n-way提高miss rate也有降低的趨勢。

## 心得

從這次專案實作中可以很清楚的了解三種不同cache的配置方式，分別有Direct Mapped、Fully Associative、N-Way Associative。在實作前還先拿紙筆親自手算一遍並了解每一個步驟流程，有助於對程式化的思路更清晰。實作中面對到的第一個困難是如何處理要被取代的Block，取代的機制為LRU。一開始的想是將每個Block都給予一個計數器count每次要抉擇哪一個將被取代掉的時候就找count最少的，不過此方法最後測出來有Bug會有例外狀況。最後採取的方法是將每一個Cache Block都賦予一個 orderList，這一個序列清楚地記錄著下一個將被取代的順序0~n，數字越小代表取代優先權越高。透過後者方法可以完整地模擬LRU的機制。第二個遇到的問題是，當初沒有詳細看到專案說明cache\_size的單位是Kb，所以老師在群組Po的測試範例答案與我的程式結果不同。最後才發現原來我忘了將Kb轉換成Byte。