**Computer Organization**

**0716214 江岳勳**

**0716222 黃偉傑**

**Detailed description of the implementation:**

* **direct\_mapped\_cache**

程式在啟動的時候會先將LRU.txt中的資料先讀取進來，然後按照各種不同的cache size/associativity組合來執行測試。

1. 用 ifstream 讀檔案，經過 std::hex() 之後輸入會從字串被轉成整數，再用 vector<unsigned int> 作為 container 儲存
2. "memory\_access()" 這個 function 可以輸入 cache type, cache size, and block size ，所以在 main function 裡面可以一次運行所有 required combinations
3. 在"memory\_access()" 這個 function裡面，用一個 defined structure 的 vector 來模擬 direct mapped 中的 table，每一筆資料記錄 valid bit, tag, and index，並且以預先計算好的 number of blocks 來設定大小。提出每一筆輸入時，先把offset去掉，再比對 index，若相符的資料為 invalid (即 compulsory miss )的情形，就直接更新資料進去並且設為 valid；如果是 valid 的資料，就比對 tag 來確定有沒有 miss。 Miss handling 在這裡因為只有儲存 memory address，所以只做了更新tag
4. 每做完一種(cache type, cache size, block size)的組合後，會呼叫 print function，把記錄的hit, miss次數算出需求並輸出

* **set\_assoiciative\_cache**

程式在啟動的時候會先將LRU.txt中的資料先讀取進來，然後按照各種不同的cache size/associativity組合來執行測試。

測試的一開始會先根據cache size和associativity算出block和set的數量，使用一個vector<vector<unsigned int>>來模擬set associative cache，內層的vector代表一個一個的set，新touch過的位址會被push到尾端，這會讓LRU永遠在vector的頭部。接著便從預先讀好的data中拿記憶體位址來做處理。

讀出來的記憶體位址要先除以block size，轉換成block address，然後再用set\_count對block address分別取餘和商，得到該block address的set編號及tag。接下來分為三種情況(從上至下if、else if、else)：

1. cache[set\_idx]包含tag，這樣的話代表該block已經在cache中了，將tag從vector erase掉，重新push\_back到尾端。**(HIT + 1)**
2. cache[set\_idx]大小未滿associativity，代表該block不在cache中，且該set仍有invalid的空白區域，直接將tag push\_back進去。**(MISS + 1)**
3. 以上皆非的話意味著要將LRU給替換成要寫入的tag，將vector的頭部元素erase掉，再將tag push\_back進去。**(MISS + 1)**

模擬完後只要將hit rate和miss rate算出輸出便結束了。

**Implementation results:**

* **direct\_mapped\_cache (Output Result 在報告最後面)**

以下分別是I-cache與D-cache下，miss rate 在不同 Cache Size 和 Block Size 組合下的數值：

* **I-cache**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 16B | 32B | 64B | 128B | 256B |
| 4K | 2.17% | 1.09% | 0.54% | 0.27% | 0.14% |
| 16K | 2.17% | 1.09% | 0.54% | 0.27% | 0.14% |
| 64K | 2.17% | 1.09% | 0.54% | 0.27% | 0.14% |
| 256K | 2.17% | 1.09% | 0.54% | 0.27% | 0.14% |

* **D-Cache**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 16B | 32B | 64B | 128B | 256B |
| 4K | 5.56% | 3.17% | 1.59% | 0.79% | 0.79% |
| 16K | 5.56% | 3.17% | 1.59% | 0.79% | 0.79% |
| 64K | 5.56% | 3.17% | 1.59% | 0.79% | 0.79% |
| 256K | 5.56% | 3.17% | 1.59% | 0.79% | 0.79% |

* **set\_assoiciative\_cache (Output Result 在報告最後面)**

以下是miss rate在不同cache size和associativity組合下的數值：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1K | 2K | 4K | 8K | 16K | 32K | 64K | 128K |
| 1-way | 11.07% | 8.28% | 5.47% | 4.03% | 3.16% | 2.54% | 2.34% | 2.33% |
| 2-way | 8.36% | 5.18% | 3.63% | 2.98% | 2.37% | 2.33% | 2.29% | 2.28% |
| 4-way | 7.78% | 4.19% | 3.07% | 2.67% | 2.34% | 2.28% | 2.28% | 2.28% |
| 8-way | 7.83% | 3.98% | 2.81% | 2.45% | 2.29% | 2.28% | 2.28% | 2.28% |

從圖表中可以發現，miss rate隨著cache size和associativity的變大而降低，而且有出現趨近一個定值的情況。其中比較有趣的是1K-8way的miss rate反而比1K-4way的miss rate高了一些，我推測應該是由於測資的量不夠大，剛好有一些巧合讓8way反而比4way的miss rate還大。

**Problems encountered and solutions:**

* **direct\_mapped\_cache**

一開始最花心思處理的是輸入，原本的做法是將輸入的Hex number 轉成 binary，然後再根據上課教的做法炮製，用Log2()算tag, index需要幾個digits，然後再用substring的函式去擷取，但是後來覺得自己造輪子實在太傻了，發現了std::hex()之後就輕易搞定了。

後來發生過一個bug是忘記做轉型(cast)，結果miss rate怎麼算都是0，後來加上cast就解決了。

* **set\_assoiciative\_cache**

在一開始讀檔的時候，我使用ifstream搭配std::hex來處理十六進位的資料，卻發現一直失敗，打開LRU.txt手動模擬了一下後才發現裡面有會讓signed int overflow的數字，將code內的int換成unsigned int後便解決了。

模擬set associative cache的部分則沒有什麼問題。

**Comment:**

**0716214 江岳勳：**

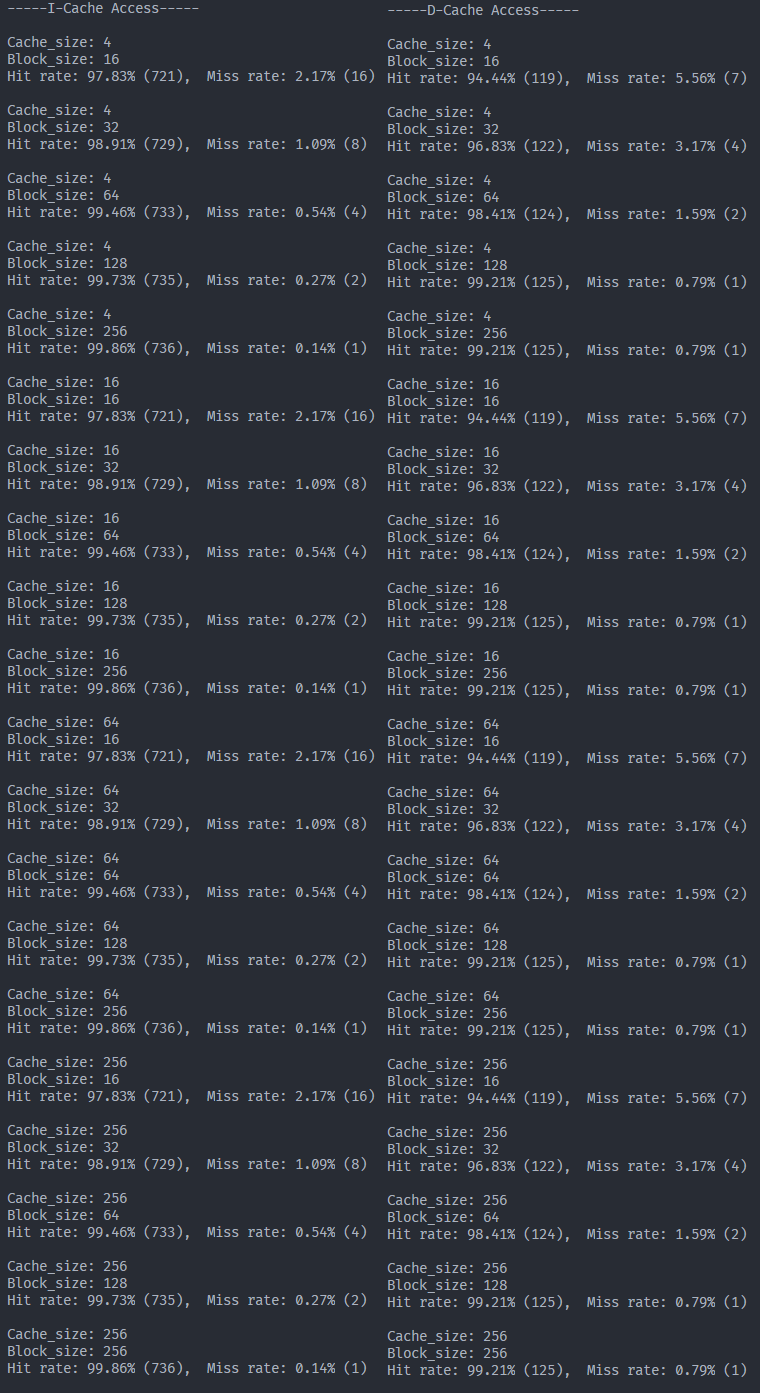
本次我負責set associative的部分，實作過程中剛好在複習了一次整個第五章有關cache的部分，期末考前在複習時就有發現我常常會把byte address、block address、tag、index之類的名詞搞混在一起，經過這次作業有確實感覺到自己整個搞懂這個部份了。

**0716222 黃偉傑：**

　　寫C++真的是比Verilog快樂多了……同樣的東西如果用Verilog來刻可能會很崩潰，感謝助教。不過這次作業帶給我最大的影響應該是實際實驗過Direct Mapped的Miss Rate有多麼差勁，如果存取都像給定的Data一樣一直讀小範圍的block的話，硬體加多大真的都沒什麼用。

**Implementation Result (Output Screenshots):**

* **direct\_mapped\_cache**



* **set\_associative\_cache**

