105-2 CA with Embedded DSD
Homework 4
Cache Behavior Model Design with C++

Announced at April 20, 2017

Deadline at 14:00pm, May 4, 2017

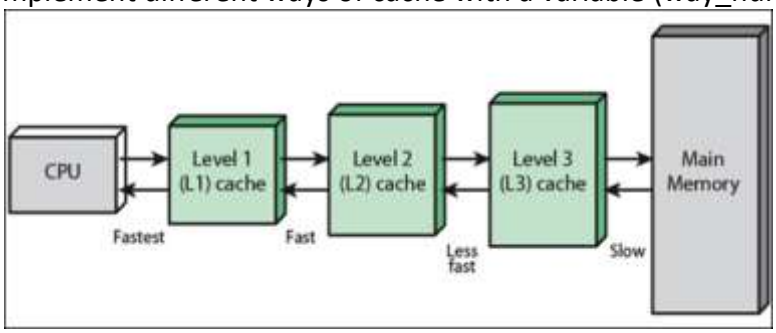
TA information
陳奕達，郭泓圻，電二 232 實驗室
edan@access.ee.ntu.edu.tw, charleykuo@access.ee.ntu.edu.tw

1. Problem Statement

Many system designs involve processors, memory, and peripherals which communicate with each other with system bus or other protocols. Usually, the clock rate of the processor is much higher than the memory. Therefore, when the processor accesses the memory, it will take lots of time for the processor to wait for the data. To conquer this problem, the cache is needed between the processor and the memory.

In this homework, you have to design a cache with C++ which matches the constraint below:

(1) Cache size is 8 blocks and each block has four words.
(2) Implement the cache with **write back** mechanism.
(3) The policy of cache-replacement is **LRU (Least Recently Used)** by using release bit in every cache line.
(4) Need to implement different ways of cache with a variable (way_number)



2. Data structure of given code

File name	Description	Modify
Makefile	For Linux system: (1) “make test” → generate “test” → “./test” to execute (for test.cpp) (2) “make quick” → generate “quick” → “./quick” to execute (for quicksort.cpp) (3) “make merge” → generate “merge” → “./merge” to execute (for mergesort.cpp)	No
test.cpp	This is the simple test bench, which checks the functionality of your cache and memory. If you pass the	No

	test bench, it will print “Well done!! You have passed test bench 1” on the screen.	
quicksort.cpp	This test bench is a simple quicksort program which uses the cache and memory to access the data. This program will first directly write the memory with a list of numbers. After finishing writing data into memory, the program will access the data from cache, sort it with quicksort algorithm, and save it back to cache. Finally, the test bench will check the data. If there is no error, the program will print “Well done!! You have passed test bench 2” on the screen.	No
mergesort.cpp	This test bench is a simple mergesort program which uses the cache and memory to access the data. This program will first directly write the memory with a list of numbers. After finishing writing data into memory, the program will access the data from cache, sort it with mergesort algorithm, and save it back to cache. Finally, the test bench will check the data. If there is no error, the program will print “Well done!! You have passed test bench 3” on the screen.	No
Mem.h	Header file for Mem.cpp	No
L1cache.h	Header file for L1cache.cpp	No
Mem.cpp	Memory code, see Section 3 for details	No
L1cache.cpp	Cache code, see Section 4 for details	**Yes**

3. Memory code description

In this project, you DO NOT needed to modify the mem.cpp file, but need to call the function of mem.cpp. Every variable and function will be described below:

Name (Input)	Type (Output)	Description
MEMsize	predefined (int)	Size of memory, each block has four words.
mem[MEMsize][4]	private variable (int)	Memory unit in mem.cpp. Save data in this two-dimension matrix
Mem ()	constructor	You need to initialize the value of the memory.
getfromMem (int)	public function (int*)	input a four-word address and return a pointer to 4-word array.
writetoMem (int, int*)	public function (void)	input a four-word address and an integer data pointer, and write the data into the memory. You need to check the value of

		the address, which must be between 0 and 255.

4. Cache code description

In this project, you needed to **modify the L1cache.cpp** file. Every variable and function will be described below:

Name (Func. input)	Type (Func. return)	Description
L1size	predefined (int)	Size of cache, each block has four words.
L1readmiss	private variable (int)	counters of cache read miss.
L1readhit	private variable (int)	counters of cache read hit.
L1writemiss	private variable (int)	counters of cache write miss.
L1writehit	private variable (int)	counters of cache write hit.
cache[L1size][8]	private variable (int)	Cache unit in cache.cpp. In every cache: cache[*][0]: valid bit cache[*][1]: dirty bit cache[*][2]: release bit cache[*][3]: tag cache[*][4]- cache[*] [7]: a block of data (4 data) from memory.
mem	private variable (Mem*)	pointer to the memory.
way_number	private variable (int)	the integer to decide whether the cache is direct-mapped, 2-way, 4-way, or fully-associative. direct map for 1 / 2-way for 2 / 4-way for 4 / fully associative for 8
L1cache (Mem*,int)	constructor	constructor, initial cache and the hit/miss rate.
getfromCache (int)	public function (int)	input a integer address, return a pointer which points to the data.
writetoCache(int, int)	public function (void)	input a integer address and a pointer to the data which you want to write into cache, return nothing.
getReadHit(void)	public function (int)	return L1readhit
getReadMiss(void)	public function (int)	return L1readmiss
getWriteHit(void)	public function (int)	return L1writehit
getwriteMiss(void)	public function (int)	return L1writemiss
getHit(void)	public function (int)	return L1readhit+ L1writehit
getMiss(void)	public function (int)	return L1readmiss+ L1writemiss

TODO: **getfromCache (int)** and **writetoCache(const int, const int)**

int getfromCache (const int address)

1. Each address point to a data in memory (4 data/memory block, so 4 address for a block)
2. getfromCache return the data corresponding to **address**
3. Increase L1readhit / L1readmiss when read hit / miss occurs

void writetoCache(const int address, const int indata)

1. Replace the data corresponding to **address** in memory by **indata**
2. Need to use **write back** method
3. Increase L1writehit / L1writemiss when write hit / miss occurs

Note:

1. The behavior of cache should be **write back**
2. The policy of cache-replacement should be **LRU (Least Recently Used)**
3. Should support different way_number：
   direct map for 1 / 2-way for 2 / 4-way for 4 / fully associative for 8
   way_number is determined in constructor, modify "int way_number" in **quicksort.cpp/ mergesort.cpp** to test the correctness of your cache
4. The reported hit/miss rate of **quicksort.cpp/ mergesort.cpp** should be correct to get the full credit.

TABLE I: Hit rate of quicksort.cpp

	Direct map (way_number=1)	2-way (way_number=2)	4-way (way_number=4)	Fully associative (way_number=8)
Hit rate	88.99 %	91.19 %	91 %	90.33 %

TABLE II: Hit rate of mergesort.cpp

	Direct map (way_number=1)	2-way (way_number=2)	4-way (way_number=4)	Fully associative (way_number=8)
Hit rate	80.5 %	84.99 %	86.98 %	85.25 %

5. Interface:

This chapter describe some special issues about the interface between processor (testbench) and cache and the interface between cache and memory.

(1) Processor and cache: While reading data, the processor gives the cache the word address(0~1023) and gets a data from the cache. It is the same for writing data.

(2) Cache and memory: While reading data, the cache gives the memory the 4-word address(0~255) and gets a 4-word data from the memory. It is the same for writing data.

6. Homework requirements:
   (1) C++ code: please hand in the "**L1cache.cpp**".
   (2) Report: In this report, you have to answer the questions below:
   Q1: Please draw the flow chart to describe the read/write behavior your cache. Start from address sent to cache, and end in return data. Vice versa.

   Q2: Please modify the test bench, and compare the hit rate and miss rate of different structure of cache (direct-mapped, 2-way, 4-way, or fully-associative).
   Q2-1: Build a table of hit/miss as TABLE I/II for both quicksort and mergesort.
   Q2-2: Discuss why there's difference of hit rate between different structures?
   Q2-3: Is hit rate keeps going up as the way_number goes up, why or why not?

   Q3: Please describe the difference between quicksort algorithm and merge sort algorithm.
   Q3-1: What the computation complexity of quicksort and merge sort algorithm?
   Q3-2: Which algorithm has the higher hit rate? Why?

   (3) Grading policy: Note that the report is as important as the code, so it is suggested that the code should be finished as soon as possible to reserve time for a qualified report. Here are the rules:
      a. A functionality correct code get 30%.
      b. A hit/miss rate correct code get 30%.
      c. Report for 40%.

7. Submission requirement
   (1) All the files need to be compressed as a single ZIP or RAR file. Send this file to TA via FTP:
   Address : **140.112.20.128** Port : **1232**
   Account : **CA_STUDENT** Password : **ca2017**
   Example of filename
   CA_HW4_b03943135.zip
   CA_HW4_b03943135_v2.zip

   Your submitted file should include the following files:
   L1cache.cpp
   Report_ HW4_b03943135.pdf

   (2) The homework will be graded ONLY IF the filename of your submission are correct!