Objective:
With this assignment you will:
- learn SIM-CACHE which is the one of the benchmarking tool of SimpleScalar simulation environment.
- understand the memory hierarchy performance with memory hierarchy organization.

Background
- **Read** [http://www.simplescalar.com/docs/users_guide_v2.pdf](http://www.simplescalar.com/docs/users_guide_v2.pdf) about sim-cache. or type the following to seek help about sim-cache.
  - `./sim-cache -h`
  - Help can also be invoked just by typing simulator name without any arguments.

- With sim-cache, you don't specify the size of the cache directly. Instead you specify the number of lines in the caches, (SETS) the associativity, and the block size in the cache.
  - **Example:** The size of the cache is the product of these three numbers. Thus, 4-way set associative cache with 1024 lines of 16 bytes each is $4 \times 1024 \times 16 = 262144$ bytes (256 kilobytes).

- The cache to be configured is mentioned twice on the command line. (The phrase "dl1" will appear twice when configuring the L1 data cache.)
- The parameters to sim-cache frequently uses both "1" (the numeral "one") and "l" (a lower-case letter "L").
  - For example, to configure a machine with an 8KB, direct-mapped L1 data *cache* with 32 byte blocks, use this command:`-cache:dl1 dl1:256:32:1:l`
  - Notice that 256 blocks times 32 bytes per block equals 8192 bytes. (Cache size 8K bytes)

Notes:
Problem 5: Block size : you may choose blocks size 16 or 32

Problem 4: Replacement policy LRU

Unified L1 cache configuration, no L2 example:
- `-cache:il1 dl1 -cache:il2 none -cache:dl1 ul1:256:32:1:l -cache:dl2 none`

Separate Inst and data L1 cache with no L2 example:
Part-1 Cache Analysis

**Deliverable 1:**

- Use the same integer and floating-point benchmarks you used in HW1 according to the last digit of your student id.
- Assume a directed mapped cache.
  - Change number of lines in the cache as 1K, 2K, 4K, 8K, 16K, 32K, 64K, 128K each with Different block sizes: 8, 16, 32, 64.
  - Use Least-Recently-Used (LRU) replacement policy
  - Compare and plot the data cache miss rates.

**Sample Output:**

![Miss ratios for Different Cache Capacities vs Block Size](image)

**Deliverable 2**

**Cache associativity** Simulate *your benchmarks* with different set associativity configurations and use a single run of *sim-cache* to simulate the performance of the cache under the following conditions:

- 32, 64, 128, 256 and 512 sets
- 1-way, 2-way, 4-way, 8-way and 16-way associativity
- 16-byte cache lines (block size)
- Least-Recently-Used (LRU) replacement policy
- Compare and plot the data cache miss rates.
- Compare and plot the data cache miss rates.
**Sample Output**

![Graph: Miss ratios for Different Sets vs Associativity levels](image)

**Analysis based on Deliverable-1 and Deliverable-2**

Investigate the code in the sim-cache simulator and answer the following questions based on the above results.

Q1) What is the difference between the read_miss_rate and miss_rate in the cache statistics computed by sim-cache? What do these statistics mean for each cache in the hierarchy (i.e., L1 instruction cache, L1 data cache and L2 unified cache)?

Q2) What is the formula to estimate the average memory access time for data using the statistics provided by sim-cache? Assume that the L1 instruction miss rate is negligible and you are given Thit−L1Data, Thit−L2, Thit−Memory (the hit times in the L1 data cache, L2 and memory, respectively).

Q3) For a given number of sets, what effect does increasing associativity have on the miss ratio?

Q4) For a given associativity, what is the effect of increasing the number of sets?

Q5) For a given cache size, how does the miss ratio change when going from an associativity of one to two to four? Explain.

Q6) If you were to design an Instruction cache, limited to a total cache size of 4 Kbytes, which cache organization would you choose, based solely on performance? Justify your answer.

Q7) If you were to design a data cache, limited to a total cache size of 4 Kbytes, which cache organization would you choose, based solely on performance? Justify your answer.

**Deliverable 4: Unified v.s. Split cache** Simulate *your benchmarks with* both unified and split cache. Assume one level and 2-way set associative cache with total sizes of 4k, 8k, 16k, 32k, 64k, 128k and block size of 32. Compare and plot the cache miss rates for instruction and data memory access. Summarize your observations and conclusions.

**Deliverable 5: Block replacement policy** Simulate *your benchmarks with* three different block replacement policies. Assume one level data cache with total cache sizes as 4K and 32K, each of which has degree of set associativity as 1, 4, and 32. Compare and plot the data cache miss rates. Summarize your observations and conclusions.
Deliverable 6:
Compare the performance of the not-recently-used (NRU) replacement policy (described below) with a pure LRU replacement policy for a unified (data and instructions) second level cache. The SimpleScalar cache simulator already comes with an implementation of the LRU policy. You are required to implement the NRU policy inside sim-cache and answer the following questions:

Question 1: Explain how you verified that your implementation is correct.

Question 2: Provide the L2 miss rates and L2 read miss rates as reported by simple-scalar for your benchmarks when run with LRU and NRU using the configuration files provided with the simulator. Please include a table with the following heading:

<table>
<thead>
<tr>
<th>Bench</th>
<th>LRU MissRate</th>
<th>LRU ReadMissRate</th>
<th>NRU MissRate</th>
<th>NRU ReadMissRate</th>
</tr>
</thead>
</table>

Question 3: Why would a commercial processor implement an NRU cache replacement policy instead of a pure LRU one?

NRU:
The OpenSparc T1/T2 processors implement a not-recently-used (NRU) replacement policy for the unified second level cache. This section describes a (simplified) NRU policy that you are required to implement inside the SimpleScalar sim-cache simulator.

The NRU replacement policy is based on a used-bit scheme. Each cache line has a used bit associated with it. Initially, all bits are reset (i.e., all used bits in the cache are zero). The used bit is set each time a cache line is accessed or when initially fetched/filled from memory. If setting the used bit for the current line causes all used bits within the corresponding cache set to be set to one, all other bits are reset. This ensures that at any time there is at least one cache line within a set that has its used bit clear.

On a cache miss, a cache line with its used bit clear is chosen to be replaced. The L2 cache has a single rotating replacement pointer, which is the “starting point” to find the way to replace. On a miss, the L2 looks for the first line within the set with its used bit clear, starting with the way pointed at by the replacement pointer. The replacement pointer is then rotated forward one way. Using the same replacement pointer for all sets of the L2 cache makes the replacement be more random than if each set had its own replacement pointer. Initially, the replacement pointer points to the first way (i.e., way zero).

Coding Hints:
The files that you are most likely to work with are sim-cache.c, cache.c, cache.h. Please note that the code in the cache.[ch] files is also used in the timing simulator part of the SimpleScalar infrastructure. Since this assignment involves only the functional cache simulator, you can safely ignore any timing/latency parameters in these files.

sim-cache works similarly to the way sim-safe works in that it executes the program instruction by instruction. The main difference between the two is that sim-cache models the behavior of the cache hierarchy as the program executes. This is done by modeling cache accesses when
instructions are fetched for execution and when executing memory instructions (e.g., load, stores). To achieve this, a cache access is simulated in the main loop of the simulator (in sim_main) starting at the L1 instruction cache. In addition, all macros used for accessing memory (such as READ_BYTE or READ_WORD) involve cache accesses as well.

The sim_check_options function parses the configuration parameters and creates the data structures corresponding to the specified memory hierarchy. To understand how caches work in the simulator, you may want to take a look at the data structures in the cache.h file and understand the code in the cache_create function. The same code is used to create the different levels in the cache hierarchy. Note that, as part of the initialization, a function pointer is passed as argument to the cache_create function. This function is used whenever an access to the lower cache level is needed (e.g., upon a cache miss or a write back).

Once you understand the data structures used for simulating the caches, you may want to read the code inside the cache_access function. This is the core function of the cache simulation. Since you are required to implement a new replacement policy, a good way of trying to understand the code is to understand what happens on a hit and on a miss, respectively. Follow the actions that are performed when an LRU replacement policy is simulated and try to figure out how these actions will change when implementing the NRU policy explained above. You may need to add new fields to the cache data structures to keep track of the “used” bits in the NRU scheme and the “rotating pointer”.

You shouldn’t need to modify the sim-cache.c file for this assignment. The simulator already has code to account for cache hits/misses, miss rates and prefetch statistics. Your code should be implemented only in the cache.c and cache.h files. You

In cache.c and cache.h, identify all modifications with the comments
/* ECE462 Assignment 2 - BEGIN CODE*/
... your code in here...
/* ECE462 Assignment 2 - END CODE*/
Extra Credit - Due: December 1, 2012
(40 pts to be added to your exam score, corresponds to 10 pts in the overall)

Optional

Anant Agarwal and Steven D. Pudar, “Column-Associative Caches: A Technique for Reducing the Miss Rate of Direct-Mapped Caches,” ISCA 1993

Guideline

Direct-mapped caches are the solution for simple and easy-to-design caches with short hit access time. However, the biggest drawback of using direct-mapped caches is the large number of conflict misses. Pseudo-associative caches resolve conflicts by allowing alternate hashing functions and show much higher hit rate than normal direct-mapped caches while maintaining almost the same hit access time. Basically a pseudo-associative cache is the same as a direct-mapped cache. The fundamental idea is to resolve conflicts by dynamically choosing different locations, which are accessed by different hashing functions. When a conflict miss happens, the pseudo-associative cache tries to avoid it by relocating the cache block using another rehashing function. The simplest solution of rehashing function is bit selection with the highest-order bit inverted, which is called bit flipping.

In order to avoid secondary thrashing effect, which is explained in detail in the reference paper each cache block is expanded to have extra 1-bit information called a rehash bit that indicates whether the block is a rehashed location or not. The rehash bit must be set to 1 when initialized in cache_create() function in cache.c.

Design

Add a new CACHE_TAG_PSEUDOASSOC macro in cache.c to get a tag value with the high-order bit of the index appended at the end.

#define CACHE_TAG_PSEUDOASSOC(cp, addr) …

Add one more variable in struct cache_blk_t for the rehash bit as following. The rehash bit must be initialized to 1 when the pseudo-associative cache is first created in cache_create() function in cache.c.

int rehash_bit;

You must modify cache_access() function in cache.c to implement the pseudo-associative cache for L1D. Since cache_access() is a general function used by all caches in the system and the pseudoassociative cache is only for L1D, you need to write new code for pseudo-associative cache specific to L1D.

Implementation

Add the following options for pseudo-associative cache.
-pseudoassoc <true/false> # false # use pseudo-associative cache in L1D
**Comparison**

Compare performance of the two L1D cache configurations assuming the same size (128 sets * 32-byte block size = 4KB).

(2) Pseudo-associative L1D : -cache:dl1 dl1:128:32:1:1 -pseudoassoc true
You do not need to consider various hit access times in the pseudo-associative cache. **Focus on only hit/miss rates** (dl1.hits/misses/miss_rate in SimpleScalar results).

You can get some additional information regarding HW3 in tutorial slides. 
ece.arizona.edu/~akoglu/classes/ece462/Assignments/Assignment3/hw3.pptx

**Reference**

[1] Todd Austin and Doug Burger, SimpleScalar Tutorial, (for tool set release 2.0);
[2] Todd M. Austin, A User’s and Hacker’s Guide to the SimpleScalar Architectural Research (for tool set release 2.0);