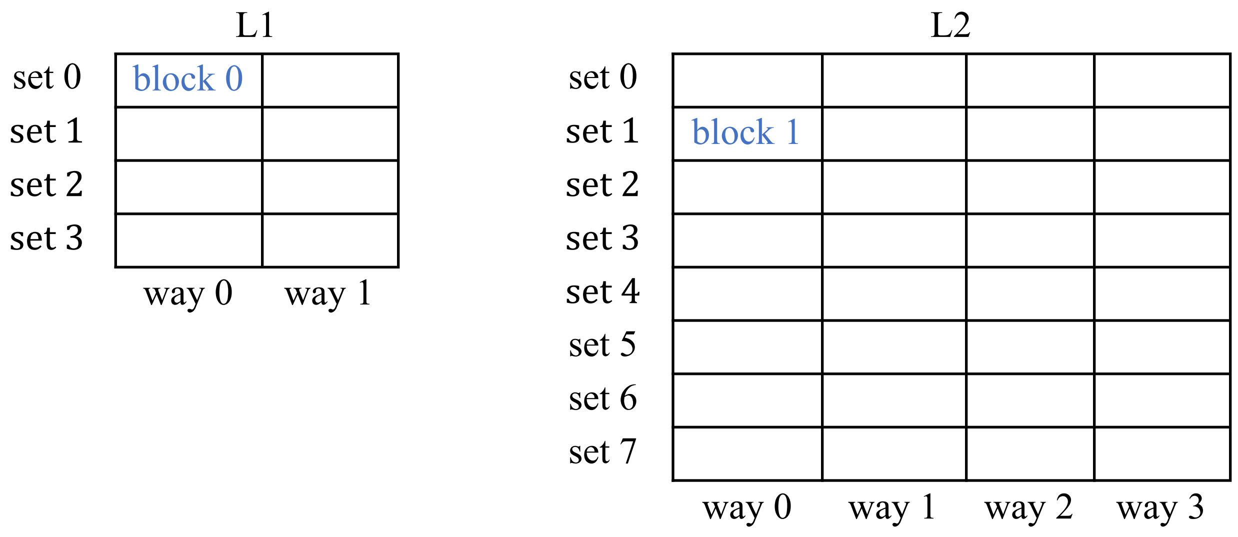
Lab 3: Cache Simulator

Due: Nov 3, 2023 11:59PM EST

In this lab assignment, you will implement a two-level (L1 and L2) cache simulator in C++. The cache simulator will take in several parameters as input to configure the cache (block size, associativity, *etc.*) along with a memory address trace file as input.

Cache System

Here is an example diagram of a two-level cache.
L1: Two-way set associative cache with a block size of 4 bytes.
L2: Four-way set associative cache with a block size of 4 bytes.



In general, a cache can be thought of as an array of sets where each set is a pointer to an array of ways. The size of each way is determined by the block size. You can consider a direct-mapped cache as a "one-way" set associative cache (i.e., having many sets with each set having exactly one way) and a fully associative cache having a single set with many ways.

Here’s a closer look of block 0 (L1) using 4 Bytes per block.

Byte 0	Byte 1	Byte 2	Byte 3	tag	dirty	valid

We will assume L1 and L2 have the same block size.

Since we are simulating cache operations, we will not consider actual data (i.e., the Bytes in the above figures) when testing inputs. You only need to model the accesses to capture hit and miss behavior, tracking all data movement and updating caches (with addresses) properly.

Address

An example of an address used in the cache (MSB → LSB):

tag	index	offset
↔	↔	↔

The index is used to select a set (i.e., a row in the above table). The tag is used to decide whether it “hits” in one of the set’s ways. The offset is used to pick a Byte out of a block.

Cache Policy to Implement

- (Exclusive) the L1 and L2 are exclusive. [Reference]
- For a W-way cache, the ways are numbered from {0,1, 2, …, W-1} for each set. If an empty way exists (i.e., the valid bit is low), data is placed in the lowest numbered empty way.
- (Eviction) When no empty way is available, eviction is performed using a round-robin policy. To select an eviction Way#, there is a counter initialized to 0 that counts to W-1 and loops back to zero. The current value of the counter indicates the Way from which data is to be evicted. The counter is incremented by 1 after an eviction occurs in the corresponding set. **Note that there is a counter for every set in the cache.**

Read Miss: on a read miss, the cache issues a read request for the data from the lower level of the cache.

- Tips for Read:
    - Read L1 hit: easy case, the data is available
    - Read L1 miss, Read L2 hit: first set the L2 valid bit low; the cache block is *moved* from the L2 cache to the L1 cache.
        - What if L1 is full? *Evict* a block from L1 to make space for the incoming data. Evicted L1 block will be placed to L2. What if L2 is full? Similarly evict a block from L2 before placing the data.
        - "first set the L2 valid bit low":
            1. At first the hit block is kicked out from L2 (thus it gives us an *empty* space of L2); after that you can continue to check if L1 is full (and if L2 is full).
    - Read L1 miss, Read L2 miss: read request for main memory; data fetch to be used and stored in L1. Again, what if L1 is full? Eviction.

Eviction follows the round-robin policy mentioned above.

- Tips for Eviction:
    - Evicting from L1: evicted data **must be placed into L2** (this might trigger an L2 eviction!)
    - Evicting from L2 for incoming data: check L2 dirty bit first
        - if the dirty bit is high, evict L2 data to main memory
        - otherwise, no need to write main memory; simply overwrite L2 with incoming data

Write Hit: both the L1 and L2 caches are **write-back** caches.
Write Miss: both the L1 and L2 caches are **write no-allocate** caches. On a write miss, the write request is forwarded to the lower level of the cache.
the cache.

- Tips for Write:
    - Write Hit L1: set the dirty bit high in the L1 cache
    - Write Miss L1, Write Hit L2: set the dirty bit high in the L2 cache
    - Write Miss L1, Write Miss L2: write data directly to main memory

Configuration File (cacheconfig.txt):

The parameters of the L1 and L2 caches are specified in a configuration file. The format of the configuration file is as follows.

- **Block size:** Specifies the block size for the cache in *bytes*. This should always be a non-negative power of 2 (i.e., 1, 2, 4, 8, etc.). **Block sizes will be the same for both caches in this lab.**
- **Associativity:** Specifies the associativity of the cache. A value of "1" implies a direct-mapped cache, while a "0" value implies fully-associative. Should always be a non-negative power of 2.
- **Cache size:** Specifies the total size of the cache data array in KiB.

An example config file is provided below. It specifies a 16KiB direct-mapped L1 cache with 8 byte blocks, and a 32KiB 4-way set associative L2 cache with 8 byte blocks.

Sample configuration file for L1 and L2 Cache (**cacheconfig.txt**):

```
L1:
8
1
16
L2:
8
4
32
```

In the above example, the L1 cache will have 2048 sets each with one way. The L2 cache will have 1024 sets each with 4 ways. The skeleton code provided reads the config file and initializes variables for the L1 and L2 cache parameters.

Trace File (trace.txt):

Your simulator will need to take as input a trace file that will be used to compute the output statistics. The trace file will specify all the data memory accesses that occur in the sample program. Each line in the trace file will specify a new memory reference. Each line in the trace cache will therefore have the following two fields:

- **Access Type:** A single character indicating whether the access is a read (R) or a write (W).
- **Address:** A 32-bit integer (in unsigned hex format) specifying the memory address that is being accessed.

Fields on the same line are separated by a single space. The skeleton code provided reads the trace file one line at a time in order. After each access, your code should emulate the impact of the access on the cache hierarchy.

Given a cache configuration and trace file, your cache simulator should be called as follows:

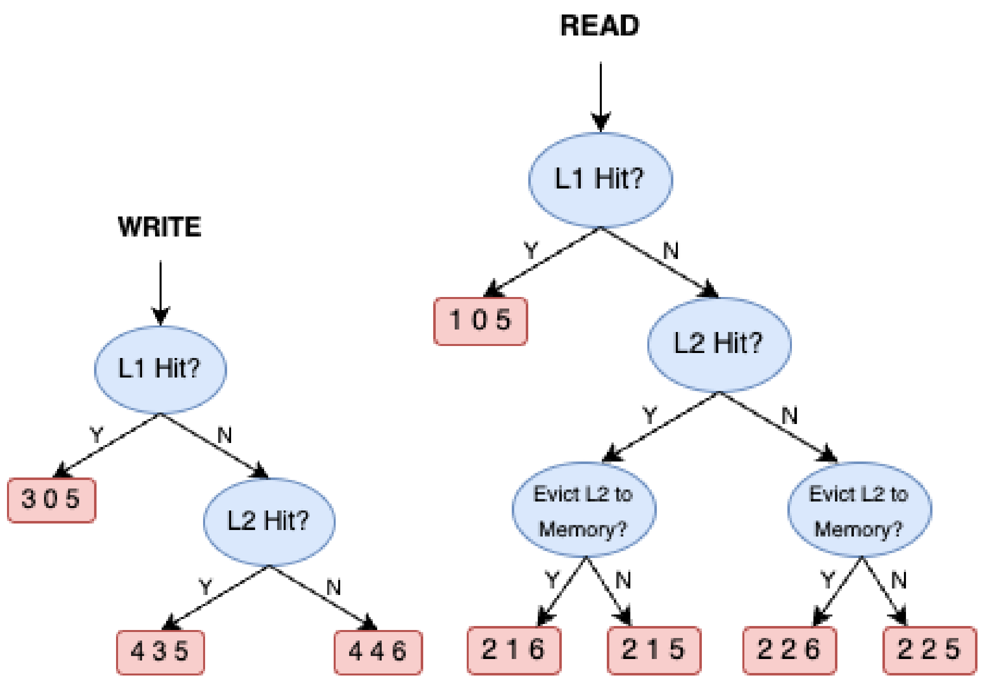
```
>>> ./cachesimulator.out cacheconfig.txt trace.txt
```

Simulator Output (trace.txt.out):

For each cache access, your simulator must output whether the access caused a read/write, hit/miss, or not accessed in the L1 and L2. Additionally, you must also output whether the current access caused a write to main memory. Each event is coded with a number, as defined in the skeleton code. Codes 0-4 are for the caches. Codes 5 and 6 are for main memory.

0: No Access
1: Read Hit
2: Read Miss
3: Write Hit
4: Write Miss
5: NO Write to Main Memory
6: Write to Main Memory

Familiarize yourself with the codes and make sure you understand each scenario below along with what your simulator output should be.



For example, if a write access misses in both L1 and L2, we must write the data directly to main memory so your cache simulator should output:

```
4 4 6
```

where the first number corresponds to the L1 cache event and the second to the L2 cache event, and the third to the main memory.

What you need to submit:

Similar to the previous labs, we will use GradeScope.

You must upload ONLY one file on Gradescope: "cachesimulator.cpp". Do not zip the file or upload any other file.