Computer Assignment 2 — Deadline: Sunday, Dec. 3, 11:55 PM
Fall 2023 — (Upload it to Gradescope.)

Here are some general guidelines:

- You should do all your work in the C++ programming language (sorry, no Python this time!). Your code should be written using only the standard libraries. You may or may not decide to use classes and/or structs to write your code. Regardless of the design, your code should be modular with well-defined functions and clear comments.
- You are free to use as many helper functions/class/definitions as needed in your project.
- There is no restriction on what data type (e.g., int, string, array, vector, etc.) you want to use for each parameter.
- Unfortunately, experience has shown that there is a very high chance that there are errors in this project description. The online version will be updated as errors are discovered, or if something needs to be described better. It is your responsibility to check the website often and read new versions of this project description as they become available.
- Sharing of code between students is viewed as cheating and will receive appropriate action in accordance with University policy. You are allowed to compare your results with others or discuss how to design your system. You are also allowed to ask questions and have discussions on Campuswire as long as no code is shared.
- You have to follow the directions specified in this document and turn in the files and reports that are asked for.
- You can use LLMs (e.g., ChatGPT) to generate the code for you. It is your responsibility to check the correctness of your code. You should also add comments where needed.

Project Description

Overview

In computer assignment 2, we want to design a memory hierarchy. The goal is to design a memory controller.

The input to your memory is a 32-bit address. Your code is responsible for loading the data from the memory (if it is an `LW`) or storing the value at the correct address if it is an `SW`. A skeleton code (`memory_driver.cpp`) is provided to you for initializing the

variables, creating the cache object and main memory array, reading a trace, and passing the relevant information to your memory hierarchy. You are free to change this skeleton as you see fit!

Your memory hierarchy **has two levels of cache and the main memory**. Your L1 cache should be a direct-mapped cache with 1 block per line with a total of 16 lines/sets. Each block in our cache should be 4 bytes (i.e., an int). Your cache and memory should be byte-addressable. It is your responsibility to correctly calculate the index, block offset, and tag in your code.

Your L2 cache should be an 8-way set-associative (SA) cache. Each block should be still 4 bytes and there should be 1 block per line (same as L1). There should be 16 sets for L2. Same as above, your code should correctly compute tag, index, and block offset. Your main memory has 4096 lines each one 4 bytes. For L2, use *LRU replacement policy* (L1 is DM, so no replacement policy is needed).

In addition to L1 and L2, you need to add a **victim cache** to your L1 cache. The victim cache has four entries and has to be fully-associative with 1 block per line and 4B per block (same as DM). The victim cache has to use LRU as the replacement policy.

Data evicted from the L1 cache should be installed in the victim cache (remember there is only one victim cache from the entire L1 cache so any set from L1 can install their line into the victim cache). The oldest line in the victim cache should be then replaced by this new line and the evicted line from the victim cache should be installed in L2 (and the evicted line from L2 should be removed).

On an L1 miss, you have to first search your Victim cache. If data is there, you should update the victim cache hit stat, and bring the data back to L1 (swap it with the corresponding line in L1). You should update the LRU states for the victim (with the new line as the most recent). If data is not in the Victim cache, you should search L2 similar to before. If data is in L2 or memory, it should be installed in L1 (and not the victim cache).

Your cache design should be ***exclusive/write-no-allocate/write-through.*** You should assume that your cache is initially randomly initialized with all valid bits equal to zero.

Trace

Each line in a trace has four values `(MemR, MemW, adr, data)`. The first two are either zero or one indicating whether this is a LOAD instruction or STORE (always only one of them will be equal to one). The address is a number between 0 and 4095 (you can safely assume this), and the data is a signed integer value. The code already reads the trace line by line and stores the values in an array called `myTrace`.

In the main loop of the driver code, each entry is read one by one and should be handled by your memory hierarchy.

As mentioned before, you can assume that your cache and memory are initially empty (with random values) and we first store things before reading anything.

We have provided three trace files. Same as CA1, your code will be tested with multiple different traces on Gradescope. We strongly recommend creating your own traces and testing your code using those.

Design

The entry for your code is your memory controller (`controller`) which is called in a loop in the main function of `memory_driver.cpp`. For each item in the trace, this controller function should be called, and appropriate actions should be taken. Apart from these actions, you should also update stats (i.e., various counters at different cache levels to keep track of the number of accesses per level and hits/misses per level) in this controller. We strongly recommend adding more functions to your cache object for each action (e.g., search, update, evict, etc.)

The controller should first check whether this request is a load or store and then follow an algorithm for each. For load, the controller should first search L1, Victim cache, and then L2. If data is not found in either, data should be brought from memory (remember that there is no *miss* in main memory and the address provided in the trace is the index for the main memory array).

Once data is found, you should make sure to **(1)** Update the stats for each cache, and **(2)** Update data, tag, and LRU positions if needed.

If data is in L1, you just need to update the stats. If it is in Victim, you should bring it to L1 and update LRU and tag. Similarly, if it is in L2, you should bring data to L1 (and remove it from L2) and update LRU, data, and tag. The evicted line from L1 should go to Victim and the evicted line from Victim should go to L2.

If data is not in any cache, data should be brought from memory and installed in L1 (update tag and data), the evicted data from L1 should be placed Victim, etc. (should be put as most recently used), and evicted data from L2 should be removed.

If it is a store, you should do similar things, but remember that we use a *write-no-allocate write-through* strategy (i.e., if data is in both cache and memory, both should be updated. If it is not in the cache, only memory should be updated, but update the stats for each level.) Remember that you need to search both caches (and Victim) for the store the same as the load, and you have to update the stats in both cases.

Cache Driver Code and Benchmarks:

The entry point of your project is "`memory_driver.cpp`". The program should run like this:

"`./memory_driver <inputfile.txt>`",

Your program should print the miss-rate for L1 and L2, as well as the average access time (AAT) of the system in the terminal.

"`(L1,L2,AAT)`"

For AAT, you can assume that the hit time for L1 is 1 cycle, for Victim cache is 1 cycle, for L2 is 8 cycles, and for main memory is 100 cycles. You are free to add more stats in your code if needed.

Three traces are provided.

Same as the previous project, it is your choice how you want to structure your code (e.g., whether you want to have separate objects for each class, or you want to instantiate an object within another class, or even not use any class at all and utilize functions and structs, etc.).

What to submit.

You need to submit the following files on Gradescope.

1. Your well-commented code (all the files – DO NOT put it under a folder). Note that we will use different traces to test your code's correctness. Your code should be compiled with the following command: "g++ *.cpp -o memory_driver". If the code fails to compile, you will lose points. Your code should produce the results

exactly as hard-coded in your skeleton code. Failing to create that format will result in losing points.