1 INTRODUCTION

Read the entire document before starting. There may be critical pieces of information and hints along the way.

Caches are complex memory and sometimes difficult to understand. One way to understand them is to build them. However, due to time constraints and lack of hardware expertise, we will instead write a cache simulator. In this project, you will be implementing a cache simulator for a multi-level cache hierarchy. We have provided you the following files, written in your favorite programming language **C**:

- **cachesim_driver.c** - The driver for your cache simulator.
- **cachesim.h** - Header file containing important declarations.
- **cachesim.c** - The cache simulator, missing some vital portions of code that you will be filling in.
- **Makefile** - To compile your code. Modify at your own risk.
- **traces** - The workloads inside the **traces** directory.
- **solution.log** - The solution generated by the TA's. You must debug your simulator until all the test cases match.
- **run_script.sh** - A script to run all the test cases.

You will be filling in some of the subroutines in the cachesim.c file, and then validating your output against the sample outputs that the TA's have generated. Keep in mind that we are simulating accesses, so there is no need to have a data store, you only need to keep track of the metadata.

This is **A fairly challenging** assignment. Start early and use office hours / Piazza for doubts and questions.

2 RULES

Please make sure to follow the following rules:

- All code must be your own work. There should be no sharing of code. **Please follow the Georgia Tech Honor Code**
- You may not use any standard libraries, or code that you have written in the past for completing this assignment. All work must be yours and done anew.
- You may discuss implementation details with your peers, but they cannot debug or provide you with any code.
- **DO NOT MODIFY** the driver file. This may cause our auto-grader to break, and you may get no credit for your hard work.

3 CACHE - CORE CAPABILITIES

Here are the specifications that your simulator must meet. There is a **direct** mapped L1 cache, followed by a **set associative** L2 cache. The L2 cache is followed by main memory in the hierarchy. The two caches are inclusive, which means that all the data contained in the L1 cache is also available in the L2 cache. The below picture details the hierarchy of the cache, along with showing the access times for each unit.

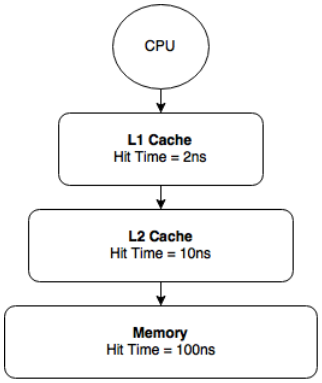

Figure 1: Memory Hierarchy of the Cache System with Access Times for each Level

3.1 L1 - Direct Mapped Cache

The below rules summarize the configuration of the L1 cache:

- The size of the cache is given by the command line parameter **C**. This means that the cache's data store is 2^C bytes.
- The size of each data block is given by the command line parameter **B**. The size of each block is 2^B bytes.
- The cache uses a **write back**, **write allocate** policy. This means that if the cache misses on a write, it must fetch a block of data from the next level of memory hierarchy, and it only writes data to the next level when that block is evicted, and the dirty bit is set.
- All the data contained in the L1 cache must also be contained in the L2 cache.

3.2 L2 - Set Associative Cache

The below rules summarize the configuration of the L2 cache:

- The size of the L2 cache's data store is given by the command line parameter **c**. The size of the data store is 2^c bytes.
- The set associativity of the L2 cache is given by the command line parameter **S**. And the number of sets in each cache line 2^S.
- The block size for this cache is the same as the block size for the L1 cache.
- The L2 cache is also **write back**, **write allocate**.
- Further the L2 cache uses **least recently used (LRU)** replacement policy. This means that if a set (aka line) is completely occupied with valid blocks, the least recently used block (i.e. The block used furthest back in time) will be evicted.

3.3 Some More Helpful Information

- Look into using arrays as your backing data structure. (Hint: 2D arrays are useful)
- You can use a clock counter to simulate LRU. That is, whenever a cache block is accessed, update the block's timestamp with the current clock counter value. Make sure to increment this counter in each access.
- Use **GDB** to debug your implementation and take care of segmentation faults.

3.4 Order of Operations

- When you access the L1 cache, and it is a hit, don't forget to update the corresponding L2 cache block's time stamp.
- Whenever you evict a block from the L2 cache, do not forget to evict the corresponding block from the L1 cache as well.
- Make sure to check for block validity before deciding to use LRU replacement in the L2 cache.

4 IMPLEMENTATION

Your Task is to fill out the following functions:

```
static uint64_t get_tag(uint64_t address, uint64_t C, uint64_t B, uint64_t S)
```

→ This subroutine computes the tag of an accessed address. When using this function make sure to pass the correct parameters corresponding to the cache whose tag you want to compute.

```
static uint64_t get_index(uint64_t address, uint64_t C, uint64_t B, uint64_t S)
```

→ This subroutine computes the index corresponding to the input address.

```
void cache_init(uint64_t C1, uint64_t C2, uint64_t B, uint64_t S)
```

→ This is the subroutine that initializes the cache. Allocate space for your caches in this function. You may add and initialize as many global or heap (malloc'ed) variables here.

```
void cache_access(char type, uint64_t address, cache_stats_t *stats)
```

→ This is the subroutine that will simulate cache accesses, one access at a time. The 'type' is going to be either READ or WRITE kind. The 'address' field is the memory address that is being accessed, 'stats' should be used for updating the cache statistics. Make sure to think of all the scenarios and to access both L1 and L2 caches based on misses and hits.

```
void cache_cleanup(cache_stats_t *stats)
```

→ Use this function to clean up any memory and for calculating final statistics. Update changes in the 'stats' parameter. Hint - All malloc'ed memory must be freed here.

NOTE: The TA's have been kind enough to provide detailed comments and give you the 'block' struct. Please read these comments to help you as you implement the simulator.

5 STATISTICS - AKA THE OUTPUT

The output from your final cache is the statistics that your cache calculates for each workload. Here is the list of fields inside the `cache_stats_t` struct and their meaning:

1. accesses - The total number of memory accesses your cache gets
2. reads - The total number of accesses that were reads (type == READS)
3. read_misses - The total number of reads that were cache misses
4. writes - The total number of accesses that were writes (type == WRITE)

5. write_misses - The total number of writes that were cache misses
6. misses - The total number of misses (Reads + Writes)
7. write_backs - The total number of times data was written back to memory
8. l1_read_misses - Misses in the L1 cache that were reads
9. l1_write_misses - Misses in the L1 cache that were writes
10. l2_read_misses - Misses in the L2 cache that were reads
11. l2_write_misses - Misses in the L2 cache that were writes
12. l1_access_time - The access time of the L1 cache (Already set in the driver)
13. l2_access_time - The access time of the L2 cache (Already set in the driver)
14. memory_access_time - The access time of main memory (Already set in the driver)
15. l1_miss_rate - Miss rate of the L1 cache
16. l2_miss_rate - Miss rate of the L2 cache
17. miss_rate - The overall miss rate of the cache hierarchy
18. l2_avg_access_time - The average access time of the L2 cache
19. avg_access_time - The average access time for your cache system. Aka the EMAT

The driver displays the output for these parameters onto the terminal screen. You only need to fill in the 'stats' structure passed in the `cache_access` and `cache_cleanup` functions.

6 HOW TO RUN YOUR CODE

We have provided you a 'Makefile' that you can use to compile your code. **Modify the Makefile at your own risk**

Use these commands to run the cache simulator driver:

```
$: make
$: ./cachesim -C <L1 size> -c <L2 size> -b <block size> -s <associativity> -i <Trace name>
```

- If you don't provide the cache sizes, block size and set associativity, the default values will be used.
- **Note:** To store the output of your simulator in a text file, pipe the output of the simulator to a text file adding '>> <File Name >' to the above command.

To run the script that will run all the tests, do the following:

```
$: bash run_script.sh
```

Finally: To pipe the output for all the test cases to a text file, do the following:

```
$: bash run_script.sh >> <Filename>
```

To **verify** that you are matching the TA generated output, you can use the diff command. Here is an example:

```
$: diff <Filename> solution.log
```

7 WHAT TO SUBMIT

To generate a submittable version of your project, type in `make submit` into the terminal. This will generate a tarball of your source code.

You need to submit the following files:

- The tarball you just generated

Don't forget to sign up for a demo. We will announce when these are available. Failure to demo will result in a score of 0.