INTRODUCTION

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 and then running some design experiments to find an optimal cache for the given workloads. We have provided you the following files, written in your favorite programming language **C**:

- **cachesim_driver.c** - The file that drives your cache.
- **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.txt** - The solution generated by the TA's. You must debug your simulator untill all the test cases match.
- **run_script.sh** - A script to run all the test cases.

You will be filling in the subroutines in the **cachesim.c** file, and then validating your output against the sample output the TA's have generated.

This is a simple assignment that will check your understanding of how caches work. **However, keep in mind that your choice of data structure used will greatly impact the difficulty of coding the simulator** If you are struggling to write code, then step back and review the concepts.

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 your code 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.

CACHE - CORE CAPABILITIES

Here are the specifications that your simulator must meet:

1. The simulator must model a cache with 2^C bytes of data storage, having 2^B byte blocks with 2^S blocks per set. **Note:** S = 0 implies a direct map cache, and S = C - B is a fully associative cache.
2. The cache implements a **write back, write allocate** strategy. This means there has to be a **dirty** bit for each block in the cache.
3. There is a **valid bit** for each block. This should be set to 0 at the start of the simulation.

4. The cache implements an **LRU** replacement policy. This means that the least recently used block is chosen for replacement.
5. All memory accesses are 64-bit (i.e Address length is 64 bits).
6. The cache is byte addressable.
7. The following formula is used for Average Access Time (aka EMAT)
   - AAT = Hit Time (HT) + Miss Rate(MR) * Miss Penalty(MP)
   - Given that HT = 2ns, and MP = 100ns (Set in cachesim.h)

IMPLEMENTATION

Your task is to fill out the following functions:

```
void cache_init(uint64_t c, uint64_t s, uint64_t b)
```

This is the subroutine that initializes the cache. 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.

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

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

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 number of accesses that were reads (type == READ)
3. read_misses - The total number of misses that were cache misses
4. writes - The 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. access_time - The access time of the cache (It is already set to 2ns in the driver)
9. miss_penalty - The miss penalty for a cache miss (It is already set to 100ns in the driver)
10. miss_rate - The miss rate for the cache
11. avg_access_time - The average access time for your cache, aka 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.

VALIDATION REQUIREMENT

Four sample simulation outputs will be provided on T-square. You must run your simulator and debug till your output **perfectly** matches all the statistics given in the sample output.

DESIGN EXPERIMENT

After validating your cache against the output given by the TA's, you need to design a cache for each trace in the trace directory under the following constraints:

1. Maximum cache size is **32 Kilo Bytes** including the cache tag store, valid bits, dirty bits and the actual data. You can exclude the bits used for LRU replacement.
2. The cache should have the lowest possible AAT.

You can vary any parameter, cache size, associativity and block size (C, B, S). However, keep in mind that the maximum block size is 64 bytes, that is **B is maximum 6**.

Write a **1 - 2 page report** giving the specifications of the best cache you came up with for each of the 4 traces we have provided you.

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 <Cache Size>-b <Block Size>-s <Associativity>-i <Trace File Name>
```

- If you don't provide the cache size, block size and set associativity, the default value 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 by adding '>>' <File Name> to the above command.

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

```
$: bash run_script
```

Finally: To pipe the output for all the test cases to a text file, you could run

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

To verify that you are matching the TA generated output, you can use the 'diff' command. Here is how you can do it:

```
$:    diff <File Name> solution.txt
```

WHAT TO SUBMIT

To generate a submittable version of your project, type in '`make submit`' into the terminal. You need to submit the following:

- The tarball you just generated
- The report summarizing the best cache for each trace

HINTS

- Before you start implementing the project, consider using structs for each block and think of what all you might need to store in the struct.
- For LRU replacement, we suggest using a global clock to assign times to each cache access. This will let you find the oldest block in a set when you need to evict a block.
- We suggest writing helper methods to break the address into tag, index and offset.
- Visualize how a cache looks in the book, and use that to organize the structure in your implementation. Arrange the data structure you are using in this manner.

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!