Computer Science 141
Lab 5.1: Cache Simulator

Due: 12/8 5:00pm

1 INTRODUCTION

1.1 CACHES

In this assignment, you will be creating three caches: a direct-mapped cache, a fully-associative cache, and an n-way set-associative cache.

Let's begin with a simple question: what is a cache? As far as this computer architecture course is concerned, a cache is a relatively small amount of really fast memory. When building a computer, it would be extremely expensive to store all of your data in fast memory (i.e. SRAM), so it all gets put in slower memory (DRAM) instead. We call the slow memory that stores all of the computer's data the main memory. A few blocks of memory – the ones that we are actively using – get copied from the main memory to a much smaller memory made out of SRAM called the cache. Our CPU stores data to and loads data from the cache, so as far as it is concerned, it is dealing with fast memory. We thus have two memories, both of which are cheap: the main memory is cheap because it is slow and the cache is cheap because it's small.

Side note: in real computer architectures, there isn't just a single cache and a main memory, but rather there is a multi-layered hierarchy. In order from fast/small to slow/big, there is an L1 cache, an L2 cache, the main memory, and the hard drive, and maybe some other levels in between! For this lab, though, the simplified version in the previous paragraph is all you need to worry about.

As you can imagine, there are times when the CPU asks the cache for some data and the cache doesn't have it! We call this a cache miss. (We call it a cache hit when the CPU asks the cache to interact with data that the cache does have.) In the case of a cache miss, the cache must request a new memory block from main memory that contains the requested address. If the cache is full when the request is received, it must first evict a block of memory that it is currently storing to make room for the new block.

This eviction process has three steps: first, the cache must choose which block to evict. Hopefully, the cache's design results in the evicted memory block being one we aren't likely to need again any time soon. Second, the cache must check to see if the to-be-evicted memory block is "dirty," meaning that at some point the processor has written to the memory block. If the block is dirty, the cache should write it back to main memory; if the block is not dirty, it would be a waste of time to write it back to main memory, so this is skipped. Third, the cache can complete the eviction once and for all by freeing the memory block. Once the memory block has been evicted, its spot is free for a new block of memory containing the data that the processor needs.

Cache misses are slow because they require interaction with main memory. Because of this, different cache designs have been designed to minimize the cache miss rate. In Lab 5.1, you will build three of the most common cache designs in C. In Lab 5.2, you will build one of these three designs in Verilog.

1.2 Assignment Infrastructure

This assignment is written in C, so to run your code, you will need a C compiler. Specifically, we ask that you use GCC. If you are using a Mac (or many Linux platforms), GCC should already be installed. *(Side note: if we want to be really precise, on a Mac you're actually compiling with Apple LLVM through the same commands as you would use for GCC.)* If you are using Windows, you are probably best off using the CS50 IDE: just go to cs50.io to get started.

We have written a Makefile, so to compile, you can just type `make all`. This will automatically detect which `.c` and `.h` files have been changed since a compilation last took place and will recompile only the necessary files. If you want to compile from scratch, type `make clean` and then `make all`.

To run the program, type `./main sc tests/t22.test`. You should see the following output:

```
MM: Read 32 bytes at 0x0.
Read from 0x4: 1037588349

MM: Read 32 bytes at 0x60.
Read from 0x64: -1126779786

MM: Read 32 bytes at 0x60.
MM: Wrote 32 bytes at 0x60.
Wrote to 0x64: -4

MM: Read 32 bytes at 0x0.
Read from 0x4: 1037588349

MM: Read 32 bytes at 0x60.
Read from 0x64: -4

*******************************************
Write Hit Rate:        0% (0/1)
Read Hit Rate:         0% (0/4)
Total Hit Rate:        0% (0/5)
Writes to Main Memory:  1
Reads from Main Memory: 5
*******************************************
```

If this displays: Congratulations! Your environment is up and running! If not, let us know on Piazza what went wrong and we'll make some recommendations.

1.3 STARTER CODE

To get you up and running, we have written a bunch of code that the caches you build will interact with. **The only files you should change are** `direct_mapped.c`, `fully_associative.c`, `set_associative.c`, **and the** `struct` **portions of** `direct_mapped.h`/`fully_associative.h`/`set_associative.h`. (You also can and should create additional tests in the `tests` folder.)

`memory_block.h, memory_block.c`

A `memory_block` is exactly what it sounds like: a chunk of continuous memory. Such a chunk of memory has a starting address (`start_addr`), a `size`, and the `data` itself. When you perform a read from main memory, main memory will create a memory block of size `MAIN_MEMORY_BLOCK_SIZE` starting at the address you requested and will return a pointer to it. You will also need to pass main memory a memory block whenever you do a write to main memory.

- You can create your very own memory block with a call to `memory_block* mb_new(void* start_addr, size_t size, void* source)`, where `start_addr` is a label representing the location where the memory block's data starts, `size` is the data's size (in bytes), and `source` is a pointer to the data that the memory block should copy into itself upon initialization.
- You can modify data in a memory block by using its `data` pointer. For an example, see the section of `simple.c` labeled `// Update relevant word in memory block`.
- When you are completely done with a memory block, you should free it from memory by calling `void mb_free(memory_block* mb)`.

`main_memory.h, main_memory.c`

The main memory is big enough to hold all of the data on the computer, and as such it is your cache's one-stop-shop should it find that it is missing the data that is requested of it. The main memory is initialized with the data located in `mm_init.data`. You can disregard the contents of the actual `struct main_memory` itself: you need not and should not ever use them directly. In other words, you should never write anything like `mm->data`, `mm->w_queries`, or `mm->r_queries`. You also should not call `mm_init()` or `mm_free()` – this is taken care of by the `main` function. But enough of what you should not do; here's what you can do with the main memory:

- You can write to the main memory with a call to `void mm_write(main_memory* mm, void* start_addr, memory_block* mb)`, where `mm` is a pointer to the main memory that our program is using (our program, from start to finish, only uses one main memory), `start_addr` is the address of the first byte you are writing, and `mb` is the memory block that you are writing. A few requirements also apply.
    1. It almost goes without saying that `start_addr` should actually be the address of the first byte of your `mb`'s data.
    2. The `mb` that you write must be aligned to a main memory block. Picture the main memory, which has a size of `MAIN_MEMORY_SIZE` as being split up

into `(MAIN_MEMORY_SIZE / MAIN_MEMORY_BLOCK_SIZE)` regions of size `MAIN_MEMORY_BLOCK_SIZE`. The `mb` that we write to main memory must start at the address of one of those regions, given that the main memory starts at the address `MAIN_MEMORY_START_ADDR`.
3. `main_memory` requires that the `mb` fed to it have a size of exactly `MAIN_MEMORY_BLOCK_SIZE`.
4. The `mb` that we are writing to main memory must actually fit in main memory. For example, if we were living in a universe where `MAIN_MEMORY_START_ADDR` is 0x4 and `MAIN_MEMORY_SIZE` is 16, it would be incorrect to write an `mb` starting at 0x0 or an `mb` of size 8 starting at 0x8.

- You can read from the main memory with a call to `memory_block* mm_read(main_memory* mm, void* start_addr)`, where `mm` is a pointer to the main memory that our program is using and `start_addr` is the starting address of the memory block that you want to retrieve. Two of the requirements from before apply:
    1. As with requirement 2 in `mm_write`, the `mb` that we request must be aligned to one of main memory's internal memory blocks if we assume main memory is split into regions as described above.
    2. As with requirement 4 in `mm_write`, the `mb` that we request should not be out of bounds.

`cache_stats.h, cache_stats.c`

This is a simple `struct` that holds four statistics every cache should track: the number of read and write queries that the cache has received, and the number of read and write misses that the cache has experienced. `cache_stats cs_init()` returns a `cache_stats` object with all four variables set to 0.

`main.c`

This code will test your data structure. To run a test, compile as explained in the Assignment Infrastructure section, and then run `./make sc tests/t22.test` or something like that. You can change `sc` to `dmc` (for direct mapped cache), `fac` (for fully associative cache), or `sac` (for set associative cache). You can also, of course, change the test file invoked. Note that, since our default value of `MAIN_MEMORY_SIZE` is 65536, addresses in the provided test files range from 0x0000 to 0xFFFC.

`direct_mapped.h, fully_associative.h, set_associative.h`

These are the header files for the caches that you will implement. You should only change the `struct` portion of these files.

`direct_mapped.c, fully_associative.c, set_associative.c`

This is where you will implement four functions for each of the caches that you build:

- `simple_cache* sc_init(main_memory* mm)` and company are called once at the beginning of main. It is their job to `malloc` room for the cache, to provide the cache with initial values, and to return a pointer to the cache that has been created. Each cache should hold onto a pointer to the main memory (which is passed in when `sc_init` is called), a `cache_stats` object, and whatever else you deem necessary depending on the specific cache you are implementing.

- `void sc_store_word(simple_cache* sc, void* addr, unsigned int val)` and company should store `val` at `addr`. You can assume that `addr` is properly aligned (i.e. is a multiple of 4 bytes away from `MAIN_MEMORY_START_ADDR`).
- `int sc_load_word(simple_cache* sc, void* addr)` and company should return the value stored at `addr`.
- `void sc_free(simple_cache* sc)` and company should free any resources that the cache is currently still holding on to (i.e. anything that has been `malloc`'ed but not yet `free`'d). Since memory blocks are on the heap (i.e. contain a `malloc` in their code), you will need to `mb_free` any that haven't been `mb_free`'d already.

`simple_cache.h, simple_cache.c`

To get you up and running, we have written the code for a "simple cache," which can be run with `./main sc [test file]`. The simple cache really isn't much of a cache at all, since it doesn't actually cache any data. Rather, it just passes every load and store request along to main memory in the form of reads and writes. You should definitely take a look at the code in `simple_cache.c`, as it is a great template for the code that you will have to write.

2 Direct Mapped Cache (50 pts)

Implement a direct mapped cache by filling out the `TODO` section of `direct_mapped.h` and the four `TODO` sections of `direct_mapped.c`. Your direct mapped cache should store `DIRECT_MAPPED_NUM_SETS` memory blocks, each of size `MAIN_MEMORY_BLOCK_SIZE`. Each memory block's mapping to a set should be based on the memory block's `DIRECT_MAPPED_NUM_SETS_LN` least significant bits (not, for example, the most significant bits). Do not edit anything in `direct_mapped.h` outside of the TODO section, and do not edit any other files. You have at your disposal the memory block, main memory, and cache stats functions described in the Starter Code section.

Requirements:

Core requirements: • Cache doesn't store more blocks than `DIRECT_MAPPED_NUM_SETS`. • Cache returns correct values when `dmc_load_word()` is called. When `dmc_store_word()` is called and `dmc_load_word()` is later called, the returned value should match what was stored. • Cache writes to main memory if and only if a **dirty** memory block is evicted. • Eviction is properly triggered with respect to a direct mapped cache. • Only `direct_mapped.c` and the `TODO` section of `direct_mapped.h` are modified. • Code compiles without warnings or errors.	35 pts
Stores correct statistics to `cache_stats` object.	5 pts
No memory errors.	2.5 pts
No memory leaks.	2.5 pts
Code works even if `DIRECT_MAPPED_NUM_SETS`, `MAIN_MEMORY_BLOCK_SIZE`, etc. are changed from initial values.	2.5 pts

Hints:

- Implement core requirements first. Next, add in statistics. Finally, test for and correct any remaining memory errors/leaks.
- Use `simple_cache.c` as a template. Make sure you understand what each line of `simple_cache.c` does before beginning.
- You may want to begin by writing a helper function `static int addr_to_set(void* addr)` that, for a starting address of a memory block, calculates the correct set index.
- You may, at some point, need to convert an address of type `void*` into an `unsigned int` or an `int`. If the compiler gets mad at you for doing this, use a cast of the form `int result = (uintptr_t) addr`.
- For this assignment, you only need to write to main memory during eviction. It is never necessary to flush all of the cache contents to main memory.
- Start by going through the tests in order, but only the tests that are a number by itself or a number followed by a d. Then, test against the f/s tests in order. We have provided the correct results that you should see for each test.
- It's not a bad idea to create test cases beyond the ones we give you. You can create really simple test cases and evaluate what should happen on paper or in your head to find bugs.
- Use a tool such as Valgrind's MemCheck or Dr. Memory to check for memory errors and leaks. If using Dr. Memory, perform a `make clean` and then recompile with `make all MODE=drmem`.
- Finish this part and test it against the test cases we give you before moving on to the next part. You'll want to make sure you really understand this before moving on to the next parts.

3 FULLY ASSOCIATIVE CACHE (25 PTS)

Implement a fully associative cache by filling out the `TODO` section of `fully_associative.h` and the four `TODO` sections of `fully_associative.c`. Your fully associative cache should store `FULLY_ASSOCIATIVE_NUM_WAYS` memory blocks, each of size `MAIN_MEMORY_BLOCK_SIZE`. You should use a **least recently used** eviction policy. Do not edit anything in `fully_associative.h` outside of the TODO section, and do not edit any other files. You have at your disposal the memory block, main memory, and cache stats functions described in the Starter Code section.

Requirements:

Core requirements: • Cache doesn't store more blocks than `FULLY_ASSOCIATIVE_NUM_WAYS`. • Cache returns correct values when `fac_load_word()` is called. When `fac_store_word()` is called and `fac_load_word()` is later called, the returned value should match what was stored. • Cache writes to main memory if and only if a **dirty** memory block is evicted. • Eviction is properly triggered with respect to a fully associative cache and an LRU policy. • Only `fully_associative.c` and the `TODO` section of `fully_associative.h` are modified. • Code compiles without warnings or errors.	13 pts
LRU tracker will never overflow.	2 pts
Stores correct statistics to `cache_stats` object.	4 pts
No memory errors.	2 pts
No memory leaks.	2 pts
Code works even if `FULLY_ASSOCIATIVE_NUM_WAYS`, `MAIN_MEMORY_BLOCK_SIZE`, etc. are changed from initial values.	2 pts

Hints:

- Implement core requirements first. Next, add in statistics. Finally, test for and correct any remaining memory errors/leaks.
- Again use `simple_cache.c` as a template.
- You may want to begin by writing a helper function `static void mark_as_used(fully_associative_cache* fac, int way)` that, when called for a given way, updates the cache's internal LRU tracker.
- You may also want to write a helper function `static int lru(fully_associative_cache* fac)` that, when called, returns the way that has least recently been used.
- For this assignment, you only need to write to main memory during eviction. It is never necessary to flush all of the cache contents to main memory.
- Start by going through the tests in order, but only the tests that are a number by itself or a number followed by an f. Then, test against the d/s tests in order. We have provided the correct results that you should see for each test.

- It's not a bad idea to create test cases beyond the ones we give you. You can create really simple test cases and evaluate what should happen on paper or in your head to find bugs.
- Use a tool such as Valgrind's MemCheck or Dr. Memory to check for memory errors and leaks. If using Dr. Memory, perform a `make clean` and then recompile with `make all MODE=drmem`.
- Finish this part and test it against the test cases we give you before moving on to the last part.

4 SET ASSOCIATIVE CACHE (15 PTS)

Implement a set associative cache by filling out the `TODO` section of `set_associative.h` and the four `TODO` sections of `set_associative.c`. Your set associative cache should store `SET_ASSOCIATIVE_NUM_SETS` sets, each with `SET_ASSOCIATIVE_NUM_WAYS` memory blocks of size `MAIN_MEMORY_BLOCK_SIZE`. Each memory block's mapping to a set should be based on the memory block's `SET_ASSOCIATIVE_NUM_SETS_LN` least significant bits and you should again use a **least recently used** eviction policy within each set. Do not edit anything in `set_associative.h` outside of the TODO section, and do not edit any other files. You have at your disposal the memory block, main memory, and cache stats functions described in the Starter Code section.

Requirements:

Core requirements: • Cache doesn't store more blocks than `SET_ASSOCIATIVE_NUM_SETS * SET_ASSOCIATIVE_NUM_WAYS`. • Cache returns correct values when `sac_load_word()` is called. When `sac_store_word()` is called and `sac_load_word()` is later called, the returned value should match what was stored. • Cache writes to main memory if and only if a **dirty** memory block is evicted. • Eviction is properly triggered with respect to a set associative cache and an LRU policy. • Only `set_associative.c` and the `TODO` section of `set_associative.h` are modified. • Code compiles without warnings or errors.	9 pts
LRU tracker will never overflow.	1 pt
Stores correct statistics to `cache_stats` object.	2 pts
No memory errors.	1 pt
No memory leaks.	1 pt
Code works even if `SET_ASSOCIATIVE_NUM_SETS`, `SET_ASSOCIATIVE_NUM_WAYS`, `MAIN_MEMORY_BLOCK_SIZE`, etc. are changed from initial values.	1 pt

Hints:

- Implement core requirements first. Next, add in statistics. Finally, test for and correct any remaining memory errors/leaks.

- It may be easier to adapt your code from fully_associative.c to include direct mapped cache ideas than to adapt your code from direct_mapped_cache.c to include fully associative ideas.
- You will want to copy your `addr_to_set()`, `mark_as_used()`, and `lru()` functions to `simple_cache.c` and make changes. For example, your might want to change the function signatures to `static void mark_as_used(set_associative_cache* sac, int set, int way)` and `static int lru(set_associative_cache* sac, int set)`.
- You may, at some point, need to convert an address of type `void*` into an `unsigned int` or an `int`. If the compiler gets mad at you for doing this, use a cast of the form `int result = (uintptr_t) addr`.
- For this assignment, you only need to write to main memory during eviction. It is never necessary to flush all of the cache contents to main memory.
- Start by going through the tests in order, but only the tests that are a number by itself or a number followed by an s. Then, test against the d/f tests in order. We have provided the correct results that you should see for each test.
- It's not a bad idea to create test cases beyond the ones we give you. You can create really simple test cases and evaluate what should happen on paper or in your head to find bugs.
- Use a tool such as Valgrind's MemCheck or Dr. Memory to check for memory errors and leaks. If using Dr. Memory, perform a `make clean` and then recompile with `make all MODE=drmem`.
- Congratulations when you've made it to the end!!!

5 BRIEF QUESTIONS (10 PTS)

Each of these questions is worth 2.5 points. You need not provide long responses.

1. Why do we use the least significant bits when building a DMC's address to set mapping rather than the most significant bits?

2. `t19.test` accesses memory blocks with indexes separated by multiples of 16. It only accesses 16 distinct memory blocks. Which of the three caches you built is worst at this test and why is it the worst? Which of the three caches you built is best at this test and why is it the best?

3. `t20.test` repeats through the same 17 memory blocks. Which of the three caches you built is worst at this test and why is it the worst? Which of the three caches you built is best at this test and why is it the best?

4. `t21.test` accesses memory blocks with indexes separated by multiples of 8. It repeats through the same three memory blocks. Which of the three caches you built is worst at this test and why is it the worst? Which of the three caches you built is best at this test and why is it the best?