Homework 5

System Skill (Term III/2022)

built on 2023/07/12 at 03:20:49 — ***due:*** *Saturday, July 22th @ 11:59pm*

This assignment will give you practice with manipulating bits, two's complement integers, and basic assembly. You will write some code and hand it in.

Hand-in Instructions

To submit this assignment, please follow the steps below:

1. The files to hand in are

   ```
   cacheLab.c cacheLab.h
   ```

   Zip up all the files as `a5.zip`

2. Log on to Canvas, go to assignment 5 submission page and submit your zip.

1 CacheLab (50 points)

The purpose of this assignment is to become more familiar with how cache works.

1.1 Handout Instructions

You will need a starter package, which ships in the form of a zip file.

Once extracted, you will find three files: `cacheSim.h`, `cacheSim.c` and `input.trace`.

Inside cacheSim.c, we have provided the skeleton code for our simple cache, which has the following properties:

- The cache has two levels
- A cache block is 16 bytes.
- Each address is accessing 1 bytes of data.
- The L1 cache is a 64 Bytes, 2-way set associative cache.
- The L2 cache is a 256 Bytes, 4-way set associative cache.
- The cache is inclusive, which mean that data in L1 cache will also remain in the L2 cache. In other word, the data in L1 is a subset of the data in L2 (This assumption simplify your design).
- The cache is using a **least recently used (LRU)** cache replacement policy. Note that this is simpler to implement than the LRU policy.

Your task is to build a cache simulator using the skeleton code we provided. To help you with the task, we have provided you with the following functions

- `input.trace` captures the trace of cache accesses. Each line in this file will contain three items: [0/1] [address] [data], where 0/1 is a single number (0 means this is a read request and 1 means this is a write request), [address] is a hexadecimal representation of the address of this particular cache access, and [data] is either 0 or the actual data the cache access need to write to (Note that a read request does not care about what the data is and you can ignore it. This piece of information is specifically for the write request). Please note that you can also create your own trace to test out your code. **Testing your code is a part of this assignment, and I am going to test your implementation using traces that I collect from real programs.**
- The `main()` function is where the simulator handle reading the input trace. You do not need to touch this except for the line that open the trace file (you should use your own trace file to test your cache).
- `init_DRAM()` initializes the value of DRAM. Please do not touch this.
- `printCache()` prints the content of the cache, which you can call at any time to test your code.
- `readInput()` reads one line of the input trace. This function is used in tandem with the `main` function and you do not have to touch this.
- `L1Lookup` checks the cache and see if your access hits in the L1 cache or not
- `L2Lookup` checks the cache and see if your access hits in the L2 cache or not

1.2 Part 1: Getting Tags, SetID and Performing a Cache Access

For the first part, you are going to write six functions.

- `unsigned int getL1SetID(u_int32_t address)` returns the setID assocated with the input address for L1 cache.
- `unsigned int getL1Tag(u_int32_t address)` returns the tag assocated with the input address for L1 cache.
- `unsigned int getL2SetID(u_int32_t address)` returns the setID assocated with the input address for L2 cache.
- `unsigned int getL2Tag(u_int32_t address)` returns the tag assocated with the input address for L2 cache.
- `int L1access(u_int32_t address, int type, u_int32_t data)` performs a L1 cache access. This function returns 1 if the input `address` is in the cache and 0 otherwise. Please note that this function **UPDATE** the cache content if we are writing to the cache.
- `int L2access(u_int32_t address, int type, u_int32_t data)` performs a L2 cache access. This function returns 1 if the input `address` is in the cache and 0 otherwise. Please note that this function **UPDATE** the cache content if we are writing to the cache.

1.3 Part 2: A Simple LRU-cache

For this part, you are going to implement the cache insertion and eviction based on the LRU replacement policy. Please note that this policy is different and simpler (and likely to be worse) than the LRU policy we covered in class.

You are going to write another three functions.

- `void L1replace(cacheBlock block, int setID)` performs a L1 cache eviction and insert `block` to the L1 cache using the LRU policy.
- `void L2replace(cacheBlock block, int setID)` performs a L2 cache eviction and insert `block` to the L1 cache using the LRU policy.
- `uint32_t read(u_int32_t address)` processes a read request by properly going though the cache and updating the cache content based on the FIFO replacement policy. The function should return the data you are reading.

Simplifying Things: For our question, we guarantee that the address is divisible by 4 so that the integer that you update fits within a single cache block.

1.4 Part 3: Handling Writes

After you correctly handle read requests (and assume that all write requests behave like a read request), your next task is to handle write requests. For this assignment, we assume a write-through cache, which performs a write to all cache levels (i.e., if there is a write to address `addr`, this new value should be updated for all the cache levels as well as in the DRAM).

Specifically, you need to finish up the following function:

`void write(uint32_t address, uint32_t data)`, which performs a write to `address` by updating the byte at our address with `data`.

Hint 1: When you have a write hit, please carefully check which bytes in the cache block should be updated.

Hint 2: Please reuse your code. This part should be very simple now

1.5 Your Reference Traces

Midway through your assignment (on Friday July 14th), you will be given a few new test trace files (i.e. a subset of our test inputs), a reference solution for all these traces. Please generate your own tests. You can use Linux's `diff` command to compare your output and the reference output. Your work will be graded based on correctness when running against our real trace inputs.