Homework 5

System Skill (Term III/2021)

built on 2022/07/12 at 02:46:18 ***due:*** *Monday, July 25th @ 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. Log-in to Syskill using your Mahidol ID (i.e., uXXXXXX).
2. The files to hand in are

   ```
   cacheLab.c
   ```

   Zip up all the files as `a5.zip`
3. Find out the MD5 hash of your zip file. You will need to submit this code on Canvas. We use it for keeping track of your submission time. You may resubmit your work but the MD5 hash has to match.

   ```
   > md5sum a5.zip
   ```
4. Copy the zip file to the directory `/handin/a5/u5712345` where `u5712345` is your student ID.

   ```
   > cp a5.zip /handin/a5/u5712345
   ```
5. Log on to Canvas, go to assignment 5submission page and enter the MD5 hash.

1 CacheLab (50 points + 10 points EC)

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 tar file. On syskill, you can find `cacheLab-handout.tar` at

```
/handout/a5/cacheLab-handout.tar.gz
```

You can extract everything using the `tar` command in a similar fashion as assignment 3.

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 **first-in-first-out** 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.

1.2 Part 1: Getting Tags, SetID and Performing a Cache Lookup (20 points)

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

- `unsigned int getL1SetID(uint32_t address)` returns the setID assocated with the input address for L1 cache.
- `unsigned int getL1Tag(uint32_t address)` returns the tag assocated with the input address for L1 cache.
- `unsigned int getL2SetID(uint32_t address)` returns the setID assocated with the input address for L2 cache.
- `unsigned int getL2Tag(uint32_t address)` returns the tag assocated with the input address for L2 cache.
- `int L1lookup(uint32_t address)` performs a L1 cache lookup. This function returns 1 if the input `address` is in the cache and 0 otherwise. Please note that this function **DOES NOT** perform any actual insertion or eviction.
- `int L2lookup(uint32_t address)` performs a L2 cache lookup. This function returns 1 if the input `address` is in the cache and 0 otherwise. Please note that this function **DOES NOT** perform any actual insertion or eviction.

1.3 Part 2: A Simple FIFO-cache (30 points)

For this part, you are going to implement the cache insertion and eviction based on a FIFO replacement policy. The FIFO cache replacement policy choose to evict the cache block that was inserted first (literally, first-in first-out). 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.

- `uint32_t read_fifo(uint32_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.

Be careful: The input request can go to a 4-byte address that spans two cache blocks. Note that the compiler will usually break this particular data access into two different requests, but we are simplifying the cache here so we will assume that every request is for a 4-byte data and address can start at any byte.

1.4 Part 3: Handling Writes (Extra Credit: 10 points)

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: You can reuse most of the cache eviction code from part 2 in order to *both* update the data and insert/evict cache blocks at the same time.