CS 6354: Graduate Computer Architecture Fall 2024

Homework 4

HW Out: Oct 22 **HW Due:** Nov 7th *(100 points)*

Student Name: ______________________________

Collaborator Name(s): ______________________________

Instructions

In this homework, you will create a **direct-mapped cache** and **set-associative cache** using Python. Each memory address (byte or word address) points to some data in the memory. This cache simulator essentially works only on addresses. The actual value of the data is not of interest to us in this cache simulator project.

1. You will need Python3 installed on the machine/laptop you are using. We have tested the program on Ubuntu systems and we recommend the students use the same. CS/CpE students with CS-Portal accounts should be able to SSH and run their experiments on the portal servers. Make sure you use *python3*. You might also have to install *numpy* with *pip3* if it is not already installed.
2. To simulate a direct mapped cache, go the directory containing the *sim_dm.py* file and use the command ***python3 sim_dm.py <PATH TO TRACE> <SIZE OF CACHE> <WAYS> <BLOCK SIZE>***. Replace *sim_dm.py* with *sim_sa.py*. You can redirect the standard output to a file if you need.

Deliverables

1. **2 python files**, **cache_dm.py** for the Direct Mapped Cache and **cache_sa.py** for the set associative cache. DO NOT CHANGE THE NAME OF THE PYTHON FILES.
2. **1 report** containing,
   (a) Output of the Code
   (b) Explanation of the your code(logic used, clarifications about the output)
   (c) Your work for the warm up sections and verification sections for both the Direct Mapped Cache and Set Associative Cache.

   Showing your work is not optional and your final score will be highly dependent on the quality of the report.

1 Direct Mapped Cache (50)

1.1 Warm-up

Below is a sequence of twelve 32-bit memory address references given as word addresses. 1, 18, 2, 3, 4, 20, 5, 21, 33, 34, 1, 4. Note that word and byte addresses are different. The word address appended by offset (two zeros) gives you the byte address. In other words, word addresses are multiples of 4. For example, if the word address is "1", the equivalent byte address is "4" (100).

1. Assuming a direct-mapped cache with one-word blocks and a total size of 16 blocks, list if each reference is a hit or a miss assuming the cache is initially filled with word address 0, 1, 2, . . . 15 memory data. Show the state of the cache after the last reference. Calculate the hit rate for this reference string. (Correct answer: hit rate = 4/12)
2. Now, assuming a direct-mapped cache with two-word blocks and a total size of 8 blocks, list if each reference is a hit or a miss assuming the cache is initially empty. Show the state of the cache after the last reference. Calculate hit rate for this reference string. (Correct answer: hit rate = 1/12)

1.2 Direct Mapped Cache (40 points)

We will incrementally build a simple cache simulator in python. The goal is to understand the concepts studied in the class in more depth. We will start by building a simple direct-mapped cache simulator. There are three folders in the resources folder: *traces, code,* and *reference_outputs.* In the code directory, we have two python files: a) *sim_dm.py* and *cache_dm.py.* Read the python files carefully. The *sim_dm.py* is the direct mapped cache simulator file. *cache_dm.py* is the class file, which contains cache class definition and empty functions. These functions are called appropriately in *sim_dm.py.* You need to implement these functions – there is no need to change anything else in the code. In summary, you need to write the following four python functions.

- **find_set** function takes an address and returns the set value (Note: for direct-mapped caches, number of ways = 1, and the number of sets = number of blocks in the cache).
- **find_tag** function takes an address and returns the tag value.
- **find** function takes an address and returns a bool (false for a cache miss, true for a cache hit). You can reuse *find_set* and *find_tag* functions to implement this find function.
- **load** function takes an address and loads the appropriate data in the cache. Note that load function is only called in *sim_dm.py* when there is cache miss. You can reuse the above implemented functions if you need them.

The documentation related to traces folder is available in *sim_dm.py.* The traces folder essentially contain reference strings (second column). These reference strings are byte addresses in decimal.

1.2.1 Example command line

```
python3 sim_dm.py test.trace 16 1 8
```

where

- test.trace is a trace file stored in ../traces
- 16 is the cache size in bytes (size = block size x sets x ways; All units in Bytes)
- 1 is the number of ways
- 8 is the number of bytes per block

1.3 Verification (10 points)

Verify that the hit rate obtained from the python cache simulator is the same as what you got from the calculation performed in the warm-up section. The related trace files for are *dm_a.trace* and *dm_b.trace.* Note that in *dm_a.trace*, the first 16 lines are just there to fill the cache as per the warm-up requirement; do not consider them for the calculation for miss-rates.

2 Set Associative Cache (50 points)

2.1 Warm-up

Below is a sequence of twelve 32-bit memory address references given as word addresses. 1, 18, 2, 3, 4, 20, 5, 21, 33, 34, 1, 4. Note that word and byte addresses are different. The word address appended by offset (two zeros) gives you the byte address. In other words, word addresses are multiples of 4. For example, if the word address is "1", the equivalent byte address is "4" (100).

Assuming a set-associative cache with two ways, two-word blocks and a total size of 8 blocks, list if each reference is a hit or a miss assuming the cache is initially empty (assume LRU replacement). Show the state of the cache after the last reference. Calculate the hit rate for this reference string. (Correct answer: hit rate = 5 / 12)

2.2 Set Associative Cache (40 points)

We will incrementally build a simple cache simulator in python. We will be building a set associative cache. There are three folders in the resources folder: *traces, code,* and *reference_outputs.*

In the code directory, we have two python files: a) *sim_sa.py* and *cache_sa.py.* Read the python files carefully. The *sim_sa.py* is the set associative cache simulator file. *cache_sa.py* is the class file, which contains cache class definition and empty functions. These functions are called appropriately in *sim_sa.py.* You need to implement these functions – there is no need to change anything else in the code. In summary, you need to write the following four python functions.

- **find_set** function takes an address and returns the set value (Note: for a set-associative cache, number of ways is greater than 1, and the number of sets = number of blocks/ways in the cache).
- **find_tag** function takes an address and returns the tag value.
- **find** function takes an address and returns a bool (false for a cache miss, true for a cache hit). You can reuse *find_set* and *find_tag* functions to implement this find function.
- **load** function takes an address and loads the appropriate data in the cache. Note that load function is only called in *sim_dm.py* when there is cache miss. You can reuse the above implemented functions if you need them.

The documentation related to traces folder is available in *sim_sa.py.* The traces folder essentially contain reference strings (second column). These reference strings are byte addresses in decimal.

2.2.1 Example command line

```
python3 sim_sa.py test.trace 16 2 8
```

where

- test.trace is a trace file stored in ../traces
- 16 is the cache size in bytes (size = block size x sets x ways; All units in Bytes)
- 1 is the number of ways
- 8 is the number of bytes per block

2.3 Verification (10 points)

Verify that the hit rate obtained from the python cache simulator is the same as what you got from the calculation performed in the warm-up section. The related trace file for is sa.trace.