CS 6354 Graduate Computer Architecture Fall 2024

Homework 4 Report

Student Name: Jing-Ning Su (fzf9mg)

1. Direct Mapped Cache

1.1 Warm-up Exercise

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).

Question 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)

From the table below, we can see hit rate is 4/12, and cache is 0, 1, 34, 3, 4, 21, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 after the last reference. (view table “Last” below)

(each box in the follows represent a word, each box forms a block)

Addr	1	18	2	3	4	20	5	21	33	34	1	4	Last
H/M	H	M	M	H	H	M	H	M	M	M	M	M	
Cache Status	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	1	1	1	1	1	1	1	1	33	33	1	1
	2	2	18	2	2	2	2	2	2	2	34	34	34
	3	3	3	3	3	3	3	3	3	3	3	3	3
	4	4	4	4	4	4	20	20	20	20	20	20	4
	5	5	5	5	5	5	5	5	21	21	21	21	21
	6	6	6	6	6	6	6	6	6	6	6	6	6
	7	7	7	7	7	7	7	7	7	7	7	7	7
	8	8	8	8	8	8	8	8	8	8	8	8	8
	9	9	9	9	9	9	9	9	9	9	9	9	9
	10	10	10	10	10	10	10	10	10	10	10	10	10
	11	11	11	11	11	11	11	11	11	11	11	11	11
	12	12	12	12	12	12	12	12	12	12	12	12	12
	13	13	13	13	13	13	13	13	13	13	13	13	13
	14	14	14	14	14	14	14	14	14	14	14	14	14
	15	15	15	15	15	15	15	15	15	15	15	15	15

Question 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

From the table below, we can see hit rate is 1/12, and cache is [0, 1][34, 35][4, 5][-, -][-, -] [-, -] [-, -] [-, -] after the last reference. (view table "Last" below)

(each box in the follows represent a word, every two boxes form a block)

Addr	1		18		2		3		4		20	
H/M	M		M		M		H		M		M	
Cache Status			0	1	0	1	0	1	0	1	0	1
					18	19	2	3	2	3	2	3
											4	5

5		21		33		34		1		4		Last	
M		M		M		M		M		M			
0	1	0	1	0	1	32	33	32	33	0	1	0	1
2	3	2	3	2	3	2	3	34	35	34	35	34	35
20	21	4	5	20	21	20	21	20	21	20	21	4	5

1.2 Direct Mapped Cache Implementation

Code Implementation:

1. find_set

```
def find_set(self, address):
    return (address >> self.blockBits) & (self.sets - 1)
```

Find the cache set identifier corresponding to the address by shifting the address right by a number of bits equal to the block size and then subtracting one from the resulting value.

2. find_tag

```
def find_tag(self, address):
    return address >> (self.blockBits + self.setBits)
```

The tag field used in the cache indexing process is derived by shifting the address right, effectively discarding the bits corresponding to the cache set index and block offset.

3. find

```
def find(self, address):
    set_index = self.find_set(address)
    tag = self.find_tag(address)

    if self.metaCache[set_index][0] == tag:
        self.hit += 1
        return True
    else:
        self.miss += 1
        return False
```

Use the `find_set` function to determine the cache set index corresponding to the address, then use the `find_tag` function to extract the tag, and then verify whether there is a matching tag in the identified cache set to determine whether it is a cache hit.

4. load

```
def load(self, address):
    set_index = self.find_set(address)
    tag = self.find_tag(address)

    self.metaCache[set_index][0] = tag

    self.cache[set_index][0] = address
```

When a cache miss occurs, the `find_set` function determines the cache set index for the address and then extracts the tag. Subsequently, the tag in the corresponding cache set is updated to the tag of the new address, and the data block associated with the new address is loaded from memory into the cache to implement the cache replacement strategy.

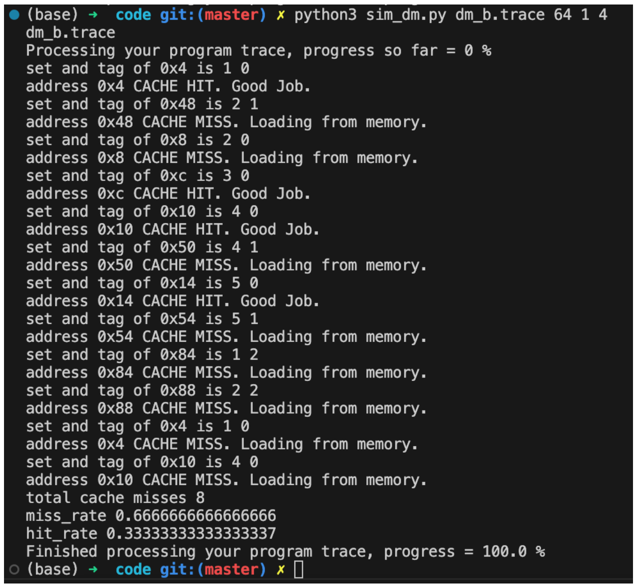
Simulation and Results:

This part is just the testing way when I was coding, for the way of using dm_a.trace, see “2.3 Verification”.

For question 1 in warm-up, since the cache is initially filled, I modified the following code temporarily to simulate the situation. Then, I can use `dm_b.trace` directly.

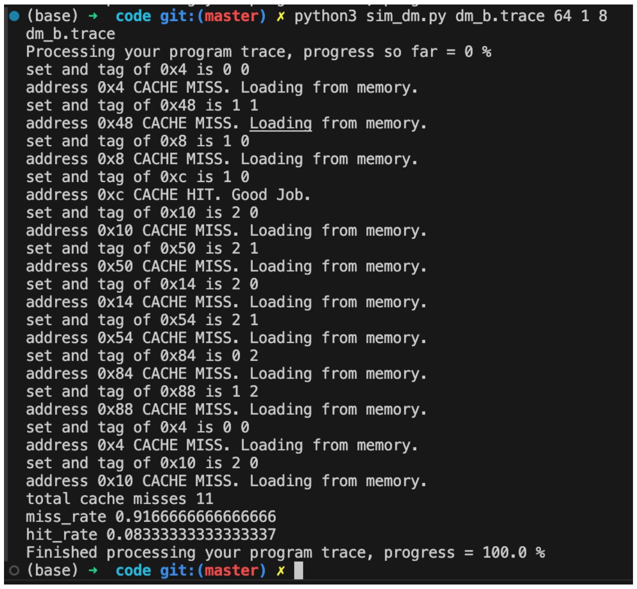
```
# modified here
def reset(self):
    // …
    for i in range(self.sets):
        for j in range(self.ways):
            start_value = j * self.sets * self.blockSize + i * self.blockSize
            self.cache[i][j] = np.arange(start_value, start_value + self.blockSize, dtype=int)
    // …
```

The output of the first question is as follow, corresponds with the result of hit rate = 4/12.



```
(base) → code git:(master) ✗ python3 sim_dm.py dm_b.trace 64 1 4
dm_b.trace
Processing your program trace, progress so far = 0 %
set and tag of 0x4 is 1 0
address 0x4 CACHE HIT. Good Job.
set and tag of 0x48 is 2 1
address 0x48 CACHE MISS. Loading from memory.
set and tag of 0x8 is 2 0
address 0x8 CACHE MISS. Loading from memory.
set and tag of 0xc is 3 0
address 0xc CACHE HIT. Good Job.
set and tag of 0x10 is 4 0
address 0x10 CACHE HIT. Good Job.
set and tag of 0x50 is 4 1
address 0x50 CACHE MISS. Loading from memory.
set and tag of 0x14 is 5 0
address 0x14 CACHE HIT. Good Job.
set and tag of 0x54 is 5 1
address 0x54 CACHE MISS. Loading from memory.
set and tag of 0x84 is 1 2
address 0x84 CACHE MISS. Loading from memory.
set and tag of 0x88 is 2 2
address 0x88 CACHE MISS. Loading from memory.
set and tag of 0x4 is 1 0
address 0x4 CACHE MISS. Loading from memory.
set and tag of 0x10 is 4 0
address 0x10 CACHE MISS. Loading from memory.
total cache misses 8
miss_rate 0.6666666666666666
hit_rate 0.33333333333333337
Finished processing your program trace, progress = 100.0 %
(base) → code git:(master) ✗
```

For question 2, the cache is not initially filled, so I removed the change in `reset` function. The command and result are as follow; it corresponds with the result of hit rate = 1/12.



```
(base) → code git:(master) ✗ python3 sim_dm.py dm_b.trace 64 1 8
dm_b.trace
Processing your program trace, progress so far = 0 %
set and tag of 0x4 is 0 0
address 0x4 CACHE MISS. Loading from memory.
set and tag of 0x48 is 1 1
address 0x48 CACHE MISS. Loading from memory.
set and tag of 0x8 is 1 0
address 0x8 CACHE MISS. Loading from memory.
set and tag of 0xc is 1 0
address 0xc CACHE HIT. Good Job.
set and tag of 0x10 is 2 0
address 0x10 CACHE MISS. Loading from memory.
set and tag of 0x50 is 2 1
address 0x50 CACHE MISS. Loading from memory.
set and tag of 0x14 is 2 0
address 0x14 CACHE MISS. Loading from memory.
set and tag of 0x54 is 2 1
address 0x54 CACHE MISS. Loading from memory.
set and tag of 0x84 is 0 2
address 0x84 CACHE MISS. Loading from memory.
set and tag of 0x88 is 1 2
address 0x88 CACHE MISS. Loading from memory.
set and tag of 0x4 is 0 0
address 0x4 CACHE MISS. Loading from memory.
set and tag of 0x10 is 2 0
address 0x10 CACHE MISS. Loading from memory.
total cache misses 11
miss_rate 0.9166666666666666
hit_rate 0.08333333333333337
Finished processing your program trace, progress = 100.0 %
(base) → code git:(master) ✗
```

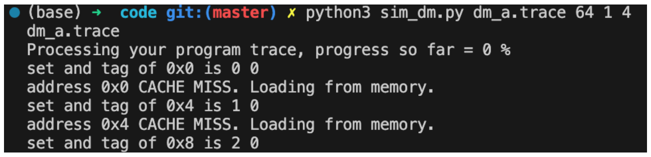
2.3 Verification

The way of using `dm_a.trace` and `dm_b.trace` for testing is as follows.

Question 1:

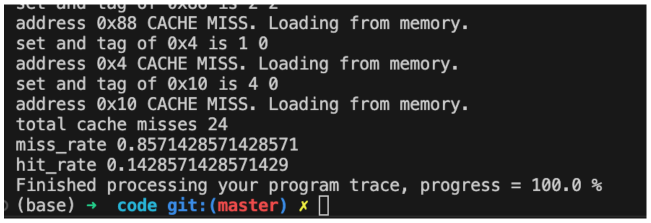
Since question 1 has one-word block and 16 blocks in total, total cache size is 16 * 4 with block size 4. The command line is `python3 sim_dm.py dm_a.trace 64 1 4`.

The output is as follows, however, the middle of it is omitted due to the length.



```
(base) ➜  code git:(master) ✗ python3 sim_dm.py dm_a.trace 64 1 4
dm_a.trace
Processing your program trace, progress so far = 0 %
set and tag of 0x0 is 0 0
address 0x0 CACHE MISS. Loading from memory.
set and tag of 0x4 is 1 0
address 0x4 CACHE MISS. Loading from memory.
set and tag of 0x8 is 2 0
```

…

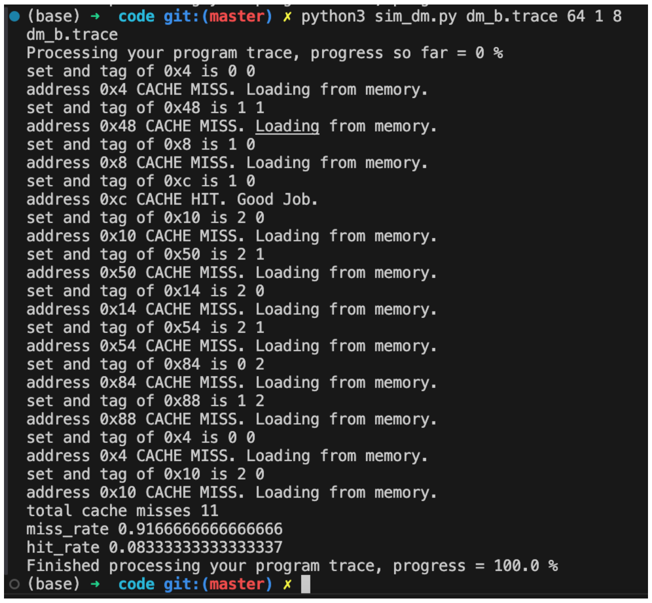


```
address 0x88 CACHE MISS. Loading from memory.
set and tag of 0x4 is 1 0
address 0x4 CACHE MISS. Loading from memory.
set and tag of 0x10 is 4 0
address 0x10 CACHE MISS. Loading from memory.
total cache misses 24
miss_rate 0.8571428571428571
hit_rate 0.1428571428571429
Finished processing your program trace, progress = 100.0 %
(base) ➜  code git:(master) ✗
```

The result is 24 misses, but we need to minus 16 because they were the lines used to initialize the cache. Therefore, the number of true misses is 8, which corresponds with the result of question 1.

Question 2:

Since question 2 has two-word block and 8 blocks in total, total cache size is 8 * 8 with block size 8. The command line is `python3 sim_dm.py dm_a.trace 64 1 8`.



```
(base) ➜  code git:(master) ✗ python3 sim_dm.py dm_b.trace 64 1 8
dm_b.trace
Processing your program trace, progress so far = 0 %
set and tag of 0x4 is 0 0
address 0x4 CACHE MISS. Loading from memory.
set and tag of 0x48 is 1 1
address 0x48 CACHE MISS. Loading from memory.
set and tag of 0x8 is 1 0
address 0x8 CACHE MISS. Loading from memory.
set and tag of 0xc is 1 0
address 0xc CACHE HIT. Good Job.
set and tag of 0x10 is 2 0
address 0x10 CACHE MISS. Loading from memory.
set and tag of 0x50 is 2 1
address 0x50 CACHE MISS. Loading from memory.
set and tag of 0x14 is 2 0
address 0x14 CACHE MISS. Loading from memory.
set and tag of 0x54 is 2 1
address 0x54 CACHE MISS. Loading from memory.
set and tag of 0x84 is 0 2
address 0x84 CACHE MISS. Loading from memory.
set and tag of 0x88 is 1 2
address 0x88 CACHE MISS. Loading from memory.
set and tag of 0x4 is 0 0
address 0x4 CACHE MISS. Loading from memory.
set and tag of 0x10 is 2 0
address 0x10 CACHE MISS. Loading from memory.
total cache misses 11
miss_rate 0.9166666666666666
hit_rate 0.08333333333333337
Finished processing your program trace, progress = 100.0 %
(base) ➜  code git:(master) ✗
```

The result is 11 misses, which corresponds with the result of question 2.

3. Set Associative Cache

3.1 Warm-up Exercise

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).

Question: 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)

From the table below, we can see hit rate is 5/12, and cache is [[0,1], [34,35],[4,5][-,-]], [[32,33],[2,3],[20,21],[-,-]] after the last reference. (view table “Last” below)

(each box in the follows represent a word, every two boxes form a block, two tables represent ways)

Addr	1	18	2
H/M	M	M	M
Cache Status	[, ▨] [,] [,] [,] [,] [,] [,] [,]	[0, 1] [,] [▨,] [,] [,] [,] [,] [,]	[0, 1] [,] [18, 19] [▨,] [,] [,] [,] [,]

3	4	20	5
H	M	M	H
[0, 1] [,] [18, 19] [2, 3] [,] [,] [,] [,]	[0, 1] [,] [18, 19] [2, 3] [▨,] [,] [,] [,]	[0, 1] [,] [18, 19] [2, 3] [4, 5] [▨,] [,] [,]	[0, 1] [,] [18, 19] [2, 3] [4, 5] [20, 21] [,] [,]

21	33	34	1
H	M	M	H
[0, 1] [,] [18, 19] [2, 3] [4, 5] [20, 21] [,] [,]	[0, 1] [, ▨] [18, 19] [2, 3] [4, 5] [20, 21] [,] [,]	(LRU Applies) [0, 1] [32, 33] [18, 19] [2, 3] [4, 5] [20, 21] [,] [,]	[0, 1] [32, 33] [34, 35] [2, 3] [4, 5] [20, 21] [,] [,]

4				Last			
H							
0	1	32	33	0	1	32	33
34	35	2	3	34	35	2	3
4	5	20	21	4	5	20	21

3.2 Set Associative Cache Implementation

Code Implementation:

1. find_set

```
def find_set(self, address):
        set_mask = (1 << self.setBits) - 1
        return (address >> self.blockBits) & set_mask
```

Create a mask that is used to extract the collection index portion of the address. By shifting the address right by bits of the block and then AND it with the mask, we get the set number.

2. find_tag

```
def find_tag(self, address):
        return address >> (self.blockBits + self.setBits)
```

The tag field used in the cache indexing process is derived by shifting the address right, effectively discarding the bits corresponding to the cache set index and block offset.

3. find

```
def find(self, address):
        set_number = self.find_set(address)
        tag = self.find_tag(address)

        for way in range(self.ways):
            if self.metaCache[set_number][way] == tag:
                self.hit += 1

                self.move_element_to_head(self.metaCache[set_number], way)
                self.move_element_to_head(self.cache[set_number], way)
                return True

        self.miss += 1
        return False
```

The goal of this function is to search for a given address in the cache. First, I calculate the set index and tag, and then search for a matching tag in all paths to the specified set. If a match is found, it is a cache hit and it updates the LRU information. If no match is found, it is a cache miss and the function increments the miss counter.

4. load

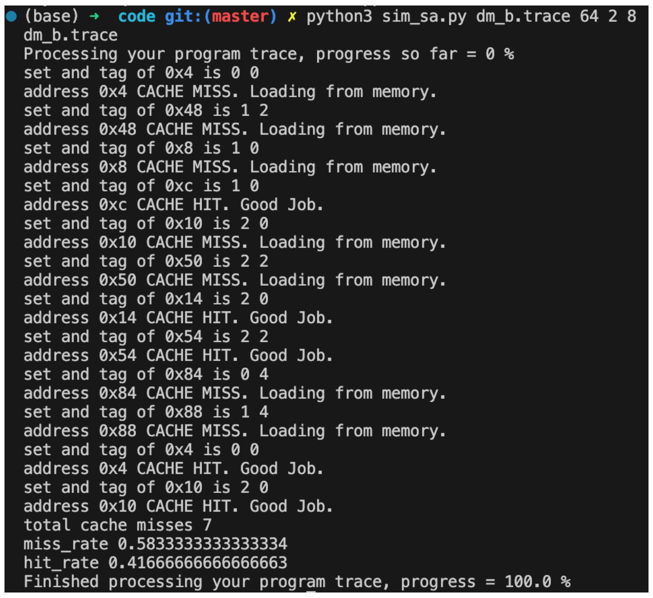
```
def load(self, address):
        set_number = self.find_set(address)
        tag = self.find_tag(address)

        use_way = -1
        for way in range(self.ways):
            if self.metaCache[set_number][way] == -1:
                use_way = way
                break

        if use_way == -1:
            use_way = self.ways – 1

        self.metaCache[set_number][use_way] = tag
        self.cache[set_number][use_way] = address

        self.move_element_to_head(self.metaCache[set_number], use_way)
```

This function integrates new data into the cache on a cache miss. First, I identify empty ways or determine the least recently used (LRU) ways to replace, then I store the tag of the address in the metadata cache and the actual address value in the data cache, ensuring that the cache reflects recent memory access patterns.

3.3 Verification

Since the warm-up question has two-word block and 8 blocks in total, total cache size is 16 * 4 with block size 4. The command line is `python3 sim_sa.py dm_b.trace 64 2 8`.

The result shows that there are 7 misses, which corresponds to the hit rate of 5/12.



```
● (base) ➜  code git:(master) ✗ python3 sim_sa.py dm_b.trace 64 2 8
  dm_b.trace
  Processing your program trace, progress so far = 0 %
  set and tag of 0x4 is 0 0
  address 0x4 CACHE MISS. Loading from memory.
  set and tag of 0x48 is 1 2
  address 0x48 CACHE MISS. Loading from memory.
  set and tag of 0x8 is 1 0
  address 0x8 CACHE MISS. Loading from memory.
  set and tag of 0xc is 1 0
  address 0xc CACHE HIT. Good Job.
  set and tag of 0x10 is 2 0
  address 0x10 CACHE MISS. Loading from memory.
  set and tag of 0x50 is 2 2
  address 0x50 CACHE MISS. Loading from memory.
  set and tag of 0x14 is 2 0
  address 0x14 CACHE HIT. Good Job.
  set and tag of 0x54 is 2 2
  address 0x54 CACHE HIT. Good Job.
  set and tag of 0x84 is 0 4
  address 0x84 CACHE MISS. Loading from memory.
  set and tag of 0x88 is 1 4
  address 0x88 CACHE MISS. Loading from memory.
  set and tag of 0x4 is 0 0
  address 0x4 CACHE HIT. Good Job.
  set and tag of 0x10 is 2 0
  address 0x10 CACHE HIT. Good Job.
  total cache misses 7
  miss_rate 0.5833333333333334
  hit_rate 0.41666666666666663
  Finished processing your program trace, progress = 100.0 %
```

4. Conclusion

Through this assignment, I have deepened my understanding and application of directmap and set associative cache, and through the actual implementation of the LRU strategy, I have thought about what information this strategy uses and how to mark the frequency of use. In the testing process after writing the code, comparing the handwritten calculation of the hit and miss index is also a very important testing step, which helps to confirm whether my implementation is correct.

5. References

1. To run all test cases with one command, I added files as follows:

1.1 Script for mac:

```
#!/bin/bash
python ../code/sim_dm.py first.trace 16 1 4
python ../code/sim_dm.py first.trace 16 1 8
python ../code/sim_dm.py pingpong.trace 16 1 4
python ../code/sim_sa.py pingpong.trace 16 2 4
python ../code/sim_sa.py test_q.trace 64 2 4
python ../code/sim_sa.py sa.trace  64 2 8
python ../code/sim_dm.py dm_a.trace 64 1 4
python ../code/sim_dm.py dm_b.trace 64 1 8
```

1.2 Script for windows:

```
python ../code/sim_dm.py first.trace 16 1 4
python ../code/sim_dm.py first.trace 16 1 8
python ../code/sim_dm.py pingpong.trace 16 1 4
python ../code/sim_sa.py pingpong.trace 16 2 4
python ../code/sim_sa.py test_q.trace 64 2 4
python ../code/sim_sa.py sa.trace  64 2 8
python ../code/sim_dm.py dm_a.trace 64 1 4
python ../code/sim_dm.py dm_b.trace 64 1 8
```

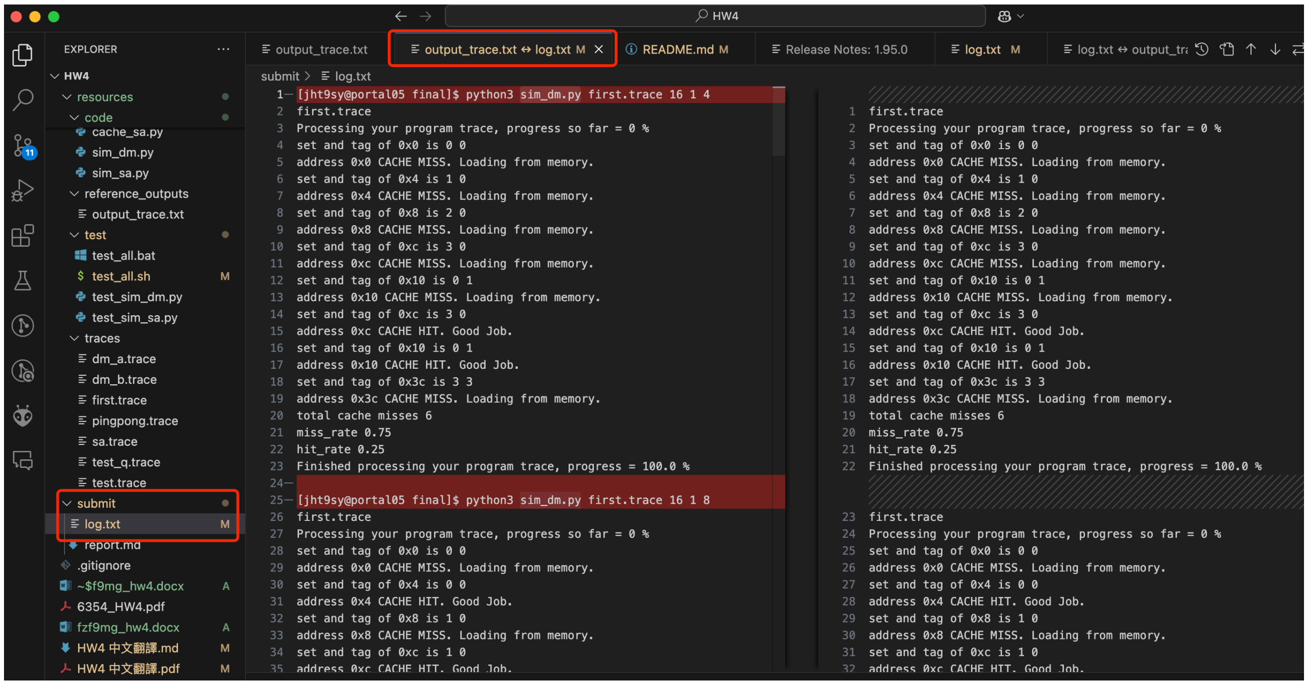
2. Run with the following command:

```
# windows
..\test\test_all.bat > ..\..\submit\log.txt
# mac
../test/test_all.sh > ../../submit/log.txt
```

3. If shows permission denied, run the following command:

```
chmod +x ../test/test_all.sh
chmod u+w ../../submit/
```

4. Automatically generates file ‘log.txt’, can be selected to compare with ‘output_trace.txt’ in VsCode.



```
[jht9sy@portal05 final]$ python3 sim_dm.py first.trace 16 1 4
first.trace
Processing your program trace, progress so far = 0 %
set and tag of 0x0 is 0 0
address 0x0 CACHE MISS. Loading from memory.
set and tag of 0x4 is 1 0
address 0x4 CACHE MISS. Loading from memory.
set and tag of 0x8 is 2 0
address 0x8 CACHE MISS. Loading from memory.
set and tag of 0xc is 3 0
address 0xc CACHE MISS. Loading from memory.
set and tag of 0x10 is 0 1
address 0x10 CACHE MISS. Loading from memory.
set and tag of 0xc is 3 0
address 0xc CACHE HIT. Good Job.
set and tag of 0x10 is 0 1
address 0x10 CACHE HIT. Good Job.
set and tag of 0x3c is 3 3
address 0x3c CACHE MISS. Loading from memory.
total cache misses 6
miss_rate 0.75
hit_rate 0.25
Finished processing your program trace, progress = 100.0 %

[jht9sy@portal05 final]$ python3 sim_dm.py first.trace 16 1 8
first.trace
Processing your program trace, progress so far = 0 %
set and tag of 0x0 is 0 0
address 0x0 CACHE MISS. Loading from memory.
set and tag of 0x4 is 0 0
address 0x4 CACHE HIT. Good Job.
set and tag of 0x8 is 1 0
address 0x8 CACHE MISS. Loading from memory.
set and tag of 0xc is 1 0
address 0xc CACHE HIT. Good Job.
```

```
first.trace
Processing your program trace, progress so far = 0 %
set and tag of 0x0 is 0 0
address 0x0 CACHE MISS. Loading from memory.
set and tag of 0x4 is 1 0
address 0x4 CACHE MISS. Loading from memory.
set and tag of 0x8 is 2 0
address 0x8 CACHE MISS. Loading from memory.
set and tag of 0xc is 3 0
address 0xc CACHE MISS. Loading from memory.
set and tag of 0x10 is 0 1
address 0x10 CACHE MISS. Loading from memory.
set and tag of 0xc is 3 0
address 0xc CACHE HIT. Good Job.
set and tag of 0x10 is 0 1
address 0x10 CACHE HIT. Good Job.
set and tag of 0x3c is 3 3
address 0x3c CACHE MISS. Loading from memory.
total cache misses 6
miss_rate 0.75
hit_rate 0.25
Finished processing your program trace, progress = 100.0 %
first.trace
Processing your program trace, progress so far = 0 %
set and tag of 0x0 is 0 0
address 0x0 CACHE MISS. Loading from memory.
set and tag of 0x4 is 0 0
address 0x4 CACHE HIT. Good Job.
set and tag of 0x8 is 1 0
address 0x8 CACHE MISS. Loading from memory.
set and tag of 0xc is 1 0
address 0xc CACHE HIT. Good Job.
```