Mengjia Li lim14 400011362

Lab Report

Part 1:

Problem 1.

main memory = 2^8 bytes

block size = 4 bytes

8 lines.

(1). tag = ? line # = ? byte # = ? for an 8-bit address?

Since block size = 4 bytes, offset = 2^2 => 2 bits

To represent 8 lines = $2^3 = 8$ => 3 bits

tag = 8 - 3 - 2 = 3 - bits

$\frac{2^8}{2^5} = 2^3 = 8$

7	5	2 0
tag	line	offset

(2). Into what line would bytes with each of the following address be stored?

000 110 11 ⇨ 110 = line 6

001 101 00 ⇨ 101 = line 5

110 100 00 ⇨ 100 = line 4

101 010 10 ⇨ 010 = line 2

$2^5 + 2^3 = 2^8$

(3). Suppose the byte with address 10100001 is stored in the cache. what are the addresses of the other bytes stored along with it?

000 → line 0

the block size = 2 bits

⇨ $\boxed{1010\ 0000}$

1010 0001

$\boxed{1010\ 0010}$

$\boxed{1010\ 0011}$

} the all bytes stored in the block 1010 0000, 1010 0010 and 1010 0011 are the other 3 bytes.

(4) How many total bytes of memory can be stored in the cache.

Cache size = (2^3 lines) × (2^2 bytes per line) = 2^5 = $\boxed{32 \text{ bytes}}$

(5) Why is tag also stored in the cache?

To distinguish between 2^3 different blocks that can be stored in the same cache line.

Mengjia Li lim14 400011362

Problem 2: following code:

```
cout << "Hello World";   (1)
cin >> a;                (2)
for (i=0; i<50; i++)     (3)
cout << i;               (4)
```

(1). Give one example of the spatial locality in the code?

spatial locality: instants near recently executed instants likely to be executed soon.

In the for loop, a[0] and a[1] will be next to eachother in the memory, so if a[i] is used, the next iteration would likely to be a[i+1]. And it is executed sequentially.

(2). Give one example of the temporal locality in the code?

Temporal locality: recently executed instants would likely to be executed again; inner loops.

cout is executed first in line 1, it is also used in line 4, which would be likely to executed again first than cin.

Part 3:

Part 3:

A[0] → address 0x0

B[0] → address 0x100

num. elements = 8 , 4-way associative

cache size = 256

block size = 16

(2) offset number = $\log_2(\text{block size}) \Rightarrow 2^4 = 16 \Rightarrow 4$ bits

set number = $\log_2\left(\frac{\text{cache size}}{\text{block size} \times \text{associative}}\right)$

$= \log_2\left(\frac{256}{16 \times 4}\right) = 4 = 2^2 \Rightarrow 2$ bits

tag = 26 − 4 − 2 = 20 bits.

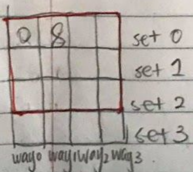
addr. 0x0 ⇒ … 0000 : 00 : 0000 } in binary

0x100 ⇒ … 0100 : 00 : 0000

tag set offset

In memory:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
0x0	0x4	0x8	0xC	0x10	0x14	0x18	0x1C
b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	b[7]
0x100	0x104	0x108	0x10C	0x110	0x114	0x118	0x1C



1 miss → tag = 0

2 miss → tag = 4

3+4 hit → tag = 0

5+6 hit → tag = 4

set 7+8 hit → tag = 0/4 (1100)

9+10 miss (10000)

11+12 hit (10100)

13+14 hit (11000)

15+16 hit (11100)

set 1

hit rate (Block size = 16) = 12 ÷ 16 = $\boxed{0.75}$

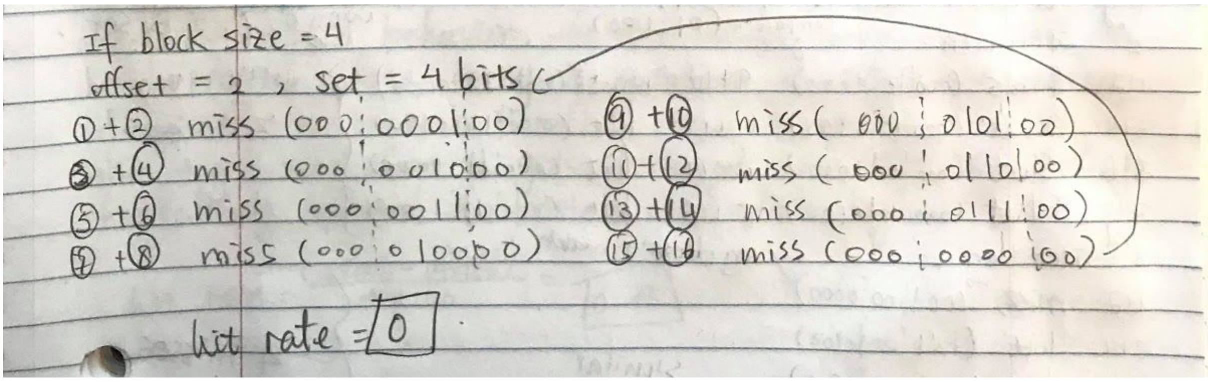
If block size = 8 : offset bit = 3 bits / set bits = 3 bits / tag = 4 bits

1+2 miss (000:000) 9+10 miss (010:000)

3+4 hit (000:100) 11+12 hit (010:100) hit rate = $\frac{8}{16}$ = $\boxed{0.5}$

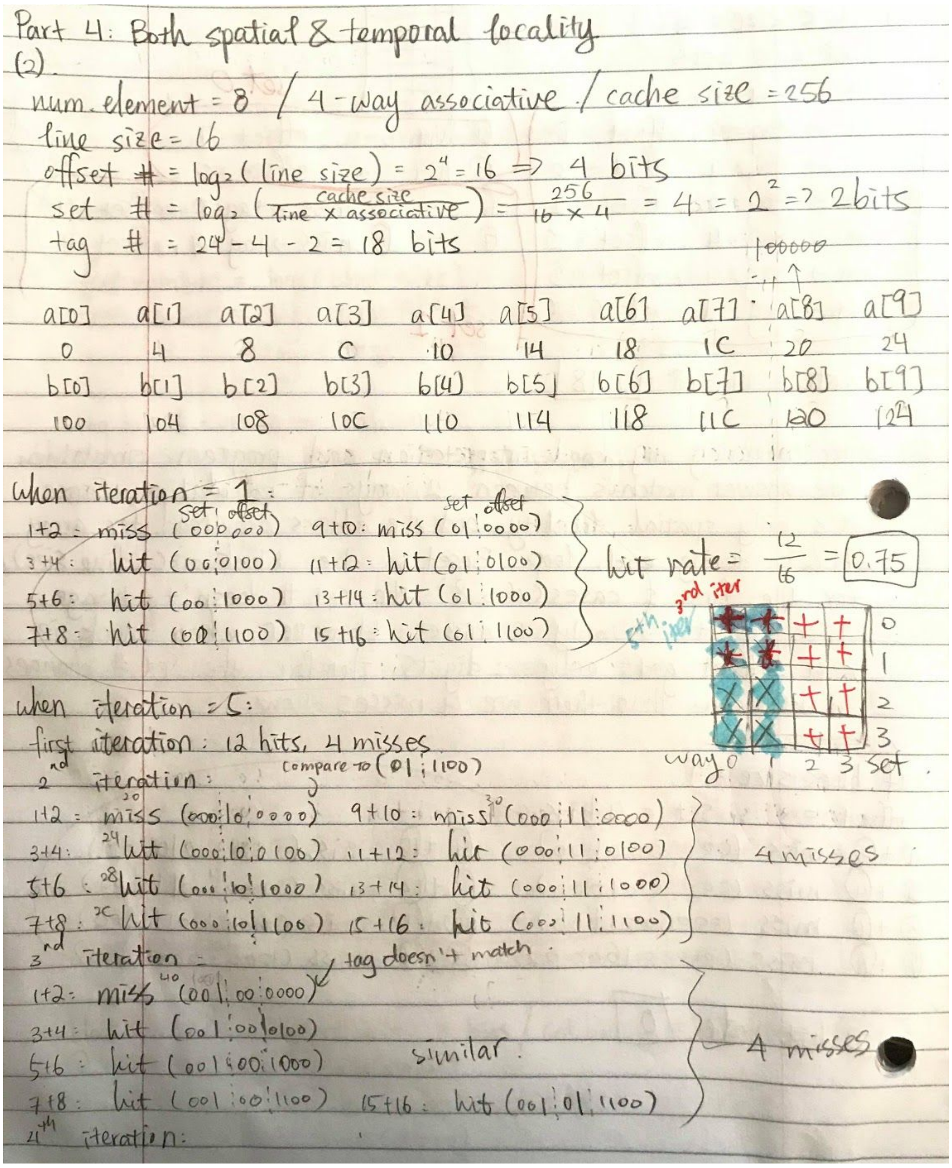
5+6 miss (001:000) 13+14 miss (011:000)

7+8 hit (001:100) 15+16 hit (011:100)


If block size = 4
offset = 2 , set = 4 bits
①+② miss (000 : 0001 : 00) ⑨+⑩ miss (000 : 0101 : 00)
③+④ miss (000 : 0010 : 00) ⑪+⑫ miss (000 : 0110 : 00)
⑤+⑥ miss (000 : 0011 : 00) ⑬+⑭ miss (000 : 0111 : 00)
⑦+⑧ miss (000 : 0100 : 00) ⑮+⑯ miss (000 : 0000 : 100)
hit rate = $\boxed{0}$.

The results between hand crafted calculation and simulator is the same. As the block size gets smaller, the offset + set digits varies, and when block size is smaller, as address changes the rate of miss is much higher compared to bigger cache block size since now it has higher chance to enter different sets.

Part 4:


Part 4: Both spatial & temporal locality
(2)
num. element = 8 / 4-way associative / cache size = 256
line size = 16
offset # = $\log_2$(line size) = $2^4 = 16 \Rightarrow$ 4 bits
set # = $\log_2\left(\frac{\text{cache size}}{\text{line} \times \text{associative}}\right) = \frac{256}{16 \times 4} = 4 = 2^2 \Rightarrow$ 2 bits
tag # = 24 − 4 − 2 = 18 bits

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
0	4	8	C	10	14	18	1C	20	24
b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	b[7]	b[8]	b[9]
100	104	108	10C	110	114	118	11C	120	124

100000

when iteration = 1 :
set offset
1+2 : miss (00 : 0000) 9+10 : miss (01 : 0000)
3+4 : hit (00 : 0100) 11+12 : hit (01 : 0100)
5+6 : hit (00 : 1000) 13+14 : hit (01 : 1000)
7+8 : hit (00 : 1100) 15+16 : hit (01 : 1100)
hit rate = $\frac{12}{16}$ = $\boxed{0.75}$

3rd iter
5th iter
set 0 1 2 3
way 0 1 2 3

when iteration = 5 :
first iteration : 12 hits, 4 misses
2nd iteration : compare to (01 ; 1100)
1+2 : miss (000 : 10 : 0000) 9+10 : miss (000 : 11 : 0000)
3+4 : hit (000 : 10 : 0100) 11+12 : hit (000 : 11 : 0100)
5+6 : hit (000 : 10 : 1000) 13+14 : hit (000 : 11 : 1000)
7+8 : hit (000 : 10 : 1100) 15+16 : hit (000 : 11 : 1100)
4 misses
3rd iteration : tag doesn't match
1+2 : miss (001 : 00 : 0000)
3+4 : hit (001 : 00 : 0100)
5+6 : hit (001 : 00 : 1000) similar
7+8 : hit (001 : 00 : 1100) 15+16 : hit (001 : 01 : 1100)
4 misses
4th iteration :

1+2 : miss (001 : 10 : 0000)
3+4 hit (001 : 10 : 0100)
...
...

9+10 : miss (001 : 11 : 0000)
11+12 : hit (001 : 11 : 0100)
...

} 4 misses

5th iteration :
1+2 : miss (010 : 00 : 0000)
3+4 : hit (010 : 00 : 0100)
...
...

9+10 : miss (010 : 01 : 0000)
11+12 : hit (010 : 01 : 0100)

} 4 misses

hit rate = $\frac{60}{80}$ = $\boxed{0.75}$

when iteration = 10
6th iteration : (now tag in set 0 way 0 is 010)
1+2 : miss (010 : 10 : 0000)
3+4 : hit (010 : 10 : 0100)
...

9+10 = miss (01 : 11 : 0000)
...

....
....

Overall hit rate (iter = 10) = $\frac{120}{160}$ = $\boxed{0.75}$

100 iteration :
according to the behavior, the tag in the cache will start to be overwritten after every 4 iteration, therefore the pattern of misses = 4 in each iteration won't be afected even if the tag & set & offset # gets overlaped, since the LRU policy has already updated the leasted used data tag.

hit rate = $\frac{(16\times100)-(4\times100)}{16\times100}$ = $\boxed{0.75}$

(5). if use uint_8 instead of int, the address increment is 1 byte instea of 4 byte, there fore each iteration only have 2 misses. The hit rate is much higher than 0.75 ($\frac{16\times100 - 2\times100}{16\times100}$) = $\boxed{0.94}$

Part 5:

Part 5:

case 1: 100 iterations, strides = 1

a[0] a[1] a[2] a[3] a[4] .... a[127]

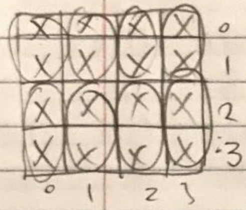
0 4 8 C 10 14 ... 1fC

a[128] a[129] a[256]

200 204 ... 3f9

iteration 1:

2 × (128 ÷ 8) = 2 × 16 = 32 misses



for 1 iteration:

- every 8 num element = 2 misses
- cache is full,

for 100 iteration:

- tag & set & offset never gets overlaped

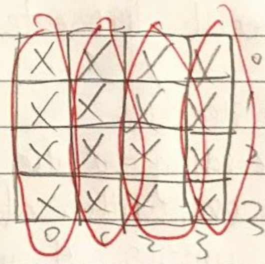
hit rate = $\frac{(128 \times 100) - (32 \times 100)}{128 \times 100} = \boxed{0.75}$

same applies to 10 iterations & 50 iterations.

case 2: 100 iterations, strides = 2

a[0] a[2] a[4] a[6] a[8] ...

0 +8 8 10 18 20 ...



1: miss (00:00:0000) 5: miss (00:10:0000)

2: hit (00:00:1000) 6: hit (00:10:1000)

3: miss (00:01:0000) 7: miss (00:11:0000)

4: hit (00:01:1000) 8: hit (00:11:1000)

9. miss (01:00:0000) ...

...

65: miss (1000:00:0000)

each iteration: 4 misses × 16 = 64 misses

100 iterations: $\frac{(128 \times 100) - (64 \times 100)}{128 \times 100} = \boxed{0.5}$

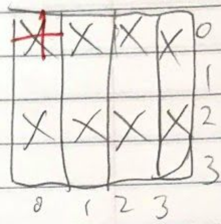
- LRU cache table starts overwrite every 64 elements.

strides = 8

a[0] a[8] a[16] a[24] a[32] ...

0 20 40 60 80 ...

① miss (000 00 0000)
② miss (000 10 0000)
③ miss (001 00 0000)
④ miss (001 10 0000)
⑨ miss (100 00 0000)
⑤ miss (010 00 0000)
⑥ miss (010 10 0000)
⑦ miss (011 00 0000)
⑧ miss (011 10 0000)
...



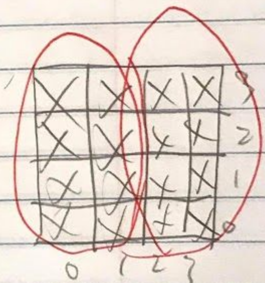
hit rate = 0 for 100 iteration, as well as 10/50 iterations

strides = 4

a[0] a[4] a[8] a[16] a[20] ...

0 10 20 30 40 ...

① miss (000 00 0000)
② miss (000 01 0000)
③ miss (000 10 0000)
④ miss (000 11 0000)
⑤ miss (001 00 0000)

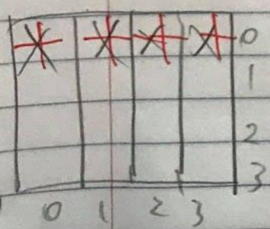


hit rate = 0 for 100 iterations, as well as 10/50 iterations

case 16:

a[0] a[16] a[32] ...

0 40 80 ...



hit rate = 0 !!!
so inefficient !!!

since here it goes spatial first, then temporal, very bad.

strides number is a very inefficient way, it wastes cache memory ↑ as strides ↑

case 21 & 64 = even worse ...

Part 6:

Case 1: direct mapped

offset digit = log2(line_size) = 4 bits

set digit = log2(256/(16 x 1)) = 4 bits

tag digit = 24 - 4 - 4 = 16 bits

number of sets = 256 / (16 * 1) = 16

1: miss 000 0000 0000

2: hit 000 0000 0100

3: hit 000 0000 1000

4: hit 000 0000 1100
5: miss 000 0001 0000
6: hit 000 0001 0100
7: hit 000 0001 1000
8: hit 000 0001 1100
9: miss 000 0010 0000
10: hit 000 0010 0100
11: hit 000 0010 1000
12: hit 000 0010 1100
13: miss 000 0011 0000
14: hit 000 0011 0100
15: hit 000 0011 1000
16: hit 000 0011 1100
...
For every 8 elements, 2 misses happens, for every 128 elements, 32 misses happens, therefore:
$hit\ rate = \frac{(128\times100-(32\times100))}{128\times100} = 0.75$
The final cache content would all be tag = 199.

Case 2: 2-way associative
offset digit = log2(line_size) = 4 bits
set digit = log2(256/(16 x 2)) = 3 bits
tag digit = 24 - 3 - 4 = 17 bits
number of sets = 256 / (16 * 2) = 8
1: miss 0000 000 0000
2: hit 0000 000 0100
3: hit 0000 000 1000
4: hit 0000 000 1100
5: miss 0000 001 0000
6: hit 0000 001 0100
7: hit 0000 001 1000
8: hit 0000 001 1100
9: miss 0000 010 0000
10: hit 0000 010 0100
11: hit 0000 010 1000
12: hit 0000 010 1100
13: miss 0000 011 0000
14: hit 0000 011 0100
15: hit 0000 011 1000
16: hit 0000 011 1100
17: miss 0000 100 0000
...

For every 8 elements, 2 misses happens, for every 128 elements, 32 misses happens, therefore:

$$hit\ rate = \frac{(128\times100-(32\times100)}{128\times100} = 0.75$$

The final cache content would all be tag = 399 (00 01100 01111).

Case 3: 4-way associative
offset digit = log2(line_size) = 4 bits
set digit = log2(256/(16 x 4)) = 2 bits
tag digit = 24 - 2 - 4 = 18 bits
number of sets = 256 / (16 * 4) = 4

1: miss 00000 00 0000
2: hit 00000 00 0100
3: hit 00000 00 1000
4: hit 00000 00 1100
5: miss 00000 01 0000
6: hit 00000 01 0100
7: hit 00000 01 1000
8: hit 00000 01 1100
9: miss 00000 10 0000
10: hit 00000 10 0100
11: hit 00000 10 1000
12: hit 00000 10 1100
13: miss 00000 11 0000
14: hit 00000 11 0100
15: hit 00000 11 1000
16: hit 00000 11 1100
17: miss 00001 00 0000 (different tag)
...

For every 8 elements, 2 misses happens, for every 128 elements, 32 misses happens, therefore:

$$hit\ rate = \frac{(128\times100-(32\times100)}{128\times100} = 0.75$$

The final cache content would all be tag = 799 (00 11000 11111).

Case 4: fully associative
offset digit = log2(line_size) = 4 bits
set digit = log2(256/(16 x 16)) = 0 bits
tag digit = 24 - 2 - 4 = 18 bits
number of sets = 1

1: miss 0000000 0000
2: hit 0000000 0100
3: hit 0000000 1000
4: hit 0000000 1100
5: miss 0000001 0000

6: hit 0000001 0100
7: hit 0000001 1000
8: hit 0000001 1100
9: miss 0000010 0000
10: hit 0000010 0100
11: hit 0000010 1000
12: hit 0000010 1100
13: miss 0000011 0000
14: hit 0000011 0100
15: hit 0000011 1000
16: hit 0000011 1100
17: miss 0000100 0000 (different tag)
...
For every 8 elements, 2 misses happens, for every 128 elements, 32 misses happens, therefore:

$$hit\ rate = \frac{(128 \times 100 - (32 \times 100)}{128 \times 100} = 0.75$$

The final cache content would all be tag = 3199 (11 00011 11111).