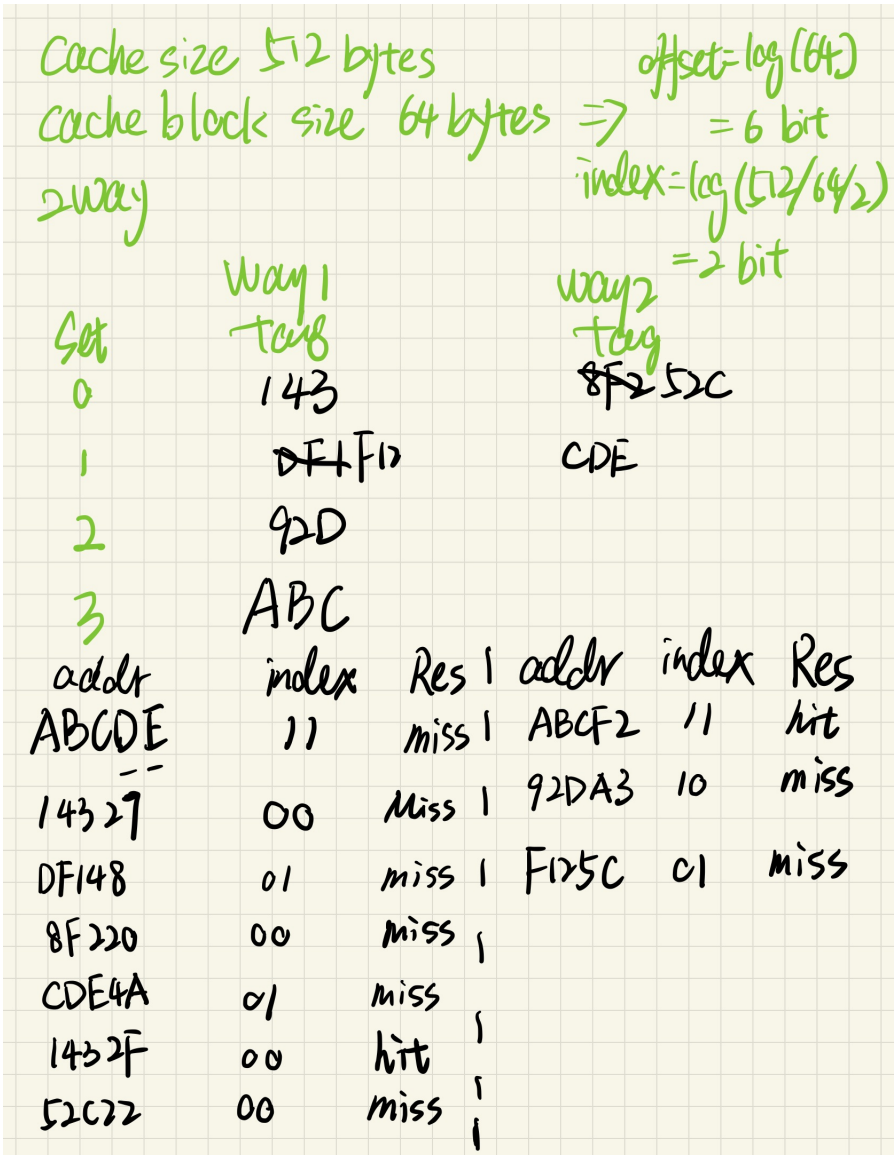
ECE565 Homework 2

Chengke Tang

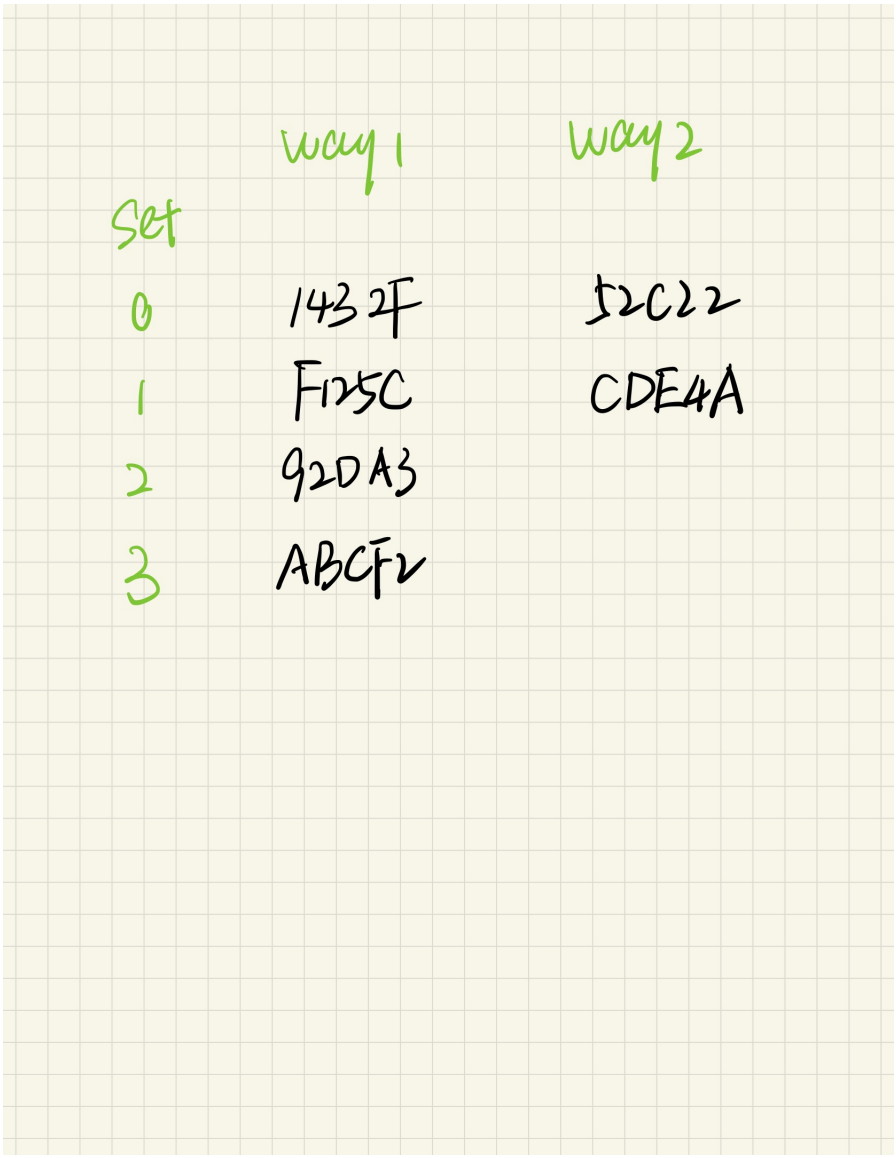
October 2022

1 Question 1

Result for each process:



Result for final contents:



2 Question 2

a) $1 + (3\% * (15 + (30\% * 300))) = 4.15$

b) $1 + (10\% * (15 + (5\% * 300))) = 4$

3 Question 3

(b)(c) According to lscpu, the cache size for Data is 128 KiB, which means L1 cache could hold 128 * 1024 / 8 = 16384 numbers. The program would take two command line arguments. The first is the size of the array, the second is the number of traversals. To test L1 cache, we could run

```
./1 8000 10000
```

where 8000 is the number of elements in array and 1000 is for traversals ./1 is for write only. ./2 is for 1:1 read-write. ./3 is for 2:1 read-write. The code for achieving write traffic only is the following:

```
clock_gettime(CLOCK_MONOTONIC, &start_time);
for (i=0; i<num_traversals; i++) {
   for (uint64_t j = 0;j<num_elements;j++)
     {
       array[j] = j;

     }
 }
 clock_gettime(CLOCK_MONOTONIC, &end_time);
 double elapsed_ns = calc_time(start_time, end_time);
 printf("bandwidth = %f GB/s\n",
 (((uint64_t)num_elements * (uint64_t)num_traversals * 8))
 /elapsed_ns);
```

it will write current index to array. The code for 1:1 read-to-write ratio is the following:

```
 uint64_t temp1 = 0;
 clock_gettime(CLOCK_MONOTONIC, &start_time);
 for (i=0; i<num_traversals; i++) {
   for (uint64_t j = 0;j<num_elements;j++)
     {
      // b.
       array[j] = j;
       temp1 = array[j];
     }
 }
```

```
clock_gettime(CLOCK_MONOTONIC, &end_time);
double elapsed_ns = calc_time(start_time, end_time);
printf("bandwidth = %f GB/s\n",
(((uint64_t)num_elements * (uint64_t)num_traversals * 8*2))
/elapsed_ns);
```

It will write index to array and then load the array elements to a new variable. The following is the code for 2:1 read-to-write ratio:

```
uint64_t temp1 = 0;
uint64_t temp2 = 0;
clock_gettime(CLOCK_MONOTONIC, &start_time);
for (i=0; i<num_traversals; i++) {
  for (uint64_t j = 0;j<num_elements;j++)
    {
          temp1 = array[j];
          array[j] = j;
          temp2 = array[j];
    }
}
clock_gettime(CLOCK_MONOTONIC, &end_time);

double elapsed_ns = calc_time(start_time, end_time);
printf("bandwidth = %f GB/s\n",
(((uint64_t)num_elements * (uint64_t)num_traversals * 8*3))
/elapsed_ns);
```

The code will first load array elements to a variable, then write to array, then load the array elements again.

The following is the result:

write only	1:1	2:1
23.04	50.76	70.71

(d) The L3 cache size for this virtual machine is 143MiB. So it could hold 2342912 numbers. To test this, I used 2500000. I ran the following command:

```
./1 2500000 1000
```

The following is the result:

write only	1:1	2:1
17.53	33.68	50.94

From the above data, we could conclude that more read instruction has more bandwidth than write instruction. Also, the more hierarchy of cache we tried to access, the smaller the bandwidth is.

4 Question 4

(a)(b) To run the program, type "./ikj 1" for i-j-k testing. Type "./ikj 2" for testing of i-k-j. Type "./ikj 3" for testing of j-k-i. Type "./ikj 4" for testing of i-j-k tiling. The following is the result I got for testing:

i-j-k	i-k-j	j-k-i
6.70	3.03	17.11

The results meet my expectation because misses per iteration i-k-j<i-j-k<j-k-i, which leads to this run time.

(c)(d) My L2 cache size is 4MiB. And my cache block size is 64bytes. I tested with several block sizes. 64 could produce best performance. It is faster than original i - j - k, but still slower than i-k-j. Because function tiling could improve the performance by reducing the missing rate.