Problem 1

- Since it is a 2-way set associative cache, each set is 64B * 2 = 128B, and therefore there is 512B / 128B = 4 sets. The tag would be 20 - log2(64) + log2(4) = 12 bits.

Addresses	Tag	Index	Hit or Miss
ABCDE	ABC	11	M
14327	143	00	M
DF148	DF1	01	M
8F220	8F2	00	M
CDE4A	CDE	01	M
1432F	143	00	H
52C22	52C	00	M
ABCF2	ABC	11	H
92DA3	92D	10	M
F125C	F12	01	M

- Result contents of the cache

	Way 0	Way 1
Set 0	1432F	52C22
Set 1	F125C	CDE4A
Set 2	92DA3	
Set 3	ABCF2	

Problem 2

(a)

- AAT = 1 + 3% * (15 + 30% * 300) = 4.15 CPU cycles

(b)

- AAT = 1 + 10% * (15 + 5% * 300) = 4 CPU cycles

Problem 3

(b,c)

- I wrote 3 programs in total for different access patterns. They all take 2 arguments to run. The first one is the size of the uint_64 array and the second one is the travesal times.

Write only code

```
for(int i = 0;i<num_traversals;i++){
    for (uint64_t j = 0; j < num_elements; j++) {
        array[j] = j;
    }
}
printf("Bandwidth = %f GB/s\n", ((uint64_t)num_elements * num_traversals * 8) /
elapsed_ns);
```

1:1 read:Write code

```
uint64_t last_one;
for(int i = 0;i<num_traversals;i++){
    for (uint64_t j = 0; j < num_elements; j++) {
        array[j] = j;
        last_one = array[j];
    }
}
printf("the last one in array is %ld\n",last_one);
printf("Bandwidth = %f GB/s\n", ((uint64_t)num_elements * num_traversals * 8 * 2) /
elapsed_ns);
```

2:1 read:Write code

```
uint64_t curr_one, last_one;
for(int i = 0;i<num_traversals;i++){
    for (uint64_t j = 0; j < num_elements; j++) {
        curr_one = array[j];
        array[j] = j;
        last_one = array[j];
    }
}
printf("the curr one in array is %ld\n",curr_one);
printf("the last one in array is %ld\n",last_one);
printf("Bandwidth = %f GB/s\n", ((uint64_t)num_elements * num_traversals * 8 * 3) /
elapsed_ns);
```

- For the write-only program, the code is rather simple, just write values into the array. For the ones that involve reading from cache, I use variables to store the result of read, and print it at the end to ensure the compiler actually did the read.
- Since I was using uint_64 array to test, each number would take up 64 bits, and the L1 cache of the VM I used was 32K, which means the number of numbers should not exceed 32 * 1024 * 8 / 64 = 4096. I decided to use 2048 for this test. And I used 1000000 as the traversal time. The results are shown below.

	Only Write	**1:1 Read:Write**	**2:1 Read:Write**
Bandwidth (GB/s)	34.110478	67.705091	92.526809

- The results show that the higher portion the read takes, the higher the bandwidth is, which matches my expectation because read takes less time than write.

(d)

- The L3 cache on my VM was 30720K, correspond to the number of numbers 30720 * 1024 * 8 / 64 = 3,932,160. I used 4,000,000 for this test, and reduced the traversal time to 1000. The results are shown below.

	Only Write	**1:1 Read:Write**	**2:1 Read:Write**
Bandwidth (GB/s)	10.311625	20.238326	32.780958

- The results show that the bandwidth for same access pattern is lower than the results in *(b,c)*, which matches my expectation because the lower hierarchy we access, the higher the latency will be, therefore the bandwidth will be lower.

Problem 4

(a)

- run the program matrix with ./matrix num, where num can be 1,2,3,4, corresponding to the i-j-k, the k-j-i, the j-k-i and the tiled i-j-k.

(b)

- The results are shown below.

	i-j-k	**k-i-j**	**j-k-i**
Time elapsed (s)	1.668811	0.498872	22.089819

- The results match expectations. The reason is that C store data in a row-major manner, and j-k-i will result in the most cache misses.

(c,d)

- The L2 cache on my VM was 256K. And i experiment with block size 32, 64, 128. Block size 64 resulted in the best performance with 1.553059s and is faster than the original i-j-k and j-k-i, but still slower than k-i-j. The reason for this is that although it reduce some miss, L2 cache is inherently slower than L1 cache.