Computer Architecture
Assignment 4 - Simulating the Cache

As always it is important you adhere to the following guidelines:

(**1**) Work individually;

(**2**) The submission date is January 5, 23:59;

(**3**) Submission is via the "submit" system;

(**4**) This assignment should be written in C and not Assembly!

(**5**) Ensure that your submission compiles and runs without errors or warnings on BIU's servers before submitting. Failure to do so will result in docked points;

(**6**) At the beginning of every file you submit, add your ID and name in a comment. For example: `/* 123456789 Israel Israeli */`;

(**7**) It is forbidden to use AI tools like ChatGPT when writing your homework. Doing so is tantamount to cheating, and will be treated as such;

(**8**) You should use the given general main and print_cache functions in order to be compatabile with the auto checking;

(**9**) You can (and should) assume that the input is valid in terms of read and write requests (addresses are valid) and that s+t+b is indeed the number of bits that are needed for addressing all the data in the RAM.

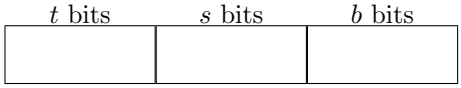
In this assignment, you are asked to write code which simulates how a cache operates. To do so, you will need to define the two following structs, which will be utilized to represent cache sets and the cache itself respectively:

```
typedef unsigned char uchar;

typedef struct cache_line_s {
    uchar valid;
    uchar frequency;
    long int tag;
    uchar* block;
} cache_line_t;

typedef struct cache_s {
    uchar s;
    uchar t;
    uchar b;
    uchar E;
    cache_line_t** cache;
} cache_t;
```

Then you will need to define the following three functions:

```
cache_t initialize_cache(uchar s, uchar t, uchar b, uchar E);
uchar read_byte(cache_t cache, uchar* start, long int off);
void write_byte(cache_t cache, uchar* start, long int off, uchar new);
```

The type `cache_t` contains in it four parameters: s, t, b, E which are defined as in the book: $S = 2^s$ is the number of sets, $B = 2^b$ is the number of blocks per line in the set, E is the number of lines per set, and t is the tag length. Recall that given an address of length $m = s + t + b$, partition the bits into



The `cache` field is an array of array of cache lines. Each array of cache lines can be thought of as a cache set, and so `cache` is simply an array of sets. Each `cache_line_t` has a valid bit (ignore the fact that it is of course actually a byte), its frequency, its tag (which should be t bits long, but we made it a long to make your (my) life easier), and the block of memory.

The cache you implement should replace lines using the LFU (least frequently used) method, meaning if it tries to read in memory to a set with no available lines then it replaces the line whose frequency is minimal. If two lines both have the same minimal frequency, then the first line is chosen (ie. if line 3 and line 10 both have frequency 1 which is the smallest, then line 3 is chosen).

The `read_byte` and `write_byte` functions accept as inputs `start` as well as `off`. You should act as if `start` is the zero address and `off` is the address that you insert into your cache. More specifically, insert the contents of `start[off]` into your cache, while using the bits of `off` as your offset.

Your cache is write-through, meaning that when `write_byte` is called, it writes `new` both to the cache and to memory.

The `read_byte` function should return eventually the asked byte from the cache, after making sure that the cache is updated as we saw in class.

We also provide you with the `print_cache` function (that you must use, for proper formatting), so you don't need to worry about whitespace errors But fair warning: it's going to paste weirdly into your text editor, sorry. You win some, you lose some.

```
1  void print_cache(cache_t cache) {
2     int S = 1 << cache.s;
3     int B = 1 << cache.b;
4
5     for (int i = 0; i < S; i++) {
6        printf("Set %d\n", i);
7        for (int j = 0; j < cache.E; j++) {
8           printf("%1d %d 0x%0*lx ", cache.cache[i][j].valid,
9              cache.cache[i][j].frequency, cache.t, cache.cache[i][j].tag);
10          for (int k = 0; k < B; k++) {
11             printf("%02x ", cache.cache[i][j].block[k]);
12          }
13          puts("");
14       }
15    }
16 }
```

So for example,

```
1  int main() {
2     uchar arr[] = {1, 2, 3, 4, 5, 6, 7, 8};
3     cache_t cache = initialize_cache(1, 1, 1, 2);
4     read_byte(cache, arr, 0);
5     read_byte(cache, arr, 1);
6     read_byte(cache, arr, 2);
7     read_byte(cache, arr, 6);
8     read_byte(cache, arr, 7);
9     print_cache(cache);
10 }
```

Should print

```
Set 0
1 2 0x0 01 02
0 0 0x0 00 00
Set 1
1 1 0x0 03 04
1 2 0x1 07 08
```

We supply you here your main function (that must be included in your final code!). It simply gets as input the size of your data (which simulates RAM), the actual data to store in "RAM", the cache parameters, and then the bytes it should read (which it reads until it gets a negative value):

```c
int main() {
   int n;
   printf("Size of data: ");
   scanf("%d", &n);
   uchar* mem = malloc(n);
   printf("Input data >> ");
   for (int i = 0; i < n; i++)
      scanf("%hhd", mem + i);

   int s, t, b, E;
   printf("s t b E: ");
   scanf("%d %d %d %d", &s, &t, &b, &E);
   cache_t cache = initialize_cache(s, t, b, E);

   while (1) {
      scanf("%d", &n);
      if (n < 0) break;
      read_byte(cache, mem, n);
   }

   puts("");
   print_cache(cache);

   free(mem);
}
```

So for example, the previous example can be equivalently run like so:

```
Size of data: 8
Input data >> 1 2 3 4 5 6 7 8
s t b E: 1 1 1 2
0 1 2 6 7 -1

Set 0
1 2 0x0 01 02
0 0 0x0 00 00
Set 1
1 1 0x0 03 04
1 2 0x1 07 08
```

What to Submit

You must submit using the `submit` system all your code plus a makefile that compiles your code to an executable named `cache`. Notice that the files containing your code can be named whatever you like, just make sure that the result of the makefile is named as mentioned and is compiled without any warnings or errors!