The purpose of this lab is to give you experience programming in C and to reinforce what you learned in ECE 232 about cache organization.

Prepare for this lab by reading the coverage in Hennessy and Patterson of set-associative caches. Recall that the key parameters of such a cache are its line (also called the block) length (in bytes), L, the set associativity (which is the number of lines per set), K, and the total cache size, C. An important measure of cache performance is the miss rate, which is the fraction of memory accesses that are not found in the cache.

Modern processors contain multiple levels of cache, although we will only consider one level here.

Write a simulator in C for set-associative caches under the LRU replacement strategy. The simulator takes as input the cache configuration and reads a trace which is a sequence of memory addresses. It then outputs the cache miss rate.

Your simulator should include the following functions in addition to the main() function:

- int whichSet(...): Outputs the cache set in which the address falls.
- int setIndexLength(...): Outputs the number of bits in the set index field of the address.
- int offsetLength(...): Outputs the number of bits in the line offset field of the address.
- int tagBits(...): Outputs the tag bits associated with the address.
- hitWay(...): If there is a hit, this outputs the cache way in which the accessed line can be found; it returns -1 if there is a cache miss.
- updateOnHit(...): Updates the tagArray and lruArray upon a hit. This function is only called on a cache hit.
- updateOnMiss(...): Updates the tagArray and lruArray upon a miss. This function is only called on a cache miss.

You should thoroughly test each of these functions separately after writing them: do not wait for the entire program to be put together before starting the testing process. The additional
time spent in writing the test setup code is well worth it; it will likely save you debug time in the long run.

The inputs to these functions (indicated here by ...) should be defined by you as appropriate. The LRU status of a line in a set should be captured by a number which indicates the relative age of that line with respect to the lines in that set. Thus, the relative age of the most recently accessed line will be 0. Invalid lines are indicated by an LRU status of $-1$. Initialize the cache to be entirely invalid at the beginning of the simulation.

When programming, keep the following two facts in mind. First, the C standard does not specify whether the $\gg$ symbol indicates an arithmetic or logical right shift. For this reason, anything to be right-shifted must be defined as an unsigned int or masked to get rid of any spurious sign-extending 1s. Second, do not use the $\log_2()$ function to get the address field length: you should write your own code for this (remembering that lengths of the line and set-index fields are always powers of 2).

The state of the cache will be stored in two dynamically sized arrays: tagArray and lruArray. In particular, tagArray[i][j] and lruArray[i][j] hold the tag and LRU status of the line in set $i$, way $j$. To give you experience with pointers and malloc(), these should be implemented using pointers (scheduled to be covered in Lecture 3 or 4). In particular, declare unsigned int **tagArray and int **lruArray and use malloc() appropriately. (When we get a little further in this class, we will cover C structs which will allow us to group both these items as fields in another structure; for now, you don’t need structs.)

When you write your code, keep readability foremost in mind. Part of the grade will depend on how easy it is to follow your logic. Pay attention to how you lay out the code as well as how clear the variable names are. Variable names should be chosen so that it is easy to figure out what they stand for. For example, using tfl as tag field length is not good; instead, use tagFieldLength. This makes for somewhat more typing but is much clearer.

Keep your functions short to the extent possible. Program using assertions to ensure that properties you know should hold at key points in the program actually do so.

Your program should read the trace line by line and process it before moving on to the next entry in the trace. Do NOT read the entire trace into some big array before proceeding: traces in real life are typically billions of addresses long.

**Input and Output Formats:** Please be sure to follow this format since otherwise your code will not test properly when our grading script tries to execute it.

In the main function, you should define the argv[] array so that argv[1] specifies the set associativity (no of lines in a set), argv[2] specifies the line size in bytes, and argv[3]
specifies the total cache size in kBytes (NOT bytes). So, if your object file is called a.out, you should be able to run it with the command line `a.out K L C traceFile` where K, L, C are the associativity, line size and cache size, respectively. `traceFile` is the name of the trace file.

Your output should print the following data on a single line: `trace_file_name C K L miss rate`. So, for example, the output `traceX 512 8 16 0.15` means that traceX running on a cache of size 512 kBytes with \( K = 8 \) lines per set and each line being \( L = 16 \) bytes long yielded a miss rate of 0.15.

**Test on Quark:** When grading, the TA will test your code using the gcc compiler in the Unix quark machine in ECS, so all your code should be tested and debugged in that framework before submission. Do NOT expect C code to be perfectly portable between platforms. Your code should not require any C99 or similar flags to be set in order to run properly.

You should allocate enough time and effort to thoroughly test the code prior to submission. Create some short test traces for which you know the miss rate. Check also if the trends are what you would expect (e.g., impact of miss rate on increasing the cache size). You can also use the sample trace on the lab website: test it with for line sizes of 4, 8, 16, 32, 64, 128, 256 and set associativities 1, 2, 4 for cache sizes ranging from 4 KB to 512 KB. Check any trends you see with what you would expect (from what you know from your ECE 232 cache lectures). (There is no need to submit results: we will obtain results by running your code.)

**Submission:** Submit the following items:

- Via Moodle, submit a filled-in summary sheet.
- On quark, submit your fully documented code. The submission procedure for code can be found on Moodle.