The purpose of this lab is to give you some experience programming in C and to reinforce what you learned in ECE 232 about cache organization. In particular, you will get some basic exposure to the use of pointers to set up dynamically-sized arrays.

You will write a simulator in C for set-associative caches under the LRU replacement strategy. The inputs to the simulator will be the cache configuration (the line length, \( L \), the number of lines per set, \( K \), and the total cache size, \( C \)). The number of sets must be computed from these inputs. This simulator will be run on the traces available from the Moodle course site. Each entry in the traces consists of a memory address indicating the virtual address of what is being accessed. For our purposes, treat this virtual address as a physical address; do not worry about the virtual-to-physical translation that you learned about in ECE 232 (using page tables). The address space is 32 bits long.

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

The inputs to these functions (indicated here by ...) should be defined by you as appropriate. LRU status should be captured by a number: if this number is non-negative, it reflects the relative age of the lines at last access. That is, an LRU status of 0 indicates this is the most recently accessed line in that set. Relative age is in relation to the lines within each set, not across sets. Thus, the age of a valid line can range from 0 (most recently accessed line in its set) to \( K - 1 \) (least recently accessed line in its set) where \( K \) is the set associativity. Invalid lines are indicated by an LRU status of \(-1\). Initially, the entire `tag array` will be invalid.

Keep the following two facts in mind. First, the C standard does NOT specify whether \( \gg \) indicates an arithmetic or logical right shift; it is up to the compiler-writer as to what this is. For this reason, anything that is 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 \( \log2() \) function to get the address field lengths: this is specified as a double-precision function and is not guaranteed to return perfectly accurate results for integer powers of 2. (Every installation that I have checked does return accurate results; however, this is not guaranteed and so you cannot rely on this.)

The state of the cache will be stored in two dynamically sized arrays: `tagArray` and `lruArray`. `tagArray[i][j]` and `lruArray[i][j]` holds the tag and LRU status of the line in set `i` way `j`. To give you some experience with pointers and the `malloc()` function, these should be implemented using
pointers. In particular, declare `unsigned int **tagArray` and `int **lruArray` and use `malloc()` appropriately. This will be discussed in the lecture. (When we go a little further in this class, you will come across C structs, which allow us to group both `tagArray` and `lruArray` as fields in another structure; we will not be using structs in this lab. If you have extensive prior C experience, you may use them; if you have never seen structs before, do not attempt to use them for Lab A; leave them for Lab C.)

When you write the code, keep readability foremost in your mind. Part of the grade you receive will depend on how easy it is for me to follow your logic. This means that you should pay attention to how you lay out the code as well as how clear the variable names are. Variable names should NOT be cryptic; it should be reasonably clear from a glance what a variable stands for. For example, using `tfl` as tag field length is confusing; use something more descriptive like `tagFieldLength` instead. It will take a little more time to type out, but that is time well spent. Your code should be its own best documentation. This does not mean that you should ignore the need to add comments to describe the flow of logic, just that the comments should not be the sole means of following the code. Furthermore, keep your functions short to the extent possible; don’t cram huge amounts of code into individual functions.

Also, program defensively. Use assertions plentifully to ensure that the properties you know should hold at key points in the program actually do so. Assertions are extremely important since they permit the system to keep checking correctness as the program runs. It is very easy to create incorrect programs with subtle logic errors, that do not crash or produce obviously wrong results; these are very difficult to catch and assertions go a long way in detecting them.

You are free to develop your code in any environment you choose. I usually see a wide variety of environments being used in this course. However, since I will be running your code using the gcc compiler in the Unix `quark` machine in ECS, make sure that any portability issues are ironed out and that the program does indeed run correctly there, in that environment. (All ECE students should have an ECS account.)

Submit the following items by September 19, 2014.

- Via Moodle, submit a filled-in summary sheet summarizing the functional status of your program and providing other relevant information.
- On Quark, submit your fully documented code. The documentation and organization of the code should be sufficient for me to readily understand the flow of logic. Points will be deducted for obscure code. The submission procedure for quark will be posted by September 21.
- Via Moodle, submit your results, if you have a fully functional program. The report should include two plots.
  - The first plot will show the miss rate (y-axis) as a function of the cache size; the cache sizes studied should be $2^n$ bytes, where $n = 7, 8, \ldots, 20$. Select $K = 2, L = 16$. The x-axis should be linear in $n$ (and therefore logarithmic in the cache size). It will have five curves, one for each of the five traces provided.
  - The second plot will show the miss rate as a function of the cache associativity, $K$. For a cache size of 32 KBytes and $L = 32$, plot the miss rate against $K$ where $K$ takes the values $1, 2, 4, 8, 16, 32, 64$. The x-axis will be linear in $\ell$, where $K = 2^\ell$. Once again, your plot will have five curves, one for each of the five traces provided.
- If your program is not fully functional on `quark`: Provide a detailed description of where it fails. List everything that works correctly and where you believe the problem to be. Indicate the steps you took to identify the problem.