For this assignment, you will be writing a small Matlab program to implement an LCP-search tool. The tool will:

- use a library routine to create a model of an integrated circuit with several random critical paths.
- run tests (using another library routine) with whatever LCPs pushed forwards and backwards that you choose.
- once you've decided where the critical paths are in the chip, report them to another library routine that tells you if you got them correct

You can get the file /comp/150CAD/public_html/HWs/code/lcp_main.m to get you started. It is the only file you need. It contains the main function lcp_main() that you can call from the Matlab command window. It also contains skeletons of various function for you to fill in yourself, and also contains utility functions that you can use.

The library routines are described in lcp_main. They use the following data structures and constants:

- N_LCPS = 32. An LCP is simply modeled as an integer; e.g., in this setup there are 32 LCPs, numbered from 0 through 31.
- struct ('D', number, 'R', number);. A critical path is simply a driver/receiver pair. So, e.g., we might have a path that starts at a clock controlled by LCP #3 and ends at a clock controlled by LCP #7. In this case, it would be the structure struct ('D', 3, 'R', 7). We call this structure a Path.

The function lcp_main (n_paths, seed), which is supplied for you, first calls create_random_critical_paths(*n_paths*, *seed*) to create a random collection of critical paths (i.e., a model of the paths on a real chip), so that you can try to find them. Of course, it doesn't tell you which paths it created! *N_paths* controls how many different critical paths it creates. *Seed* is used to seed the random-number generator so that you can get repeatability for easier debugging of your code.

It then calls LCP_search() to run an LCP search and find the paths that were created. A skeleton of this function (and other functions that it calls) are supplied for you; you must flesh them out.

Finally, lcp_main() calls check_results (*failures*) to check the paths that LCP_search() found. It will return true if you got them correct, and also print error messages telling you what was wrong otherwise.

The following routines are also supplied for you to use:

- set_passing (*forw*, *back*). You supply the LCPs that you want to push forwards (making the clocks they controlarrive later) and those you want to pull back (making the clocks they control arrive earlier). The library examines these, computes how they stress the critical paths it has created, and thus decides how fast the chip can run with these LCP settings. It then virtually adjusts the chip's clock period so that tests with this LCP recipe will subsequently pass.
- run_test_OK (*forw*, *back*). Run a test; return true if it passes. By "passes," we mean that the given LCP recipe stresses the chip no more than the recipe you previously gave to set_passing(). If you stress the chip worse than that, the test will be deemed to fail and run_test_OK() will return false.
- printLCPvec (LV); printPath (p); printPaths(paths). These function provide support to print data structures in case it helps you for debugging your code. They all return a string that represents the given data.

The file lcp.m provides skeletons of the functions for you to write; your job is to fill them in. When you're done, turn in just the file lcp.m via the Provide interface.