For this assignment, you will be writing a small C++ 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 following files from /comp/150CAD/public_html/HWs/code:

- lcp.cxx: contains a main() function and skeletons of the functions you are to write.
- lcp_utils.cxx: contains library routines to help you.
- lcp.hxx: an include file defining functions and data types.

The library routines are in lcp_utils.cxx and lcp.hxx. They use the following data structures and types:

- typedef int LCP; const int 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 path { LCP D, R; }; typedef struct path Path. 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.

The following routines are supplied for you in lcp_utils.cxx:

- void create_random_critical_paths (int *n_paths*, int *seed*). This creates 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. This function is called by main(); you don't have to call it.
- void set_passing (vector<LCP> *forw*, vector<LCP> *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.
- bool run_test_OK (vector<LCP> *forw*, vector<LCP> *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.
- bool check_results (vector<Path> &*failures*). Once you've decided which critical paths exist, you should package them up in failures[] and call check_results(). It will return true if you got them correct, and also print error messages telling you what was wrong otherwise. This function is called by main(); you don't have to call it.

- string printLCPvec (const vector<LCP> &LV); string printPath (const Path &p); string printPaths(const vector<Path> paths). These function provide support to print data structures in case it helps you for debugging your code.

The file lcp.cxx 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.cxx via the Provide interface.