# CS-345/780, Fall 2005
Laboratory I

Objectives

The purpose of this lab is to introduce you to the laboratory facilities you will be using this semester and to investigate some of the ways in which Handel-C is different from a conventional compiled or interpreted programming language.

Projects

Before you do any projects, you will need to make sure your account is set up and working correctly. Begin by logging into your account using the user ID and password supplied to you (<first initial><last name> for both). Be sure you log into the TREE domain, not into "This Computer." The first time you log in will take some time as Windows sets up your "roaming profile."

1. Be sure to change your password to something secure as soon as you log in the first time.

2. Click on the Firefox icon on the desktop and verify that it runs, and displays my home page. If not, you may need to start Firefox in "Safe Mode" from the Start menu, and tell it you want to use the default profile for yourself. Exit Firefox if you wish.

3. Click on the Acrobat Reader icon on the desktop an open the file C:\Program Files\Celoxica\PDK\Documentation\PAL\Pal Manual.pdf. Verify that you can read PDF files okay. Close Acrobat if you wish.

4. Open the My Documents folder and create a subdirectory named "My Projects" there.

5. Click on the DK icon on the desktop.

   a. Use the File menu to create a new workspace named "Laboratory I" in your "My Projects" directory.

   b. Make sure DK creates a new subdirectory (named Laboratory I) for the workspace under My Projects.

   c. Open the DK Tools→Options panel and make sure the Editor is set to insert spaces for tabs.

   d. Change the default tab width from 4 to 2 if you wish; do not use any value other than 2 or 4 for the code you write for this course.

   e. Verify that the Format panel has Monaco as the font for Output Windows. If not, change it so it does. (Monaco is a monospaced font, which is needed to preserve spacing in some of the reports that are displayed in the output window.)

   f. Verify that the Directory pane has paths into the Celoxica installation directories for Includes and Libraries.

   g. Close the Options Panel, and exit DK.

6. Log out of the computer you were using. Log into a different one and verify that the My Projects directory is there, that it has a subdirectory named Laboratory I, and that there are two files, named Laboratory I.hw (a workspace file) and Laboratory I.pref (a preferences file) in that directory.

7. Click on the workspace file to start DK.

   a. Use File→New to create a new project, named "Parallel Threads" in your Laboratory I workspace. Select "Xilinx Virtex II" as the target device in the big panel on the left.

   b. Use File→New to create a new Handel-C source file named "parallel_threads.hcc" in the Parallel Threads project.

   c. Put a comment line in the editor window for parallel_threads.hcc that says:
   // parallel_threads.hcc

   d. Save the source file by typing Control-S. The asterisk in the title bar of the editor window will go away.

   e. Use Internet Explorer to verify that you have the following situation:
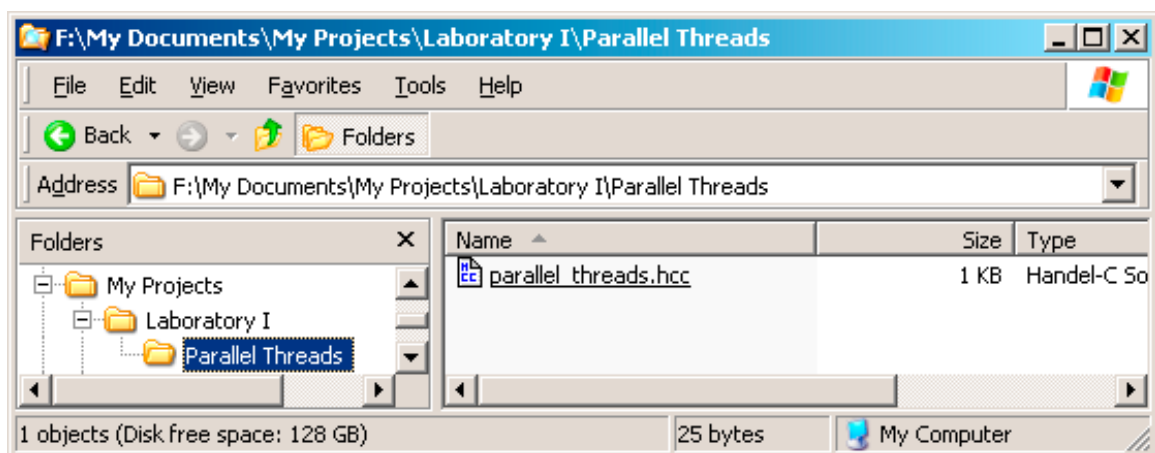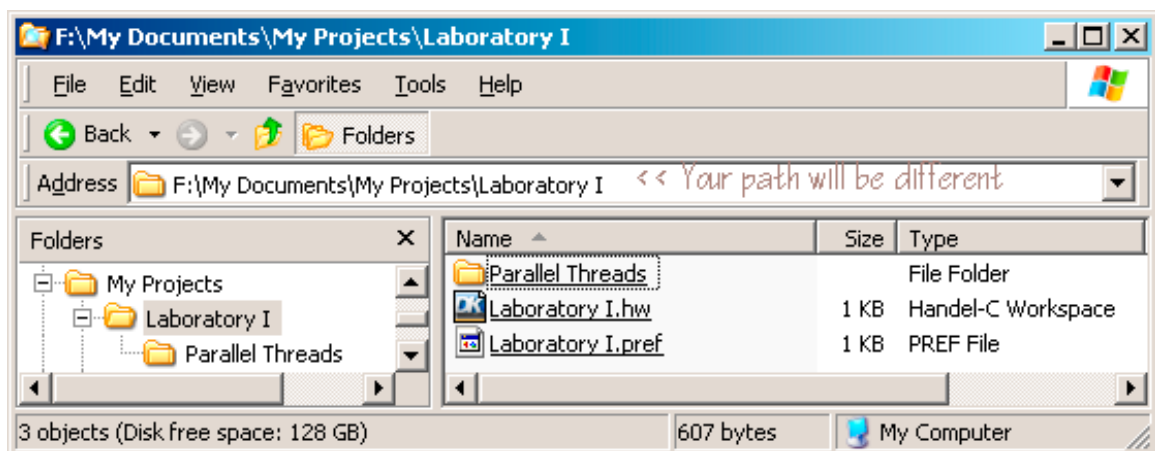
   f. Now set up your project for simulation using the Debug configuration.

i. Project→Settings opens up a configuration panel with 8 tabs. The General tab should have the "Generate Debug information" and "Generate XML reports" options checked. Add "USE_SIM" to the list of Preprocessor definitions. (Separate it from the others with a comma.) All three checkboxes in the Debug pane should be checked, all Synthesis options will be grayed out, you can select High-level optimization, and can leave the other panes unchanged for now.

ii. Now write the code for parallel_threads.hcc as follows: Declare three global variables, all four bits wide. Make one of them unsigned and the other two signed. Put a "set clock=external "V12";" statement before the *main()* function. Inside *main()*, have two parallel endless loops. One of the loops is to increment one of the signed variables by one on each clock pulse. The other loop is to increment the other two variables sequentially. Make sure the editor is managing tabs and spaces correctly as you enter your program.

iii. Build the program. (Press F7, or use the Build menu, or click the build button on the toolbar.) Correct any syntax errors and verify that the program compiles and builds with no warnings and no errors.

iv. Run the program by typing F5, using the Build menu, or by clicking the "Go" button on the toolbar. All you will see is a black "DOS Command Window" that shows up. Remember, simulations are actually Windows executable programs; this window is used in case your simulation program reads from *stdin* or writes to *stdout*. (We will probably never develop that kind of simulation in this course, but the option is there; for now just move the black window out of the way so you can see the DK GUI again.)

v. Use the View menu to turn on two debugging windows: Clocks and Threads (Alt-5) and Variables (Alt-4). You can make these windows come and go by using the keyboard shortcuts repeatedly. The clocks and threads window will show just one line with a column heading for number of cycles. Resize the column headings so you can read the number of cycles; the number will change periodically. The Variables window will show nothing for now.

vi. To debug the program you will need to interrupt its (simulated) execution, which you can do either by clicking the Debug→Break menu item or by setting a breakpoint in the code. (Alternatively, you could have started the program running in single-step mode by typing the F11 key instead of F5 back in step iv.) To set a breakpoint, click in the selection margin next to one of the three statements that increment a variable. (If you can't see the selection margin, use Tools→Options→Editor to see the panel where you can check an option to turn it on.) Click on the hand icon in the toolbar, and the breakpoint will be set and the program will stop executing at the line where you set it. If you do set a breakpoint to stop the program, you might as well turn

it off again right away; leaving it set will cause the program to stop every time that statement is reached.

vii. Now it's time to examine the debugging windows, starting with the Clocks and Threads. Because the two endless loops go in parallel, there are two finite state machines running simultaneously. Each one is represented by a "thread of execution." (Hence the little symbols that are supposed to look like spools of thread.) One thread will be printed in **boldface** and the other one not. Look in the editor window and you should see five arrows, a green one pointing at *main()*, two gray ones in one loop, and a white and yellow one in the other loop. The white and yellow ones correspond to the boldfaced thread in the Clock/Thread window. If you right-click on the other thread in the Clock/Thread window, you will have the option to follow that thread: it will become boldfaced and the white/yellow arrows will shift into that loop in the editor window. Practice going back and forth between the two threads and observing the changes in the Clock/Thread and Editor windows.

viii. Now look at the Variables window. There are two tabs, one marked Auto, and the other marked Local. If there are a lot of variables, you can use the Auto tab to view just the ones that are changing at a particular time. But you have only three variables and it's most instructive to watch all three at once; click the Locals tab to see them. I know, the variables are global (declared outside of any function), but you get to see them using the Local tab. Go figure. You should see all three of your variables and their values, presumably using decimal representation. The actual values are probably rather random depending on when you interrupted the simulation. Right click in this window and you will see that you can display the values in binary, octal, decimal, or hexadecimal. Click the binary option, and you will see Handel-C's syntax for representing binary numbers: 0b0000, for example. The number of digits after the 'b' will always match the actual width of the variable you are looking at during debugging.

ix. You can use the F11 key to step through your program one clock cycle at a time. Using the binary display notice that all three variables simply increase in binary counting order (0b0000, 0b0001, 0b0010, … 0b1111, 0b0000 …), but the two variables that are incremented in the second loop only change values on alternate clock cycles, whereas the other variable increments every time. Switch to the decimal view and observe the difference between the way the binary numbers are interpreted for the signed and unsigned variables.

x. You can stop the simulation in any of several ways: press Shift-F5, or use the Debug→Stop Debugging menu item, or click the Stop Debugging toolbar button (it has a red X in it), or close the black DOS Command window. Pick one and do it.

g. Now you are to set up your project so it works on the RC200E
    development kit. Start by switching the "Active Configuration" from
    Debug to EDIF in the "Active Configuration" drop-down list in the
    toolbars (the one that says Debug in it). Now go to Project→Settings and
    observe that the General tab now is set to use EDIF as the names of the
    intermediate and final directories. Add USE_RC200E to the
    preprocessor symbols list. Go over to the Chip tab and verify that the
    Family is *Xilinx Virtex-II* (you set this when you started the new project,
    remember?) and select xc2v1000 for the Device number. The Package,
    Speed Grade, and Part number will all fill in automatically with the correct
    values for the FPGAs on the RC200E kits. On the Linker tab, enter
    rc200e.hcl in the Object/library modules text field. (This is the platform
    support layer library for the RC200 and RC200E boards.) The Build
    commands tab is to the right of the Linker tab. With the View set to
    Commands, click in the blue bar and type the following commands on
    three successive lines (do not hit Enter at the end of a line or you will
    dismiss the whole Options dialog; use the down arrow key instead. The
    three commands are *cd EDIF*, *call edifmake_rc200 "Parallel Threads"*,
    and *beep*. Don't hit Enter yet! Note the quotation marks in the second
    command; they are necessary because the project name has a space in the
    middle. Now go up to the View part of the pane and switch to Outputs.
    Type EDIF (the name of the directory where you want the outputs of the
    build commands to go). Now you can type Enter to dismiss the dialog.

h. Press F7 to build the project and generate the bit file. Fix the project
    settings until you see something like the screen shot on the next page.
    Notice that there are no errors after PAR, nor any warnings or errors after
    BITGEN completes.
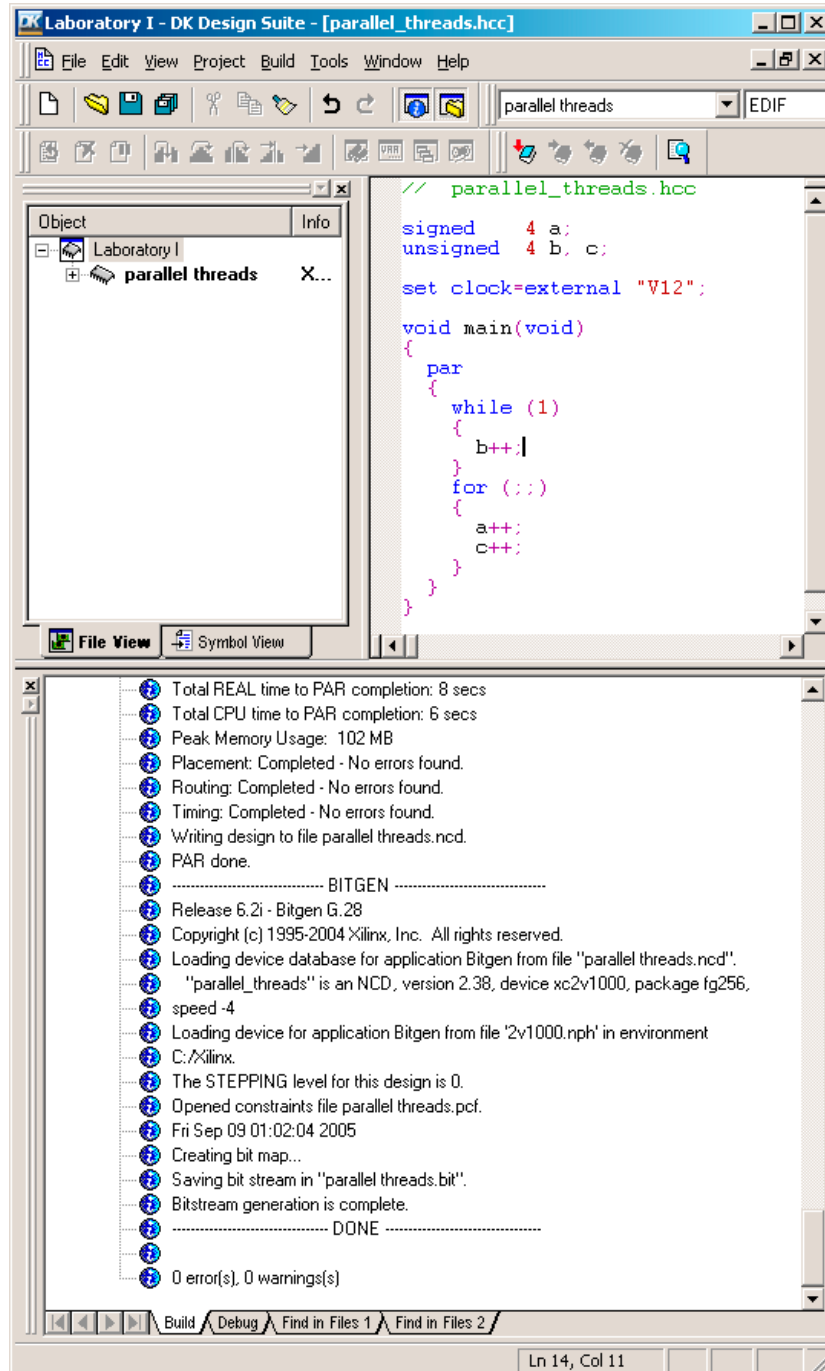
i. You could now download the .bit file to an RC200E for execution, but you may have noticed that the program does no I/O, so there would be no way to know what the program is doing. We'll handle that in the next project. For now, examine the contents of the EDIF and DEBUG subdirectories of the Parallel Threads project. Use Vim to look at the build.log files. They should look familiar. Find parallel threads.bit. Look at the contents of the Reports directories; they won't make much sense, but use Firefox to open some of them to see what they look like.

8. Save and close the Parallel Threads project and exit DK. Make a backup copy of your project on a floppy disk, memory stick, or another computer. Log out so your files will be saved on the lab's server, Maple.

9. Log in again, start DK, and open the Laboratory I workspace. Create a new project called Blink LEDs. Add a Handel-C source file named *blink_leds.hcc* to the project. Set up the Debug configuration with the USE_SIM preprocessor symbol and the EDIF configuration with the USE_RC200E symbol. On the Linker tab, add the following modules: *stdlib.hcl*, *pal_sim.hcl*, and *sim.hcl*. Also, add the following file to the Additional C/C++ Modules: C:\Program Files\Celoxica\PDK\Software\Lib\PalSim.lib.

10. Add the following lines of code to the beginning of *blink_leds.hcc*:

```
#ifdef USE_SIM
#define PAL_TARGET_CLOCK_RATE 200000
#else
#define PAL_TARGET_CLOCK_RATE 50000000
#endif
#include <pal_master.hch>
#include <stdlib.hch>
```

11. Add two *main()* functions to the source file. One is to toggle the value of a one-bit unsigned variable every 500 milliseconds. The other will display the value of this variable as the state of one of the LEDs. Instructions will be given in class on how to do these two things.

12. Simulate your program and make sure it works.

13. Change the EDIF configuration so it links to the *stdlib.hcl, pal_rc200e.hcl*, and *rc200e.hcl* library modules. Also on the Link tab, click to turn on the *Generate estimation info* option. Update the Chip tab and add the Build commands similarly to the way you did for the Parallel Threads project.

14. Build the EDIF version and verify that there were no errors during the build process.

15. Download the *Blink Leds.bit* file to an RC200E kit using the FTU2 utility, as explained in class.

16. Use Firefox to look at the file named Summary.html in the EDIF subdirectory. Click on the links to see more details.

Report

For this Laboratory, you are to write out the answers to the following questions and submit them along with a copy of your .hcc files. (Starting with Laboratory II, you will be preparing structured Laboratory Reports.)

1. Look at the build.log file for the EDIF configuration, and tell what the four Xilinx programs are that lead to the generation of the final .bit file. How long did the placer and router parts of PAR take to execute?

2. Look at the .edf file for the project. This is generated by the Handel-C compiler and passed on to the Xilinx *ngdbuild* command. How does the Xilinx tool know which FPGA part number is being targeted? How did the Handel-C compiler know this?

3. Compare the sizes of the files generated for the two projects.

4. How long did it take to build the .bit file for the second project?

5. How much of the FPGA was used for the second project? (Look at the build log.) And what can you say from the build log about what timing constraint was met and by how much?

6. What is in an .hch file? What is in an .hco file? What is in an .hcl file? What is in a .lib file?

7. What file contains the PSL for the RC200E? What one contains the PAL for the RC200E?

8. Pal_master.hch defines PAL_ACTUAL_CLOCK_RATE. How does this happen? Be both as specific and as general as you can. That is, tell how it works when using the RC200E, and generalize your answer to tell how it works for both Debug and EDIF configurations.

9. Use the build log file for the EDIF configuration of the second project to tell how many lookup tables and flip-flops the *sleep()* macro procedure uses. What two factors determine the answer to this question? Note: A "slice" consists of two LUTs and Flip-Flops.

10. Explain the three numbers in the row marked NET "WG10_rc200e_hcl_O" PERIOD of the PAR Clock Report. Where does the 20.00ns value come from, and how were the other two numbers determined?