CSCI 345 Laboratory III

September 24, 2003

Objective

This laboratory session continues with your introduction to project development for the RC200E using the DK software from Celoxica. In this session, you are to set up and develop an application that works with the RC200 switches, LEDs, and seven segment displays using Platform Abstraction Layer (PAL) macros to avoid the need to do interface-level I/O.

Although it would be easier to do this lab by cutting and pasting from sample code available in the Celoxica\PDK directory, this lab session will guide you through the essential elements of project setup step-by-step. After this lab, you can cut and paste all you want.

Once again, you are to work in groups of two or three during the laboratory session. Do the work for the session in one person's account, and then make the directory tree containing the laboratory workspace available to the other members of your team to copy.

Project Specifications

The program you are going to develop is an up-down counter. When the bit file is loaded into the RC-200, the seven segment displays are to count in decimal either up towards 99 or down towards 00. Pressing one pushbutton will cause the counter to count up, pressing the other will cause the counter to count down, and pressing neither is to leave the counter unchanged. You get to decide what happens if both buttons are pressed at the same time. When counting, the counter must step at exactly one-second intervals. We will simulate the code before downloading it.

Lab Activities

1. Create a Workspace and Add a Project/File to It
2. Configure the Project for Simulation
3. Use PAL Code to Turn the Simulated LED on/off
4. Configure the Project for Downloading
5. Make the Hardware LED Blink at 1 Hz
6. Build the Up Down Counter
7. Submit a Report of Your Lab Activities

Create a Workspace and Add a Project/File to It

If you haven't done so already, click "My Documents" on the desktop, and create a new folder named "My Projects." It will be on the same level as 'My Pictures," My Music," etc. *Note the path to your "My Projects" directory in the Address bar of Windows Explorer.*

Start DK, select File→New, and choose the Workspace tab on the dialog box that comes up. Use "Laboratory III" as the *Workspace name*. For the *Location*, browse to your "My Projects" directory at the address you noted in the previous paragraph.

Use File→New again, but this time, select the Project tab. Name the new project "Blinking LED," select Xilinx Virtex II in the left-hand pane, and be sure the project is part of the Workspace you just created. Now use File→New again to add a Handel-C source file named "*blinking_led.hcc*" to the project. If you haven't done so already, select Tools→Options and go to the Tabs tab. Set it so the editor substitutes spaces for tabs. (A Vickery pet peeve.) You might also want to set the tab stop width to 2 instead of 4, and be sure auto-indent is checked. Put a comment line containing the file name at the beginning of *blinking_led.cc*, and put a comment block at the beginning of the file that describes the program briefly and that lists the names of the people in your lab group as authors.

Configure the Project for Simulation

The DK software comes with some preconfigured build configurations. We will use the "Debug" configuration for simulation and the "EDIF" configuration for generating the .bit file for downloading. We will have to modify both configurations to get the project to build correctly. A requirement for the lab is that the same Handel-C source code is to be used for both configurations without change, so we will make the changes needed for simulation while keeping in mind that they must be compatible with the build configuration for downloading.

For simulation, we will define the USE_SIM preprocessor directive. But rather than put the #define statement in the source code, as we did last week, we'll predefine it so we can test it in the source code using a #ifdef statement. From Project→Settings, be sure the Debug configuration is selected, and add USE_SIM to the preprocessor symbols. Write code before your *main()* function that sets the PAL_TARGET_CLOCK_RATE to 1000000 (1,000,000) if USE_SIM is defined. Write your code to #include *pal_master.hch* (unconditionally). Define a one-bit variable and write a *main()* function that endlessly alternates between turning the bit on and off. You will need to link to *sim.hcl* and *pal_sim.hcl* in order to generate a simulation of your program, so add those two libraries on the Linker tab of the settings for the Debug build configuration. (Be sure your Tools→Options Directories tab has C:\Program Files\Celoxica\PDK\Hardware\Include for include files, and C:\Program Files\Celoxica\PDK\Hardware\Lib for libraries. You also need to add C:\Program Files\Celoxica\PDK\Software\Lib\PalSim.lib to the Additional C/C++ Modules list on the Linker tab of the Simulation build settings.

Simulate the program and verify that the bit turns on and off on alternate clock steps.

Use PAL Code to Turn the Simulated LED on/off

To use the PAL, you use macros that are documented in the PAL API Reference Manual.

In your *main()* function you need to do a couple of tests at the beginning, one is to call *PalVersionRequire major, minor)* (page 6). This project will work with any PAL

version, but you can specify a major version of 1 and a minor version of 2 to be sure you are not working with an old version. Be sure you capitalize the method name properly. You also need to make sure that your target platform has LEDs that you can access through the PAL. Use the *PalLEDRequire(num)* method to do this. You need only one LED for now, but you can experiment to see what happens if you specify more than 8 when building for simulation if you want to.

These two method calls do not generate any run-time code, but they look syntactically like function calls, and must therefore be placed *after all variable declarations* in your *main()* function. (Unlike C++ and Java, C and Handel-C require all local variables to be declared before any executable statements in a function.)

To turn the LED on or off, you use the *PalLEDWrite(handle, value)* method (page 12). The *handle* argument is an identifier for the LED you want to light up. The PAL lets you obtain a valid handle for a LED at compile time using a macro named *PalLEDCT(index)*, where *index* is a number between 0 and one less than the argument you passed to *PalLEDRequire()*. With only one LED required, your options are pretty limited for the value of this argument. The "CT" at the end of the macro name indicates that it is to be evaluated at "compile time" rather at run time. There is also a run-time version that will work, but the compile time version doesn't generate any hardware.

Simulate your program and verify that the LED turns on and off. Don't bother to try getting it to cycle exactly once a second yet.

Configure the Project for Downloading

First of all, the simulated and downloaded versions of the program will need two different clock rates. In the build configuration for EDIF, define the preprocessor symbol USE_RC200, and in your code #define PAL_TARGET_CLOCK_RATE to 1 (1 Hz) if this symbol is true. Be sure the Chip tab has the correct part number (XC2V1000-4FG456), and put both the *rc200e.hcl* and *pal_rc200e.hcl* libraries in the list of object modules of the Linker tab. Finally, you need to configure the commands for generating the .bit file, which is done on the Build Commands tab. The first command is to change directory to the EDIF subdirectory for the project (cd EDIF), and the second command is to use the DOS *call* command to invoke the *edifmake_rc200* batch file. The last argument on call command line is the name of the projects, which has to be put in quotes because there is a space in the name, *Blinking LED*.

On the Build Commands tab, you also need to specify the directory where the output files will be placed. This directory should be named EDIF, to match the name of the configuration, and gets entered by selecting "Outputs" instead of "Commands" in the View List Box on the Build tab and just typing the directory name in the text box.

Build your program using the EDIF configuration and fix the problem due to the target clock rate being too low for the RC-200. You should also still be able to build and simulate your code, but there is no point in downloading the .bit file yet because the LED will blink way to fast to see.

Make the Hardware LED Blink at 1 Hz

You need to introduce a delay in your main loop so the LED doesn't blink too fast. The delay depends on the clock rate for *main()*, which in turn depends on whether the code is being built for simulation or for download. The most general-purpose technique would be to call a function that delays the code for an amount of time passed as a parameter to the function. But function calls take place at run-time and take clock cycles to make the call and return. Instead, we'll use a macro to generate the necessary delay at compile time. Handel-C provides a way of writing compile-time macros that look like procedures, called "macro procs." (Macro procs are described starting on page 131 of the Handel-C Language Reference Manual.) Write a macro proc named *msec_delay()* that accepts a number of milliseconds as its argument. Because this is a macro, you don't declare the type of the argument anywhere:

```
macro proc msec_delay( msec ) { … }
```

Call this macro proc with an argument of 500 every time the LED turns on or off. Leaving the macro proc body empty for now (replace the "…" above with nothing), be sure you can compile your code for both simulation and download.

You can find examples of code like *msec_delay( )* throughout the sample code provided by Celoxica. Here, we'll write our own version to see what's going on. The first thing to note is that the header file *pal_master.hch* will #define a symbol PAL_ACTUAL_CLOCK_RATE to be the, uh, actual clock rate that will drive your *main()* function. This clock rate will most likely be a little different from the target clock rate because there may not be a way to divide any of the hardware clocks on the RC200 to exactly the rate you asked for.

What you want to do is to have a loop inside your macro proc that delays the code for the proper number of clock cycles to get the number of milliseconds needed. We know that incrementing or decrementing a register takes one clock cycle:

```
while ( count > 0 ) count--;
```

There are two things that need to be done: one is to assign a value to count based on the actual clock rate and on the number of milliseconds passed as a parameter to the macro proc. This value can be determined at compile time provided the millisecond parameter is a constant (like the number of milliseconds in half a second). You can use a Handel-C *macro expression* (Macro expressions are described startinf on page 123 of the Handel-C Language Manual.) to do this:

```
macro expr cycles = f(PAL_ACTUAL_CLOCK_RATE, msec);
```

It's an exercise for you to figure out what code to write on the right side of the equal sign. That is to figure out what expression to write in place of what looks like a function call above.

Notice that the macro expression defines a value for the symbol *cycles*, which is not the variable *count.* Cycles is a constant determined at compile time when macro substitution takes place, but count is a register that has to be loaded and decremented at run time. And how many bits must there be in this register? You want just enough bits to hold the value of cycles and no more in order to make efficient use of the hardware. The solution

is to use one of Handel-C's standard library functions, *log2ceil(arg)*, which tells how many bits are needed to represent the value of arg. A piece of overhead is that you need to put the reference to this function inside parentheses to get your register declaration to compile. For example:

```
int (log2ceil(6)) x;
```

would declare x to be a 3 bit integer.

This function is declared in the header file *stdlib.hch,* and you will need to add the standard library, *stdlib.hcl* to the linker tab for both the simulation and EDIF build configurations.

Compile your code for both simulation and download, and adjust the target clock rates for both so the LED blinks exactly once per second.

Build the Up Down Counter

Using your blinking LED project as a model, implement the Up Down Counter project.

Submit a Report of Your Lab Activities

Submit copies of *blinking_led.hcc* and *up_down_counter.hcc* by email by Wednesday October 1.