Embedded Systems Lab 4: The NiosII soft core processor

Introduction

Embedded Systems are built around more than just FPGAs. In fact, the majority will use a microprocessor of some sort. This lab is designed to introduce you to a microprocessor called the "Nios II", which can be synthesised and downloaded in to the FPGA.

This type of processor is called a "soft core", a reflection of the fact that it is written in a Hardware Description Language rather than being permanently wired together in the silicon of the device. This makes it suitable for educational use, as it allows us to quickly turn various processor features on and off and have them removed from the actual implementation.

Soft Core processors are rarely used in commercial embedded systems however, primarily as the generic nature of an FPGA means they aren't able to sustain as high a clock speed as dedicated hardware. For example, our DE2-115 board has a 50MHz oscillator driving the Nios II where any processor of similar capability implemented in actual hardware would likely run at least 10 times as fast.

Once the processor is in place, we can program it either with Assembly Language code, or with C; we will stick with C for this lab. The C compiler, assembler, debugger and related tools that we will use are part of the Altera University Package. These are cut-down and simplified versions of the full commercial tools, but are very much adequate for our purposes.

Nios II Peripherals

The Nios II itself is just a processor; it has no memory in which to store a program and no peripherals with which to interact with the world. In this lab we will connect peripherals allowing a number of the simple devices on the board to be controlled from software. We will also connect the Nios II to the board's Dynamic RAM (DRAM).

A peripheral is a piece of hardware that acts as a bridge between the processor and a device on the board. For this lab, we will only be using "Parallel I/O" or "Parallel Port" peripherals. These are exactly like the "PORTx" peripherals you may have used on the ATMega8 microcontroller in previous courses. Each parallel port peripheral is given a memory address when we create the system; a read or write to that memory address from the processor will interact with the peripheral. In the case of a Parallel Port, the values we read or write are the logic states of the output of the peripheral. For example, if we have an 8-bit port connected to the green LEDs and write "8'b00001000" to it, then we will have turned the 4th LED on and the rest off. Similarly, if we have a parallel port connected to the 18 switches and a read from that peripheral's address returns "18'b000000000000000111", we know that the last three switches are on and the rest are off.

One of the stranger aspects of this lab is the use of different clocks for the DRAM and the processor. The specifics of this connection are quite complex and not important for you to understand, however they come about from the fact that the DRAM and processor both operate on the same

clock edge. Simplistically, you can imagine that if you read from the DRAM, the data will be ready on a positive edge but that the processor will try and latch the data on that same positive edge. Because the data transition is not instantaneous, the DRAM's output may not yet have settled and the processor might get an incorrect value. We get around this by using two clocks slightly offset from each other such that the DRAM's positive edge occurs just before the processor's positive edge.

The final peripherals that must be added to the Nios II system are the "System ID", "JTAG UART" and "Interval Timer". These are all required by the University Monitor program in order to correctly download and debug your code, you will not interact with them directly.

Part 1: Building the Processor

Quartus has the ability to build up complex systems by specifying connections between pre-built blocks using the SOPC Builder[1]. This functionality is related to, but separate from, the MegaWizard Plug-ins that were examined in a previous lab.

The Nios II soft-core processor is one element that can be assembled in this way, along with peripherals to drive all the components on the DE2-115 development board.

Activities

Open Quartus and start a new project called "Lab4" and select the correct FPGA type from the list.

Select Tools > SOPC Builder

Name the system "Lab4_Nios" and select "Verilog" as the implementation language.

Quartus will prompt you to use the successor to the SOPC Builder, called Qsys instead, however this is not fully supported on the university machines. Check "Don't show again" and click "OK".

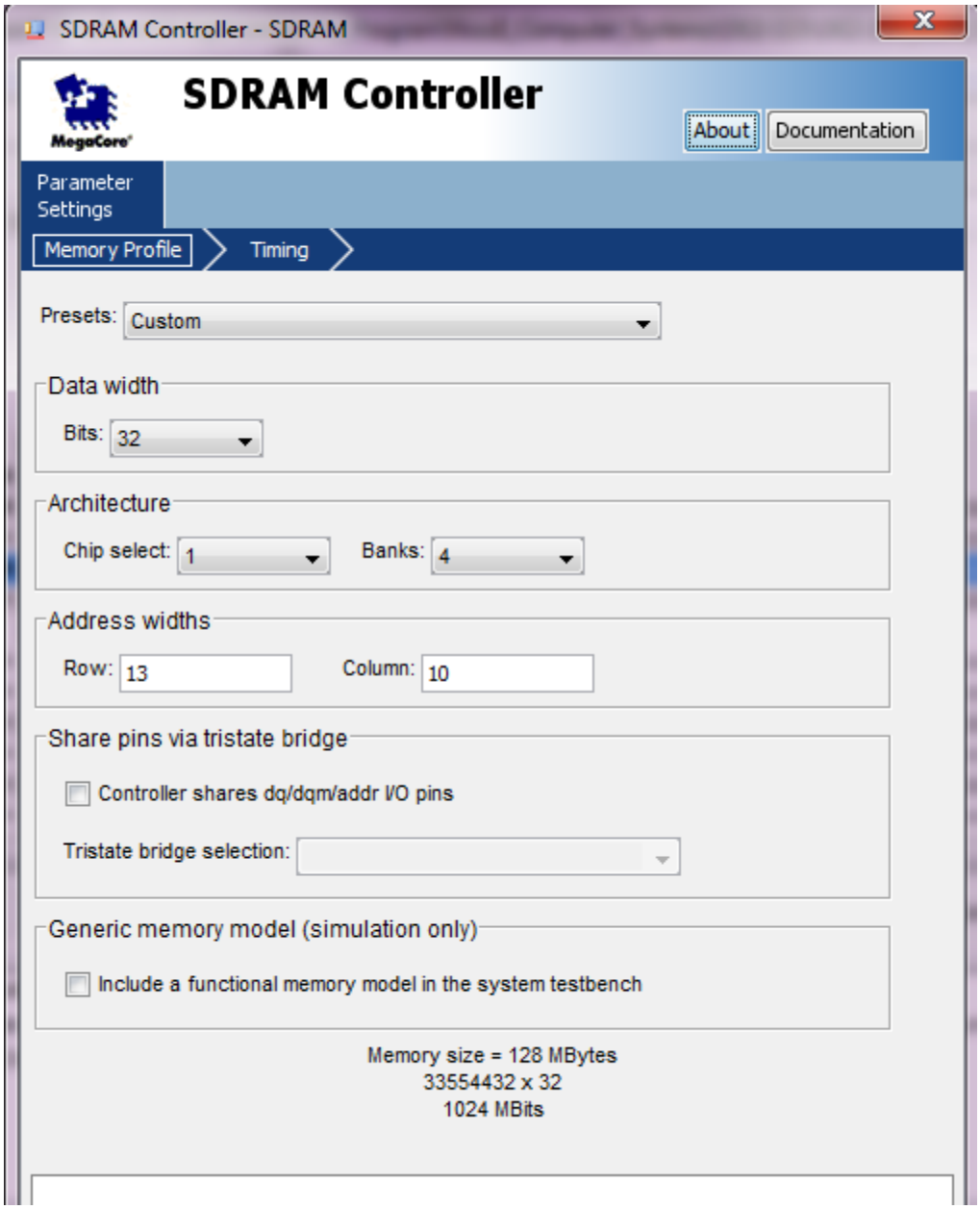
The component library pane on the left-hand side lists all the devices that may be integrated in to your system. Expand the Processors section and double-click "Nios II Processor".

There are three different types of Nios II processor available to be created at this point: The Nios II/e, /s and /f. Each of these processors have different capabilities including caches, branch prediction, hardware multipliers and dividers and maximum instruction throughputs.

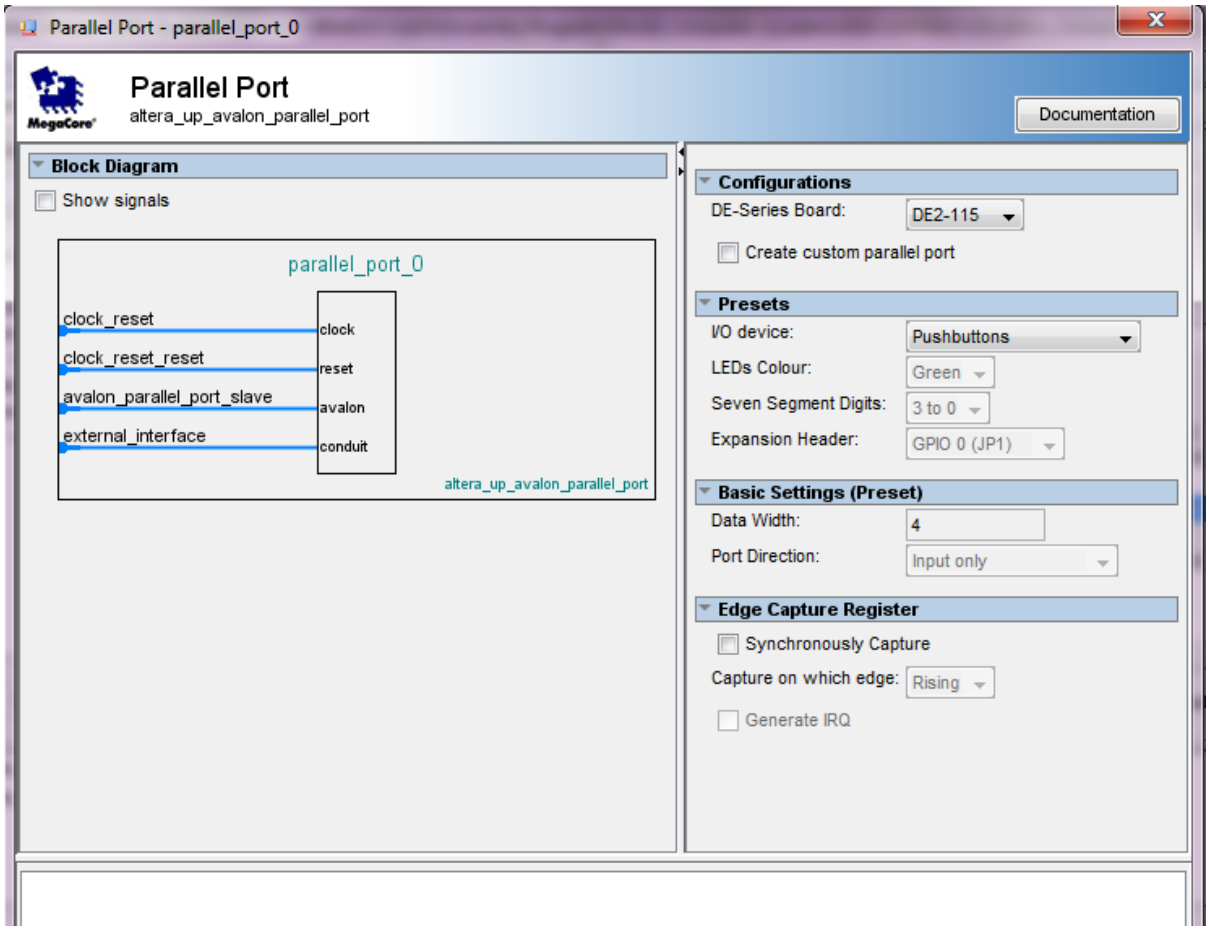
For this lab, we will use the simplest processor first. Select Nios II/e and click "Finish".

The next piece of the system that must be added is some memory for the processor to use. Select "Memories and Memory Controllers" > "External Memory Interfaces" > "SDRAM Interfaces" and double-click "SDRAM Controller" to add it to the design. Change the preset menu to "Custom". Edit the settings to match those in the image below.

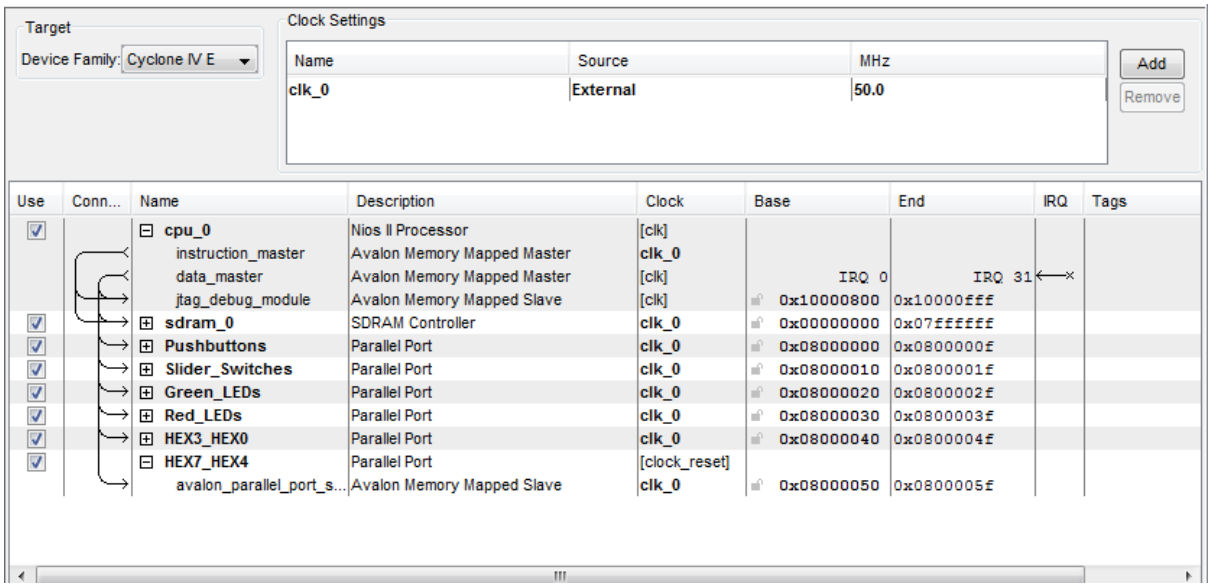
[1] SOPC System On Programmable Chip



The simple peripherals on the board (switches, buttons and LEDs) are all driven by generic Parallel I/O modules. You may select either the generic PIO peripheral or the University Program Parallel Port peripheral to get access to these devices but we will use the latter as it contains some preset values for the devices on our board. Expand the "University Program" > "Generic IO" section and double-click "Parallel Port". Select "DE2-115" in the DE-Series Board drop-down menu then select Pushbuttons in the I/O Device section as shown below. Click Finish.

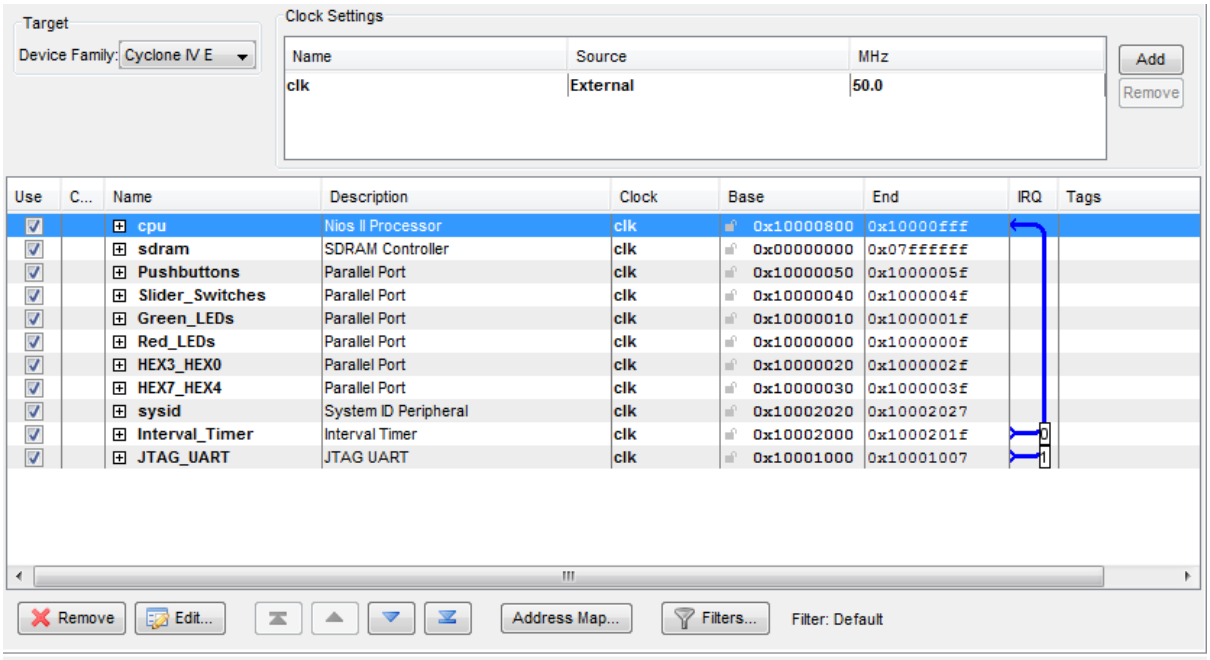


Right-click on the newly created peripheral in the main SOPC view and rename the new peripheral to "Pushbuttons". Repeat this process for each of "Red_LEDs", "Green_LEDs", "Slider_Switches", "HEX3_HEX0" and "HEX7_HEX4" respectively, choosing the appropriate "I/O device" selection for each one. Your main SOPC view should now look approximately as below.



Use the same process along with the Component Library search window to add a "JTAG UART", "Interval Timer" and "System ID Peripheral". All of these devices may have their settings left at default for the time being.

Double-click on "clk_0" in the Clock Settings box at the top and rename it "clk". Rename the other peripherals in the main SOPC view as shown in the image below, noting that capitalisation will matter.



Click System > Assign Base Addresses. This will allocate each peripheral a new spot in the memory space of the processor.

Double-click the CPU element of the SOPC builder and select "sdram" from the drop-down box for both Reset Vector and Exception Vector. Click Finish again.

Take a screen shot or write down the addresses of each peripheral as stated in the "Base Address" column, we will require this number when we come to write code to drive the processor.

NOTE: This number may be different for everyone and is absolutely important to get right!

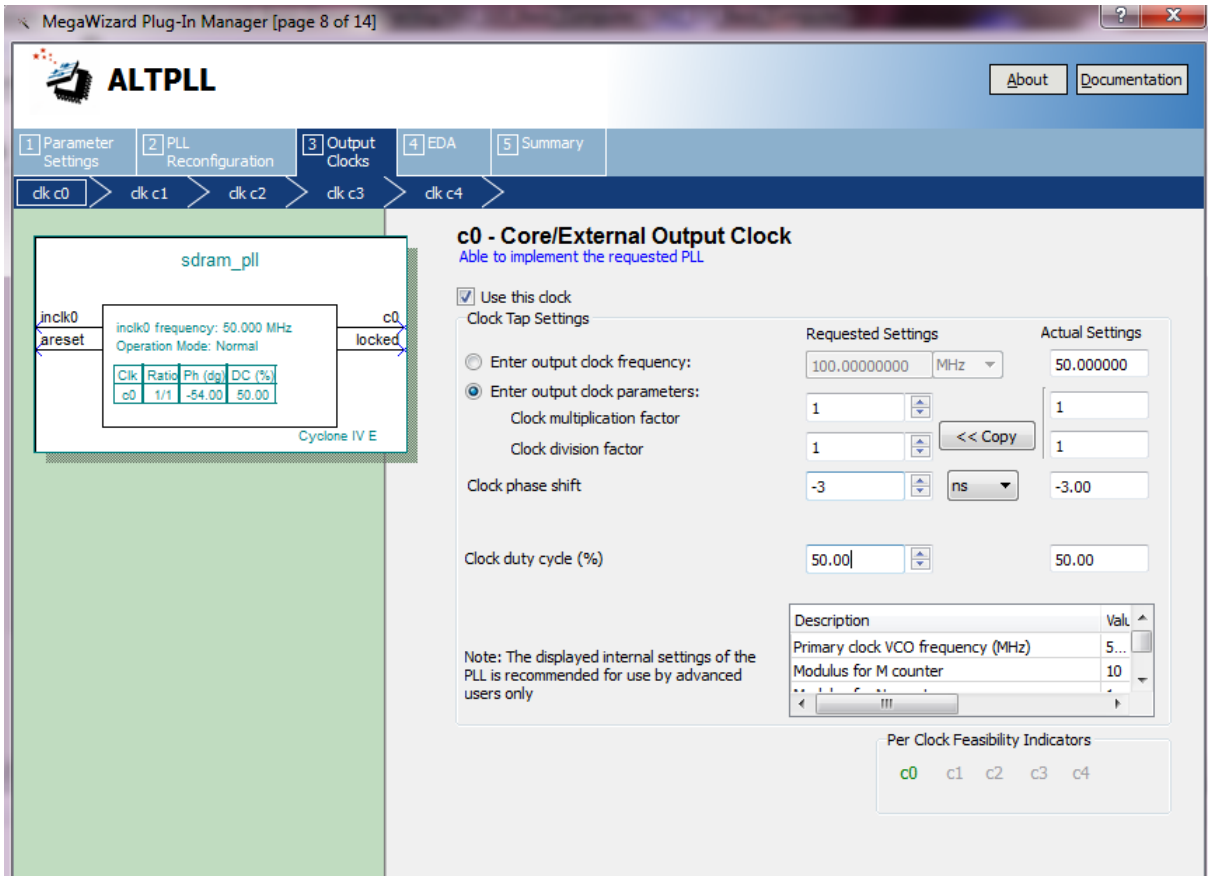
We're now ready to build the system. Click Generate at the bottom of the SOPC Builder window and, if prompted, save the SOPC file as "Lab4_Nios.sopc". Once the window shows "System generation was successful", you may exit the SOPC Builder.

Copy "Lab4.v" from the Lab File Package in to your working directory. Add it to the project, locate it in the Files tab, right click on it and select "Set as Top-Level Entity".

As discussed in the introduction, we need to use a bit of clock trickery to get the DRAM to interface correctly with the Nios II processor.

Open the MegaWizard Plug-in Manager and create a new ALTPLL item (found in the I/O tab). Name it sdram_pll.v and ensure the Verilog language is selected.

Set the Input Frequency to 50MHz and click "Next". Uncheck the boxes corresponding to the creation of the 'areset' input and 'locked' output. Continue clicking "Next" until the Output Clock configuration window is open. Leave the multipliers and dividers at 1 but change the Clock phase shift to -3 ns as shown below.



Click "Next" to configure clock c1. Enable "Use this clock" but leave all settings on that tab at defaults and click "Finish" twice.

Enter the following line at the end of Lab4.v:

```
sdram_pll sdpll (CLOCK_50, DRAM_CLK, system_clock);
```

Compile the design however **do not** program it to the board at this stage.

Part 2: Software

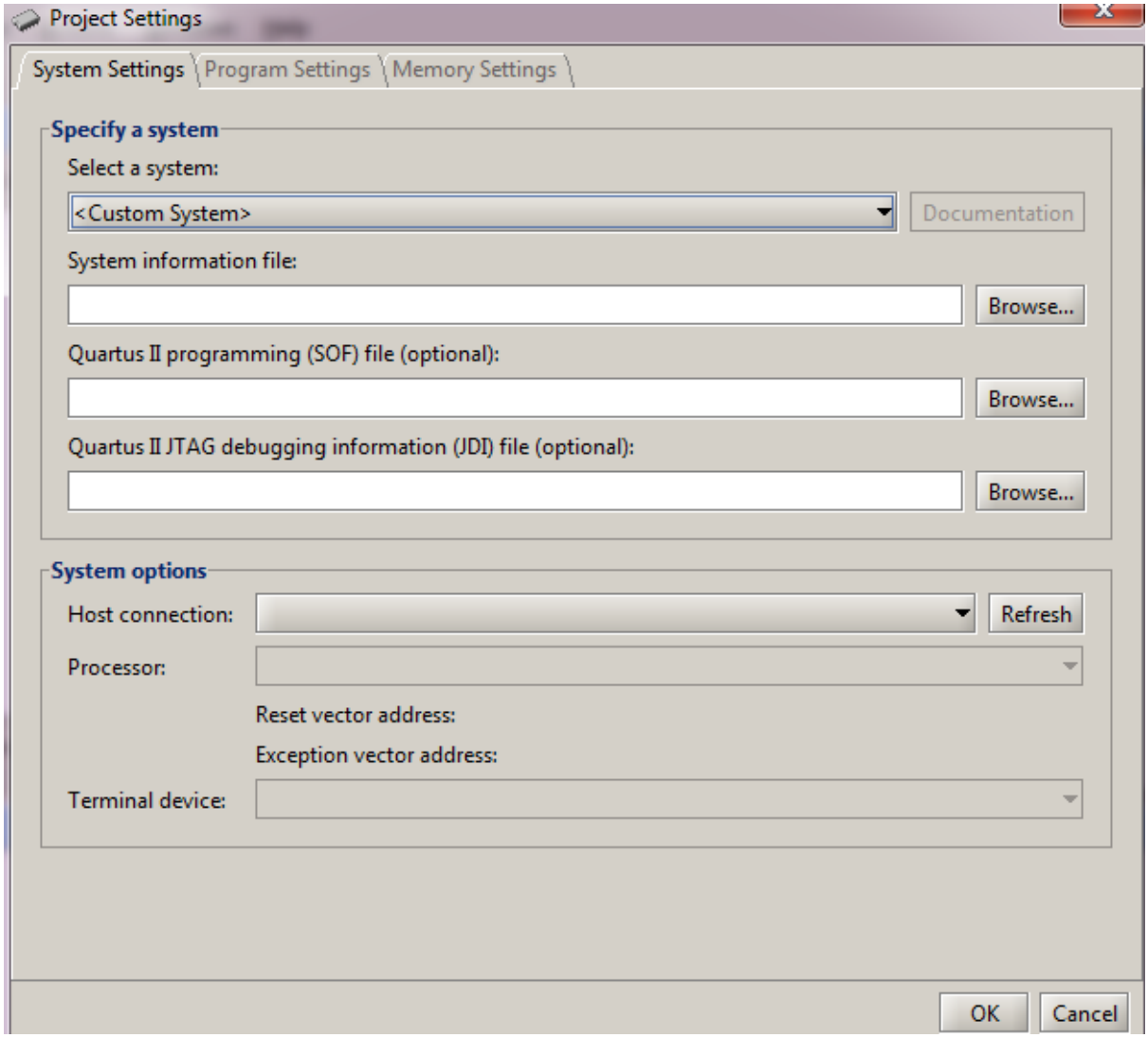
Software for the Nios II processor can be written in assembly language or in C, we will write it in C. Complex programs should use the Nios II EDS suite for compilation and debugging, however our requirements are more modest; we will make use of the simple Altera University Monitor Program to compile, download and debug our software.

Activities

Open the Altera University Monitor Program. Select File > Open Project and select ESLab4.ncf from the lab software package.

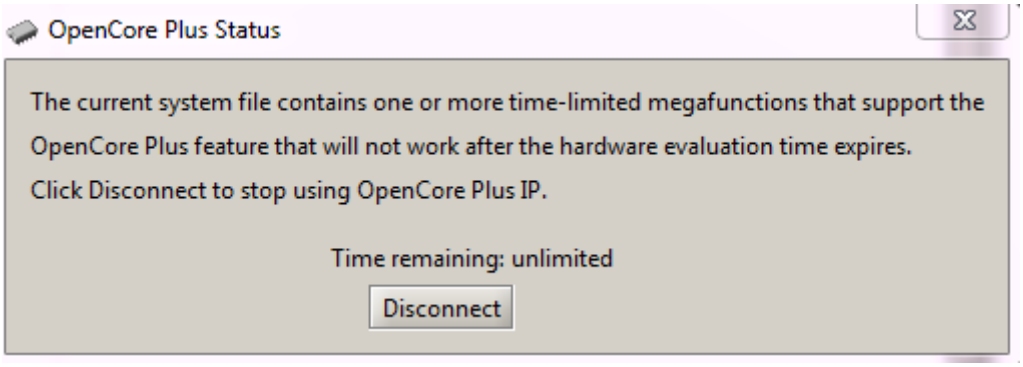
Open eslab4.c from that same package in a text editor and note the portion of the code that initialises pointers to the data registers that will be used to control our peripherals. Modify the code so that these pointer values match the Base Address values for each module as recorded above, for example if your Red_LEDs module has a base address of 0x10001020 in your system, change the value of red_LED_ptr to that value. Briefly examine the file and understand what is expected to happen if it runs correctly.

Save the file and return to the Monitor Program. Select Settings > System Settings and change the "System Information File" and the "Quartus II programming (sof) file" to those generated in Part 1. Click OK.



Select Actions > Download System and click "Download" to program your Nios II-based system to the board. If a box appears telling you that the file is time limited (like the box below), **do not** close

the box or click Disconnect. Simply move the box to one side of the screen and keep working.



Select Actions > Compile and Load to build your code and program it to the board. An assembly version of your C program will appear in the Disassembly window with the first instruction highlighted in yellow. Click the Continue button to start the program running.

Move the switches, press the buttons and verify that the design is working as you understand it. Try changing portions of the code, including the delay time and the pattern on the HEX displays. Recompile and re-download your code and verify your changes.

Click the Pause button, whichever instruction was executing at the time you clicked that button will be highlighted in yellow. Scroll through the code and note that the C from which the assembly was generated is inserted throughout the code. Find the piece of assembly that implements

```
HEX_bits = HEX_bits << 1;
```

How does it work? Is this how you would expect to do a logical shift?

Change the code that to implement a C equivalent of the following Verilog code:

```
assign LEDR = SW[15:8] + SW[7:0];
```

Note that C doesn't have bit-wise slicing operations like this so you will have to come up with a combination of binary operations and shifts to perform the isolation of the different components of SW.

Debugging

A common feature required when debugging is to set a break point. A break point is a mark on an instruction such that the system will pause ("Break") when it executes that instruction. To place a breakpoint, click in the margin on the left of the disassembly window on the line that contains the instruction you wish to break upon.

Pause the execution of the program, scroll through the code and place a breakpoint on an instruction inside the 'if' statement that checks for a key press. Start the code running again and verify that when you do press a key, the program pauses and the Monitor Program window updates.

Nios II/f experiment

Open the SOPC builder in your Quartus project from Part 1. Double-click the CPU element and select Nios II/f. Regenerate the system, exit the SOPC builder and recompile your project.

Back in the Monitor Program window, click "Disconnect" then open Settings > System Settings and update the "System Information File" and "Quartus II Programming (SOF) File" boxes with those from the above project. Note this must be done *even if those file names haven't changed*. The Monitor program caches copies of these files, so these boxes must be 'touched' every time you change your Quartus project in order for the monitor program to refresh its copies. Note that you may now have two programming files, "Lab4_Nios.sof" and "Lab4_Nios_time_limited.sof"; the more powerful II/f processor has different licence restrictions and therefore generates a different type of programming file. To determine which file to use, look at the "Date Modified" field for each file and pick which ever was updated the most recently.

Download the system to the board again as above, once again noting that if a box pops up regarding licencing and time limitations, do not click disconnect, just move the box to one side.

Recompile your code and load it on to the board. Does it now work as you expect? What happens when you pause and resume the code a number of times? The two types of Nios core are behaving differently; can you go back in to the SOPC builder and disable the specific feature that's causing the different behaviour?

Now that your program's working again, you may note that the delay has been shortened due to the II/f executing faster. This is of course a good thing, however you can now see that this extra speed has come at the cost of unanticipated behaviour in certain circumstances!