LAB 4: The Xilinx SDK and ZedBoard

This lab will guide you through running a simple assembly language program on the ZedBoard hardware kit. The software used with the ZedBoard is the Xilinx Software Development Kit (SDK).

ZedBoard is a computing and educational PWB device based on a Xilinx Zynq SOC (System On a Chip).

The Xilinx Zynq chip contains an ARM processor (two cores, actually). However, the Zynq chip is also *a powerful FPGA device*. An FPGA is a programmable logic chip that can be configured to implement different hardware configurations. For our purposes Zynq will be configured to behave similarly to a microcontroller. A pre-built FPGA logic design will be used to provide the ARM processor with access to several on-board peripheral devices. *Zynq is equipped with a Dual core Arm Cortex A9 working at up to 667MHz.* As a consequence of using a Cortex A9, it will support a more advanced ARMv7 instruction set than the ARMv4 instruction set we have been using so far, but for our purposes the differences will be minimal. The Zynq chip supports two ARM cores but we will only be using one of them.

As mentioned, the ZedBoard also has a number of peripherals in addition to the Zynq FPGA including LEDs, slider switches, push buttons, and a small OLED display. Example code is provided in this lab to use some of the peripherals on the ZedBoard. This document is a step-by-step guide that should help you to get acquainted with the Xilinx SDK environment and the tools necessary to implement and deploy projects on the ZedBoard hardware using assembly, C, and even C++ languages.

We will program the ZedBoard from a host PC via the Xilinx SDK software. Xilinx SDK is based on the popular Eclipse IDE (Integrated Development Environment) and it will aid us to write C/C++ and assembly code to target an ARM CPU core on the board. It also allows us to debug code running on the target.

First of all, login on a Lab Computer. The release version we will be using is 2018.1.

1. Loading the Xilinx SDK "workspace"

When using Eclipse, all code you write must be created within a project and these projects reside inside of a workspace. On Canvas we have provided the file “ensc254_lab4-1194.zip” that contains a Xilinx SDK workspace with some hardware configuration (Hardware Platform and Board Support Package) and a lab application project already set up. The hardware configuration enables using different peripherals on the board including the LEDs, slider switches, push buttons, and small OLED display mentioned above.

Download the abovementioned zip file from Canvas, and extract it to your U: drive. You must end up with a folder U:\XilinxWorkspace and inside that folder there must be a number (3 to 5) of subdirectories.

When you launch the Xilinx SDK it may prompt you to select a directory to use for the workspace. If so, pick the "U:\XilinxWorkspace" folder that you just extracted.

If all goes well, this will bring you to the main window layout. You should see several project folders (for Lab 4, the hardware platform, and the board support package) in the Project Explorer pane on the far left. The file asm.S is probably open in the editor pane.

The system.hdf file in project "SfuEnscZedboardProject" is useful because it contains the address map of peripherals as seen from the CPU cores. The peripherals we are using are memory-mapped into the CPU address space and as such we can use normal LOAD and STORE operations to access them. For reference, the base addresses of the OLED display, switches, LEDs and push buttons are:

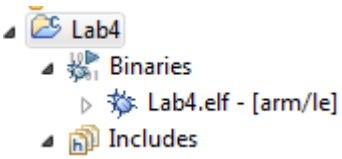
axi_oled_controller	0x41240000
axi_gpio_oled	0x41230000
axi_gpio_sws	0x41220000
axi_gpio_leds	0x41210000
axi_gpio_btns	0x41200000

2. Building the Lab Assembly Program

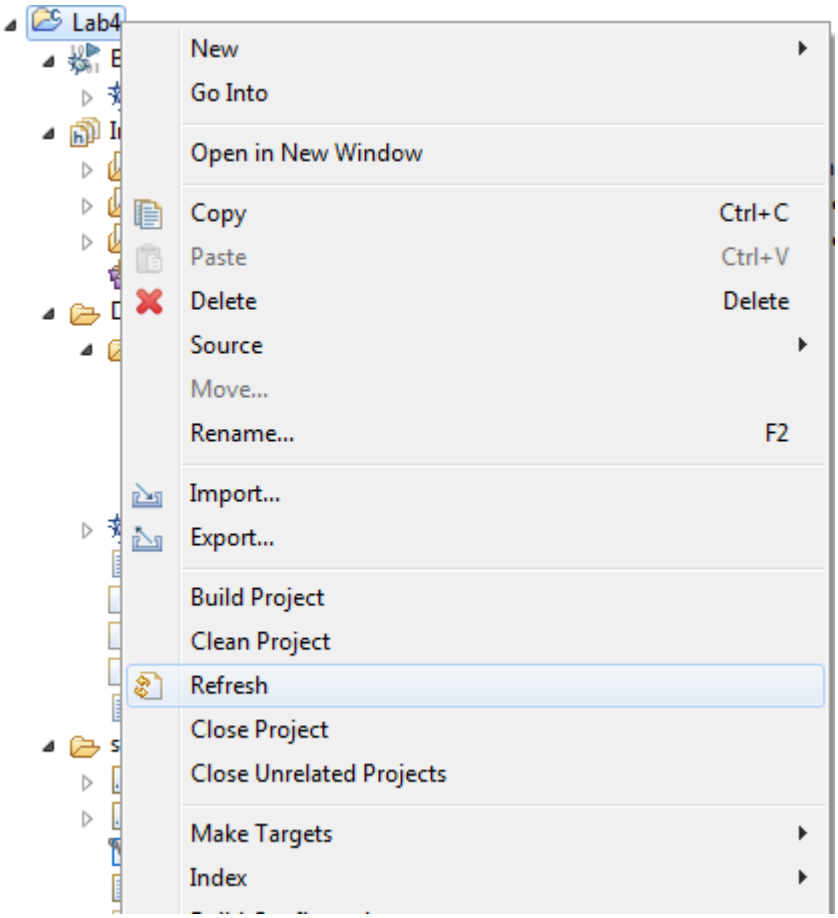
A simple lab application has been created in the workspace under the 'Lab4' directory. It contains in the src subdirectory the files main.c and asm.S. The main.c file simply calls the 'asm_main' subroutine/function in the assembly file.

To build the project, right click on the project in the explorer pane and select "Build Project"

If the compilation has been successful, a binary executable .elf file should be produced and you can move on to running or debugging your code on the hardware. If you do not see this .elf file under "Binaries",



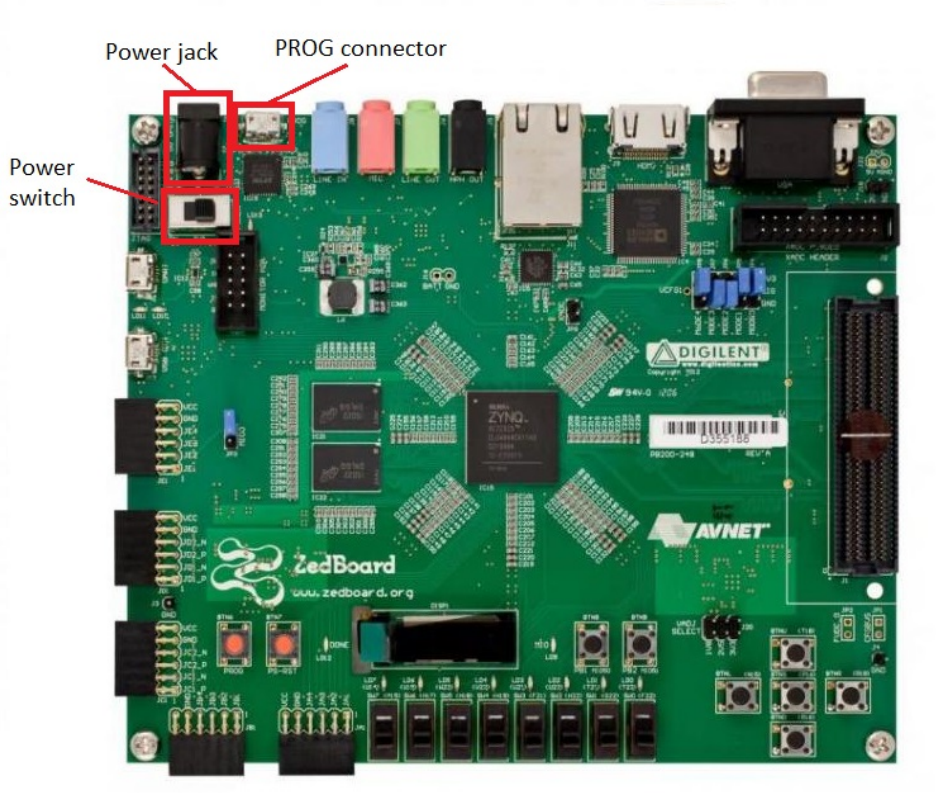
get the context menu for the Lab4 project and select "Refresh":



Do this a couple times if necessary. Before running or debugging you will need to connect some cables to your board.

Connect the power cable that came with your kit to the board. The power plug is right above the power switch on the top left of the board. Next connect a micro USB cable to the connector labeled PROG. This connector is between the power plug and the coloured audio jacks. Connect the other end to a USB port on your lab computer.

You can turn on the board by turning on the power switch. A green Power LED should light up on the board.



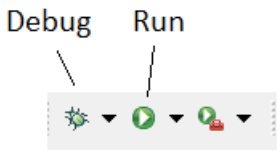
3. Running and Debugging your Program on the ZedBoard

Now that the board is setup we can download our code to the board and debug (or run) it. This is done by launching a 'Debug Configuration' for the lab program. A debug configuration called 'System Debugger using Debug_Lab4.elf on Local' has already been created for you.

To select it, use the following menu item:

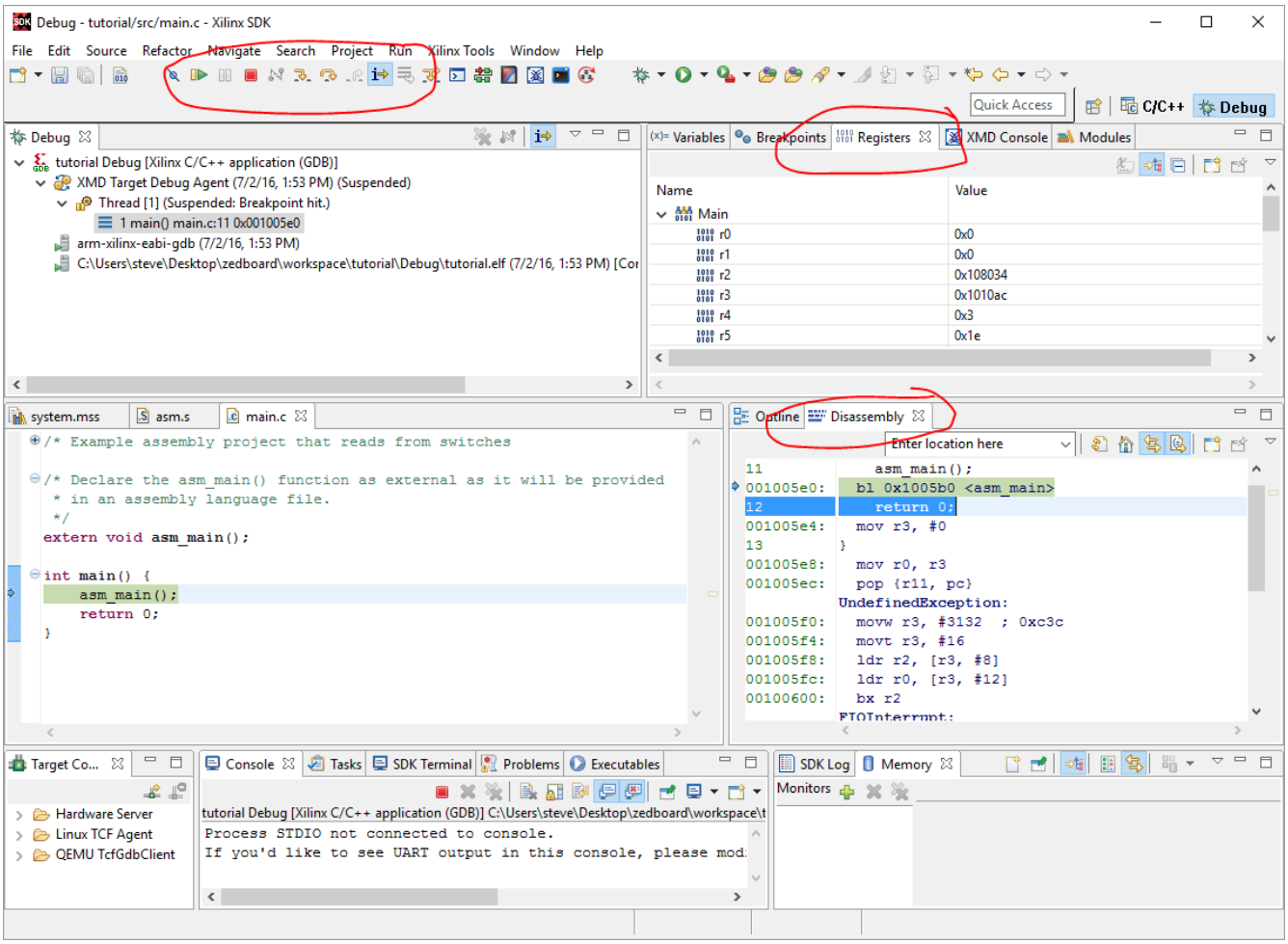
Run->Debug Configurations…

and then select 'System Debugger using Debug_Lab4.elf on Local'. Finally, click on the 'Debug' button. The next time you want to launch in the debugger, you can do so by selecting the debug icon in the tool bar shown below:



The FPGA programming and your ARM program are now being loaded in sequence on the ZedBoard through the ZedBoard USB connection. Downloading will take a few seconds so please wait patiently until the download has concluded. Check the SDK icon on the windows taskbar, it will show the progress of the FPGA and program load mechanism. Once the FPGA has been programmed a blue LED should light up and the OLED display should have a message on it.

In the 'debug perspective' you are able to see registers and other information, similar to the figure below. Make sure you can follow what is happening in your code, especially with the help of the Registers tab. ***



You also have the usual options of creating breakpoints, stepping through the code, or resuming execution.

If you have not already done so, use the 'Step Into' button to follow the asm_main() call into your assembly language file. After stepping through the code for a while, you can resume continuous execution using the 'Run->Resume' menu option or hitting F8 or using the appropriate icon button in the IDE's toolbar

When you are done debugging, you can edit your code, compile again and run again on the board. Since you have already used your 'System Debugger using Debug_Lab4.elf on Local' configuration once, the SDK has it in its history and you can easily repeat the process by selecting the triangle next to the 'Debug' icon and choosing 'Lab4 Debug' as shown below. Simply clicking on the debug icon will relaunch in the debugger the last program you had tried to launch.

4. Memory-mapped General Purpose I/O

The example code in this lab uses two sets of hardware peripherals: the slider switches and the LEDs. After you have completed it, the program will continuously read the values of the switches and turn on and off as necessary the corresponding LEDs. Both the switches and LEDs are connected to our ARM "microcontroller" system via General Purpose Input Output (GPIO) ports provided by the FPGA logic. These ports are named this way because they allow the processor to drive output signals or read input signals in a general purpose way. Each GPIO port has a memory mapped Data Register that allows the processor to read the state of input pins, and/or control the state of output pins that are connected to the switches or LEDs, as configured, by simply issuing load and store instructions.

As mentioned, a Data Register is used to read or write the value of the pins on a device port. If the pins for a port are configured as inputs (like for the GPIO port for the switches), then current port values can be read by reading the appropriate Data Register. For a port with pins configured as outputs, the port values can be set by writing to the Data Register for that port. The Data Register for a GPIO port is located at an offset of 0x0 from the GPIO port's base address. A preprocessor define, *XGPIO_DATA_OFFSET*, has been created for the Data Register offset.

On some systems, GPIO pins are configured as inputs when a device powers on. That is the case for us. To be able to drive the LEDs we have to set the LED GPIO pins as output pins. This is done by setting the corresponding LED bits in a 'Tristate Register' to zero. The Tristate Register is located at an offset of 0x4 from the GPIO's base address. Another preprocessor define, *XGPIO_TRI_OFFSET*, has been created for the Tristate Register offset.

The assembly code provided to you reads from switches and tries to output to LEDs. The base addresses of the GPIO ports have been defined as preprocessor defines via header files that have been included.. The assembly code loads those addresses into registers r3 and r1. Inside of a loop, the status of the switches is read from the Switch GPIO Data Register and we attempt to write this value to the LED GPIO Data Register. The desire is that the LEDs are continually updated to reflect the state of the switches.

You will notice that none of the LEDs turns on, however. This is because we have removed a line of assembly code from the file. We never actually clear the least-significant 8 bits before writing back to the Tristate Register for the LEDs. Fill in this missing line, rebuild the project, and test that you can get LEDs turning on according to how you set the switches.

5. Modify the Code to Read Buttons

Now let's modify the lab assembly file to read from the button data register instead of the slider switch register. The GPIO base address for the buttons is defined as *XPAR_AXI_GPIO_BTNS_BASEADDR*. Each individual button corresponds to a bit in the GPIO data register as shown in the table below.

Button Label	GPIO Data Bit	Define	Description
BTNC	0	BtnCMask	Center Button
BTND	1	BtnDMask	Down Button
BTNL	2	BtnLMask	Left Button
BTNR	3	BtnRMask	Right Button
BTNU	4	BtnUMask	Up Button

The defines are included via the file definitions.i -- while a button is being pressed its corresponding bit will read as a '1' in the data register.

Modify the assembly code to poll the button data register in the loop instead of the switch data register. If the 'up' button is pressed and then released your code should increment a counter variable that you need to allocate space for in an appropriate section of memory. While a button is being pressed down its corresponding bit in the GPIO Data register will read as a '1'. When released it will read as a '0'. If the 'down' button is pressed and released, then decrement the same counter. If the 'center' button is pressed and released, then reset the counter to zero. Display the current counter value on the LEDs by writing the count to the LED data register. Writing the count value to the LEDs will display the lower 8 bits of the count in binary so that you can see if your design is operating as expected.

6. Notes on the Debugging Facilities

While the Xilinx SDK comes included with a debugger, you need to remember that you are not running a software simulation but are monitoring the *actual status* of the Zynq chip on the board through a complex cabled connection.