

Induction Lab (Lab 1): Eclipse Project Setup

Introduction and Learning Outcomes

To gain proficiency with Eclipse for configuring projects tailored to run on specific FPGA development boards, you'll delve into its disassembly capabilities as well. This journey encompasses:

- Understand the full potential of Eclipse as your Integrated Development Environment (IDE), essential for efficiently managing FPGA and RTOS projects
- Uncover the intricacies of disassembling simple assembly programs within Eclipse
- Leverage Eclipse's disassembly window to update and refine assembly programs directly

Lab Overview:

In order to complete the lab, you will need to install the Quartus and Eclipse by following section 1 of [this](#) document. You can skip this step if you have already installed Quartus previously from ECE2072. Ensure you have *Nios II Software Build Tools for Eclipse (Quartus Prime 18.1)* installed.

Key Information and Notices:

On-campus Equipment Requirements per Group

- DE10 Lite board or a DE2-115 board
- Laptop (with Quartus installed) or Lab PC
- Nios II Software Build Tools for Eclipse (Quartus Prime 18.1)

Board	Family	Device Name
DE10 (take-home boards)	Max 10	10M50DAF484C7G
DE2 (on-campus boards)	Cyclone IV	EP4CE115F29C7

Estimated Time Commitment: 3 hours per student (note: this is only an estimate of how long the lab will take you)

Academic Integrity:

The ECE3073 labs are to be completed **individually**. You can and should discuss the questions and concepts with other students, however, **you may not:**

- write Hardware Description Language (HDL), code, or design circuits together with other students
- show your HDL to other students
- copy other students' work
- present work you have found online as your own (i.e. without attribution).

This is plagiarism and/or collusion, and is prohibited by Monash's academic integrity policy. Breaches of this policy attract serious consequences.

Collaboration policy:

- Each student should prepare their own work
- You should work with people around you during your lab, this is good
- Your solution should be your own, and you must be able to describe it to a demonstrator when being marked

Your lab quizzes must be done individually.

Submission:

As you progress through the lab, ensure that you demonstrate the understanding of your code to a demonstrator and get checked off for each task. Checkpoints are indicated by a checkbox (see figure below).

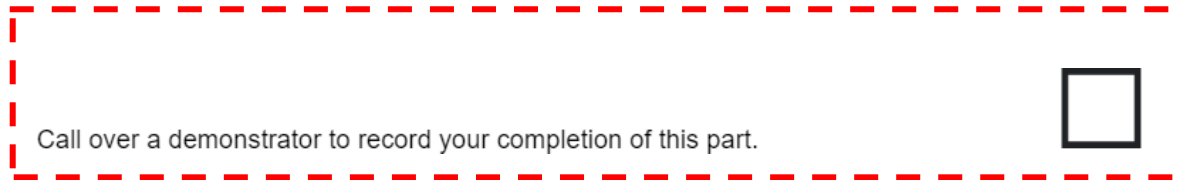


Figure 1. Example of a checkpoint

By the end of the lab, ensure that a demonstrator has entered your into the administrative spreadsheet. Marks will be updated on Moodle at the end of every week.

Student Progress Self-identification:

Lab 1 is a one week introductory lab. You must complete the lab within the first week of semester. Although there are no marks associated with lab 1, the processes introduced in lab 1 will be fundamental for the rest of the labs.

DE10-Lite board reminder:

When you receive your DE10-lite board, ensure that you take care of it, and note that it is a piece of loan equipment. This means that you are **expected to return the board** at the end of the semester. If you choose to withdraw from the unit, you are still expected to ensure that the board is returned to the ECSE department. Failure to return the board will result in a notification to the ECSE Department Manager, who will take steps to recover the equipment. These steps may include results and enrollment being encumbered until equipment has been returned, as well as future borrowing of equipment requiring additional approvals from the ECSE Department Manager.

When you need to return your board, you can do so at any laboratory session, workshop, or helpdesk.

Part 1 - Preliminary Work

Ensure you have collected all materials from Ms Fatiah/Mr Suresh. Installed Quartus 18.1.

Laboratory Activities:

Part 2 - Running your first program

2.1 Project Setup

In this first part, you will use the *Nios II Software Build Tools for Eclipse (Quartus Prime 18.1)* to download a pre-written hardware design and assembler program onto the DE-board. To prepare the workspace do the following:

1. Download the [lab0.zip](#) file from Moodle. This will contain the .sof file and .sopcinfo files required to program the DE-board.
2. Create a new folder for the files required for this exercise. Make sure the folder name is unique and the path **does not contain any spaces**. If you are using lab computers, don't save files on the desktop folder as it is a network mapped folder. Try "C:\Temp". (Please note that this folder gets cleaned on a daily basis)
3. Start the *Nios II Software Build Tools for Eclipse (Quartus Prime 18.1)* either using the desktop shortcut or by searching for it in the windows start menu. From now on, we will refer *Nios II Software Build Tools for Eclipse (Quartus Prime 18.1)* as "Eclipse"
4. When you open Eclipse, it will ask for a workspace, choose the folder you created for this project.

Now we will start programming the DE-10 or DE-2 board by doing the following:

1. In the Eclipse program, open Quartus Prime Programmer with "Nios II > Quartus Prime Programmer".
2. Once the Programmer window shows up, confirm that the hardware is detected in hardware setup. Click on **Add File ...** and navigate to **lab0** folder and look for your correct hardware i.e. if you use DE10 board, the file would be: **.\Lab0\hardware\DE10\DE10_Lite_Computer.sof**.
3. Click on the chip image and press **Start**, the board configuration will be downloaded to the physical device. Now the DE2 (or DE10) board should be configured as a Nios II microprocessor system. (If the **Start** button is greyed out, click on hardware setup and in the drop-down menu "Currently selected hardware" choose USB-blaster [USB-0])

If this is successful, you should see "100% successful" highlighted in the top right "progress" section.

2.2 Running the program

1. Create a new project with “File > New > Nios II Application and BSP from Template”
2. In this project enter in the details like in the figure below:
 - a. Within “Target hardware information > SOPC Information file name:” Click the “...”, and choose the .sopcinfo file for the board you are programming. It should be in “./hardware/DE2_115/” or “./hardware/DE10/”
 - b. Name this project lab0

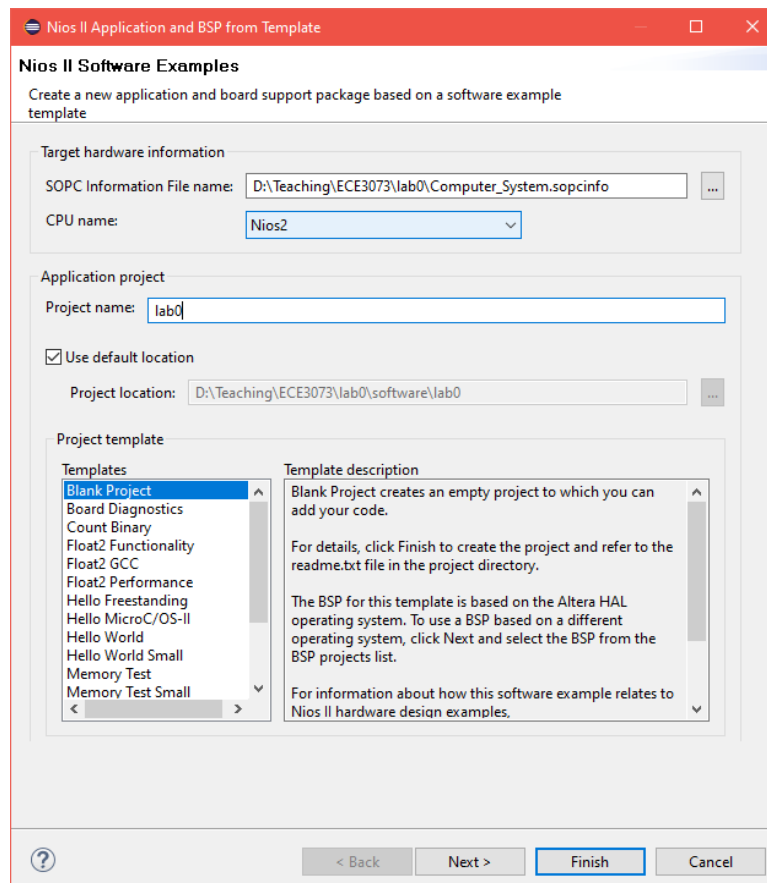


Figure 1: Nios II application from template setup window

3. Right click on “lab0” inside the project explorer.
 - a. Choose “New > file” and ensure lab0 is the parent folder
 - b. Name the file main.S

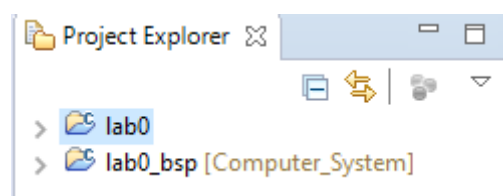



Figure 2: Project Explorer Tab

4. Copy the code within the “getting_started.s” file into the main.S file.
5. Repeat Step 3 but name the file “address_map_nios2.S”
 - a. Copy the contents of the file with same name into this file
6. Click on the “Build All Icon,  , or press “Ctrl + B”.

7. Download the compiled file to the board by selecting “Run > Run Configurations”. Double click Nios II Hardware. See Figure 3 as an example of what you will see.

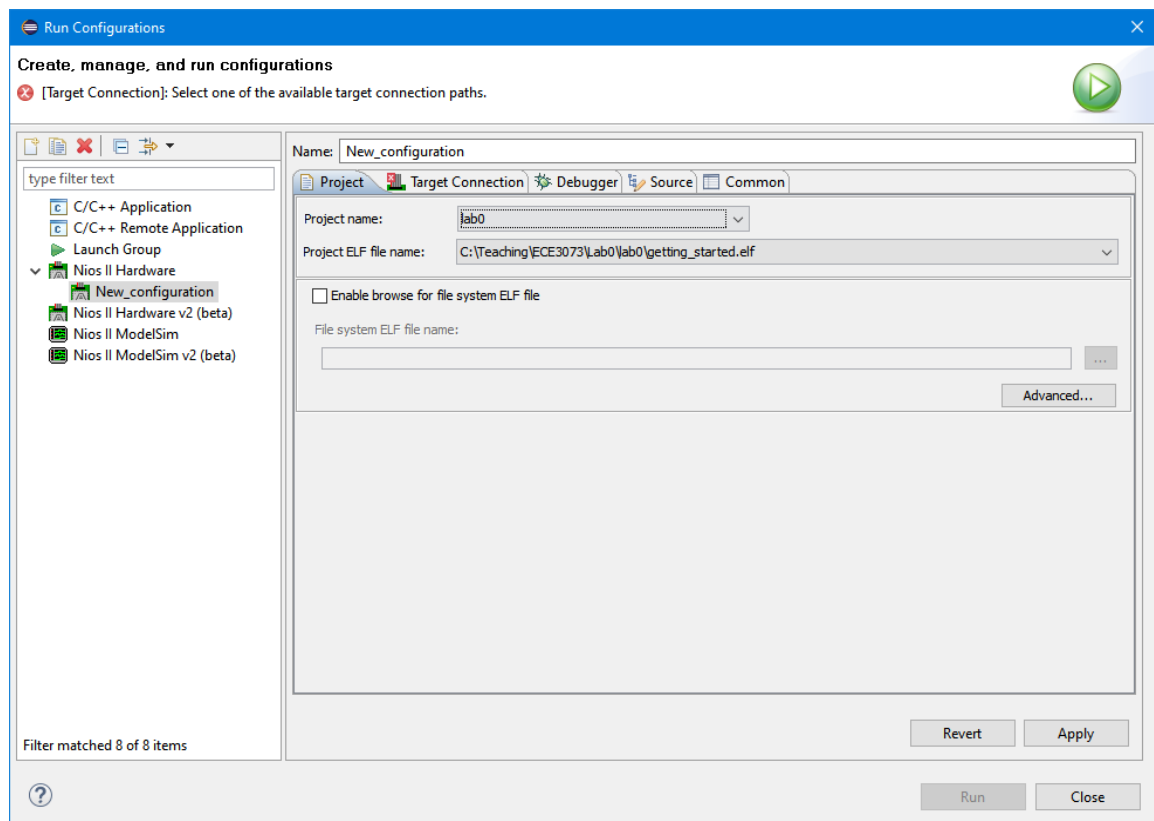


Figure 3: Run configurations

- a. Next, cycle to the “Target Connections” tab. Applying the following settings
 - i. Tick “Ignore mismatched system ID” and “Ignore mismatched system timestamp” shown in Figure 4. Then click “Refresh Connection” to display the device. The downloaded hardware configuration will contain two different options, either option will work.
 - ii. Click “Apply”, Then “Run”

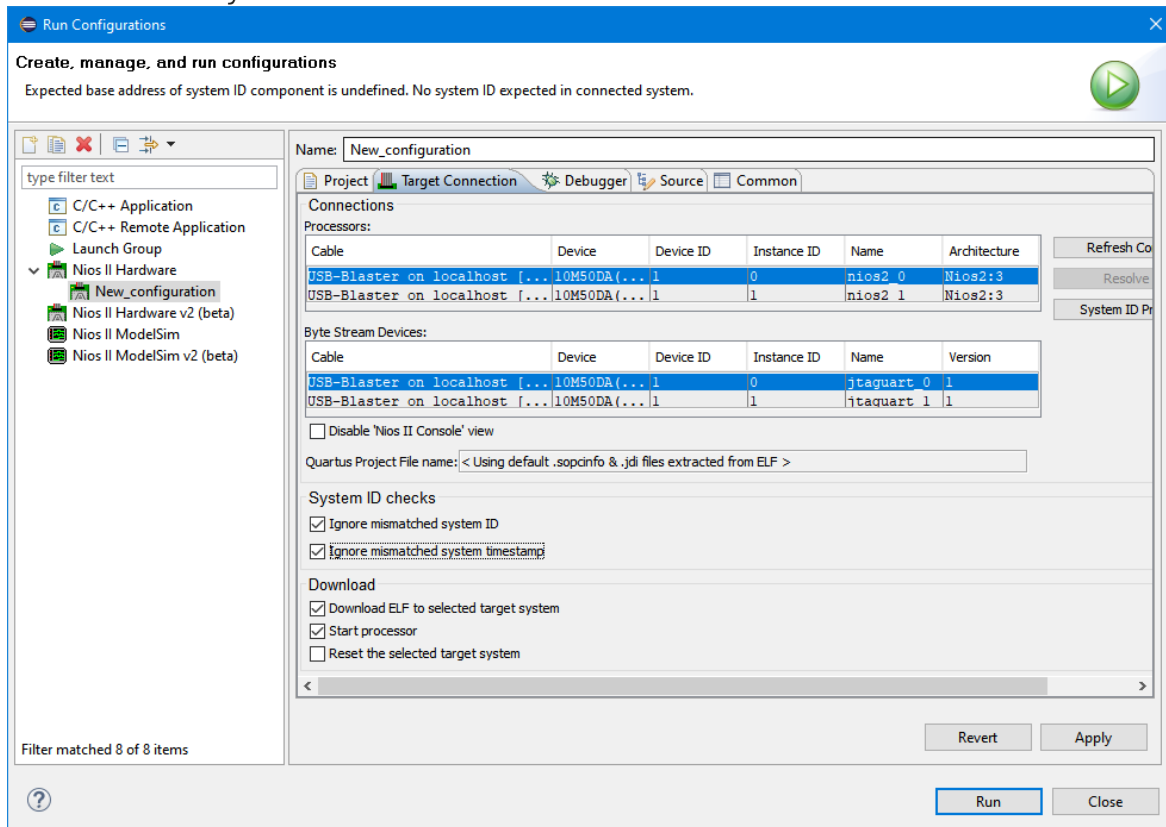


Figure 4: Target Connection Tab

8. Confirm that the program does the following:
 - a. Displays the DE2 (or DE10) board's SW switch states on the red lights (LEDR)
 - b. Shows a rolling LEDR display. If any of the KEYs are pressed, the pattern is changed to correspond to the settings of the SW switches.
 - i. Try setting the first 2 bits of the SW high and pressing the KEYs. What happens?

For DE10 boards: These only have 10 switches, so the value is referred to as "SW[9:0]". Also note that the behaviour of the SW to LEDR is slightly different. The DE10 sample program rolls the values in the 8 left-most switches.

In *Nios II Software Build Tools for Eclipse (Quartus Prime 18.1)*, there is no way to stop the assembly code once you run it. To run a different program you can simply load the new software and the current program will be overwritten.

Call over a demonstrator to record your completion of this part.



Template code for part 2 (getting_started.s)

```
.include      "address_map_nios2.s"

/*****
*****
* This program demonstrates use of parallel ports
*
* It performs the following:
*   1. displays a rotating pattern on the LEDs
*   2. if any KEY is pressed, the SW switches are used as the
rotating pattern
*****
*****/

.text                          # executable code follows
.global      main
main:
    NOP
/* initialize base addresses of parallel ports */
    movia    r15, SW_BASE      # SW slider switch base
address
    movia    r16, LED_BASE     # LED base address
    movia    r17, KEY_BASE     # pushbutton KEY base address
    movia    r18, LED_bits
    ldwio    r6, 0(r18)        # load pattern for LED lights

DO_DISPLAY:
    ldwio    r4, 0(r15)        # load slider switches

    ldwio    r5, 0(r17)        # load pushbuttons
    beq      r5, r0, NO_BUTTON
    mov      r6, r4            # copy SW switch values onto
LEDs
    roli     r4, r4, 8         # the SW values are copied
into the upper three
                                # bytes of the pattern
register
    or       r6, r6, r4        # needed to make pattern
consistent as all
                                # 32-bits of a register are
rotated
    roli     r4, r4, 8         # but only the lowest 8-bits
are displayed on
                                # LEDs
    or       r6, r6, r4
    roli     r4, r4, 8
    or       r6, r6, r4

WAIT:
    ldwio    r5, 0(r17)        # load pushbuttons
    bne      r5, r0, WAIT      # wait for button release
```

NO_BUTTON:

```

        stwio    r6, 0(r16)          # store to LED
        roli    r6, r6, 1           # rotate the displayed
pattern
        movia    r7, 1500000         # delay counter
DELAY:
        subi     r7, r7, 1
        bne      r7, r0, DELAY

        br       DO_DISPLAY

EXTRA:
        nop
        nop
        nop
        nop
        nop

/*****
*****/
.data                                # data follows

LED_bits:
.word      0x0F0F0F0F

.end

```

End of Week 1 Task

To be completed by week 2

Preliminary Work

Assume a 16-bit number is stored in register `r4`. Using the [Instruction Set Reference](#), write Nios II assembler instructions that will modify the bits stored in Register `r4` that will:

- (a) Invert bits 1 and 0
- (b) Set (turn on) bits 5 and 4
- (c) Reset (turn off) bits 7 and 6

All other bits should be left unchanged. Finish off this piece of code with a jump immediate to address `0x0000002c`:

(d) `jmp i 0x0000002c`

Write down the instructions and then convert these into machine code using the instruction set reference. These will be used for Exercise 3. Write or the machine codes for

(a)

(b)

(c)

(d)

Part 3 - Debugging Assembly

3.1 Debug Setup

In this second exercise, we want to have a closer look into the assembly code under the hood. We will achieve that through the debugger that comes with *Nios II Software Build Tools for Eclipse (Quartus Prime 18.1)*.

To debug the assembly file:

1. Open the file "getting_started.s" file in the file explorer menu.

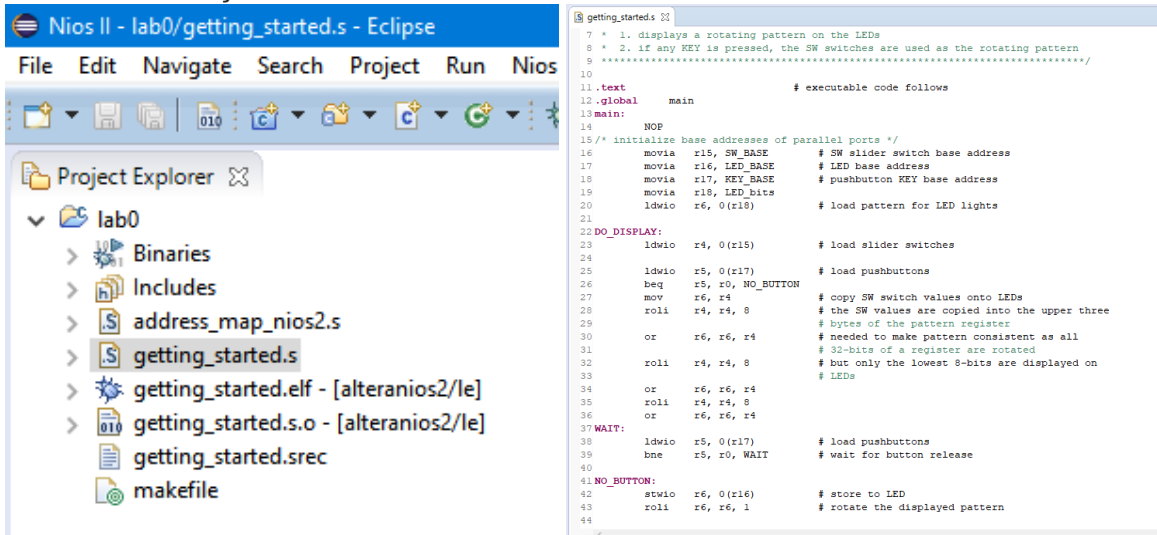


Figure 5: Opening the assembly code

- Place a breakpoint on line 14 (by double clicking), you will see a breakpoint inserted like so:

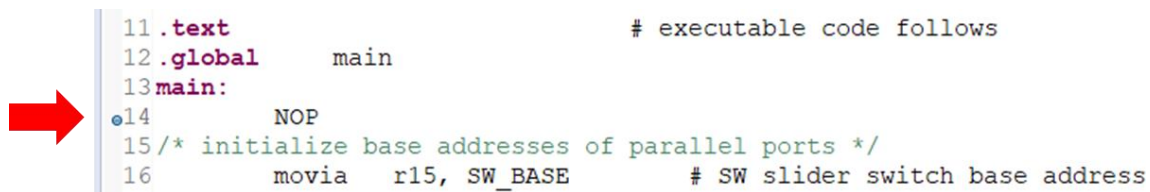


Figure 6: Inserting a breakpoint at line 14.

3. Go to “Run > Debug Configuration”. Click **Debug** in the window shown:

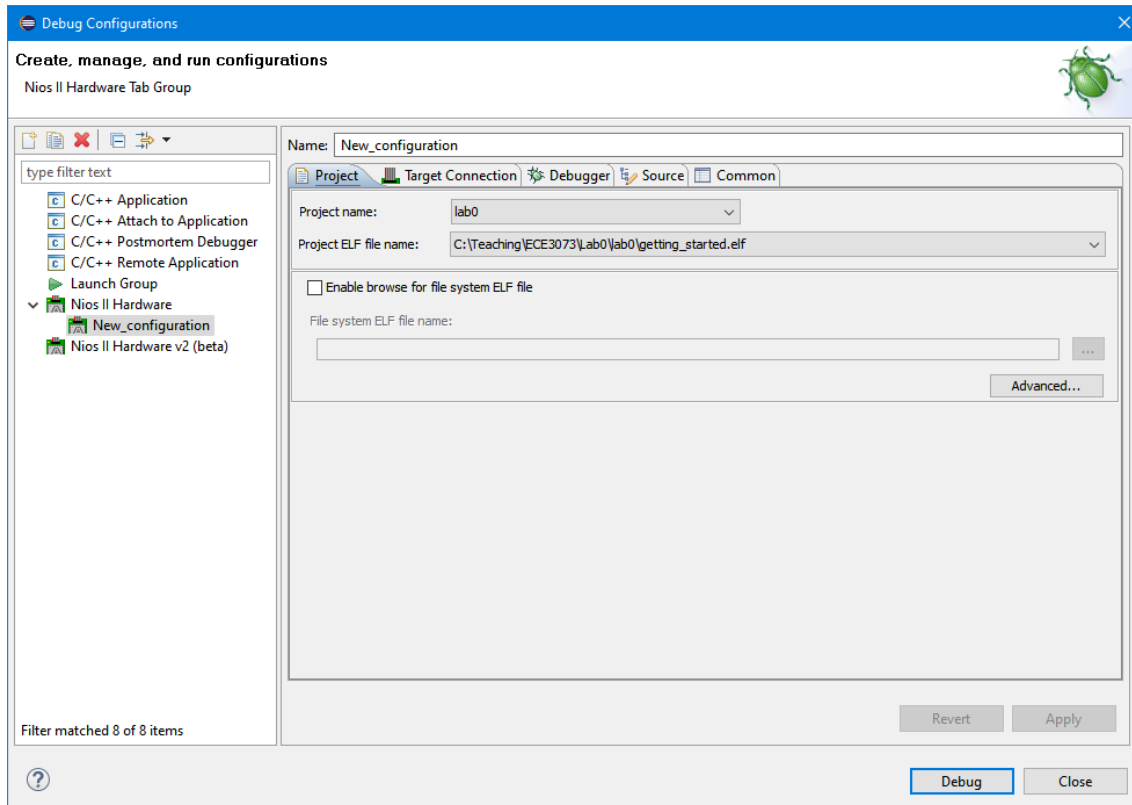


Figure 7: Debug Window

A window will pop up asking if you want to change to Debug Perspective. Please choose “Yes”.

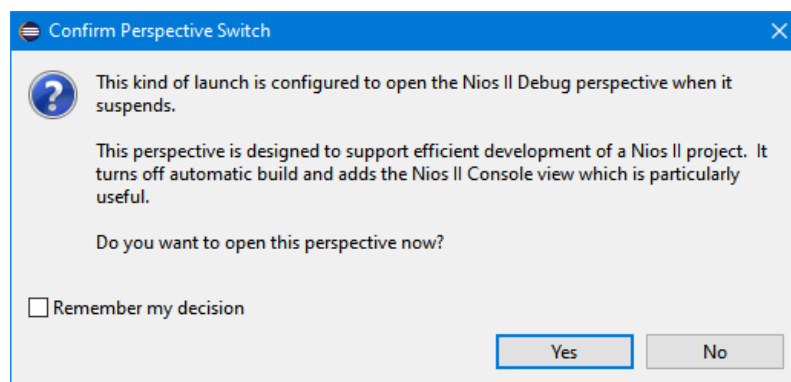


Figure 8: Perspective switch window

You should now see something like so:

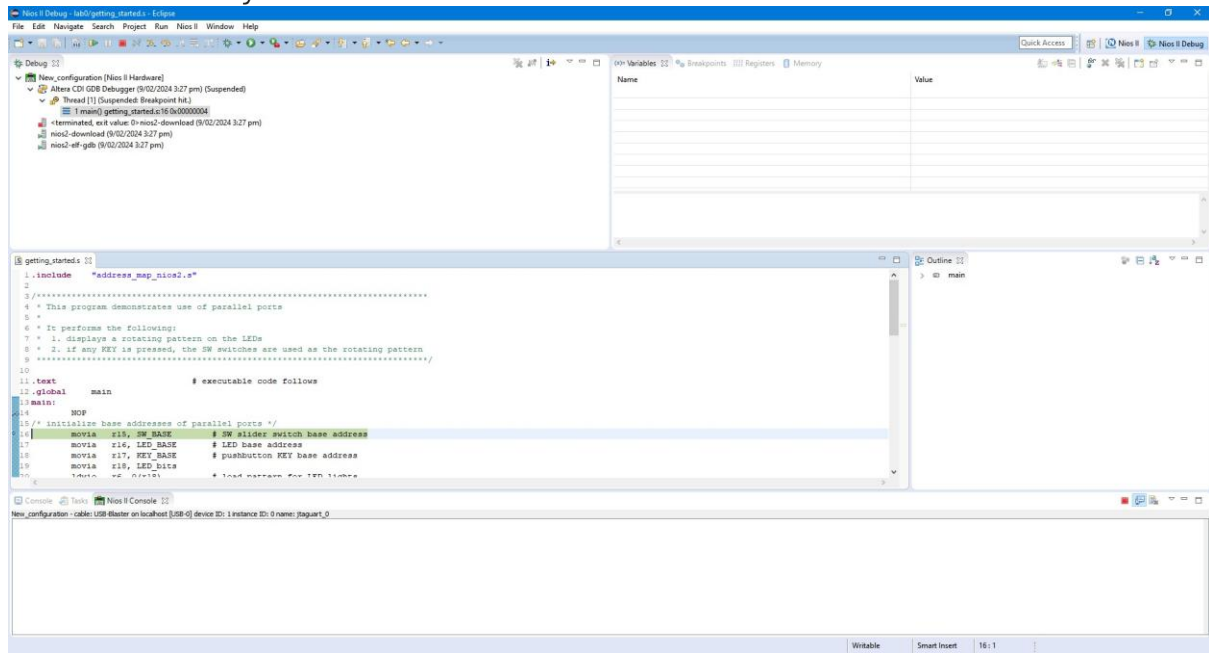


Figure 9: Debug Perspective

We will dive deeper into breakpoints and the debugger in [section 3.3](#).

3.2 Using the Debugger

Using the Disassembly Window:

1. Pseudo operations are instructions where the original source code instruction is not the same as the disassembled instruction/s as displayed in the Disassembly Window. Use this fact (or refer to the Nios instruction reference on Moodle) to identify all of the pseudo operations in "getting_started.s" and highlight them on the program listing in the [getting_started.s section](#).
2. To view the Disassembly window, Go to "Windows > Show View > Other..." Then choose "Debug > Disassembly". Then you can drag the Disassembly window so that it can be next to other locations such as this:

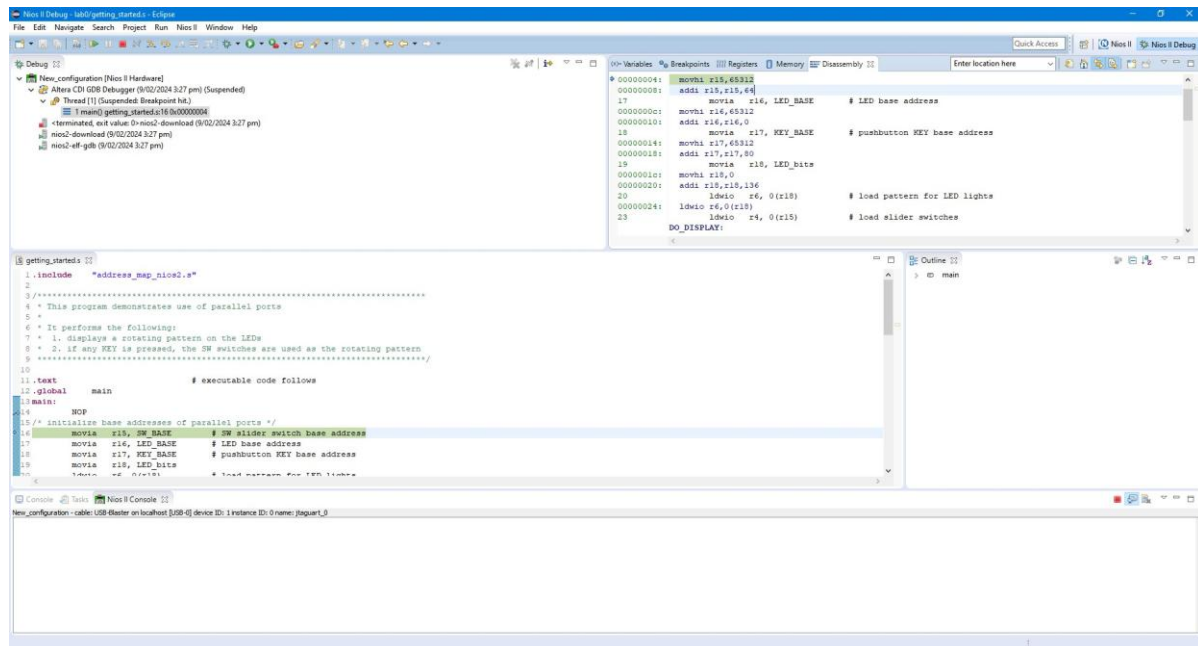


Figure 10: Added Disassembly viewer

Single Stepping is a key tool for debugging as it allows you to see what the code is doing line-by-line and diagnose the specific point at which errors occur. To do so:

1. Single step the program up to the delay loop (you can do that with step into or step over highlighted in Figure 10):

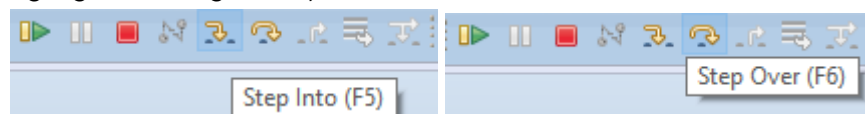


Figure 11: Debugger options with shortcuts

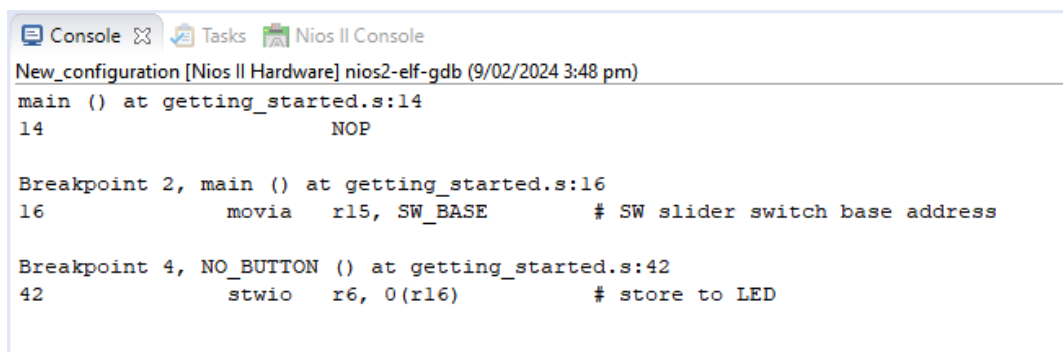
2. Before you single step the program you should anticipate what the result of the next instruction will be in terms of changes to registers including the program counter, changes to memory, lights on the DE2 (or DE10) board, etc. After each single step check that you anticipated correctly.

(By default, the Register view is displayed in decimal, you can `Right Click > Format` any register and change it to Hexadecimal.)

3.3 Deeper dive into the debugger

Using Breakpoints – An instruction breakpoint provides the means to stop a Nios II program when it reaches a specific address in the assembly code. A simple procedure for setting an instruction breakpoint is:

1. In the Disassembly window, scroll to display the instruction address that will have the breakpoint. As an example, scroll to the instruction at the label `NO_BUTTON`, which is address `0x00000058`. In a larger program, it may be easier to enter the address directly. You can do this by entering the hexadecimal address into the “Enter Location Here” option. For example, we could have entered “0x58” to jump to the same instruction as before.
2. Double click on the space left of the address `0x00000058` in the Disassembly window. The window displays a blue dot next to the address to show that an instruction breakpoint has been set. Double clicking the same location again removes the breakpoint.
3. Once the instruction breakpoint has been set, run the program. The breakpoint will trigger when the pc register value equals `0x00000058`. Control then returns to the Debugger, and the Disassembly window highlights in green the instruction at the breakpoint. A corresponding message is shown in the Console pane:



```

Console Tasks Nios II Console
New_configuration [Nios II Hardware] nios2-elf-gdb (9/02/2024 3:48 pm)
main () at getting_started.s:14
14          NOP

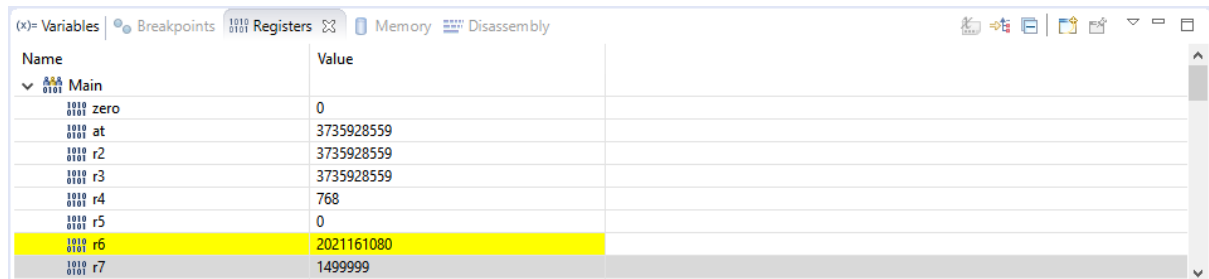
Breakpoint 2, main () at getting_started.s:16
16          movia    r15, SW_BASE          # SW slider switch base address

Breakpoint 4, NO_BUTTON () at getting_started.s:42
42          stwio    r6, 0(r16)           # store to LED
  
```

Figure 12: Console pane after stopping at breakpoint

3.3.1 Examining and changing register contents

1. The Registers tab, displayed in Figure 12, shows the value of each register in the Nios II processor and allows the user to edit most of the register values. The number format of the register values can be changed by right-clicking each register and selecting “Change Value...”. Each time program execution is halted, the Debugger updates the register values. If you select another register, the register you just updated will be highlighted in yellow. The user can also edit the register values while the program is halted. Any edits made are visible to the Nios II processor when the program’s execution is resumed.



Name	Value
0101 0101 Main	
0101 0101 zero	0
0101 0101 at	3735928559
0101 0101 r2	3735928559
0101 0101 r3	3735928559
0101 0101 r4	768
0101 0101 r5	0
0101 0101 r6	2021161080
0101 0101 r7	1499999

Figure 13: Registers Tab

2. As an example of editing a register value, first scroll the Disassembly window to the label DELAY, which is at address 0x00000068. Set a breakpoint at this address and then run the program. After the breakpoint triggers and control returns to the Register tab, notice that there is a large value in register r7. This value is used as a counter in the delay loop (it will decrement at every iteration). Single Click (or “Right Click > Change Value...”) on the contents of register r7 and edit it to the value 1. Press Enter on the computer keyboard, or click away from the register value to apply the edit. Now, single-step the program to see that it exits from the delay loop after one more iteration, when r7 becomes 0.

Call over a demonstrator to record your completion of this part.

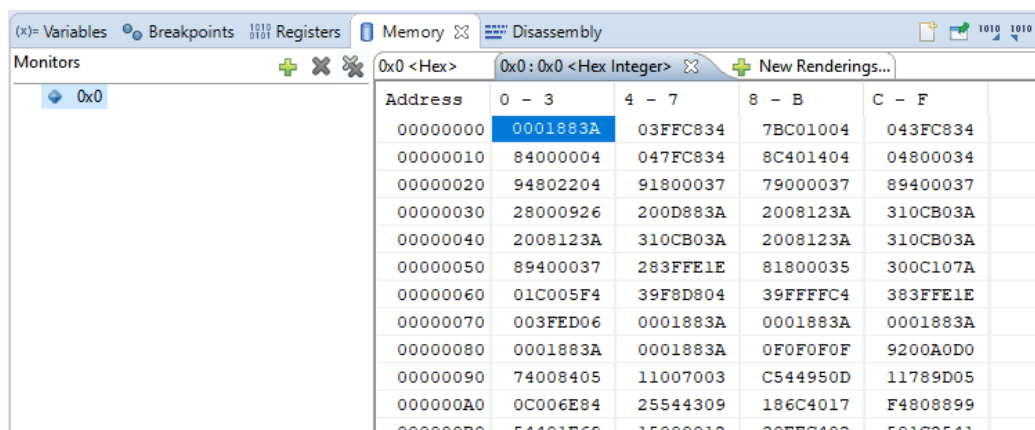


Part 4 - Modifying Machine Code in the Debugger

In this last exercise, you are asked to modify the assembler code in “getting_started.s” and use the functions available in the Debugger to test your modifications. The changes to the program will be made by directly adding NIOS machine code instructions to program memory.

As it originally stands, “getting_started.s” reads the state of the slider switches on the DE2 (or DE10) board and then immediately writes those values to the red LEDs. The changes that you will make to the program will modify the data read from the slider switches before writing it to the LEDs. These modifications are described in the preliminary work section. You should already have machine code instructions for performing these operations in answer to the preliminary work (Part 1).

To view the memory, go to the memory tab, click on the Green Plus (Add Memory Monitor). Type 0x0 to the field, then click “OK”. A view of the memory block neighbouring address 0x0 will show up. Click on “New Renderings...” and Choose “Hex Integer”. “Hex Integer” will display the memory content in hexadecimal which is consistent with how we display these values in the workshops and preliminary work. You should see something like this



Address	0 - 3	4 - 7	8 - B	C - F
00000000	0001883A	03FFC834	7BC01004	043FC834
00000010	84000004	047FC834	8C401404	04800034
00000020	94802204	91800037	79000037	89400037
00000030	28000926	200D883A	2008123A	310CB03A
00000040	2008123A	310CB03A	2008123A	310CB03A
00000050	89400037	283FFE1E	81800035	300C107A
00000060	01C005F4	39F8D804	39FFFFC4	383FFE1E
00000070	003FED06	0001883A	0001883A	0001883A
00000080	0001883A	0001883A	0F0F0F0F	9200A0D0
00000090	74008405	11007003	C544950D	11789D05
000000A0	0C006E84	25544309	186C4017	F4808899
000000B0	54401F69	15000012	30FFC402	501C2541

Figure 14: Memory Monitor

All of the required changes are as follows:

1. Overwrite the `ldwio r4,0(r15)` instruction at label `DO_DISPLAY`: with an unconditional branch “`br`” to memory address `0x00000074`. Of course you can work out the corresponding machine code to practice this however the machine code is given below. As well, the changed memory address is shown in Figure 14 below. Make sure to note down the machine code value that you replaced as you will use it later.

(Note: to change the machine code at this location you will need to change in the Memory tab instead of the Disassembly tab. Record your machine code for `ldwio r4,0(r15)` and branch instruction here: `ldwio r4,0(r15):79000037,br 0x74:00001206)`

0x0 <Hex>	0x0: 0x0 <Hex Integer>	+ New Renderings...		
Address	0 - 3	4 - 7	8 - B	C - F
00000000	0001883A	03FFC834	7BC01004	043FC834
00000010	84000004	047FC834	8C401404	04800034
00000020	94802204	91800037	00001206	89400037
00000030	28000926	200D883A	2008123A	310CB03A
00000040	2008123A	310CB03A	2008123A	310CB03A
00000050	89400037	283FFE1E	81800035	300C107A
00000060	01C005F4	39F8D804	39FFFFFFC4	383FFE1E
00000070	003FED06	0001883A	0001883A	0001883A
00000080	0001883A	0001883A	0F0F0F0F	9200A0D0
00000090	74008405	11007003	C544950D	11789D05
000000A0	0C006E84	25544309	186C4017	F4808899
000000B0	54401F69	15000012	30FFC402	501C2541

Figure 15: Changed machine code

- Write the machine code for the `ldwio r4, 0(r15)` instruction that you overwrote in part (1) at memory location `0x00000074`.
- Then in the following successive memory locations (groups of 4 bytes) insert the 4 machine code instructions **from your preliminary work**. You should see something like Figure 15, though note these values are not correct.

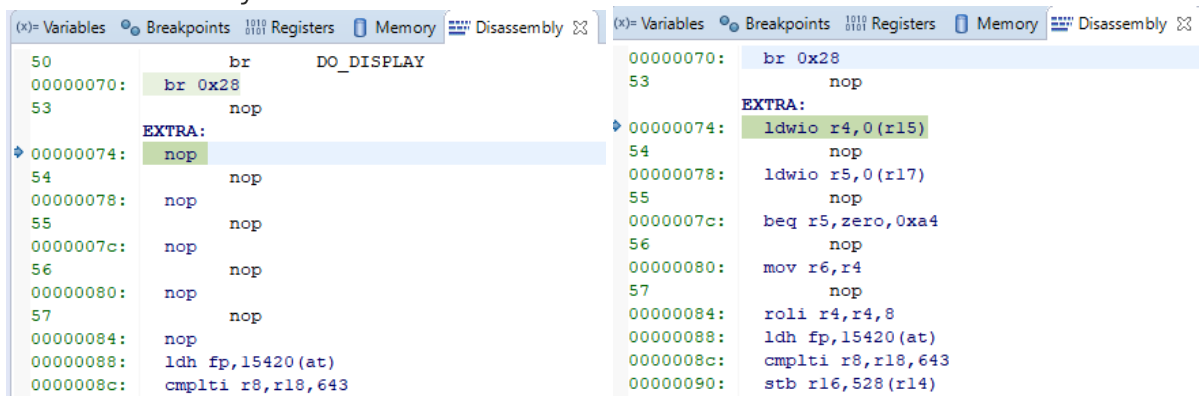
0x0 <Hex>	0x0: 0x0 <Hex Integer>	+ New Renderings...		
Address	0 - 3	4 - 7	8 - B	C - F
00000000	0001883A	03FFC834	7BC01004	043FC834
00000010	84000004	047FC834	8C401404	04800034
00000020	94802204	91800037	00001206	89400037
00000030	28000926	200D883A	2008123A	310CB03A
00000040	2008123A	310CB03A	2008123A	310CB03A
00000050	89400037	283FFE1E	81800035	300C107A
00000060	01C005F4	39F8D804	39FFFFFFC4	383FFE1E
00000070	003FED06	79000037	89400037	28000926
00000080	200D883A	2008123A	0F0F0F0F	9200A0D0
00000090	74008405	11007003	C544950D	11789D05
000000A0	0C006E84	25544309	186C4017	F4808899
000000B0	54401F69	15000012	30FFC402	501C2541

Figure 16: Further changes to machine code

Note that the Disassembly window will not immediately reflect the changes that you have made to memory. To disassemble your added code, click on the refresh view on the top right corner of the tab and you should see changes in the disassembly after the "Extra" label similar to those in Figure 17 below. Again note that these instructions are not correct.



Figure 17: Refreshing Disassembly View



Address	Instruction
50	br DO_DISPLAY
00000070:	br 0x28
53	nop
EXTRA:	
00000074:	nop
54	nop
00000078:	nop
55	nop
0000007c:	nop
56	nop
00000080:	nop
57	nop
00000084:	nop
00000088:	ldh fp,15420(at)
0000008c:	cmplti r8,r18,643

Address	Instruction
00000070:	br 0x28
53	nop
EXTRA:	
00000074:	ldwio r4,0(r15)
54	nop
00000078:	ldwio r5,0(r17)
55	nop
0000007c:	beq r5,zero,0xa4
56	nop
00000080:	mov r6,r4
57	nop
00000084:	roli r4,r4,8
00000088:	ldh fp,15420(at)
0000008c:	cmplti r8,r18,643
00000090:	stb r16,528(r14)

Figure 18: Changes after refreshing disassembly window

Single step through your code to ensure that it works correctly before selecting run mode. Try setting the first 2 bits of the SW high and pressing the KEYs. What happens?



Call over a demonstrator to record your completion of this part.