Eugene Ngo

EE 469

May 26, 2023

Lab 5 Report

Procedure:

Task 1: Sample C code for Peripheral Devices

Part 1: Sample C Program for LEDs

1. What does the (volatile long *) keyword mean?
   Volatile long means that the variable is declared as a long that can be changed at any point. Longs are also double the length of an int, thus, in the case of a 32-bit language, the long is declared as a 64-bit value. The volatile keyword makes it so that the compiler ignores optimizations that essentially reduce variables down to register values because a volatile variable is not just a variable that's stored, it is changed often, hence the name. Thus, the volatile long* declaration, declared "LED" as an address pointing to a volatile long, a 64 bit "integer" that is changed often, thus it ignores optimizations that just reduce it to a register value.

2. Verify that this program works as intended in the CPUlator. **Make sure that the Language set on the CPUlator is in C.** Take a screenshot of the LEDs, showing that the right-most LED is illuminated. Make sure to include this screenshot in your report.

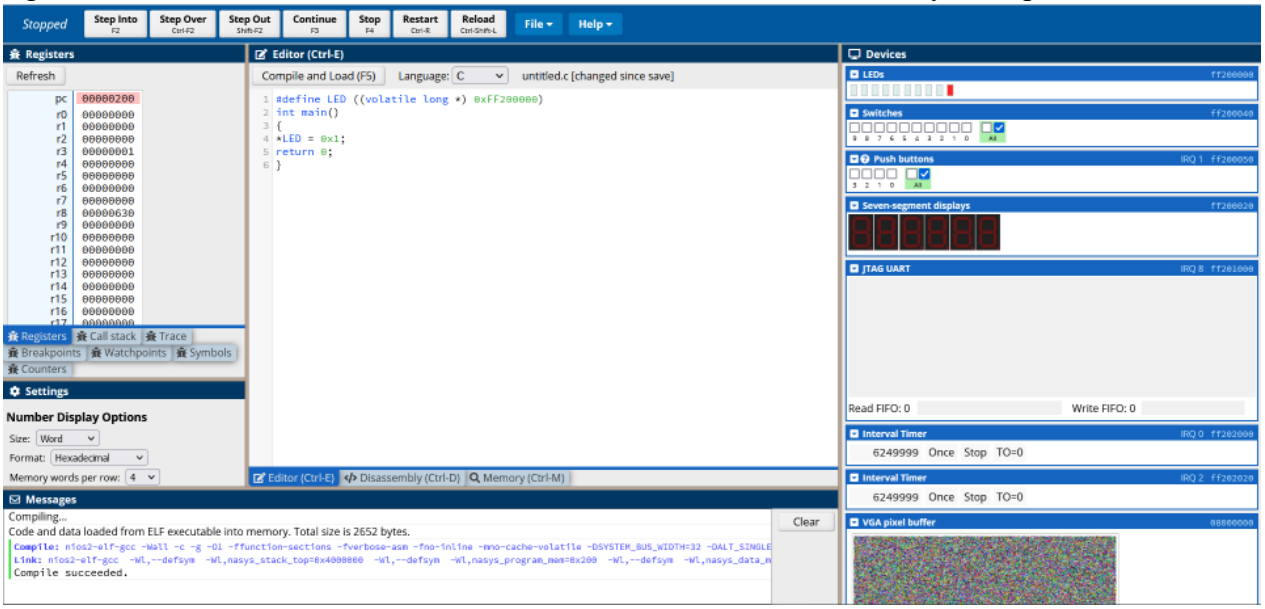

Figure 1: Full Screenshot of provided LED Sample code

Figure 2: Zoomed in on LED panel of Figure 1

Figure 1 shows the full screenshot after the provided LED sample code is run and figure 2 is a zoomed in screenshot of figure 1, specifically on the LED panel showing that the code runs as expected and illuminates the right-most LED.

3. Modify this example code to illuminate all 10 of the LEDs on the NIOS board. What value did you choose to write to *LED? Why? Take a screenshot of the LEDs, showing that all of the LEDs are illuminated. Make sure to include this screenshot in your report.

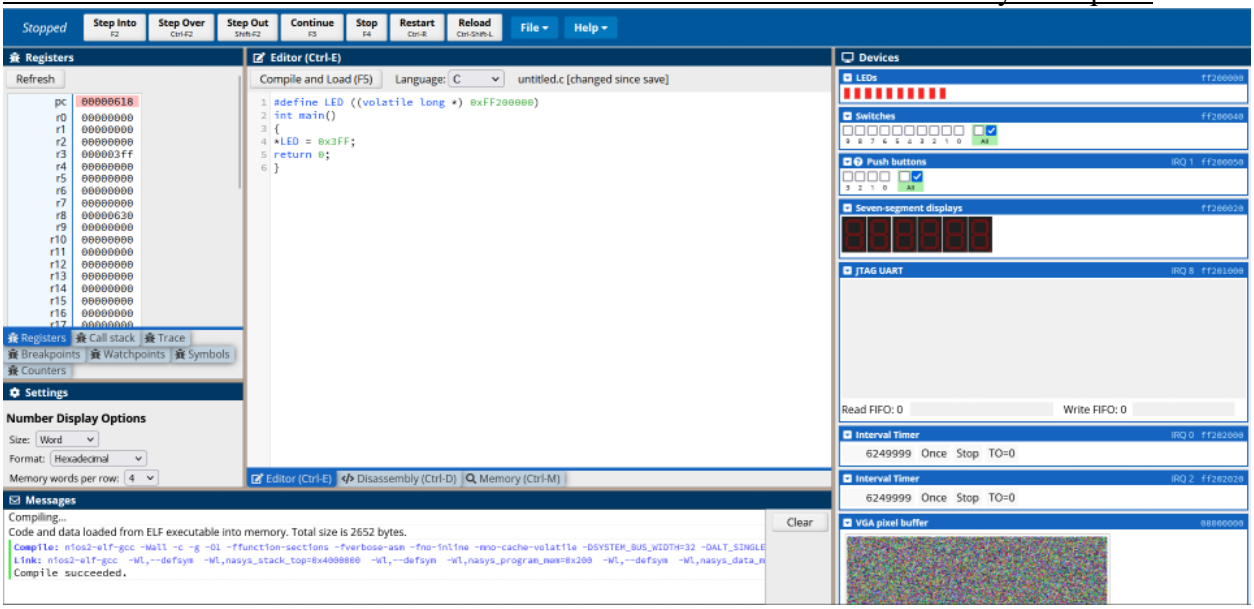

Figure 3: Full Screenshot of adjusted code & result

Figure 4: Zoomed in on LED panel of Figure 3

The value used to illuminate all 10 LEDs was 3FF as seen in Figure 3. 3FF is the HEX representation of binary 1111111111 which represents a 1 for each LED resulting in all the LEDs being turned on as seen in Figure 4.

Part 2:

1. Compile this program. In the Disassembly tab, continuously Step Over the generated Assembly code until you are looping through the main branch.
   Okay.

2. What is the PC address of the main branch? Why is it not at address 0?
   PC address is 230 for main. That's because the define preprocessor statement and all the setup code must run first before the assembly can reach the main method. The assembly must reach a one-to-one equivalent to the C code, thus it must have a long pre-processor section for linking files, etc. Thus, PC for main is 230 and not 0.

3. Why is this while(1){} loop necessary?
   The while (1) loop is necessary to ensure that the switch is being constantly monitored for the value produced by the switches.

4. Which temporary register (R0, R1, R2, etc.) holds the value of Swval?
   R2

5. Notice that when we do a Step Over in the main branch, the temporary register R2 toggles between three different hexadecimal values. What's the significance of these three different values that you see?
   The three values that is toggled through are: 0XFF200000, 0XFF200040, and 0 (the value of the switches). The significance between these values is that 0XFF200040 is the number corresponding to the register of the switches so it uses the value in 0XFF200040 to load the value in the 0XFF200040 external register to the R2 register which then gets the 0 value. The 0XFF200000 that it starts with is just the default base register that corresponds to the LEDs which I assume the compiler defaults to because it is common for the compiler to first set the R2 value to the base value of the external registers (0XFF200000) and then use an addi instruction to travel to the register that is required (0XFF200040) and then uses the ldwio instruction to load the value from that register in.

Task 2:

Part 1:

Task 2 Part 1 was to take the sample C code from Task 1 and convert it into taking in inputs from the push buttons and mapping them to the LEDs. Similarly to Task 1, I used define statements for the push button and LED registers. Then, the while loop for constantly mapping the LEDs to the push button register's value was taken from Task 1 Part 2's while loop. The actual register addresses used were derived by looking at the CPULator UI and the LEDs and Pushbuttons have their addresses located on next to them on the top right. The questions in the spec were answered in the 'Results' section below. Below are screenshots showing various combinations of push

buttons being on and off and the corresponding LEDs responding to demonstrate the functionality of my code.

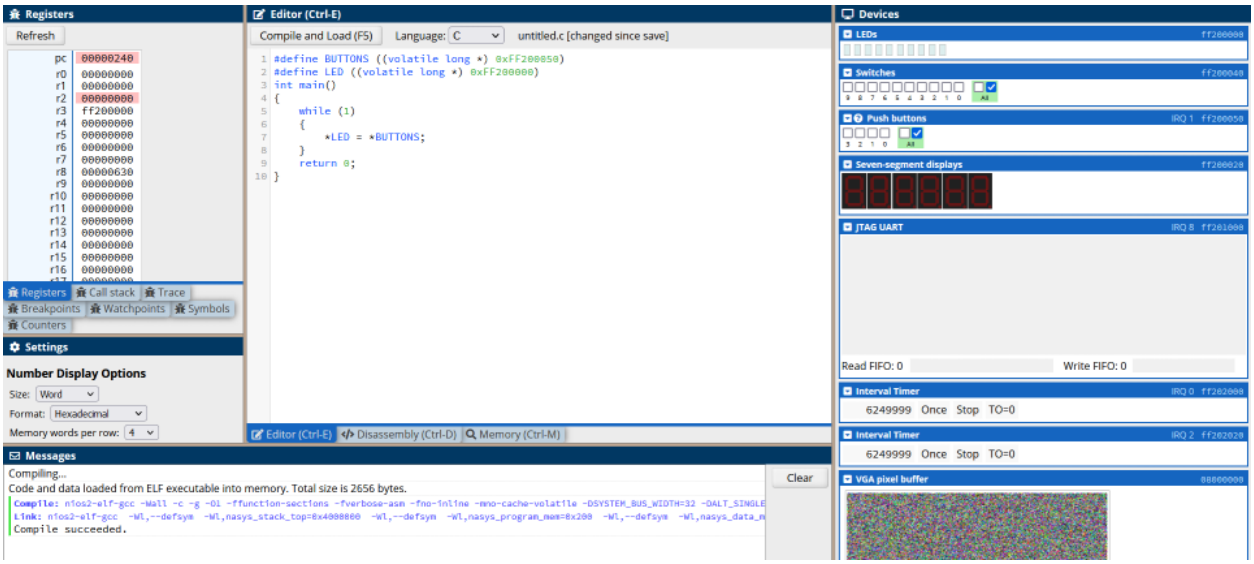

Figure 5: All push buttons off and all LEDs off

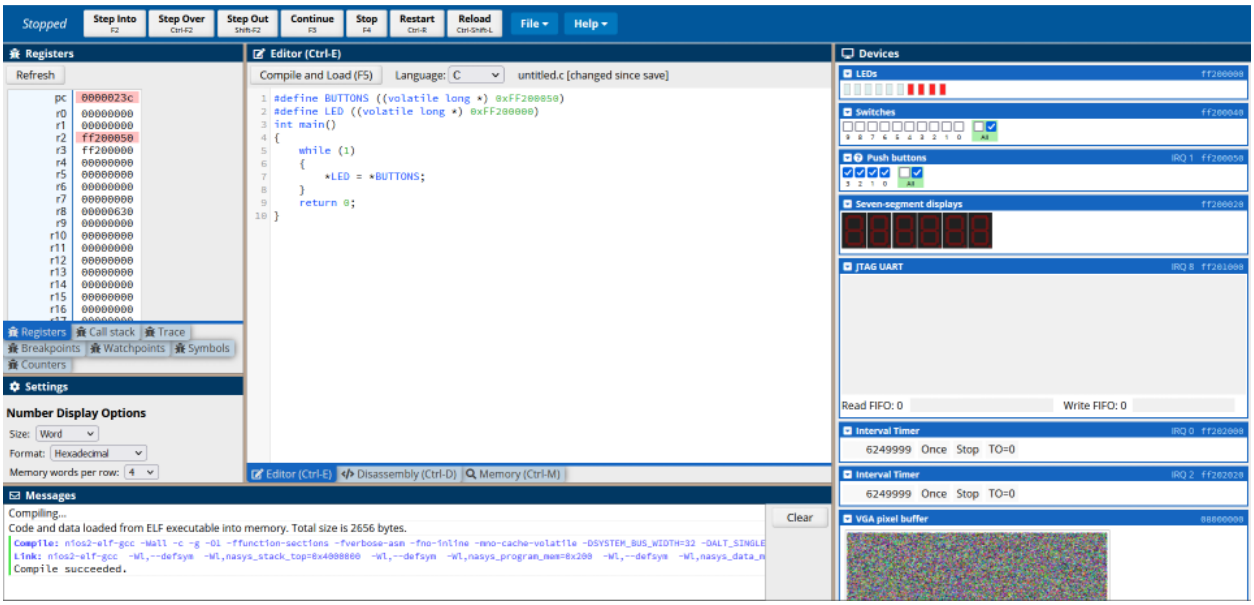

Figure 6: All push buttons on and all LEDs on

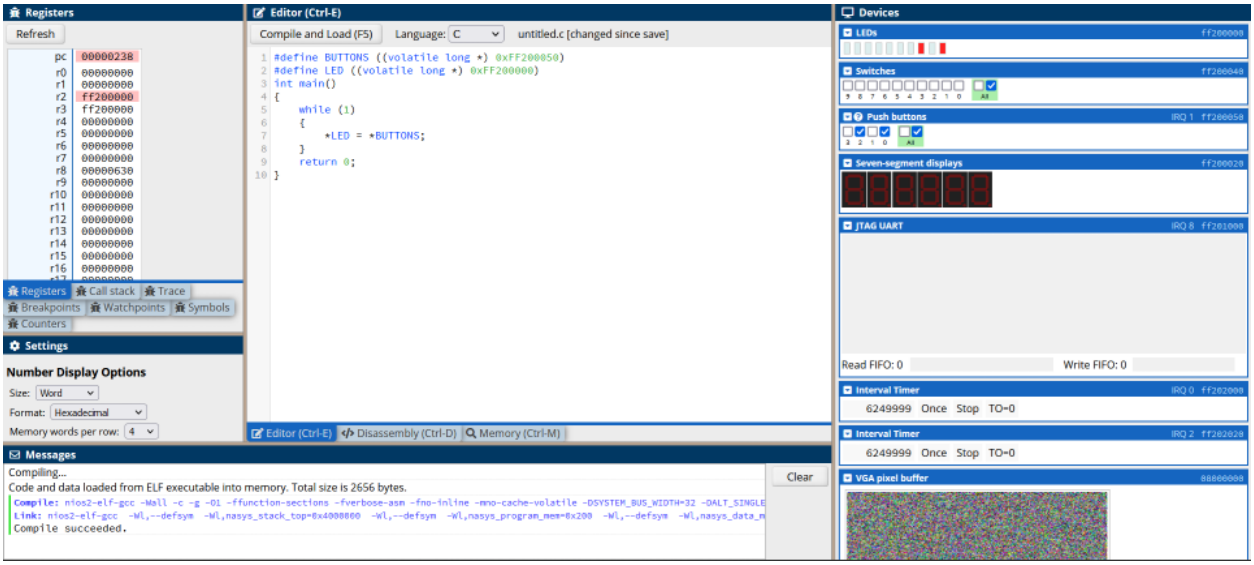

Figure 7: Push buttons 0 & 2 on and corresponding LEDs on while all other push buttons and LEDs are off

Part 2:

```
.equ PUSHBUTTON, _________
.equ LED, _________

start:
  movia r2,PUSHBUTTON
  ldwio r3,(r2)   # Read in buttons - active high
  movia r2,LED
  stwio r3,0(r2)    # Write to LEDs
  br start
```

Figure 8: Sample Assembly Code

Task 2 Part 2 was to take the given sample assembly code shown above in Figure 8 and convert it to have the same functionality as the C code from Task 2 Part 1 to see the efficiency of assembly compared to C. The code was changed to add the integer values for the PUSHBUTTON and LED registers since assembly code requires the integer value of the corresponding HEX address used to access these registers, so I used 4280287312 for PUSHBUTTON and 4280287232 for LED. Below are screenshots of the same combinations I used in part 1 to demonstrate functionality.

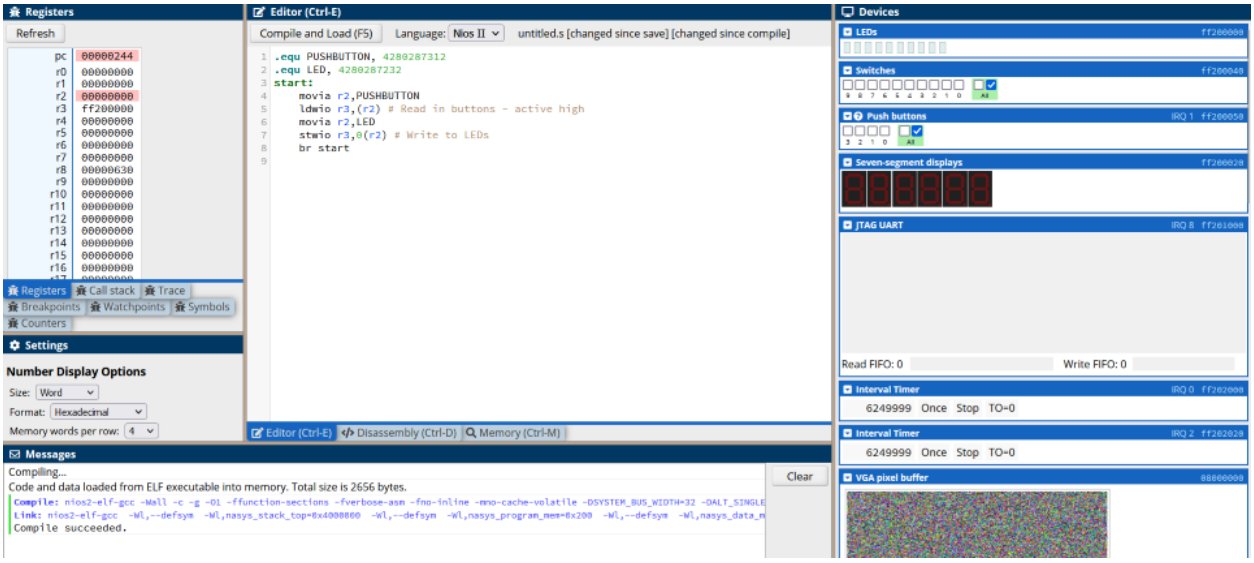

Figure 9: All push buttons off and all LEDs off

\

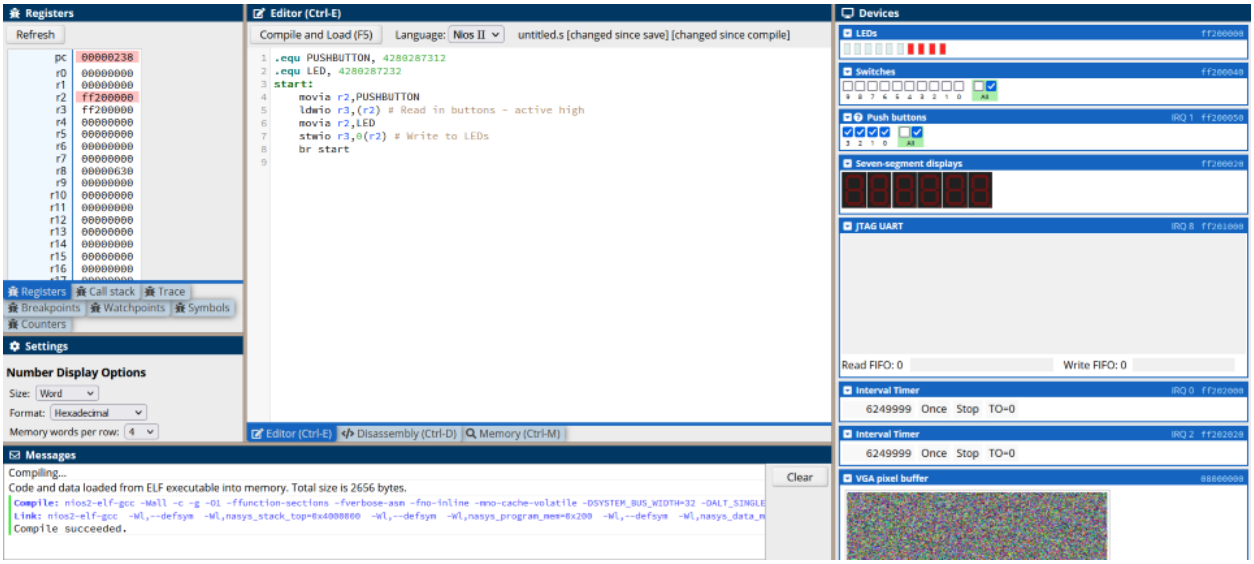

Figure 10: All push buttons on and all LEDs on

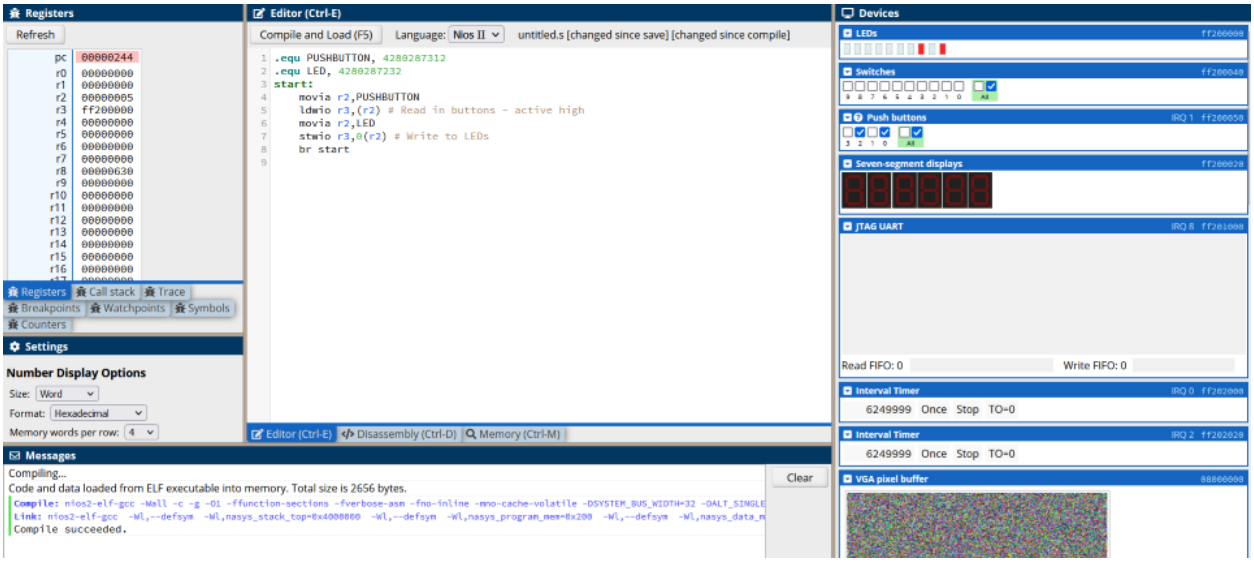

Figure 11: Push buttons 0 & 2 on and corresponding LEDs on while all other push buttons and LEDs are off

Results:

Task 1:

Part 1:

This part of this task was essentially just following instructions, so the procedure and results are pretty much the same and can be found above in the 'Procedure' section.

Part 2:

Same thing as above. The given instructions were followed in the spec and I answered the questions in the 'Procedure' section above so it would be redundant to put in this section as well.

Task 2:

Part 1:

1. <u>What is the register address for the LEDs on the NIOS processor? Taking a look at the example codes in Task 1, how would we define this register address in C?</u>
   The register address for the LEDs is 0xFF200000. In C, it would look like:
   #define LED ((volatile long *) 0xFF200000)

2. <u>What is the register address for the Pushbuttons on the NIOS processor? Taking a look at the example codes in Task 1, how would we define this register address in C?</u>
   The register address for the buttons is 0xFF2000050. In C, it would look like:
   #define buttons ((volatile long *) 0xFF200050)

3. How do we obtain (in C) the information on which button was pressed?
   You can obtain the value of the buttons pressed by using '*BUTTONS' which will retrieve the long value stored at that address.

4. When you compile your program, what is the size of the ELF executable (you can find this information in the Messages section of the CPUlator)?
   2656 bytes.

5. Take a screenshot of the LED and Pushbutton peripherals on CPUlator and include them in your report. Also save your .c source code (Ctrl + S, or go to File -> Save) as *Lab5_task2_part1.c*.
   Okay.

Part 2:

1. Fill in the register address for the PUSHBUTTON and LED. Compile this program on the CPUlator. **For this, make sure to set the Language to Nios II.** Verify that the functionality for this assembly code is the same as the C code you created in Task 2 part 1.
   Okay.

2. When you compile this program, what is the size of the ELF executable (you can find this information in the Messages section of the CPUlator)?
   28 bytes.

3. Is the size of this ELF executable similar or different from the size produced from the C code? Please explain thoroughly why these are similar or different.
   The size of the ELF executable is significantly smaller. The C code must be compiled into assembly and in theory, be the same if not very similar to directly using assembly code, however, the compiler has to compile assembly to directly match the C code. The C code contains methods and pointers and preprocessors. The preprocessor adds a lot of memory to the ELF to go through and link all other files (or not to) and then process the variable ahead of compiling which adds to its size.

4. Save your assembly code (Ctrl + S, or go to File -> Save) as a .s file called *Lab5_task2_part2.s*.
   Okay.

Appendix:

Lab5_task2_part1.c

```c
#define BUTTONS ((volatile long *) 0xFF200050)
#define LED ((volatile long *) 0xFF200000)
int main()
{
    while (1)
    {
        *LED = *BUTTONS;
    }
    return 0;
}
```

Lab5_task2_part2.s

```
.equ PUSHBUTTON, 4280287312
.equ LED, 4280287232
start:
    movia r2,PUSHBUTTON
    ldwio r3,(r2) # Read in buttons - active high
    movia r2,LED
    stwio r3,0(r2) # Write to LEDs
    br start

```