EECE 2160

Embedded Design: Enabling Robotics

Fall 2020

Lab Assignment 3

Section # 06

Maulik Patel, Tracy Qiu

Date Submitted: 10/17/2020

Abstract

The main purpose of this lab was to introduce the concept of memory-mapped I/O to access devices available on the DE1-SoC, such as the LEDs, pushbuttons, and switches. The end goal was to explore how, through object-oriented programming, all three of those devices would be enabled to interact with one another. In order to achieve this, functions were written one by one to accomplish small tasks such as reading the status of a switch, or turning one LED on and off, which would later be connected through a class format.

Introduction

The main premise of this lab was to practice modifying the memory mapped I/O devices on the DE1-SoC board, which includes pushbuttons, LEDs, and switches. This idea was then expanded into object-oriented programming with a class that would house all of the methods needed in order to control the LEDs as if they were a binary number.

For the pre-lab of this report, students were asked to understand and utilize RegisterRead() and RegisterWrite() functions to have the program read in and output the configuration of switches based on a binary number, and display a certain LED configuration based on a predetermined integer, that would be represented in binary fashion by the LEDs. Moving on the first assignment of the lab, asked the students to modify and utilize past written functions to have respective LEDs and switches to follow one another's behavior. Hence, if the fourth switch is turned on, the fourth LED should also be turned on. For the second assignment in the lab, the students were asked to write a Write1Led() function, in order to turn on or off a specific LED, no matter the switch configuration. For the third assignment, students were asked to write a function that would read in whether or not a specific switch was on or off. In the fourth assignment, buttons were implemented to modify the integer represented by the switches by one or more bitwise operators. Lastly, in the fifth assignment, object-oriented programming was used to abstract the functionality related to I/O operations on the board, which included all of the functions written in the previous assignments.

Lab Discussion

In this lab, the DE1-SoC board was utilized as the hardware equipment. Before powering on the board, the microSD card was placed into the slot as well as the mode select (MSEL) switches

were in a specific configuration. The switches were in the settings are shown below in Figure 1.

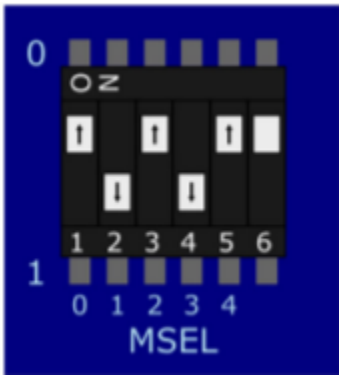

Figure 1: MSEL switches settings

The DE1-SoC board was connected to the host computer with the Type A to Mini-B USB cable in the serial UART-to-USB connection port. The board was connected to power using the Power DC Adapter. The red power button powered the DE1-SoC and from there, the user opened the MobaXterm program to run a serial connection to the DE1-SoC.

To start a new session in MobaXterm, the menu up top was used. First clicked Sessions → New Sessions → Serial. Once the serial port connected to the board was selected, the *Speed* was set to 115200 bits per second and in the advanced serial settings tab, the *Data bits* was set to 8, stop bits was 1, and *Parity and Flow control* to none. After clicking ok, the serial terminal opened, and the user was automatically logged in as root. From there commands could be typed in.

The source code for the three problems was coded in CLion the pasted into the linux terminal. To allow the linux terminal to paste code, the settings had to be changed. In the terminal, Settings then configuration options were selected. Then Terminal was selected where the check box for paste using right-click was turned on. Returning to the terminal screen, right clicking would allow a user to paste code from another IDE. To compile the C++ programs, a user can use g++ as the GCC compiler.

To transfer files between the host computer and the DE1-SoC board, a USB flush drive was used. First, the DE1-SoC board was powered and MobaXterm terminal opened on Serial communication. On the terminal, the fdisk -l command listed all drives that included the USB drive. From there, the USB drive was mounted into a folder where files could be transferred in and out of. A target folder was created in the /media directory. After the USB drive was mounted, the cd command was used to go into the USB directory and files could be copied from the USB directory to the DE1-SoC board or vise versa. The USB was then unmounted. After this initial setup, the only commands required to transfer files between USB drive and the DE1-SoC board are mount, cp, and unmount.

Prelab

Part 1:

For the prelab assignment, the code behind the functions RegisterRead() and RegisterWrite() were first explained. The RegisterWrite() function essentially takes in the value of pBase, an offset value to map from one LED to the next, and a value that would represent the address of an LED. Within the function the pBase and the offset value are added and equated to the third parameter, which the function uses to write a 4-byte value. The RegisterRead() on the other hand just returns the 4-byte value.

Part 2:

The next part of the lab had the group write a ReadAllSwitches() function that would simply read in the binary value the switches' configuration represented and output that value on the terminal. The code for this program is shown below.

```
int ReadAllSwitches() {
        return RegisterRead(SW_BASE) & 1023;
    }
```

Figure 2: ReadAllSwitches program for Pre-Lab.

As shown in the program in order to accomplish the goal, all that was needed was to "&" the value read by the function RegisterRead() to the value of 1023, representing a binary number, 1111111111, which is the largest number the LED's could represent. Using the & operation cleared the higher 22 bits of the 32-bit data register by masking them.

Part 3:

For this portion, a WriteAllLeds() function was written in order to write an integer value to the board's LEDs data register. As shown in the figure below, the value that was chosen to be displayed onto the LEDs was 35, which was represented correctly by the LED lights as the binary number 100011.

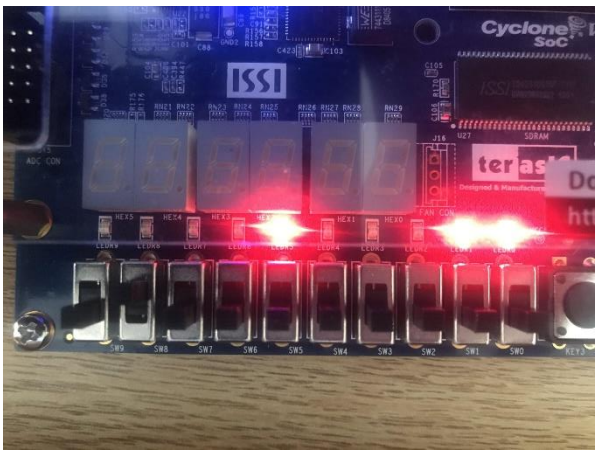

Figure 3: LED output for the integer value set at 35 by WriteAllLeds() function.

The following figure shows that code written for the WriteAllLeds() function.

```
void WriteAllLeds(char *pBase, int value){
    //call register write with parameter of led offset
   RegisterWrite(pBase, LEDR_BASE, value);
}
```

Figure 4: Program for WriteAllLeds() in Pre-Lab.

As shown in the function, the only thing that happened was that the RegisterWrite() function was called to edit the status of the LEDs to represent the value of 35 as previously discussed.

Results and Analysis

Assignment 1

For the first assignment of the lab, the LedNumber.cpp file from canvas was uploaded into the DE1-SoC in a directory named *Lab3*. From there the functions created in the pre-lab assignments, ReadAllSwitches() and WriteAllLeds(), were copy and pasted. The code for the WriteAllLed() function was then modified to simply represent the integer dictated by the switch configuration. The code for the modified WriteAllLed() is shown below.

```
void WriteAllLeds(char *pBase){
    int masked = ReadAllSwitches(pBase);
    RegisterWrite(pBase, LEDR_BASE, masked);
}
```

Figure 5: Modified WriteAllLeds() Function to represent switch configuration.

The main function was then modified to run and test these two functions. The following figure shows the configuration of the switches on the board and the resultant output of the ReadAllSwitches() and WriteAllLeds() functions on the terminal.

```
root@de1soclinux:~/lab3# ./LEDNumber
486
```

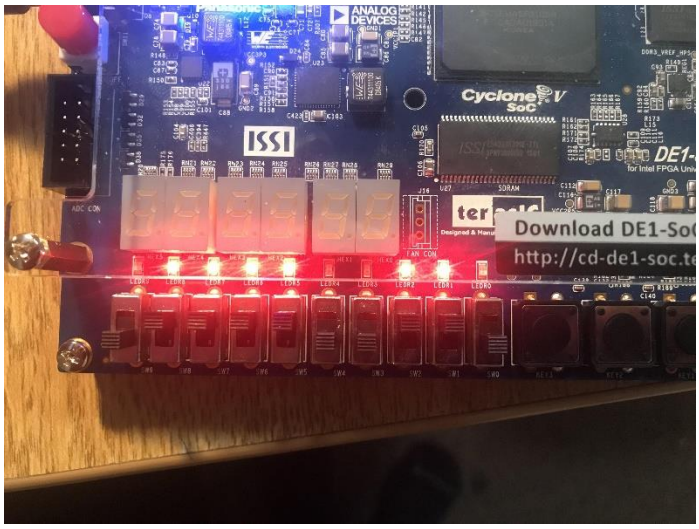

Figure 6: LED and terminal result from WriteAllLeds() and switch configuration responsible for terminal output.

Assignment 2

For the second assignment for this lab, the Write1Led() function was written in order to specifically dictate which LED to turn on, no matter the status of the switch while keeping the other LEDs unchanged. The result of this code on LED #5 is shown below.

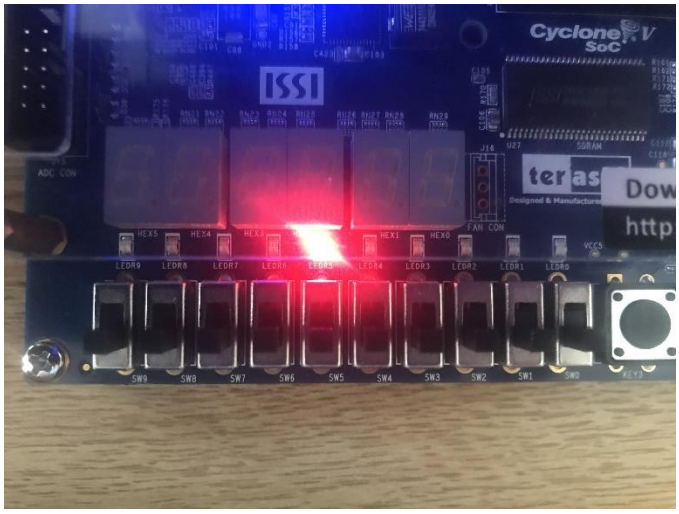

Figure 7: Write1Led() Turns on Only LED #5

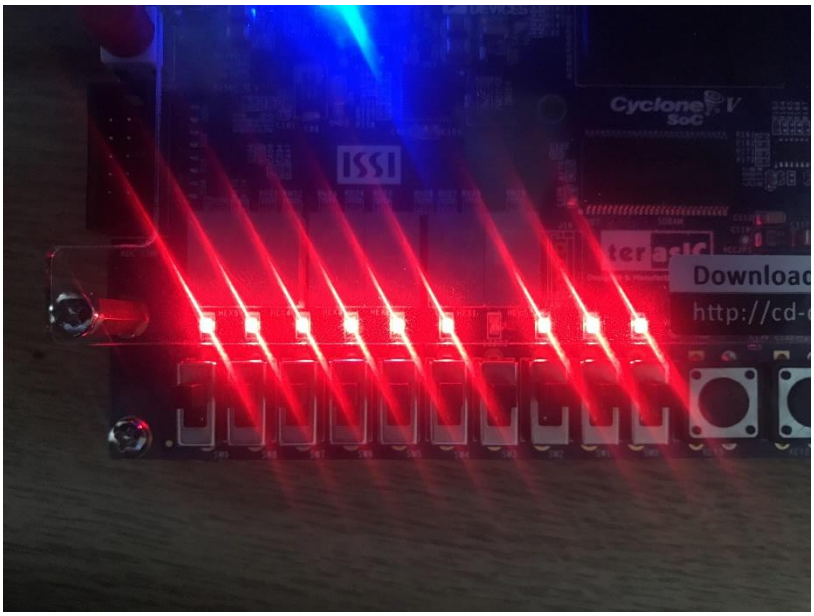

Figure 8: Write1Led() Only Turns Off LED #3

As shown in the image above, all of the switches were kept off, however, with the use of the Write1Led function, LED # 5 was still turned on. The following figure shows the code behind the function.

```
void Write1Led(char *pBase, int ledNum, int state){
    int masked = ReadAllSwitches(pBase);
    if (state == 1) {
        int num = masked | int(pow(2, ledNum));
        WriteAllLeds(pBase, num);
    }
    if (state == 0){
        int num = masked & ~int(pow(2,ledNum));
        WriteAllLeds(pBase, num);
    }
}
```

Figure 9: Write1Led( ) function code

In the function, the user passes two extra parameters, besides pBase, where ledNum determined the led that was to be affected, while the “state” parameter determined whether the led was going to be on or off (1 and 0 respectively). In the beginning of the function, the ReadAllSwitches() function is called to determine the value in binary the current switch configuration represents. After this, an if statement was used where the “state” parameter was checked to be either one or zero, resulting in different outcomes. If the state variable was one, then the decimal value of the switches was bit-masked with the decimal value of the led based on its placement in the binary number with the OR operator “ | ”. This allowed the original configuration of the LEDs to have the previously described led to be turned on, by changing its value in the binary number to 1. On the other option of the if statement where “state” is to be

zero, the configuration of the switches is masked with the inverse of the value of the led in binary, to allow for the rest of the LEDs to remain constant, while the chosen LED gets turned off.

Assignment 3

For the third assignment of this lab, the Read1Switch() function was integrated into the function in order to allow the board to read in the fact that whether or not a particular switch, defined in the call of the function, is on or off. The function therefore would return a 1 on the terminal if the switch was determined to be turned on, and a 0 if the switch was turned off. The code for the function to read in the status of the switch is shown below.

```
int Read1Switch(char *pBase, int switchNum){
    int masked = ReadAllSwitches(pBase);
    int num = int(pow(2, switchNum));
    int switch1 = masked & num;
    int on;
    if (switch1)
        on = 1;
    else
        on = 0;
    return on;
}
```

Figure 10: Read1Switch() function code.

From the code it is shown that the value determining the status of the switch, “switch1”, is determined through the use of bit masking with the operator “&” between the value determined by all of the switches, and the value of the binary value of the switch whose status is of interest.

Assignment 4

For the fourth assignment, the previous LedNumber.cpp file was copied into a new file called PushButton.cpp within the same directory. The goal of this new program was to implement the input of push buttons, such that they would help change the numerical value of predetermined switches, and show this change on the LEDs on the board. The four buttons in this lab were referred to as KEY0. KEY1, KEY2, and KEY3 respectively. The function, PushButtonGet(), was created to house all of the different commands required by the buttons. KEY0 was first programmed in the function to allow the user to increment the value predetermined by the switches by one. The value was then immediately showed in binary by the LEDs, with an on LED representing the number 1, while an off one represents a zero. KEY1, was implemented so that when a user presses it, the current value represented by the LEDs was decremented by one. This again was programmed to immediately reflect in a binary fashion on the LEDs. Key2 was then programmed so that it would shift the current value of the counter to the left by one every time it was triggered, while inserting zeros to the right. This again was also immediately reflected on the LEDs. KEY3 was programmed to shift the current value of the

counter to the right by one every time it was triggered, while inserting zeros to the left. This again was reflected on the LEDs in a binary fashion. If multiple buttons were to be pressed, the program was set to reset the LEDs to the value predetermined by the switches. Additionally, the function was set so that the terminal would output a “-1” if no buttons were pressed, a “0” if KEY0 was pressed, a “1” if KEY1 was pressed, a “2” if KEY2 was pressed, and a “3” if KEY3 was pressed. The code for the function is shown in the following figure.

```
void PushButtonGet(){
        unsigned int counter = ReadAllSwitches();
        WriteAllLeds(counter);

        unsigned int key;

        int x = 0;
        while(x == 0) {
            key = RegisterRead(KEY_BASE);
            sleep(1);
            switch (key) {
                case 0:
                    cout << "-1" << endl;
                    break;
                case 1:
                    counter++;
                    WriteAllLeds(counter);
                    cout << "0" << endl;
                    break;
                case 2:
                    counter--;
                    WriteAllLeds(counter);
                    cout << "1" << endl;
                    break;
                case 4:
                    counter = counter >> 1;
                    WriteAllLeds(counter);
                    cout << "2" << endl;
                    break;
                case 8:
                    counter = counter << 1;
                    WriteAllLeds(counter);
                    cout << "3" << endl;
                    break;
                default:
                    WriteAllLeds(ReadAllSwitches());
                    cout << "-1" << endl;
                    x = 1;
                    break;
            }
        }
    }
```

Figure 11: PushButtonGet() function.

As shown in the function, the first steps were to read in the predetermined value of the switches, and initialize its value to an unsigned integer named counter. Then the value was

displayed onto all of the LEDs with the WriteAllLeds() function. In order to continuously read in the inputs onto the buttons a while loop was used, where the temporary variable, x, was used to exit the loop in order to end the program. Within the while loop the RegisterRead() function was used to check whether or not a button was pressed, and the sleep() function was called after it to compensate for any false open/ close transitions often generated by a pushbutton, so that the program does not interpret one press as multiple presses. A switch statement was then used to interpret separate button presses as separate commands described before, by reading the buttons as numerical values, similar to how the switch buttons were translated to a numerical value. Because of this KEY0, KEY1, KEY2, and KEY3 was represented by the value 1, 2, 4, and 8 respectively in the switch statements. The default is any value besides those listed and would mean that multiple buttons were pushed. This would set the exit variable to 1 and end the while loop. The commands discussed for each button as discussed before are also clearly shown. The demonstration of this code was uploaded with the other files.

Assignment 5

Assignment five used object-oriented programming to create the same functionality as the PushButton program. This was done by creating two classes in a new program PushButtonClass. The two classes were DE1SoCfpga and LEDControl that implemented the same functions to read the switches, LEDs, and buttons and write different LEDs outputs. The DE1SoCfpga class contained two private attributes pBase and fd which were used to read in the state of the board. By creating a class, this took away the need to pass fd and pBase as parameters in each function. The default constructor contained the initialize function which opened the access to the physical memory of the input output device. The initialize function in the PushButton program had a return value of the base pointer, but the default constructor now assigned the virtual base in the function to pBase. This initialized pBase which would be accessed throughout the class. The destructor implemented the finalize function and was called at the end of main to free the object's memory. RegisterRead and WriteRegister were the two functions in the DE1SoCfpga class and the parameters of these functions no longer needed base pointer pBase as an argument because it was an attribute of the class. The class was tested in the main by creating an object of class DE1SoCfpga then calling each method. The register read and write methods made the LEDs on the board match the states of the switches.

The second class LEDControl implemented other functions that controlled the LEDs, switches, and buttons. The LEDControl class has public inheritance of the DE1SoCfpga class and inherits the same constructor and destructor. The Write1Led, WriteAllLeds, Read1Switch, ReadAllSwitches, and PushButtonGet functions were all the same as the ones implemented in PushButton program. The main difference made was taking out the base pointer from the parameters of each function because they were no longer necessary. In the main, to test the LEDControl class, an object of this class was created and passed the virtual device file fd as the parameter. Each method was called to test the functionality. Each method produced results that were the same as those of the PushButton program. This is demonstrated in the video file that is submitted with this assignment.

This assignment demonstrated how functions can be implemented differently and the efficiency of object-oriented programming. By creating two classes that used inheritance, this reduced the need for passing the same base pointer parameter each time and reduced the need to redefine certain functions.

Conclusion

The results of this lab although not quantitative, revealed how object-oriented programming could be used to map I/O devices on the DE1-SoC, such as LEDs, pushbuttons, and switches. Programs were written to access these devices, such as LEDNumber.cpp, PushButton.cpp, and PushButtonClass.cpp. While there were no quantitative experimental results, the fact that all of the functions worked as they were expected to shows that the overall lab was a success. For example, the Write1LED() function was successful in independently turning on a single LED, and the DE1SoCfpga class was also successful in testing out all of the LEDs controls done in the previous assignments.

One way that this lab could be bettered in the future would be to have the students implement some function that would allow the students to make inputs through the pushbuttons without having to wait for about a second for the board to read in the input. This would allow for the buttons to have a more instant and tactile feel to them, rather than having some of the pushbutton’s inputs be ignored, since they fell during the sleep() function’s delay. Otherwise, the lab ended up being a challenging and fruitful learning experience with object-oriented programming of I/O devices on the DE1-SoC.