Lab Assignment 2
Memory-Mapped I/O andObject-Oriented Programming

Levi Kaplan
kaplan.l@northeastern.edu

Submit Date: 9/29/20
Due Date: 9/28/20

Abstract

In this lab, we worked with interfacing with the physical components of the board—the LEDs, switches, and push buttons. We worked with bitwise operators, reading from and writing to the board, and class abstraction. We learned more about binary and decimal conversion and binary mathematics. We also learned how to debounce buttons and read the right inputs.

Introduction

Software and Hardware Used

Hardware

- DE1-SoC board
- Ethernet cable

Software

- Atom text editor
- Online C++ compiler (https://www.onlinegdb.com/online_c++_compiler)
- ssh and scp

Lab Steps

Prelab

1. We added what we believed would be proper code for the Write1Led() and ReadAllSwitches() methods
2. We ensured that the starter code worked as expected

Lab 2.1

1. We implemented the Write1Led() and ReadAllSwitches() methods
2. We came across an error with writing the Leds
3. We fixed this error and tested that the LEDs can turn on properly
4. We next figured out how to turn off the Leds
5. We tested the program and ensured it worked

Lab 2.2

1. We duplicated the content of Lab 2.1
2. We added the Read1Switch() method
3. We added the method call to main
4. We tested the method

5. We added the capability to read a switch when multiple switches are turned on
6. We tested the program and ensured its functionality

Lab 2.3
1. We duplicated the content of Lab 2.2
2. We added the PushButtonGet() method
3. We added the HandlePushButtons() helper, which delegates to the appropriate method depending on the value of the push buttons
4. We implemented the methods to increment, decrement, shift left, and shift right
5. We tested this functionality and found errors
6. We experimented with various ways of continually getting user input from buttons
7. We added DetectPushbuttonChanges(), which detects if there has been a change
8. We tested the method and ensured proper functionality

Lab 2.4
1. We created the DE1SoCfpga() class
2. We copied over the methods needed in the class from the lab
3. We copied over the content of the Initialize() and Finalize() methods from main into the constructor and destructor for the class
4. We copied over our methods and helper methods from Lab 2.3
5. We fixed a bug where the constructor would break
6. We adjusted main to create an instance of the DE1SoCfpga() class and call the proper method from there
7. We ran the method and checked its functionality

Lab 2.5
1. We copied over the contents of Lab 2.4
2. We created a new class called LedControl
3. We made LedControl inherit the DE1SoCfpga class
4. We copied over all methods having to do with Led, Switch, and Button control

5. We edited main to call the LedControl's DetectPushButtonChange() method
6. We tested the method and discovered a visibility issue
7. We tested the method and ensured proper functionality

Lab Discussion

Prelab

1. We added what we believed would be proper code for the Write1Led() and ReadAllSwitches() methods
    a. because we hadn't added this functionality to main, we couldn't test it yet
    b. We would test this functionality and realize we had a lot to tweak in Lab 2.1
2. We ensured that the starter code worked as expected

Lab 2.1

1. We implemented the Write1Led() and ReadAllSwitches() methods
2. We came across an error with writing the Leds
    a. running it with 0 worked, but 1 turned on the 8th led and none of the other ones worked
    b. The issue was that we were passing in the actual values instead of the binary values
    c. Once we figured this out, we still had to figure out how to convert the integer to binary
    d. We tried using pow(), but this wasn't supported without the Math library
    e. We used a for loop to calculate the base 2 value of the LED we want to affect by adding the right powers of 2
    f. We used bitwise operators to turn on and off the correct LED based on the value of the LED
3. We fixed this error and tested that the LEDs can turn on properly
4. We next figured out how to turn off the Leds

   a. We weren't able to just run the program again, this time with a state of 0
   b. We tried to figure out when the LED was already on, and then pass a value of 0 if this was the case
   c. We calculated the current state of the LED and checked to see if it was greater than 0 and the passed in state was 0, in which case the LED should be turned off
   d. The calculations for the current state of the LED weren't working properly because we were trying to read the current state of the board at the specified base 2 value instead of reading the state and seeing if it matches with the base 2 value that was passed in
   e. We realized we could turn off the LED through a combination of bitwise operators, and that this solution was more elegant
5. We tested the program and ensured it worked
   a. We passed in various numbers and ensured the correct LED turned on
   b. We then turned that LED off by passing in that numbered LED and a state of 0
   c. We confirmed the functionality by looking at the state of our board after every iteration

Lab 2.2

1. We duplicated the content of Lab 2.1
2. We added the Read1Switch() method
   a. We used a similar technique to Lab 2.1 to read the current value and compare it to the value of the passed-in switch
3. We added the method call to main
   a. We added debugging information by printing an error if the user enters anything less than 0 or greater than 9
   b. We only ran the method if it met the desired parameters
4. We tested the method
   a. In testing, we discovered that our version of the code only worked if only one switch was on
   b. Current state was always different from the value of the switch we were checking
   c. We were checking whether the current state was equal

5. We added the capability to read a switch when multiple switches are turned on
    a. We realized that we need to calculate the state of all the switches
    b. We used a for loop to iterate over all the values, using int i for iteration
    c. We determined whether 2 to the power of i was greater than or equal to the current base 2 value we were looking at
        i. If this was the case, we knew that the switch had to be turned on, so we added the value of 1 to the array at that position
        ii. If it wasn't the case, we knew the switch was off
    d. Once we determined the bit value of all the switches, we were able to check the value of the passed-in switch and return that value
6. We tested the program and ensured its functionality

Lab 2.3

1. We duplicated the content of Lab 2.2
2. We added the PushButtonGet() method
    a. it just returned a call to ReadAllButtons, which reads the button values
3. We added the HandlePushButtons() helper, which delegates to the appropriate method depending on the value of the push buttons
4. We implemented the methods to increment, decrement, shift left, and shift right
    a. we had a method for each, which performs the appropriate operation on the current value and passes that value to WriteAllLeds()
5. We tested this functionality and found errors
    a. The debouncing of the buttons caused the button's method to be repeatedly activated
    b. We tried adding an exit capability that would cause the methods for detecting button changes to not be called properly
6. We experimented with various ways of continually getting user input from buttons
    a. This is detailed further in Analysis
    b. We tried using recursion, while(true) in helper methods

   c. Eventually we realized that we needed to have a while(true) loop in main that called a method to detect button changes
7. We added DetectPushbuttonChanges(), which detects if there has been a change
   a. This allowed us to solve the problem of continually performing an operation
   b. it used a global variable representing the current value, and checked to see if the current button value was different
8. We tested the method and ensured proper functionality

Lab 2.4

1. We created the DE1SoCfpga class
2. We copied over the methods needed in the class from the lab
3. We copied over the content of the Initialize() and Finalize() methods from main into the constructor and destructor for the class
4. We copied over our methods and helper methods from Lab 2.3
5. We fixed a bug where the constructor would break
   a. we were dereferencing a variable that wasn't an address
   b. Intitialize() was being passed in an address that needed to be dereferenced, but in the class the fd variable was class-specific so we didn't need to dereference it
   c. Once we removed the dereference operators from fd, the method worked
6. We adjusted main to create an instance of the DE1SoCfpga() class and call the proper method from there
7. We ran the method and checked its functionality

Lab 2.5

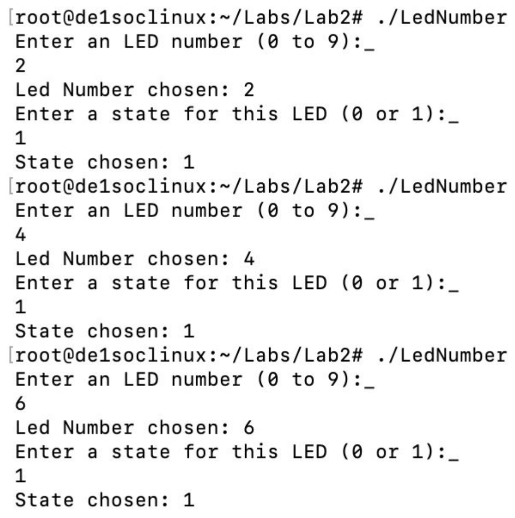
1. We copied over the contents of Lab 2.4
2. We created a new class called LedControl
3. We made LedControl inherit the DE1SoCfpga class
   a. We added : public De1SoCfpga to the end of the class declaration
4. We copied over all methods having to do with Led, Switch, and Button control
5. We edited main to call the LedControl's DetectPushButtonChange() method

6. We tested the method and discovered a visibility issue
   a. We discovered that the methods from within LedControl were private, so main couldn't access them
   b. We changed the visibility of the class contents to be public
7. We tested the method and ensured proper functionality

Results

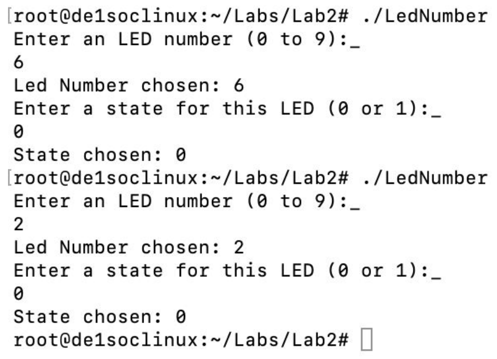
Lab 2.1

Test turning on LEDs 2, 4, and 6
Results:



```
[root@de1soclinux:~/Labs/Lab2# ./LedNumber
 Enter an LED number (0 to 9):_
 2
 Led Number chosen: 2
 Enter a state for this LED (0 or 1):_
 1
 State chosen: 1
[root@de1soclinux:~/Labs/Lab2# ./LedNumber
 Enter an LED number (0 to 9):_
 4
 Led Number chosen: 4
 Enter a state for this LED (0 or 1):_
 1
 State chosen: 1
[root@de1soclinux:~/Labs/Lab2# ./LedNumber
 Enter an LED number (0 to 9):_
 6
 Led Number chosen: 6
 Enter a state for this LED (0 or 1):_
 1
 State chosen: 1
```

Test turning off LEDs 4 and 6
Results:

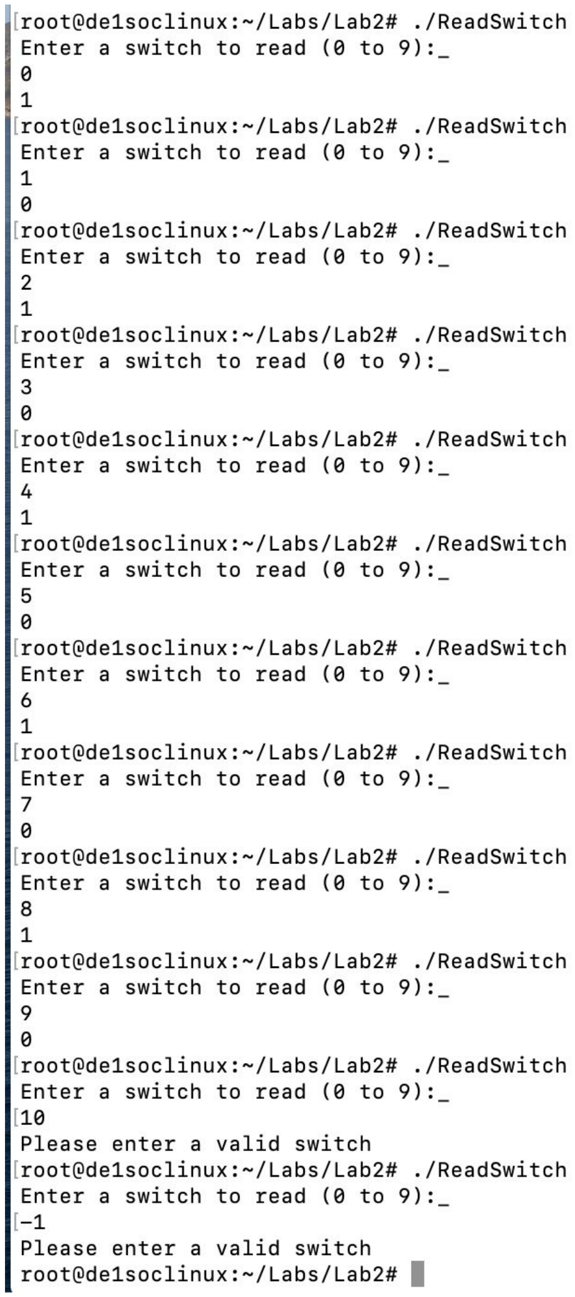


```
[root@de1soclinux:~/Labs/Lab2# ./LedNumber
 Enter an LED number (0 to 9):_
 6
 Led Number chosen: 6
 Enter a state for this LED (0 or 1):_
 0
 State chosen: 0
[root@de1soclinux:~/Labs/Lab2# ./LedNumber
 Enter an LED number (0 to 9):_
 2
 Led Number chosen: 2
 Enter a state for this LED (0 or 1):_
 0
 State chosen: 0
 root@de1soclinux:~/Labs/Lab2# 
```

Lab 2.2

Test 1: Switches 0, 2, 4, 6, and 8 on, rest are off

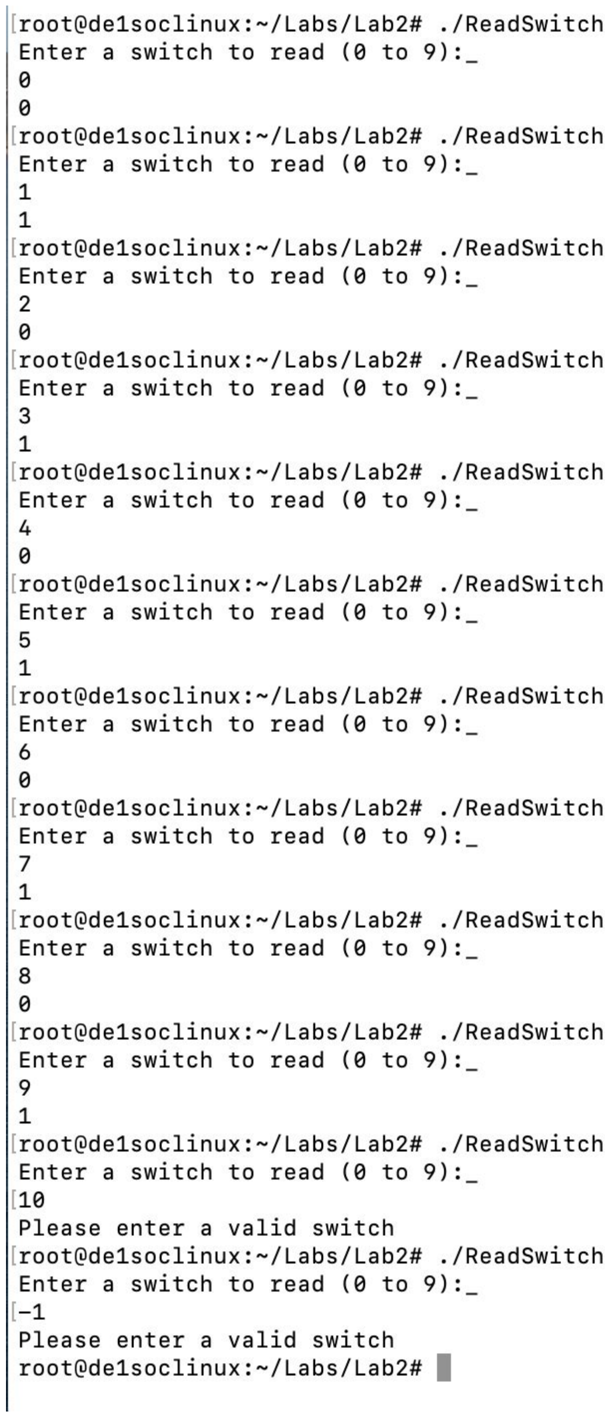
Results:



```
root@de1soclinux:~/Labs/Lab2# ./ReadSwitch
Enter a switch to read (0 to 9):_
0
1
root@de1soclinux:~/Labs/Lab2# ./ReadSwitch
Enter a switch to read (0 to 9):_
1
0
root@de1soclinux:~/Labs/Lab2# ./ReadSwitch
Enter a switch to read (0 to 9):_
2
1
root@de1soclinux:~/Labs/Lab2# ./ReadSwitch
Enter a switch to read (0 to 9):_
3
0
root@de1soclinux:~/Labs/Lab2# ./ReadSwitch
Enter a switch to read (0 to 9):_
4
1
root@de1soclinux:~/Labs/Lab2# ./ReadSwitch
Enter a switch to read (0 to 9):_
5
0
root@de1soclinux:~/Labs/Lab2# ./ReadSwitch
Enter a switch to read (0 to 9):_
6
1
root@de1soclinux:~/Labs/Lab2# ./ReadSwitch
Enter a switch to read (0 to 9):_
7
0
root@de1soclinux:~/Labs/Lab2# ./ReadSwitch
Enter a switch to read (0 to 9):_
8
1
root@de1soclinux:~/Labs/Lab2# ./ReadSwitch
Enter a switch to read (0 to 9):_
9
0
root@de1soclinux:~/Labs/Lab2# ./ReadSwitch
Enter a switch to read (0 to 9):_
10
Please enter a valid switch
root@de1soclinux:~/Labs/Lab2# ./ReadSwitch
Enter a switch to read (0 to 9):_
-1
Please enter a valid switch
root@de1soclinux:~/Labs/Lab2#
```

Test 2: Switches 1, 3, 5, 7, and 9 are on, rest are off

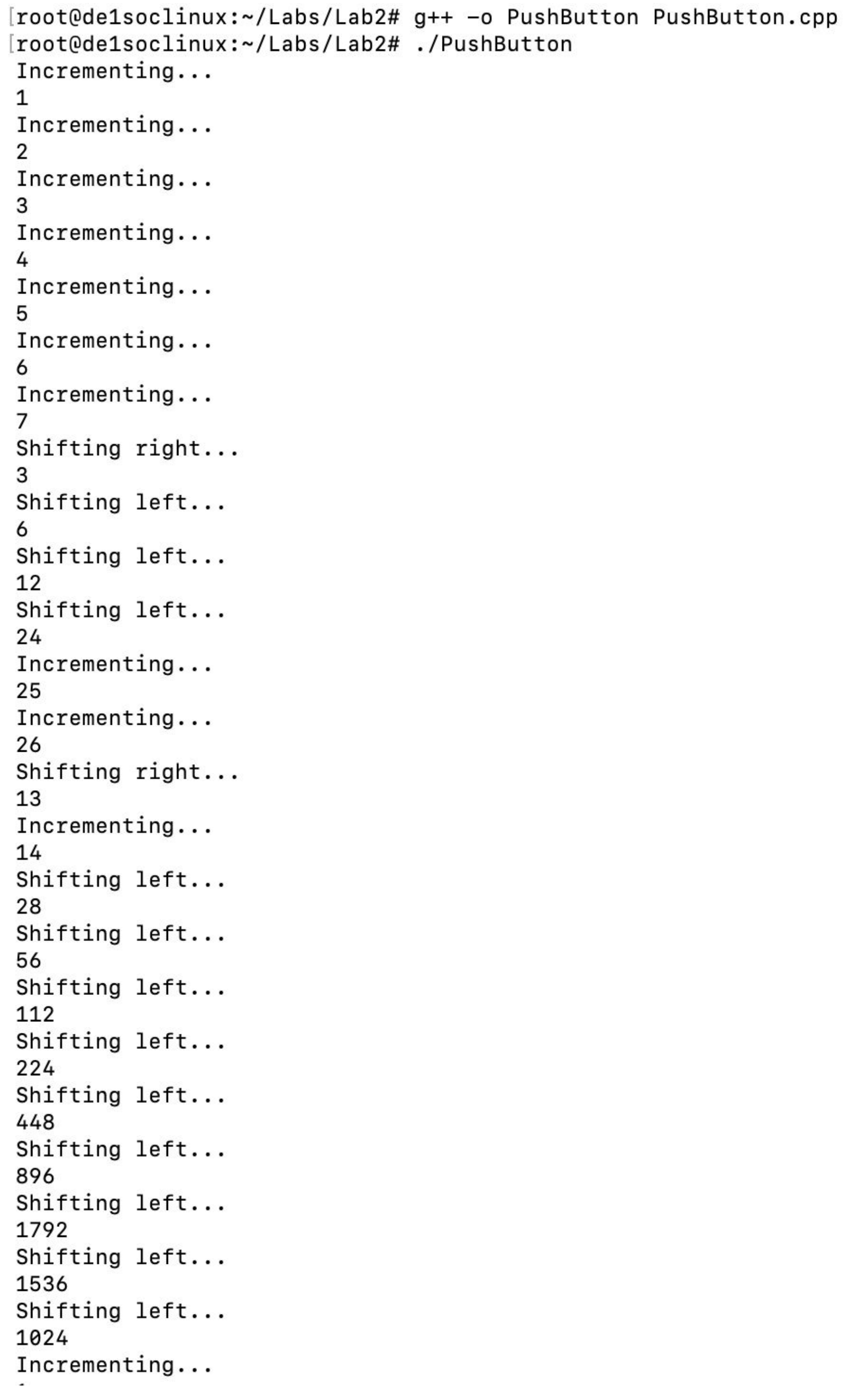
Results:



```
root@de1soclinux:~/Labs/Lab2# ./ReadSwitch
Enter a switch to read (0 to 9):_
0
0
root@de1soclinux:~/Labs/Lab2# ./ReadSwitch
Enter a switch to read (0 to 9):_
1
1
root@de1soclinux:~/Labs/Lab2# ./ReadSwitch
Enter a switch to read (0 to 9):_
2
0
root@de1soclinux:~/Labs/Lab2# ./ReadSwitch
Enter a switch to read (0 to 9):_
3
1
root@de1soclinux:~/Labs/Lab2# ./ReadSwitch
Enter a switch to read (0 to 9):_
4
0
root@de1soclinux:~/Labs/Lab2# ./ReadSwitch
Enter a switch to read (0 to 9):_
5
1
root@de1soclinux:~/Labs/Lab2# ./ReadSwitch
Enter a switch to read (0 to 9):_
6
0
root@de1soclinux:~/Labs/Lab2# ./ReadSwitch
Enter a switch to read (0 to 9):_
7
1
root@de1soclinux:~/Labs/Lab2# ./ReadSwitch
Enter a switch to read (0 to 9):_
8
0
root@de1soclinux:~/Labs/Lab2# ./ReadSwitch
Enter a switch to read (0 to 9):_
9
1
root@de1soclinux:~/Labs/Lab2# ./ReadSwitch
Enter a switch to read (0 to 9):_
10
Please enter a valid switch
root@de1soclinux:~/Labs/Lab2# ./ReadSwitch
Enter a switch to read (0 to 9):_
-1
Please enter a valid switch
root@de1soclinux:~/Labs/Lab2#
```

Lab 2.3

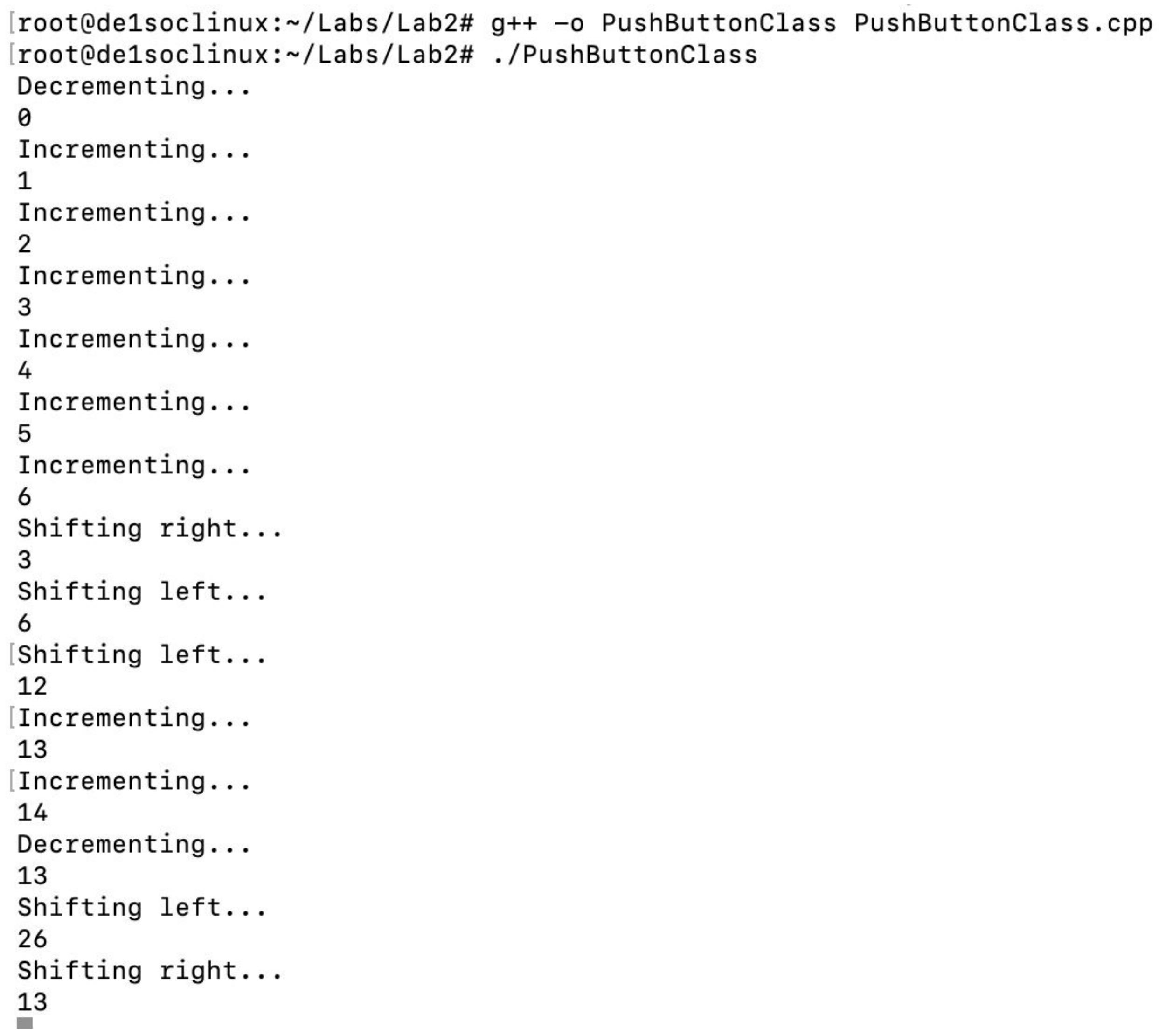
Example pushing Increment until the value of 7, shifting right, shifting left 3 times, incrementing twice, shifting right, incrementing, and shifting left until the LEDs display 0.



```
[root@de1soclinux:~/Labs/Lab2# g++ -o PushButton PushButton.cpp
[root@de1soclinux:~/Labs/Lab2# ./PushButton
Incrementing...
1
Incrementing...
2
Incrementing...
3
Incrementing...
4
Incrementing...
5
Incrementing...
6
Incrementing...
7
Shifting right...
3
Shifting left...
6
Shifting left...
12
Shifting left...
24
Incrementing...
25
Incrementing...
26
Shifting right...
13
Incrementing...
14
Shifting left...
28
Shifting left...
56
Shifting left...
112
Shifting left...
224
Shifting left...
448
Shifting left...
896
Shifting left...
1792
Shifting left...
1536
Shifting left...
1024
Incrementing...
```

Lab 2.4

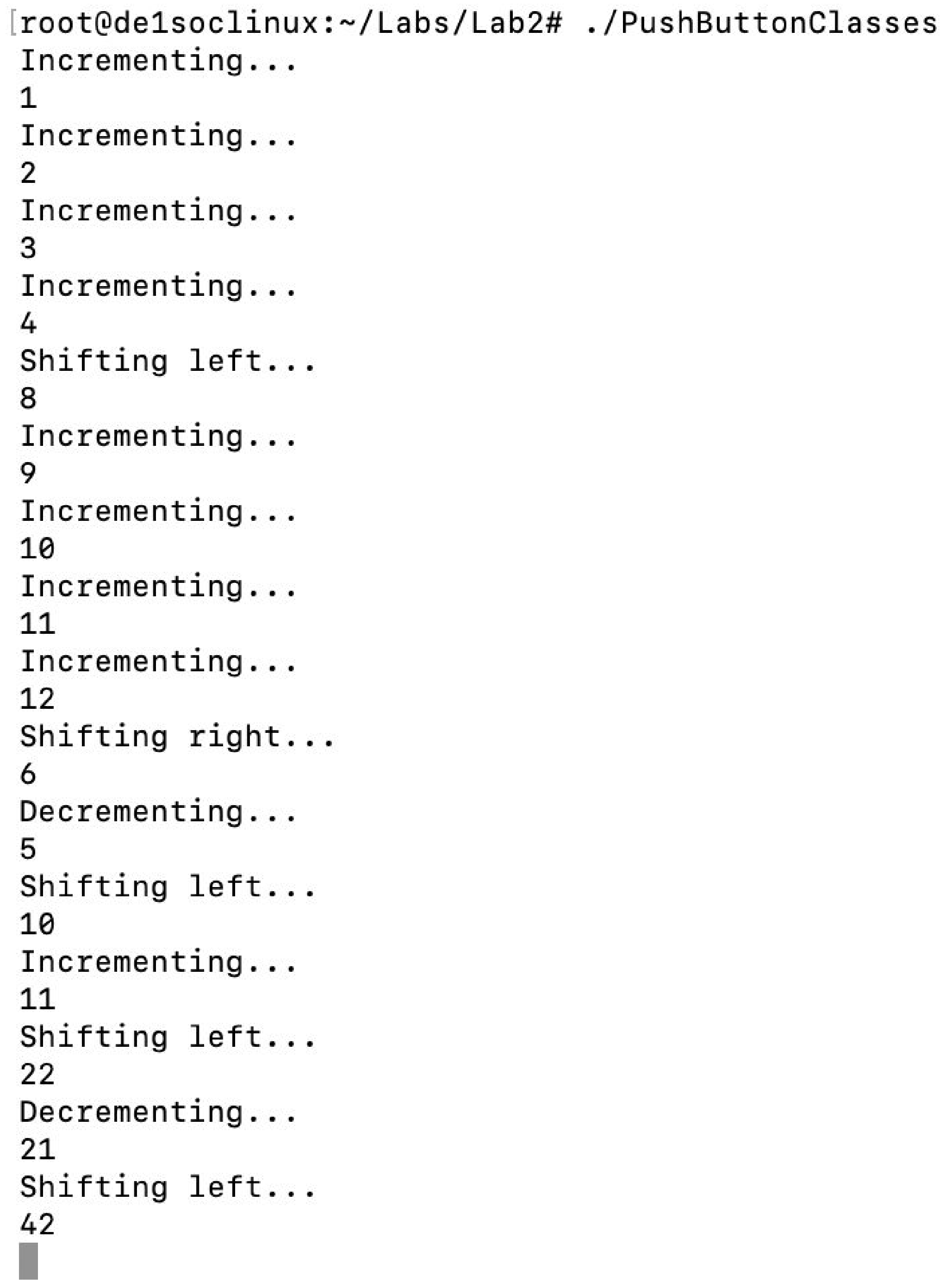
Showing that the push button functionality still works after the class abstraction:



```
[root@de1soclinux:~/Labs/Lab2# g++ -o PushButtonClass PushButtonClass.cpp
[root@de1soclinux:~/Labs/Lab2# ./PushButtonClass
Decrementing...
0
Incrementing...
1
Incrementing...
2
Incrementing...
3
Incrementing...
4
Incrementing...
5
Incrementing...
6
Shifting right...
3
Shifting left...
6
[Shifting left...
12
[Incrementing...
13
[Incrementing...
14
Decrementing...
13
Shifting left...
26
Shifting right...
13
```

Lab 2.5

Showing that the push button functionality still works after further class abstraction:



```
[root@de1soclinux:~/Labs/Lab2# ./PushButtonClasses
Incrementing...
1
Incrementing...
2
Incrementing...
3
Incrementing...
4
Shifting left...
8
Incrementing...
9
Incrementing...
10
Incrementing...
11
Incrementing...
12
Shifting right...
6
Decrementing...
5
Shifting left...
10
Incrementing...
11
Shifting left...
22
Decrementing...
21
Shifting left...
42
```

Analysis

Lab 2.1

We had a lot of difficulty with Lab 2.1 because of our lack of familiarity with writing to memory and working with values in memory. Our first issue came from our Prelab code not working as we expected at all. We thought that we could just add the passed-in ledNum to the LEDR_BASE, read the value, check whether the value was greater than 1 and the state desired was 0, and turn of the LED from there, otherwise we could just turn it on. We were passing in the LEDR_BASE + num into RegisterWrite instead of calling WriteAllLeds(), and we weren't converting the number to a binary representation. This all had to be fixed, and it took a lot of trial and error to figure out why it wasn't working. The biggest help was that the lab came with code that worked when displaying binary values to the LED, so we were able to leverage this to figure out the solution. We had to first convert the number to a base-2 representation, so 3 had to be converted to 8 and 4 needed to be 16 and so on. This is because the data needs to be written to its binary representation on the board. We needed to pass this value into WriteAllLeds() to get them to turn on. This allowed for the LEDs to be turned on, but not turned back off. To get that functionality working, we figured out we had to check to see if the state had been changed at the ledNum's base 2 position, and then set the value to change the LEDs to to 0 if the state passed in was 0. This ended up being much trickier than expected, because we thought that to check the state of the board we needed to call RegisterRead() with LEDR_BASE plus the value of the LED's base 2 representation. The logic of this was that we thought we would get a value of 1 if the state at that particular address was on, and 0 if it wasn't. The reality of the RegisterRead() method, however, is that it returns the value of the board, so if light 4 is turned on, it'll return 16. This meant that we only needed to check if the value (the base 2 representation of the ledNum) was equal to the board state and the state passed in to the method was 0. After all this work, we realized that we really should be using bitwise operators to turn on and off the board. We were just writing the desired values directly to the board, but this was much more clunky and inelegant. Additionally, it didn't support turning on multiple LEDs and only turning off specific LEDs. We were able to experiment with bitwise operators, combining multiple operators to get the 'off' functionality by first using OR to combine the value to the curState, and then use the XOR operator to turn off the value that was different.

Lab 2.2

In Lab 2.1, a lot of the work from Lab 2.1 carried over, and we got a testable version working pretty quickly. However, we implemented it so that it only worked properly if only one switch was turned on. This is because when trying to access the state of a switch when multiple switches were on, it would always output 0 because our method was checking if the value of that switch was equal to the current state of the board, which it wouldn't be because multiple switches were on. We realized that we need to calculate the state of all the switches, so we used a for loop to iterate over all the values, using int i for iteration. We determined whether 2 to the power of i was greater than or equal to the current base 2 value we were looking at, and if this was the case, we knew that the switch had to be turned on. We added the value of 1 to the array at that position. Therefore, if the value of the array at the index of the switch we're looking at was 1, the switch was on, and if it was 0, the switch was off, and we could return that value. This was tricky to implement and we tried a number of ways to get the current state of the board and compare it to the desired switch value. The solution is perhaps not very elegant, because we have to calculate multiple powers of 2, iterate over them, and subtract the appropriate values from a value representing the current value. We think there may be a way to use bitwise operators for this, but we weren't able to figure it out.

Lab 2.3

In this portion of the lab, we found that a lot of the work in writing to LEDs carried over from the previous portions of the lab, but we had a lot of difficulty with continuously reading input. We tried using a while(true) loop in a helper function and had the ability to exit properly by typing in 'quit', but we found that this caused the checking for 'quit' to happen first and look for user input from the keyboard, and the checking for button inputs wouldn't happen. We then tried getting rid of the quitting functionality and used a recursive helper function to check whether the input was different from the previous input, and call the correct function if that was the case. We got a "Segmentation Fault" error because of the recursion. We found that adding the while(true) loop in main(), calling the helper checking if the inputs had changed within that loop, worked perfectly.

Lab 2.4

This portion of the lab involved abstracting functionality into classes. We were able to pretty easily follow the class template and copy over functionality, but we came across an error that read “invalid type argument of unary '*' (have 'int')”. This was because we were trying to dereference the fd value in our constructor, because we copied it over from the Initialize() class, which took in an address and needed to dereference that address. Once we deleted these pointer values and substituted them for just fd, it was able to run properly.

Lab 2.5

Lab 2.5 went smoothly for the most part. We were able to easily transition over to another layer of abstraction, but we now had to call the DetectPushButtonChange() method from LedControl, which was by default set to private, so we had to quickly change that to be public before the rest of it worked.

Conclusion

This lab gave us broad and valuable experience with interfacing with the board. We gained an understanding on how to read data on the board, process user input to modify that data, and write the updated state back to the board. We debounced button inputs to only process user input once, processed the input differently based on the button pressed, and performed four different numerical operations on the board, writing the new values of these numerical operations onto the board’s LEDs. We also learned how to read the switch data. This lab also focused on class abstraction and object oriented design. We abstracted our code into multiple classes, with the LedControl class inheriting the De1SoCfpga class and interfacing with its methods. We created constructors and destructors and interfaced with them properly, instantiating classes in this way.

Appendix

Lab 2.1

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <iostream>
using namespace std;
// Physical base address of FPGA Devices
const unsigned int LW_BRIDGE_BASE  = 0xFF200000;  // Base offset
// Length of memory-mapped IO window
const unsigned int LW_BRIDGE_SPAN  = 0x00DEC700;  // Address map size
// Cyclone V FPGA device addresses
const unsigned int LEDR_BASE       = 0x00000000;  // Leds offset
const unsigned int SW_BASE         = 0x00000040;  // Switches offset
const unsigned int KEY_BASE        = 0x00000050;  // Push buttons offset


/**
* Write a 4-byte value at the specified general-purpose I/O location.
*
* @param pBase   Base address returned by 'mmap'.
* @parem offset  Offset where device is mapped.
* @param value   Value to be written.
*/
void RegisterWrite(char *pBase, unsigned int reg_offset, int value)
{
  * (volatile unsigned int *)(pBase + reg_offset) = value;
}


/**
* Read a 4-byte value from the specified general-purpose I/O location.
*
* @param pBase   Base address returned by 'mmap'.
* @param offset  Offset where device is mapped.
* @return        Value read.
*/
int RegisterRead(char *pBase, unsigned int reg_offset)
{
   return * (volatile unsigned int *)(pBase + reg_offset);
}
```

```
void WriteAllLeds(char *pBase, int value)
{
  RegisterWrite(pBase, LEDR_BASE, value);
}


/**
 * Initialize general-purpose I/O
 *  - Opens access to physical memory /dev/mem
 *  - Maps memory into virtual address space
 *
 * @param fd      File descriptor passed by reference, where the result
 *               of function 'open' will be stored.
 * @return  Address to virtual memory which is mapped to physical, or MAP_FAILED on error.
 */
char *Initialize(int *fd)
{
 // Open /dev/mem to give access to physical addresses
 *fd = open( "/dev/mem", (O_RDWR | O_SYNC));
 if (*fd == -1)   //  check for errors in openning /dev/mem
  {
   cout << "ERROR: could not open /dev/mem..." << endl;
   exit(1);
  }
  // Get a mapping from physical addresses to virtual addresses
  char *virtual_base = (char *)mmap (NULL, LW_BRIDGE_SPAN, (PROT_READ
   | PROT_WRITE), MAP_SHARED, *fd, LW_BRIDGE_BASE);
     if (virtual_base == MAP_FAILED)       // check for errors
     {
      cout << "ERROR: mmap() failed..." << endl;
      close (*fd);           // close memory before exiting
      exit(1);        // Returns 1 to the operating system;
      }
     return virtual_base;
}


/**
  * Close general-purpose I/O.
  *
  * @param pBase   Virtual address where I/O was mapped.
  * @param fd      File descriptor previously returned by 'open'.
```

```
*/
void Finalize(char *pBase, int fd)
{
   if (munmap (pBase, LW_BRIDGE_SPAN) != 0)
   {
    cout << "ERROR: munmap() failed..." << endl;
    exit(1);
    }
  close (fd);    // close memory
}

/*
 * Calculates the value of the base rased to the power of the exponent.
 * @Param base - the base to calculate the power for
 * @Param exp - the exponent the base should be raised to
 * Returns value representing the value of the operation
 */
int CalculatePower(int base, int exp) {
  int value = 1;
  for(int i = 0; i < exp; i++) {
    value = value * base;
  }
  return value;
}

/*
 * Changes the state of an LED (ON or OFF)
 *  @param pBase          Base address returned by 'mmap'
 *  @param ledNum         LED Number (0 to 9)
 *  @param state          State to change to (ON or OFF)
 */
void Write1Led(char *pBase,int ledNum, int state)
{

    // calculate the base 2 representation of ledNum
    int value = CalculatePower(2, ledNum);

    // read the current value of the led
    int curState = RegisterRead(pBase, LEDR_BASE);
    // // if the LED is on and it should be off, set the value to 0
    // if(curState == value && state == 0) {
    //   value = 0;
    // }
  if(state == 1) {
    curState = curState | value;
  } else {
    curState = (curState | value) ^ value;
  }
  // write the value to the board
  WriteAllLeds(pBase, curState);
}



/*
 * Reads all the switches and returns their value in a single integer
 * @param pBase    Base address for general-purpose I/O
 * @return         A value that represents the value of the switches
 */
int ReadAllSwitches(char *pBase)
{
 return RegisterRead(pBase, SW_BASE);
}

int main()
{
 // Initialize
 int fd;
 char *pBase = Initialize(&fd);

 int value = 0;
 int ledNum = 0;
 // get user input
 cout << "Enter an LED number (0 to 9):_" << endl;
 cin >> ledNum;
 cout << "Led Number chosen: " << ledNum << endl;
 cout << "Enter a state for this LED (0 or 1):_" << endl;
 cin >> value;
 cout << "State chosen: " << value << endl;
 if(ledNum > 9 || ledNum < 0 || (value != 1 && value != 0)) {
   cout << "Please enter valid values" << endl;
 } else {
   // write the passed-in LED to the board
   Write1Led(pBase, ledNum, value);
 }
 // Done
```

```
  Finalize(pBase, fd);
}
```

Lab 2.2

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <iostream>
using namespace std;
// Physical base address of FPGA Devices
const unsigned int LW_BRIDGE_BASE  = 0xFF200000;  // Base offset
// Length of memory-mapped IO window
const unsigned int LW_BRIDGE_SPAN  = 0x00DEC700;  // Address map size
// Cyclone V FPGA device addresses
const unsigned int LEDR_BASE       = 0x00000000;  // Leds offset
const unsigned int SW_BASE         = 0x00000040;  // Switches offset
const unsigned int KEY_BASE        = 0x00000050;  // Push buttons offset


/**
* Write a 4-byte value at the specified general-purpose I/O location.
*
* @param pBase   Base address returned by 'mmap'.
* @parem offset  Offset where device is mapped.
* @param value   Value to be written.
*/
void RegisterWrite(char *pBase, unsigned int reg_offset, int value)
{
  * (volatile unsigned int *)(pBase + reg_offset) = value;
}


/**
* Read a 4-byte value from the specified general-purpose I/O location.
*
* @param pBase   Base address returned by 'mmap'.
* @param offset  Offset where device is mapped.
* @return        Value read.
*/
int RegisterRead(char *pBase, unsigned int reg_offset)
{
  return * (volatile unsigned int *)(pBase + reg_offset);
}

void WriteAllLeds(char *pBase, int value)
{
  RegisterWrite(pBase, LEDR_BASE, value);
}


/**
 * Initialize general-purpose I/O
 *  - Opens access to physical memory /dev/mem
 *  - Maps memory into virtual address space
 *
 * @param fd      File descriptor passed by reference, where the result
 *               of function 'open' will be stored.
 * @return  Address to virtual memory which is mapped to physical, or MAP_FAILED on error.
 */
char *Initialize(int *fd)
{
 // Open /dev/mem to give access to physical addresses
 *fd = open( "/dev/mem", (O_RDWR | O_SYNC));
 if (*fd == -1)   //  check for errors in openning /dev/mem
  {
   cout << "ERROR: could not open /dev/mem..." << endl;
   exit(1);
  }
  // Get a mapping from physical addresses to virtual addresses
  char *virtual_base = (char *)mmap (NULL, LW_BRIDGE_SPAN, (PROT_READ
   | PROT_WRITE), MAP_SHARED, *fd, LW_BRIDGE_BASE);
     if (virtual_base == MAP_FAILED)       // check for errors
     {
      cout << "ERROR: mmap() failed..." << endl;
      close (*fd);           // close memory before exiting
      exit(1);        // Returns 1 to the operating system;
      }
     return virtual_base;
}


/**
  * Close general-purpose I/O.
  *
  * @param pBase   Virtual address where I/O was mapped.
  * @param fd      File descriptor previously returned by 'open'.
*/
void Finalize(char *pBase, int fd)
{
   if (munmap (pBase, LW_BRIDGE_SPAN) != 0)
    {
    cout << "ERROR: munmap() failed..." << endl;
    exit(1);
    }
  close (fd);    // close memory
}

/*
 * Calculates the value of the base rased to the power of the exponent.
 * @Param base - the base to calculate the power for
 * @Param exp - the exponent the base should be raised to
 * Returns value representing the value of the operation
 */
int CalculatePower(int base, int exp) {
  int value = 1;
  for(int i = 0; i < exp; i++) {
    value = value * base;
  }
  return value;
}

/*
 * Changes the state of an LED (ON or OFF)
 *  @param pBase          Base address returned by 'mmap'
 *  @param ledNum         LED Number (0 to 9)
 *  @param state          State to change to (ON or OFF)
 */
void Write1Led(char *pBase,int ledNum, int state)
{

    // calculate the base 2 representation of ledNum
    int value = CalculatePower(2, ledNum);

    // read the current value of the led
    int curState = RegisterRead(pBase, LEDR_BASE);
    // // if the LED is on and it should be off, set the value to 0
    // if(curState == value && state == 0) {
```

```
    //   value = 0;
    // }
    if(state == 1) {
      curState = curState | value;
    } else {
      curState = (curState | value) ^ value;
    }
    // write the value to the board
    WriteAllLeds(pBase, curState);
}



/*
 * Reads all the switches and returns their value in a single integer
 * @param pBase    Base address for general-purpose I/O
 * @return         A value that represents the value of the switches
 */
int ReadAllSwitches(char *pBase)
{
  return RegisterRead(pBase, SW_BASE);
}

/*
 * A helper to determine whether the passed in switch (represented by value) is on or off.
 * @param switchNum representing the number switch the user wishes to see the state of
 * @param curSwitchValues - the decimal form of the base 2 representation of the switch State
 * Returns boolean representing whether the value is present in the curSwitchValues, meaning the switch
is on
 */
bool ReadSwitchHelper(int switchNum, int curSwitchValues) {
   // set alias of curSwitchValues to subtract from
   int curBase2 = curSwitchValues;
   // set default value of returnBool to false
   bool returnBool = false;

   // set up an empty array to store switch values
   int switches[10];

   // for each switch
   for(int i = 9; i >= 0; i--) {
     // calculate the power of 2 to the power of i
     int power = CalculatePower(2, i);
     // if the current base 2 value is greater than power, the switch is on
     if(curBase2 >= power) {
       // set the switch array value to 1
       switches[i] = 1;
       // subtract the power so the next switch can be read
       curBase2 = curBase2 - power;
     } else {
       // the current switch isn't on because the power is greater than the current base
       switches[i] = 0;
     }
   }
   // return the switch state of the desired switch, which is 1 or 0 which translates to true or false
   return switches[switchNum];
 }

/*
 *Reads the value of a switch
 * -Uses base address of I/O
 * @param pBaseBase address returned by 'mmap'
 * @param switchNumSwitch number(0 to 9)
 * @returnSwitch value read
 */
 int Read1Switch(char *pBase,int switchNum) {
   // calculate the binary value of the switchNum by calculating 2 ^ switchNum
   int value = CalculatePower(2, switchNum);
   // read the value of the current switches
   int curSwitchValues = ReadAllSwitches(pBase);
   // if the value equals the value of the current switches, the passed-in switch is on
   if(ReadSwitchHelper(switchNum, curSwitchValues)) {
     return 1;
   // otherwise, the passed-in switch is off
   } else {
     return 0;
   }
 }



int main()
{
  // Initialize
  int fd;
  char *pBase = Initialize(&fd);
```

```
  int switchNum = 0;
  // get user input
  cout << "Enter a switch to read (0 to 9):_" << endl;
  cin >> switchNum;
  // if the input is out of range, let the user know
  if(switchNum > 9 || switchNum < 0) {
    cout << "Please enter a valid switch" << endl;
  } else {
    // read the value of the user-defined switch
    cout << Read1Switch(pBase, switchNum) << endl;
  }
  // Done
  Finalize(pBase, fd);
}
```

Lab 2.3

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <iostream>
using namespace std;
// Physical base address of FPGA Devices
const unsigned int LW_BRIDGE_BASE  = 0xFF200000;  // Base offset
// Length of memory-mapped IO window
const unsigned int LW_BRIDGE_SPAN  = 0x00DEC700;  // Address map size
// Cyclone V FPGA device addresses
const unsigned int LEDR_BASE       = 0x00000000;  // Leds offset
const unsigned int SW_BASE         = 0x00000040;  // Switches offset
const unsigned int KEY_BASE        = 0x00000050;  // Push buttons offset

int buttonValues = 0;

/**
* Write a 4-byte value at the specified general-purpose I/O location.
*
* @param pBase   Base address returned by 'mmap'.
* @parem offset  Offset where device is mapped.
* @param value   Value to be written.
*/
void RegisterWrite(char *pBase, unsigned int reg_offset, int value)
```

```
{
  * (volatile unsigned int *)(pBase + reg_offset) = value;
}


/**
* Read a 4-byte value from the specified general-purpose I/O location.
*
* @param pBase   Base address returned by 'mmap'.
* @param offset  Offset where device is mapped.
* @return        Value read.
*/
int RegisterRead(char *pBase, unsigned int reg_offset)
{
  return * (volatile unsigned int *)(pBase + reg_offset);
}

void WriteAllLeds(char *pBase, int value)
{
  RegisterWrite(pBase, LEDR_BASE, value);
}


/**
 * Initialize general-purpose I/O
 *  - Opens access to physical memory /dev/mem
 *  - Maps memory into virtual address space
 *
 * @param fd      File descriptor passed by reference, where the result
 *               of function 'open' will be stored.
 * @return  Address to virtual memory which is mapped to physical, or MAP_FAILED on error.
 */
char *Initialize(int *fd)
{
 // Open /dev/mem to give access to physical addresses
 *fd = open( "/dev/mem", (O_RDWR | O_SYNC));
 if (*fd == -1)   //  check for errors in openning /dev/mem
  {
   cout << "ERROR: could not open /dev/mem..." << endl;
   exit(1);
  }
  // Get a mapping from physical addresses to virtual addresses
  char *virtual_base = (char *)mmap (NULL, LW_BRIDGE_SPAN, (PROT_READ
   | PROT_WRITE), MAP_SHARED, *fd, LW_BRIDGE_BASE);
      if (virtual_base == MAP_FAILED)       // check for errors
      {
       cout << "ERROR: mmap() failed..." << endl;
       close (*fd);           // close memory before exiting
       exit(1);        // Returns 1 to the operating system;
       }
      return virtual_base;
}


/**
  * Close general-purpose I/O.
  *
  * @param pBase   Virtual address where I/O was mapped.
  * @param fd      File descriptor previously returned by 'open'.
*/
void Finalize(char *pBase, int fd)
{
   if (munmap (pBase, LW_BRIDGE_SPAN) != 0)
    {
    cout << "ERROR: munmap() failed..." << endl;
    exit(1);
    }
  close (fd);    // close memory
}

/*
 * Calculates the value of the base rased to the power of the exponent.
 * @Param base - the base to calculate the power for
 * @Param exp - the exponent the base should be raised to
 * Returns value representing the value of the operation
 */
int CalculatePower(int base, int exp) {
  int value = 1;
  for(int i = 0; i < exp; i++) {
    value = value * base;
  }
  return value;
}

/*
 * Changes the state of an LED (ON or OFF)
 *  @param pBase         Base address returned by 'mmap'
 *  @param ledNum        LED Number (0 to 9)
 *  @param state         State to change to (ON or OFF)
 */
void Write1Led(char *pBase,int ledNum, int state)
{

   // calculate the base 2 representation of ledNum
   int value = CalculatePower(2, ledNum);

   // read the current value of the led
   int curState = RegisterRead(pBase, LEDR_BASE);
   // // if the LED is on and it should be off, set the value to 0
   // if(curState == value && state == 0) {
   //   value = 0;
   // }
   if(state == 1) {
     curState = curState | value;
   } else {
     curState = (curState | value) ^ value;
   }
   // write the value to the board
   WriteAllLeds(pBase, curState);
}



/*
 * Reads all the switches and returns their value in a single integer
 * @param pBase    Base address for general-purpose I/O
 * @return         A value that represents the value of the switches
 */
int ReadAllSwitches(char *pBase)
{
  return RegisterRead(pBase, SW_BASE);
}

/*
 * Reads all the buttons and returns their value in a single integer
 * @param pBase    Base address for general-purpose I/O
 * @return         A value that represents the value of the buttons
 */
int ReadAllButtons(char *pBase)
```

```
{
 return RegisterRead(pBase, KEY_BASE);
}


/*
 * A helper to determine whether the passed in switch (represented by value) is on or off.
 * @param switchNum representing the number switch the user wishes to see the state of
 * @param curSwitchValues - the decimal form of the base 2 representation of the switch State
 * Returns boolean representing whether the value is present in the curSwitchValues, meaning the switch
is on
 */
bool ReadSwitchHelper(int switchNum, int curSwitchValues) {
   // set alias of curSwitchValues to subtract from
   int curBase2 = curSwitchValues;
   // set default value of returnBool to false
   bool returnBool = false;

   // set up an empty array to store switch values
   int switches[10];

   // for each switch
   for(int i = 9; i >= 0; i--) {
     // calculate the power of 2 to the power of i
     int power = CalculatePower(2, i);
     // if the current base 2 value is greater than power, the switch is on
     if(curBase2 >= power) {
       // set the switch array value to 1
       switches[i] = 1;
       // subtract the power so the next switch can be read
       curBase2 = curBase2 - power;
     } else {
       // the current switch isn't on because the power is greater than the current base
       switches[i] = 0;
     }
   }
   // return the switch state of the desired switch, which is 1 or 0 which translates to true or false
   return switches[switchNum];
 }

/*
 *Reads the value of a switch
 * -Uses base address of I/O
 * @param pBaseBase address returned by 'mmap'
 * @param switchNumSwitch number(0 to 9)
 * @returnSwitch value read
 */
 int Read1Switch(char *pBase,int switchNum) {
   // calculate the binary value of the switchNum by calculating 2 ^ switchNum
   int value = CalculatePower(2, switchNum);
   // read the value of the current switches
   int curSwitchValues = ReadAllSwitches(pBase);
   // if the value equals the value of the current switches, the passed-in switch is on
   if(ReadSwitchHelper(switchNum, curSwitchValues)) {
     return 1;
   // otherwise, the passed-in switch is off
   } else {
     return 0;
   }
 }

/*
 * Gets the value of all the push buttons that have been pressed.
 * @param pBase - Base address returned by 'mmap'
 */
int PushButtonGet(char *pBase) {
  return ReadAllButtons(pBase);
}

/*
 * Increments the value of the LEDs by one, and displays the updated LEDs.
 * @param pBase - Base address returned by 'mmap'
 */
void IncrementLeds(char *pBase) {
  cout << "Incrementing..." << endl;
  int currentValue = RegisterRead(pBase, LEDR_BASE);
  currentValue = currentValue + 1;
  cout << currentValue << endl;
  WriteAllLeds(pBase, currentValue);
}

/*
 * Decrements the value of the LEDs by one, and displays the updated LEDs.
 * @param pBase - Base address returned by 'mmap'
 */
void DecrementLeds(char *pBase) {
  cout << "Decrementing..." << endl;
  int currentValue = RegisterRead(pBase, LEDR_BASE);
  currentValue = currentValue - 1;
  cout << currentValue << endl;
  WriteAllLeds(pBase, currentValue);
}

/*
 * Shifts the value of the LEDs by one to the right, inserting a 0 bit on the far left of the binary.
 * @param pBase - Base address returned by 'mmap'
 */
void ShiftRight(char *pBase) {
  cout << "Shifting right..." << endl;
  int currentValue = RegisterRead(pBase, LEDR_BASE);
  currentValue = currentValue >> 1;
  cout << currentValue << endl;
  WriteAllLeds(pBase, currentValue);
}

/*
 * Shifts the value of the LEDs by one to the left, inserting a 0 bit on the far right of the binary.
 * @param pBase - Base address returned by 'mmap'
 */
void ShiftLeft(char *pBase) {
  cout << "Shifting left..." << endl;
  int currentValue = RegisterRead(pBase, LEDR_BASE);
  currentValue = currentValue << 1;
  cout << currentValue << endl;
  WriteAllLeds(pBase, currentValue);
}

/*
 * Handles the different cases for the values when the push buttons are pressed
 * @param pBase - Base address returned by 'mmap'
 * @param buttonValue - the value of the button that has been pressed
 */
void HandlePushButtons(char *pBase, int buttonValue) {
  // if the button's value is 0, don't do anything
  if (buttonValue != 0) {
    // if the value is 1, run the IncrementLeds function
    if(buttonValue == 1) {
      IncrementLeds(pBase);
    // if the value is 2, run the DecrementLeds function
    } else if (buttonValue == 2) {
      DecrementLeds(pBase);
   // if the value is 4, run the ShiftRight function
    } else if (buttonValue == 4) {
      ShiftRight(pBase);
   // if the value is 8, run the ShiftLeft function
    } else if (buttonValue == 8) {
      ShiftLeft(pBase);
   // otherwise, output an error but keep going
    } else {
      cout << "error" << endl;
    }
  }
}

/*
 * Detects whether a push button has been changed - either a new button has been pressed,
 * or a button has been let go of.
 * @param pBase - Base address returned by 'mmap'
 */
void DetectPushButtonChange(char *pBase) {
  // get the current button values
  int newButtonValues = PushButtonGet(pBase);
  // if there's been a change, set the new value of the buttons
  if(newButtonValues != buttonValues) {
    buttonValues = newButtonValues;
    // handle the button press properly
    HandlePushButtons(pBase, newButtonValues);
  }
}

int main()
{
  // Initialize
  int fd;
  char *pBase = Initialize(&fd);
  // detect push button changes
  while(true) {
    DetectPushButtonChange(pBase);
  }
}
```

Lab 2.4

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <iostream>
using namespace std;
// Physical base address of FPGA Devices
const unsigned int LW_BRIDGE_BASE  = 0xFF200000;  // Base offset
// Length of memory-mapped IO window
const unsigned int LW_BRIDGE_SPAN  = 0x00DEC700;  // Address map size
// Cyclone V FPGA device addresses
const unsigned int LEDR_BASE       = 0x00000000;  // Leds offset
const unsigned int SW_BASE         = 0x00000040;  // Switches offset
const unsigned int KEY_BASE        = 0x00000050;  // Push buttons offset
int buttonValues = 0;
class DE1SoCfpga {
  public:
    char *pBase;
    // File descriptor passed by reference, where the result of function 'open' will be stored.
    int fd;


    /**
     * Initialize general-purpose I/O
     *  - Opens access to physical memory /dev/mem
     *  - Maps memory into virtual address space
     */
  DE1SoCfpga()
  {
    // Open /dev/mem to give access to physical addresses
    fd = open( "/dev/mem", (O_RDWR | O_SYNC));
    if (fd == -1)   //  check for errors in openning /dev/mem
     {
       cout << "ERROR: could not open /dev/mem..." << endl;
       exit(1);
     }
     // Get a mapping from physical addresses to virtual addresses
     char *virtual_base = (char *)mmap (NULL, LW_BRIDGE_SPAN, (PROT_READ
       | PROT_WRITE), MAP_SHARED, fd, LW_BRIDGE_BASE);
          if (virtual_base == MAP_FAILED)       // check for errors
          {
           cout << "ERROR: mmap() failed..." << endl;
```

```
        close (fd);            // close memory before exiting
        exit(1);       // Returns 1 to the operating system;
        }
  pBase = virtual_base;
}

/**
  * Close general-purpose I/O.
*/
~DE1SoCfpga()
{
  if (munmap (pBase, LW_BRIDGE_SPAN) != 0)
   {
   cout << "ERROR: munmap() failed..." << endl;
   exit(1);
   }
 close (fd);    // close memory
}

/**
* Write a 4-byte value at the specified general-purpose I/O location.
* @parem offset  Offset where device is mapped.
* @param value   Value to be written.
*/
void RegisterWrite(unsigned int reg_offset, int value)
{
  * (volatile unsigned int *)(pBase + reg_offset) = value;
}


/**
* Read a 4-byte value from the specified general-purpose I/O location.
* @param offset  Offset where device is mapped.
* @return        Value read.
*/
int RegisterRead(unsigned int reg_offset)
{
   return * (volatile unsigned int *)(pBase + reg_offset);
}

void WriteAllLeds(int value)
{
   RegisterWrite(LEDR_BASE, value);
```

```
}

/*
 * Calculates the value of the base rased to the power of the exponent.
 * @Param base - the base to calculate the power for
 * @Param exp - the exponent the base should be raised to
 * Returns value representing the value of the operation
 */
int CalculatePower(int base, int exp) {
  int value = 1;
  for(int i = 0; i < exp; i++) {
    value = value * base;
  }
  return value;
}

/*
 * Changes the state of an LED (ON or OFF)
 *  @param ledNum         LED Number (0 to 9)
 *  @param state          State to change to (ON or OFF)
 */
void Write1Led(int ledNum, int state)
{

    // calculate the base 2 representation of ledNum
    int value = CalculatePower(2, ledNum);

    // read the current value of the led
    int curState = RegisterRead(LEDR_BASE);
    // // if the LED is on and it should be off, set the value to 0
    // if(curState == value && state == 0) {
    //   value = 0;
    // }
    if(state == 1) {
      curState = curState | value;
    } else {
      curState = (curState | value) ^ value;
    }
    // write the value to the board
    WriteAllLeds(curState);
}
```

```
/*
 * Reads all the switches and returns their value in a single integer
 * @return         A value that represents the value of the switches
 */
int ReadAllSwitches()
{
  return RegisterRead(SW_BASE);
}

/*
 * Reads all the buttons and returns their value in a single integer
 * @return         A value that represents the value of the buttons
 */
int ReadAllButtons()
{
  return RegisterRead(KEY_BASE);
}


/*
 * A helper to determine whether the passed in switch (represented by value) is on or off.
 * @param switchNum representing the number switch the user wishes to see the state of
 * @param curSwitchValues - the decimal form of the base 2 representation of the switch State
 * Returns boolean representing whether the value is present in the curSwitchValues, meaning the switch
is on
 */
bool ReadSwitchHelper(int switchNum, int curSwitchValues) {
   // set alias of curSwitchValues to subtract from
   int curBase2 = curSwitchValues;
   // set default value of returnBool to false
   bool returnBool = false;

   // set up an empty array to store switch values
   int switches[10];

   // for each switch
   for(int i = 9; i >= 0; i--) {
     // calculate the power of 2 to the power of i
     int power = CalculatePower(2, i);
     // if the current base 2 value is greater than power, the switch is on
     if(curBase2 >= power) {
       // set the switch array value to 1
```

```
      switches[i] = 1;
      // subtract the power so the next switch can be read
      curBase2 = curBase2 - power;
    } else {
      // the current switch isn't on because the power is greater than the current base
      switches[i] = 0;
    }
  }
  // return the switch state of the desired switch, which is 1 or 0 which translates to true or false
  return switches[switchNum];
 }

/*
 *Reads the value of a switch
 * -Uses base address of I/O
 * @param switchNumSwitch number(0 to 9)
 * @returnSwitch value read
 */
 int Read1Switch(int switchNum) {
  // calculate the binary value of the switchNum by calculating 2 ^ switchNum
  int value = CalculatePower(2, switchNum);
  // read the value of the current switches
  int curSwitchValues = ReadAllSwitches();
  // if the value equals the value of the current switches, the passed-in switch is on
  if(ReadSwitchHelper(switchNum, curSwitchValues)) {
    return 1;
  // otherwise, the passed-in switch is off
  } else {
    return 0;
  }
 }

/*
 * Gets the value of all the push buttons that have been pressed.
 */
int PushButtonGet() {
 return ReadAllButtons();
}

/*
 * Increments the value of the LEDs by one, and displays the updated LEDs.
 */
void IncrementLeds() {
  cout << "Incrementing..." << endl;
  int currentValue = RegisterRead(LEDR_BASE);
  currentValue = currentValue + 1;
  cout << currentValue << endl;
  WriteAllLeds(currentValue);
}

/*
 * Decrements the value of the LEDs by one, and displays the updated LEDs.
 */
void DecrementLeds() {
  cout << "Decrementing..." << endl;
  int currentValue = RegisterRead(LEDR_BASE);
  currentValue = currentValue - 1;
  cout << currentValue << endl;
  WriteAllLeds(currentValue);
}

/*
 * Shifts the value of the LEDs by one to the right, inserting a 0 bit on the far left of the binary.
 */
void ShiftRight() {
  cout << "Shifting right..." << endl;
  int currentValue = RegisterRead(LEDR_BASE);
  currentValue = currentValue >> 1;
  cout << currentValue << endl;
  WriteAllLeds(currentValue);
}

/*
 * Shifts the value of the LEDs by one to the left, inserting a 0 bit on the far right of the binary.
 */
void ShiftLeft() {
  cout << "Shifting left..." << endl;
  int currentValue = RegisterRead(LEDR_BASE);
  currentValue = currentValue << 1;
  cout << currentValue << endl;
  WriteAllLeds(currentValue);
}

/*
 * Handles the different cases for the values when the push buttons are pressed
 * @param buttonValue - the value of the button that has been pressed
```

```
  */
 void HandlePushButtons(int buttonValue) {
  // if the button's value is 0, don't do anything
  if (buttonValue != 0) {
   // if the value is 1, run the IncrementLeds function
   if(buttonValue == 1) {
    IncrementLeds();
   // if the value is 2, run the DecrementLeds function
   } else if (buttonValue == 2) {
    DecrementLeds();
   // if the value is 4, run the ShiftRight function
   } else if (buttonValue == 4) {
    ShiftRight();
   // if the value is 8, run the ShiftLeft function
   } else if (buttonValue == 8) {
    ShiftLeft();
   // otherwise, output an error but keep going
   } else {
    cout << "error" << endl;
   }
  }
 }

 /*
  * Detects whether a push button has been changed - either a new button has been pressed,
  * or a button has been let go of.
  */
 void DetectPushButtonChange() {
  // get the current button values
  int newButtonValues = PushButtonGet();
  // if there's been a change, set the new value of the buttons
  if(newButtonValues != buttonValues) {
   buttonValues = newButtonValues;
   // handle the button press properly
   HandlePushButtons(newButtonValues);
  }
 }
};

int main()
{
 DE1SoCfpga board;
 // detect push button changes
```

```
  while(true) {
    board.DetectPushButtonChange();
  }
  return 0;
}
```

Lab 2.5

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <iostream>
using namespace std;
// Physical base address of FPGA Devices
const unsigned int LW_BRIDGE_BASE  = 0xFF200000;  // Base offset
// Length of memory-mapped IO window
const unsigned int LW_BRIDGE_SPAN  = 0x00DEC700;  // Address map size
// Cyclone V FPGA device addresses
const unsigned int LEDR_BASE       = 0x00000000;  // Leds offset
const unsigned int SW_BASE         = 0x00000040;  // Switches offset
const unsigned int KEY_BASE        = 0x00000050;  // Push buttons offset
int buttonValues = 0;

class DE1SoCfpga {
  public:
    char *pBase;
    // File descriptor passed by reference, where the result of function 'open' will be stored.
    int fd;


    /**
     * Initialize general-purpose I/O
     *  - Opens access to physical memory /dev/mem
     *  - Maps memory into virtual address space
     */
  DE1SoCfpga()
  {
    // Open /dev/mem to give access to physical addresses
    fd = open( "/dev/mem", (O_RDWR | O_SYNC));
    if (fd == -1)   //  check for errors in openning /dev/mem
     {
       cout << "ERROR: could not open /dev/mem..." << endl;
       exit(1);
     }
     // Get a mapping from physical addresses to virtual addresses
     char *virtual_base = (char *)mmap (NULL, LW_BRIDGE_SPAN, (PROT_READ
       | PROT_WRITE), MAP_SHARED, fd, LW_BRIDGE_BASE);
         if (virtual_base == MAP_FAILED)       // check for errors
         {
          cout << "ERROR: mmap() failed..." << endl;
          close (fd);            // close memory before exiting
          exit(1);        // Returns 1 to the operating system;
          }
    pBase = virtual_base;
  }

  /**
    * Close general-purpose I/O.
  */
  ~DE1SoCfpga()
  {
    if (munmap (pBase, LW_BRIDGE_SPAN) != 0)
    {
     cout << "ERROR: munmap() failed..." << endl;
     exit(1);
     }
   close (fd);    // close memory
  }

  /**
  * Write a 4-byte value at the specified general-purpose I/O location.
  * @parem offset  Offset where device is mapped.
  * @param value   Value to be written.
  */
  void RegisterWrite(unsigned int reg_offset, int value)
  {
    * (volatile unsigned int *)(pBase + reg_offset) = value;
  }


  /**
  * Read a 4-byte value from the specified general-purpose I/O location.
  * @param offset  Offset where device is mapped.
  * @return        Value read.
  */
```

```
  int RegisterRead(unsigned int reg_offset)
  {
    return * (volatile unsigned int *)(pBase + reg_offset);
  }


};


class LedControl : public DE1SoCfpga
{
public:
  LedControl() {}

  ~LedControl() {}

  void WriteAllLeds(int value)
  {
    RegisterWrite(LEDR_BASE, value);
  }

  /*
   * Calculates the value of the base rased to the power of the exponent.
   * @Param base - the base to calculate the power for
   * @Param exp - the exponent the base should be raised to
   * Returns value representing the value of the operation
   */
  int CalculatePower(int base, int exp) {
   int value = 1;
   for(int i = 0; i < exp; i++) {
     value = value * base;
   }
   return value;
  }

  /*
   * Changes the state of an LED (ON or OFF)
   *  @param pBase          Base address returned by 'mmap'
   *  @param ledNum         LED Number (0 to 9)
   *  @param state          State to change to (ON or OFF)
   */
  void Write1Led(int ledNum, int state)
  {
    // calculate the base 2 representation of ledNum
    int value = CalculatePower(2, ledNum);

    // read the current value of the led
    int curState = RegisterRead(LEDR_BASE);
    // // if the LED is on and it should be off, set the value to 0
    // if(curState == value && state == 0) {
    //   value = 0;
    // }
    if(state == 1) {
      curState = curState | value;
    } else {
      curState = (curState | value) ^ value;
    }
    // write the value to the board
    WriteAllLeds(curState);
}



/*
 * Reads all the switches and returns their value in a single integer
 * @return         A value that represents the value of the switches
 */
int ReadAllSwitches()
{
  return RegisterRead(SW_BASE);
}

/*
 * Reads all the buttons and returns their value in a single integer
 * @return         A value that represents the value of the buttons
 */
int ReadAllButtons()
{
  return RegisterRead(KEY_BASE);
}


/*
 * A helper to determine whether the passed in switch (represented by value) is on or off.
 * @param switchNum representing the number switch the user wishes to see the state of
```

```
  * @param curSwitchValues - the decimal form of the base 2 representation of the switch State
  * Returns boolean representing whether the value is present in the curSwitchValues, meaning the switch
is on
  */
 bool ReadSwitchHelper(int switchNum, int curSwitchValues) {
   // set alias of curSwitchValues to subtract from
   int curBase2 = curSwitchValues;
   // set default value of returnBool to false
   bool returnBool = false;

   // set up an empty array to store switch values
   int switches[10];

   // for each switch
   for(int i = 9; i >= 0; i--) {
     // calculate the power of 2 to the power of i
     int power = CalculatePower(2, i);
     // if the current base 2 value is greater than power, the switch is on
     if(curBase2 >= power) {
       // set the switch array value to 1
       switches[i] = 1;
       // subtract the power so the next switch can be read
       curBase2 = curBase2 - power;
     } else {
       // the current switch isn't on because the power is greater than the current base
       switches[i] = 0;
     }
   }
   // return the switch state of the desired switch, which is 1 or 0 which translates to true or false
   return switches[switchNum];
  }

 /*
  *Reads the value of a switch
  * -Uses base address of I/O
  * @param switchNumSwitch number(0 to 9)
  * @returnSwitch value read
  */
  int Read1Switch(int switchNum) {
   // calculate the binary value of the switchNum by calculating 2 ^ switchNum
   int value = CalculatePower(2, switchNum);
   // read the value of the current switches
   int curSwitchValues = ReadAllSwitches();
   // if the value equals the value of the current switches, the passed-in switch is on
   if(ReadSwitchHelper(switchNum, curSwitchValues)) {
     return 1;
   // otherwise, the passed-in switch is off
   } else {
     return 0;
   }
 }

/*
 * Gets the value of all the push buttons that have been pressed.
 */
int PushButtonGet() {
  return ReadAllButtons();
}

/*
 * Increments the value of the LEDs by one, and displays the updated LEDs.
 */
void IncrementLeds() {
  cout << "Incrementing..." << endl;
  int currentValue = RegisterRead(LEDR_BASE);
  currentValue = currentValue + 1;
  cout << currentValue << endl;
  WriteAllLeds(currentValue);
}

/*
 * Decrements the value of the LEDs by one, and displays the updated LEDs.
 */
void DecrementLeds() {
  cout << "Decrementing..." << endl;
  int currentValue = RegisterRead(LEDR_BASE);
  currentValue = currentValue - 1;
  cout << currentValue << endl;
  WriteAllLeds(currentValue);
}

/*
 * Shifts the value of the LEDs by one to the right, inserting a 0 bit on the far left of the binary.
 */
void ShiftRight() {
  cout << "Shifting right..." << endl;
  int currentValue = RegisterRead(LEDR_BASE);
  currentValue = currentValue >> 1;
  cout << currentValue << endl;
  WriteAllLeds(currentValue);
}

/*
 * Shifts the value of the LEDs by one to the left, inserting a 0 bit on the far right of the binary.
 */
void ShiftLeft() {
  cout << "Shifting left..." << endl;
  int currentValue = RegisterRead(LEDR_BASE);
  currentValue = currentValue << 1;
  cout << currentValue << endl;
  WriteAllLeds(currentValue);
}

/*
 * Handles the different cases for the values when the push buttons are pressed
 * @param buttonValue - the value of the button that has been pressed
 */
void HandlePushButtons(int buttonValue) {
  // if the button's value is 0, don't do anything
  if (buttonValue != 0) {
    // if the value is 1, run the IncrementLeds function
    if(buttonValue == 1) {
      IncrementLeds();
    // if the value is 2, run the DecrementLeds function
    } else if (buttonValue == 2) {
      DecrementLeds();
    // if the value is 4, run the ShiftRight function
    } else if (buttonValue == 4) {
      ShiftRight();
    // if the value is 8, run the ShiftLeft function
    } else if (buttonValue == 8) {
      ShiftLeft();
    // otherwise, output an error but keep going
    } else {
      cout << "error" << endl;
    }
  }
}
```

```
  /*
   * Detects whether a push button has been changed - either a new button has been pressed,
   * or a button has been let go of.
   */
  void DetectPushButtonChange() {
    // get the current button values
    int newButtonValues = PushButtonGet();
    // if there's been a change, set the new value of the buttons
    if(newButtonValues != buttonValues) {
      buttonValues = newButtonValues;
      // handle the button press properly
      HandlePushButtons(newButtonValues);
    }
  }
};

int main()
{
  DE1SoCfpga board;
  LedControl control;
  // detect push button changes
  while(true) {
    control.DetectPushButtonChange();
  }
  return 0;
}
```