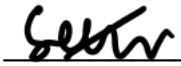
Lab 1: LED Lights Control

Name: Seth Myers
UIN: 01228160
ECE 346 – Microcontrollers
Dr. Chen
Spring 2025

Honor Pledge:

"I pledge to support the Honor System of Old Dominion University. I will refrain from any form of academic dishonesty or deception, such as cheating or plagiarism. I am aware that as a member of the academic community, it is my responsibility to turn in all suspected violators of the Honor System. I will report to an Honor Council hearing if summoned."

Signature: 

Introduction:

In this lab we introduced basic Assembly and C++ code implementation and modification. We also learned how to compile and run this code on an FPGA board. The goal of the program is simple – to display several different patterns of lights on the FPGA's on-board LED array. Although this is a simple program, it helps to become familiar with many of the processes that will be used in future labs and in actual FPGA development environments. This lab will also allow me to become familiar with the differences between a low-level coding language like Assembly and higher-level languages like C.

Background:

Field-programmable gate array (FPGA) boards are commonly used in many different technology fields today because of their versatility and efficiency for solving individual problems. FPGAs must be reconfigured with new instructions for each new problem they are trying to solve, instead of being generalized to solve many problems. While this makes them more complicated to configure than a CPU, it means that they are much more efficient at solving the problem they are configured for. Creation of programs for the FPGA can be done with low-level Assembly code or higher-level C code, which can then be compiled to Assembly code by the Intel FPGA software.

Preliminary:

Since this experiment is focused on the software, there were no wire lists or physical materials used. The only preparations made were ensuring that Quartus and the Intel FPGA program were correctly installed and downloading the files from Canvas. During the in-class section of the lab, we implemented the given code to make the on-board LED lights flash one at a time from right to left. This was accomplished by treating the row of LEDs as an 8-bit binary number and performing logical shifts by 1 bit to the left starting at 00000001 and ending at 10000000. The current location of the '1' corresponds to the currently lit LED. In the Assembly code, this value was stored in the registry location R6. Both the Assembly code and C code were provided to me for this lab.

Experiment:

For the take-home section of this lab, I was tasked with modifying the code from the in-class section to perform two new tasks. For the first part, I needed to make the light pattern move in the opposite direction (left-to-right). To accomplish this, I needed to make several changes to the given code. Firstly, I changed the initial value of R6 to be the rightmost bound 10000000 instead of the leftmost 00000001. After this, I changed the logic of the check function to perform a logical right shift instead of a left shift. This required me to change the comparison condition for the loop to compare against the leftmost bound instead of the rightmost bound.

```assembly
.include    "address_map_arm.s"

.text       /* executable code follows */
.global     _start
_start:
    LDR     R2, =LED_BASE   // base address of LED lights
    LDR     R3, =0b00000001 // LED right bound
    LDR     R4, =0b10000000 // LED left bound
    LDR     R5, =0X02FFFFFF // for delay
    MOV     R6, R4          // R6 --> LED initialization
    MOV     R7, #0      // R7 counter

DISPLAY:
    STR     R6, [R2]        // load SW switches

DELAY:
    ADD     R7, R7, #1  // R7 = R7 + 1
    CMP     R7, R5
    BEQ     CHECK
    B       DELAY

CHECK:
    MOV     R7, #0
    CMP     R6, R3
    BEQ     RESET
    LSR     R6, R6, #1
    B       DISPLAY

RESET:
    MOV     R6, R4
    B       DISPLAY
```

Remote Window

```c
#include "./address_map_arm.h"

int main(void) {

    //base address of LED lights
    volatile int* LED_ptr = (int*)LED_BASE;

    //LED right bound
    int rightBound = 1;

    //LED left bound
    int leftBound = 0b10000000;

    // for delay
    int DELAY = 0X02FFFFFF;

    int currentState = leftBound*2;

    while (1) {
        if (currentState != rightBound) { //check to see if Left bound reached
            currentState = currentState >> 1; // if not, shift state to the right
        }
        else {
            currentState = leftBound; // if yes, set current state to right bound again
        }

        *(LED_ptr) = currentState; //update LEDs to current state
        for (int i = 0; i < DELAY; i++) {}; //DELAY
    }
}
```

Figure 1 & Figure 2: Assembly and C code for Lab 1_3

Finally, for the last part I needed to combine the two pieces of code to make the LEDs flash in both directions. Although there may have been a more efficient method, I found that simply combining the two programs and changing where each bridge points to worked well enough for the Assembly code. I created separate left-to-right and right-to-left functions for displaying and moving the current location. The C code was modified in a similar way, adding a variable for the current direction of movement and cycling through a different loop based on the direction.

```assembly
.include    "address_map_arm.s"

.text       /* executable code follows */
.global     _start
_start:
    LDR     R2, =LED_BASE   // base address of LED lights
    LDR     R3, =0b00000001 // LED right bound
    LDR     R4, =0b10000000 // LED left bound
    LDR     R5, =0X02FFFFFF // for delay
    MOV     R6, R3          // R6 --> LED initialization
    MOV     R7, #0      // R7 counter

DISPLAYL:
    STR     R6, [R2]        // load SW switches

DELAYL:
    ADD     R7, R7, #1  // R7 = R7 + 1
    CMP     R7, R5
    BEQ     CHECKL
    B       DELAYL

CHECKL:
    MOV     R7, #0
    CMP     R6, R4
    BEQ     CHECKR
    LSL     R6, R6, #1
    B       DISPLAYL

DISPLAYR:
    STR     R6, [R2]        // load SW switches

DELAYR:
    ADD     R7, R7, #1  // R7 = R7 + 1
    CMP     R7, R5
    BEQ     CHECKR
    B       DELAYR

CHECKR:
    MOV     R7, #0
    CMP     R6, R3
    BEQ     CHECKL
    LSR     R6, R6, #1
    B       DISPLAYR
```

```c
#include "./address_map_arm.h"

int main(void) {

    //base address of LED lights
    volatile int* LED_ptr = (int*)LED_BASE;

    //LED right bound
    int rightBound = 1;

    //LED left bound
    int leftBound = 0b10000000;

    // for delay
    int DELAY = 0X02FFFFFF;

    int right = 0;

    int currentState = rightBound;

    while (1) {
        if (right == 0) {
            if (currentState != leftBound) { //check to see if left bound reached
                currentState = currentState << 1; // if not, shift state to the left
            }
            else {
                right = 1;
            }
        }

        if (right == 1) {
            if (currentState != rightBound) { //check to see if right bound reached
                currentState = currentState >> 1; // if not, shift state to the right
            }
            else {
                right = 0;
            }
        }


        *(LED_ptr) = currentState; //update LEDs to current state
        for (int i = 0; i < DELAY; i++) {}; //DELAY
    }
}
```

Figure 3 & Figure 4: Assembly and C code for Lab1_4

Results:

I was able to successfully modify the program to switch the direction of the LEDs and combine the two programs to make the LEDs flash in both directions. In the attached video demonstration, we can see that the light pattern now flows from left to right as expected. Another video demonstration shows the lights going from right-to-left and then back in the opposite direction.

However, as the video also demonstrates, the code that I created to make the lights go in both directions is not perfect. For a reason I was unable to determine, the program shifts one too many bits to the left (100000000), creating a brief period where none of the LEDs are lit after it reaches the leftmost 10000000. None of the LEDs are illuminated because the value of R6 at that moment overflows 8 bits so it cannot be displayed on the 8-bit mapped LEDs. I tried several debugging steps, but I was ultimately unable to solve this issue.

Conclusion:

In conclusion, this lab was a good introduction to FPGAs, Assembly instructions, and creating programs with Assembly and C. I successfully created a simple program to display a binary pattern on the FPGA's LED light array and modified it to display different patterns. This required me to look at the code and decipher the purpose of each line to modify it. This knowledge forms the foundation of FPGA programming, and we will continue to build on it for the rest of this lab.