ECSE 324 Lab Report 3

Marie-Lynn Mansour - 260770547
Haluk Calin - 260790895

Part 1: Basic I/O

1.1 Slider Switches & LEDs Program

The overall goal of this lab was to utilize the I/O functionalities of the DE1-SoC computer. The focus of this particular section was to work with and get accustomed to the slider switches and LED functionalities of the board. The first part of this lab was to create an interaction between the LEDs and the slider switches just beneath it. When the switches were switched to the "on" mode, the LEDs were expected to light up. Similarly, if the switches were "off", the LEDs were expected to turn off.

The actual functionality was taken care of in the main.c file by initializing an infinite loop that would consistently poll the register that incorporated the data for all the switches, since each slider switch was associated with one bit in the register. The same method was used for the LEDs since they would receive the value of the slider switch using the *read_slider_switches_ASM()* function. The value corresponding to the particular slider switch would then be placed inside the LED register using the *write_LEDS_ASM()* function. Hence, once a slider switch was turned on, the LED associated to it would light up as well.

read_slider_switches_ASM() and *write_LEDS_ASM()* are the drivers used to communicate with the slider switches and the LEDs. They are subroutines that can communicate directly with the devices. Most of the communication with the board's devices is done through registers which contain up-to-date information about their associated devices, such as what data it contains or which mode it's currently in. For LEDs, each bit in the register for LEDs represents if the LED is "on" (associated LED bit set to 1) or "off" (associated LED bit to 0). The figure below demonstrates our use of assembly instructions to set the register, and therefore the state, of the LEDs.

```
write_LEDs_ASM:
    LDR R1, =LED_BASE           // Load memory address of LEDs in R1
    STR R0, [R1]                // Write what is in R0 in the memory address of R1
    BX LR                       // Branch out of the subroutine
```

figure 1: setting the register and the state of the LEDs

A possible improvement for this part of the lab would be to use an interrupt-based system instead of a constant polling system for the state of the slider switches. This would reduce the impact on the CPU and therefore increase its efficiency and improve its performance.

Part 2: Drivers for HEX displays and pushbuttons

2.2 HEX Displays and Pushbuttons

The purpose of this part of the lab was to display multiple digits on all the 7-segment displays (HEX displays) using the 4 pushbuttons available to us on the board in order to specify which display should be "on". The program displays numbers from 0-15 into the desired hexadecimal display according to the binary number represented by the first 4 slider switches. The last switch was used to clear all the hexadecimal displays.

In order to implement this, we used 4 devices: the 10 LEDs, the 10 slider switches, the 4 pushbuttons, and the 6 hexadecimal displays. We wrote multiple assembly drivers for each of these devices. Most of the drivers use a similar logic as the *write_LEDS_ASM()* driver described in Part 1.1. The drivers work by accessing the specific memory address associated with a particular device and, if the device is available, write/read data from it.

The main.c file implements the behavior required during the demo. The overall structure of the program is simple. We have a while loop that constantly polls the state of the slider switches and pushbuttons. There is a switch case for every possible combination of these inputs. We then enter the case associated with the combination of inputs and display the right number and turn on the appropriate LEDs. This is demonstrated in figure 2 below.

```
HEX_flood_ASM(HEX4);
HEX_flood_ASM(HEX5);
char hex_val = (0xF & read_slider_switches_ASM());
int pushbutton = (0xF & read_PB_data_ASM());
hex_val = hex_val + 48;                          // For ASCII usage
HEX_write_ASM(pushbutton, hex_val);              // Write the value on display
```

figure 2: HEX displays using pushbuttons

One issue we faced with this part was with the pushbuttons; we had initially programmed the methods to read from the wrong address on the board. In order to identify this problem, we went through all the methods we had created to make sure that they were properly implemented, finding that the addresses were wrong.

One improvement for this part of the lab would be to implement a mapping system from the hexadecimal number to the proper bits that are required to light up in the 7-segment display. This would increase the efficiency of the program since all the locations would already be stored in an array and the program would just go directly to the specified address instead of incrementing the index in order to reach the right location.

Part 3: Timers

3.1 Polling-Based Stopwatch

The goal of this part of the lab was to create a stopwatch using the hexadecimal displays 0-5 and 3 of the 4 pushbuttons in order to start (first pushbutton), pause (second pushbutton), and reset (third pushbutton) the stopwatch. We had to produce a stopwatch that incremented its count

every 10 milliseconds. The various hexadecimal displays available to us were used for different values of the stopwatch: HEX0 and HEX1 were used to display the milliseconds elapsed, HEX2 and HEX3 were used to display the total seconds elapsed, and HEX4 and HEX5 were used to display the total minutes elapsed.

The board had 4 hardware timers that were available to us. Since the drivers for this part of the lab were provided, we simply implemented our timer in the main.c file. Our code structure had the following specifications: use the hardware timer 0, set the timeout to 10ms, enable loading and interrupts, and start the driver. The address of this structure was passed as an argument to the setup function *HPS_TIM_config_ASM* and our hardware timer was ready to use. This is illustrated in figure 3.

```
HPS_TIM_config_t hps_tim;
hps_tim.tim = TIM0;                          // Use the hardware timer TIM0
hps_tim.timeout = 10000;                     // Set the timeout to 10ms
hps_tim.LD_en = 1;                           // Enable loading
hps_tim.INT_en = 1;                          // Enable interrupts
hps_tim.enable = 1;
HPS_TIM_config_ASM(&hps_tim);                // Configuration for timer 1
```

figure 3: initializing specifications for the timer

The rest of the code was simply an algorithm of a timer. We constantly checked if the 10ms has passed by polling the s-bit of the timer 0. If that bit is 1, that means 10ms has passed and we increment the time counter by 10ms. The time counter was made up of 6 sub-counters: 10th of a millisecond, 100th of a millisecond, seconds, etc. Overflow of these counters was handled by a system of if statements. At the end of each polling cycle, we display the content of each of the 6 sub-counters. The start, stop, and reset functionalities were also implemented by polling the state of the first 3 pushbuttons and applying the corresponding effect when they were pressed. Another thing to note is that we had to implement functionalities that a) the stopwatch can be reset without having to pause the stopwatch first and b) upon resetting, the stopwatch continues right away after being reset (without having to push start again). We implemented these by, when stopping the timer, timerstart is not set to 0, and by setting *timerstart =1* at the end of the if statement respectively.

Part 4: Interrupts

4.1 Interrupt-Based Stopwatch

This part of the lab was very similar to Part 3.1. The difference arises in how we are updating the time on the stopwatch in addition to how we were finding out if a pushbutton was pressed or not. We used an interrupt-based program, implemented in the main.c file, where the timer notified the program if 10ms has passed. More specifically, to check if 10ms had passed, we simply read a flag instead of calling a function to

retrieve the state of the timer (or pushbuttons). It only notified the program when a pushbutton was pressed instead of constantly polling.

One important thing to mention is that interrupts must be set up and they need to be enabled on the devices prior to using them in the C program.

In order to increase efficiency, we would have modularized the code into even smaller subsections to increase efficiency and reduce the number of lines in the code.