Giuseppe Lipari (260742911)
Alex Masciotra (260746829)
Group 45

ECSE 324 Lab 3

Introduction:

This lab introduces the basic input/output capabilities that we've been speaking about for numerous weeks in class. In this lab, we learned to use the slider switches, push-buttons, LEDs and 7-Segment displays. We needed to write assembly drivers that interacted with the input and output. In addition, we needed to use timers and interrupts to demonstrate polling and interrupts in written C language by creating a stopwatch.

Slider Switches and LEDs program:

In the first section of the lab, we were tasked with creating two different routines. Specifically, the slider switches and the LEDs. This program required us to turn on the LED that corresponds to its slider switch right below it. On the contrary, it should also be turned off when its corresponding slider switch is off. The first one was the slider switches. Essentially, we read the value of the memory location designated for the slider switches data into the register R0 and then branched to the link register.

In our program we implemented a subroutine called read_slider_switches_ASM() that reads the value at the memory location of the slider switches into the R0 register.

In the second part of this first section, we needed to create an LEDs routine. This routine contained two subroutines. The first one being the read_LEDs. This subroutine was very similar to the read_slider_switches_ASM(). This read_LEDs was created so that it will load the value at the LEDs memory location into R0 and then branch to LR. The second subroutine was the write_LEDs, which stored the value in R0 at the LEDs memory location, and then branched to LR.

Finally, we were tasked with putting both these routines all together in a C program. This was done by using a loop which would poll the register that contains all the switches constantly, which was implemented on our main c file.

Challengers and improvements:

All in all, this section of the lab was not very challenging and probably the easiest since most of the code was already given to us. In essence, all we had to do was understand the given code and then make it do the opposite for the LEDs.

We don't believe there could have been many improvements in this section of the lab given that it was quite straightforward.

HEX display and pushbutton program:

For this section of the lab we were required to use the 7-segment display on the DE1-SoC board. A new data type known as Hex_t, which is based off a one hot encoding system was introduced in this section of the lab. This encoding system was helpful because it allowed us to write multiple displays in the same function call. The Hex_t uses a logical AND, which turns on the displays that are one-hot encoded. Within this section, we needed to implement three different subroutines.

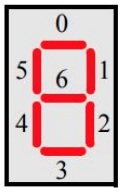
The first subroutine was Hex_clear_ASM which would turn everything off in the seven-segment display. The way we did this was by TSTing the one hot encoded entry with 1, 2, 4, 8 to see which HEX display needed to be cleared. The following snapshot shows how it was then cleared once the right HEX display was found:

```
CLEAR:          CMP R5, #3          //checking if in hex0 base or hex 4 base
                MOVGT R1, R2        //give R1 hexbase4 so above code works
                LDR R7, [R1]
                TST R0, #1
                ANDNE R7, R7, #0xFFFFFF00
                TST R0, #2
                ANDNE R7, R7, #0xFFFF00FF
                TST R0, #4
                ANDNE R7, R7, #0xFF00FFFF
                TST R0, #8
                ANDNE R7, R7, #0x00FFFFFF
                TST R0, #16
                ANDNE R7, R7, #0xFFFFFF00
                TST R0, #32
                ANDNE R7, R7, #0xFFFF00FF

                STR R7, [R1]        //load contents from register of which hexbase is being used


end_Clear:      LSL R3, #1
                ADD R5, R5, #1
                B LOOP_clear

Exit_Hex_Clear: POP {R0-R12,LR}
                BX LR
```

Once the code fell into the proper HEX, the corresponding bits in memory were then cleared by ANDing with 0s. To handle the different memory address of HEX 4 and 5, a counter was implemented to count how many times it goes through the loop, and when it was greater than 3, we loaded the address of the other memory location for HEX 4 and 5.

The next subroutine that we wrote was the HEX_flood_ASM which was very similar to the HEX_clear_ASM. The only difference was that a logical OR with 1s was used in order to have the bits flooded (turn the bits on).

The next subroutine that was written was the HEX_write_ASM. This subroutine takes a character that is between 0 and 15 as its second argument, which will display the hexadecimal digit. To do this, each input was compared to see which hexadecimal needed to be displayed and the corresponding hex value to turn on the appropriate 7-segment bits from this diagram



Segments

was written into the register. To actually write the value, the hex display was first cleared using the code from the clear, and then ADDNE was used adding

the hexadecimal number required to turn on the appropriate bits shifting the right amount. An example is shown below:

```
TST R0, #2
ANDNE R7, R7, #0xFFFF00FF
ADDNE R7, R7, R6, LSL #8
```

As for the pushbutton, in the first part, we mostly read the data register to see which push button was pressed. This was directly mapped to the corresponding HEX display. This data register had 4 bits we cared about, 0-3, when a button was pushed, the value of the data register for the corresponding push button would read 1. For example for PB0, the data register would read 0001. For the data register, once the button was released, the value would go back to zero. The other routine we used was the read edgecap, this one was used for the stopwatch. This works kind of like the data, however the bits stayed 1 until the bits were cleared. They did not automatically reset to 0 when released.

In order to implement all this in our C program, we used a loop to read the slider switches to determine which value to write to display and read the Data register of the push buttons to map to the corresponding HEX display.

Challenges and improvements:

This section of the lab was a little challenging in the sense that it was a little hard to understand how to implement the three subroutines. However, writing the HEX_clear_ASM was not so difficult because one we figured out how to write the HEX_flood_ASM it was more or less the same code with a logical AND, instead of a logical OR. The hardest subroutine was the HEX_write but it wasn't too bad when we realized that it was similar to the other two subroutines with a little bit of added work. The biggest problem faced was that we tried byte accessing instead of word accesing for this register. Once word accessing was used, everything worked.

We believe we wrote this part of the lab efficiently and not many changes need to be made. The only thing that could be changed would be to read edgecaps instead of data since its more precise but for this part of the lab, we did not need to.

Timers:

Stopwatch (with polling):

In this part of the lab we were tasked with creating a stopwatch that ran on increments of 10 milliseconds. It was quite simple in the sense that we needed to display milliseconds on HEX0 and HEX1. HEX2 and HEX3 were used for seconds and finally, HEX4 and HEX5 were used for minutes. The way this works was that we piggy backed every display onto each other. Incrementing the HEX0 with the time clock, HEX1 was relying on HEX0 to reach 10 for HEX1 to increment, HEX 2 was waiting for HEX1 to be 10, HEX 3 was waiting for HEX 2 to reach 10, HEX4 was waiting for HEX 3 to reach 6 since this meant there was 60 seconds in the stopwatch. The pushbuttons were also used in this section of the lab. PB0 was used for the start

button. PB1 was used for the pause button. Finally, PB2 was used for the restart of the stopwatch. Another clock was used that was polling the Edgecap register. When the Edgecap register was not equal to 0, this meant a button was pressed, and then further investigation as to which button was pressed (either start, stop or reset the stopwatch).This was set 10x faster than the stopwatch clock. The stopwatch was controlled by playing with the ENABLE bit of the timer. The program started with the enable set to 0, when start was pressed, it was set to 1, stop was 0 and reset restarted the counters to zero and enable to 0.

PROS - used edgcapture instead of data so when same button was pressed it wouldn't stop, to improve, nothing,

Hard part of this section was clearing edgecaps and reconfiguring the timer after each button was pressed.

INTERRUPTS:

For the interrupt section, it was very similar to the polling part of the code. The way the counters on the hex displays worked was the exact same, but instead of using two timers, one for the stopwatch and one polling the edgecap display, only one timer was used for the stopwatch. Instead, interrupts were used and we implemented in the ISR a keys routine. With the help of the int_setup file, instead of polling the edgecap register, we were "polling" the pb keys flag. This meant when a button was pressed, the program would stop and enter into the ISR and read the pb keys routine. This routine essentially just read the edgecap register and then cleared the edgecap register. The rest was the same in terms of figuring out what button was pressed and so on.

Same difficulty and writing the ISR.

Conclusion:

To conclude, the purpose of this lab was to introduce us to basic input and output applications. We used various attributes on the fpga such as slider switches and pushbuttons in order to display to do three sections. One being to have the slider switches turn on LED lights. The other being to display zeros on HEX0 to HEX5. Finally, we were tasked with creating a stopwatch using both polling method and interrupt method to understand the pros and cons of each.