Abhijeet Praveen
ID: 260985492

Lab 2 Report

ECSE 324
Fall 2021

Part 1.1

Brief Description:

In this part of the lab our task was to use the provided drivers, namely, read_slider_switches_ASM and the write_LEDs_ASM to turn the LEDs on or off.

Approach Taken:

The approach for this part of the lab was simple. Since I just had to turn the LEDs on or off, I would write a subroutine called loop and which would just branch back to itself, making it an endless loop. The body of the loop would consist of branching and linking to the two provided drivers. This can be seen in the code below:

```
loop:
        BL read_slider_switches_ASM
        BL write_LEDs_ASM
        B loop
```

Challenges faced and solution:

I did not find any challenges during completing this section of the lab.

Possible Improvements:

The program is simple and extremely short. It is already optimized and does not need improvements.

Part 1.2

Brief Description:

In this section of the lab, I had to write several drivers so that the seven segment display lights up with the numbers that were selected by a user. The value to be displayed was chosen through pressing any combination of the first four slider switches. As for which segment would display the number, that was governed by the clicking and then de-clicking of one of the four push buttons, which was determined using another driver. Furthermore, drivers were also needed for clearing the display or flood (i.e., light up the complete hex).

Approach Taken:

The drivers for clearing a hex or flooding a hex were extremely similar. The 6 different hex displays were identified using the one-hot encoding scheme. Both the drivers used the scheme to determine which of the six displays needed to be cleared or flooded. In case of flooding, I used the str instruction and moved all 1s to the address of the hex, hence lighting up all the seven segments. To clear, I used the BIC (bit-clear instruction), which would clear every bit and store it back to the address of the hex.

The write was also very similar to the clear and flood drivers. However, this time I would display the actual value that the user wants onto the display. The approach was simple, as we would first test which hex the user wants to write to. Thereafter, we would load the corresponding address of that hex into one of our registers. We would then branch to a subroutine called “loop”; in “loop” we would test which value the user wants to display. Based on the selected value, we would store its corresponding bits to the address of the desired hex display.

Challenges faced and solution:

The hardest challenged I faced in this section was to write the driver for the clear function. Although, in hindsight the code for it was simple. I was not aware of the BIC instruction when I had first started writing the subroutine. Therefore, clearing the whole display was a difficult task but the BIC instruction resulted in an extremely simple solution.

Possible Improvements:

In the driver called HEX_write_ASM, there is a subroutine called “loop” as mentioned earlier. However, when I check for which value is to be displayed in that subroutine, I check each of the 16 values (0-15) each time. Even after figuring out which value needs to be displayed, we still check every value that comes after it. Therefore, after we have stored the value onto the display, we can go branch back instead of checking every value unnecessarily.

Part 2.1

Brief Description:

In this section of the lab, I was tasked with writing drivers for the timer. We would use these drivers to create a counter program. The counter would count from 0 to 15 (“F” in hex), after that cycle is finished, we would go back to 0. The value would be displayed on the first hex (HEX0). Furthermore, the values on the hex display needed to govern which of the LEDs were turned on/off corresponding to the value’s representation in binary. Meaning the first four LEDs would be on if the value was “F” as that is 1111 in binary.

Approach Taken:

The “HEX_write_ASM” was used again to write the display onto only the last display. Drivers were written to configure the timer and for reading and clearing the timer interrupt. If the interrupt was 1, then it would be cleared, and we would execute the main logic of the program and branch back to the beginning of the program as this is a polling-based program. Furthermore, since the display should only count from 0 to 15, if we see our count (register R5) is 16, we cycle back to 0. This can be seen in the code below:

```
BL ARM_TIM_read_INT_ASM
CMP R2,#1
BLEQ ARM_TIM_clear_INT_ASM
ADDEQ R5,R5,#1
CMP R5,#16
MOVEQ R5,#0
MOV R1,R5
MOV R0,#1
BL HEX_write_ASM
BL LED_SWITCH
B main
```

Challenges faced and solution:

During the first test, I realized that my program was going very fast. It would count from 0 to 15, very quickly. The solution was very simple as I realized I was not clearing my ‘f’ bit after clearing it. After doing so, my program was executing its task perfectly.

Possible Improvements:

Like part 1.2, the HEX_write_ASM can be optimized. Furthermore, this program is based on polling which is a very inefficient method. A better approach would be to use interrupts, thereby enabling the CPU to focus on more important tasks.

Abhijeet Praveen
ID: 260985492

Lab 2 Report

ECSE 324
Fall 2021

Part 2.2

Brief Description:

In this section of the lab, I was tasked with creating a stopwatch program using the drivers I had been implementing in the previous parts. All 6 hex displays are used, the first two will showcase the milliseconds, the next two will be seconds, and the last two will be for minutes. Furthermore, only the first three pushbuttons will be used. The first one is equivalent to a start button, the second is a stop button while the third is a reset button.

Approach Taken:

There are three subroutines in my program. These correspond to the three different functions of the three pushbuttons. Therefore, the names of the subroutines are "stop_stopwatch", "start_stopwatch" and "reset_stopwatch". The reset is the simplest of them all, as we just move 0 onto the display and and branch to the stop_stopwatch. In the stop portion, we will check if we want to branch to start or reset, if not we keep branching back to stop as the watch will be stopped. In the start subroutine, we first check if we want to stop or reset the stopwatch. If not, we execute the start function of the stopwatch.

As for the display, I only use one register for the count value. Due to this approach, when my register reaches 10 instead of displaying it, I increment it by 6. This is because the value of 10 corresponds to an "A" being displayed, however, if I add 6 to register, it will mean the value becomes 16, which is 10 in HEX and I will then isolate each digit, the "1" and "0" and display them on the corresponding hex display. This is the same logic that is used for the seconds and minutes as well, however, we check if we the value is a 6 and instead of 10 as there are 60 seconds in one minute and 60 minutes in 1 hour.

Challenges faced and solution:

The hardest challenged I faced in this section was to figure out how I would increment the counter to display all the milliseconds, seconds, and minutes on the display. The solution was easy to think of but hard to implement. It was difficult to only use one register and check different boundaries such as a value of 10 and 6. We would check each of the 6 different digits and then for the display we needed to isolate them and then write them onto the display. Although, the approach was tedious, it ended in the desired result.

Possible Improvements:

As earlier the HEX_write_ASM can be optimized as well using interrupts and not polling. Furthermore, to make the program less complicated and easier to understand, we could use six different registers, each corresponding to one hex display. This would use more registers, but it would be easier to understand if one were to look at the code.

Abhijeet Praveen
ID: 260985492

Part 3

Brief Description:

This section was an altered version of part 2.2. In this portion of the lab, we execute the stopwatch through dealing with interrupt requests. In this section, a lot of the code had already been provided and we had to implement a few additional aspects as well as write an IDLE subroutine which corresponded to the main body of our code from part 2.2.

Approach Taken:

In the _start section of the code, the code for branching to configure timer and enable push buttons was added as specified in the lab instructions. Moreover, the subroutine called "SERVICE_IRQ" was also modified to check for both interrupts. The interrupt service routine called ARM_TIM_ISR was written which would write the value of the interrupt status to an address in memory with the label tim_int_flag, it would then clear the interrupt status. The same was done for the push button in a subroutine called KEY_ISR and placing the contents edgecpature register to a place in memory labelled PB_int_flag. The CONFIG_GIC was modified to configure both the ARM A9 Timer and Pushbutton interrupts by using their corresponding IDs. Furthermore, the IDLE code was written which was essentially the same as the main code for part 2.2. There was a small change as we would now be reading the address in memory corresponding to PB_int_flag and tim_int_flag.

Challenges faced and solution:

The hardest challenged I faced in this section was to correctly read the labels of PB_int_flag and tim_int_flag. I would read and then clear both. This would result in my stopwatch stopping just after starting. I realized this happened because I was clearing the PB_int_flag and thereby losing the value for which push button has been pressed and the stopwatch to only stay in the stop mode. However, this was fixed by not clearing the flag for the push button and read to see if any changes were made and which pushbuttons had been pressed. After this change, the code executed with the desired results.

Possible Improvements:

Like the previous parts, the HEX_write_ASM can be optimized. Furthermore, there was another realization that this part of the lab was also another form of polling and did not really use interrupts. It polled the addresses of the two flags instead of polling flags directly from the I/O devices. Changing it to execute with interrupts would result in time being saved and the CPU would perform other tasks instead of polling flags continuously.