ECSE 324 – Computer Organization (Fall 2018)
Lab 3 – Basic I/O, Timers and Interrupts
Group 12: Alain Daccache-260714615 ; Michel Abdel Nour-260725050

Lab 3 is mainly focused on basic I/O functions. By combining the use of the switches, the push-buttons, the hex displays, the timers and interrupts, we were able to run the ARM code for solving the different parts of this lab.

In this lab, we mainly understood how **modularity** renders efficient code that can be reused in many parts throughout the project. For instance, the write, clear, and flood subroutines of the HEX_display.s file were constantly called in the main method, from flooding the first two displays, to clearing for the stopwatch… We understood how bigger codes are structured, and the importance of keeping files organized (header files in a folder, assembly files in another). Also, we learnt the purpose of **enumerations** and **structs.**

Part 2.A – Basic I/O

Part 1 – Slider switches & LEDs

This part deals with switching on and off the LEDs on top of the corresponding slider switches. With the code provided, it was simple to understand how (1) I/O registers store the input of the user in specific addressable space in memory, and (2) to extract the user's input from the base address of the slider switches on to a processor register

The read LEDs ASM subroutine loads the value at the LEDs memory location into R0 and then branch to LR. The write LEDs ASM subroutine stores the value in R0 at the LEDs memory location, and then branch to LR.

The line *write_LEDs_ASM(read_slider_switches_ASM())* in the main function shows that the subroutine write_LEDs_ASM uses the read_slider_Swtiches_ASM subroutine as an argument. This subroutine loads the value at the LEDs memory location into R0. Then, we can use this value in the write_LEDs_ASM subroutine to store it at the LEDs memory location. As a result, the switch will turn on at the corresponding LED. It is important to note that the read subroutine uses the R0 the load the value read at the location of switches, then it can be used as an argument to the write function where R0 is stored in the memory location of LEDs...

In the part, the challenge was in understanding that the argument of the calling function in main.c is passed through register R0 that can be used in the assembly subroutine. The read functions load the values at some memory location into R0, while the write functions write the value stored at R0 into the memory location.

Part 2.B – Drivers for HEX Displays and push-buttons

Part 2.1 – HEX Displays:

In this section, we wrote subroutines that manipulate the 7-segment displays.

- The HEX_clear_ASM function turns off all of the segments of the specified display(s).
- The HEX_flood_ASM function turns on all of the segments of the specified display(s).

- The HEX_write_ASM function displays the specified hexadecimal digit on the specified display(s).

Since an **enumeration** was used, we could access the HEX displays one by one: HEX0 is at b1, HEX1 is at b10, HEX2 is at b100, HEX3 is at b1000 etc. so we notice the one-hot encoded pattern. This way, we can test the displays one by one to see which to clear or flood based on the user's input given in the first argument of the C method (R0 in assembly). After reaching the display in question:

- For clearing, we AND the Data register depending on the display to clear. For instance, for the first register (right-most), we AND with 0xFFFFFF00 to clear the rightmost 8 bits (remember, each hexa character represents 4 bits *nibble*), leaving all the rest the same (1 AND 0 gives 0, and 0 AND 0 gives 0, and 1 AND 1 gives 1)
- For flooding, we OR the Data register depending on the display to flood. For instance, to flood the third register (from the right), we OR with 0x00FF0000, this way ensuring we filled the 8 bits of the 3rd register.

Since we were given no assembly code, we didn't have a defined approach to follow. Initially we started writing the whole code by only storing bytes, which failed. In fact, since the STRB and LDRB instructions overwrite the rest of the word, they do not work to load and store individual bytes. Meanwhile, AND and OR instructions are byte operations that can separate bits from the rest of their binary string, thus helping read a specific bit from the register. This way, we can read the whole word then edit it and then store the whole word again in memory.

Our approach for writing is as follows:

- The input can be between 0-15, which then has to be decoded to convert those numbers to the correct pattern that displays a HEX character, 0-9,A-F.
- Since input is encoded using the ASCII format, we compare the contents of the second argument (value of store) i.e. R1 with each of the hexadecimal numbers (0-9,A-F) and offsetting with a magic number, 48, because it represents 1 in the ASCII table. Therefore, comparing to a 2 is equivalent to comparing to a 49 etc.
- After matching the argument with a number, we can turn off an on the corresponding bits to display the hexa number.
- Then we implement a similar approach of clear and flood to find the display in which we want to write the number

The challenges was to figure out that we have to offset by ASCII else we would be comparing to other characters, thus never matching with a number. Also, we needed to know how to display the hexadecimal number we wanted, until we realized that the locations of segments 6 to 0 in each seven-segment display on the DE1-SoC board is illustrated on the figure given in the guide. The rest was just manual labor.

<u>Part 2.2 – Push-buttons:</u>

In this section, we had to write subroutines to manipulate memory associated with the push keys on the FPGA board.
The read functions return the push-buttons that are pressed during the demo, the read_edgecap functions return the value of the edge-capture memory location.
The function clear_ edgecap clears the edge captured in memory location by setting it to zero.
Moreover, there are interrupt functions that will either disable or enable interrupts based on the need.

R1 was the register holding the PUSH_BUTTON values from memory and successfully implementing their different functionalities.
It is important to talk about the hot-one encoding of the push-buttons' functions: By relying on that technique, it allows to perform "binarization" of the push-buttons' different categorical functionalities in a clearer and more efficient manner, especially when it came to reviewing the code.

Challenges

There weren't that many challenges faced when writing that subroutine. By referring to DE1-SOC manual, we were able to get ideas from already written subroutines in order to implement the subroutines at hand efficiently and with less need and time allocated for debugging.

Part 2.2 – Putting it together:

As the title of this section refers to it, the task was to combine the previous subsections. The slider switches SW3-SW0 are used to represent the 4-bit binary value of a digits to display on the HEX displays HEX3-HEX0. The displays HEX4-HEX5 are flooded at all times in this section. The way it works is that by changing the value of the slider switches, we can represent 4-bit numbers of value ranging between 0-15, which in hexadecimal representation is equivalent to value ranging from 0 to F.
In order to represent these values, the low-level assembly code needed to fit the need for an ASCII offset. If we look at line number "48", we can determine the need for the offset, since the numbering 0 to 9 takes place starting line 48. (Figure 1)

In order to make this description clearer, it is important that we provide details about the how the functionalities of the push-buttons are embedded into the logic of the C code:

- By giving the address 0x200, which is 256 in decimal, that refers to the last slider switch, we are allowing the activation of that switch to clear HEX3-HEX0
- After setting a number with a specific value to display with the slider switches, the push buttons 1-4 allow us to write the value given by SW3-SW0 to the HEX0-HEX3 respectively. (button 1 responsible for the display in HEX0, button 2 for the display in HEX1, …).
- The functions used are read_PB_data_ASM, read_slider_switches_ASM(), HEX_write_ASM and HEX_clear_ASM.

Part 3 – Timers

Stopwatch

In this section of the lap, we had to make a stopwatch to be dynamically displayed and controlled on the FPGA board.
This section was also an introduction to C structures, data types that allow the grouping of several variables. These different variables were able to be accessed by a pointer, similarly to arrays.
The stopwatch was displayed on all HEX displays. The core of the problem was to make sure the different units of time (ie. milliseconds, seconds, minutes) are incremented as the stopwatch records the time that has passed. Knowing how to update the HEX displays accordingly was essentially one of the challenges we faced.

Furthermore, the push-buttons were used to serve the stop-time, start-time and reset functionalities of the stopwatch. Pushbutton 1 allowed to start the timer. Pushbutton 2 allowed to pause the stopwatch. Pushbutton 3 allowed to reset the timer

The timer program in assembly creates a basic timer, with subroutines of configure, clear, and read. All of these subroutines share a common approach, in which there is an outerloop counter examining what timer we should be looking at, and a short script identifying this counter's location in memory.
We'll initially look at the configure subroutine, since it served as a basis for the other two subroutines that followed. The configure subroutine takes in a pointer, since there are multiple inputs for this subroutine. As a result, we coded an initial part for the subroutine, that placed the pointer into a register, while also clearing everything ahead of it. This is also where we pushed the needed registers for the subroutine and created the outer counter. The next segment of this subroutine is indicating if we need to configure this timer and what timer we would be configuring. To do so, there is a compare statement, evaluating if the specific timer needs to be configured. If this is true, it is passed onto the next stage and evaluated for what location in memory that timer is located. The major issue for this section was figuring out how to find what timer needs to be configured, through trial and error we realized it was simply just a 1 bit shift to the right for each individual timer. The final segment of the configuration subroutine is storing all of the parameters into the appropriate timer memory position.
This was relatively simple compared to the previous part. The read subroutine borrowed from the first two parts of configuration, but required fewer registers to be pushed initially. Outside of the similar first two parts, the final part pulls the correct s-bit from memory and loads it into right-most bit of R0. The register can now be read outside of the subroutine.

The final subroutine for timers uses similar initial parts as the first two subroutines, then clears the values in selected timers. Again, this part of the code was fairly simple, since it borrowed extensively from the configuration code.

Part 4 – Interrupts

Interrupt based stopwatch

This section was very similar to the last section. In the previous section, a separate timer was polled to determine when to check whether or not a button had been pressed. In this section, a button being released fires an interrupt which starts, stops, or resets the timer.

The only functional difference between the two timers is that the polling timer acts as soon as button is pressed, while the interrupt timer acts once a button is released. Most of the code was provided to implement the interrupts, so all we had to write was the ISR.s file and the addition in main.c. In the ISR.s file, spaces in memory were set apart for flags for each of the timers, and simple interrupt service routines that handled timer selection were written.

In main.c, the structure was mostly copied from Section 3, requiring only the replacement of the timer poller with an if statement that watches for the interrupt flag to be raised.

There were few challenges faced with this subroutine because it was so similar to the previous section. After reading the DE1-SOC computer manual and installing the provided code for handling interrupts, we experienced few setbacks en route to a successful timer.
Since most of the code was provided, we don't have much in the way of potential areas of improvement.

One improvement that could be made in main.c is to allow the user to user one of the slider switches to switch between the polling stopwatch and the interrupt stopwatch. This would not only speed up the demo process, but it would also reuse more code in main.c, making that file shorter