ECSE 324 Lab 2: Basic I/O, Timers, and Interrupts

Karim Elgammal

McGill ID: 260920556

Task 1: Basic I/O

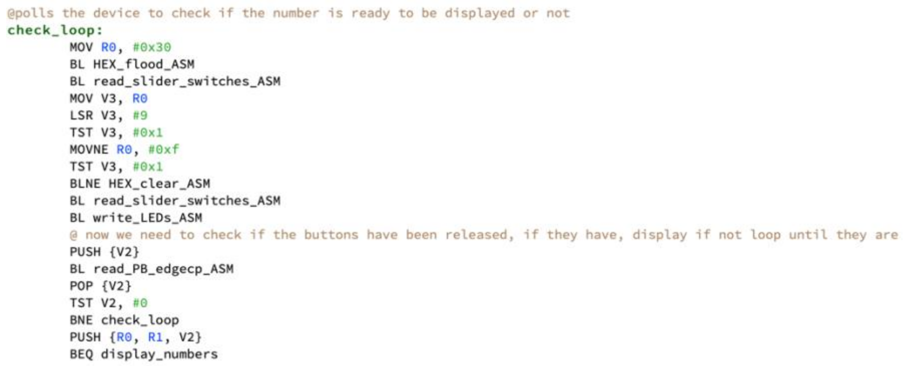
Problem Description:

Throughout this lab, an exploration and implementation of the basic I/O capabilities of the DE1-SoC computer was conducted. The first task within this lab pertains to implementing a program which allows a user to input a binary number using the switches present on the DE1-SoC board. Specifically, the program takes the input number from SW0 – SW3, where SW0 is the LSB of the binary number. This number is then displayed on to the hex displays, where each instance of display on the HEX display, corresponds to a push button of the same index. So, the program allows the user to choose which – one or many-HEX display presents the number, according to which push button has been pushed and then released. Additionally, since there are no pushbuttons which correspond to HEX4 and HEX5, we have kept these displays flooded during the execution of this program. Finally, SW9 is used to clear all the displays (excluding HEX4 and HEX5).

To realize this functionality, the program has been implemented using drivers, which are similar to subroutines that perform instructions or calculate indices/values. The task at hand asks to implement three drivers for the HEX display, which allow us to clear, flood and write to the six different HEX displays. Further, we are asked to implement seven different drivers for the pushbuttons which read or write to the three different device registers of the pushbuttons; that is, the Data Register, the Interruptmask Register and the Edgecapture Register of the Pushbutton device. Using these drivers, we have implemented the functionality of the program in a polling-style implementation of I/O devices.

Design Decisions:

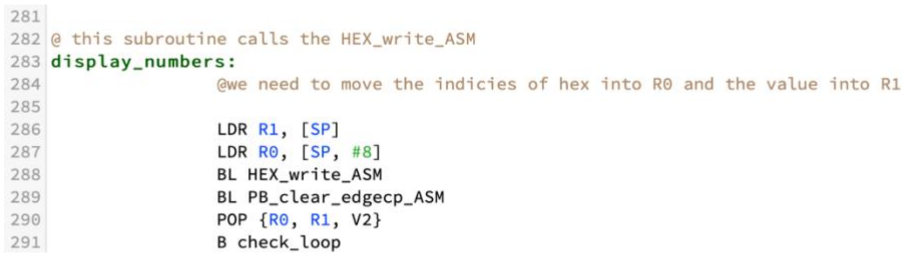
When it comes to handling I/O devices, there are two main styles that could be used to manage data transfer between the device and the processor: polling and interrupts. While polling allows us a less complex implementation, interrupts allow for a higher degree of efficiency, as the processor is not kept busy until the data is ready to be read/written. A more efficient approach would have been to use the drivers enable_PB_INT_ASM, and disable_PB_INT_ASM, to enable and disable interrupts from the push button respectively. This would have then asserted an interrupt service routine (ISR), which realizes the functions described above. The polling technique was used in order to actualize the task at hand due to its reduced level of implementation 'complexity'.



```
@polls the device to check if the number is ready to be displayed or not
check_loop:
        MOV R0, #0x30
        BL HEX_flood_ASM
        BL read_slider_switches_ASM
        MOV V3, R0
        LSR V3, #9
        TST V3, #0x1
        MOVNE R0, #0xf
        TST V3, #0x1
        BLNE HEX_clear_ASM
        BL read_slider_switches_ASM
        BL write_LEDs_ASM
        @ now we need to check if the buttons have been released, if they have, display if not loop until they are
        PUSH {V2}
        BL read_PB_edgecp_ASM
        POP {V2}
        TST V2, #0
        BNE check_loop
        PUSH {R0, R1, V2}
        BEQ display_numbers
```

Figure 1: Endless Loop for Polling Pushbuttons

Polling within the context of our task essentially consists of continuously checking the Edgecapture Register of the pushbutton device. While the Edgecapture Register is empty, the program continues to loop the subroutine check_loop (Fig 1) and continues to read the value of the number we want to represent from the SW's data register. Once the Edgecapture Register is not empty, we use the driver discussed previously, which loads the value stored within the Edgecapture Register into V2. Once we have this value, a helper subroutine display_numbers (Fig 2) is called which drives the value we received from the Edgecapture Register to be decoded according to the one-hot encoding that the HEX_write_ASM subroutine uses. Since both the pushbuttons and HEX displays are encoded in the same way, we simply need to relay the decoded value and our value represented by SW0-SW3 into HEX_write_ASM for the subroutine to write the values onto the display. At each instance that display_numbers is called, the Edgecapture Register is reset in order to allow new indexes to be specified by the user.

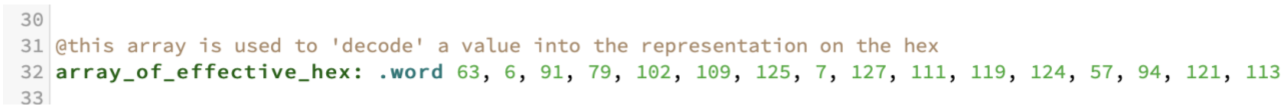


```
@ this subroutine calls the HEX_write_ASM
display_numbers:
                @we need to move the indicies of hex into R0 and the value into R1

                LDR R1, [SP]
                LDR R0, [SP, #8]
                BL HEX_write_ASM
                BL PB_clear_edgecp_ASM
                POP {R0, R1, V2}
                B check_loop
```

Figure 2: The display_numbers subroutine.

HEX_write_ASM is contingent on accessing a pre-defined (hard-coded) array of values which correspond to the 7-bit representation of the HEX display (Fig 3). So, for example, a 1 on the HEX display is equivalent to $(0000110)_2$, which is represented in our array_of_effective_hex as $(6)_{10}$. The subroutine then receives the index(es) along with the number to display. The number to display is kept track of by way of the indexes of the array which we traverse. The program then checks each HEX display for a match with the index value we fed it. Once it finds a match, the subroutine displays the number (by storing the corresponding value of array_of_effective_hex into the data register of the corresponding HEX display) and continues to the next HEX display. This is done by traversing through the memory locations of HEX0 – HEX3 and using the TST instruction to compare each bit of the index(es) we fed and the index of the HEX display under examination.



```
@this array is used to 'decode' a value into the representation on the hex
array_of_effective_hex: .word 63, 6, 91, 79, 102, 109, 125, 7, 127, 111, 119, 124, 57, 94, 121, 113
```

Figure 3: Array of values that correspond to numbers displayed on HEX display

Optimization:

Throughout this implementation, we can recognize certain instructions or techniques which can be modified to optimize the performance, time complexity or efficiency of the program in terms of the processor's tasks. The first optimization/improvement that is suggested, is implementing the program using an interrupt technique rather than the polling technique. Owing to the fact that when the processor is constantly polling the device, it is in a 'busy' state and further uses more registers than needed to perform the task at hand. An interrupt technique allows us to instead wait for the device to request an interrupt, which proves to be more efficient.

Secondly, there are some instances within our code, where rather than using relative addressing syntaxes, we directly use counter registers to hold the counter values. This increases the number of registers which are being used and in turn, decreases the overall performance of our program.

Task 2: Timers

Problem Description:

The second task within this lab asks for an implementation of a simple stopwatch which counts in increments of 100ms. Further, the task at hand asks for functionalities of stopping, resetting, and starting the timer using the pushbutton device on the DE1-SoC. Further, the hex displays are used to display the state of the timer in terms of milliseconds, seconds, minutes, and hours. We are asked within this task to follow a polling-based implementation; that is implement the program in a design that allows for the pushbutton devices to be constantly polled until it is ready. More specifically, we are polling the Edgecapture Register to check if it is empty or not; if it is empty, then the device is not ready, and no actions have/should be taken. When the Edgecapture Register is not empty the device is ready to be read, through the very same register.

Task 2 utilizes the ARM A9 private timer, which is a down timer device within the DE1-SoC board. This timer must also be polled in a constant loop to verify if the device is ready to receive data, more specifically, the F bit within the ARM A9 private timer interrupt status register is polled until it is active to receive data.

Task 3: Interrupts

Problem Description:

Finally, task 3 asks us to implement the same stopwatch described in task 2, however, within the context of an interrupt-based implementation. That is, we enable the interrupt bits from the Interruptmask Register, which allow the program to receive requests from that device. Further, we must configure the interrupts from the pushbuttons with the ARM A9 private timer. Once an interrupt is received, the program enters the ISR which reacts to that interrupt. Unlike the polling method, when using interrupts, the processor is not kept busy while we wait for a device to enter a ready state. This implementation of the stopwatch proves to be more efficient in terms of complexity, reusability, and efficiency. Hence, compared to task 2, the interrupt-based implementation supplies us with the optimized program implementation.