Lab 2 Kristian Myzeqari 261 037 094

The goal of the lab:

The main goal of Lab 2 was to familiarize us with using different input/output functionalities of the DE1-SoC computer using ARM assembly programming. The main functionalities covered were the slider switches, pushbuttons, LEDs, hex displays, and timers. An in-depth understanding of the private timer was necessary to successfully complete the lab. The three increasingly complex tasks and their subtasks assigned for this lab varied from simple slider drivers to a stopwatch program making use of interrupts.

Task 1:

For task 1, we were assigned to write an assembly program that allows us to write down different hex values in four of the six designated 7-segment displays. As assigned, the written assembly code makes use of the previously implemented subroutines to function. The task requires us to use the switch sliders and the provided code that makes them work to select the value to be displayed. The display drivers call the base memory address 0xff200020 to access the first four displays, and the address 0xff200030 to write to HEX 4 and HEX 5.

```
HEX_flood_ASM:
    push    {V1-V3}
    ldr v2, =ADDR_FIRST4
    ldr v3, =ADDR_LAST2
    MOV v1, #0xff

    flood5:
    CMP A1, #32
    BLT flood4
    SUB A1, A1, #32
    strb v1, [v3, #1]
```

Figure 1: sample of flood subroutine

The functionality of the flood and clear subroutines is very simple. The program goes through every single display and stores the hex value 0xff (flood) or 0x00 (clear) into the addresses in the memory associated with each display. By storing the value 0xff, we light up all the segments on the display and show the number "8". By storing 0x00, we turn off any segment that was previously lit.

```
48 HEX_write_ASM:
49     push    {V1-V3}
50     LDR a3, =arr
51     ldr v2, =ADDR_FIRST4
52     ldr v3, =ADDR_LAST2
53     LDR v1, [A3, A2, LSL #2]
54
55     write4:
56     CMP A1, #8
57     BLT write3
58     SUB A1, A1, #8
59     strb v1, [v2, #3]
```

Figure 2: write subroutine featuring digit selection logic (line 53)

The write subroutine uses a similar logic as the flood and clear subroutines. The program goes through every single display by checking if the value used as input is larger than the hot-encoded value of the display. It then stores the specified value in the memory address of the selected display. To easily assign values to the hex displays, an array containing the hex values of all 16 possible digits. The array is stored in memory, and all the different elements are word addressable from the memory with an offset equivalent to the numerical value of the desired hex digit multiplied by 2. In the example above, A2 represents the value of the display to be shown.

The final step for this task was to read input from the push buttons to select the display to be used. In order to complete the task, I made use of the edge capture register to only get a reading from the buttons if the button is pressed and released.

```
read_PB_edgecp_ASM:
    LDR R3, =PUSHBUTTONS
    ADD R3, R3, #12 // pushbutton edbge register is 12 bits away
    LDR R0, [R3]
    BX LR

PB_edgecp_is_pressed_ASM:
    MOV R4, #0
    PUSH {LR}
    BL read_PB_edgecp_ASM
    POP {LR}
    TST R0, R2
    BEQ returnE
    MOV R4, #1
    returnE:
    BX LR

PB_clear_edgecp_ASM:
    PUSH {v1}
    LDR R3, =PUSHBUTTONS
    ADD R3, R3, #12
    LDR R0, [R3]
    STR R0, [R3]
    POP {v1}
    BX LR
```

Figure 3: edge capture register subroutines

It was necessary to read the edge capture register and clear it to avoid false positives when running the program.

There are many possible optimizations for this task in the lab. The most important one that I noticed was related to the edge capture functionalities of the pushbuttons. If the button is pressed and released fast enough, the release might be done before or after the instructions check for the state of the buttons, not showing that the button was pushed and released. The implementation could have been done in a way that checks for the button press/release faster.

Task 2:

The simple timer task's base is made of the private timer included in the DE1-SoC computer board. At a frequency of 200MHz, we had the task of setting a constant of 20 000 000 so that the timer can increment every tenth of a second (100 milliseconds). Once the timer is set up, it was important to implement the other subroutines in task 2.1: ARM_TIM_read_INT_ASM and ARM_TIM_clear_INT_ASM. These two subroutines take care of reading the "F" bit at address 0xFFC0800C and clearing the bit to bring it back to its original state.

```
ARM_TIM_read_INT_ASM:
    PUSH    {V1}
    LDR     V1, =TIMER
    ADD     V1, V1, #12
    LDR     A1, [V1]
    POP     {V1}
    BX LR

ARM_TIM_clear_INT_ASM:
    PUSH    {V1}
    LDR     V1, =TIMER
    ADD     V1, V1, #12
    LDR     A1, [V1]
    STR     A1, [V1]
    POP     {V1}
    BX LR
```

Figure 4: read and clear subroutines for the edge capture register

To complete this task, it was necessary to write the main entry for the program in a loop to be able to poll any needed input. A general for loop that checked for when the timer is stopped, started, or reset. Once the program reads a pushbutton corresponding to the according functionality (stop, start, reset). The program recognizes the selected functionality through the polling system, which parses through register A1 returned by the "ARM_TIM_read_INT_ASM" subroutines. For example,

in a situation where the program reads that the button associated with a stop task calls the subroutine “timer_stop”. The same applies to all pushbuttons, each with their respective subroutine calls to manage the timing elements of the clock.

```
41  // CHECK FOR TIMER STOP
42  TST     V8, #0x02
43  BLNE    timer_stop
```

Figure 5: check for when the stop button is pressed, next, call timer_stop

The part of the task that had to do with displaying the stopwatch values on the 7-segment display was unfortunately incomplete in this lab. Although this section was not complete, I set up the “ARM_TIM_write_INT_ASM” and “ARM_TIM_clear_INT_ASM” subroutines from task 1 to implement the first as the subroutine dealing with all changes to the clock. The second subroutine setting is the subroutine in charge of running the reset button. To keep the display up to speed with the program, the “write” call for the displays was called at every iteration of the loop.

There is not much improvement that I managed to think of while I was working on this task, having not finished it, I worked much more on writing the program itself rather than focusing on the optimizations I could have brought. I did notice that I could have optimized

Task 3:

For task 3, I did not manage to work on the task, but I took the time to try and understand the necessary tasks to successfully complete it.

From the provided assembly code examples, I understand that the task is very similar to task 2, but to include the interrupts, it is necessary to implement a few additional subroutines. Firstly, “SERVICE_IRQ” will be modified to allow for the IRQ to check for two interrupts at the same time: the pushbuttons and the A9 private timer interrupt service routines. Since the program is not using a polling mechanism to detect input/output data, it is necessary for the interrupts to verify that any possible signal is detected to keep the right functionality.

Next, the “KEY_ISR” subroutine must be modified to write the content of pushbuttons edge capture register into the PB_int_flag memory and to clear the interrupts. This subroutine uses the base addresses of the pushbuttons to gain access to the edge capture register. It goes through every key to verify which one has been pressed.

Finally, “CONFIG_GIC” is the subroutine that will be modified from the sample code to allow us to configure the pushbuttons and the A9 private timer interrupt service routines. Only with the configuration will we be able to gain access to the features allowing us to complete this task.