ECSE 324 Computer Organization

Lab 2

Kua Chen 260856888

Introduction

In this lab, we explored I/O devices of the DE1-SoC computer including the slider switches, pushbuttons, LEDs, 7-Segment (HEX) displays and timers. We applied polling method for most assembly programs and applied interrupt (ISR) method for the final stopwatch application.

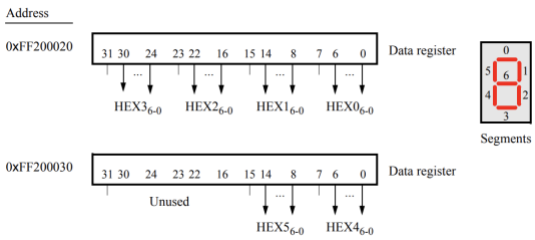
Part 1.1

This part is an introduction for us to get started to interact with I/O devices. I wrote a program to read slider switches and turn the LEDs on if corresponding switches were on. The memory address associated with switches is 0xFF200040. For example, if slider switch 0 is on, the bit 0 in memory address 0xFF200040 will be 1. If slider switch 0 is off, then bit 0 in this memory address will be 0. There are 10 switches from indices 0 to 9 and there are also 10 LEDs from indices 0 to 9. The memory address associated with LEDs is 0xFF2000000. For example, in order to turn on $LEDR_0$, bit 0 in memory address 0xFF2000000 need to be set to 1 and vice versa.

Since we had given two subroutines interacting with slider switches and LEDs, we only needed to write an endless loop in the main body of our program and constantly accessed (LDR) the content of memory address 0xFF200040 (slider switch) and wrote (STR) this content into memory address 0xFF2000000 (LED). This part was very straightforward, but we would build more complex programs on this basis.

Part 1.2

In the part, we should first understand how HEX displays and pushbuttons work and then combined these I/O devices to implement the required program. The memory addresses associated with HEX displays are 0xFF200020 and 0xFF200030. There are 7 segments in one HEX display, and we can display a digit by configuring these 7 segments correctly. For example, digit 1 can be displayed on HEX0 by turning on segment 1 and segment 2, then we need to write 0b000110 for $HEX0_{6\text{-}0}$.



There are 15 digits (0 to F) we want to display so there are 15 binary numbers to represent HEX configurations. If we want to display a number on HEX1, we could

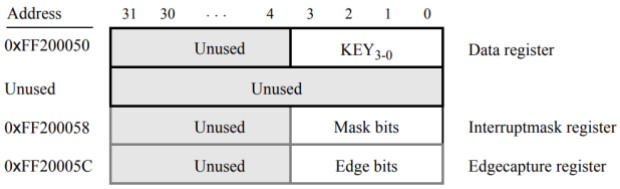
simply do a left shift operation to the binary number to make sure it could fit in bit 8 to bit 14. We store these 15 binary numbers in an array *HEX_CHARA* in order to access them conveniently.

The first challenge I encountered was that I wanted to update HEX displays in a beautiful and concise for-loop manner, just like what we did for programs in lab 1. This idea seemed feasible at beginning, but I soon realized it was very hard. First problem was that 6 HEXs are divided into two registers which increased the overall complexity. Second problem was that in order to update one specific HEX display, we should only change the bits that were corresponding to this HEX and keep the other bits unchanged. For example, to update *HEX2,* we should change bit 16 to bit 22. Unfortunately, I could not come up with a direct instruction that can write a sequence of bits into a register starting from a chosen position. Therefore, I decided to adopt an exhaustive method instead of a for-loop.

An exhaustive method means I will write subroutines for all HEX displays. This is a very important point. Using exhaustive methods to interact with I/O devices can reduce the complexity and increase the code readability. Back to our case:
HEX_0: If R0=XXX1, execute instructions then B HEX_1. If bit 0 !=1, B HEX_1.
HEX_1: If R0=XX1X, execute instructions then B HEX_2. If bit 1 !=1, B HEX_2.
HEX_2: And so on
…
HEX_5: And so on.
It became much more straightforward to just focus on single scenario. Take *HEX_2* as an example. Firstly, we used *AND R5, R0, #4, CMP R5, #4* instructions to check whether the bit 3 in *R0* was 1 or not because we used one-hot encoding for *R0*. If *R5=4,* then we cleared the bit 16 to bit 22 in *R3* (value of address 0xFF200020) by *AND R3, R3, #0XFF00FFFF* instruction. We actually cleared bit 16 to bit 23 with this instruction but bit 23 was not used so this would not create any problem. *R1* was the binary number representing the HEX configuration and we wanted to write it to bit 16 to bit 22 in *R3*. The reason why *R1* is only left shifted by 8 here was that it had already been left shift by 8 during *HEX_1_wirte.* So, *R1* had been left shifted by 16 bits compared to its original value. Then adding *R1* to *R3* would fit in the bit 16 to bit 22 in *R3*. Finally, we stored *R3* back to the memory 0xFF200020 to update HEX2. Then we moved to *HEX_3* to check if we need to update HEX3. This procedure was repeated from *HEX_0* to *HEX_5*.

```
HEX_2_write:
        LSL R1, R1, #8 //shift to the correct hex position
        AND R5, R0, #4
        CMP R5,#4
        BNE HEX_3_write
        AND R3, R3, #0XFF00FFFF //CLEAR
        ADD R3, R3, R1
        STR R3, [R2]
        B HEX_3_write
```

Then we moved to the pushbutton part. When a pushbutton was pressed and released, its corresponding Edge bit became 1. We also used an exhaustive method to check whether any Edge bit was 1 or not. In our program, pushbuttons 0 to 3 were corresponding to HEX0 to 3. Whenever an edge bit high was detected, corresponding HEX display would be updated. More importantly, after updating the HEX display, the edge bit must be cleared. I also encountered a small bug here. For real DE1-SoC computer, writing any value to Edgecapture register would reset it back to 0 but for this simulator, we must write 0xF into Edgecapture register in order to clear it.



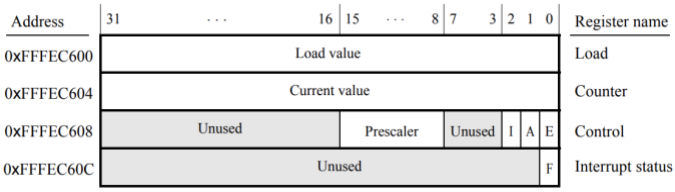
Therefore, our main body was also a loop:

Loop:

BL CHECK_SW9 // turn off all hex
BL CHECK_PB_0 // write HEX_0
BL CHECK_PB_1 // write HEX_1
BL CHECK_PB_2 // write HEX_2
BL CHECK_PB_3 // write HEX_3
B Loop

Part 2.1

In this part, we would interact with A9 timer and used HEXs to test the functionality of this timer. First thing to do was configuring the timer as required. This included *LDR 200M* into Load register and set bit I, bit A, bit E to 1 in Control register. When Counter register reached 0, Interrupt status (bit F) became high.



In the main body of our program, we used a for-loop to constantly check whether bit F was 1 or not. If it was 1, we increased the digit displayed on HEX0, then cleared bit F as well. We also stored HEX configuration binary representations in an array *HEX_CHARA*. To increase the digit on display, we could just access the next element in the array *HEX_CHARA*. If the current digit on display was *F,* then we added 1 to *F* and subtracted 16 from it so that the digit after *F* would be 0. More fundamentally, we went back to the start of array *HEX_CHARA* and accessed the first element 0. This became a circle (0 to F to 0 to F …) and looped forever.

Part 2.2

In this part, we built a stopwatch application which could start (by PB0), could be stopped (by PB1), and reset (by PB2) with min scale unit be 10 milliseconds. In order to meet min scale unit, we should *LDR 2M* into Load register so that bit F would be high 10 milliseconds after the timer starts counting. There are 6 HEXs. HEX0 has a unit of 10 milliseconds, HEX1 has a unit of 0.1 second, HEX2 has a unit of 1 second, HEX3 has a unit of 10 second, HEX4 has a unit of minute, and HEX5 has a unit of 10 minutes. Therefore, we need 6 digits for displaying time, and I stored these 6 digits in the stack separately. When HEX0 is 9, and bit F is high. This means that there is a carry from HEX0 and HEX1 should be increased by 1. Same thing happens when HEX1 has a carry. If there is no carry from HEX0, then only updated HEX0 and store the new value back to stack because if there was not carry from HEX0, then HEX1 would not increase and definitely have no carry and so on.

My main body is as shown as following:

```
mainLoop:
    check_PB_0 // if edge bit 0 is 1, go to Loop
    check_PB_2 // if edge bit 2 is 1, Branch RESET,
    B mainLoop // if nothing happens
Loop:
    check_PB_2      // if edge bit 2 is 1, B RESET, RESET
    check_PB_1      // if edge bit 1 is 1, go to mainLoop
    INCREASE  //update HEX
    B Loop
RESET: // reset 6 numbers representing the time
    B Loop
```

The program will enter *mainLoop* first. If PB0 or PB2 is pressed and released, then the program will enter *Loop.* Within the *Loop*, if ignore PB2 (reset) and PB1 (stop) first, the only function is *INCREASE* and this subroutine will update the HEX displays if bit F in 1. If PB2 is pressed and released, the program will jump to *RESET* which clear the 6 numbers in stack back to 0 as well as HEX displays and then branch back to *Loop.* If PB1 is pressed and released, the program will jump to *mainLoop* and *INCREASE* will not be executed. This is like a state machine, where *mainLoop* is the state of watch: stop, and *Loop* is the state of watch: ticking.

However, I have to acknowledge that the timer has never been stopped. In fact, I stop updating HEX displays by entering *mainLoop*. This perhaps is not the best design choice and the better one is to stop the timer by setting bit E to 0 in control register. There will be small timing error in my design. For example, I have stopped my stopwatch application but the timer inside is still ticking. Then I reset the stopwatch, but the bit F has already by high for a long time. Therefore, the application will instantly increment 1 on HEX0 without waiting for 10 milliseconds. In the next part, I will improve my design and stop the stopwatch by truly stopping the timer.

Part 3

In this part, we built a stopwatch which could start, stop and reset as the one in part 2. The stopwatch in part 2 adopted the polling method, whereas this one adopted the interrupt method. I designed the program workflow as following:

if interrupt from PB0, start
if interrupt from PB1, stop
if interrupt from PB2, reset
if interrupt from timer, increase HEX display

Most importantly, all the main operations take place within the ISR. As before, we should set various configurations first. We can take advantage of the provided code and set up GIC configuration for timer as following.

```
    MOV R0, #29                         //TIMER ID = 29
MOV R1, #1
BL CONFIG_INTERRUPT
```

For this time, the bit E in the timer will be set to 0 initially. If there is an interrupt from PB0 or PB2, bit E will become 1. If there is an interrupt from PB1, bit E will become 0 by *ARM_TIM_STOP*. The bit E can stop the timer if it is 0 and start the timer if it is 1. Since the timer is stopped, then it will not generate any interrupt, then there will be no change on HEXs. This way to stop the stopwatch is different from the one in part 2.2 and this way is more accurate and more standard way.

```
CHECK_KEY0:
    MOV R3, #0x1
    ANDS R3, R3, R1        // check for KEY0
    BEQ CHECK_KEY1
        //write edge regsiter to  PB_int_flag
        STR R3, [R2]
        PUSH {LR}
        BL PB_clear_edgecp_ASM
        POP {LR}
        PUSH {LR}
        BL ARM_TIM_START
        POP {LR}
    B END_KEY_ISR
```

```
CHECK_KEY1:
    MOV R3, #0x2
    ANDS R3, R3, R1
    BEQ CHECK_KEY2
    STR R3, [R2]
    PUSH {LR}
        BL PB_clear_edgecp_ASM
        POP {LR}
        PUSH {LR}
        BL ARM_TIM_STOP                //stop the timer
        POP {LR}
    B END_KEY_ISR
```

Within the interrupt from A9 timer, we clear the interrupt bit and increase the digits on HEX displays. As I mentioned before, we need 6 digits to store the value of time and I put them in the stack in part2.2. This was a challenge to deal with stack within the ISR because I remembered that during an interrupt, the processor has another set of registers different from the registers for the main program. To solve this problem, I took a workaround and decided to store these 6 digits in the array *Time_Data* instead of the stack.

```
ARM_TIM_ISR:
        LDR R1, =INTERRUPT
        MOV R0, #0X1
        STR R0, [R1]    //clear interrupt
        LDR R1, =tim_int_flag
        STR R0, [R1]
        push {LR}
        bl INCREASE
        pop {lr}
        BX LR
```