ECE9047: Laboratory 2 Report

Objective

This lab is about programming an ARM®Cortex-A9 on a DE1 SoC board in the ARMv7 assembly language. The goal of this lab is to implement the following system: Create a three-digit decimal counter that accurately counts 1 s intervals. The count should be displayed as a decimal number on the 7-segment display. An interrupt-enabled push button should be used to reverse the direction of the counter. The count should roll over to zero after reaching 999 (if counting up), or vice versa if counting down. The program should run in a continuous loop. It also required to use interrupts to handle the push button input. It may continue to use interrupts with the timer to handle the 1 s counter, or poll the timer for the 1 s interval. It must use the timer to count for 1 s.

Explanation

The implementation consists of multiple part to achieve the desired functionality. First of all, there should be a lookup table mapping every number displayed on 7-segment display. Since there are 3 digits to display, it is advisable to split the number into 3 parts. In this lab, I first extract the units place by subtracting the index by 10 using a loop. Similar methods are applied to extract the tens place and hundreds place, and then shift, add them up, and display it. To realize the push button that can change the counting direction, a subroutine is involved for this. The push button input is handled via an interrupt service routine (ISR), which toggles the counting direction when pressed. A timer is configured to generate an interrupt every 1 second, ensuring accurate timekeeping without CPU polling. The main program loop continuously updates the counter, displays the value on the 7-segment display, and checks for button interrupts to change direction. By utilizing interrupts instead of polling, the program achieves better efficiency, ensuring minimal CPU resource usage.

Here is the rough flowchart:

Start → Initialize Timer → Initialize Display → Enable Interrupts→ Check Button IRQ → Reverse Direction? →Wait for Timer IRQ→ Update Counter Display on 7-Segment → Repeat

Description

In the practical implementation, I met some difficulties, but finally solved them all. Since I used timer in lab1, I tried to use timer in this lab based on the former code. I soon saw the problem. In lab 1 the process has a predetermined time length, but now the total time is uncertain because of the push button. So I have to change the design of timer. After many attempts, I modified the timer implementation by replacing the original polling-based timeout detection with a timeout flag, which allow the program to trigger 1 second interval no matter what the index is and what the counting direction. Not only is the timeout flag method more versatile, it's also more concise in coding. Although I have already known how to reset the index when the index reaches maximum to restart the process, I was not sure when it is counting down to 0, how to make it continue with 999 at first. Later I found that it can be easily realized by setting R4(the counting index) to 999 when it is less than 0, and the counting will go on since R10(counting direction) is -1. As for the push button part, the way of push detection puzzles me. Theoretically it detects the address of then button whether it is 1 or not but when I test my code, it cannot run properly. Later I found that only push and release the button after the interruption will be detected, or the direction will not change.

Cost-Benefit Analysis

I spend some time in redesigning the timer, and upgrade it. It indicates that if I spend more time in lab 1, I will design a wider applicable timer and reduce the workload in this lab. During designing push button, it is recommended to add debouncing logic to simulate the real situation, but I omit this part since this is just a simulation and no signal jitter will occur. In comparison, debouncing adds minor code complexity and may slightly delay input response, but it can prevent unintended multiple triggers, improves user experience, and ensures system stability. In addition, the way I split the number is considered as low efficiency, it is recommended to use udiv to realize the same effect. However, it seems that this hardware does not support this instruction. Otherwise, the code will be more concise.

Code

```
.equ SSD_BASE, 0xFF200020      @ Address of the seven-segment display

.equ BUTTON_BASE, 0xFF200050 @ Address of the push buttons

.equ TIMER_BASE, 0xFF202000    @ Address of the timer


.syntax unified

.cpu cortex-a9

.text

.global _start
```

```
_start:
    LDR SP, =0x8000          @ Set stack pointer
    MOV R4, #0                @ Initialize the counter to 000
    MOV R10, #1               @ Counting direction (1 = increment, -1 = decrement)

MAIN_LOOP:
    PUSH {R4, LR}
    BL DISPLAY_COUNT          @ Display the value of `R4`
    BL CHECK_BUTTON           @ Handle button press (detect + toggle direction)
    BL WAIT_1_SECOND          @ Wait for 1 second
    POP {R4, LR}


    CMP R10, #1
    BEQ COUNT_UP
    SUB R4, R4, #1           @ Decrement counter
    CMP R4, #0
    BGE MAIN_LOOP
    MOV R4, #999              @ If < 0, reset to 999
    B MAIN_LOOP

COUNT_UP:
    ADD R4, R4, #1           @ Increment counter
    CMP R4, #1000
    BLT MAIN_LOOP
    MOV R4, #0                @ If 999 is reached, reset to 0
    B MAIN_LOOP
```

```
CHECK_BUTTON:
    PUSH {R0, R1, R2, LR}
    LDR R0, =BUTTON_BASE
    LDR R1, [R0]            @ Read button state
    TST R1, #1              @ Check if button 0 is pressed
    BEQ EXIT_CHECK

    CMP R10, #1
    MOVEQ R10, #-1          @ If `R10 = 1`, change to `-1`
    MOVNE R10, #1           @ If `R10 = -1`, change to `1`

    STR R1, [R0, #8]        @ Clear button state register

    LDR R2, =100000
WAIT_RELEASE:
    LDR R1, [R0]            @ Read button state
    TST R1, #1
    BEQ EXIT_CHECK

    SUBS R2, R2, #1
    CMP R2, #0
    BLE EXIT_CHECK
    B WAIT_RELEASE

EXIT_CHECK:
    POP {R0, R1, R2, PC}
```

```
DISPLAY_COUNT:
    PUSH {R0, R1, R2, R3, R5, R6, R7, R8, LR}
    MOV R6, R4
    LDR R7, =SSD_BASE
    LDR R8, =SSD_LOOKUP

    MOV R5, R6
    MOV R1, #0
DIV_LOOP1:
    CMP R5, #10
    BLT END_DIV1
    SUB R5, R5, #10
    ADD R1, R1, #1
    B DIV_LOOP1
END_DIV1:
    ADD R8, R8, R5, LSL #2
    LDR R5, [R8]

    MOV R6, R4
    MOV R1, #0
DIV_LOOP2:
    CMP R6, #10
    BLT END_DIV2
    SUB R6, R6, #10
    ADD R1, R1, #1
    B DIV_LOOP2
END_DIV2:
    MOV R6, R1
    MOV R1, #0
DIV_LOOP3:
    CMP R6, #10
    BLT END_DIV3
    SUB R6, R6, #10
    ADD R1, R1, #1
    B DIV_LOOP3
END_DIV3:
    LDR R8, =SSD_LOOKUP
    ADD R8, R8, R6, LSL #2
    LDR R6, [R8]
    LSL R6, R6, #8
    ORR R5, R5, R6

    MOV R0, R4
    MOV R1, #0
DIV_LOOP4:
    CMP R0, #100
    BLT END_DIV4
    SUB R0, R0, #100
    ADD R1, R1, #1
    B DIV_LOOP4
END_DIV4:
    LDR R8, =SSD_LOOKUP
    ADD R8, R8, R1, LSL #2
    LDR R0, [R8]
    LSL R0, R0, #16
    ORR R5, R5, R0

    STR R5, [R7]
    POP {R0, R1, R2, R3, R5, R6, R7, R8, PC}
    BX LR

WAIT_1_SECOND:
    PUSH {R0, R1, R2, LR}

    LDR R0, =TIMER_BASE
    LDR R1, =100000000      @ 1 second
    STR R1, [R0, #8]        @ Lower 16 bits
    MOV R2, R1, LSR #16
    STR R2, [R0, #12]       @ Upper 16 bits
    MOV R1, #6
    STR R1, [R0, #4]

WAIT_TIMER_LOOP:
    LDR R1, [R0]            @ Read status register
    TST R1, #1              @ Check timeout flag
    BEQ WAIT_TIMER_LOOP     @ If not timed out, continue waiting

    STR R1, [R0]            @ Clear timeout flag

    POP {R0, R1, R2, PC}
```

```
.section .data

SSD_LOOKUP:

        .word 0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F, 0x6F
```