ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 351E
MICROCOMPUTER LABORATORY EXPERIMENT REPORT

EXPERIMENT NO	:	5
EXPERIMENT DATE	:	13.12.2024
LAB SESSION	:	FRIDAY - 14.30
GROUP NO	:	G7

GROUP MEMBERS:

150210087 : HİLAL KARTAL

150210100 : SELİN YILMAZ

150210067 : ALPER DAŞGIN

FALL 2024

Contents

1 INTRODUCTION [10 points]

In the first section, we wrote a program that counts from 0 to 9 in 10 seconds while monitoring a seven-segment display. To do this, GPIO ports had to be initialised, a delay subroutine with a one-second interval had to be created, and integers had to be converted into the proper bit patterns for the display using a mapping table. We got a deeper understanding of bit manipulation and GPIO settings with this challenge, which is important when working with peripheral devices.

In the second section interrupt subroutines were added to allow for dynamic counting mode switches. The application switched between counting even (0, 2, . . . , 8) and odd (1, 3, . . . , 9) numbers by using a maskable interrupt on one of the GPIO Port pins. External control over counting action was made possible by the implementation's requirement to configure the interrupt vector and subroutine for GPIO Port 2's sixth bit. This exercise focused on how interruptions can be used effectively for real-time control in embedded systems.

To sum up, this experiment gave insight with interrupt-based program control, 7-segment display driving, and GPIO port initialisation. With regard to modular design, effective control of memory, and managing asynchronous events.

2 MATERIALS AND METHODS [40 points]

2.1 MATERIALS

- MSP430
- Selin's Laptop

2.2 METHODS

2.2.1 FIRST PART

The purpose of this part is to light correct leds of 7-segment display (Fig. 1) to make it a counter. It counts as "0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9" and then starts again.

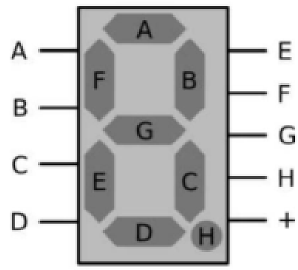

Figure 1: 7 - segment display Input Layout

First, we decided which inputs have to be set high or low in order to represent those integers in the table 1 below.

Integer	**H**	**G**	**F**	**E**	**D**	**C**	**B**	**A**
0	0	0	1	1	1	1	1	1
1	0	0	0	0	0	1	1	0
2	0	1	0	1	1	0	1	1
3	0	1	0	0	1	1	1	1
4	0	1	1	0	0	1	1	0
5	0	1	1	0	1	1	0	1
6	0	1	1	1	1	1	0	1
7	0	0	0	0	0	1	1	1
8	0	1	1	1	1	1	1	1
9	0	1	1	0	1	1	1	1

Table 1: Inputs for 7-segment display

We defined an array above `.text` to put inputs.

```
            .data
array       .word   00111111b, 00000110b, 01011011b, 01001111b, 01100110b, 01101101b, 01111101b, 0000111b, 01111111b, 01101111b
lastElement         ;   0   1   2   3   4   5   6   7   8   9
```

Listing 1: Part 1 - Input

We defined it as *word*, so to move the next element we need to increment two times.

Then, we made the initializations for the registers and GPIO we will use.

```
1 Setup   mov.b #00000000b, &P2SEL
2         mov.b #11111111b, &P1DIR
3         mov.b #00001000b, &P2DIR
4         mov.b #00000000b, &P1OUT
5         mov.w #array, r13
6         mov.b #00000000b, r10
```

Listing 2: Part 1 - Setup

Line 1: Make all of the select bits of P2. If it is not done, the computer may randomly assign some values.

Line 2: Select A, B, C, D, E, F, G, and H as output.

Line 3: Choose only the P2.3 bit as output.

Line 4: Initialize the outputs of P1 as low.

Line 5: Move the starting address of the array (we defined before) as word to R13.

Line 6: Write 0 to R10 as it will be used as counter later.

Then, we wrote the Mainloop which calls *delay* subroutine and traverses through input array.

```
1 Mainloop    mov.b 0(r13), &P1OUT
2             call  #Delay
3             inc.w r13
4             inc.w r13
5             inc.b r10
6             cmp   #10, r10
7             jl    Mainloop
8             jmp   Setup
```

Listing 3: Part 1 - Mainloop

Line 1: Shows what is written in memory where R13 (array) points to P1OUT.

Line 2: Call the *Delay* subroutine and push the address of the next instruction to stack. This subroutine creates a 1-second long delay for the output to be observed.

Line 3, 4: After returning from *Delay* increment R13 two times for R13 to point to the next element of the array.

Line 5: Increment the counter R10.

Line 6, 7, 8: Check if the counter reached to 10 (decimal). If it is less than 10 go to the beginning of Mainloop. If it reached 10, go to the beginning of Setup to reset everything and start again.

Finally, the *Delay* subroutine is given to us.

```
Delay       mov.w #0Ah, r14
L2          mov.w #07A00h, r15
L1          dec.w r15
            jnz   L1
            dec.w r14
            jnz   L2
            ret
```

Listing 4: Part 1 - Delay

This function creates a 1-second delay. To sum up it, it loads some big number to R15 and decrements this number until 0 R14 (10) times. It takes 1-second for computer to complete this.

2.2.2 SECOND PART

The purpose of this part is to create an interrupt routine so it can count even numbers or odd numbers.

To do so, we first defined the input arrays above `.text` part.

```
            .data
array_even  .word 00111111b,  01011011b, 01100110b, 01111101b,  01111111b
lastEven      ; 0     2     4    6     8
array_odd .word 00000110b, 01001111b, 01101101b, 00000111b, 01101111b
lastOdd       ; 1     3    5     7     9
boolvar   .byte 00000000b
```

Listing 5: Part 2 - Inputs

The first array is *array_even* and it includes even numbers as its name indicates. Similarly, *array_odd* has odd numbers in it. *lastEven* and *lastOdd* are defined to detect the last elements of those arrays but are not used.

Then, *init_INT* is given to us in the experiment sheet.

```
1               mov.b #00000000b, &P2SEL
2 init_INT      bis.b #040h, &P2IE
3               and.b #0BFh, &P2SEL
4               and.b #0BFh, &P2SEL2
5
6               bis.b #040H, &P2IES
7               clr   &P2IFG
8               eint
```

Line 1: We write this every time to avoid the computer assigning random values. It does not belong to *init_INT* interrupt routine.

Line 2: Enable interrupt at P2.6.

Line 3, 4: Set 0 to P2SEL.6 and P2SEL2.6.

Line 6: This is high-to-low interrupt mode.

Line 7: Clear the flag.

Line 8: Enable interrupts.

Then, we wrote the setup.

```
1               mov.b #11111111b, &P1DIR
2               mov.b #00001000b, &P2DIR
3               mov.b #00000000b, &P1OUT
4 Setup         mov.w #array_even, r13
5               mov.w #array_odd, r12
6               mov.b #00h, r11
7               mov.b #00000000b, r10
```

Line 1: Select A, B, C, D, E, F, G, and H as output.

Line 2: Choose only the P2.3 bit as output.

Line 3: Initialize the outputs of P1 as low.

Line 4: Lines before are not labeled as *Setup*. Because it is not necessary to do them again when the Setup is called (we only want to reset the arrays and the counter). And we also moved the starting address of the array *array_even* as word to R13.

Line 5: Move the starting address of the array *array_odd* as word to R12.

Line 6: Write 0 to R11 as it will be used to decide whether pressed to interrupt button.

Line 7: Write 0 to R10 as it will be used as counter later.

Then, we wrote the deciding part which checks whether an interrupt is perceived and branches to the necessary function, and those functions.

```
decidemode      cmp   #0, r11
                jeq   even_loop
                jmp   odd_loop

```

```
5  even_loop        mov.b 0(r13), &P1OUT
6                   call  #Delay
7                   inc.w r13
8                   inc.w r13
9                   inc.b r10
10                  cmp   #5, r10
11                  jl    decidemode
12                  call  #r_counter
13                  jmp   decidemode
14
15 odd_loop         mov.b 0(r12), &P1OUT
16                  call  #Delay
17                  inc.w r12
18                  inc.w r12
19                  inc.b r10
20                  cmp   #5, r10
21                  jl    decidemode
22                  call  #r_counter
23                  jmp   decidemode
```

Line 1, 2, 3: Checks R11 to decide which numbers (even or odd) to count. If R11 is equal to 0, it branches to *even_loop* to count even numbers. If not, it branches to *odd_loop* to count odd numbers.

Line 5: Show what is written in the memory where R13 (even_array) points to P1OUT.

Line 6: Call *Delay* subroutine for output to be observed.

Line 7, 8: After returning from the subroutine, increment R13 two times for the next element.

Line 9: Increment the counter.

Line 10, 11: Check if the counter reached to 5 (there are 5 elements in the array). If it is less than 5, branch to *decidemode* to check whether interrupt button pressed.

Line 12: If the counter reached to 5, call the subroutine *r_counter* to reset the counter.

Line 13: After returning from the subroutine, branch to *decidemode* to check whether interrupt button pressed.

Line from 15 to 23: We do the same thing we done in the *even_loop* but for R12 instead of R13.

We added the same *Delay* subroutine from the first part and the *ISR* (mostly) given to us in the experiment sheet.

```
Delay            mov.w #0Ah, r14
L2               mov.w #07A00h, r15
L1               dec.w r15
                 jnz   L1
```

```
5                       dec.w r14
6                       jnz   L2
7                       ret
8
9  ISR                  dint
10                      xor.b #11111111b, r11
11                      call  #r_counter
12                      clr   &P2IFG
13                      eint
14                      reti
```

Delay works same as the first part. ISR is an interrupt routine mostly given us. It works with init_INT. We added the lines:

Line 10: XOR all bits of R11 with 1. So it can change the mode.

Line 11: Call the *r_counter* subroutine to reset the counter.

Line 12: Clear the flag.

Finally, the *r_counter* subroutine which resets counter is implemented.

```
1  r_counter            mov.b #00000000b, r10
2                       mov.w #array_even, r13
3                       mov.w #array_odd, r12
4                       ret
```

Line 1: Write 0 to counter R10.

Line 2: Move the starting address of the *array_even* to R13.

Line 3: Move the starting address of the *array_odd* to R12.

Line 4: Return to the next instruction of the caller function.

3 RESULTS [15 points]

3.1 FIRST PART

We were able to see the desired result, our number counted from 0 to 9 in a loop.

3.2 SECOND PART

We were able to see the desired result, our loops counted even and odd numbers appropriately and the interrupt worked as expected. When toggle button was pushed even loop turned into odd and odd loop turned into even loop.

4 DISCUSSION [25 points]

4.1 PART 1

We started by setting up a program in place to count numbers from 0 to 9 on a 7-segment display. The knowledge obtained consists of: GPIO Setup and Control of the Seven-Segment Display: For the segments to be efficiently controlled, proper GPIO initialisation was necessary. To guarantee proper display output, the mapping between GPIO ports and the 7-segment display pins was important.A delay subroutine was used in the experiment to produce a one-second delay between counter increments. In assembly, this provided an actual understanding of loop-based timing methods.

```
Delay               mov.w  #0Ah, r14
L2                  mov.w  #07A00h, r15
L1                  dec.w  r15
                    jnz    L1
                    dec.w  r14
                    jnz    L2
                    ret
```

4.2 PART 2

The even and odd counting modes were switched using an interrupt. Our investigation of real-time responsiveness to stimulation from outside was conducted by enabling a maskable interrupt on GPIO Port 2.

```
ISR                 dint
                    xor.b  #11111111b, r11
                    call   #r_counter
                    clr    &P2IFG
                    eint
                    reti
```

A Boolean variable was toggled within the interrupt service routine to determine whether to count even or odd numbers.

5 CONCLUSION [10 points]

- In the first part, we learned how to use the port connections for the 7-segment-display, how to light up the specific numbers and how to run over them in a loop.
- In the second part we learned how to initialize a interrrupt vector and how to use it for our functions, such as a toggle button.

- We mostly struggled with how to use the interrupt. We misunderstood the specified button for interrupt and kept trying with the wrong button. Also we struggled with when to decide the mode change. However, we were able to fix our problems and get the desired results.

REFERENCES

[1] Microcomputer Lab. Micro_experiment_5. *Lab Booklet*, 2024.