ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 351E
MICROCOMPUTER LABORATORY EXPERIMENT REPORT

EXPERIMENT NO	:	7
EXPERIMENT DATE	:	27.12.2024
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]

This exercise uses an MSP430 microcontroller to create and analyse two methods for producing pseudorandom numbers: the Middle Square Weyl Sequence and the Blum Blum Shub algorithm. These methods were selected due to their unique features and possible uses in random number generation and cryptography. The experiment also entails showing the results on a seven-segment LED display after converting binary numbers to their Binary-Coded Decimal (BCD) representation.
Understanding the fundamentals of pseudorandom number generation, practicing interrupt and stack-based operation usage, and interacting with hardware elements like LEDs and 7-segment displays are among the goals. By doing this, the project aims to strengthen fundamental ideas in microcontroller-based system design and assembly programming.

2 MATERIALS AND METHODS [40 points]

2.1 MATERIALS

- MSP430
- Selin's Laptop

2.2 METHODS

2.2.1 FIRST PART

We defined the array for 7-segment display numbers.

```
      .data
array   .word 00111111b, 00000110b, 01011011b, 01001111b, 01100110b, 01101101b, 01111101b, 00000111b, 01111111b, 01101111b
```

First, we initialize the interrupt.

```
1 init_INT
2         bis.b #020h, &P2IE
3         and.b #0DFh, &P2SEL
4         and.b #0DFh, &P2SEL2
5
6         bis.b #020H, &P2IES
7         clr   &P2IFG
8         eint
```

Line 2: Clears all pin selections for Port 2.

Line 3: Enables interrupt on bit 5 (P2.5) of Port 2.

Line 4: Clears the selection for P2.5 in P2SEL (00011111b).

Line 5: Clears the selection for P2.5 in P2SEL2.

Line 6: Sets the interrupt edge for P2.5 (falling edge).

Line 7: Clears the interrupt flag for Port 2.

Line 8: Enables general interrupts.

Then, we make the *Setup*.

```
1         mov.b #11111111b, &P1DIR
2         mov.b #00100000b, &P2DIR
3         mov.b #00000000b, &P1OUT
4 Setup
5         mov.w #11, r10   ;p = 11
6         mov.w #13, r11   ;q = 13
7         mov.w #5, r12 ;s = 5
8         mov.w #0, r5  ;s = 5
9         push.w  r10      ;p
10        push.w  r11      ;q
11        call  #m_mul
12        pop.w   r10
13        pop.w r11
```

Line 1: Makes A, B, ..., H output.

Line 2: Makes only the P2.5 output.

Line 3: Makes the pins closed off at first.

Line 5-7: Initializes the algorithm variables p = 11, q = 13, and s = 5, and prepares for multiplication. And they are stored in R10-12 registers.

Line 8: Clears R5. It will hold the intermediate multiplication result.

Line 11: Calls multiplication subroutine to multiply p and q (result is M).

The main loop (*blumblum*) takes the second power of s and writes it in R13. Takes R13's mode M (p*q). The result is held in R8 (r). And makes this result (r) new seed (s). Calls the same loop again.

```
blumblum  ;push these onto stack for power
        push.w  r12   ;s
        push.w  r12   ;s (for power)
        call  #m_mul
        pop.w r13   ;r
        pop.w r12   ;empty stack

        push.w  r13   ;r
        push.w  r10   ;M
        call #modulo
```

```
        pop.w r8    ; r = cevap
        mov.w r8, r12 ; s = r
        jmp blumblum
```

*m_mull*is the multiplication subroutine. It adds A to itself B times.

The *modulo* subroutine is used to find the modulus of a given number. It subtracts the mod m from a given number until it becomes less than m.

If the button P2.5 is pressed interrupt service routine starts to run.

```
1 ISR     dint
2         call  #binary_to_bcd
3         call  #shownumber
4         clr   &P2IFG
5         eint
6         reti
```

Line 2: It calls binary_to_bcd subroutine.

Line 3: It calls the shownumber subroutine to light the 7-segment display.

Line 4: Clears the flag.

binary_to_bcd converts binary number into its Binary-Coded Decimal representation.

```
;r4 = hundreds
;r5 = tens
;r6 = ones
binary_to_bcd
        mov.w #0, r4
        mov.w #0, r5
        mov.w #0, r6

        cmp     #143, r8
        jge     error

    ; Extract hundreds digit
hundreds_loop
        cmp     #100, r8
        jl      tens_loop
        sub.w   #100, r8
        inc.w   r4
        jmp     hundreds_loop

    ; Extract tens digit
tens_loop
        cmp     #10, r8
```

```
23         jl       ones_loop
24         sub      #10, R8
25         inc      r5
26         jmp      tens_loop
27
28     ; Extract ones digit
29 ones_loop
30         mov.w     r8, r6
31         ret
32
33 error
34         mov.w     #0xFF, r4         ; Error indicator
35         mov.w     #0xFF, r5
36         mov.w     #0xFF, r6
37         ret
```

Line 5-7: Clears the hundreds, tens, and ones registers (r4, r5, r6).
Line 14-18: Subtracts 100 from R8 while incrementing the hundreds digit in R4.
Line 21-26: Subtracts 10 from R8 while incrementing the tens digit in R5.
Line 29-31: Assigns the remaining value in R8 to the ones digit in R6.
Line 33-37: If the inputs exceeds 143, sets all digits to 0xFF (error code).

The *shownumber* subroutine displays the calculated BCD digits on 7-segment leds.

```
shownumber  mov    r4, r12
        call   #led_value
        mov.b    r12, &P1OUT
        mov.b    #1, &P2OUT              ; Enable first digit (P2.0)
        call     #ShortDelay

        mov    r5, r12
        call   #led_value
        mov.b    r12, &P1OUT
        mov.b    #2, &P2OUT              ; Enable second digit (P2.1)
        call     #ShortDelay

        mov    r6, r12
        call   #led_value
        mov.b    r12, &P1OUT
        mov.b    #4, &P2OUT              ; Enable third digit (P2.2)
        call     #ShortDelay
```

1. Sequentially processes the hundreds (r4), tems (r5), and ones (r6) digits.

2. *led_value* iss called to translate each digit into a segment pattern.

3. Activates the appropriate P2 segment (e.g., P2OUT) to enable display.

led_value maps a digit (0-9) to its corresponding 7-segment pattern in *array*.

```
1 led_value push    r5                      ; Save R5 (used as index register)
2         mov.w   #array, r5        ; Load address of array
3         add.w   r12, r5               ; Offset to the correct pattern
4         mov.b   @r5, r12              ; Load segment pattern into R12
5         pop     r5                    ; Restore R5
6         ret
```

Line 2: The address of the LED segment pattern array is loaded into r5.

Line 3: The current digit (r12) is used as an offset to find the correct segment value.

Line 4: The pattern is loaded into r12 for display.

2.2.2 SECOND PART

We defined the array for 7-segment display numbers.

```
      .data
array   .word 00111111b, 00000110b, 01011011b, 01001111b, 01100110b,
   01101101b, 01111101b, 00000111b, 01111111b, 01101111b
```

First, we initialize the interrupt.

```
1 init_INT
2         bis.b #020h, &P2IE
3         and.b #0DFh, &P2SEL
4         and.b #0DFh, &P2SEL2
5
6         bis.b #020H, &P2IES
7         clr   &P2IFG
8         eint
```

Line 2: Clears all pin selections for Port 2.

Line 3: Enables interrupt on bit 5 (P2.5) of Port 2.

Line 4: Clears the selection for P2.5 in P2SEL (00011111b).

Line 5: Clears the selection for P2.5 in P2SEL2.

Line 6: Sets the interrupt edge for P2.5 (falling edge).

Line 7: Clears the interrupt flag for Port 2.

Line 8: Enables general interrupts.

Then, we make the *Setup*

```
Setup
        mov.b   #00000000b, &P2SEL
```

```
3         mov.b #00000000b, &P2SEL2
4         mov.b   #11111111b, &P1DIR
5         mov.b   #00001111b, &P2DIR
6         bic.b   #0FFh, &P1OUT
7         mov.b   #001h, &P2OUT
8         mov.w   #array, r13
9
10        mov.w #0, r12 ;x
11        mov.w #0, r11 ;w
12        mov.w #5, r10 ;s
```

Line 2: Clears all pin selections for Port 2.

Line 3: Enables interrupt on bit 5 (P2.5) of Port 2.

Line 4: Set P1DIR as output for 7-segment segments

Line 5: Set P2DIR as output for digit selection (P2.0-P2.3)

Line 6: Initialize P1OUT (all segments OFF)

Line 7: Initialize P2OUT (all digits OFF)

Line 8: Load starting address of lookup table into r13

Line 10-11-12: Initialize the x, w and s values for the algorithm

The main loop (Middle Square Weyl Sequence):

1. Takes x's square
2. Adds up s to w
3. Adds w to x
4. Right shifts x by 4 and left shifts x by 4
5. ORs these shifted values and loads them to r

To square x *square* is called it takes the square of the number

```
main
      push.w  r12      ;push x
      call  #square
      pop.w r12      ;x = x.x
      add.w r10, r11  ;w = w + s
      add.w r11, r12    ;x = x + w
      push.w  r12      ; to not lose original x
      mov.w r12, r9    ; to left shift x
      rra   r12
      rra   r12
      rra   r12
      rra   r12
```

```

    rla   r9
    rla   r9
    rla   r9
    rla   r9

    bis.w r12, r9   ;r9 = r
    pop.w r12      ; to get original x back
    push.w  r9      ;r
    push.w  #10000000b  ;129 for modulo
    call  #modulo
    pop.w   r8
    jmp   main
```

For the interrupt routine

```
1 ISR     dint
2       ;;;; what happends at interrupt
3       push.w  r9      ;r
4       push.w  #10000001b  ;129 for modulo
5       call  #modulo
6       pop.w r8
7       pop.w r9      ;empty stack
8       ;modulonun returnunu r8'e at r8 cevap
9       ;call   #binary_to_bcd
10      ;call   #shownumber
11      clr   &P2IFG
12      eint
13      reti
```

Line 3-4: Pushes the parameters for modulo

Line 5: Calls *modulo*

Line 6-7: Takes the modulo to r8 and empties stack

In modulo, mod 129 of the number is found using subtraction

We have also tried to show the numbers at 7 segment as explained in the first part but as it did not work at the first part it also did not work at the second part.

3 RESULTS [15 points]

3.1 FIRST PART

For the first part, we were able to get the desired results at the registers but not on the displays

3.2 SECOND PART

For the second part, we were able to get the desired results at the registers but not on the displays

4 DISCUSSION [25 points]

Despite the difficulties, the experiment offered a useful chance to apply learnt concepts in real-world settings. It showed mastery of hardware the interface, stack management, interrupt handling, and function calls. The persisting display problem, however, emphasises how important iterative testing and debugging are to hardware-software integration.

4.1 BLUM BLUM ALGORITHM

Using modular arithmetic, the Blum Blum Shub (BBS) algorithm generates pseudorandom numbers in a cryptographically secure manner. We were able to compute and validate intermediate outcomes at the register level in the experiment. Nevertheless, the "shownumber" subroutine's display capabilities was unsuccessful, most likely as a result of improper segment mappings or timing problems while turning on the display segments.

4.2 MIDDLE SQUARE WEYL SEQUENCE

Effective register-level pseudorandom number generation was shown by the Middle Square Weyl Sequence technique, with accurate intermediate computations noted. Notwithstanding these achievements, the generated numbers were not seen on the 7-segment display. The problem could be related to improper handling of the digit-enabling signals or misconfigured display activation.

4.3 BINARY TO BCD

Register outputs confirmed that the "binary_to_bcd" subroutine was successful in separating a binary number's hundreds, tens, and ones digits. Although the failure of the ensuing "shownumber" function prevented complete visualisation, this subroutine served as a basis for transforming numerical values into representations that could be seen.

4.4 SHOW NUMBERS

A major problem with the "shownumber" function is shown by the inability to visualise data on the 7-segment display in both experiments. Inadequate delay intervals, inaccurate timings in digit switching, or mistakes in mapping segment patterns are some possible causes. To fix this, more hardware interface and timing signal debugging is required.

5 CONCLUSION [10 points]

In conclusion to both experiments, we used all the things we learned at the term, such as:

- Function calls
- Interrupts
- Stack usage
- LED and 7 segment usage

However for both parts, we were not able to see the desired results at the displays as we could not figure out what was wrong with our *shownumber* function. So we showed them on the registers.

REFERENCES

[1] Microcomputer Lab. Micro_experiment_7. *Lab Booklet*, 2024.