Microprocessors Project

Monoalphabetic substitution encryption

name	ID	
ابراهيم علاء محمد محب	22010475	
عمر أحمد عباس شحاتة	22010944	
سلمي محمد علي محمد	22010812	
مني ياسر عوض السيد	22011272	
عمر أشرف شكري	22010951	

General overview

This assembly code is written for a 8086up that interacts with an LCD and a keypad.

(1) It begins by initializing the stack pointer and data segment. The program displays a static message ("message:") on the LCD, then waits for user input via the keypad.

(2) The user can input up to 16 characters; digits (0-9) are accepted while spaces are ignored. The '+' key triggers an encryption routine that substitutes characters using a predefined lookup table (ENC_TABLE), displaying the result prefixed by "enc:".

(3) After encryption, a decryption process reverses the substitution using another lookup table (DEC_TABLE), and the original message is shown again

under the "dec:" label. If the '-' key is pressed at any time, the program clears the LCD and restarts the process.

(4) The program utilizes hardware-mapped I/O for reading keys and writing to the LCD, includes busy-wait synchronization to ensure LCD readiness, and uses indirect addressing to handle input and buffer management.

So we use "+" to triggers an encryption routine and "-" to Clear LCD and tell microprocessor that we want to enter a new input

org	enc	org	enc	org	enc	org	enc
a	q	h	i	o	g	v	c
b	w	i	o	p	h	w	v
c	e	j	p	q	j	x	b
d	r	k	a	r	k	y	n
e	t	l	s	s	l	z	m
f	y	m	d	t	z		
g	u	n	f	u	x		

DATA79 refers to the data register of device "79", This is connected to a keypad or LCD data line, It's where the program reads or writes actual data.

CNTR79 This is the control register for device 79 (Keypad), Reading from this port lets the CPU check the status of the keypad (whether a key is pressed)

```
DATA79    EQU    0FFE8H
CNTR79    EQU    0FFEAH
IR_WR     EQU    0FFC1H
IR_RD     EQU    0FFC3H
DR_WR     EQU    0FFC5H
```

Label	Address	purpose	Direction	used for
DATA79	0FFE8H	Keypad or LCD data Register	Read	Reading key value
CNTR79	0FFEAH	Keypad or Control Register	Read	Checking if the key is pressed
IR_WR	0FFC1H	LCD instruction register	write	Sending commands to LCD , Ex : Clear
IR_RD	0FFC3H	LCD instruction register	Read	Checking if LCD is busy
DR_WR	0FFC5H	LCD data Register	write	Writing display characters

Note : these functions Deals only with AL register in 8086up

```
MAXLEN    EQU    16

CODE    SEGMENT
        ASSUME CS:CODE, DS:CODE
        ORG 0

START:
        MOV SP, 4000H
        MOV AX, CS
        MOV DS, AX
```

```
        ; ??? ?????? "message:"
        MOV AH, 01H
        CALL IRWR
        LEA SI, MSG
```

The **8086** has a segmented memory model, which divides memory into **four main segments**, **CODE SEGMENT** tells the assembler: **"The following lines belong to the code segment."**

ASSUME CS:CODE, DS:CODE

This tells the assembler to assume that both the Code Segment (CS) and Data Segment (DS) registers point to the same CODE segment, Since we're keeping things simple, both instructions and data are placed in the same segment.

ORG 0

This sets the origin (starting address) of the segment to 0, Which Meaning: All labels and offsets within this segment will be calculated starting from address 0000H.

START

This is a label that marks the entry point of the program.

END START at the bottom of the code will tell the assembler: this is the starting address for execution

MOV SP, 4000H : Sets the stack pointer (SP) to 4000H.

MOV AX, CS; MOV DS, CS : at this point, both CS and DS point to the same segment (CODE), so both code and data are accessible

MOV AH, 01H : clear LCD display before start any inputs

CALL IRWR : Calls the **IRWR subroutine**, which writes the content of AH to the LCD's Instruction Register using port-mapped I/O.

LEA SI,MSG : Load the offset address of MSG into SI

```
MAIN_LOOP:
RESTART_PROGRAM:
        CALL CLEAR_LCD
        MOV AH, 80H
        CALL IRWR
        LEA SI, MSG
PRINT_MSG:
        LODSB
        OR AL, AL
        JZ INPUT_START
        CALL OUTL
        JMP PRINT_MSG
INPUT_START:
        MOV AH, 0C0H
        CALL IRWR
        XOR SI, SI
```

1

```
NEXT_CHAR:
        CALL READ_KEY
        CMP AL, '+'
        JE ENCRYPT_PROCESS
        CMP AL, '-'
        JE RESTART_PROGRAM
        CMP AL, ' '
        JE NEXT_CHAR
        CMP AL, '0'
        JB VALID_CHAR
        CMP AL, '9'
        JBE NEXT_CHAR
VALID_CHAR:
        CALL OUTL
        MOV [INPUT_BUFFER + SI], AL
        INC SI
        CMP SI, MAXLEN
        JB NEXT_CHAR
        JMP ENCRYPT_PROCESS
```

2

3

```
ENCRYPT_PROCESS:
        CALL CLEAR_LCD      ; ????? "enc:"
        MOV AH, 80H
        CALL IRWR
        LEA SI, ENC_LABEL
PRINT_ENC:
        LODSB
        OR AL, AL
        JZ DO_ENCRYPT
        CALL OUTL
        JMP PRINT_ENC
DO_ENCRYPT:
        XOR DI, DI          ; ????ENC_BUFFER
        XOR SI, SI
ENC_LOOP:
        MOV AL, [INPUT_BUFFER + SI]
        CMP AL, 0
        JE PRINT_DEC
        CALL ENCRYPT_CHAR
        MOV [ENC_BUFFER + DI], AL
        CALL OUTL
        INC SI
        INC DI
        JMP ENC_LOOP
```

4

```
PRINT_DEC:
        MOV AH, 0C0H
        CALL IRWR
        LEA SI, DEC_LABEL
PRINT_DEC_LABEL:
        LODSB
        OR AL, AL
        JZ DO_DECRYPT
        CALL OUTL
        JMP PRINT_DEC_LABEL
DO_DECRYPT:
        XOR SI, SI
DEC_LOOP:
        MOV AL, [ENC_BUFFER + SI]
        CMP AL, 0
        JE Reset
        CALL DECRYPT_CHAR
        CALL OUTL
        INC SI
        JMP DEC_LOOP
Reset:  CALL READ_KEY
        CMP AL, '-'
        JNE  SKIP_RESTART_PROGRAM
        JMP  RESTART_PROGRAM
        SKIP_RESTART_PROGRAM:
            JMP Reset
DONE:  JMP $
```

We will talk about each step from these four steps above.

Part one

This part includes the following main stages:

- **Restarting or initializing the program**
- **Displaying a message to the user**
- **Preparing to take user input**
- **Jumping to the input handling part (NEXT_CHAR)**

CLEAR_LCD	Clears the screen of LCD
MOV AH, 80H	Load Display command
CALL IRWR	Write command to the IR
LEA SI, MSG	Load string address into SI for display
LODSB	Load each character and display it with OUTL
JZ INPUT_START	When message ends , go to input preparation
MOV AH, 0C0H	Move cursor to input field on LCD
CALL IRWR	Send this instruction to LCD
XOR SI, SI	Reset SI to 0 for input buffer indexing
JBE NEXT_CHAR	Jump to start handling key presses

First, it Clears LCD by moving 01H to AH and call IRWR , then it Moves 80H to AH (80H (1000 0000b) is a standard command in many character LCDs),

So **MOV AH,80H** is a standard command used to load the LCD.

```
IRWR:
   CALL BUSY          ; 1. Wait until the LCD is not busy
   MOV DX, IR_WR      ; 2. Set I/O port address for instruction register write
   MOV AL, AH         ; 3. Move the command into AL to send
   OUT DX, AL         ; 4. Output the command to LCD
   RET                ; 5. Return to the caller
```

After that , we use CALL IRWR to move content of AH to the LCD , so CALL IRWR is a call to a subroutine named IRWR, whose job is to write a command to the LCD using I/O ports.

LEA SI,MSG load the MSG address into SI For reading a string or array using pointer-style access

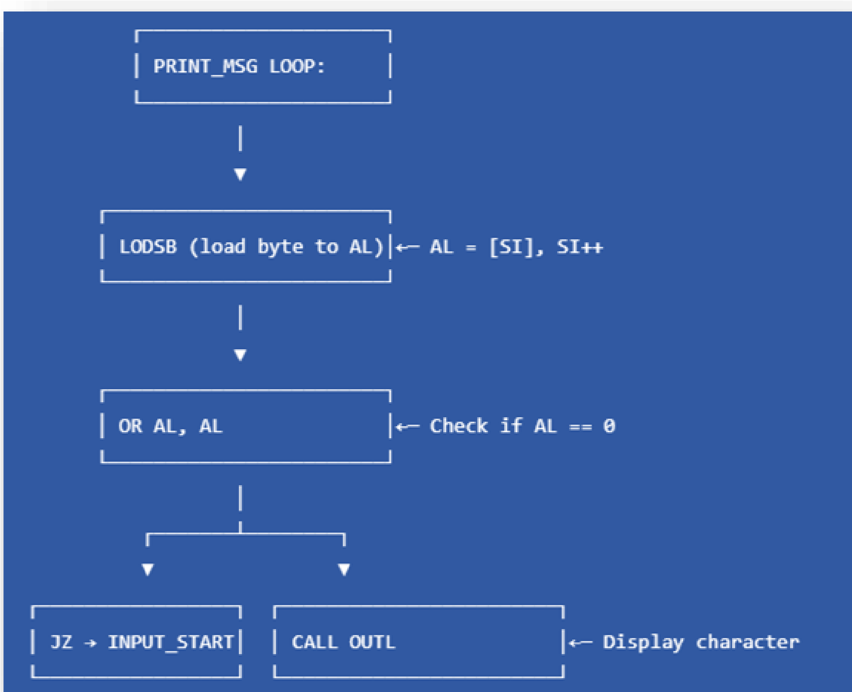
NOTE , we didn't use MOV SI,MSG so it would copy the value at MSG instead of its address

LODSB

- **Loads a single byte** from memory location DS:SI (the memory pointed to by the Source Index), **Copies it into register AL**
- **Automatically increments or decrements SI**, depending on the direction flag.

It is short for:

```
MOV AL, [SI]   ; Load the byte pointed by SI into AL
INC SI         ; Move to the next byte (if DF=0), or DEC SI if DF=1
```




```
OUTL:
        CALL BUSY
        MOV DX, DR_WR
        OUT DX, AL
        RET
```

If we print word “message” , then it is the time to take the input from the user now , by MOV AH,0C0H , it is command key means that we want to tell up to take input from second line , then CALL IRWR to move content of AH to LCD device , then we make XOR SI , IS to Reset input buffer index (SI)

Part two

CALL READ_KEY Reads a key from the keyboard and stores it in register AL

Then **if input is “+”** , **then it will jump to ENCRYPT_PROCESS** , **else if it is “-“**, **the program jumps back and resets everything**, **if it was empty space** **it will ignore it and jump back to NEXT CHAR.**

JB = Jump if below

JBE = Jump if below or equal

JE = **Jump if equal**

JNE = Jump not equal

JMP = Jump

Part three

Clear LCD again , send 80H again to LCD to tell it that we want to set cursor at first line at the top left of first line (Load Display command) – reset cursor position , then load address of ENC_LABEL into SI , which contains massage “ enc : “ then print it on the screen of LCD by PRINT_ENC process

Then Clear the registers DI and SI by XORing them with themselves (common fast way to set to zero).

```
ENC_LOOP:
        MOV AL, [INPUT_BUFFER + SI]
        CMP AL, 0
        JE PRINT_DEC
        CALL ENCRYPT_CHAR
        MOV [ENC_BUFFER + DI], AL
        CALL OUTL
        INC SI
        INC DI
        JMP ENC_LOOP
```

```
ENCRYPT_CHAR:
        PUSH BX
        MOV BX, OFFSET ENC_TABLE
        MOV DL, AL
        SUB DL, 'A'
        JB INVALID
        CMP DL, 25
        JA INVALID
        XOR DH, DH
        ADD BX, DX
        MOV AL, [BX]
        JMP EXIT_ENC
INVALID:
        MOV AL, '?'
EXIT_ENC:
        POP BX
        RET
```

Remember , we can make offset in Data segment using BX , SI , DI only , and the input text is stored in Data segment.

ENCRYPT_CHAR is a subroutine written in x86 Assembly that takes a character in AL (the lower 8 bits of register AX) and:

- **Encrypts it using a substitution cipher defined by the ENC_TABLE.**
- **If the character is not a capital letter (A-Z), it replaces it with '?'**

This function is called per character in the encryption loop (ENC_LOOP) you saw earlier.

The Idea of this code

First step : convert any capital letter to an index from 0 to 25 by subtract 'A' from the original letter , as example :

Input letter	ASCII	After conversion
'C'	67	67 – 65 = 2
'K'	75	75 – 65 = 10
'B'	66	66 – 65 = 1

Second Step : Load the encryption table by puts the **starting address** of the encryption table into register **BX**

Third step : Convert character into an index

```
MOV DL, AL      ; Copy input char (e.g., 'C') to DL
SUB DL, 'A'     ; Convert 'C' → 2 (because 67 - 65 = 2)
```

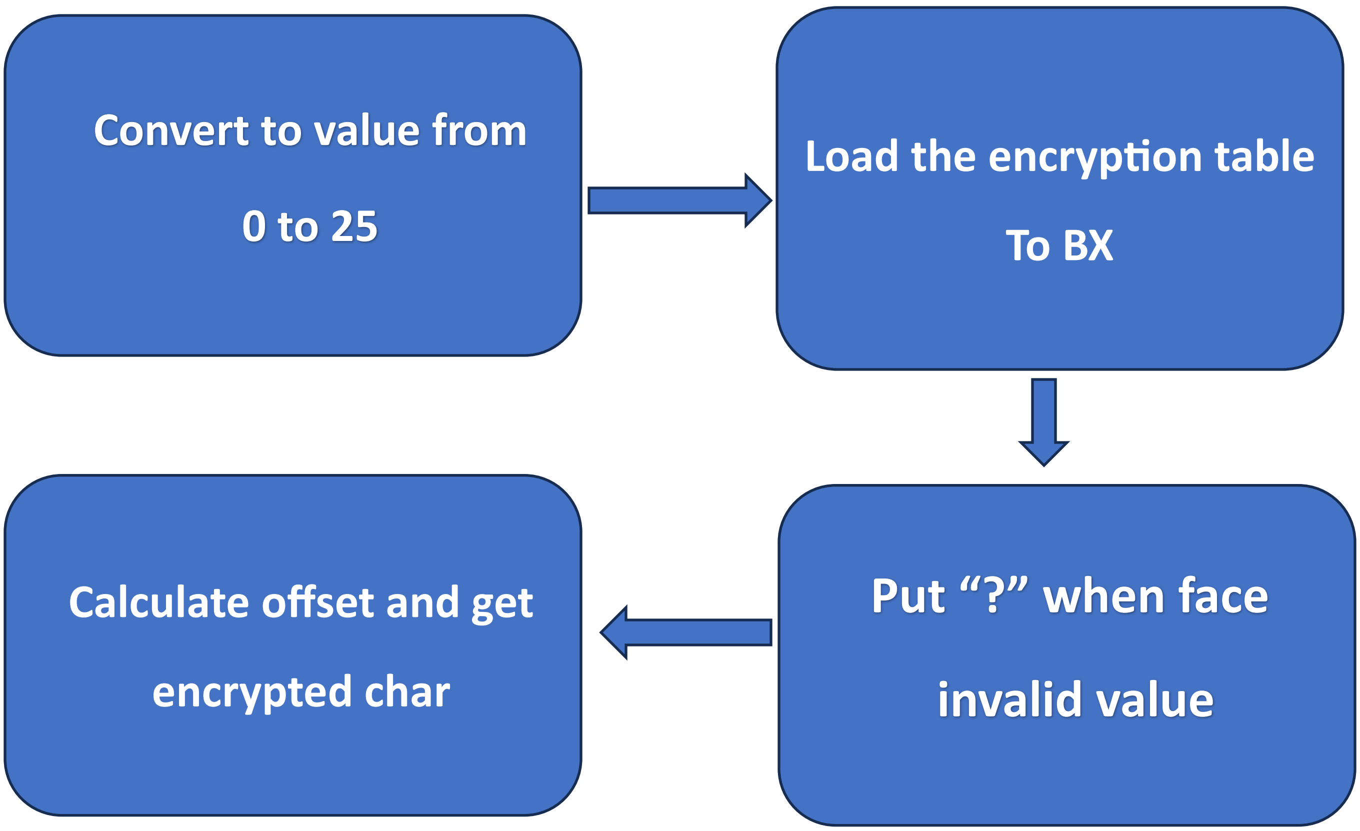
Fourth step : if DL contains value larger than 25 or smaller than 0 then it is invalid input (not letter) , so we will display it as "?" on the screen

```
JB INVALID      ; If DL < 0 → not A-Z → go to INVALID
CMP DL, 25      ; Compare index with 25
JA INVALID      ; If DL > 25 → not a valid capital letter
```

Fifth step : Calculate offset and get encrypted char

```
XOR DH, DH      ; Clear upper 8 bits of DX → now DX = DL
ADD BX, DX      ; Move to ENC_TABLE[index]
MOV AL, [BX]    ; Get encrypted letter from ENC_TABLE
```

BX contains the address of ENC_TABLE , by add the DL now which contains the index from 0 to 25 , we can arrive to encrypted char easily



Fourth part

Now we need to convert encrypted letters again to decrypted letters , the original form again

It is same idea of encryption , sub letters ASCII value from ASCII value of ‘a’ which equal 115 in ASCII table , to become a value between 0 and 25

enc	Sub	Index from Dec	enc	Sub	Index from Dec
a	a – a = 0	Dec[0] = k	p	p – a = 15	Dec[15] = j
b	b – a = 1	Dec[1] = x	q	q – a = 16	Dec[16] = a
c	c – a = 2	Dec[2] = v	r	r – a = 17	Dec[17] = d
d	d – a = 3	Dec[3] = m	s	s – a = 18	Dec[18] = l
e	e – a = 4	Dec[4] = c	t	t – a = 19	Dec[19] = e
f	f – a = 5	Dec[5] = n	u	u – a = 20	Dec[20] = g
g	g – a = 6	Dec[6] = o	v	v – a = 21	Dec[21] = w
h	h – a = 7	Dec[7] = p	w	w – a = 22	Dec[22] = b
i	i – a = 8	Dec[8] = h	x	x – a = 23	Dec[23] = u
j	j – a = 9	Dec[9] = q	y	y – a = 24	Dec[24] = f
k	k – a = 10	Dec[10] = r	z	z – a = 25	Dec[25] = t
l	l – a = 11	Dec[11] = s			
m	m – a = 12	Dec[12] = z			
n	n – a = 13	Dec[13] = y			
o	o – a = 14	Dec[14] = i			

```
DEC_TABLE    DB 'kxvmcnophqrszyijadlegwbuft'
```

Reset:

```
CALL READ_KEY        ; Waits for a key press and stores the ASCII value in AL

CMP AL, '-'          ; Compares the pressed key to the character '-'

JNE SKIP_RESTART_PROGRAM ; If it's NOT '-', skip restart

JMP RESTART_PROGRAM   ; If it IS '-', jump to RESTART_PROGRAM
```

SKIP_RESTART_PROGRAM:

```
JMP Reset            ; Loop back to wait for key input again
```

DONE:

```
JMP $                ; Infinite loop, halts the program
```

This section of code is used as a keyboard input monitor after some main logic (like encryption/decryption), It :

- **Waits for a key press**
- **If you press -, it restarts the program**
- **If you press anything else, it waits again**
- **Once done with all logic, it jumps to an infinite loop (JMP $), halting the CPU**