NAME : Katrina Ricci C. Batin
SUBJ/SEC: Computer Architecture and Organization/BSCOE4-6
DATE : November 28, 2024
PROFESSOR : Engr. Rolito L. Mahaguay

EXPERIMENT NO. 6
WORKING WITH NUMBERS

OBJECTIVE (S) :

1. Understand how math works in assembly language programming.
2. Create a program that can perform numeric operations.

REQUIREMENTS :

Personal Computer
System disk
Data disk

DISCUSSION:

An assembly program consists of a set of statements. The two types of statements are ***instructions*** and ***directives.***

Instruction ***:*** such as MOV and ADD, which the assembler translates to object code
Directives ***:*** These tell the assembler to perform a specific action, such as defining a data item.
: An assembly language supports a number of statements that enable you to control the way in which a source program assembles and lists.

An operator provides a facility for changing or analyzing operands during assembly. Operators are divided into various categories:

- ***Calculation Operators***: Arithmetic, index, logic, shift, and structure field name.
- ***Record Operators***: MASK and WIDTH
- ***Relational Operators***: EQ, GE, GT, LE, LT, and NE
- ***Segment Operators***: OffSET, SEG, and segment override
- ***Type (or attribute) operators***: HIGH, HIGHWORD, LENGTH, LOW, LOWWORD, PTR, SHORT, SIZE THIS, and TYPE.

Arithmetic Operators

These operators include familiar arithmetic signs and perform arithmetic during the assembly. In most cases, you could perform the calculation yourself, although the advantage of using these operators is that every time you change the program and reassemble it, the assembler automatically recalculates with an example of their use.

PROCEDURE:

1. Encode the given program.

```
        .model small
        .code
        org 100h

start:  jmp main
        x db “INPUT A SINGLE DIGIT NUMBER : $”
        y db “INPUT ANOTHER SINGLE DIGIT NUMBER : $”
        z db “THEIR SUM IS : $”

main    proc near
        mov dx, offset x
        call print
        call input_ok
        mov cl, al
        call down
        mov dx, offset y
        call print
        call input_ok
        mov ch, al
        call down
        mov dx, offset z
        call print
        add ch, cl
        mov ah, 2
        mov dl, ch
        add dl, ch
        mov ah, 2
        mov dl, ch
        add dl, ‘0’
        int 21h
        int 20h
main    endp
```

```
down      proc near
          mov ah, 2
          mov dl, 13
          int 21h
          mov dl, 10
          int 21h
          ret

down      endp

print     proc near
          mov ah, 9
          int 21h
          ret

print     endp

input _ok proc near
                    mov ah,1
                    int 21h
                    sub al, '0'
                    ret

input_ok  endp

end start
```

2. Execute the given program, then input 2 and 6 respectively.

```
C:\TASM>xp6samp
INPUT A SINGLE DIGIT NUMBER : 2
INPUT ANOTHER SINGLE DIGIT NUMBER : 6
THEIR SUM IS : 8
C:\TASM>
```

3. Does the expected output appear?
   Yes, the output is correct.

4. Execute the program again, then input 5 and 7.

```
C:\TASM>xp6samp
INPUT A SINGLE DIGIT NUMBER : 5
INPUT ANOTHER SINGLE DIGIT NUMBER : 7
THEIR SUM IS : <
C:\TASM>_
```

5. Does the expected sum appear?
   No, the expected sum is 12 but the output is <.

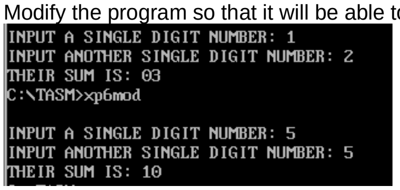
6. Input another set of single-digit numbers which will give a sum above nine. What do you observe?

```
INPUT A SINGLE DIGIT NUMBER : 4
INPUT ANOTHER SINGLE DIGIT NUMBER : 6
THEIR SUM IS : :
C:\TASM>xp6samp
INPUT A SINGLE DIGIT NUMBER : 7
INPUT ANOTHER SINGLE DIGIT NUMBER : 8
THEIR SUM IS : ?
```

48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

The symbols are based on the ASCII count of the sum.

7. Modify the program so that it will be able to display the sum of two input numbers even if their sum is double-digit.



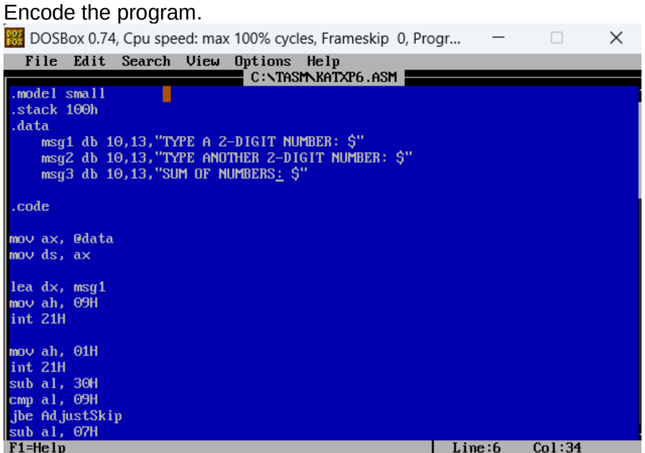
8. Write your new program in the space below.

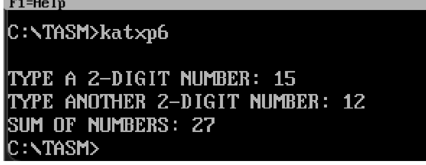
```
.model small
.code
org 100h

start:  jmp main
        x db 13,10,'INPUT A SINGLE DIGIT NUMBER: $'
        y db 13,10, 'INPUT ANOTHER SINGLE DIGIT NUMBER: $'
        z db 13,10, 'THEIR SUM IS: $'

main:
     mov dx,offset x
     mov ah,9
     int 21h

     mov ah,1
     int 21h

     mov cl, al

     sub cl, 30h

     mov dx,offset y
     mov ah,9
     int 21h
     mov ah,1
     int 21h

     sub al, 30h

     xor ah,ah
     add al,cl
     aaa

     mov cx, ax
     add cx,3030h
     mov dx, offset z
     mov ah,9
     int 21h

     mov ah,2
     mov dl, ch
     int 21h

     mov dl, cl
     int 21h

exit:
     mov ah, 4ch
     int 21h

end start
```

EXERCISES:

1. Write down the task given by your instructor.
   Write a program that adds two two-digit numbers.

2. Encode the program.





3. Ask your instructor to check your work.

ANSWER THE FOLLOWING QUESTIONS:

1. From your ASCII table, write the ASCII code for the following characters.

CHARACTER	DECIMAL	HEXADECIMAL
0	48	30
1	49	31
2	50	32
3	51	33
4	52	34
5	53	35
6	54	36
7	55	37
8	56	38
9	57	39
+	43	2B
-	45	2D
*	42	2A
/	47	2F

2. Compare the arrangement of the ASCII codes of numbers and letters.
   ASCII codes for numbers (0-9) range from 48 to 57, while uppercase letters (A-Z) are between 65 and 90, and lowercase letters (a-z) between 97 and 122. Numbers come before letters in the ASCII table, with each character having a unique code.

3. What method is used in getting the actual value of the input number?
   To obtain the actual numeric value from an input character, the ASCII value of the character is subtracted by 48 (the ASCII code for '0'). This gives the numeric value of the digit.

SUMMARY:
An assembly program consists of instructions and directives that guide the translation to object code. Operators are used to manipulate operands during assembly, such as arithmetic, relational, and segment operators. These help automate calculations, ensuring consistency in the program. ASCII codes for numbers, uppercase, and lowercase letters are used to convert characters into numeric values for further processing.

CONCLUSION:
Understanding assembly language structure, including operators and ASCII encoding, is essential for effective low-level programming. The use of arithmetic operators and converting ASCII codes to numeric values ensures accurate handling of user input and efficient programming.