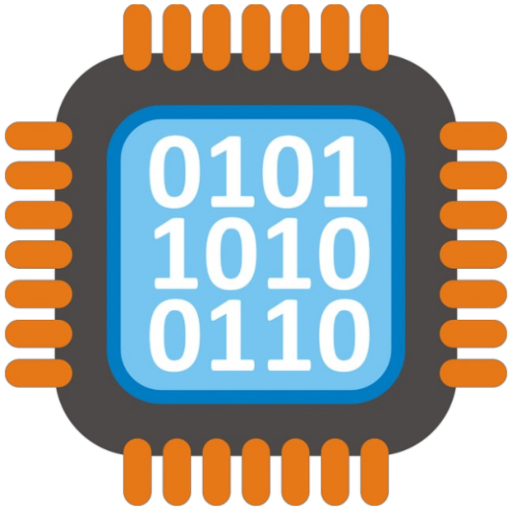
بسم الله الرحمن الرحيم

Secure Assembly Coding

Week # 5 Lectures

Qasem Abu Al-Haija, PhD

Department of Cybersecurity



Intel 8086 µp Programming using Assembly Language Programming

Introduction

```
;PROGRAM TO ADD TWO 16-BIT DATA (METHOD-1)

DATA SEGMENT                 ;Assembler directive

        ORG 1104H            ;Assembler directive
        SUM DW 0             ;Assembler directive
        CARRY DB 0           ;Assembler directive

DATA ENDS                    ;Assembler directive

CODE SEGMENT                 ;Assembler directive

       ASSUME CS:CODE        ;Assembler directive
       ASSUME DS:DATA        ;Assembler directive
        ORG 1000H            ;Assembler directive

        MOV AX,205AH         ;Load the first data in AX register
        MOV BX,40EDH         ;Load the second data in BX register
        MOV CL,00H           ;Clear the CL register for carry
        ADD AX,BX            ;Add the two data, sum will be in AX
        MOV SUM,AX           ;Store the sum in memory location (1104H)
        JNC AHEAD            ;Check the status of carry flag
        INC CL               ;If carry flag is set,increment CL by one
AHEAD:  MOV CARRY,CL         ;Store the carry in memory location (1106H)
        HLT

CODE ENDS                    ;Assembler directive
END                          ;Assembler directive
```

Program

A set of instructions written to solve a problem.

Instruction

Directions which a microprocessor follows to execute a task or part of a task.

Computer language

High Level

Low Level

Machine Language

- **Binary bits**

Assembly Language

- **English Alphabets**
- **'Mnemonics'**
- **Assembler Mnemonics → Machine Language**

Program template in EMU8086

```
DATA SEGMENT
; DEFINE YOUR DATA HERE
 ENDS

STACK SEGMENT
   DW   128  DUP(0)
ENDS
```

; keep it as is...stack contains 128 words of memory

```
CODE SEGMENT
START:
   MOV AX, DATA
   MOV DS, AX
   MOV ES, AX
```

; always include these three lines... get the address of data segment at runtime

```
; WRITE YOUR CODE HERE

   MOV AX, 4C00H
   INT 21H
```

;Two lines: exit to the operating system and terminate the program

```
ENDS
END START
```

Directives and Instructions

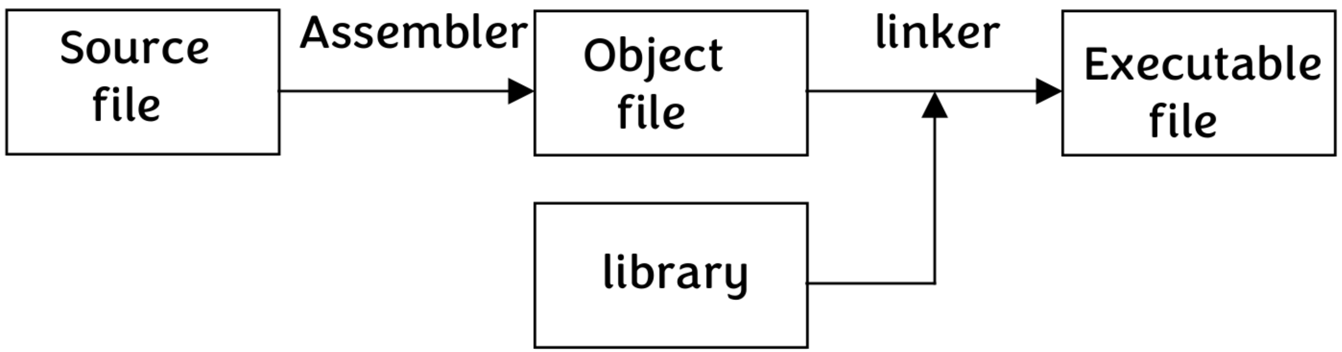
- Assembly language statements are either directives or instructions

- Instructions are executable statements. They are translated by the assembler into machine instructions. Ex:
  - CALL MySub ;transfer of control
  - MOV AX,5 ;data transfer

- Directives tells the assembler how to generate machine code and allocate storage. Ex:

```
COUNT DB 50        ;creates 1 byte
                   ;of storage
                   ;initialized to 50
```

Steps to Produce an Executable File



- The assembler produces an object file from the assembly language source
- The object file contains machine language code with some external and relocatable addresses that will be resolved by the linker. Their values are undetermined at that stage.
- The linker extracts object modules (compiled procedures) from a library and links them with the object file to produce the executable file.
- The addresses in the executable file are all resolved but they are still logical addresses.
- Note: the assembler of EMU8086 is MASM assembler which is case insensitive

Naming in Assembly

- A name identifies either:
  - a variable
  - a label
  - a constant
  - a keyword (assembler-reserved word).

Naming in Assembly (Cont.)

- A variable is a symbolic name for a location in memory that was allocated by a data allocation directive. Ex:

```
count db 50     ; allocates 1 byte to
                ; variable count
```

- A label is a name given to an instruction. It must be followed by ':'. Ex:

```
main:
          mov ax, 5
          xor ax, bx
          jump main
```

Naming in Assembly (Cont.)

- **The first character must be a letter or any one of ‘@’, ‘_’, ‘$’, ‘?’**
- **subsequent characters can include digits**
- **A programmer chosen name must be different from an assembler reserved word**
  - *Advice: avoid using ‘@’ as the first character since many keywords start with it*

Integer Constants

- **Integer constants are made of numerical digits with, possibly, a sign and a suffix. Ex:**
  - -23 (a negative integer, base 10 is the default)
  - 1011b (a binary number)
  - 1011 (a decimal number)
  - 0A7Ch (a hexadecimal number)
  - A7Ch (this is the name of a variable, a hexadecimal number must start with a decimal digit)

Character and String Constants

- **They are any sequence of characters enclosed either in single or double quotation marks. Embedded quotes are permitted. Ex:**
  - ‘A’
  - ‘ABC’
  - “Hello World!”
  - “123” (this is a string, not a number)
  - “This isn’t a test”
  - ‘Say “hello” to him’

Constants

- We can use the equal-sign (=) directive or the EQU directive to give a name to a constant. Ex:

```
one = 1   ;this is a constant
two equ 2; also a constant
```

- The EQU and = directives are equivalent
- The assembler does not allocate storage to a constant (in contrast with data allocation directives)
- It merely substitutes, at assembly time, the value of the constant at each occurrence of the assigned name

Constants (cont.)

- In place of a constant, we can use a <u>constant expression</u> involving the standard operators used in HLLs: +, -, *, /
- Ex: the following constant expression is evaluated at assembly time and given a name at <u>assembly time</u>:

```
A = (-3 * 8) + 2
```

- A constant can be defined in terms of another constant:

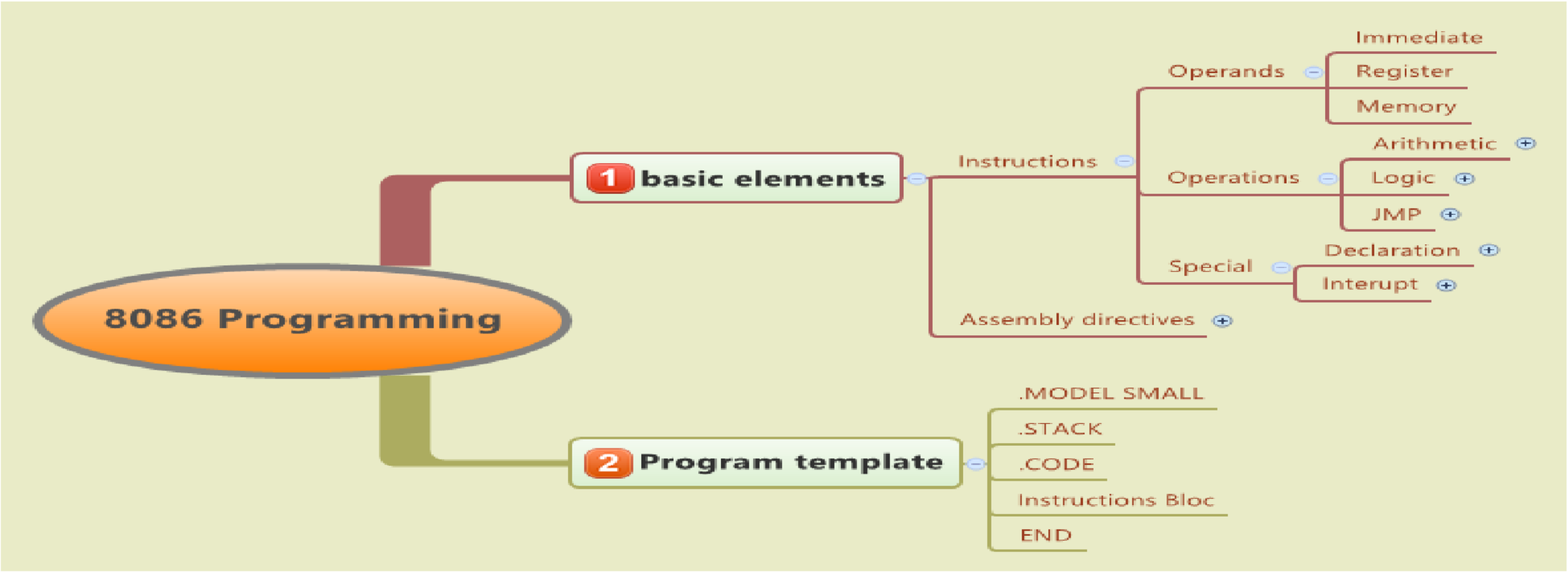
```
B = (A+2)/2
```

8086 Directives

8086 Assembler Directives

- **Assembler is a program used to convert an ALP into equivalent MLP.**
  - It also finds the address of each label.
  - It substitutes the value for each constant and variable
  - It finds out syntax errors and reports them to the programmer.
- **To do so many commands needed by a programmer such as:**
  - Required storage for each constant/variable: Byte, Word, or other.
  - Logical name of segments such as CODE or STACK or DATA segment.
  - Type of different procedures such as FAR, NEAR, PUBLIC or EXTRN
  - End of a segment, End of program, ... etc.
- **These commands are used to support the assembler and ALP instructions.**
  - Such commands need to be defined by the assembler at the assemble time.
  - Predefined alphabetical strings called assembler directives.
  - Called Directives (also called Pseudo-Instructions).

8086 Assembler Directives



- **Assembler directives can be classified as follows:**
  - Group1: Directives for variable and constant definition.
  - Group2: Directives related to code (program) location.
  - Group3: Directives for segment declaration.
  - Group4: Directives for declaring procedure.
  - Group5: Other assembler directives.

Simple Data Allocation Directives

- The DB (define byte) directive allocates storage for one or more-byte values

    [name] DB initval [,initval]

- Each initializer can be any constant. Ex:

```
a db 10, 32, 41h ;allocate 3 bytes
b db 0Ah, 20h,'A';same values as above
```

- A question mark (?) in the initializer leaves the initial value of the variable undefined. Ex:

```
c db ?    ;the initial value for c is undefined
```

- Everything that follows ";" is ignored by the assembler. It is thus a comment

Simple Data Allocation Directives (cont.)

- A string is stored as a sequence of characters. Ex:

```
aString db "ABCD"
bString DB 'A','B','C','D';same values
cString db 41h,42h,43h,44h ;same values again
```

- The (offset) address of a variable is the address of its first byte. Ex: If the following data segment starts at address 0

```
.data
Var1 db "ABC"
Var2 db "DEFG"
```

- The address of Var1 is 0 = the address of 'A'
- The address of 'B' is 1
- The address of 'C' is 2
- The address of Var2 is 3
- The address of 'E' is 4 ...

Simple Data Allocation Directives (cont.)

- **Define Word (DW) allocates a sequence of words. Ex:**

```
A dw 1234h, 5678h ; allocates 2 words
```

- **Intel's x86 are little-endian processors.**
  - This means: the lowest order byte (of a word or doubleword) is always stored at the lowest address.

- **Ex: if variable A (above) is located at address 0, we have:**
  - address: 0 1 2 3
  - value: 34h 12h 78h 56h

Simple Data Allocation Directives (cont.)

- **Define Double Word (DD) allocates a sequence of double words. Ex:**

```
B dd 12345678h ;allocates 1 double word
```

- **If this variable is located at address of 0, we have:**
  - address: 0 1 2 3
  - value: 78h 56h 34h 12h
- **If a value fits into a byte, it will be stored in the lowest ordered byte available. Ex:**

```
V dw 'A'
```

- **the value will be stored as:**
  - address: 0 1
  - value: 41h 00h

Simple Data Allocation Directives (cont.)

- The DUP operator enables us to repeat values when allocating storage (with data allocation directives). Ex:

```
a db 100 dup(?)   ;100 bytes uninitialized
b db 3 dup("Ho") ;6 bytes: "HoHoHo"
```

- DUP can be nested:

```
c db 2 dup('a', 2 dup('b'))
;this allocates 6 bytes:'abbabb'
```

8086 Assembler Directives- Variable/Constant Definition

To Sum Up

- **DB, DW, DD, DQ, DT, directives.**

  Reserve Byte, Word, Double Word, Quad Word, Ten Bytes in memory for storing variables.

- **EQU or =**

  The assembler does not allocate storage to a constant.

- **DUP directive**

  Initialize Several Locations to an Initial Value.

• BYTE	8-BIT
• WORD	16-BIT
• DWORD	32-BIT
• FWORD	48-BIT
• QWORD	64-BIT

Example

- NUMS DB 20
- LIST DB 1, 2, 8, 9, 5

8086 Assembler Directives- Variable/Constant Definition

Example	Comments
DATA1 DB 20H	Reserve one byte to store DATA1 initialized to 20H.
ARRAY1 DB 10H,20H,30H	Reserve 3 bytes to store ARRAY1 initialized with 10H, 20H, 30H
CITY DB "DAMMAM"	Reserve a list named CITYT initialized with Chars' ASCII codes.
DATA2 DW 1020H	Reserve one word to store DATA2 initialized to 1020H.
NUMBER EQU 50H	Assign the value 50H to NUMBER
NAME EQU "QASEM"	Assign the string "QASEM" to NAME
START DW 4 DUP (0)	Reserves 4 words starting at offset START in DS initialized to 0.
BEGIN DB 100 DUP (?)	Reserves 100 bytes of uninitialized data to offset BEGIN in DS.
X DW 2A05H Y DW 052AH PRODUCT EQU X * Y	Using Expressions
SUNDAY EQU 1 MONDAY EQU SUNDAY + 1	Using Expressions

Example (1): Variable Definition

- VAL1 DB 10
- VAL2 DB 0AH
- ARRAY1 DB 3, 5, 1, 0
- CHAR1 DB "A" ; SINGLE QOUTEA ARE OK TOO
- VAL3 DB ?
- STR1 DB "Hello World"
- Msg1 DB "welcome", 0Ah, 0DH
- VAL4 DW 90A1H, 0FH
- BIG DD 11223344H
- LIST DB 2, 0, 1
  DB 10
  DB 1

See the representation of data in memory – next slide

Representing the data in memory

ADDRESS	CONTENTS	
0	10	VAL1
1	0A	VAL2
2	3	ARRAY1
3	5	
4	1	
5	0	
6	“A”	CHAR1
7	-	VAL3
8	“H”	STR1
9	“E”	
A	“L”	
B	“L”	
C	“O”	
D	SP	
E	“W”	
F	“O”	
10	“R”	
11	“L”	
12	“D”	
13	“W”	MSG1
14	“E”	
15	“L”	
16	“C”	
17	“O”	
18	“M”	
19	“E”	
1A	0A	
1B	0D	
1C	A1	VAL4
1D	90	
1E	0F	
1F	00	
20	44	BIG
21	33	
22	22	
23	11	
24	2	LIST
25	0	
26	1	
27	10	
28	1	
29		
2A		
2B		
2C		
2D		
2E		
2F		

Example (2): Using DUP Operator (For Arrays)

- ARR1 BYTE 20 DUP(0) ; 20 bytes, all equal to zero
- ARR2 DB 20 DUP (0); SAME AS ABOVE
- LIST1 DB 20 DUP(?) ; 20 bytes, uninitialized

Exercise

- Suppose the following data segment starts at default:
    - **.data**
    - **A DW 1,2**
    - **B DW 6ABCh**
    - **Z EQU 232**
    - **C DB 'ABCD'**

  - A) Find the address of variable A.
  - B) Find the address of variable B.
  - C) Find the address of variable C.
  - D) Find the address of character ‘C’.

Example (3): Working with constants

```
COUNT = 5
mov al, COUNT ; AL = 5
COUNT = 10
mov al, COUNT ; AL = 10
COUNT = 100
mov al, COUNT ; AL = 100
```

8086 Assembler Directives-Related to Code Location.

- **ORG (ORIGIN) Directive.**

  Tells the assembler where to load instructions and data into memory.

  Initialize CS and IP with initial address (logical) as a starting address.

  If its not mentioned at the start of segment➔ Offset is initialized to 0000H.

- **Example: ORG 0100H**

  The first instruction is stored from at offset 0100H within the code segment.

- **OFFSET and SEG Directives.**

  Used to determine the Offset and Segment addresses of a given data item.

- **Example: MOV BX, OFFSET TABLE / MOV AX, SEG ARRAY1**

- **EVEN Directive.**

  Used to declare a data item to start at even memory address.

- **Example: EVEN / ARRAY2 DW 20 DUP (0)**

Using pointers to access memory

- You can use any of the pointers in the data segment to access your data and arrays such as BX, SI, DI.

- Assume we have the following array:

  Nums db 2, 1, 5, 0, 1 ; array contains 5 elements

1. Use a pointer BX to point at the first address in the array:

   Mov BX, offset nums or LEA BX, nums

2. Start a loop and access the contents of the array using [BX]:

   Mov AL, [BX]

3. move the pointer to the next location using:

   INC BX

4. repeat the loop until you finish all the 5 elements

Example: Accessing the contents of an Array

```
.DATA
Nums db 2, 1, 5, 0, 1                       ; array contains 5 elements
.CODE
          MOV CX, 5                         ; counter for the loop
          MOV BX, OFFSET NUMS               ; let BX points to first location in Nums
LOOP1 :   MOV AL, [BX]                      ; access location in Nums pointed at by BX
          INC BX                            ; let BX point to the next location in Nums
          DEC CX                            ;  subtract 1 from the counter
          JNZ LOOP1                         ; repeat the loop until CX=0
```

Another Example: Access arrays without using offset

In this example, we will access the memory using another method, Just like Higher-level Languages, and using any pointer (BX, DI, SI)

```
.DATA
Nums db 2, 1, 5, 0, 1               ; array contains 5 elements
.CODE
            MOV CX, 5               ; counter for the loop
            MOV BX, 0               ; initialize BX to Zero... It will be the index for the array
LOOP1 :     MOV AL, Nums[BX]        ; access location in nums pointed at by BX
            INC BX                  ; let BX point to the next location in Nums
            DEC CX                  ;  subtract 1 from the counter
            JNZ LOOP1               ; repeat the loop until CX=0
```

8086 Assembler Directives- For Segment Declaration.

- **SEGMENT and ENDS directives.**

  Indicate the Start & End of a logical segment (Segment name ≤ 31 characters).

```
Segnam SEGMENT
    ...
    ...
    ...
    ...
    ...
    ...
Segnam ENDS
```

- Example:

```
START  SEGMENT
    X1  DB    F1H
    X2  DB    50H
    X3  DB    25H
START  ENDS
```

Programmer must then use 8086 instructions to load START into DS, such as:

```
MOV    BX, START
MOV    DS, BX
```

- **ASSUME directive.**

  Links the logical segments with the declared segment names.

- Example 1:

```
CODE      SEGMENT
ASSUME    CS:CODE, DS:CODE, ES:CODE, SS:CODE
⋮
CODE      ENDS
```

- Example 2: ASSUME CS : PROGRAM_1, DS : DATA_1, SS : STACK_1

8086 Assembler Directive- Procedures Declaration.

- **PROC and ENDP directives.**

  Indicates the start and the end of a named procedure (NEAR or FAR).

- **Example1:**

```
SQUARE_ROOT  PROC        NEAR
        ⋮                  ⋮
SQUARE_ROOT  ENDP
```

  Define a procedure “SQUARE_ROOT”, which is to be called by a program located in the same segment (Near).

- **Example2:**

```
SQUARE_ROOT  PROC        FAR
        ⋮                  ⋮
SQUARE_ROOT  ENDP
```

  Define a procedure “SQUARE_ROOT”, which is to be called by a program located in another segment (Far).

8086 Assembler Directive- Macros Declaration.

- **MACRO and ENDM directives.**

  Indicates the start and the end of a named MACRO (Can take parameters).

- Example 1 :

```
CALCULATE    MACRO
             MOV AX, [BX]
             ADD AX, [BX+2]
             MOV [SI], AX
             ENDM
```

Can be used any time in the main program, just use its name

Example 2 :

```
CALCULATE    MACRO  OPERAND,  RESULT
             MOV BX, OFFSET OPERAND
             MOV AX, [BX]
             ADD AX, [BX+2]
             MOV SI, OFFSET RESULT
             MOV [SI], AX
             ENDM
```

Parameters OPERAND and RESULT can be replaced by OPERAND1, RESULT1, and OPERAND2, RESULT2 while calling the above macro as shown below:

```
............................
CALCULATE          OPERAND1, RESULT1
...........................
..........................
CALCULATE          OPERAND2, RESULT2
...........................
```

8086 Assembler Directives-Other Directives.

- **PTR (Pointer) directive.**

  Used to declare the type of memory operand (prefixed by BYTE or WORD).

- **Examples: INC BYTE PTR [SI] / INC WORD PTR [BX].**

- **NAME directive.**

  Used to assign a name to an assembly language program module.

- **Examples:** NAME “Hi-World”

- **TYPE directive.**

  Return the data type used to define a specific data (Word 2, Double 4, Byte 1).

- **Example: MOV BX, TYPE DATA1.**

- **LENGTH Directive (or $ operator).**

  Used to determine the length of an array in bytes .

- **Example: MOV CX, LENGTH ARRAY**

- **See other directives such as:**

  SHORT, LABEL, GROUP, EXTRN & PUBLIC, GLOBAL & LOCAL

Example: Using $ operator to calculate the size of arrays/lists

- **Example (1)**
  - List db 1, 5, 2, 8, 9, 10, 3, 1
  - List_size = ($ - list) ;;;; list equals 8

- **Example (2)**
  - myString "This is a long string, containing"
  - myString_len = ($ - myString) ;;;; list equals 33

More Examples

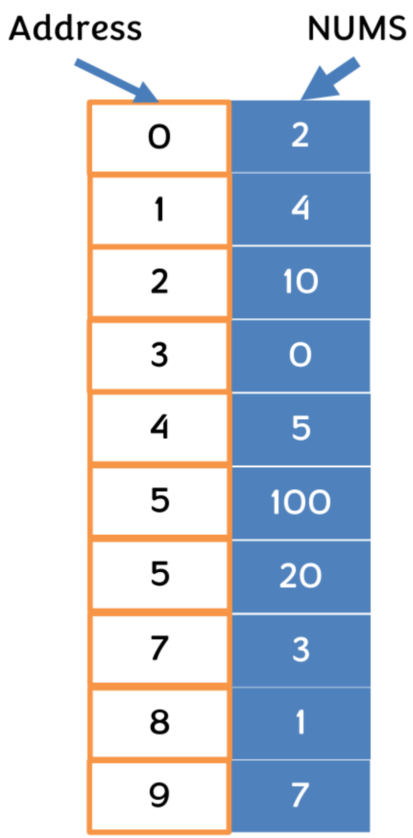
Example

Declare an array NUMS with ten 8-bit numbers, then write the code to add 2 to each number stored in NUMS

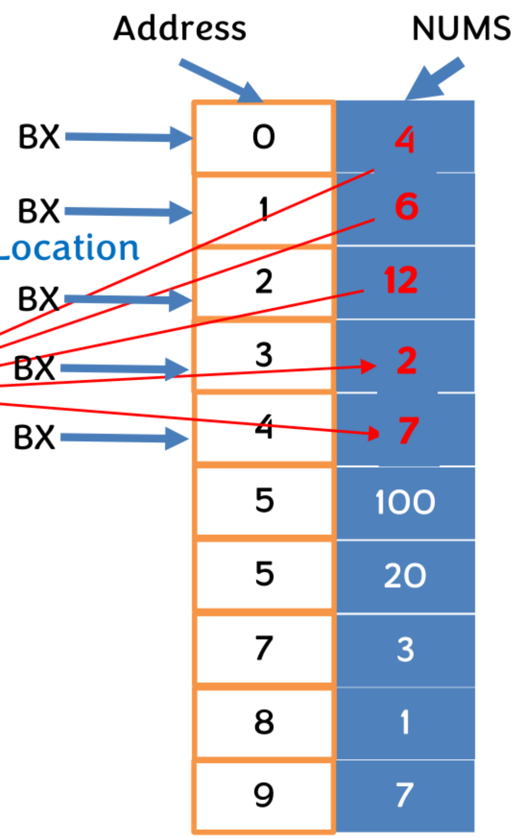
Solution 1

Declare the array in the DATA SEGMENT

```
.DATA
NUMS DB 2, 4, 10, 0, 5, 100, 20, 3, 1, 7
```



```
.CODE
        MOV CX, 10              ; Loop Counter
        MOV BX, OFFSET NUMS ; BX points at first Location
        MOV AL, 2               ;Number to be added
Next: ADD [BX], AL      ;Add 2 to array contents
        INC BX                  ;Point to next location
        DEC CX                  ;If we are not done
        JNZ NEXT                ; keep repeating
```



Main Sources for these slides

- *K. R. Irvine. Assembly Language for x86 Processors, 8th edition, Prentice-Hall (Pearson Education), June 2019. ISBN: 978-0135381656.*

- *B. Dang, A. Gazet, E. Bachaalany. Practical Reverse Engineering: x86, x64, ARM, Windows® Kernel, Reversing Tools, and Obfuscation. John Wiley & Sons, June 2014. ISBN: 978-1-118-78731-1*

- *Qasem Abu Al-Haija, "Microprocessor Systems", King Faisal University, Saudi Arabia*

- *Ghassan Issa, "Computer Organization", Petra University, Jordan.*

Thank you