LAB01: 8086 introduction

CLASS MODE	LABINF
Date	@October 4, 2021
Edited	☐
Exam Review	☐
Hours	1.5
Last Edit	@October 24, 2021 8:27 PM
SLIDES	
TOPIC	8086
WEEK	2

Introduction

Processor Status Word (PSW)

It is a 16-bit register (but only 9 bits are used) where every single bit is a flag. The **flags register** maintains the current operating mode of the CPU and some instruction state information. The carry, parity, zero, sign, and overflow flags are special because you can test their status (zero or one) with the `setcc` and conditional jump instructions. The 80x86 uses these bits, the condition codes, to make decisions during program execution. Various arithmetic, logical, and miscellaneous instructions affect the overflow flag. After an arithmetic operation, this flag contains a one if the result does not fit in the signed destination operand. For example, if you attempt to add the 16 bit signed numbers 7FFFh and 0001h the result is too large so the CPU sets the overflow flag. If the result of the arith- metic operation does not produce a signed overflow, then the CPU clears this flag.

```
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    | OF | DF | IF | TF | SF | ZF |    | AF |    | PF |    | CF |
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
```

Condition Flags

Condition Flags are automatically set at the end of some instructions:

- **SF** (**Sign Flag**): MSB of the result after an aritmethic instruction;
- **ZF** (**Zero Flag**): it is 1 if the result is zero, 0 otherwise. Various instructions set the zero flag when they generate a zero result. You'll often use this flag to see if two values are equal (e.g., after subtracting two numbers, they are equal if the result is zero). This flag is also useful after various logical operations to see if a specific bit in a register or memory location contains zero or one.

- **PF** (**Parity Flag**): it is 1 if the result has an even number of bits set to 1, 0 otherwise. The parity flag is set according to the parity of the L.O. eight bits of any data operation. If an operation produces an even number of one bits, the CPU sets this flag. It clears this flag if the operation yields an odd number of one bits.
- **CF** (**Carry Flag**): it is 1 in presence of an arithmetic carry or borrow with unsigned arithmetic instructions. The carry flag has several purposes. First, it denotes an unsigned overflow (much like the overflow flag detects a signed overflow). You will also use it during multiprecision arithmetic and logical operations. Certain bit test, set, clear, and invert instructions on the 80386 directly affect this flag. Finally, since you can easily clear, set, invert, and test it, it is useful for various boolean operations. The carry flag has many purposes and knowing when to use it, and for what purpose, can confuse beginning assembly language program- mers. Fortunately, for any given instruction, the meaning of the carry flag is clear.
- **AF** (**Auxiliary Carry Flag**): in BCD arithmetic, it is 1 with a carry or borrow of the third bit;
- **OF** (**Overflow Flag**): it is 1 in presence of an overflow with signed arithmetic instructions.

```
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    | OF | DF | IF | TF | SF | ZF |    | AF |    | PF |    | CF |
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
```

Control Flags

Control Flags can be written and manipulated by specific instructions, and are used to regulate the functioning of certain processor functions:

- **DF** (**Direction Flag**): used by the instructions for string manipulation; if it is 0, the strings are manipulated starting from the characters at the lower address, if it is 1 starting from the largest address. The 80x86 string instructions use the direction flag. When the direction flag is clear, the 80x86 processes string elements from low addresses to high addresses; when set, the CPU processes strings in the opposite direction
- **IF** (**Interrupt Flag**): if it is 1, the maskable Interrupt signals are managed by the CPU, otherwise these are ignored. The interrupt enable/disable flag controls the 80x86's ability to respond to external events known as interrupt requests. Some

programs contain certain instruction sequences that the CPU must not interrupt. The interrupt enable/disable flag turns interrupts on or off to guarantee that the CPU does not interrupt those critical sections of code.

- **TF** (**Trap Flag**): if it is 1, a trap is executed at the end of each instruction.

```
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    | OF | DF | IF | TF | SF | ZF |    | AF |    | PF |    | CF |
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
```

Emu8086

Instructions

- `.model` indicates which memory model is used. As an example, with `.model tiny` you get a program where `CS`, `DS`, and `SS` are all pointing to the same 64KB of memory. The stack is placed in the highest region of this 64KB segment. With `.model small` you get a program where `CS` points to a segment of its own, followed by the segment where `DS` and `SS` are pointing to. The stack is placed in the highest region of the `SS` segment.

Memory model	*Code*	*Data*	*Combined code and data*
TINY	NEAR	NEAR	YES
SMALL	NEAR	NEAR	NO
MEDIUM	FAR	NEAR	NO
COMPACT	NEAR	FAR	NO
LARGE or **HUGE**	FAR	FAR	NO

- `.stack {mem}` creates the stack (default size is 1Kbyte). Because 80x86 is a 16-bits processor, it has difficulties in managing memories larger than 64KB.
- `.data` creates the data segment;
- `.code` creates the code segment;
- `.startup` and `.exit` produce the machine instructions needed for executing the program in a virtual MS-DOS environment.

Exercise

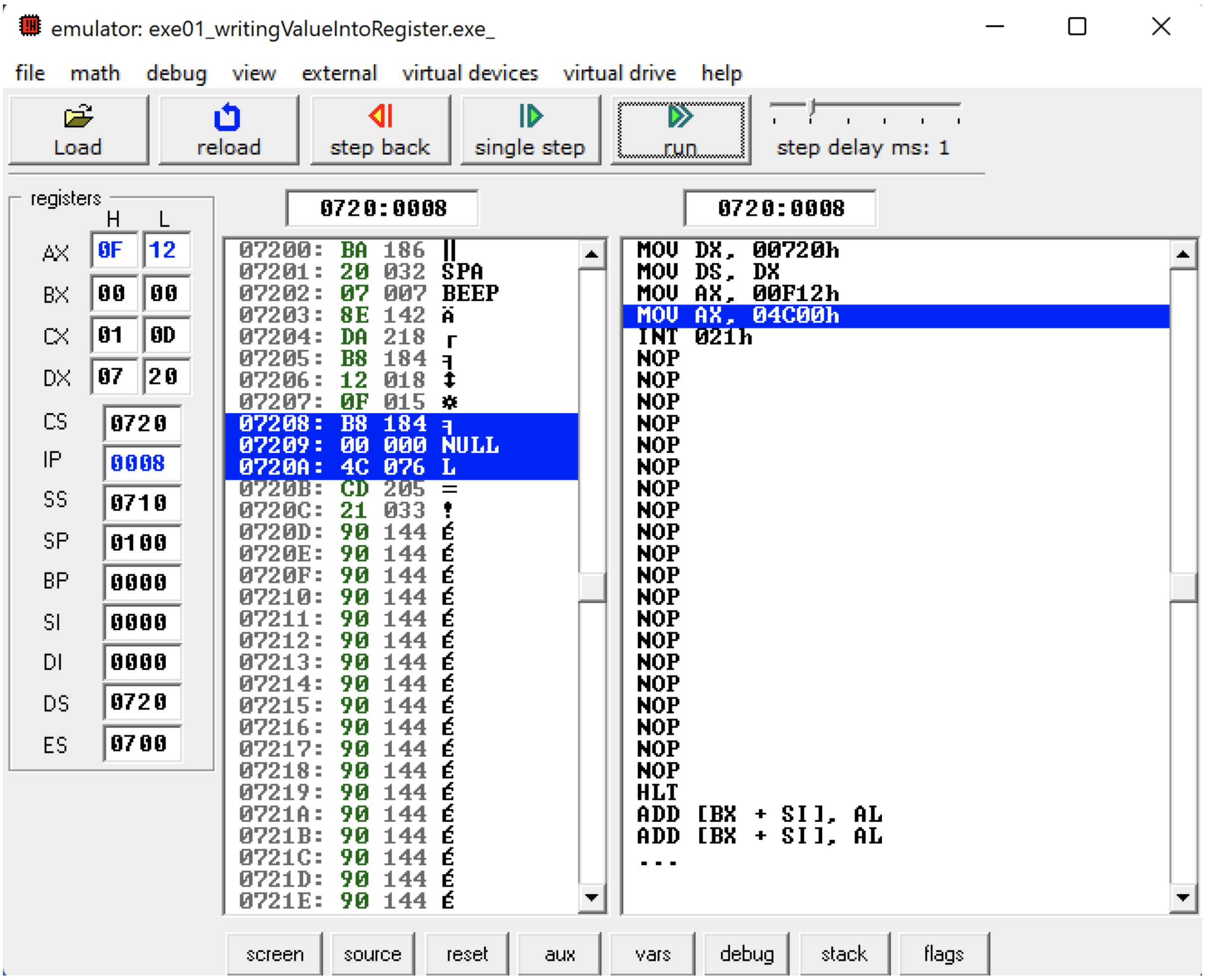
Writing a value in a register

Code

```
.MODEL small
.STACK
.DATA
.CODE
.STARTUP
    MOV AX, 0F12       ; Storing in AX the hexadecimal value 0F12H,
                       ; which corresponds to 3858
.EXIT
END
```

Analysis

At the end of the execution, the `AX` register will be storing the hexadecimal value `0F12H`.



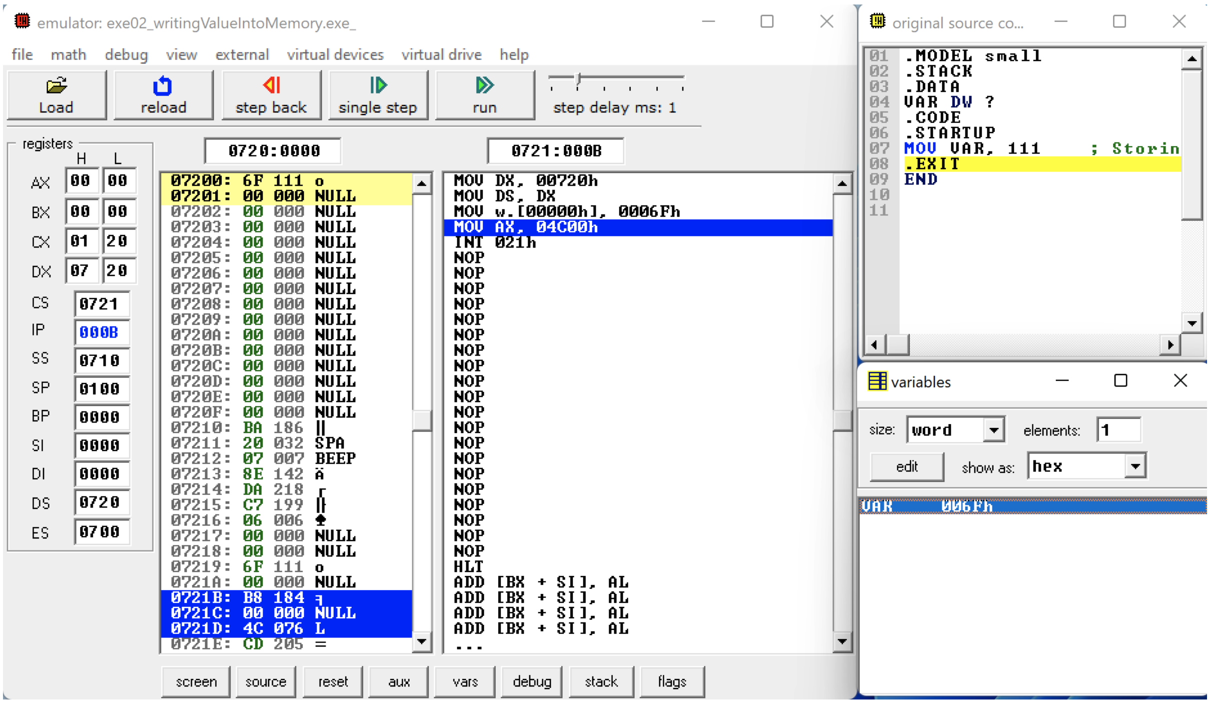
Writing a value in a memory cell

Code

```
.MODEL small
.STACK
.DATA
    VAR DW ?            ; Defining a Data Word (16-bit) variable
.CODE
.STARTUP
    MOV VAR, 111        ; Storing 006FH into VAR
.EXIT
END
```

Analysis

At the end of the execution, the variable VAR will be storing the unsigned value 111 (in hexadecima: `006FH`).



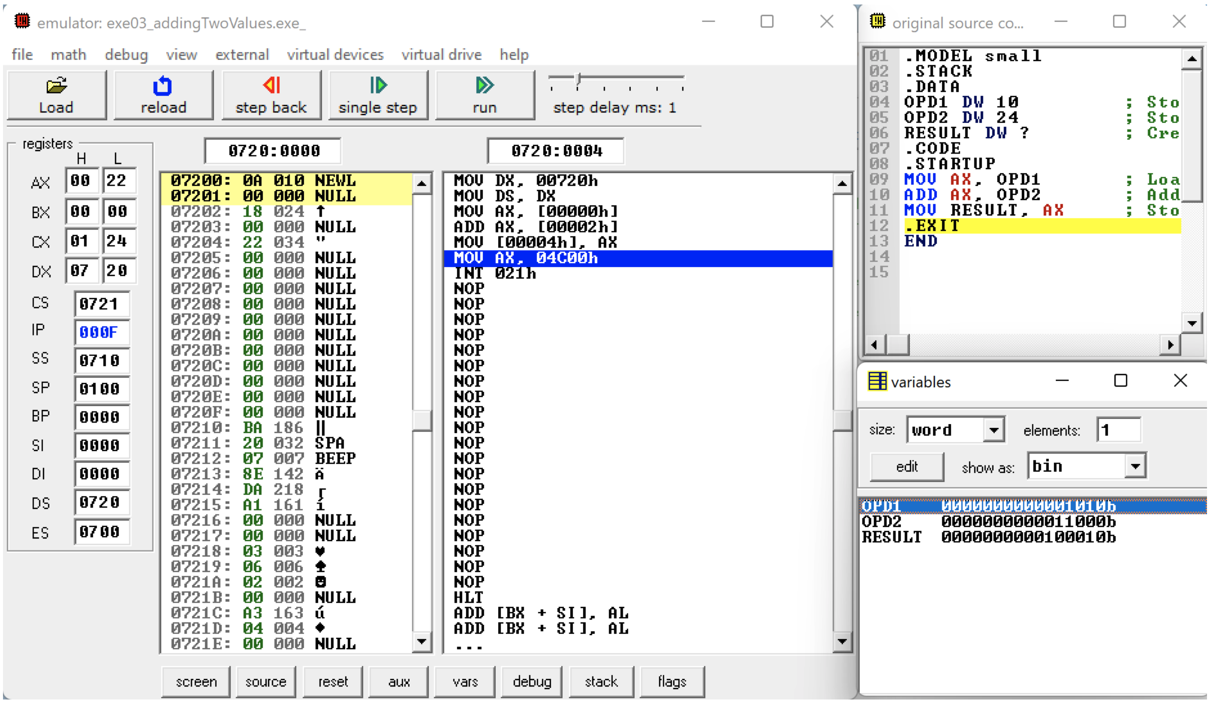
Adding two values

Code

```
.MODEL small
.STACK
.DATA
    OPD1 DW 10        ; Defining the first operand
    OPD2 DW 24        ; Defining the second operand
    RESULT DW ?       ; Creating a variable to store the result
.CODE
.STARTUP
    MOV AX, OPD1      ; Starting the addition by loading the first operand in AX
    ADD AX, OPD2      ; Adding the second operand
    MOV RESULT, AX    ; Storing the result in the RESULT variable
.EXIT
END
```

Analysis

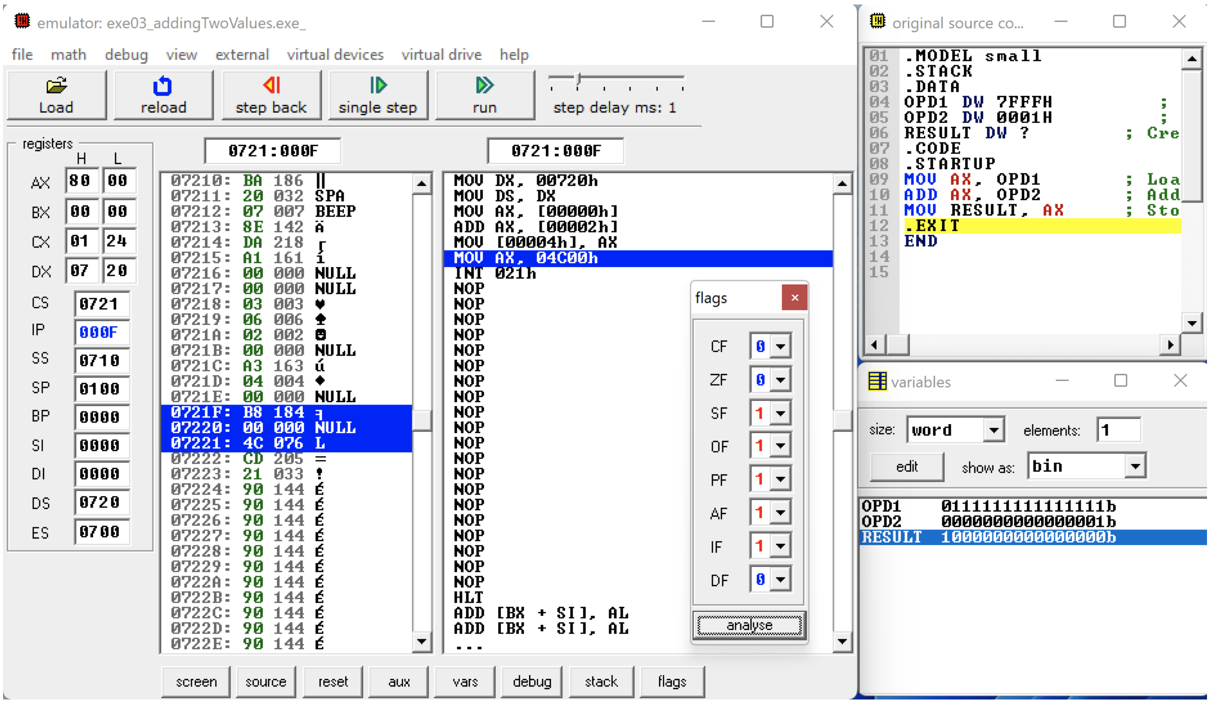
At the end of the execution, the variable RESULT will be storing the sum of the two operands variable (i.e., in decimal $10 + 24 = 34$, which corresponds to `0022H`). From the picture, it is easy to see the binary sum bit-to-bit.



On the other hand, suppose to assign the following values to the operands (remember that a Data Word, i.e. `DW`, is 16 bits variable):

- `OPD1 DW 7FFFH`, which corresponds to 32767 in signed decimal.
- `OPD2 DW 0001H`, which corresponds to 1 in signed decimal.

Therefore, the sum of the two value will generate a signed overflow ($32767 + 1 = -32768$) which will be visible thanks to the Overflow Flag (OF), which will be set to 1.



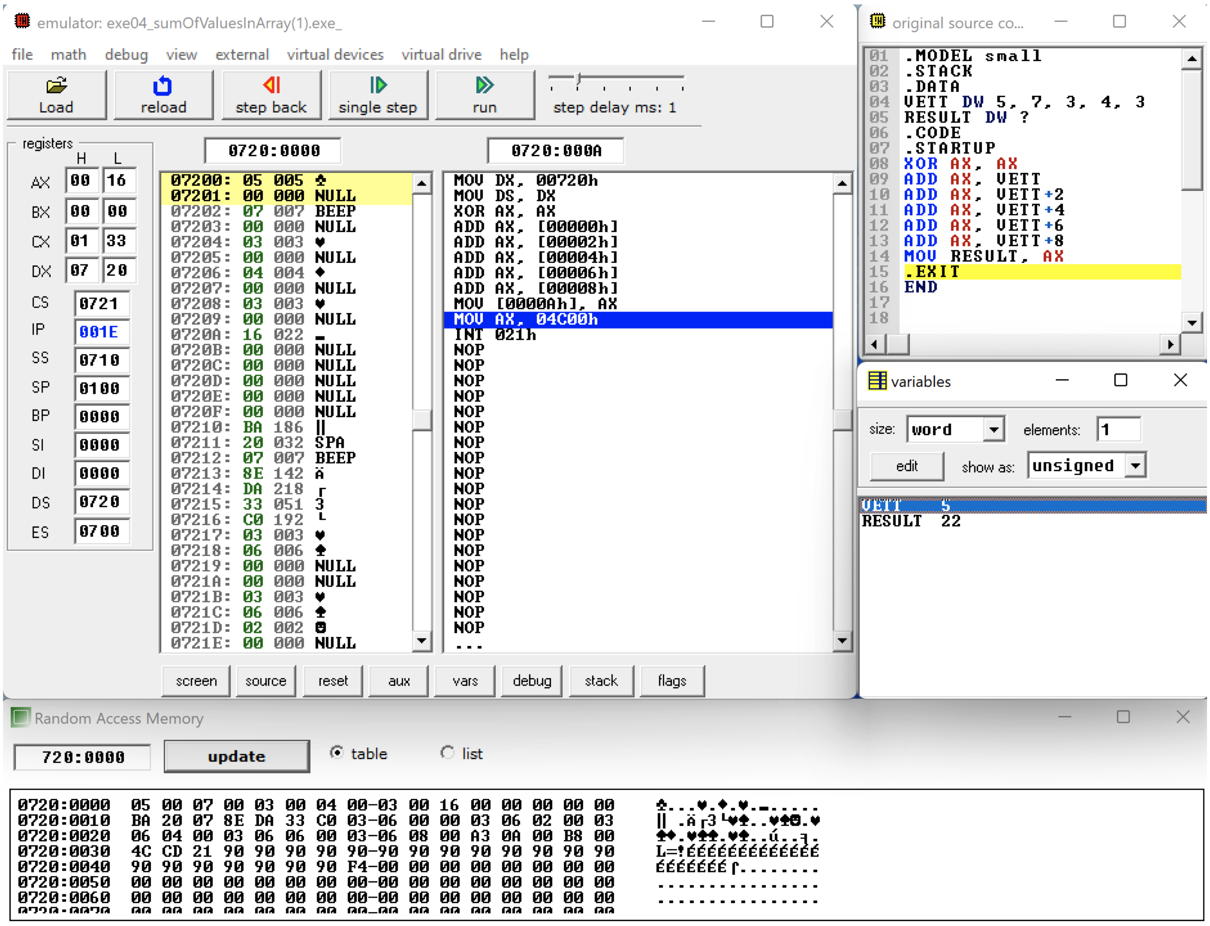
Sum of elements in array (I)

Code

```
.MODEL small
.STACK
.DATA
    VETT DW 5, 7, 3, 4, 3       ; Defining the array values
    RESULT DW ?                 ; Creating a variable to store the result
.CODE
.STARTUP
    XOR AX, AX                  ; Zeroing AX
    ADD AX, VETT                ; Starting the sum (manually)
    ADD AX, VETT+2
    ADD AX, VETT+4
    ADD AX, VETT+6
    ADD AX, VETT+8
    MOV RESULT, AX             ; Storing the result in the RESULT variable
.EXIT
END
```

Analysis

At the end of the execution, the variable `RESULT` will be storing the sum (computed manually by accessing directly to the `VETT` position in memory (`0720:0000`)).



Sum of elements in array (II)

Code

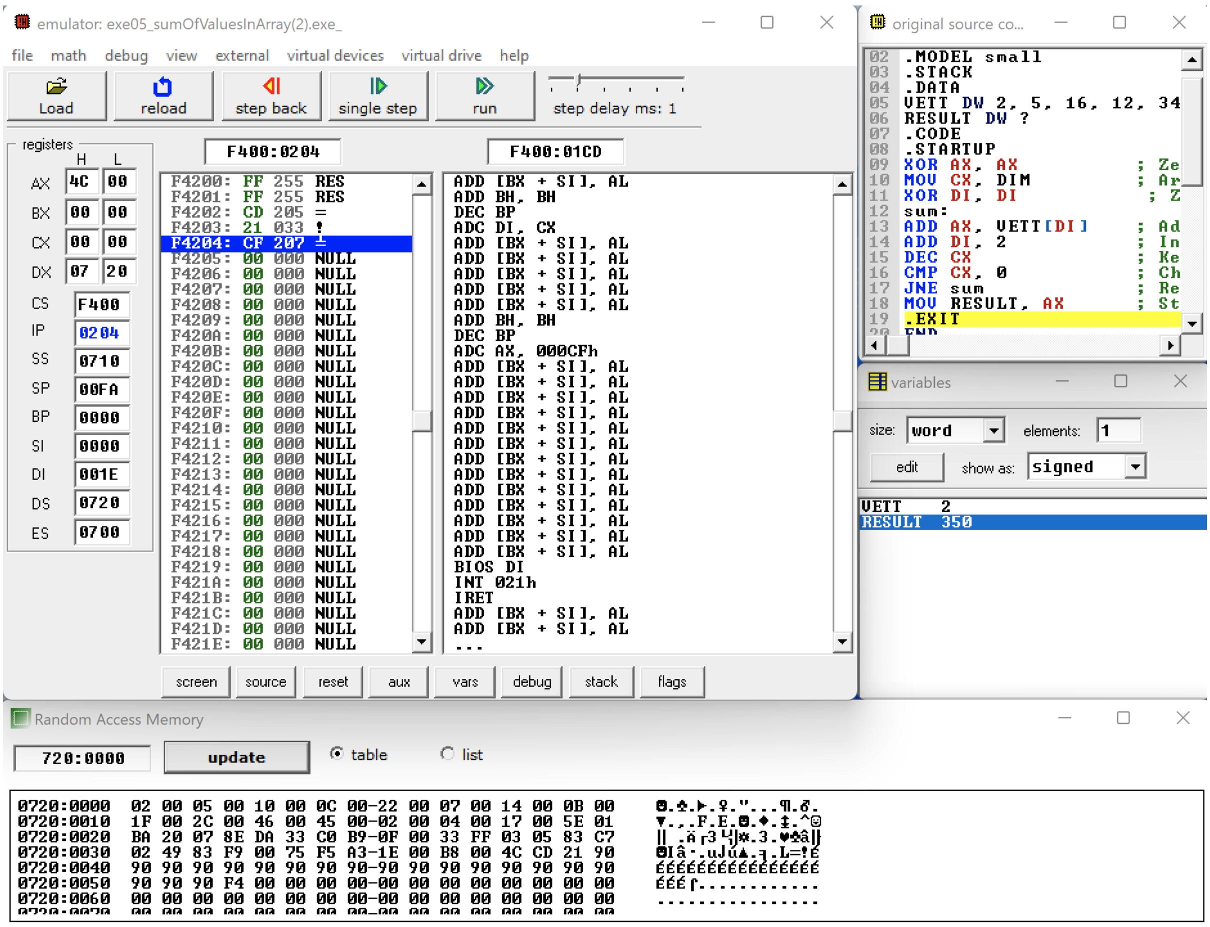
```
DIM EQU 15
.MODEL small
.STACK
.DATA
    VETT DW 2, 5, 16, 12, 34, 7, 20, 11, 31, 44, 70, 69, 2, 4, 23
    RESULT DW ?
.CODE
.STARTUP
    XOR AX, AX          ; Zeroing AX
    MOV CX, DIM         ; Array size is now stores in CX
    XOR DI, DI          ; Zeroing DI
sum:
    ADD AX, VETT[DI]    ; Add to AX the i-th element of VETT
    ADD DI, 2           ; Incrementing the pointer
    DEC CX              ; Keeping track of the number of operations left
    CMP CX, 0           ; Checking if the array has been fully scanned
    JNE sum             ; Reiterating if ZF = 0 (CX != 0 ---> CMP result is != 0)
    MOV RESULT, AX      ; Storing the result
```

```
.EXIT
END
```

Analysis

The sum of the values stored in the array is now computed automatically thank to the introduction of the **loop**. A loop is composed of:

- A `label`: for example, the `sum` label, which defines the snippet of code which can be directly accessed by a control flow instruction.
- A control flow instruction: such as `JNE`, which allows the processor to jump to a particolar part of the code.
- A counter: such as `CX`, which allows to count the number of operations performed (and, therefore, the number of operation left).
- A `CMP` instruction: which checks that the number of operations left is zero and the loop in concluded.



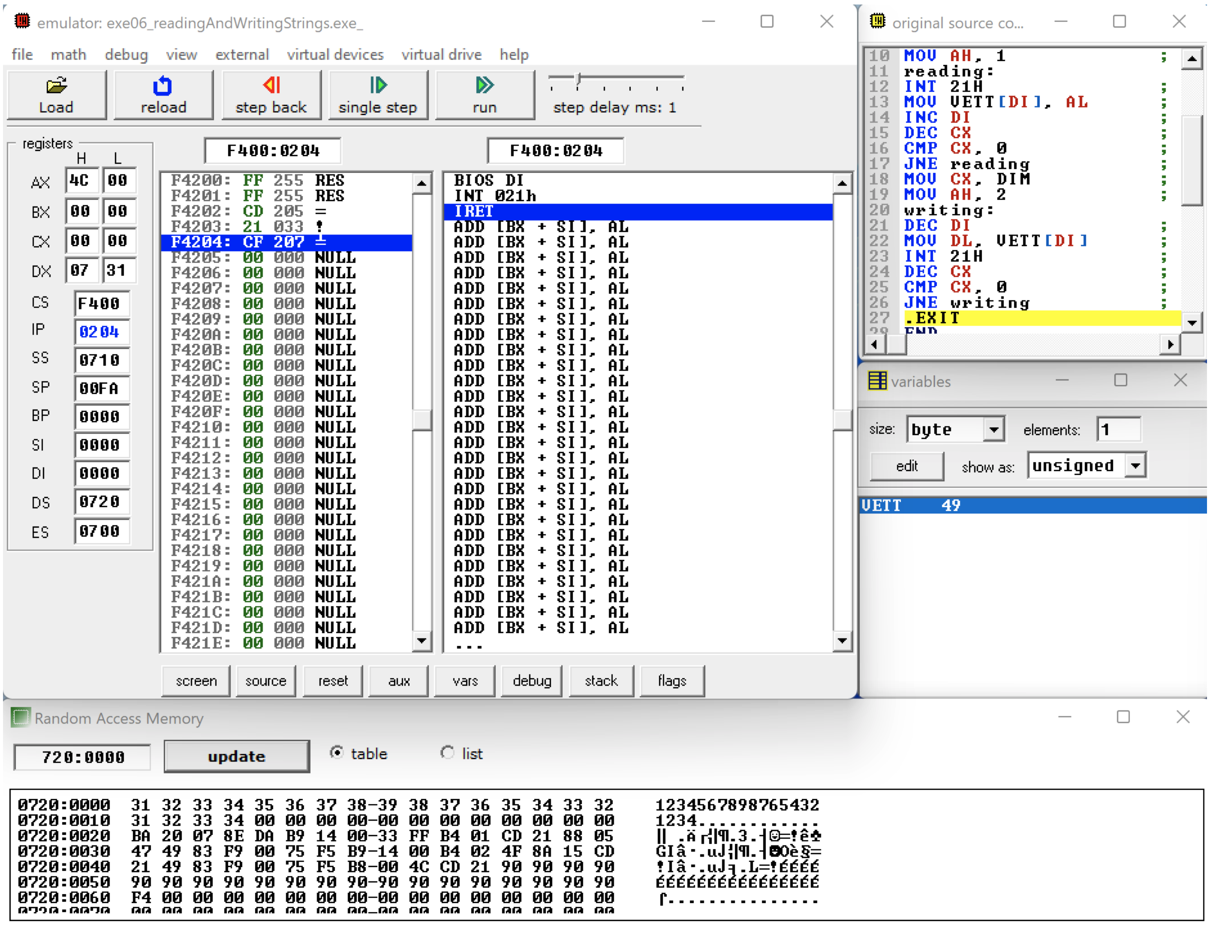
Read and display a character array

Code

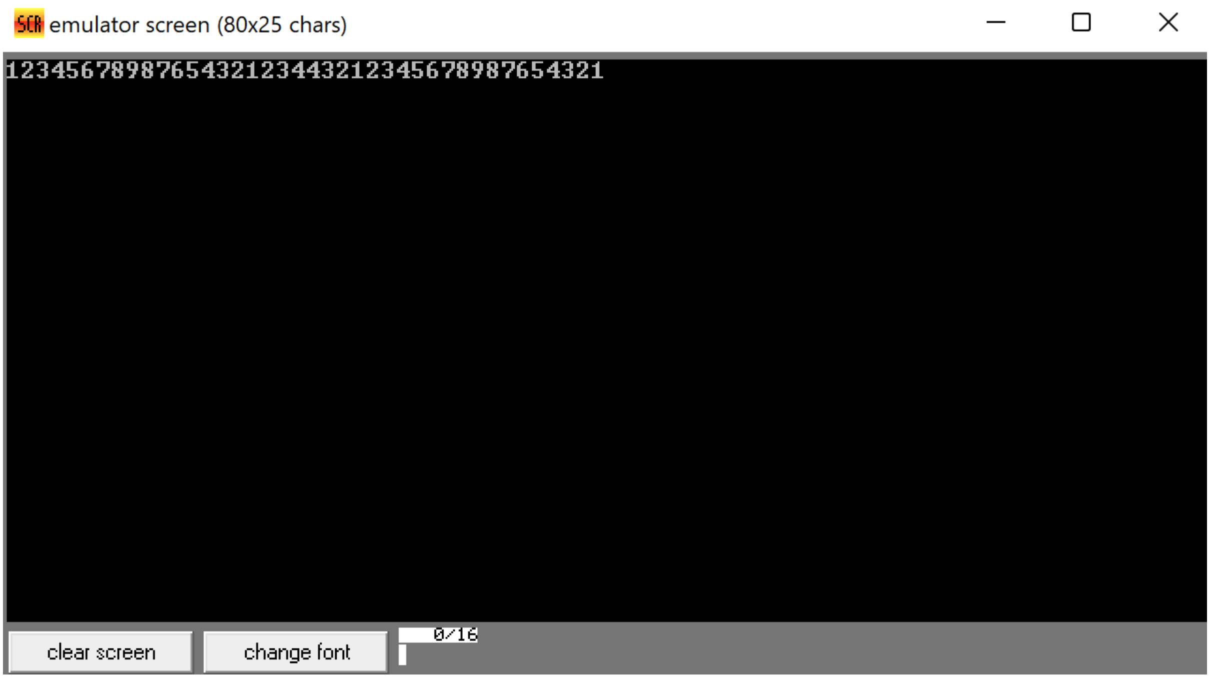
```
DIM EQU 20
.MODEL small
.STACK
.DATA
    VETT DB DIM DUP(?)
.CODE
.STARTUP
    MOV CX, DIM           ; Storing in the counter the number of operations to be perf
ormed by the loop
    XOR DI, DI            ; Zeroing DI
    MOV AH, 1             ; Set AH for reading
reading:
    INT 21H               ; Reading a character
    MOV VETT[DI], AL      ; Storing read character
    INC DI                ; Incrementing index
    DEC CX                ; Decrementing counter to keep track of operations left
    CMP CX, 0             ; Performing <CX> - 0. If <CX> = 0 --> CF = 1, else CF = 0
    JNE reading           ; Repeat if <CX> != 0 (i.e. CF = 0)
    MOV CX, DIM           ; Recharging the dimension for writing
    MOV AH, 2             ; Set AH to writing
writing:
    DEC DI                ; Starting from the last position (printing backwards)
    MOV DL, VETT[DI]      ; Storing character to write
    INT 21H               ; Writing a character
    DEC CX                ; Decrementing counter to keep track of operations left
    CMP CX, 0             ; Performing <CX> - 0. If <CX> = 0 --> CF = 1, else CF = 0
    JNE writing           ; Repeat if <CX> != 0 (i.e. CF = 0)
.EXIT
END
```

Analysis

The `reading` part will read from the user input the values to store in the `VETT` variable (composed of cells of dimensions Data Byte, i.e. 8 bits). As an example, `VETT` will store in hexadecimal `31H` which corresponds to the ASCII character '1', then will store in hexadecimal `32H` which corresponds to the ASCII character '2', and so on...



The program will then display the string passed as input in backwards (since the `writing` snippet of code will start considering `DI` from the last position and then decrementing it (`DEC DI`)):



Search for the minimum character

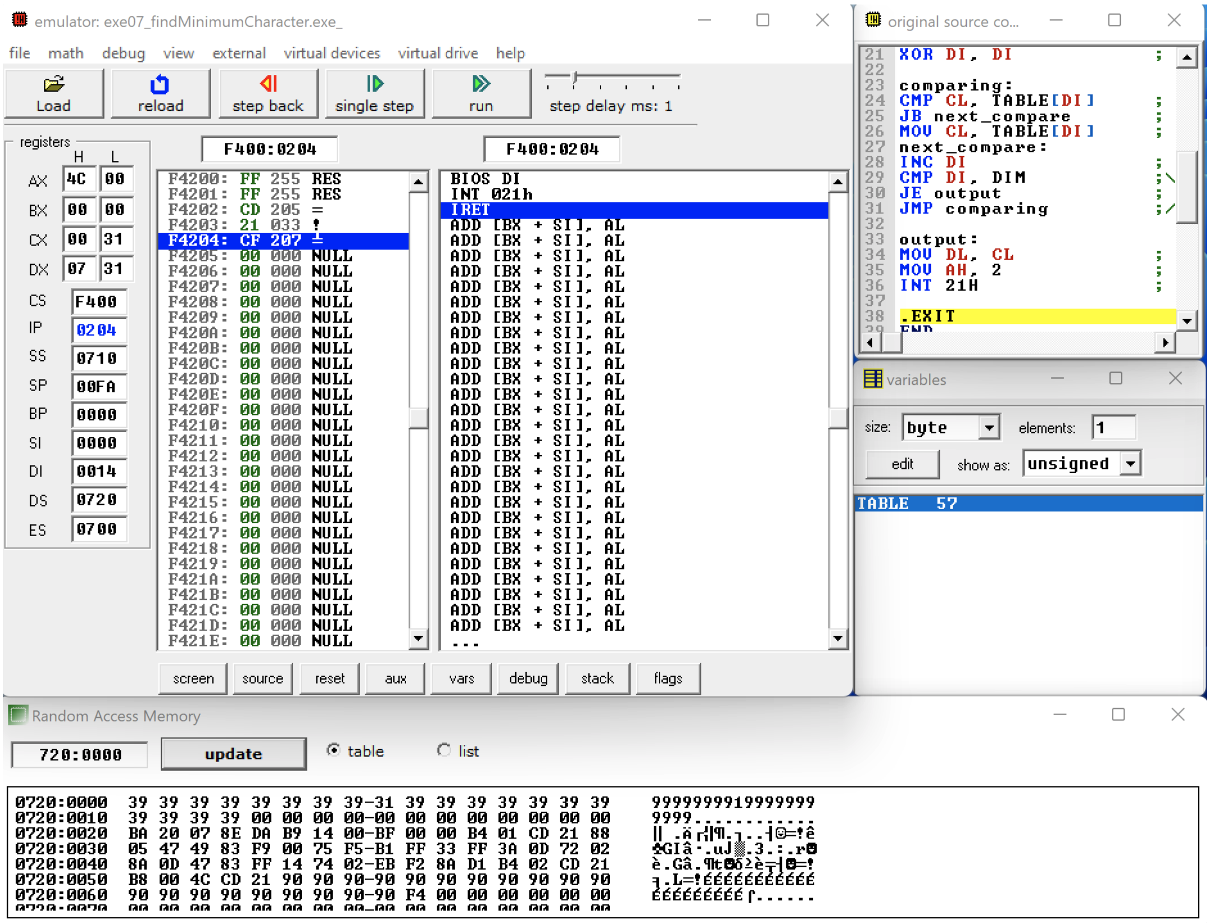
Code

```
.MODEL small
.STACK
    DIM EQU 20
.DATA
    TABLE DB DIM DUP(?)
.CODE
.STARTUP

    MOV CX, DIM             ; Storing the dimension to perform checkings
    LEA DI, TABLE           ; Load TABLE in DI
    MOV AH, 1               ; Setting AH to read

reading:
    INT 21H                 ; Reading a character
    MOV [DI],  AL           ; Storing the read character in the table
    INC DI                  ; Moving index of TABLE
    DEC CX                  ;\
    CMP CX, 0               ; } while(CX != 0) {keep reading}
    JNE reading             ;/
    MOV CL, 0FFH            ; Setting current minimum to 255_dec
    XOR DI, DI              ; Zeroing index

comparing:
    CMP CL, TABLE[DI]       ; Compare with current minimum
    JB next_compare         ; New minimum not found --> skip
    MOV CL, TABLE[DI]       ; New minimum found     --> storing new minimum
next_compare:
    INC DI                  ; Moving index
    CMP DI, DIM             ;\
    JE output               ; } if (DI == DIM) {output} else {keep comparing}
    JMP comparing           ;/

output:
    MOV DL, CL              ; Preparing output
    MOV AH, 2               ; Setting AH to write
    INT 21H                 ; Write result

.EXIT
END
```

Analysis

The code will read (`reading` part of the code) in input a list of 20 characters and store them in the `TABLE` variable (with cells of dimensions Data Byte, i.e. 8 bits). As previously, the hexadecimal value `39H` corresponds to the ASCII character '9'. Then the processor will scan the `TABLE` (`comparing` and `next_compare` part of the code) in order to search the minimum value, following the algorithm:

```
while (DI != DIM) {
  if (TABLE[DI] < CL) {
    CL = TABLE[DI] // New minimum
  } else {
    // Next compare
  }
  DI = DI + 1   // INC DI
}
```



The result is the following (where the last character is prompted by the code):


emulator screen (80x25 chars)
9999999919999999999991
clear screen
change font
0/16