Name : Tazmeen Afroz

Roll No : 22P-9252

```
; a program to add three numbers using byte variables
[org 0x0100]

   ; initialize stuff
   mov  ax, 0            ; reset the accumulator
   mov  bx, 0            ; set the counter

   outerloop:
      add  ax, [num1 + bx]
      add  bx, 2

      cmp bx, 20         ; sets ZF=0 when they are equal
      jne  outerloop


   mov  [result], ax

   mov  ax, 0x4c00
   int  0x21


   ; Intel Sotware Developer Manual - EFLAGS and Instructions (Page 435)

num1: dw   10, 20, 30, 40, 50, 10, 20, 30, 40, 50
result: dw 0
```

Code Explanation:

The given code is a program to add three numbers using byte variables.

First, the program initializes the accumulator (ax) and the counter (bx) to zero . This is done using the "mov" instruction.

Next, the program enters a loop labeled as "outerloop". In each iteration of the loop, the program adds the value at the memory location [num1 + bx] to the accumulator (ax) using the "add" instruction . The value at [num1 + bx] is accessed by adding the index bx to the base address of the num1 array.

After adding the value, the program increments the counter (bx) by 2 using the "add" instruction. This is done to move to the next index in the num1 array, as each element in the array is 2 bytes long .

The program then compares the value of bx with 20 using the "cmp" instruction. If bx and 20 are not equal, the zero flag (ZF) is set to 0. The "jne" instruction is a

conditional jump that checks the value of the zero flag. If the zero flag is not set (ZF=0), the program jumps back to the "outerloop" label and continues with the next iteration. This ensures that the loop continues until the value of bx is equal to 20.

Once the loop finishes, the program stores the final result in the memory location "result" using the "mov" instruction .

Conditional Jump Explanation:

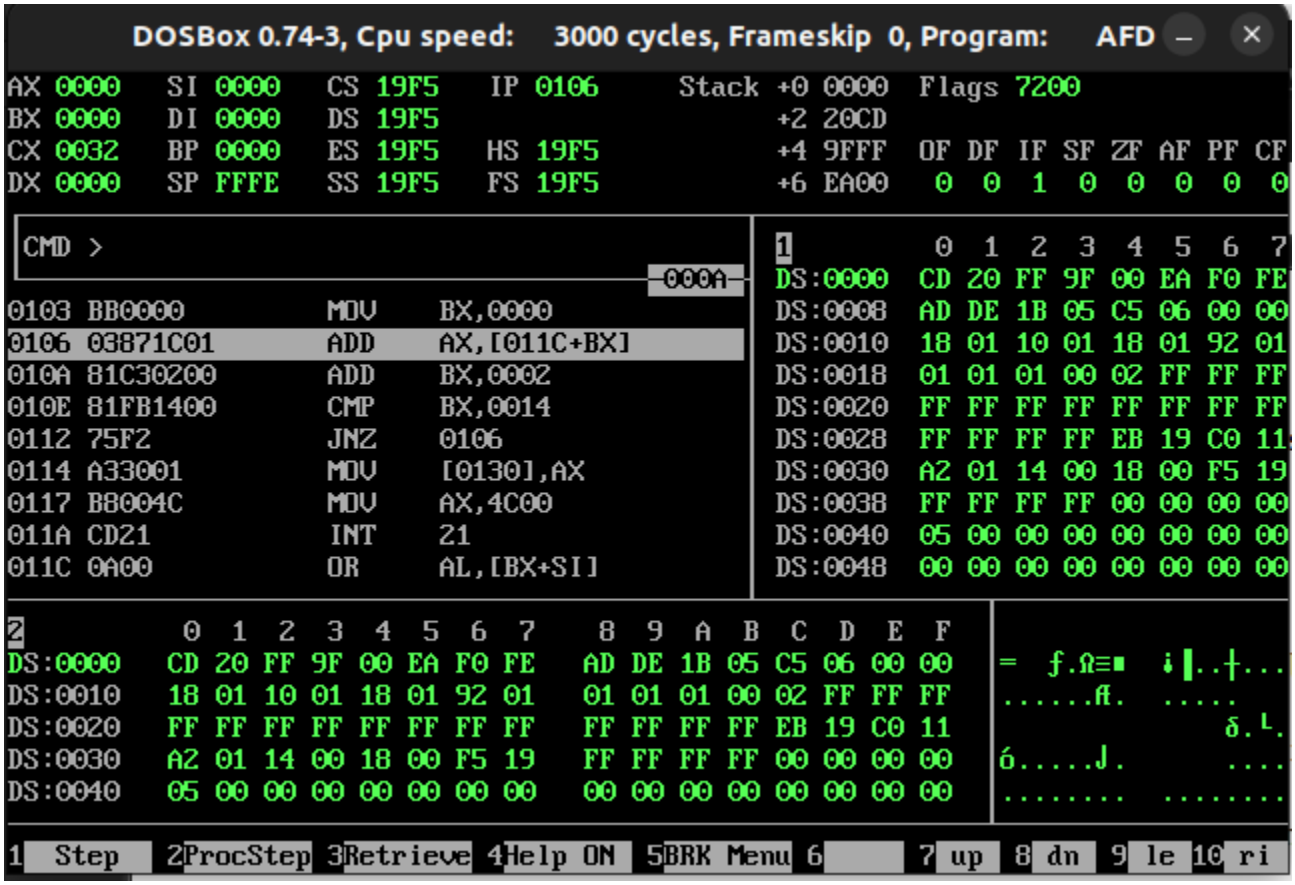
The jne instruction stands for "jump if not equal." It is used to jump to a specified location in the code if the zero flag (ZF) is not set. The zero flag is set by certain instructions (like cmp) when the result of the comparison is zero (i.e., the two operands are equal). If the zero flag is not set, it means the operands are not equal, and the jne instruction causes the program to jump to the specified location.

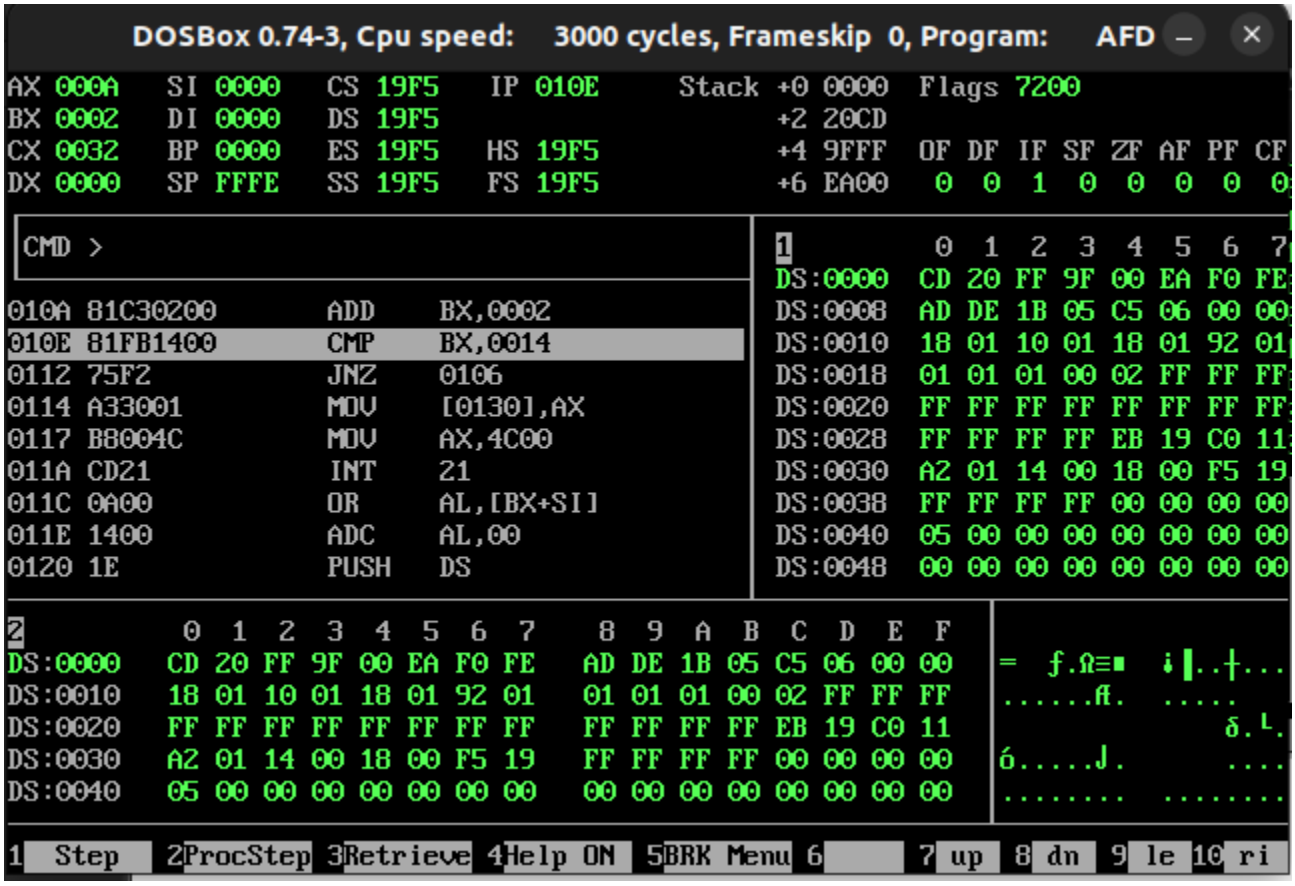
Conditional Jump is Used in My Program:

In my program, the cmp instruction is used to compare the value of the bx register with 20. This comparison is done to determine if the loop has iterated over all the elements in the num1 array. The cmp instruction sets the zero flag if bx is equal to 20, and it clears the zero flag if bx is not equal to 20.

After the cmp instruction, the jne instruction checks the state of the zero flag. If the zero flag is not set (meaning bx is not equal to 20), the jne instruction causes the program to jump back to the outerloop label, and the loop continues with the next iteration. This ensures that the loop continues to add the numbers in the num1 array until all elements have been processed.

To provide a better understanding, I will now provide a screenshot with every step, including the changes in the values of ax and bx in each loop. :

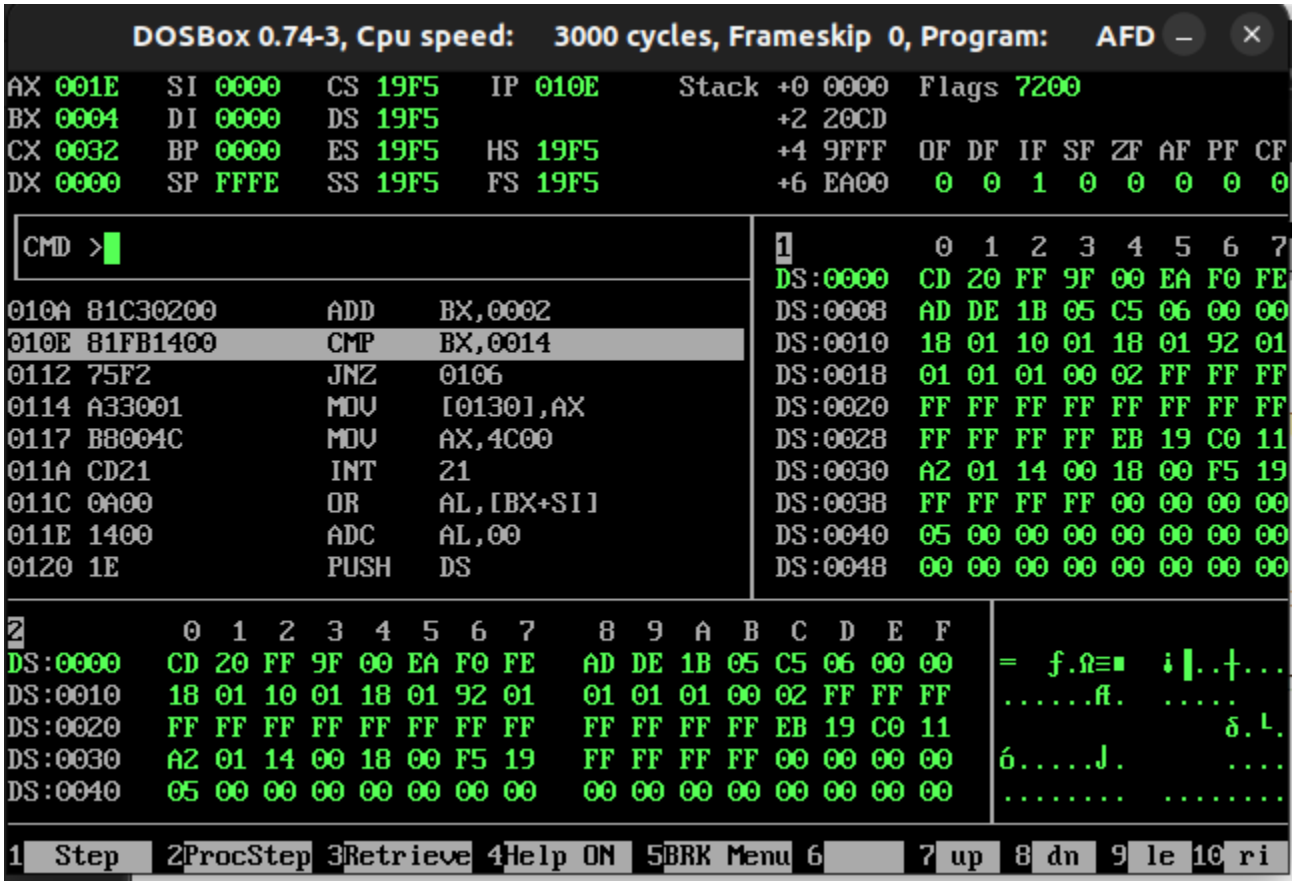




Explanation of the screenshot:

First Loop Iteration

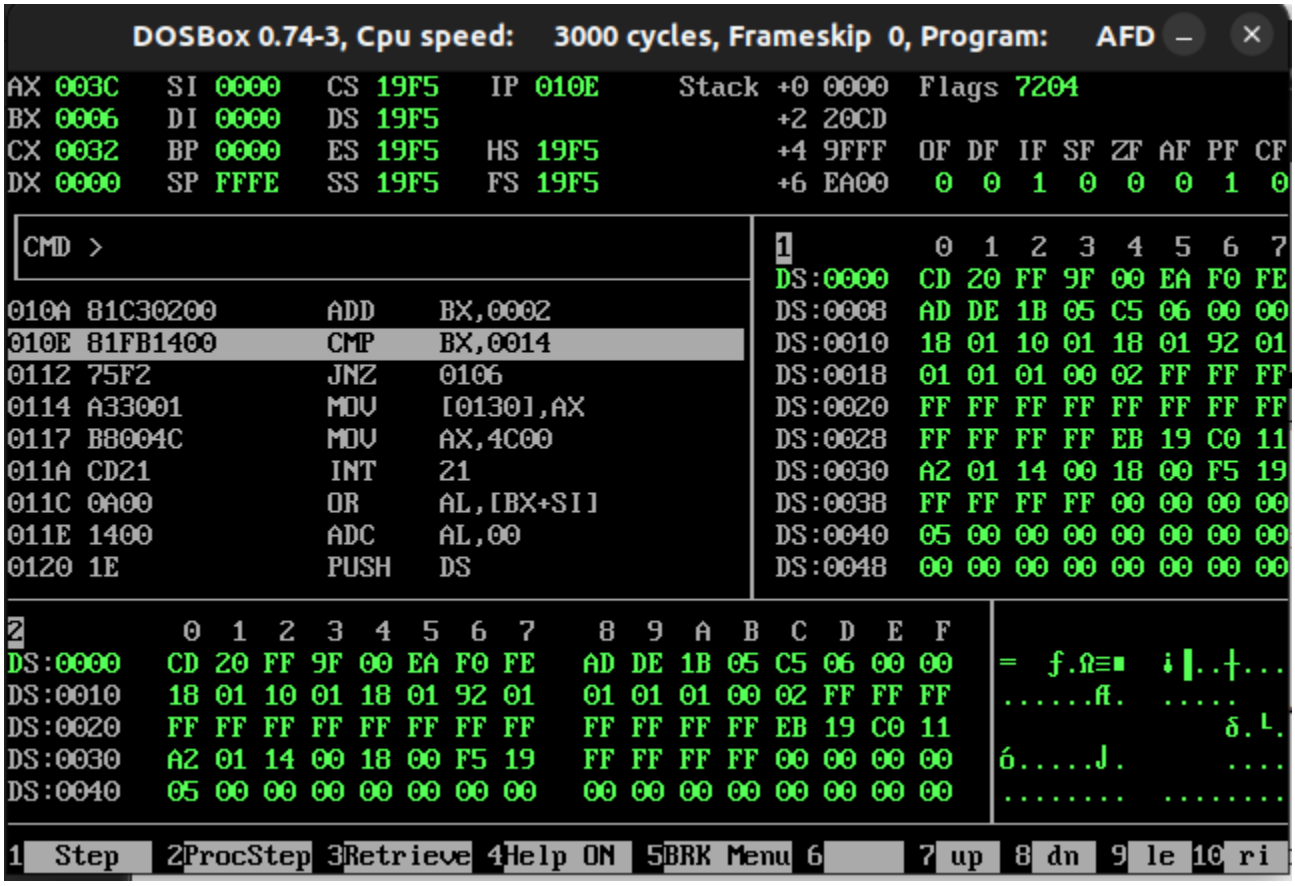
- **add ax, [num1 + bx]:** Adds the value at the memory location [num1 + bx] to ax. Since bx is initially 0, this adds the first number 10 to ax.
- **add bx, 2:** Increments bx by 2. This is because each number in the array is stored as a word (2 bytes), so we need to skip over the next number in the array.
- **cmp bx, 20:** Compares bx to 20. Since bx is now 2, the comparison fails, and the loop continues



Explanation of the screenshot:

Second Loop Iteration

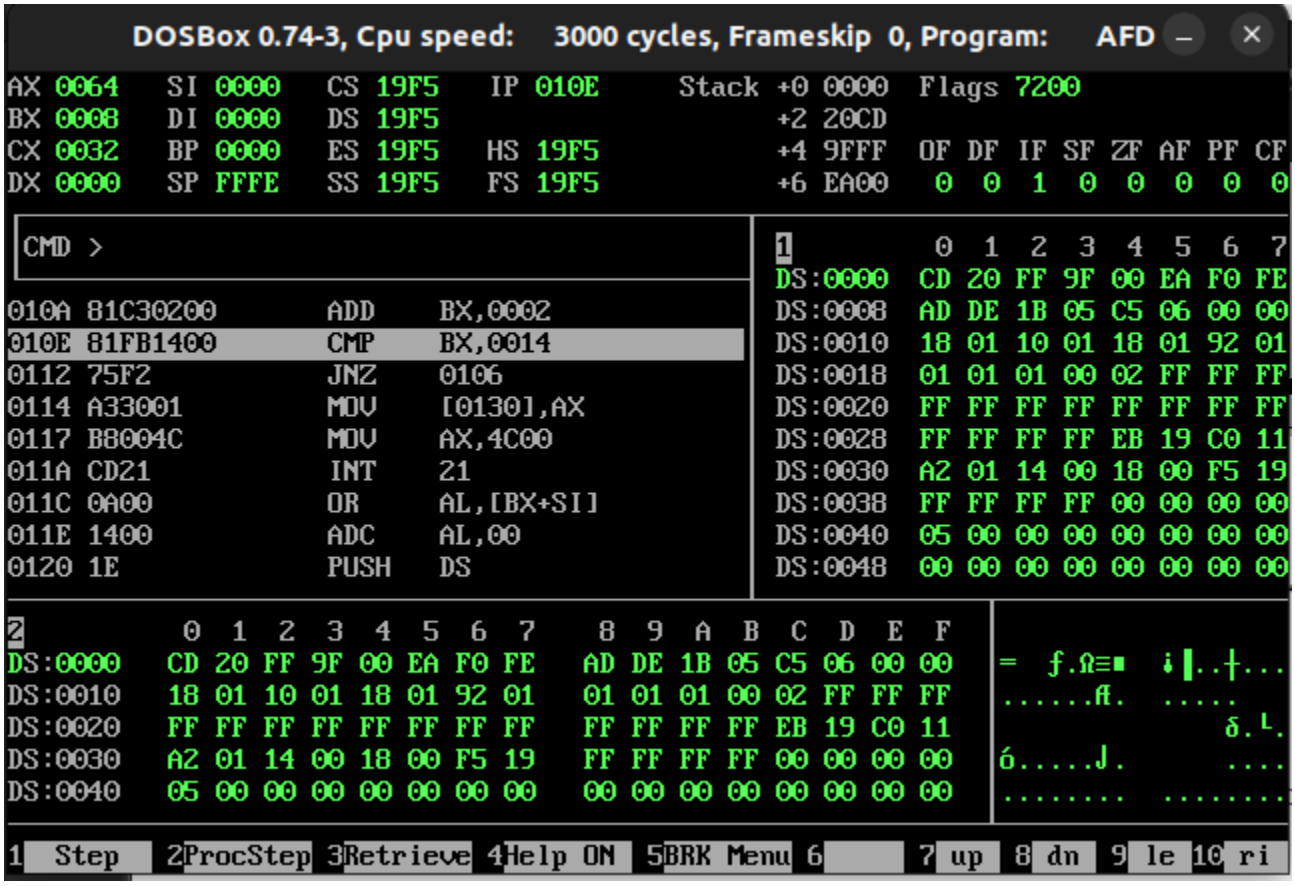
- **add ax, [num1 + bx]:** Adds the value at the memory location [num1 + bx] to ax. Since bx is now 2, this adds the second number 20 to ax. So, ax becomes 30 (10 + 20)
- **add bx, 2:** Increments bx by 2. Now bx is 4.
- **cmp bx, 20:** Compares bx to 20. Since bx is now 4, the comparison fails, and the loop continues.



Explanation of the screenshot:

Third Loop Iteration

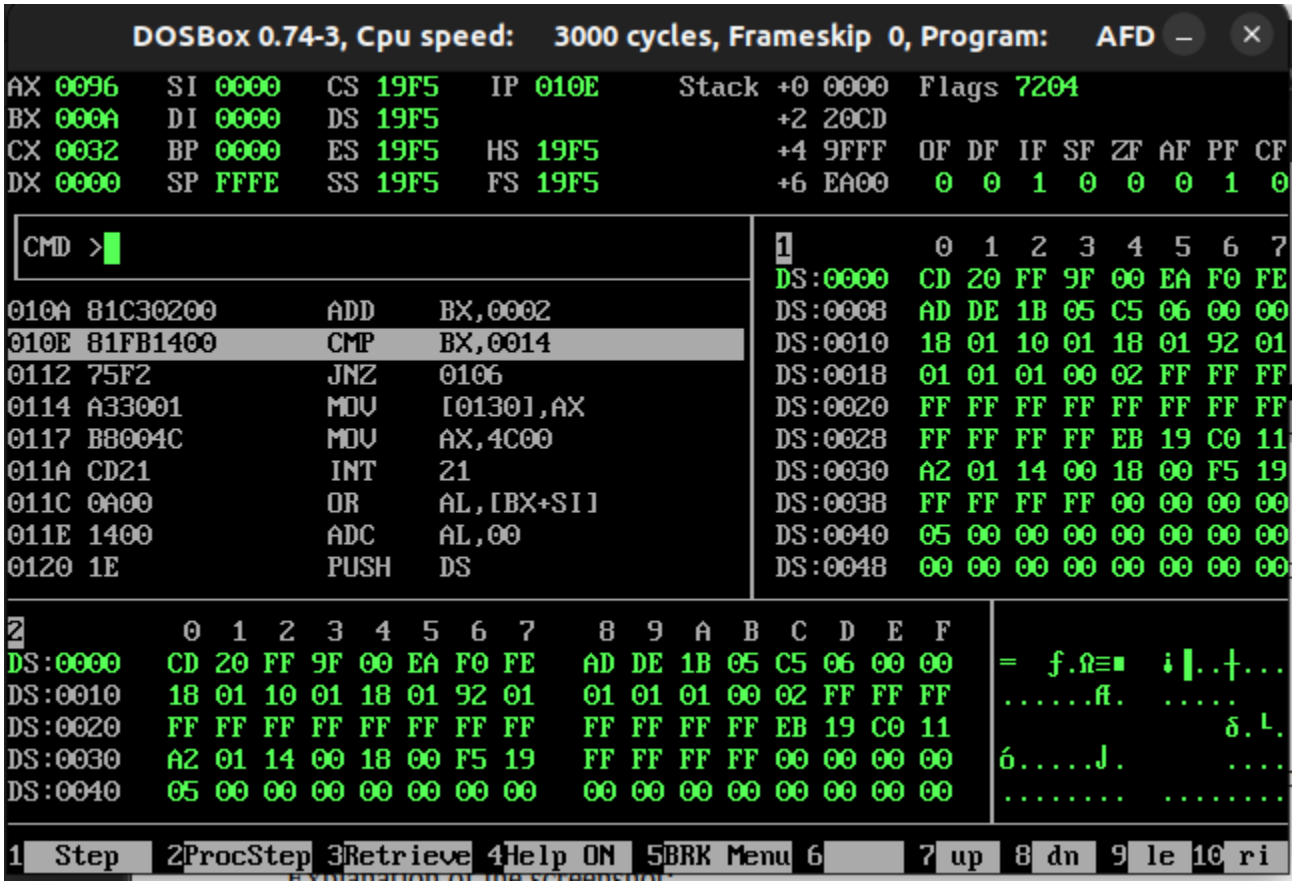
- **add ax, [num1 + bx]:** Adds the value at the memory location [num1 + bx] to ax. Since bx is now 4, this adds the third number 30 to ax. So, ax becomes 60 (30 + 30).
- **add bx, 2:** Increments bx by 2. Now bx is 6.
- **cmp bx, 20:** Compares bx to 20. Since bx is now 6, the comparison fails, and the loop continues.



Explanation of the screenshot:

Fourth Loop Iteration

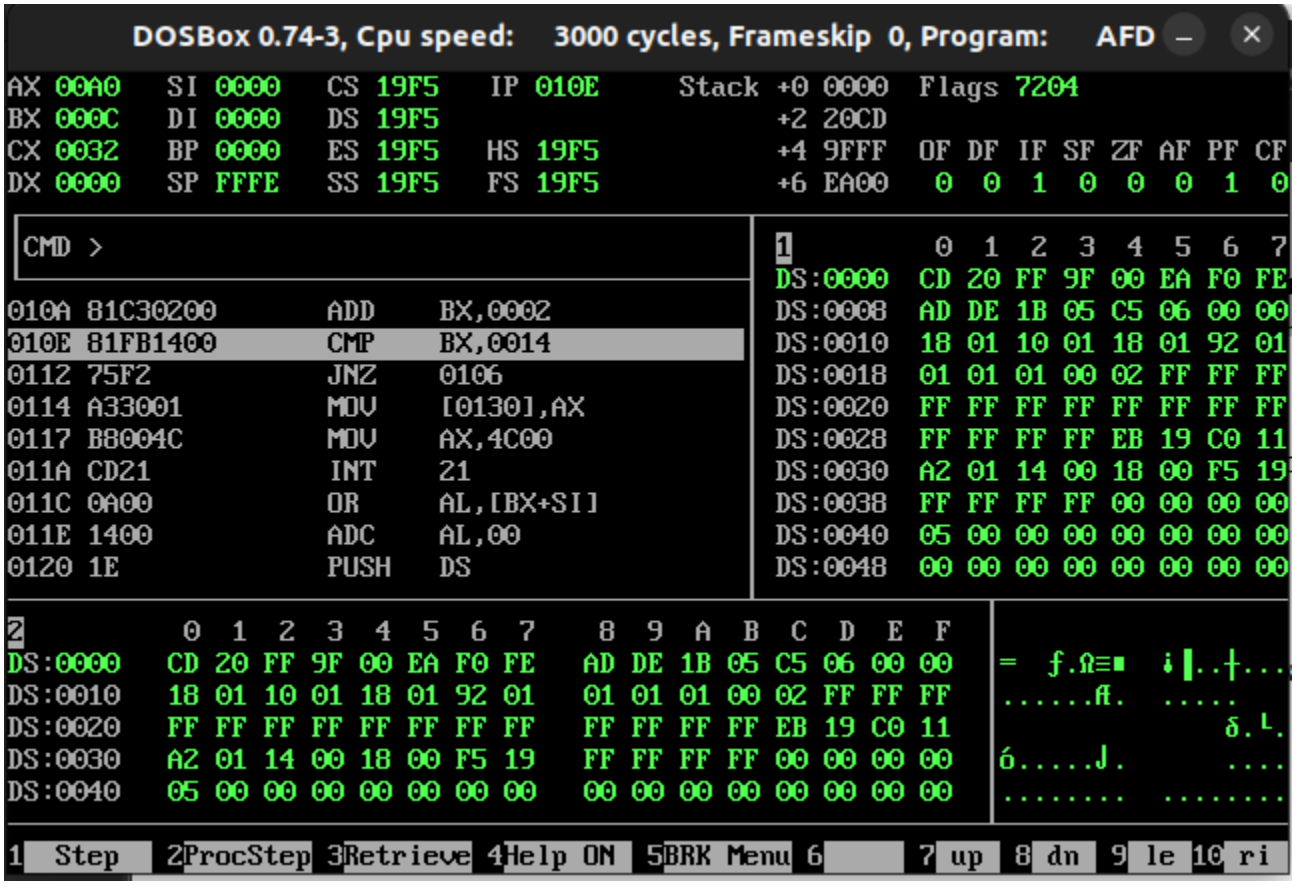
- **add ax, [num1 + bx]:** Adds the value at the memory location [num1 + bx] to ax. Since bx is now 6, this adds the fourth number 40 to ax. So, ax becomes 100 (60 + 40).
- **add bx, 2:** Increments bx by 2. Now bx is 8.
- **cmp bx, 20:** Compares bx to 20. Since bx is now 8, the comparison fails, and the loop continues.



Explanation of the screenshot:

Fifth Loop Iteration

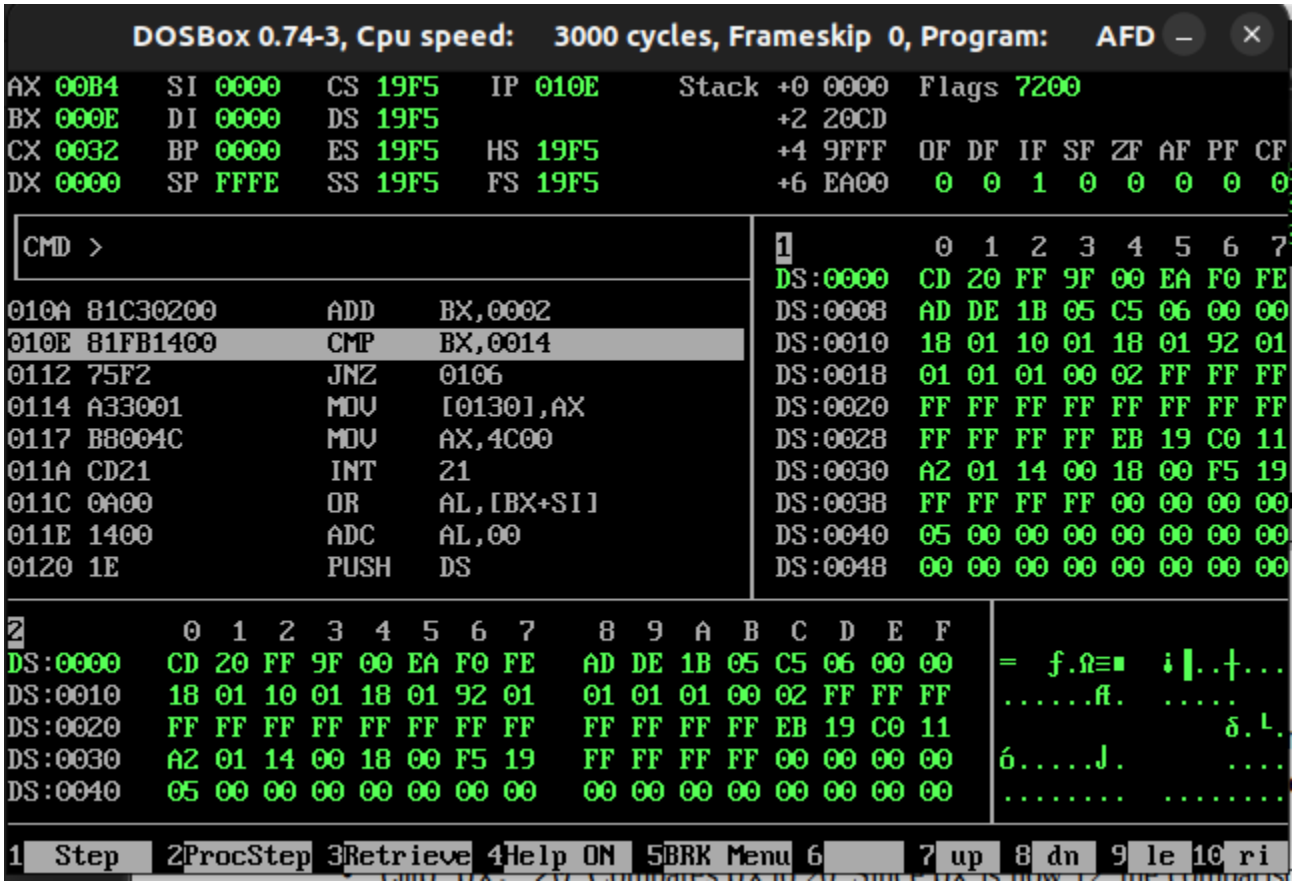
- **add ax, [num1 + bx]:** Adds the value at the memory location [num1 + bx] to ax. Since bx is now 8, this adds the fifth number in the array 50 to ax. So, ax becomes 150 (100 + 50).
- **add bx, 2:** Increments bx by 2. Now bx is 10.
- **cmp bx, 20:** Compares bx to 20. Since bx is now 10, the comparison fails, and the loop continues.



Explanation of the screenshot:

Sixth Loop Iteration

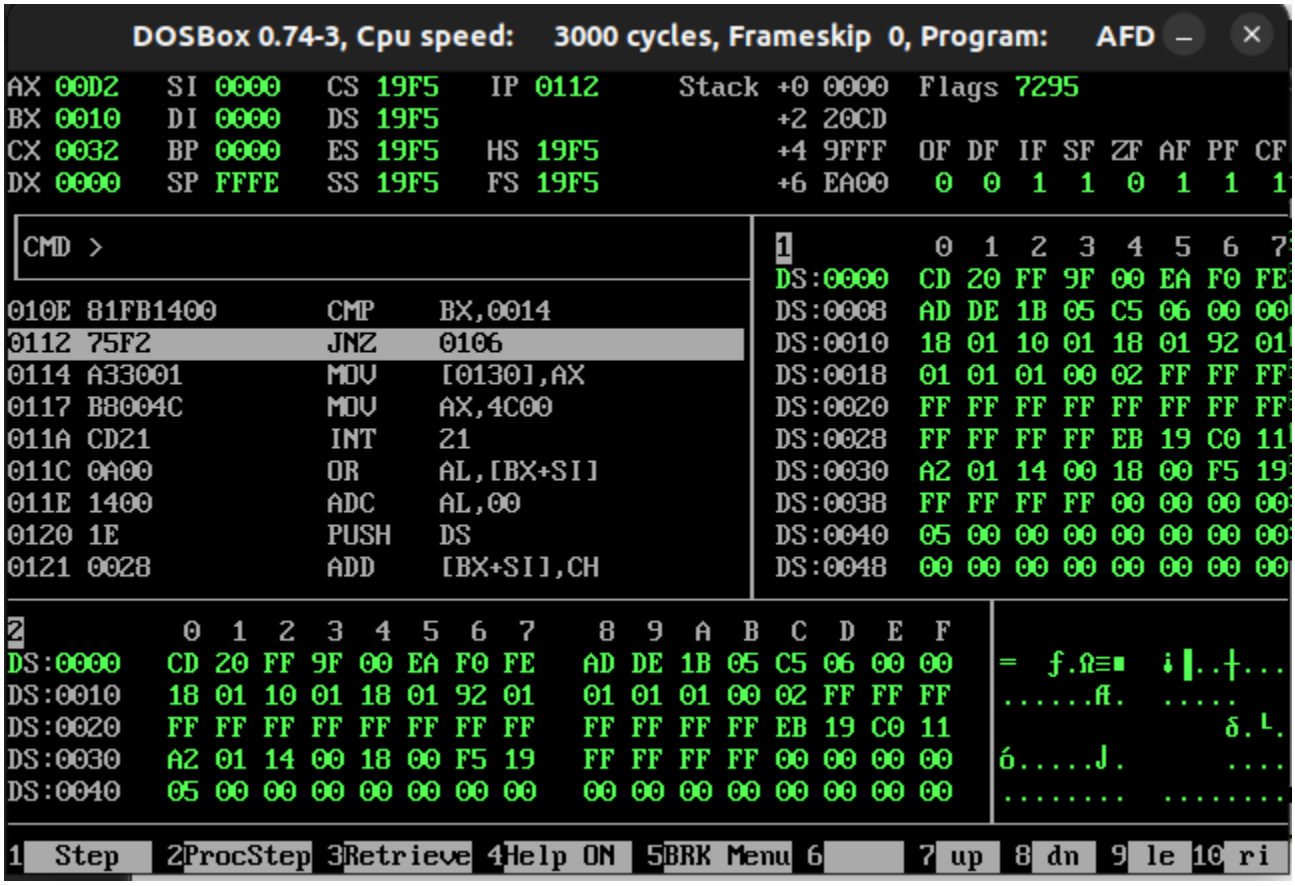
- **add ax, [num1 + bx]:** Adds the value at the memory location [num1 + bx] to ax. Since bx is now 10, this adds the sixth number 10 to ax. So, ax becomes 160 (150 + 10).
- **add bx, 2:** Increments bx by 2. Now bx is 12.
- **cmp bx, 20:** Compares bx to 20. Since bx is now 12, the comparison fails, and the loop continues.



Explanation of the screenshot:

Seventh Loop Iteration

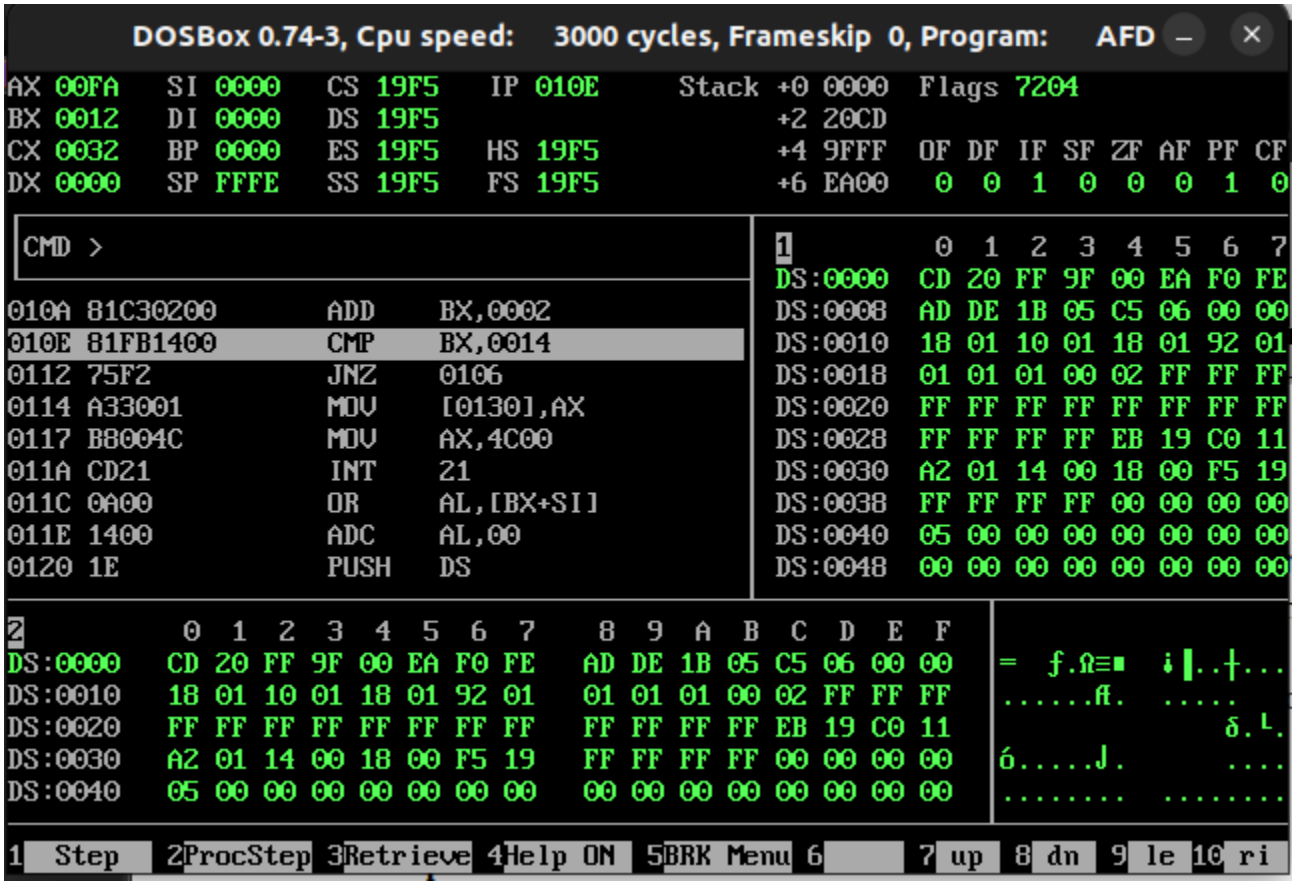
- **add ax, [num1 + bx]:** Adds the value at the memory location [num1 + bx] to ax. Since bx is now 12, this adds the seventh number 20 to ax. So, ax becomes 180 (160 + 20).
- **add bx, 2:** Increments bx by 2. Now bx is 14.
- **cmp bx, 20:** Compares bx to 20. Since bx is now 14, the comparison fails, and the loop continues.



Explanation of the screenshot:

Eighth Loop Iteration

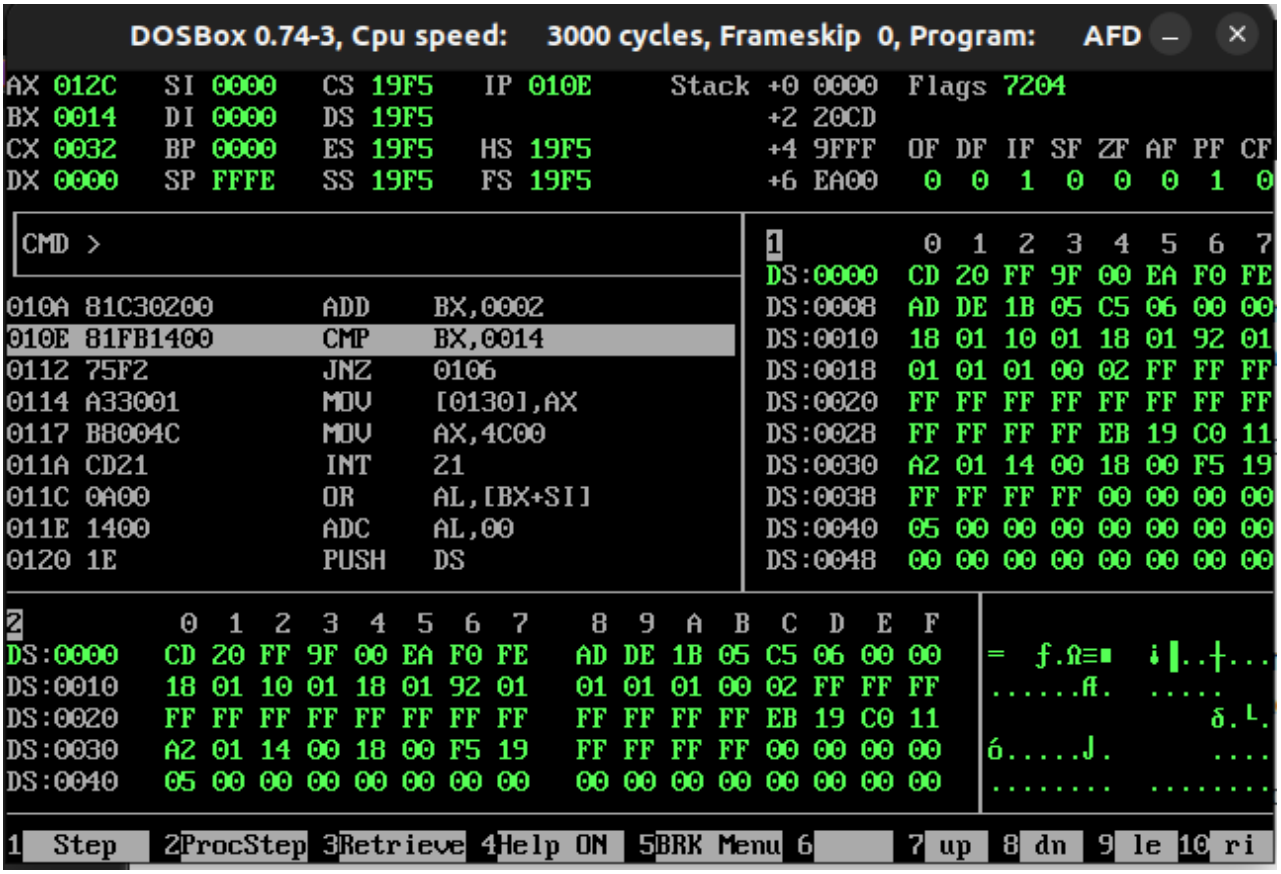
- **add ax, [num1 + bx]:** Adds the value at the memory location [num1 + bx] to ax. Since bx is now 14, this adds the eighth number 30 to ax. So, ax becomes 210 (180 + 30).
- **add bx, 2:** Increments bx by 2. Now bx is 16.
- **cmp bx, 20:** Compares bx to 20. Since bx is now 16, the comparison fails, and the loop continues.



Explanation of the screenshot:

Ninth Loop Iteration

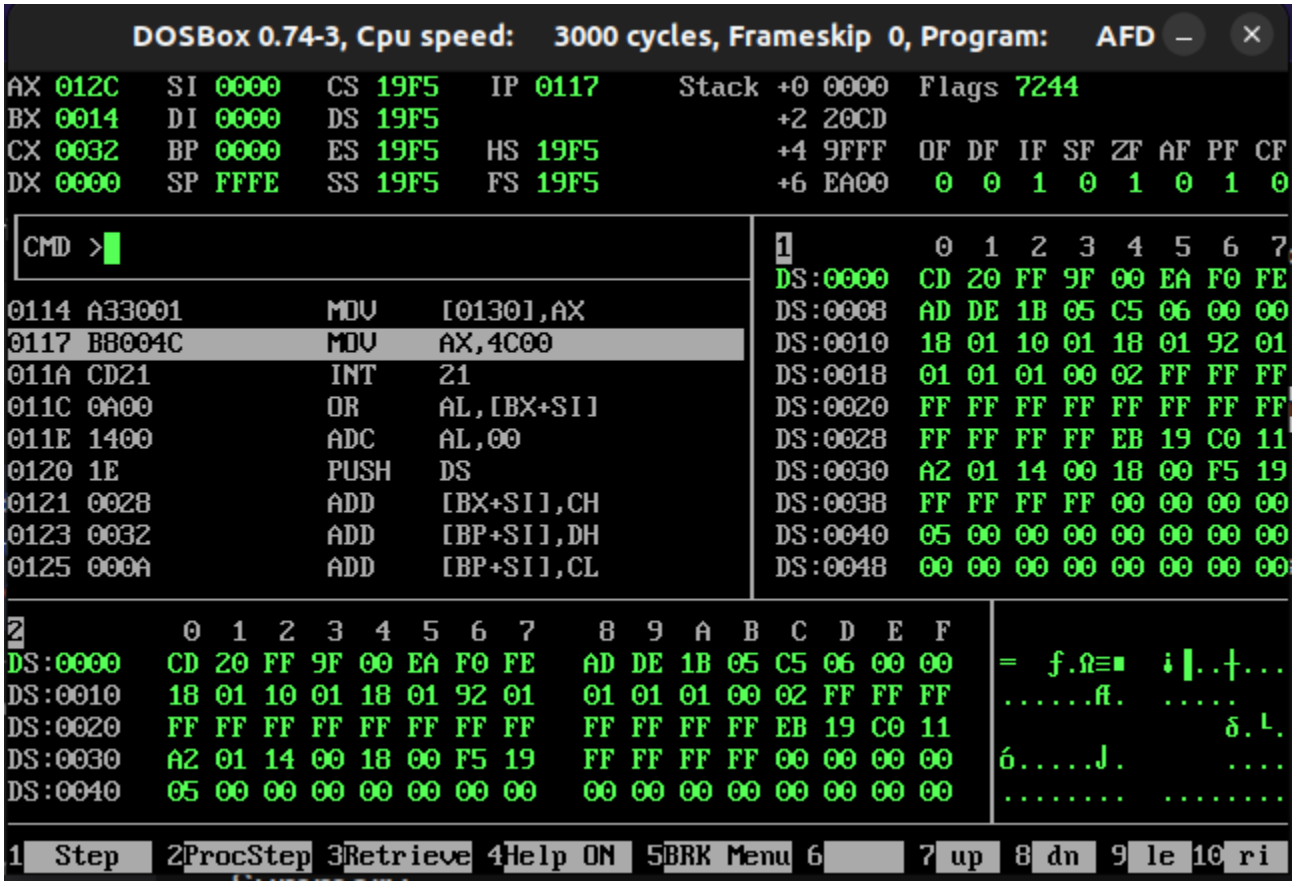
- **add ax, [num1 + bx]:** Adds the value at the memory location [num1 + bx] to ax. Since bx is now 16, this adds the ninth number 40 to ax. So, ax becomes 250(210+ 40).
- **add bx, 2:** Increments bx by 2. Now bx is 18.
- **cmp bx, 20:** Compares bx to 20. Since bx is now 18, the comparison fails, and the loop continues.



Explanation of the screenshot:

Tenth Loop Iteration

- **add ax, [num1 + bx]:** Adds the value at the memory location [num1 + bx] to ax. Since bx is now 18, this adds the tenth number 50 to ax. So, ax becomes 300 (250+ 50).
- **add bx, 2:** Increments bx by 2. Now bx is 20.
- **cmp bx, 20:** Compares bx to 20. Since bx is now 20, the comparison succeeds, and the loop terminates.



Finalization

- **mov [result], ax:** Stores the final sum in the result variable.
- **mov ax, 0x4c00:** Prepares to terminate the program.
- **int 0x21:** Calls the DOS interrupt to terminate the program.

```
                    [org 0x0100]

jmp start       ; see next instructions when you haven't yet executed this!

          num1: dw   10, 20, 30, 40, 50, 10, 20, 30, 40, 50
                    result: dw 0



                        start:
                    ; initialize stuff
              mov  ax, 0            ; reset the accumulator
                 mov  bx, 0            ; set the counter

                      outerloop:
                   add  ax, [num1 + bx]
                        add  bx, 2

            cmp bx, 20          ; sets ZF=0 when they are equal
                        jne  outerloop


                    mov  [result], ax

                    mov  ax, 0x4c00
                        int  0x21
```

Unconditional Jump:

The unconditional jump (`jmp`) instruction in assembly language is used to transfer control to a different part of the program without any condition. It simply jumps to the specified location in the code and continues execution from there.

Unconditional Jump In My Program:

In my program, the jmp start instruction is placed at the very beginning, right after the org 0x0100 directive. This directive sets the origin, or starting address, of the program in memory. The jmp start instruction then immediately transfers control to the start label, effectively skipping over the num1 and result data declarations.

After the jmp start instruction, the program initializes the ax and bx registers, enters a loop to add the numbers stored in the num1 array, and finally stores the result in the result variable. The loop continues until the bx register is equal to 20, indicating that all elements in the num1 array have been processed.

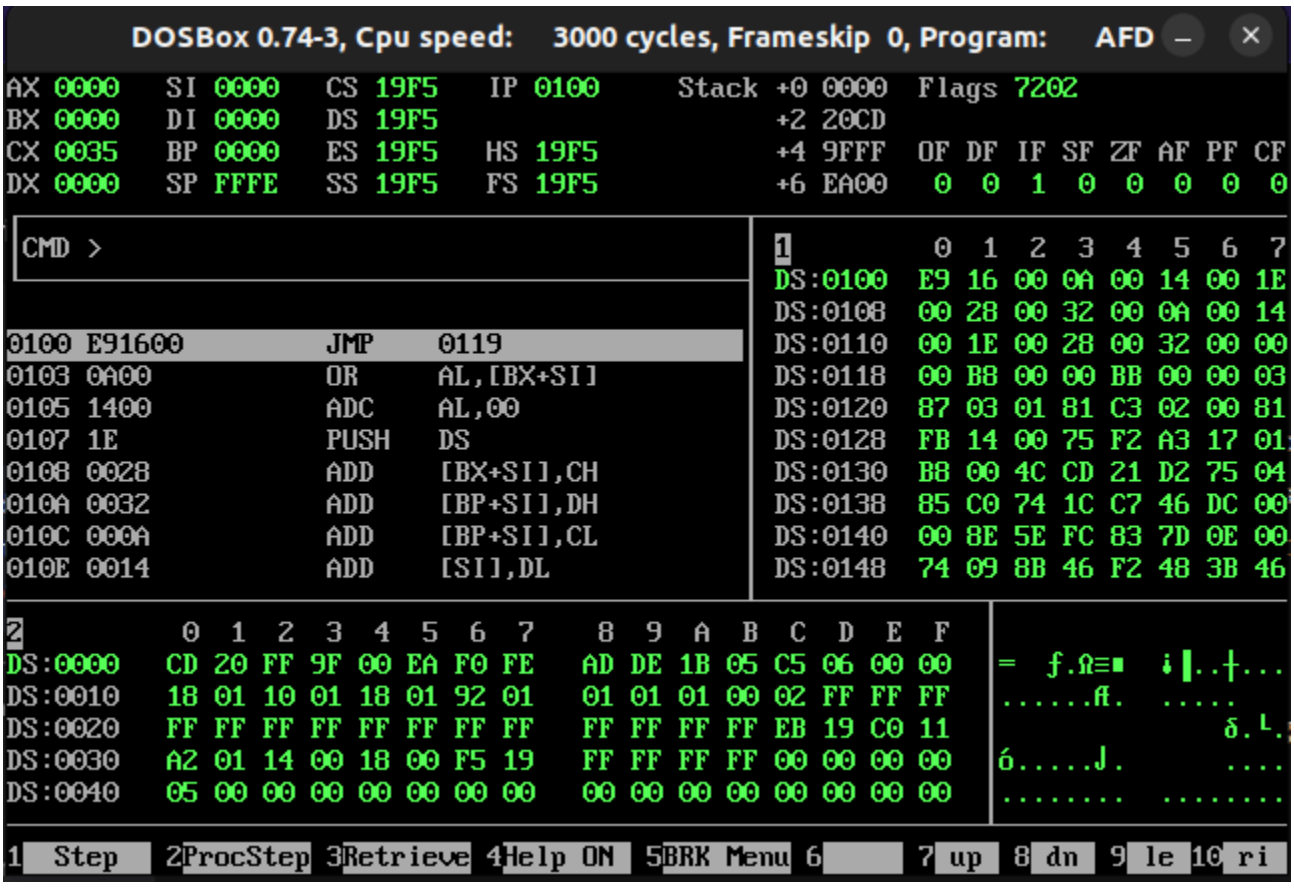
Data Stored:

Despite the unconditional jump (jmp start) skipping the num1 and result data declarations, the data is still stored in memory.

Why:

When the assembler processes the program, it allocates space in the program's data segment for each variable declared in the program.

Where Data is Stored:



In the above screenshot, the data is stored starting from address 0103. This indicates that the jump instruction itself occupies 3 bytes in memory, with its opcode being E9 16 00.