# TASK 1:

org 0x0100

mov si, num ; Point SI to the start of the input string

find\_end: ; Find the end of the string (null terminator)

mov al, [si]

cmp al, 0

je check\_palindrome ; If null terminator is found, check for palindrome

inc si

jmp find\_end

check\_palindrome:

dec si ; Move SI back to the last character

mov di, num ; Point DI to the start of the string

compare\_loop:

mov al, [si] ; Load characters from the end of the string

cmp al, [di] ; Compare with characters from the start of the string

jne not\_palindrome ; If not equal, it's not a palindrome

inc si ; Move to the next character from the end

inc di ; Move to the next character from the start

cmp al, 0 ; Check for end of string

jnz compare\_loop ; If not the end, continue comparing

mov dx, result ; If the loop completes without a mismatch, it's a palindrome

mov ah, 9

int 21h

jmp done

not\_palindrome:

mov dx, not\_result

mov ah, 9

int 21h

done:

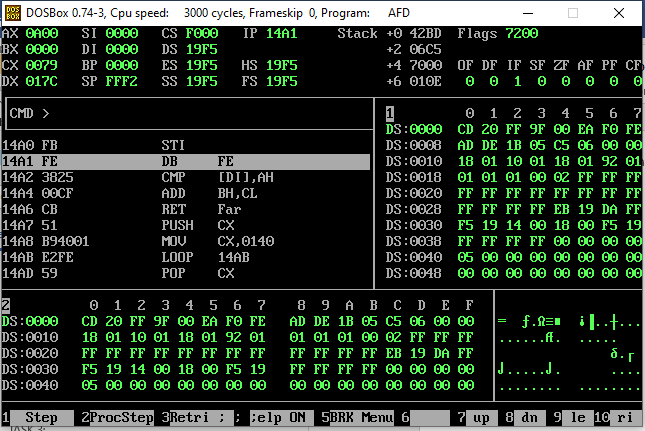
mov ax, 0x4c00

int 0x21

section .bss

num resb 10 ; Input number as a string

# OUTPUT:



# TASK 2:

org 0x0100

mov bx, 0

mov cx, 0

mov si, num

convert\_loop:

mov al, [si]

cmp al, 0x0D ; Check for Enter (Carriage Return)

je fact

sub al, '0' ; Convert ASCII to integer

imul bx, bx, 10 ; Multiply result by 10

add bx, ax ; Add the new digit

inc si

jmp convert\_loop

fact:

cmp bx, 1

jbe done ; If bx <= 1, factorial calculation is done

imul ax, bx ; Multiply result by bx

mov bx, ax ; Store the result in bx

dec bx ; Decrement bx

jmp fact

done:

mov ah, 9 ; Display the factorial result

mov dx, result

int 21h

mov ah, 0x4c00

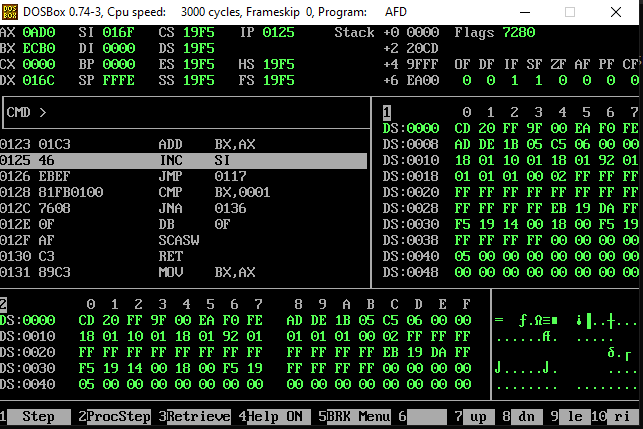
int 0x21

section .bss

num resb 5 ; Input number as a string (reserve 5 bytes)

factorial resw 1 ; Factorial result (16-bit word)

# OUTPUT:



# TASK 3:

org 0x0100

mov cx, 8 ; Set loop counter to 8 bits

mov si, binary + 7 ; Start at the end of the binary string (little-endian)

convert\_loop:

mov ax, 0 ; Clear AX for division

mov al, byte [decimal] ; Load a decimal digit

sub al, '0' ; Convert ASCII to integer

mov bx, 256 ; Set BX to 256 (2^8)

div bx ; Divide AX by 256

add ah, '0' ; Convert remainder to ASCII

mov [si], ah ; Store the binary digit

dec si ; Move to the previous position in the binary string

loop convert\_loop ; Repeat the loop for 8 bits

mov byte [binary + 8], 0 ; Null-terminate the binary string

mov ax, 0x4c00

int 0x21

section.data

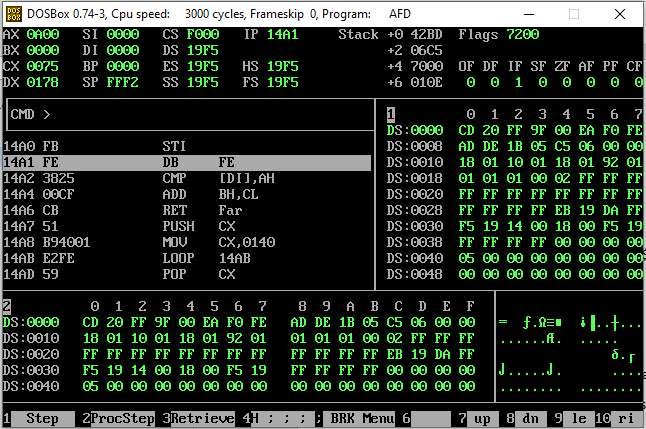
num db 10

section .bss

decimal resb 3 ; Input decimal number as a string (up to 3 characters)

binary resb 9 ; Binary representation as a string (8 bits + null terminator)

# OUTPUT:



# TASK 4:

org 0x0100

start:

mov bx, 0 ; Initialize array index to zero

mov byte [swap], 0 ; Reset swap flag to no swaps

loop1:

mov ax, [data+bx] ; Load number in ax

cmp ax, [data+bx+2] ; Compare with the next number

jge noswap ; No swap if already in order

; Swap the elements

mov dx, [data+bx+2] ; Load the second element in dx

mov [data+bx+2], ax ; Store the first number in the second

mov [data+bx], dx ; Store the second number in the first

mov byte [swap], 1 ; Flag that a swap has been done

noswap:

add bx, 2 ; Advance bx to the next index

cmp bx, 28 ; Are we at the last index? (14 elements \* 2 bytes each)

jne loop1 ; If not, compare the next two

cmp byte [swap], 1 ; Check if a swap has been done

je start ; If yes, make another pass

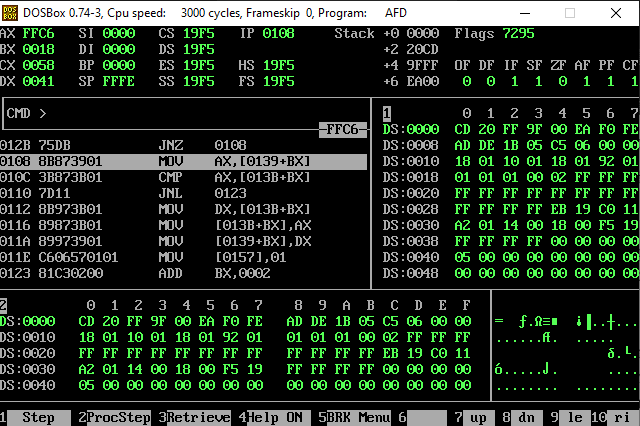
mov ax, 0x4c00

int 0x21

data: dw 60, 55, -55, 60, -60, 58, -58, -58, 25, 15, 34, 87, 90, 12, 65

swap: db 0

# OUTPUT:



**Dry Run:**

**First Pass:**

Comparing and swapping adjacent elements:

[60, 55, -55, 60, -60, 58, -58, -58, 25, 15, 34, 87, 12, 90, 65] (Swap 90 and 12)

[60, 55, -55, 60, -60, 58, -58, -58, 25, 15, 34, 12, 87, 90, 65] (Swap 87 and 12)

[60, 55, -55, 60, -60, 58, -58, -58, 25, 15, 12, 34, 87, 90, 65] (Swap 34 and 12)

Result after the first pass: [60, 60, 55, -55, -60, 58, -58, -58, 25, 15, 34, 12, 87, 90, 65]

**Second Pass:**

Comparing and swapping adjacent elements:

[60, 60, 55, -55, 58, -60, -58, -58, 25, 15, 34, 12, 87, 90, 65] (Swap -60 and 58)

Result after the second pass: [60, 60, 55, -55, 58, -58, -60, -58, 25, 15, 34, 12, 87, 90, 65]

**Third Pass:**

Comparing and swapping adjacent elements:

[60, 60, 55, -55, 58, -58, -58, -60, 25, 15, 34, 12, 87, 90, 65] (Swap -60 and -58)

Result after the third pass: [60, 60, 55, -55, 58, -58, -58, -60, 25, 15, 34, 12, 87, 90, 65]

**Fourth Pass:**

No swaps are needed. The array is sorted.

Final sorted array: [60, 60, 55, -55, 58, -58, -58, -60, 25, 15, 34, 12, 87, 90, 65] (Descending order)

**Choice of Branch:**

* **jge noswap (Jump if Greater or Equal):** This branch is used to compare two adjacent elements in the array. It checks if the current element is greater than or equal to the next element. If this condition is true, it means the elements are already in the correct order, and no swap is necessary.
* **jne loop1 (Jump if Not Equal):** This branch is used to check if the loop index (bx) has reached the last index of the array. If it hasn't, the code jumps back to loop1 to continue comparing and potentially swapping the next pair of elements. This branch ensures that the sorting algorithm iterates through the entire array until no more swaps are needed.
* **je start (Jump if Equal):** This branch is used to check if any swaps were performed in the current pass of the outer loop. If no swaps were made (i.e., byte [swap] is still 0), it means the array is already sorted, and the code jumps back to start to terminate the sorting process early. This branch optimizes the code by avoiding unnecessary iterations when the array is already sorted.