**Lecture 7 – part 1**

**Stack in assembly**

The *runtime stack* is a memory array managed directly by the CPU, using the ESP register, known as the *stack pointer register*.

The ESP register holds a 32-bit offset into some location on the stack.

We rarely manipulate ESP directly; instead, it is indirectly modified by instructions such as CALL, PUSH, and POP.

***ESP always points to the last value to be added to, or pushed on, the top of stack.***

Each stack location in this figure contains 32 bits, which is the case when a program is running in 32-bit mode.

In 16-bit real-address mode, the SP register points to the most recently pushed value and stack entries are typically 16 bits long.

# **Push Operation**

A 32-bit push operation decrements the stack pointer by 4 and copies a value into the location in the stack pointed to by the stack pointer.

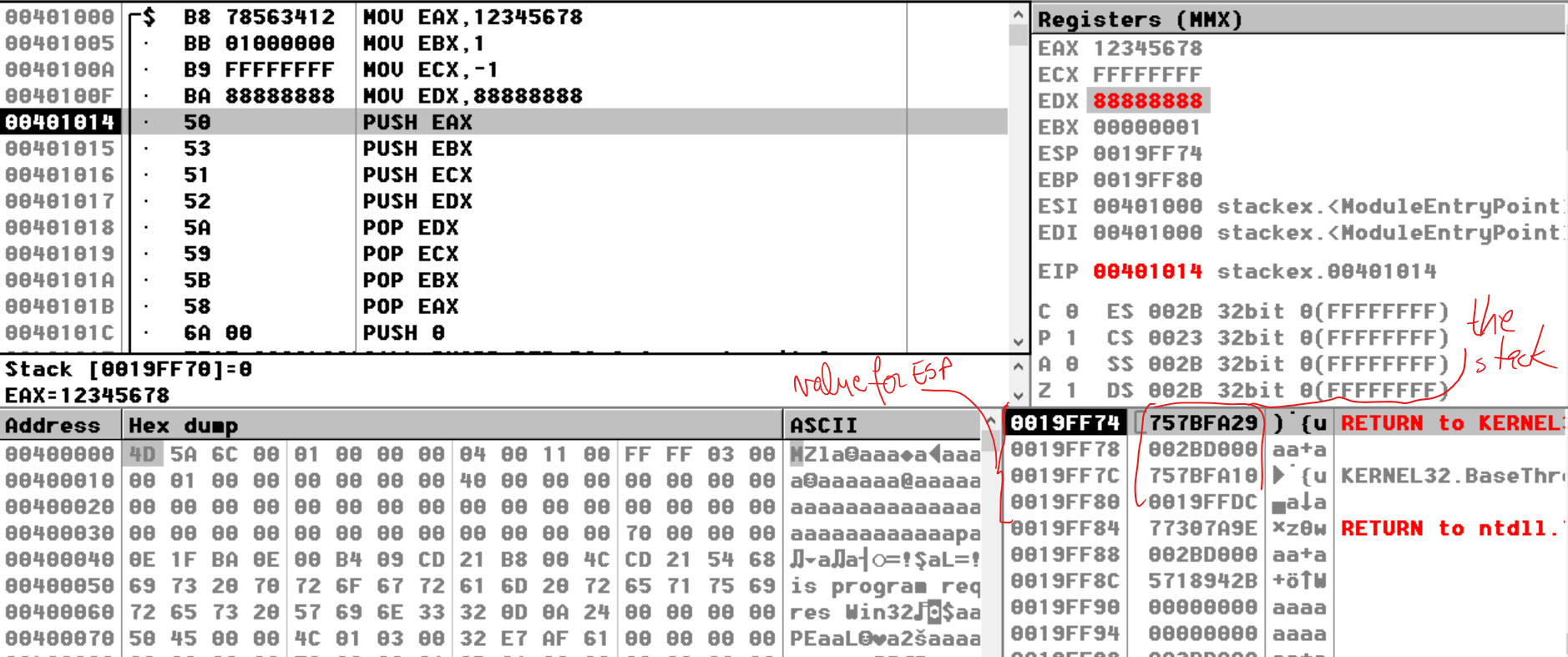
To demonstrate, let’s check the stack values if we have the next code lines:

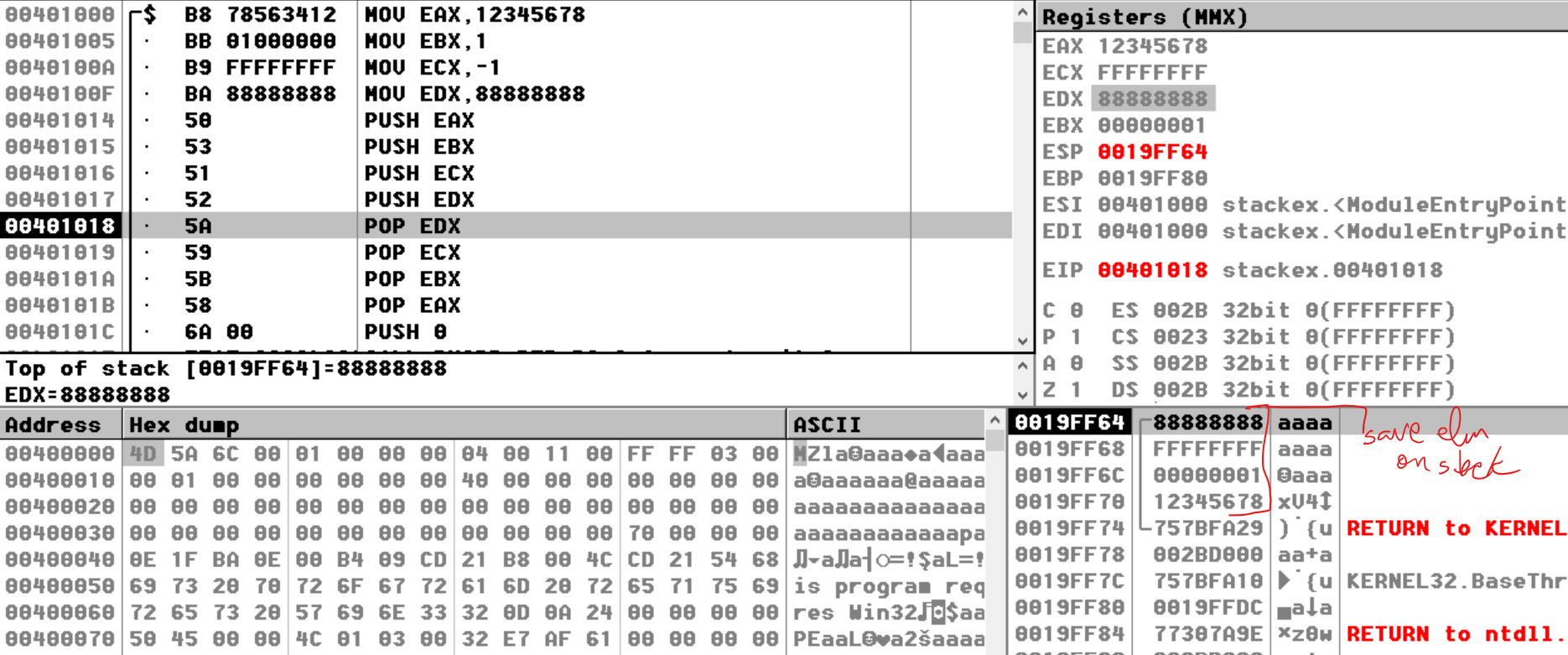
|  |  |
| --- | --- |
|  | ESP value Values saved on stack |

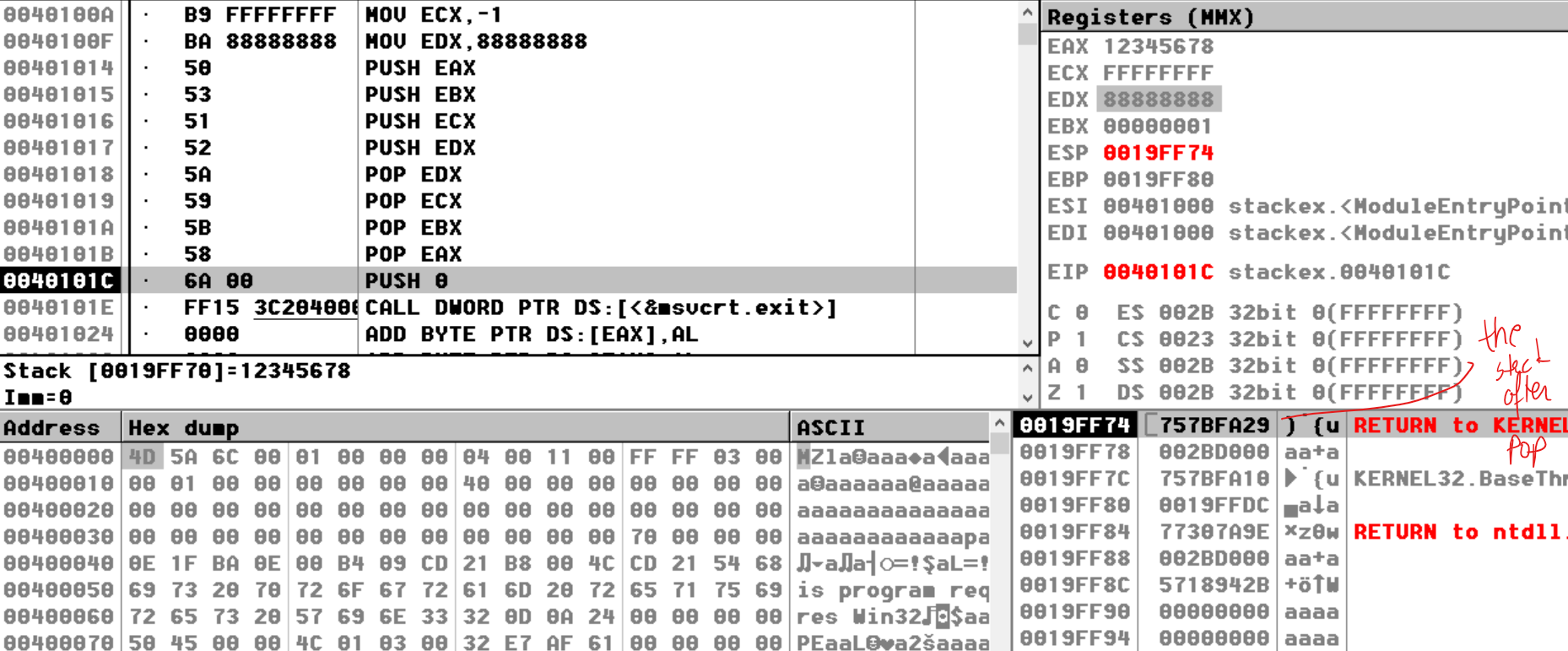
# Pop Operation

A *pop* operation removes a value from the stack.

After the value is popped from the stack, the stack pointer is incremented (by the stack element size) to point to the next-highest location in the stack.







The area of the stack below ESP is *logically empty*, and will be overwritten the next time the current program executes any instruction that pushes a value on the stack.

***PUSH Instruction***

The PUSH instruction first decrements ESP and then copies a source operand into the stack.

A 16-bit operand causes ESP to be decremented by 2.

A 32-bit operand causes ESP to be decremented by 4.

There are three instruction formats:

PUSH *reg/mem16*

PUSH *reg/mem32*

PUSH *imm32*

*push 1ah = > 1 doubleword = hexadecimal digits = 8 hexadecimal digits*

*we will have on the stack: 00001A*

*1 doubleword = 4 bytes*

*1 byte = 2 hexadecimal digits*

*1 word = 4 hexadecimal digits = 2 bytes*

*1 quadword = 2 doublewords = 16 hexadecimaldigits = 8 bytes*

***POP Instruction***

The POP instruction first copies the contents of the stack element pointed to by ESP into a 16- or 32-bit destination operand and then increments ESP. If the operand is 16 bits, ESP is incremented by 2; if the operand is 32 bits, ESP is incremented by 4:

POP *reg/mem16*

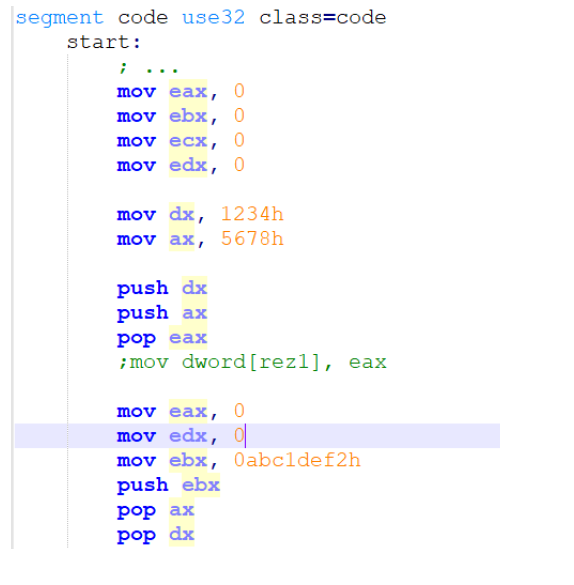
POP *reg/mem32*

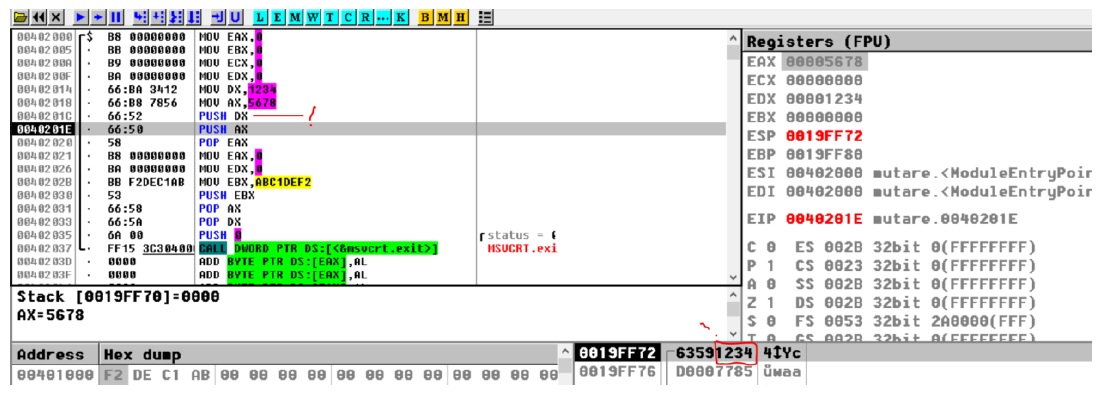
# Stack Applications

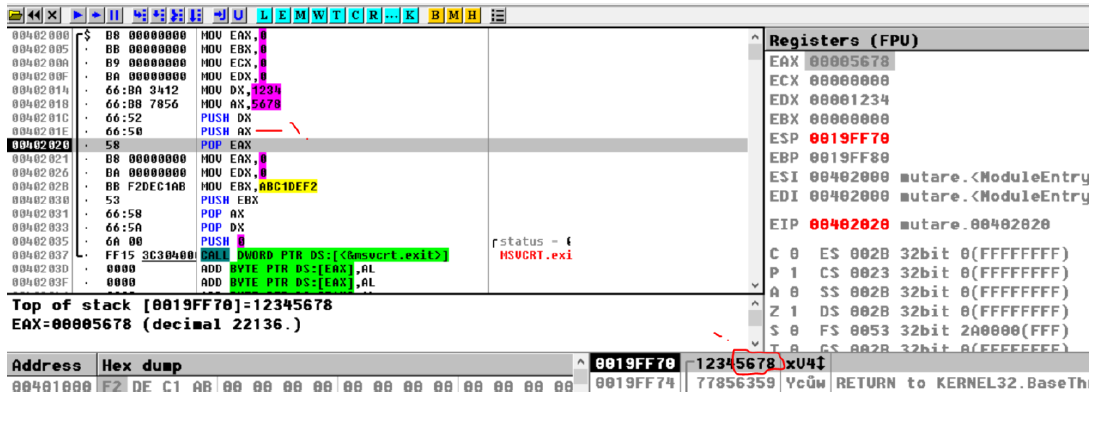
There are several important uses of runtime stacks in programs:

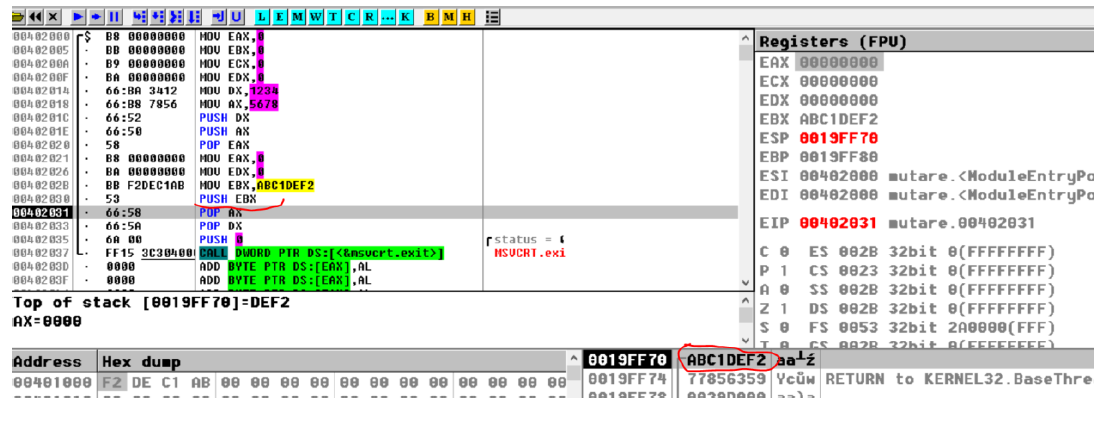
* A stack makes a convenient temporary save area for registers when they are used for more than one purpose. After they are modified, they can be restored to their original values.
* The stack provides temporary storage for local variables inside subroutines or inside particular labels.
* When the CALL instruction executes, the CPU saves the current subroutine’s return address on the stack.
* When calling a subroutine, you pass input values called *arguments* by pushing them on the stack.

*Examples:*

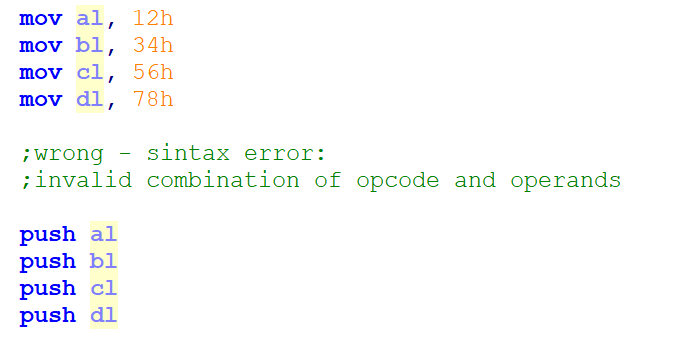




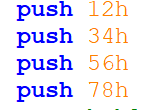




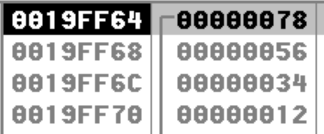
Because the stack is 32 bits memory area **we cannot add** on the stack values on 8 bits, for instace registers on 8 bits:



Still, We can add on the stack constants of 8 bits dimensions:



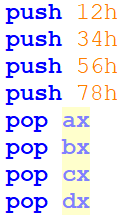
The result (the stack) will be:



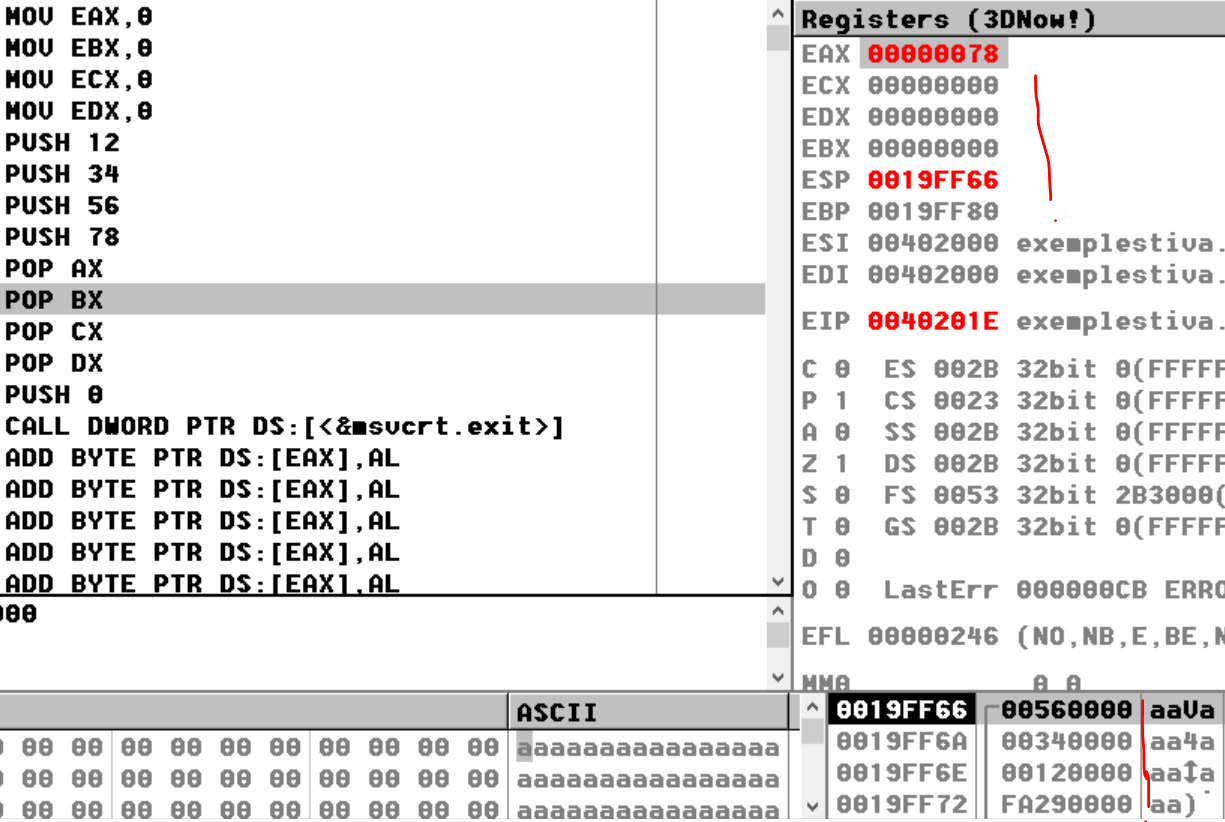
So now all the elements are on the stack, but in order to use them, we have to take all the elements and save the elements:

**BUT** we cannot use the registers on 8 bits and we cannot take values in variables on 8 bits.

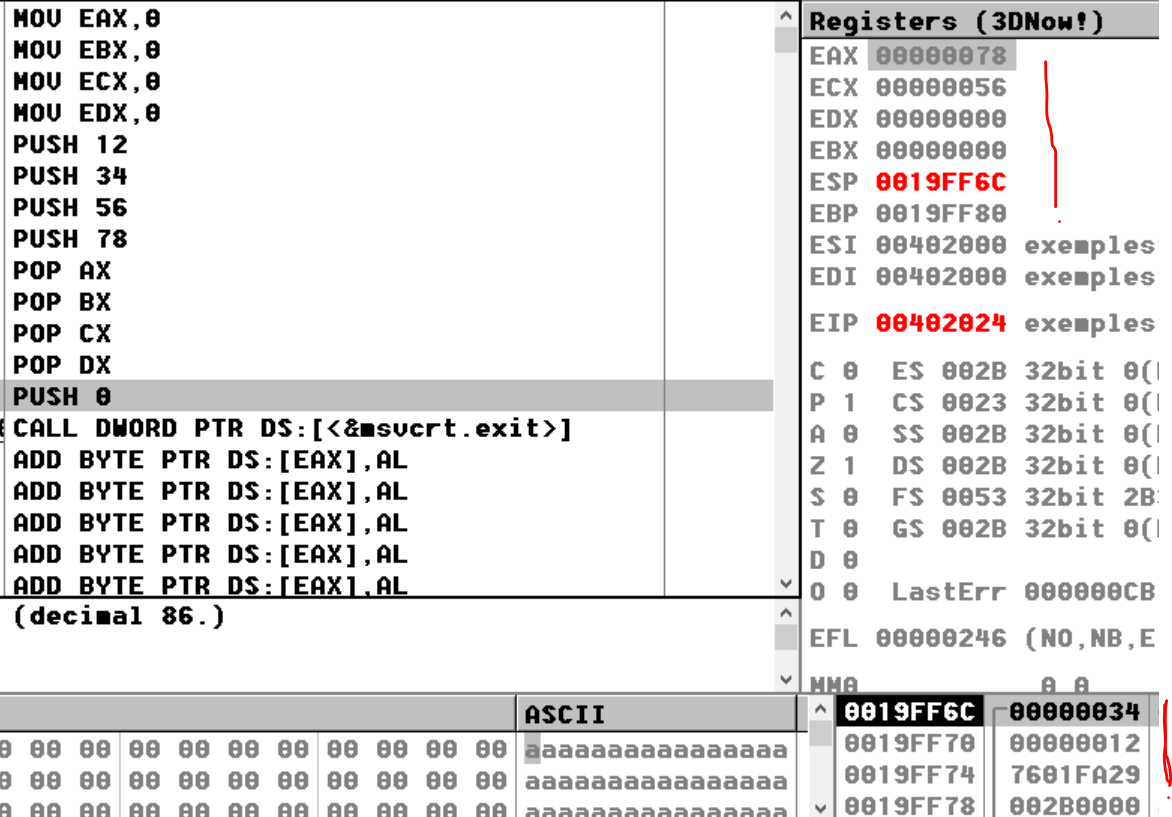
**However** we **can try** to take from the stack values on 16 bits:



**The results will be:**

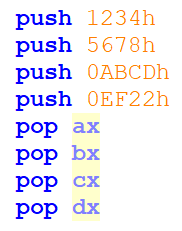


After we execute all our lines of code:

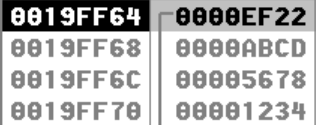


So basically we can observe that our stack is incorrectly align (stiva aliniata incorrect-stiva dezaliniata) because we were using constants and it is **not clear their dimensions**.

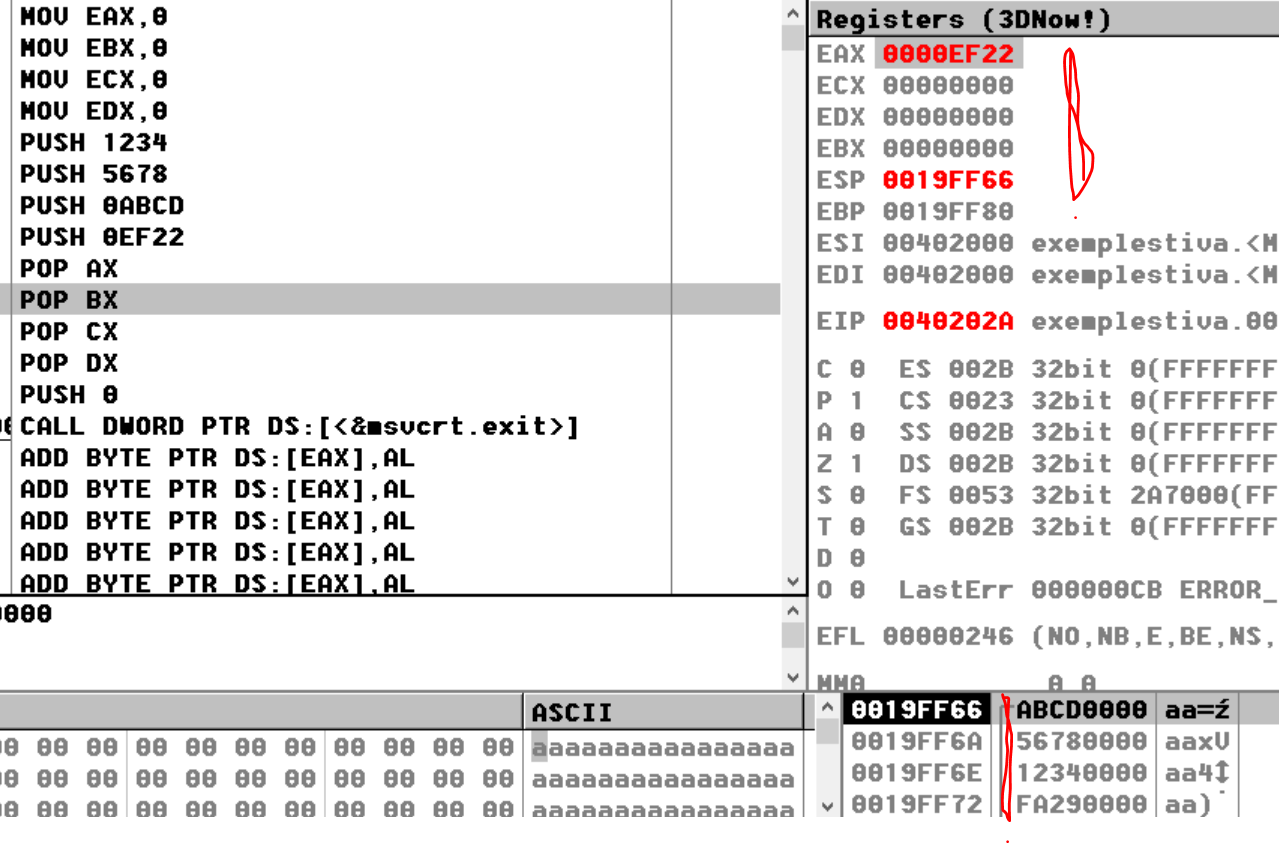
If we will add on the stack values on 4 hexadecimal digits (words):

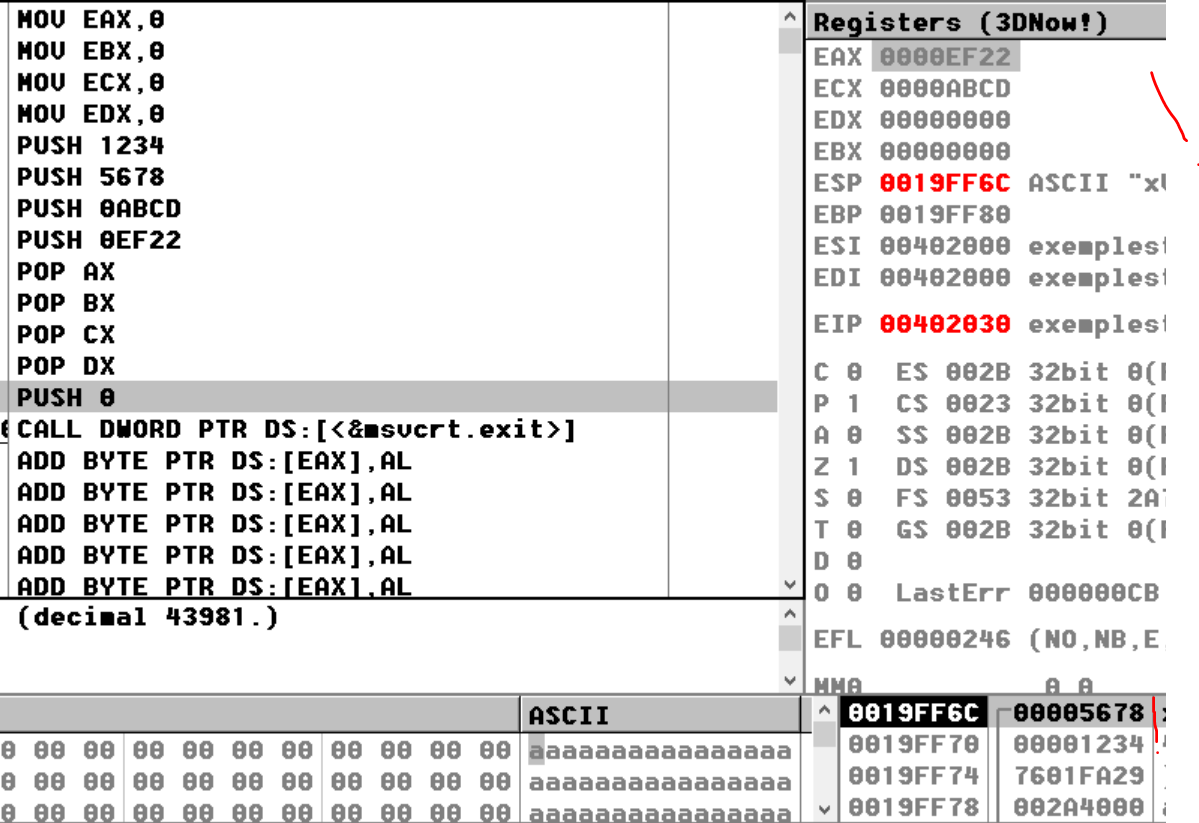


*The stack form after all the elements are on the stack:*



The same problem is here, when we try to “take” the elements from the stack:





So again basically we can observe that our stack is **incorrectly align (stiva aliniata incorrect-stiva dezaliniata)** because we were use constants and it is not clear their dimensions.

**As a conclusion! The stack works properly only with registers, therefore we can save on stack values from regsiters on 16bits or on 32 bits and we can take from stack values on 16 or 32 bits in registers!**

# PUSHFD and POPFD Instructions

The PUSHFD instruction pushes the 32-bit EFLAGS register on the stack, and POPFD pops the stack into EFLAGS:

Syntax:

pushfd

popfd

16-bit programs use the PUSHF instruction to push the 16-bit FLAGS register on the stack and POPF to pop the stack into FLAGS.

The MOV instruction cannot be used to copy the flags to a variable, so PUSHFD may be the best way to save the flags.

There are times when it is useful to make a backup copy of the flags so you can restore them to their former values later.

Often, we enclose a block of code within PUSHFD and POPFD:

pushfd ; save the flags

;

; any sequence of statements here...

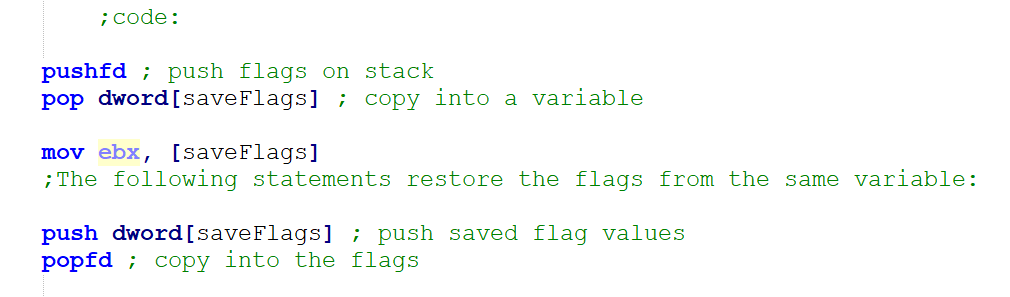
;

popfd ; restore the flags

When using pushes and pops of this type, we need be sure the program’s execution path does not skip over the POPFD instruction.

When a program is modified over time, it can be tricky to remember where all the pushes and pops are located.

The need for precise documentation is critical: A less error-prone way to save and restore the flags is to push them on the stack and immediately pop them into a variable:



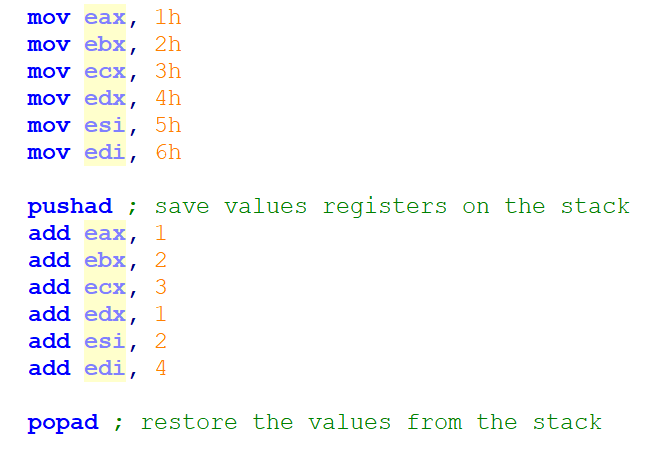
# PUSHAD, PUSHA, POPAD, and POPA Instructions

The PUSHAD instruction pushes all of the 32-bit general-purpose registers on the stack in the following order: EAX, ECX, EDX, EBX, ESP (value before executing PUSHAD), EBP, ESI, and EDI.

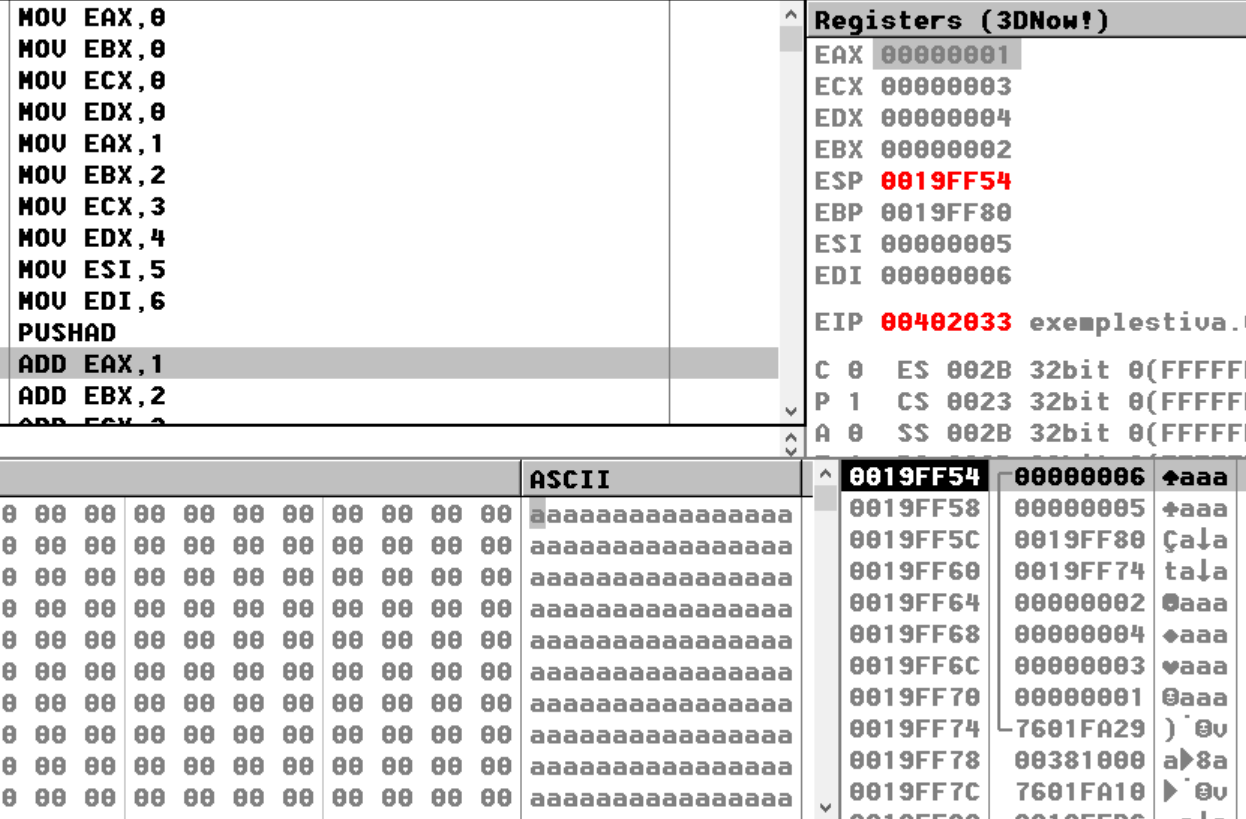
The POPAD instruction pops the same registers off the stack in reverse order.

If we are writing a procedure that modifies a number of 32-bit registers, we use PUSHAD at the beginning of the procedure and POPAD at the end to save and restore the registers.

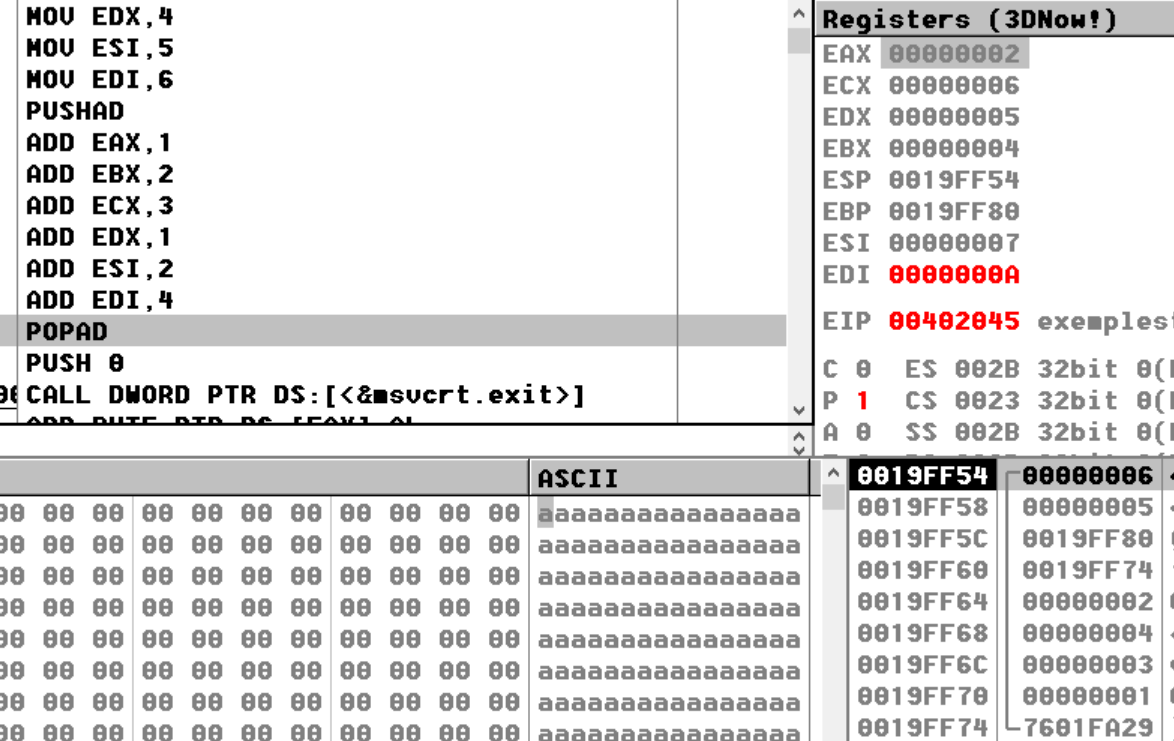
The following code fragment is an example:



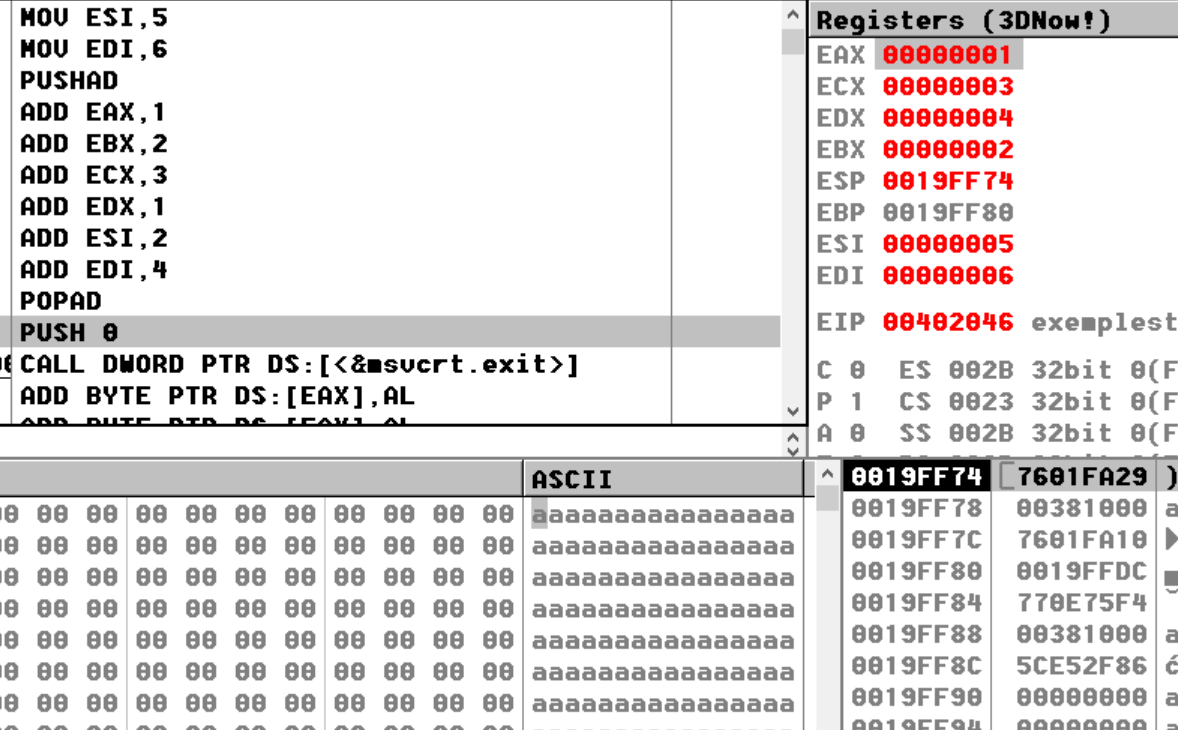
The results after we execute the pushad:



The results after we perform some modifications over the registers:



And the final results after we take the elements from the stack:



Similarly, the PUSHA instruction pushes the 16-bit general purpose registers (AX, CX, DX, BX, SP, BP, SI, DI) on the stack in the order listed. The POPA instruction pops the same registers in reverse order.

**Possible questions for the exams:**

The role of the instructions: push, pushad, pusha, pushfd pop, popad, popa, popfd

The differences between these instructions: push, pushad, pusha, pushfd pop, popad, popa, popfd