Computersystems: Session 3

David Blinder, Jan Cornelis, Stijn Bettens, Tim Bruylants and Peter Schelkens

October 24, 2020

1 PROCEDURES IN ASSEMBLER

High-level programming languages typically use the concept of functions, also called procedures, to group a set of instructions into a meaningful block of code that performs a logical task. This concept can also be used when writing assembler code. As stated before, procedures are blocks of instructions that can be invoked from anywhere in the code. In Intel Assembler syntax, a procedure can be defined as follows in the code:

```
PROC doSomething
push ebp       ; save base pointer (EBP) value on the stack
mov  ebp, esp  ; set stack pointer (ESP) as the new base pointer (EBP)

<procedure instructions>

mov  esp, ebp  ; restore the original stack pointer
pop  ebp       ; retrieve the original base pointer
ret            ; return to next instruction after the call
ENDP doSomething
```

Input and output arguments can be accessed using relative addressing w.r.t. `EBP` (e.g. `mov eax, [ebp+8]`). Arguments can also be passed by registers, but this should be done judiciously. A single numerical output argument is often returned using `EAX`. More details on the workings of this calling convention are given at the bottom of this document, in section 4.
Procedures can then be invoked using the `call` command (e.g. `call doSomething`).

2 TURBO ASSEMBLER IDEAL SYNTAX

It can be very tedious to write correct procedure prologue code (i.e. the pushes and pops at the beginning and end of each procedure). Also, calling a procedure with arguments is tricky, as it is very easy to forget the stack cleanup after the call. For this reason, many assembler tools provide some functionality to shift the responsibility of writing correct prologue code from the programmer to the tool.

For this course, we rely on the IDEAL mode of Turbo Assembler to simplify the handling of prologue code. In essence, this allows the programmer to focus on writing the actual code, instead of maintaining the stack and tracking arguments and local variables.

Most important is the calling of a procedure with arguments in TASM IDEAL:

```
; explicit call, without the help of IDEAL
push 2
push 1
call addNumbers
add  esp, 8

; same call, with help of IDEAL (arguments are pushed right-to-left)
call addNumbers, 1, 2
```

Moreover, IDEAL also writes prologue code of procedures:

```
PROC addNumbers2
 ARG @@numA:dword, @@numB:dword RETURNS eax
 LOCAL @@local1:WORD, @@local2:BYTE ; (example, not used here)
 USES ebx

mov  eax, [@@numA]
mov  ebx, [@@numB]
add  eax, ebx

ret             ; return to next instruction after the call
ENDP addNumbers2
```

`ARG` specifies the input arguments (note: non-dword input arguments may give errors, see compendium for more information). You can optionally define a return argument using `RETURNS`. The `"@@"` signifies that the label is local, meaning it can only be accessed within the procedure and thus reused elsewhere. Local variables are defines with the keyword `LOCAL`. `USES` indicates which registers should be automatically pushed on the stack and restored at the end of the procedure, so that they can be freely used within the procedure. Do not put return registers here (in this case `EAX`), otherwise the result will be overwritten by its initial register value before the procedure call!

Note: Download the template on Canvas for this exercise session.

3 EXERCISES

1. Modify the unsigned number printing procedure to make full use of the IDEAL syntax. The number to be printed should be given via an input argument (not via a register).

2. Make a procedure that prints out an array of (32-bit) unsigned integers separated by commas, given the array offset as an input. The first value contains the array length (`arrlen`), followed by the array itself (`arrdata`). They are guaranteed to be contiguous is memory.

3. Make a procedure that returns the maximum of array in `EAX`.

4. Program a bubble sort procedure that sorts an array in-place. Reference: `https://en.wikipedia.org/wiki/Bubble_sort`.

4 C DECLARATION (IN-DEPTH)

Calling a function involves more instructions than just jumping to a new address – arguments are passed to the callee (the invoked function), space for local variables is reserved, the return address should be accessible by the callee, return values are passed from the callee to the caller (function that invokes another function), and the values of some registers are preserved. A set of rules that specifies how to handle all these actions is called a calling convention. We will adhere to the C declaration (CDECL), which uses the stack to accomplish the function call. The wide use of CDECL in C compilers has allowed programmers to reuse and share functions. Indeed, if programmer A would hand-over to programmer B a function that expects arguments on the stack, and programmer B would place the arguments in global variables in the data memory, then programmer A's function will complete its task using whatever values it can find on the stack. The program will thus not crash, but the output will be wrong. Such bugs are really hard to find, especially if every programmer would define its own (set of) convention(s). Adhering to CDECL is therefore a minimum requirement for your project and you can definitely expect questions about this convention during the project defence. Follow therefore the procedure that is explained in Figures 4.1 to 4.11 attentively.
Let us consider the C function declaration in listing 1.

Listing 1: MyProc.h

```
int MyProc(int ARG1, int ARG2)
```

The main function (the caller) calls MyProc (the callee) with 2 arguments and expects a return value. The type `int` comprises dwords, just as all other registers that we will push on the stack because words are by default extended to dwords. Both the caller and callee have to accomplish some tasks before and after the execution of the function body. To indicate who excecutes what instructions in what place, we adopt the following colour scheme.

- caller prologue
- callee prologue
- caller epilogue
- callee epilogue

The state of the stack before the function call is shown in Figure 4.1. The Extended Stack Pointer (ESP) points to an address that is a multiple of 4 bytes, but is furthermore unspecified. We assume that the stack has sufficient space to grow.

Figure 4.1: The initial state of the stack.

The function call is initiated by the caller. He pushes the arguments on the stack in reverse order (i.e., first ARG2 and then ARG1). See Figure 4.2 for a graphical representation.

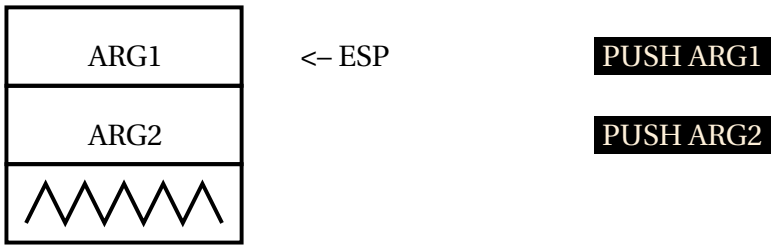

Figure 4.2: The caller pushes the arguments in reverse order on the stack.

Next, the caller invokes instruction CALL, which is a macro that encapsulates 2 actions. In particular, CALL MyProc pushes the Extended Instruction Pointer (EIP) on the stack and jumps to the function MyProc. (MyProc is a label for the address in the program memory where the corresponding function is stored.)

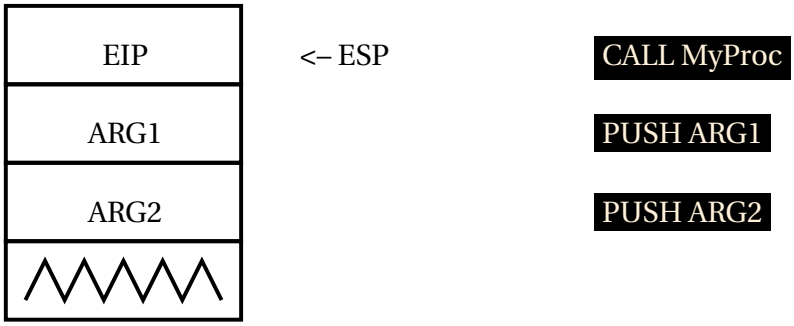

Figure 4.3: The last instruction of the caller prologue is CALL MyProc, which pushes EIP on the stack and jumps to the function MyProc.

The next instruction is invoked by the callee. The Extended Base Pointer is pushed onto the stack in order to preserve its value. The base pointer of the caller should namely be replaced by the base function of callee as long as the callee is active.

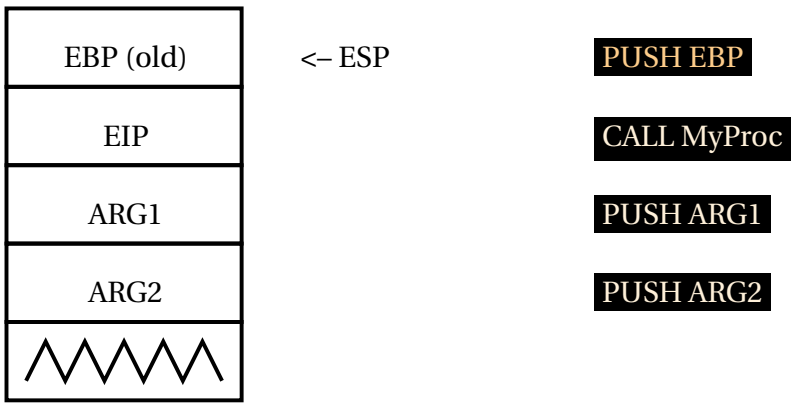

Figure 4.4: The callee prologue starts with PUSH EBP.

The current Extended Stack Pointer (ESP) becomes the new extended base pointer.

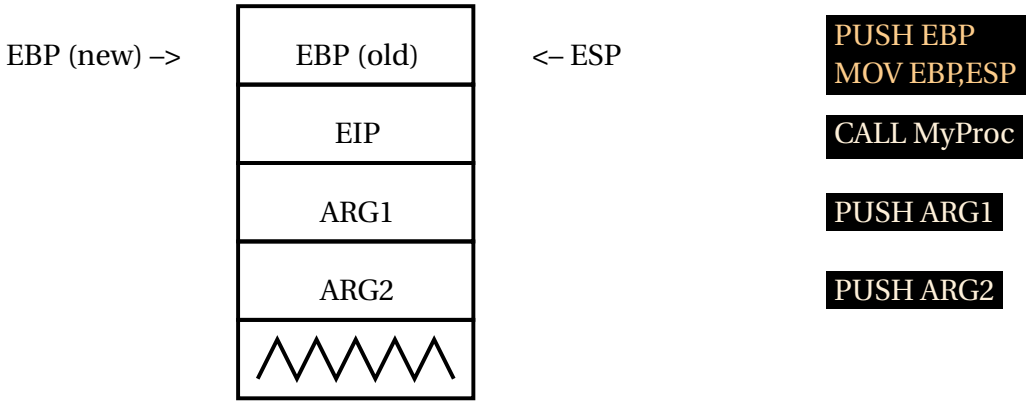

Figure 4.5: Initialize the base pointer for the callee.

The arguments that were pushed on the stack by the caller are at a fixed offset with regard to the base pointer. The callee has thus access to the arguments.

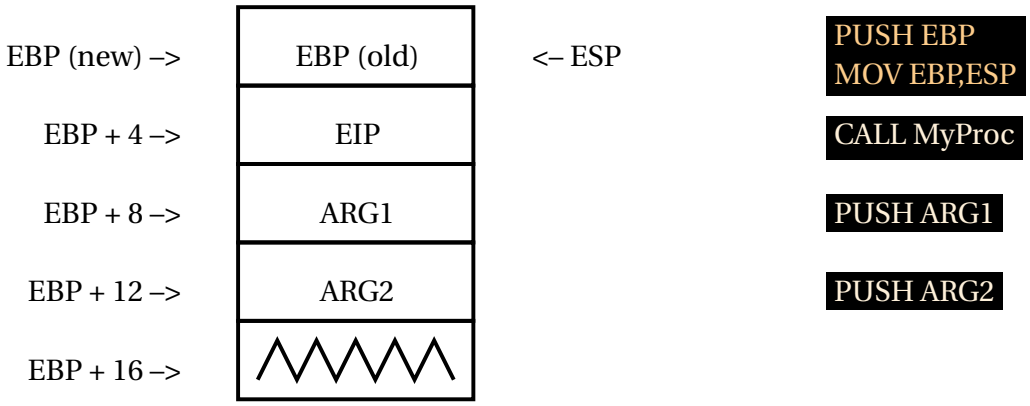

Figure 4.6: The arguments are accessible to the callee through displacement addressing with regard to the base pointer.

Next, the callee reserves space on the stack to store the local variables, which are also at a fixed offset with regard to the base pointer.

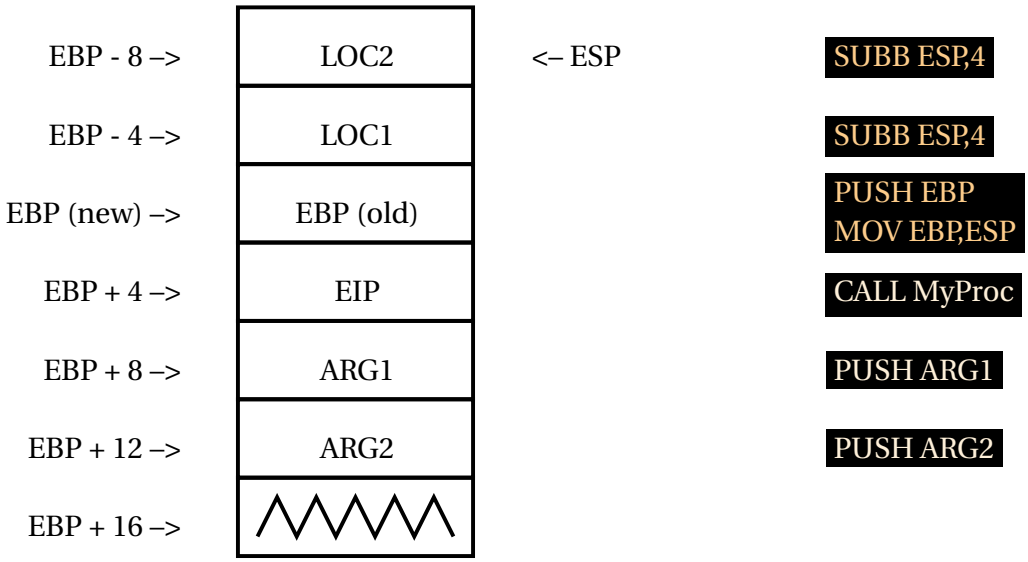

Figure 4.7: Local variables are stored at negative offsets with respect to the base pointer.

CDECL determines that registers EAX, ECX, and EDX are volatile registers, which means that the callee may not preserve their values. If you can preserve one or more of their values, then it is a good practice to do that by pushing them onto the stack in the callee prologue. In the other case, the callee will modify them so that they can act as return values. We add the additional constraint that only register EAX can be used to return values. Registers ECX and EDX must therefore be preserved (i.e., pushed onto the stack). As such, we should not include code to preserve the values of ECX and EDX in the caller, thereby effectively shortening the caller prologue and epilogue. For the non-volatile registers you have to follow this rule: if the callee alters the value of a register, then that register must be pushed onto the stack in the callee prologue.

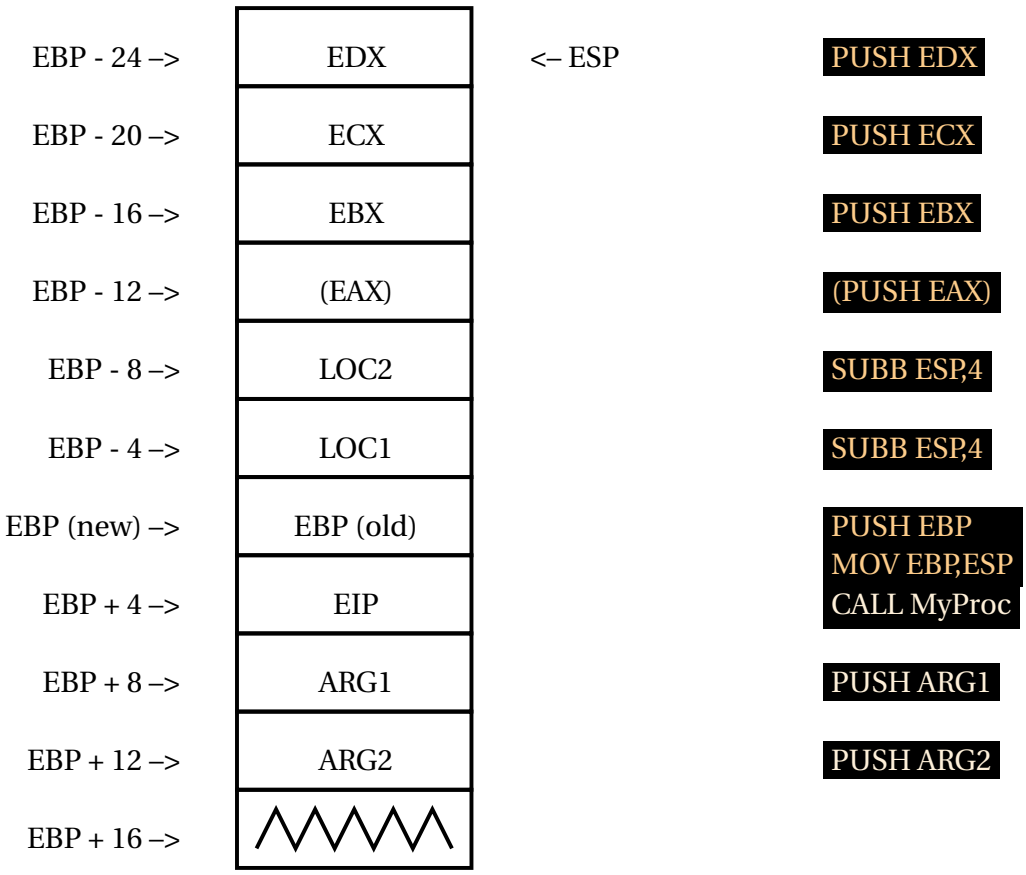

Figure 4.8: All the registers that are altered by the callee must be pushed onto the stack in the callee prologue. Note that ESI and EDI must also be preserved. For register EAX there is an exception if it holds the return value.

The function body is next and when it has finished, the callee has to clean up the part of the stack it built up before. The non-volatile registers that have to be recovered from modifications are up first. They are popped from the stack in reverse order.

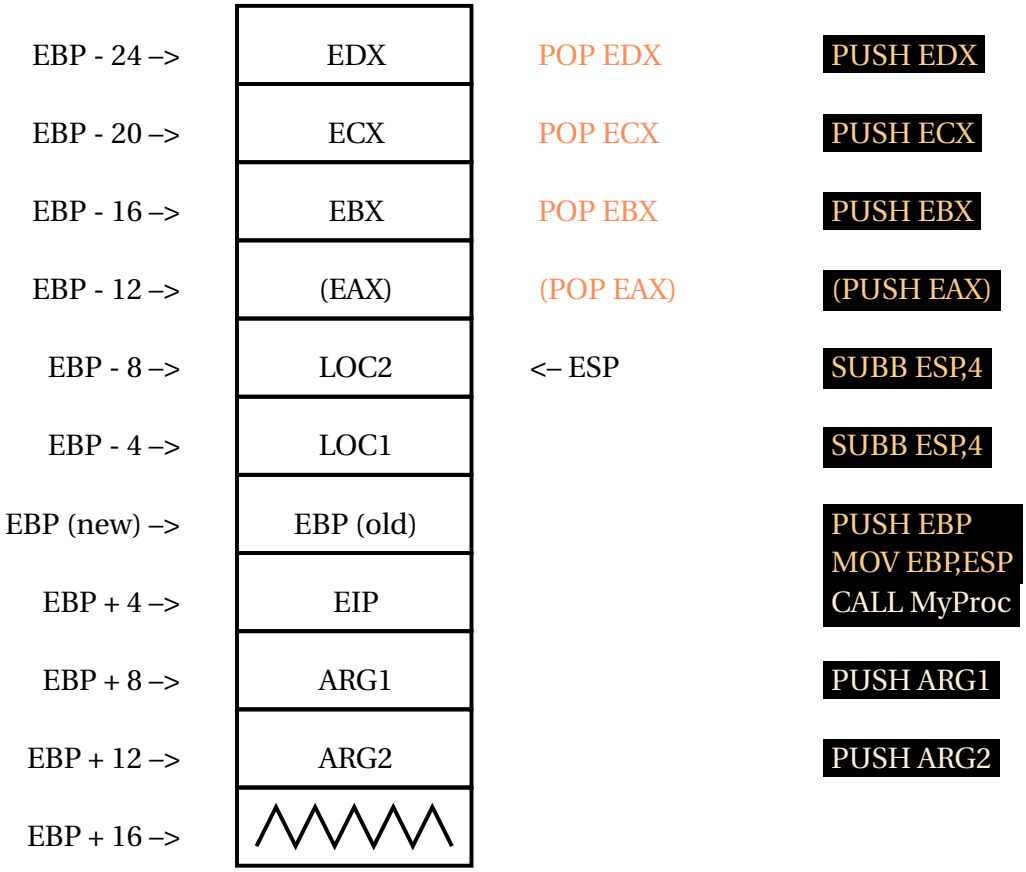

Figure 4.9: The callee recovers the non-volatile registers that were altered.

Up next are the local variables, which are rendered meaningless by copying the base pointer in the stack pointer (all values at lower addresses than the top of the stack have no meaning). The base pointer of the caller is recovered next and the callee is terminated with the instruction RETI, which pops the return address in EIP from the stack, and makes the jump to the caller. At the end of this sequence of three instructions, the stack pointer points to the arguments.

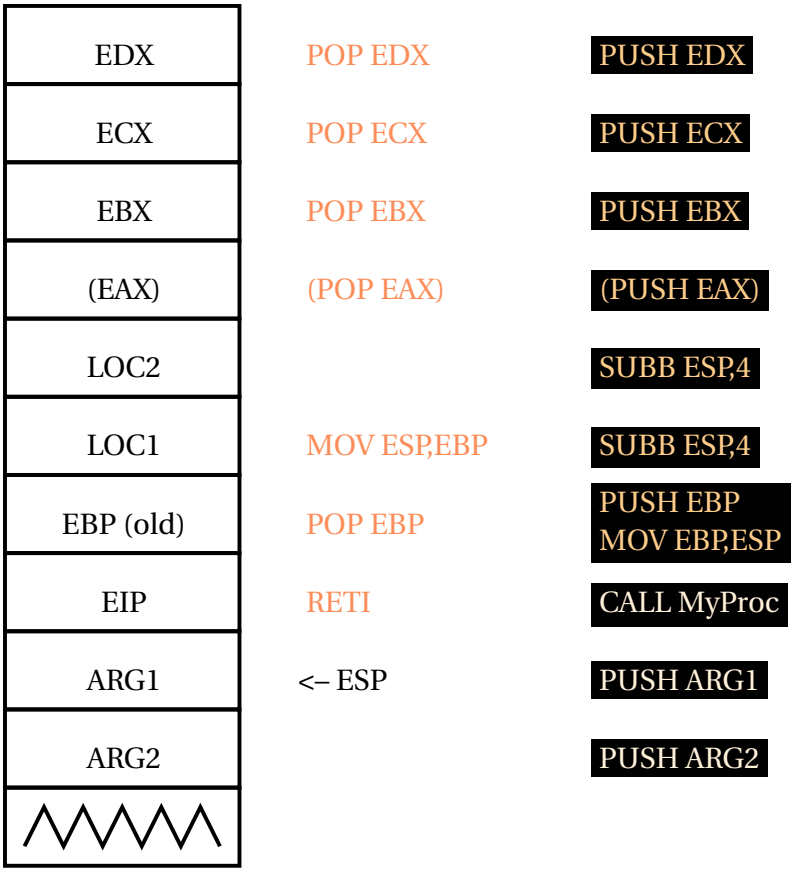

Figure 4.10: The callee terminates and the next instruction is from the caller.

In the final step, the arguments must be removed. The caller can pop them from the stack, but it is sufficient to move the stack pointer over the arguments with the instruction ADD ESP,(size of arguments) so that the arguments are rendered meaningless. The stack pointer is now at its start position like in Figure 4.1. Once finished, the callee leaves thus no traces on the stack and the caller can resume as if the callee were invisible, except for the return value in register EAX (indien van toepassing).

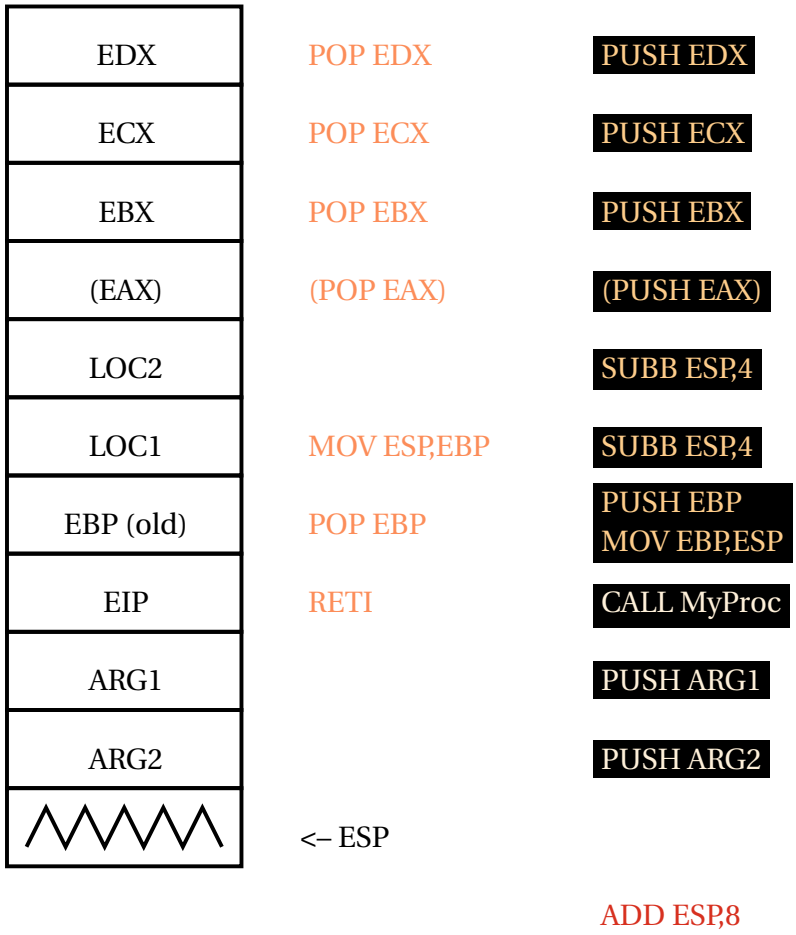

Figure 4.11: The caller must render the arguments meaningless before resuming its activities.