# *Operating systems*

## C function call conventions and stack

*Spring 2016*
*Francesco Fontanella*

# Functions (call) and stack

- When a function is called at run time, it is necessary to allocate memory for parameters and local variables

- How does the compiler arrange the stack when a function has to be called ?

# Example

```
int main (int argc, char* argv[])
{
   int a;
        .
        .
        .
   a = foo (10, 20, 30);
        .
        .
        .

}

int foo (int arg1, int arg2, int arg3)
{
   int loc1, loc2;
        .
        .
        .

}
```

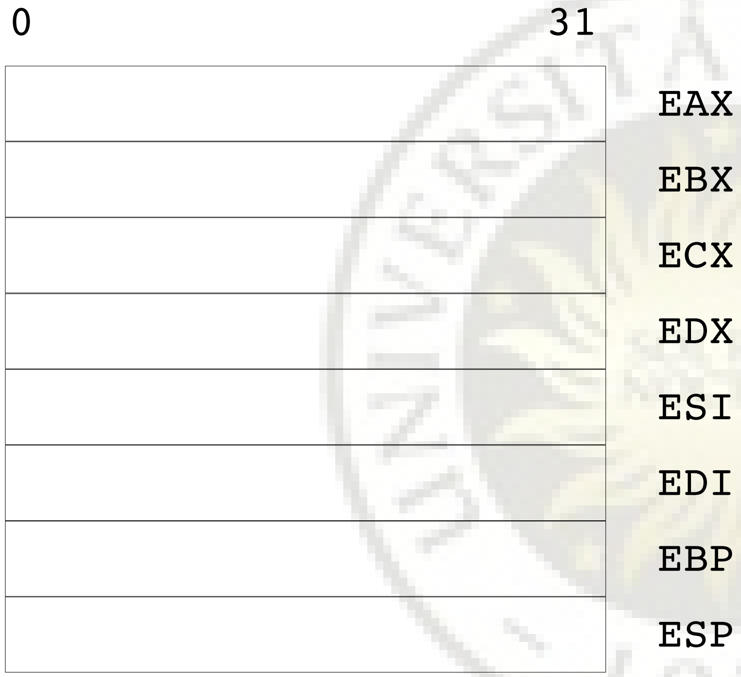
`main` is the “caller”

`foo` is the “callee”

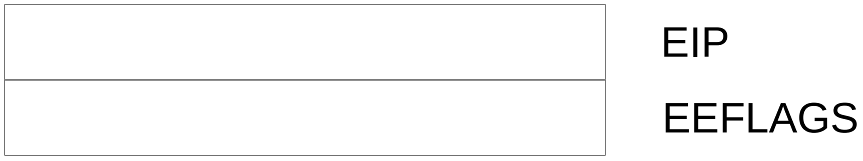
- In the following we will assume
  - sizeof(int): 4
  - compiler: gcc
  - OS: linux
  - CPU: : x86
  - The callee can modifiy the values of the EAX, ECX and EDX registers

# Registers

## General purpose registers



## Status and control registers



# X86 Instructions (assembly)

```
mov <reg>,<reg>
mov <reg>,<mem>
mov <mem>,<reg>
```

```
push <reg32>
push <mem>

pop <reg32>
pop <mem>
```

```
add <reg>,<reg>
add <reg>,<mem>

sub <reg>,<reg>
sub <reg>,<mem>

inc <reg>
inc <mem>
```

http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

- ESP register
  - stack pointer register
  - It contains the address of the top of the stack

- EBP register
  - It is a “base pointer”
  - It represents  the reference address for the frame  of the callee function (foo in the example)
  - Through this address, it is possible to refer to the local variables and the arguments of the callee

- ESP register
  - stack pointer register
  - It contains the address of the top of the stack

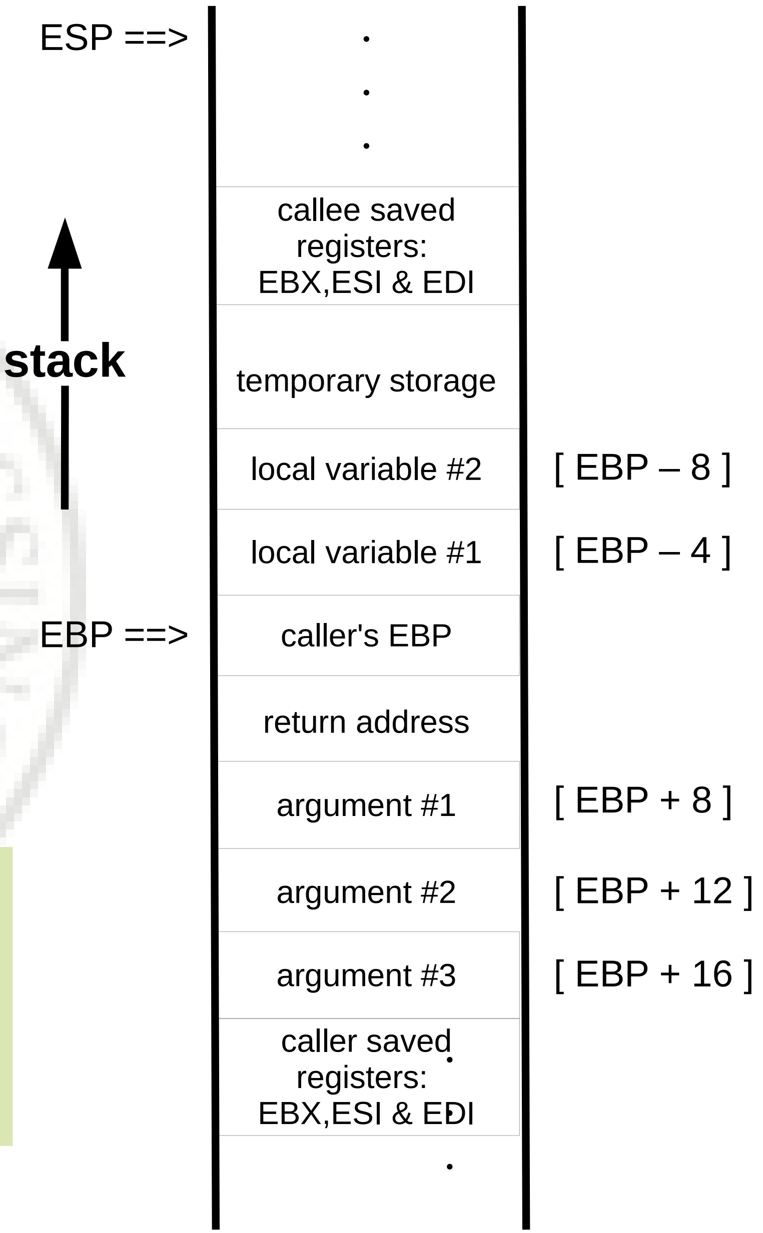
- EBP register
  - It is a “base pointer”
  - It represents  the reference address for the frame  of the callee function (foo in the example)
  - Through this address, it is possible to refer to the local variables and the arguments of the callee

# A typical stack frame

```
int foo (int arg1, int arg2, int arg3)
{
   int loc1, loc2;
         .
         .
         .

}
```

**NOTE**
We also assume that the stack grows upward (smaller address numbers on the top)



ESP ==>

.
.
.

callee saved registers: EBX,ESI & EDI

stack

temporary storage

local variable #2 [ EBP – 8 ]

local variable #1 [ EBP – 4 ]

EBP ==> caller's EBP

return address

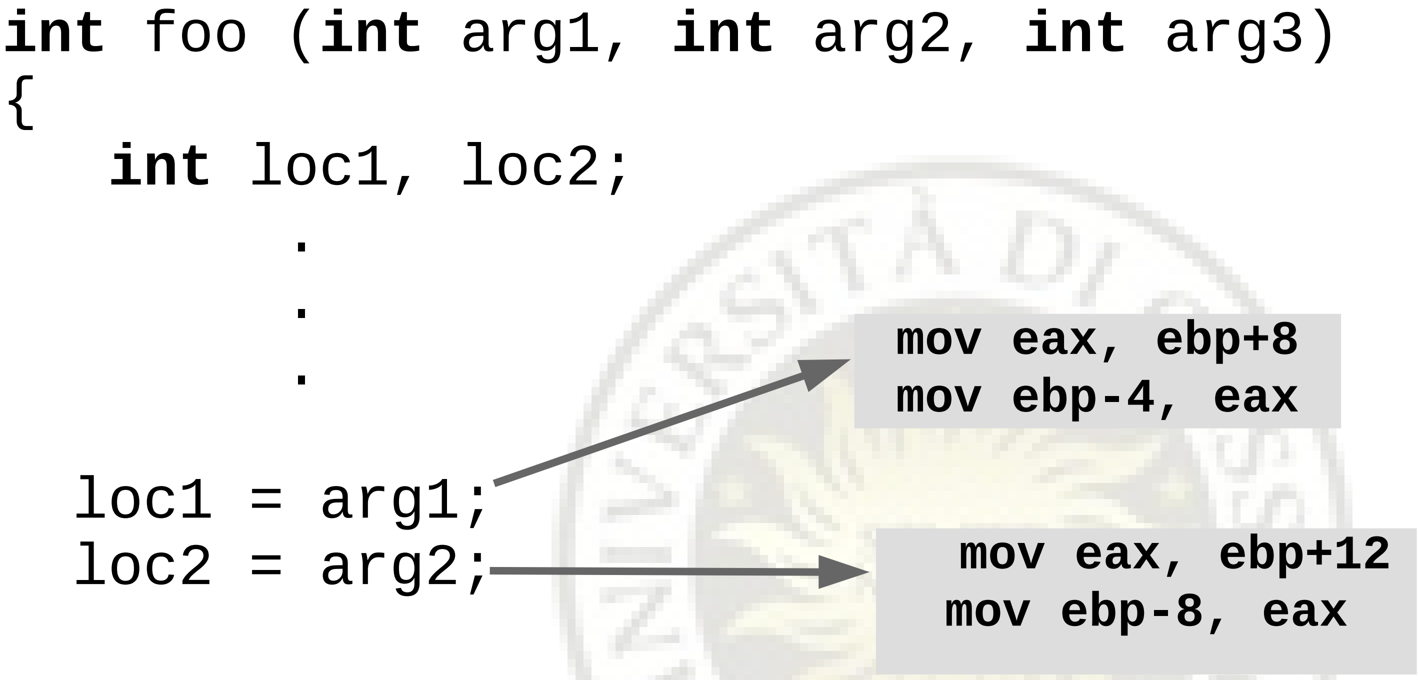
argument #1 [ EBP + 8 ]

argument #2 [ EBP + 12 ]

argument #3 [ EBP + 16 ]

caller saved registers: EBX,ESI & EDI

.
.



```
int foo (int arg1, int arg2, int arg3)
{
   int loc1, loc2;
         .
         .
         .

  loc1 = arg1;
  loc2 = arg2;

         .
         .
         .


}
```

```
mov eax, ebp+8
mov ebp-4, eax
```

```
  mov eax, ebp+12
mov ebp-8, eax
```

## NOTES

– The **mov** assembly instruction copies the data referred to by its second operand into the location referred to by its first operand

– it is not possible to  move directly between memory addresses

# Return values

- Return values of 4 bytes or less are stored in the EAX register
- If a return value with more than 4 bytes is needed, then the caller passes an "extra" first argument to the callee.
- This extra argument is the address of the location where the return value should be stored.
- In practice, the C preprocessor transforms the call

# Return values: example

```
typedef struct {
char name[100];
int ID;
} person;
person p;
    .
    .
    .
p = myfunction(a,b);
```

preprocessor

- `myfunction(&p,a,b);`

# Caller's actions before the function call

- Suppose that in the main there is the function call:

```
a = foo(12, 15, 18);
```

- Before to call the **`foo`** function the main performs the following actions:
  - pushes the contents of the registers EAX, ECX and EDX onto the stack (only if the contents of these 3 registers need to be preserved).
  - Pushes the values 18, 15, 12 onto the stack (reverse order)
- Finally, the main issues the subroutine call function:

```
call foo
```

- When the **`call`** CPU instruction is executed, the EIP (and the EEFLAGS too) is pushed onto the stack: the return address is now on the top of the stack
- The foo function starts its execution,
- Note that before the function call, main is using the ESP and EBP registers for its own stack frame

**NOTE 1**

the assembly instruction

**`push REG`**

copy onto the stack the content of the register REG

```
a = foo(12, 15, 18);
```

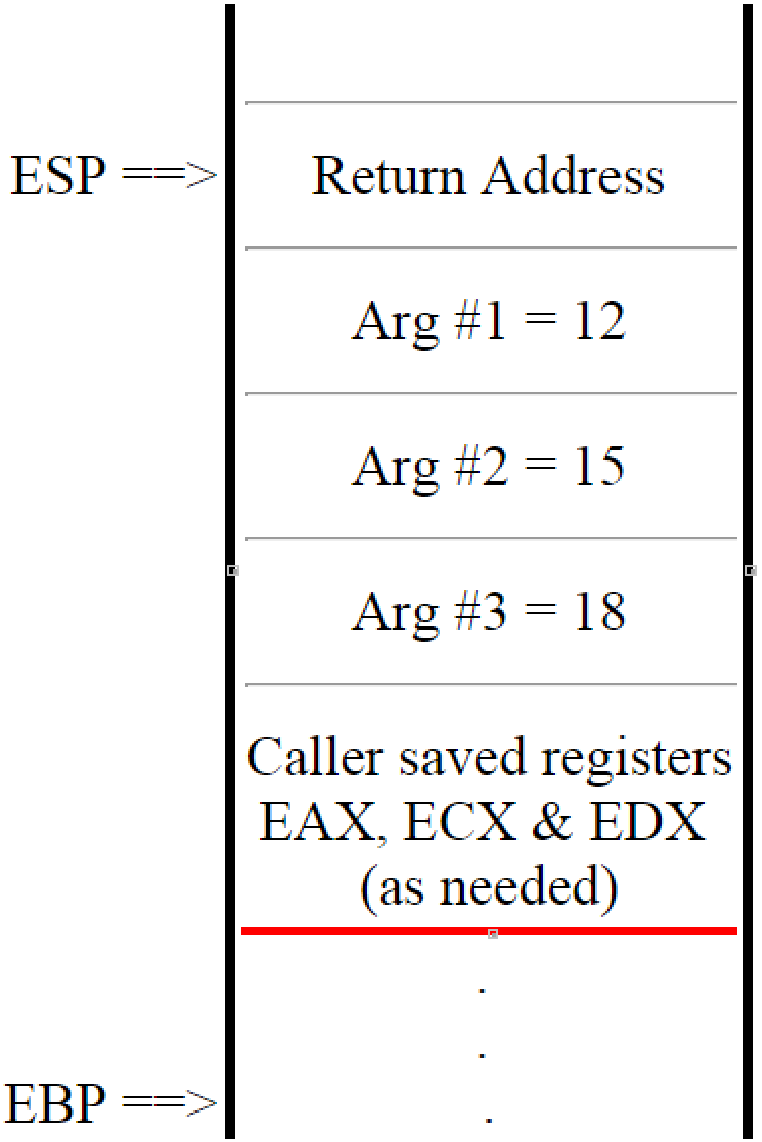
→

```
push   EAX
push   ECX
push   EDX
push   dword 18
push   dword 15
push   dword 12
call   foo
```

**NOTE 2**

The first three instructions (grey shaded) are optional: are executed only if the caller needs to preserve the contentns of these 3 registers

The stack after **`call foo`**

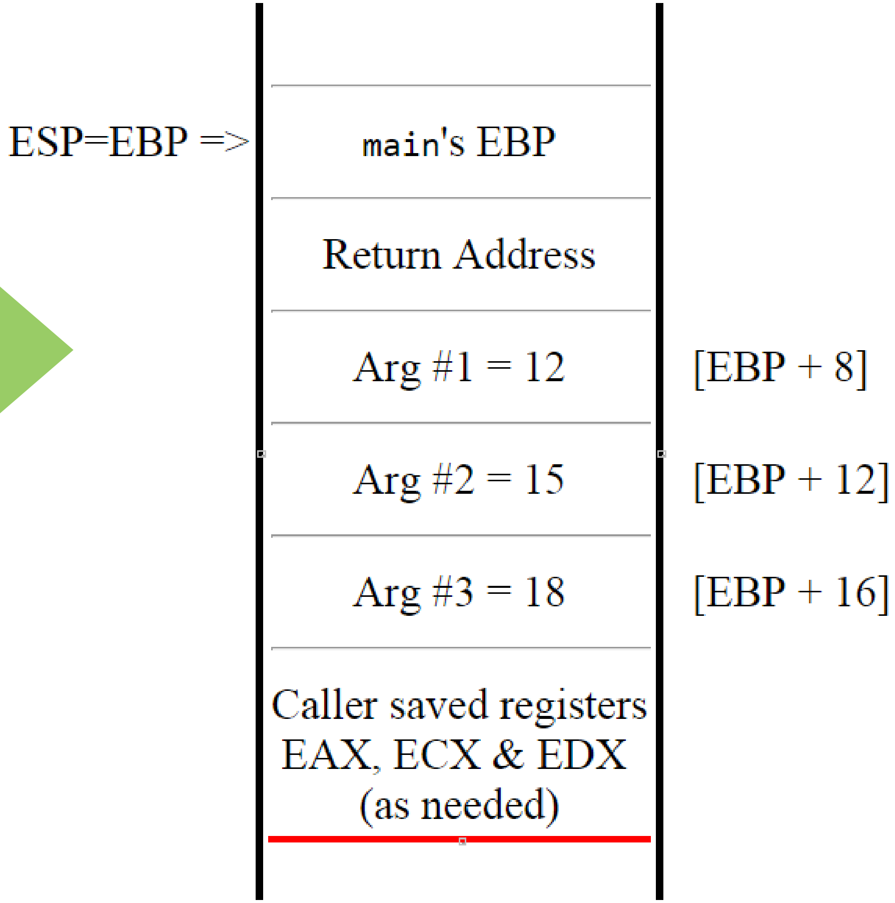


# Callee  actions after the function call

- When the function **`foo`** gets the control, the EBP register points to the base of the main's stack frame: this value must  be saved. It is pushed onto the  stack
- Then the content of the ESP register is copied into the EBP register (EBP update)
- As consequence, (just about) all C functions begin with the two instructions:

```
push   ebp
mov    ebp, esp
```

```
foo:
     push  EBP
     push  EBP, ESP
        .
        .
        .
```



**NOTE**
the address of the first argument is 8 plus EBP, since main's EBP and the return address each takes 4 bytes on the stack.

- In the next step, foo must allocate space for its local variables:
  - Defined local variables: loc1, loc2 (two integers, 8 bytes)
  - Temporary variables: suppose 12 addional bytes are needed
  -
- The 20 bytes needed can be easily allocated:

```
sub esp, 20
```

- Finally it must preserve the contents of the **`EBX`**, **`ESI`** and **`EDI`** registers

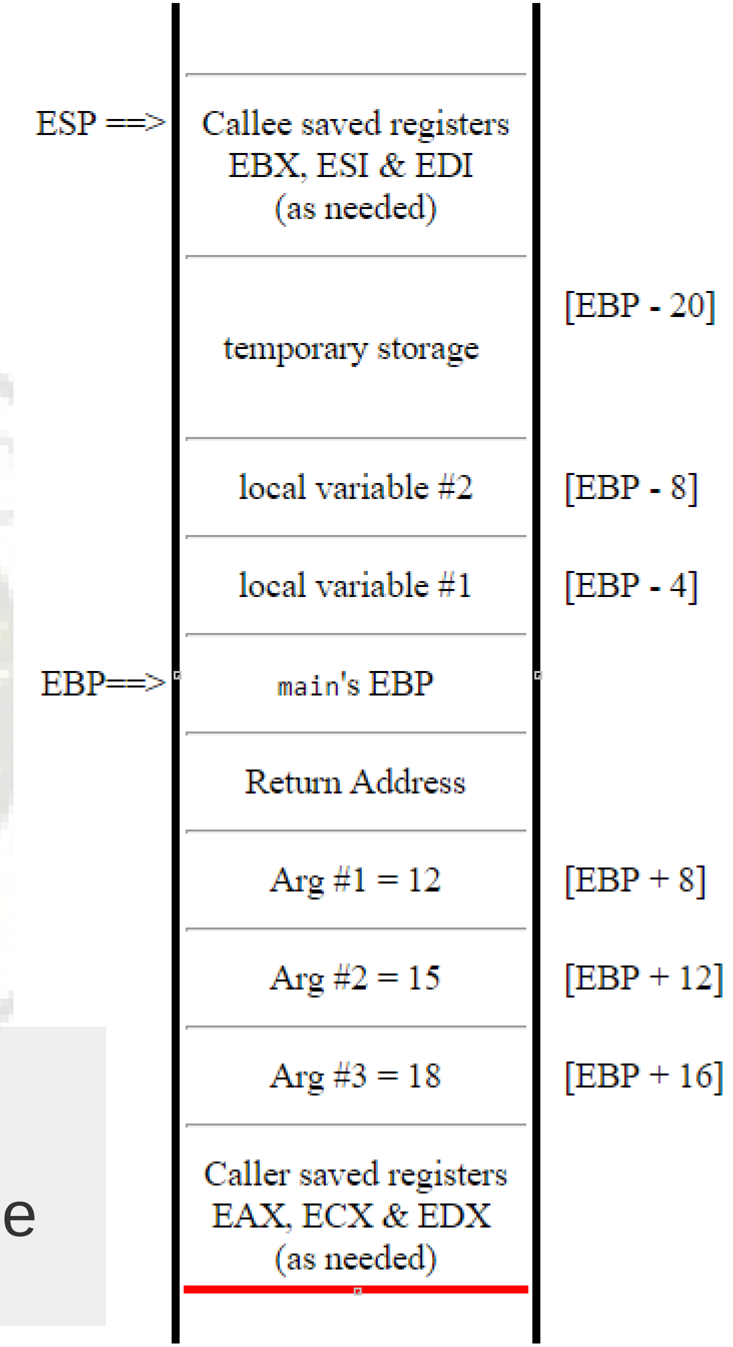
# Temporary variables

- temporary variables are automatically defined by the compiler for storing intermediate values in complicated expressions.
- For example, some C statements in foo might have complicated expressions like this:

```
arg3 = arg2  + ((loc1 + loc2)*arg1)
```

- To compute this expression, the intermediate values of the subexpressions are stored in temporary hidden (to the programmer) variables

```
foo:
      push   EBP
      push   EBP, ESP
      sub    ESP 20
      push   EBX
      push   ESI
      push   EDI
         .
         .
         .
```



ESP ==> Callee saved registers EBX, ESI & EDI (as needed)

temporary storage [EBP - 20]

local variable #2 [EBP - 8]

local variable #1 [EBP - 4]

EBP==> main's EBP

Return Address

Arg #1 = 12 [EBP + 8]

Arg #2 = 15 [EBP + 12]

Arg #3 = 18 [EBP + 16]

Caller saved registers EAX, ECX & EDX (as needed)

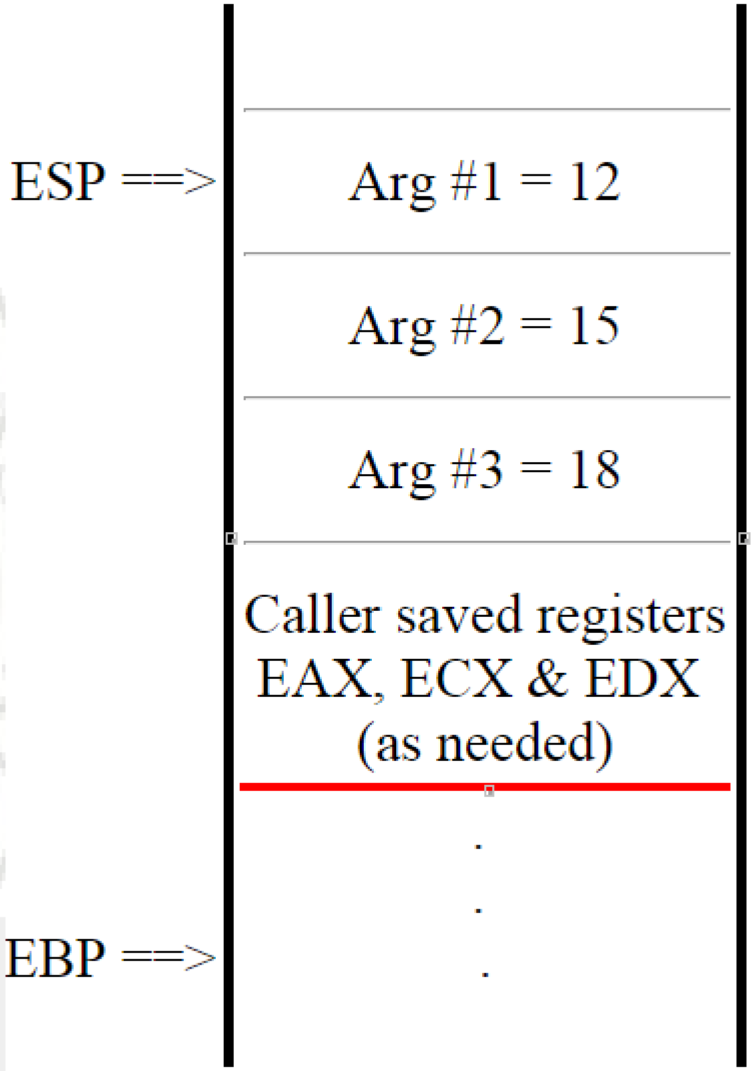
**NOTE**

The last three instructions are optional: are executed only if the callee needs to use these 3 registers

```
foo:

         .
         .
         .

         ret
```



ESP ==>

Arg #1 = 12

Arg #2 = 15

Arg #3 = 18

Caller saved registers
EAX, ECX & EDX
(as needed)

.
.
.

EBP ==>

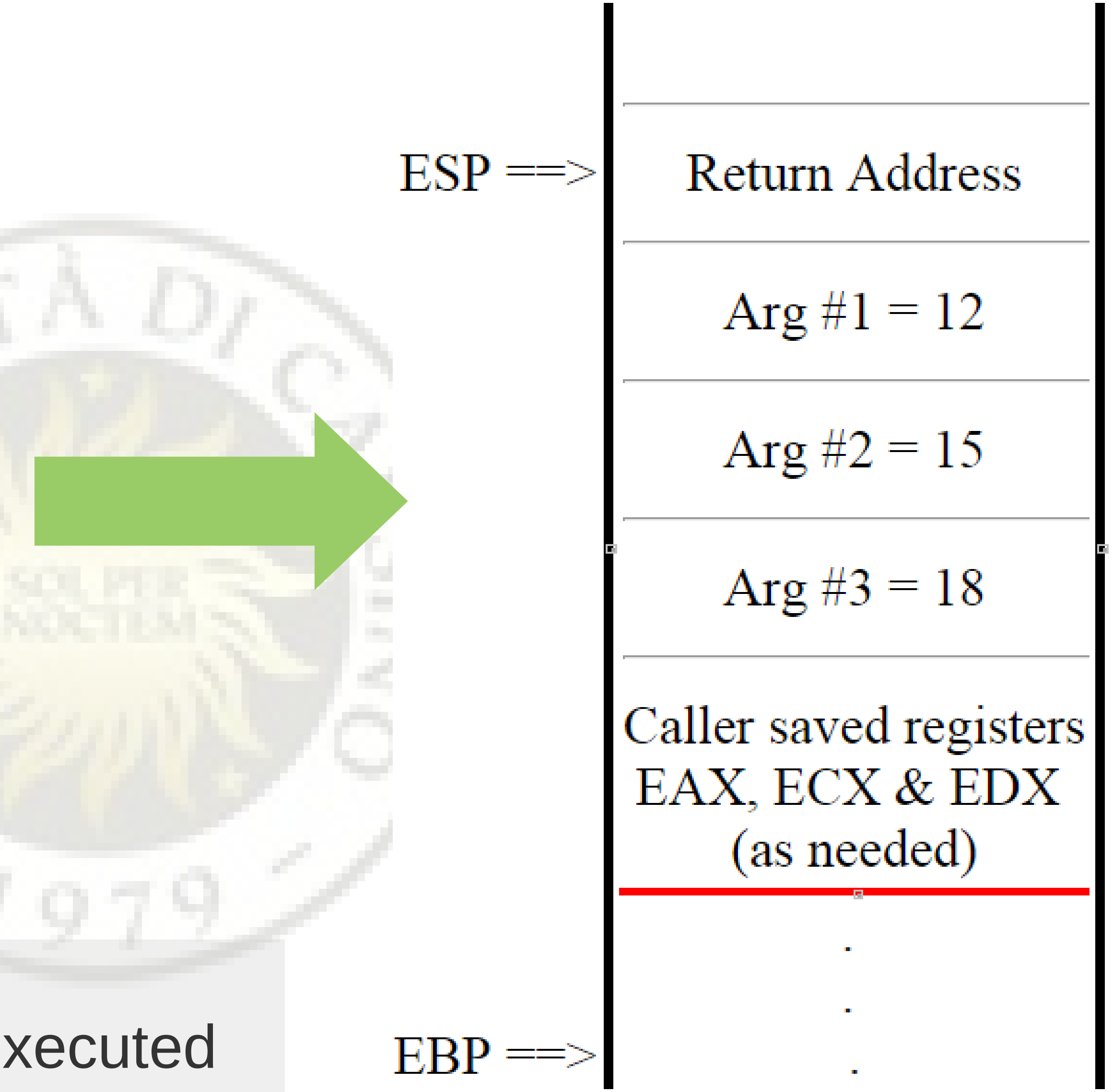
**NOTE**

the x86 ret instruction pops the return address off the stack and stores it in the EIP register

# Callee's  actions before returning

- Before returning the control  to the caller, the callee foo must:
  - Save the return value in the EAX register (4 bytes or less) or in the area pointed by the extra pointer parameter
  - Restore the values of the EBX, ESI and EDI registers (if previously saved)
  - Deallocate the stack memory for local and temp variables: they are no longer needed

```
foo:
        .
        .
        .
    pop  EDI
    pop  ESI
    pop  EBX
    mov  esp, ebp
    pop  ebp
```



ESP ==>

Return Address

Arg #1 = 12

Arg #2 = 15

Arg #3 = 18

Caller saved registers
EAX, ECX & EDX
(as needed)

.
.
.

EBP ==>

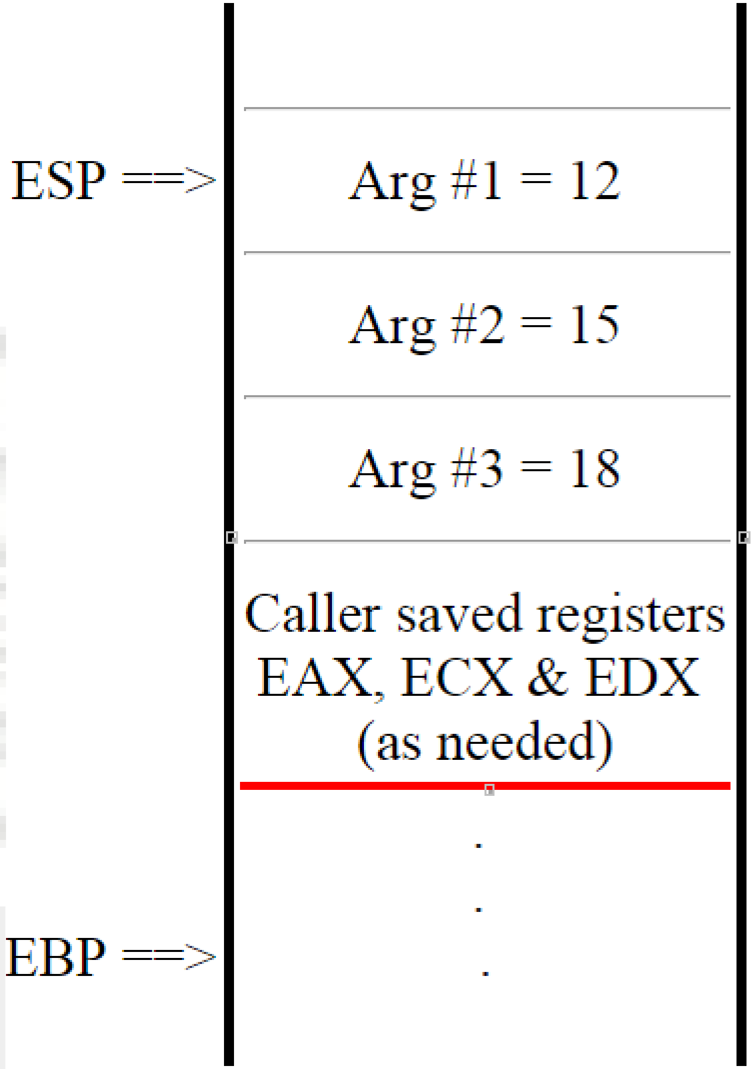
**NOTES**

– the first three instructions are executed only if these registers have been previously saved

```
foo:

        .
        .
        .

        ret
```



ESP ==>

Arg #1 = 12

Arg #2 = 15

Arg #3 = 18

Caller saved registers EAX, ECX & EDX (as needed)

.
.
.

EBP ==>

**NOTE**
the x86 ret instruction pops the caller (the main function) return address off the stack and stores it in the EIP register

# Caller's actions after returning

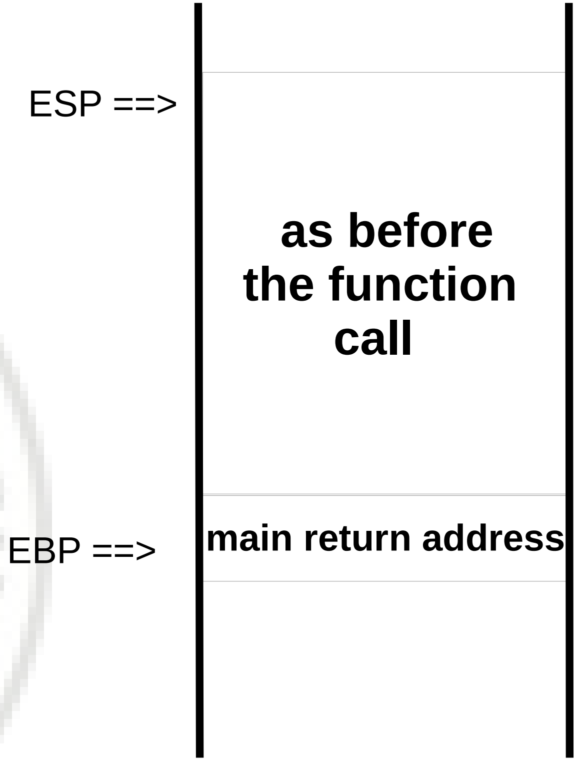
- the arguments passed to foo are no longer needed, and the stack memory can be easily deallocated:

  **`add esp 12`**

- The return value in the EAX register (4 bytes or less) is copied in the appropriate location (x variable address in our example)
- If previously saved, the values of the EAX, ECX and EDX registers are restored
- Then the stack is how it was before the beginning of the entire function call process

```
main:

        .
        .
        .
    add esp, 12
    pop EDX
    pop ECX
    pop EAX
```



ESP ==>

as before the function call

EBP ==>

main return address

**NOTE**

the last 3 instructions are executed only if these registers were previously saved

# Return address of the main

- Now an important question rises:

**where does point the return address of the main function**

## Answer

it points to the libc exit function, which issues the syscall exit of the OS