# x86 Crash Course

With a focus on Linux and a glance to x86_64

Daniele Mammone, daniele.mammone@polimi.it

April 3, 2024

Based on material from Mario Polino, Armando Bellante and Lorenzo Binosi

# The x86 Architecture

# Instruction Set Architecture (ISA)

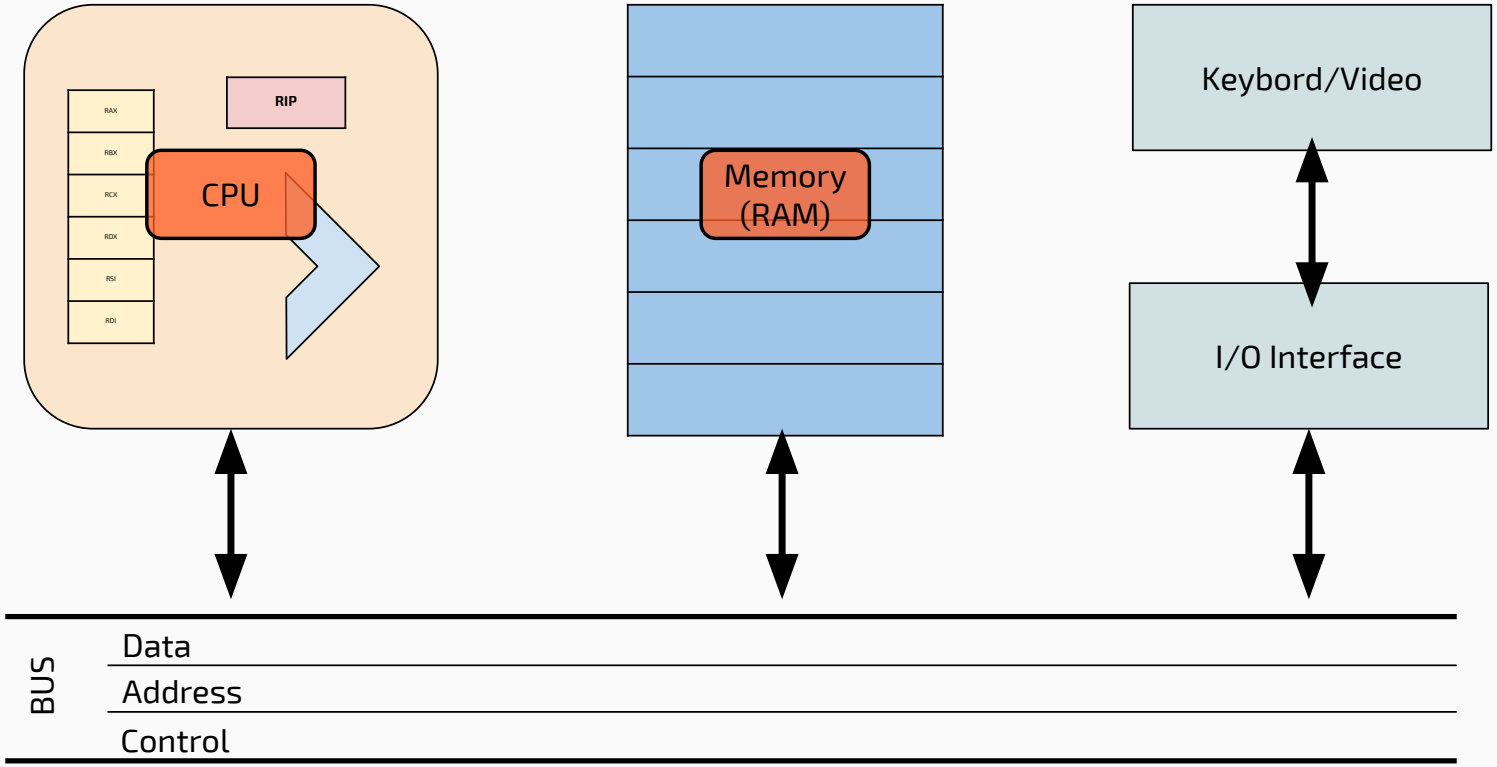
- "Logical" specification of a computer architecture
- Concerned with programming concepts
  - instructions, registers, interrupts, memory architecture, ...
- May differ (widely) from the actual microarchitecture
- Examples:
  - x86 (IA-32 and x86_64)
  - ARM (mobile devices)
  - MIPS (embedded devices, e.g., consumer routers)
  - AVR, SPARC, Power, RISC V, ...

# The x86 ISA

- Born in 1978, 16-bit ISA (Intel 8086)
- Evolved to a 32-bit ISA (1985, Intel 80386)
- Evolved to a 64-bit ISA (2003, AMD Opteron)
- CISC design (e.g., string operations)
- Many legacy features (e.g, segmentation)
- We'll see the basics of the "core" ISA
    - There is also the floating point unit, processor-specific features, and extensions such as SIMD (MMX, SSE, SSE2) with their own instructions and registers[1]

[1] Complete reference: Intel Software Developer's Manual, about 5,000 pages (`https://software.intel.com/en-us/articles/intel-sdm`)

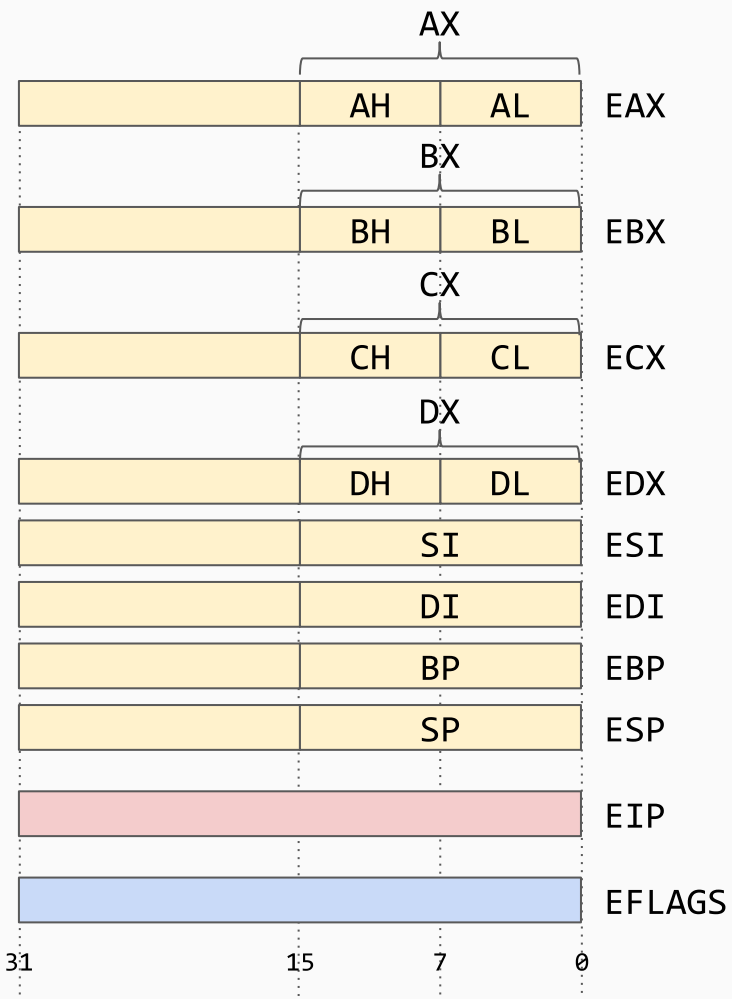
# Von Neumann Architecture



# Memory

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0x8040204... | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F |
| 0x8040203... | E8 | FF | FF | FF | E0 | FF | FF | FF | D8 | FF | FF | FF | D0 | FF | FF | FF |
| 0x8040202... | 50 | 20 | 40 | 80 | 60 | 20 | 40 | 80 | 70 | 20 | 40 | 80 | 80 | 20 | 40 | 80 |
| 0x8040201... | 48 | 65 | 6C | 6C | 6F | 20 | 77 | 6F | 72 | 6C | 64 | 00 | 00 | 00 | 00 | 00 |
| 0x8040200... | 'H' | 'e' | 'l' | 'l' | 'o' | ' ' | 'w' | 'o' | 'r' | 'l' | 'd' | 00 | 00 | 00 | 00 | 00 |

# IA-32: Registers

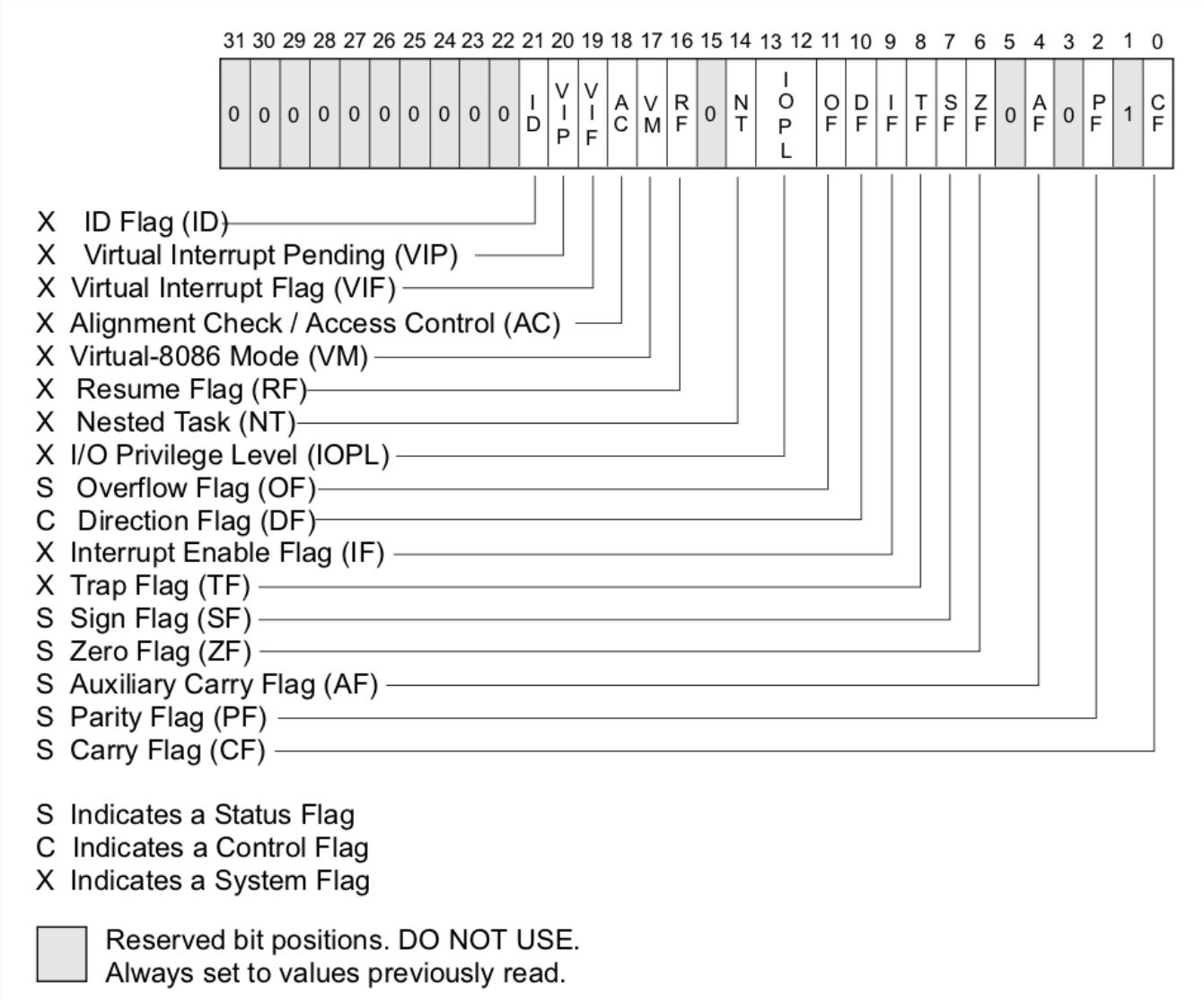



- General-purpose registers
  - `EAX`, `EBX`, `ECX`, `EDX`
  - `ESI`, `EDI` (source and destination index for string operations)
  - `EBP` (base pointer)
  - `ESP` (stack pointer)
- Instruction pointer: `EIP`
  - No explicit access
  - Modified by `jmp`, `call`, `ret`
  - Read through the stack (saved IP)
- Program status and control: `EFLAGS`
- (segment registers)

# IA-32: `EFLAGS` register

- 32-bits register, boolean flags
- **Program status**: overflow, sign, zero, auxiliary carry (BCD), parity, carry
  - Indicate the result of arithmetic instructions
  - Extremely important for control flow
- **Program control**: direction flag
  - controls string instructions (auto-increment or auto-decrement)
- **System**: control operating-system operations

# IA-32: EFLAGS register



# Fundamental data types

**byte** 8 bits

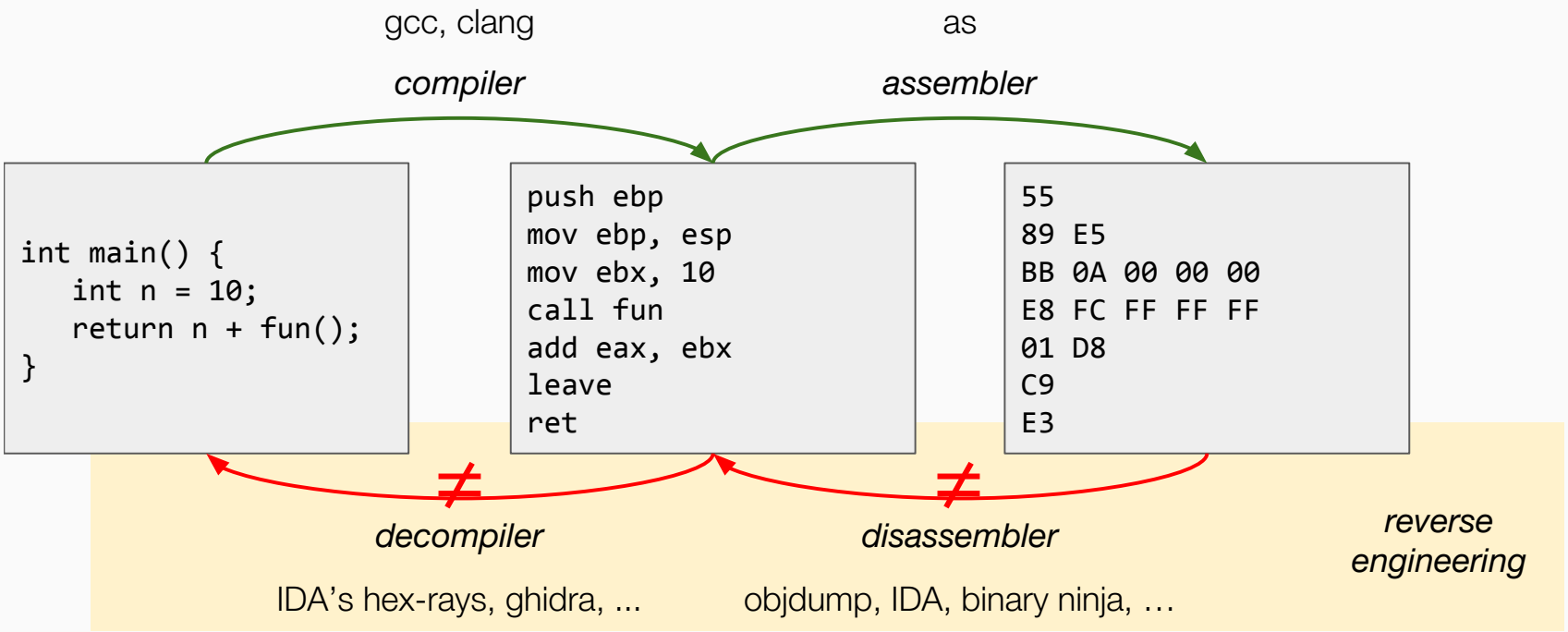
**word** 2 bytes

**dword** Doubleword, 4 bytes (32 bits)

**qword** Quadword, 8 bytes (64 bits)

# Assembly and Machine Code

Assembly language: specific to each ISA, mapped to binary code




For simplicity, we don't deal with the *linking* process.

# Assembly: Syntax

Two main syntaxes:

- **Intel**: default in most Windows programs (e.g., IDA)
- **AT&T**: default in most UNIX tools (e.g., `gdb`, `objdump`)

Beware: The order of the operands is **different**

We will use the Intel syntax

# Assembly: Syntax

*move the value 0 to EAX*

| **Intel** | **AT&T** |
|---|---|
| `mov eax, 0h` | `movl $0x0,%eax` |

*move the value 0 to the address contained in EBX+4*

| **Intel** | **AT&T** |
|---|---|
| `mov [ebx+4h],0h` | `movl $0x0,0x4(%ebx)` |

# x86: data movement

## Examples

Immediate to register:

| | |
|---|---|
| `mov eax, 4h` | EAX = 4 |

Register to register:

| | |
|---|---|
| `mov eax, ebx` | EAX = EBX |

Memory to register (and register to memory):

| | |
|---|---|
| `mov eax, [ebx]` | EAX = *EBX |
| `mov eax, [ebx + 4h]` | EAX = *(EBC + 4) |
| `mov eax, [edx + ebx*4 + 8]` | EAX = *(EDX + EBX * 4 + 8) |

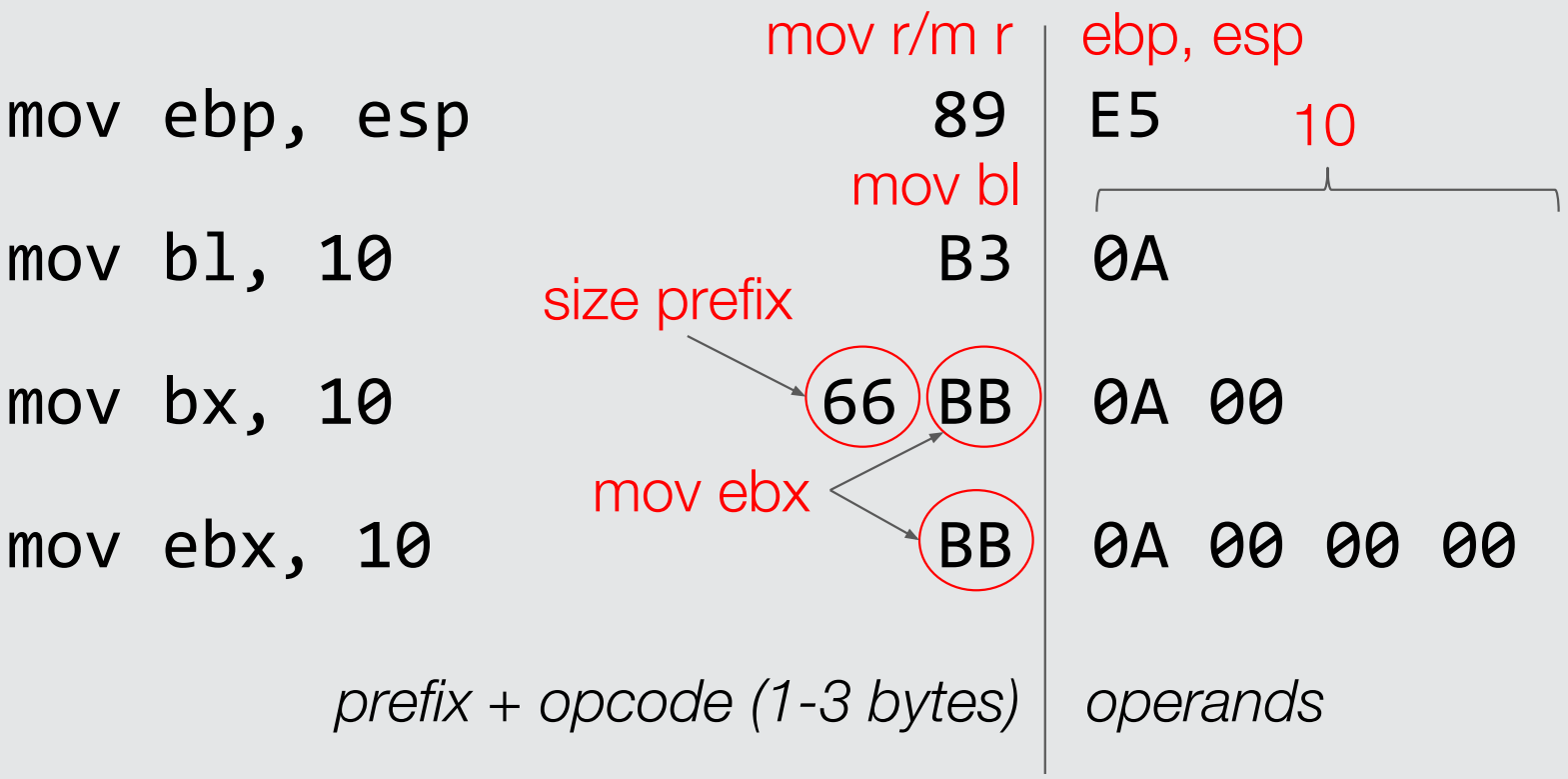
Note: memory to memory is an **invalid** combination[2]

[2] Except in some instructions, such as `movs` (move from string to string).

# x86 Assembly and Machine Code

Instruction = opcode + operand

**Example**




Beware: in x86, instructions have **variable length**.

# Basic instructions

- **Data Transfer**: `mov`, `push`, `pop`, `xchg`, `lea`
- **Integer Arithmetic**: `add`, `sub`, `mul`, `imul`, `div`, `idiv`, `inc`, `dec`
- **Logical**: `and`, `or`, `not`, `xor`
- **Control Transfer**: `jmp`, `jne`, `call`, `ret`
- and lots more...

# Data Transfer: `mov`

- `mov` **destination**, **source**

  **source**: immediate, register, memory location

  **destination**: register or memory location
- Basic load/store operations
  - Register to register, register to memory, immediate to register, immediate to memory
  - Memory to memory is INVALID (in every instruction)

**Examples**

| | | |
|---|---|---|
| `MOV eax, ebx` | `MOV eax, FFFFFFFFh` | `MOV ax, bx` |
| `MOV [eax],ecx` | `MOV [eax],[ecx]` NO!!! | `MOV al, FFh` |

# Integer Arithmetics: `add` and `sub`

| `add` **destination**, **source** | `sub` **destination**, **source** |
|---|---|
| dest $\leftarrow$ dest $+$ source | dest $\leftarrow$ dest $-$ source |

- Addressing:
  **source**: immediate, register, memory location
  **destination**: register or memory location
  (the destination has to be at least as large as the source)
- Negate a value: `neg` [op]
- Bitwise operations: `and`, `or`, `xor`, `not` work similarly

**Examples**

| | | |
|---|---|---|
| `add esp, 44h` | `add edx, cx` | `add al, dh` |
| `sub esp, 33h` | `sub eax, ebx` | `sub [eax], 1h` |

# Integer Arithmetics: unsigned multiply (`mul`)

- `mul` **source**

  **source**: register or memory location
- dest $\leftarrow$ implied_op $\times$ source
- **Implied operands** according to the size of **source**
  - First operand: `AL`, `AX`, or `EAX`
  - Destination: `AX`, `DX:AX`, `EDX:EAX` (double the size of **source**)
- Signed multiply: `imul`

**Example**

- `mul ebx`: `EDX:EAX` $\leftarrow$ `EAX * EBX`
  - most significant bits of the result in `EDX`
  - least significant bits of the result in `EAX`
- `mul cx`: `DX:AX` $\leftarrow$ `AX * CX`
- `mul cl`: `AX` $\leftarrow$ `AL * CL`

# Integer Arithmetics: unsigned divide (`div`)

- `div` **source**

  **source**: register or a memory location
- Computes quotient and remainder
- Implied operand: `EDX:EAX` (according to the size of **source**)
- Signed divide: `idiv`

**Examples**

- `div ebx` (4 bytes)
  - `EAX` ← `EDX:EAX / EBX`
  - `EDX` ← `EDX:EAX % EBX`
- `div bx` (2 bytes)
  - `AX` ← `DX:AX / BX` `DX = DX:AX % BX`
- `div bl` (1 byte)
  - `AL` ← `AX / BX` `AH = AX % BX`

# Integer Arithmetics: `cmp` and `test`

| `cmp` **op1**, **op2** | `test` **op1**, **op2** |
|---|---|
| Computes op1 − op2 | Computes op1 & op2 |

- Sets the flags (`ZF`,`CF`, `OF`, ...)
- Discards the result

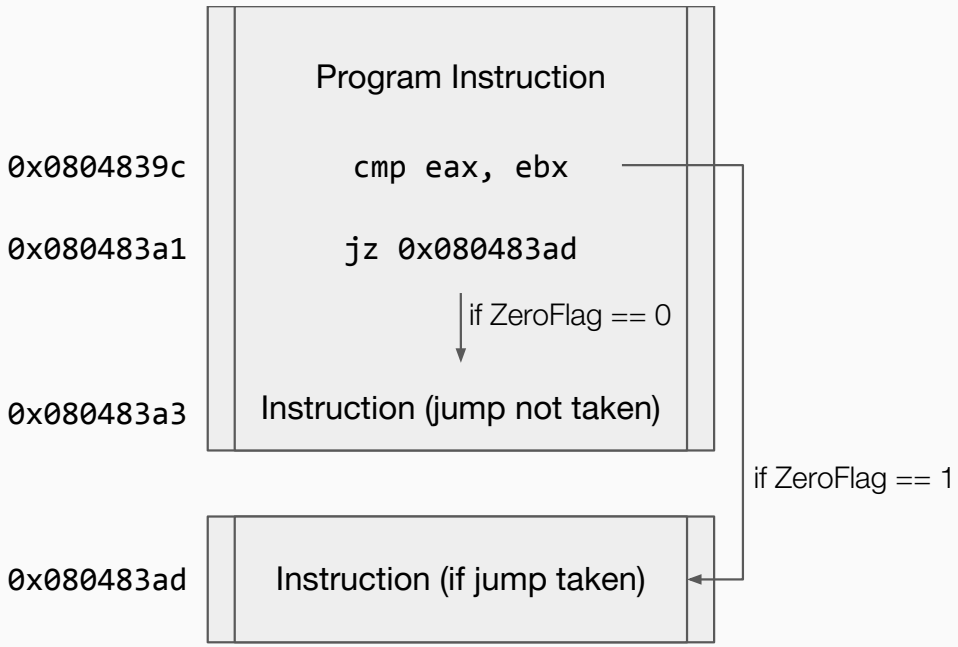
**Examples**

| | | |
|---|---|---|
| `cmp eax, ebx` | `cmp eax, 44BBCCDDh` | `cmp al, dh` |
| `cmp al, 44h` | `cmp ax,FFFFh` | `cmp [eax],4h` |

# Control-Flow Instructions: conditional jumps

`j<cc>` **address** or **offset**

Jump to **address** if and only if a certain condition is verified




`<cc>`: condition

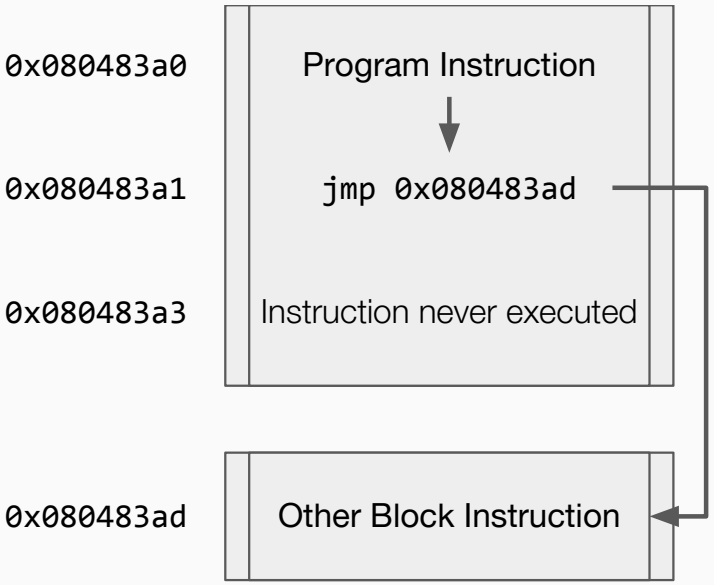
- O,NO,S,NS,E,Z,NE, ...
- based on one or more status flags of `EFLAGS`

Examples:

- `jz` = jump if zero
- `jg` = jump if greater than
- `jlt` = jump if less than

Reference: `http://www.unixwiz.net/techtips/x86-jumps.html`

# Control-Flow Instructions: unconditional jump `jmp`




- `jmp` **address** or **offset**
- Unconditional jump: just set the `EIP` to **address**
- Can be also *relative*: increment or decrement `EIP` by an offset

# Exercise 1

Translate the following C code in assembly x86. Assume EBX $\leftarrow$ b, ECX $\leftarrow$ c. Finally, a goes in EAX.

```
if (c == 0)
    a = b;
else
    a = -b;
```

# Solution

```
    mov edx, 0
    cmp ecx, edx
    jne ELSE
    mov eax, ebx
    jmp ENDIF
ELSE:
    mov eax, 0
    sub eax, ebx
ENDIF:
    nop
    ...
```

# Exercise 2

Translate the following C code in assembly x86. The variable a goes in EAX.

```
a = 0;
for(i = 0; i < 10; i++)
    a += i;
```

# Solution

```
    mov eax, 0
    mov ebx, 0
    mov ecx, 10
LOOP:
    cmp ebx, ecx
    jge END
    add eax, ebx
    inc ebx
    jmp LOOP
END:
    nop
    ...
```

# A very simple example (what does it do?)

Assume that the input is in registers: `ECX` and `EDX`; output: `EAX`

```
    mov eax, ecx
    mov ebx, edx
    cmp ebx, 0
    jz LABEL
LOOP:
    cmp ebx, 1
    jle RET
    mul ecx
    sub ebx, 1
    jmp LOOP
LABEL:
    mov eax, 1
RET:
    ...
```

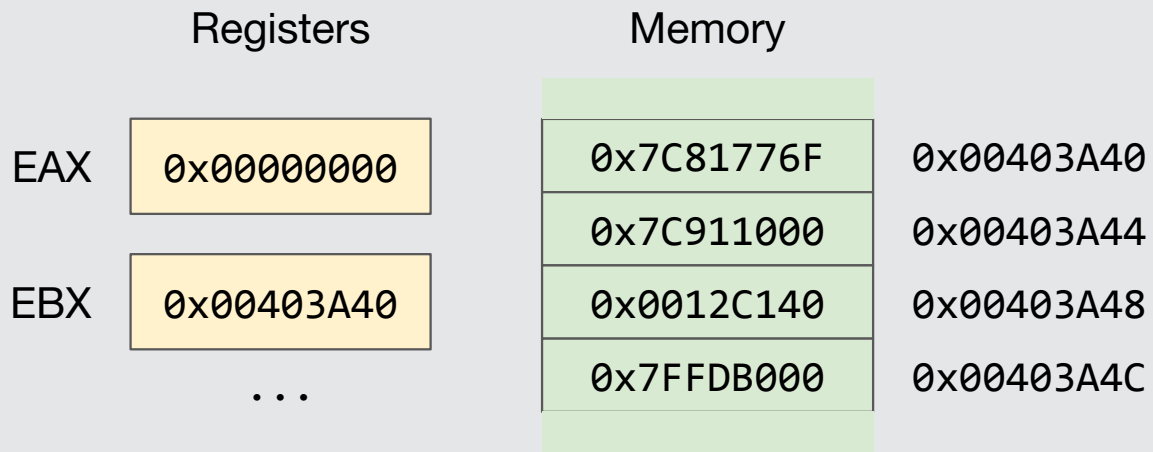
# Load effective address (`lea`)

- `lea` **destination**, **source**

  **source**: memory location

  **destination**: register
- Like a `mov`, but it is storing the pointer, not the value
- It does NOT access memory

## Example




`lea eax, [ebx + 8]` → `EAX = 0x00403A48`

**N.B.**: with `mov eax, [ebx+8]` → `EAX = 0x0012C140`

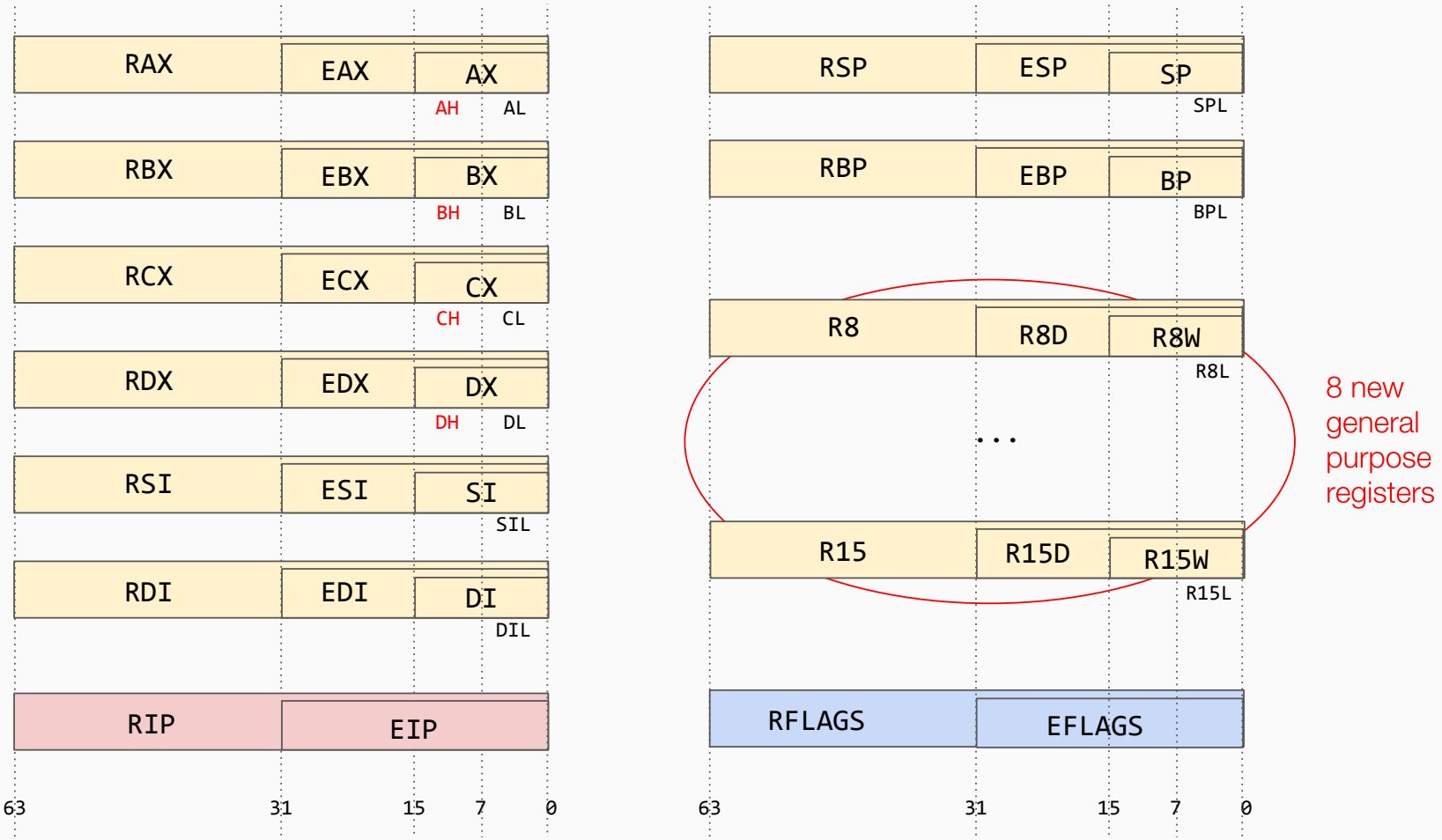
# Basic Instructions: `nop`

- `nop` = **No Operation**. Just move to next instruction.
- The opcode is pretty famous and is `0x90`
- Really useful in exploitation (we will see!)

# Interrupts and Syscalls

- `int` **value**
  - **value**: software interrupt number to generate (0-255)
  - Every OS has its set of interrupt numbers (e.g., `80h` for Linux system calls)
- `syscall` used for Linux 64-bit
- `sysenter` used by Microsoft Windows

# The x86_64 ISA

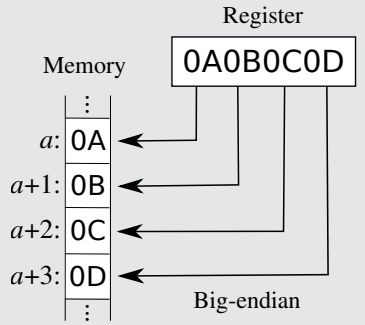


# Endianness

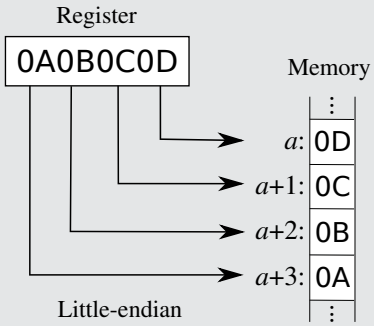
**Endianness**: convention that specifies in which order the bytes of a data word are lined up sequentially in memory.

## Big-endian (left)

Systems in which the *most significant byte* of the word is stored in the *smallest address* given.




## Little-endian




Systems in which the *least significant* byte is stored in the *smallest address*.

IA-32 is “little endian”.

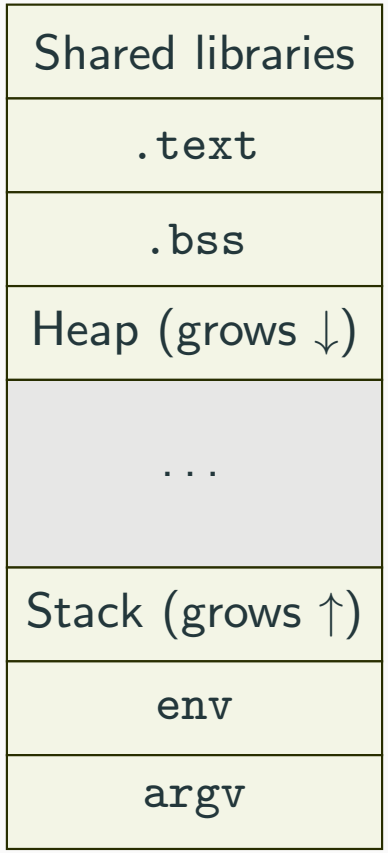
# Program Layout and Functions

# How an executable is mapped to memory in Linux (ELF)

| Executable | Description |
|---|---|
| .plt | This section holds stubs which are responsible of external functions linking. |
| .text | This section holds the "text," or executable instructions, of a program. |
| .rodata | This section holds read-only data that contribute to the program's memory image |
| .data | This section holds initialized data that contribute to the program's memory image |
| .bss | This section holds uninitialized data that contributes to the program's memory image. By definition, the system initializes the data with zeros when the program begins to run. |
| .debug | This section holds information symbolic debugging. |
| .init | This section holds executable instructions that contribute to the process initialization code. That is, when a program starts to run, the system arranges to execute the code in this section before calling the main program entry point (called main for C programs). |
| .got | This section holds the global offset table. |

# Simplified program memory layout

Low addresses (`0x80000000`)




High addresses (`0xbfffffff`)

# The Stack

- LIFO (last in first out) data structure
- Used to manage functions
  - local variables
  - return addresses
  - ...
- Handled through the register `ESP` (stack pointer)
- Remember: the stack grows **toward lower addresses** (downward the address space)

# Stack Management Instructions: `push`

`push` **immediate** or **register**

Stores the immediate or register value at the top of the stack and decrements the `ESP` of the operand size

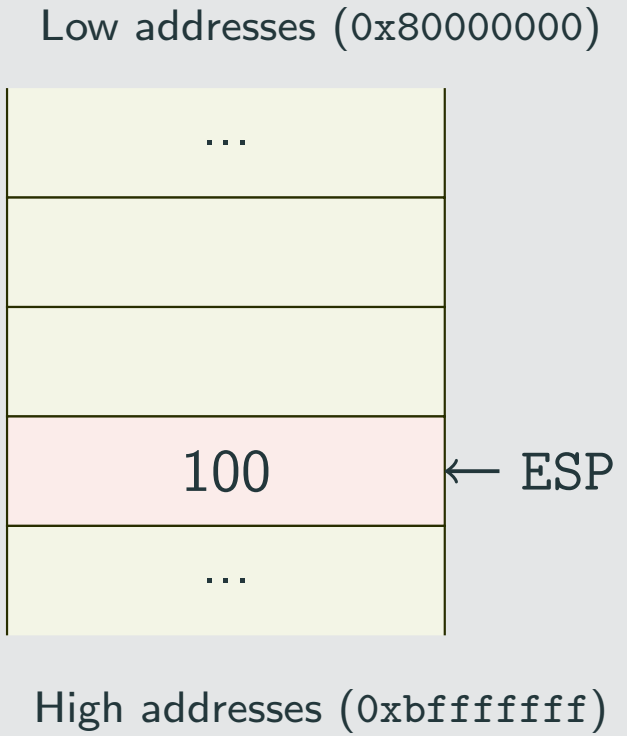
## Example

Initial condition: `EAX` = 30

```
push eax
```

is equivalent to:

```
sub esp, 4
mov DWORD PTR [esp], eax
```



# Stack Management Instructions: `push`

`push` **immediate** or **register**

Stores the immediate or register value at the top of the stack and decrements the `ESP` of the operand size

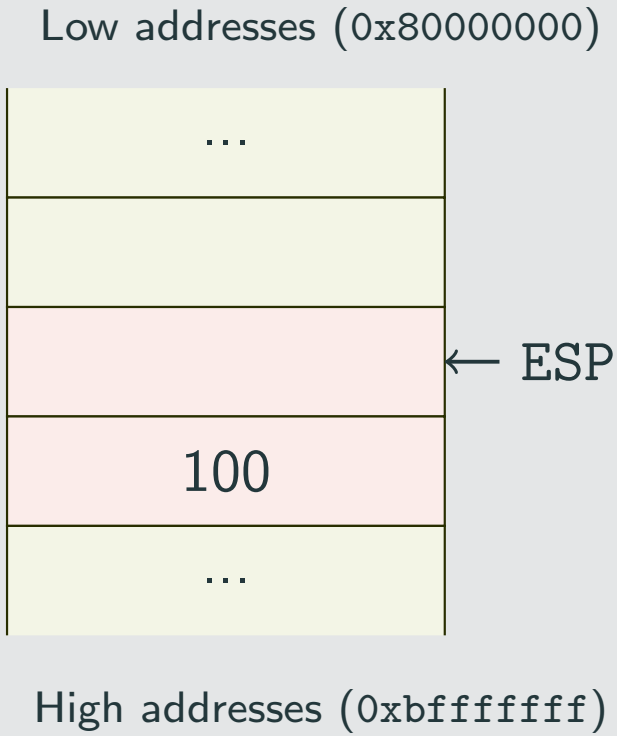
## Example

Initial condition: `EAX` = 30

```
push eax
```

is equivalent to:

```
sub esp, 4
mov DWORD PTR [esp], eax
```



# Stack Management Instructions: `push`

`push` **immediate** or **register**

Stores the immediate or register value at the top of the stack and decrements the `ESP` of the operand size

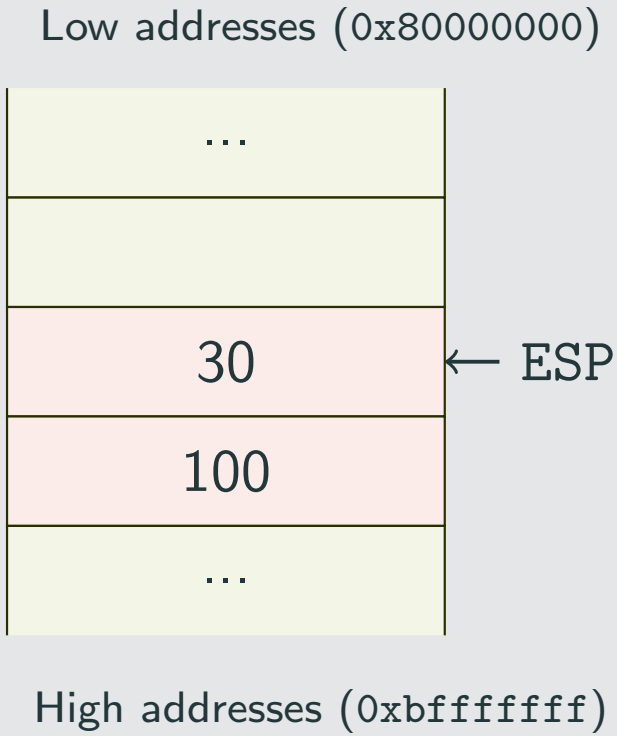
## Example

Final condition: `EAX` = 30

```
push eax
```

is equivalent to:

```
sub esp, 4
mov DWORD PTR [esp], eax
```



# Stack Management Instructions: `pop`

`pop` **destination**

Loads to the destination a word off the top of the stack, then it increases `ESP` of the operand's size.

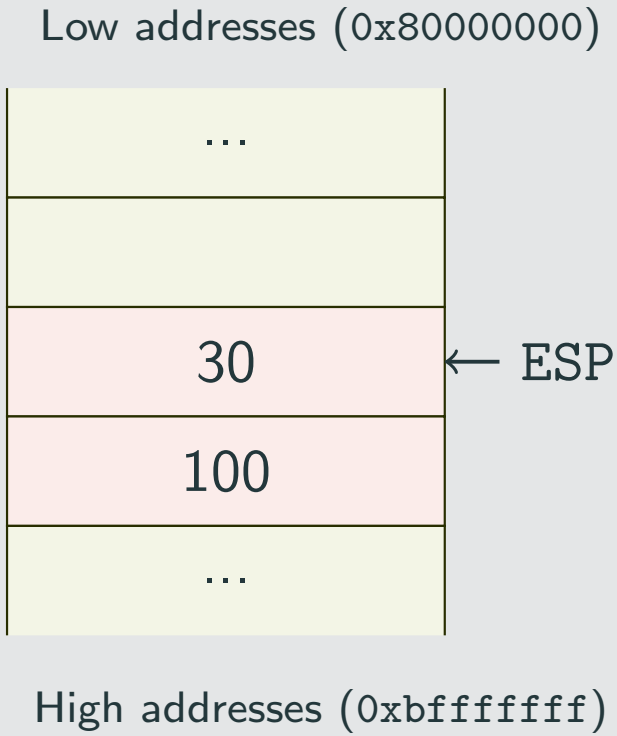
**Example**

Initial condition: `EAX` = ???

```
pop eax
```

is equivalent to:

```
mov eax, DWORD PTR [esp]
add esp, 4
```



# Stack Management Instructions: `pop`

`pop` **destination**

Loads to the destination a word off the top of the stack, then it increases `ESP` of the operand's size.

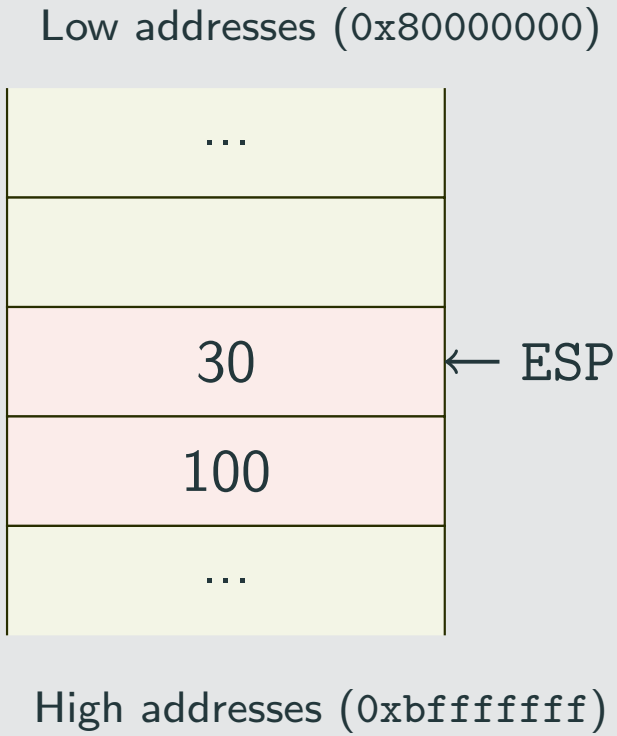
## Example

Initial condition: `EAX` = 30

```
pop eax
```

is equivalent to:

```
mov eax, DWORD PTR [esp]
add esp, 4
```



# Stack Management Instructions: `pop`

`pop` **destination**

Loads to the destination a word off the top of the stack, then it increases `ESP` of the operand's size.

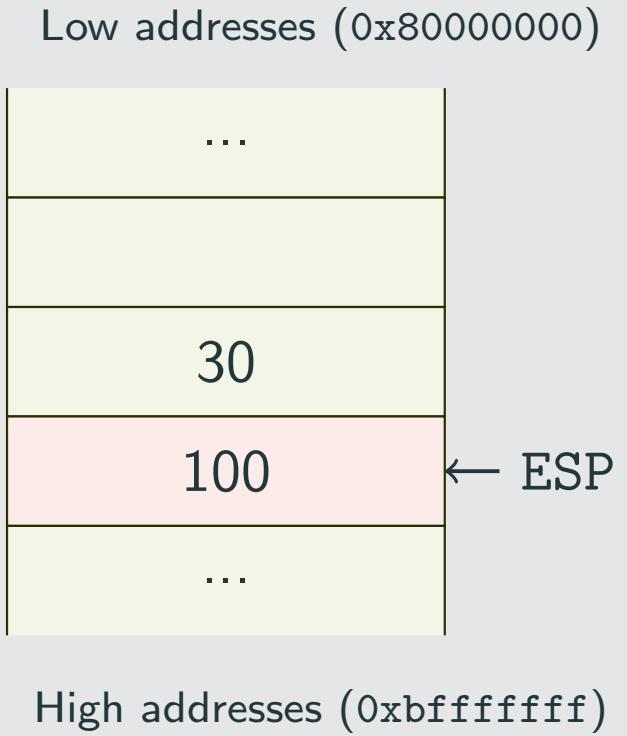
## Example

Final condition: `EAX` $= 30$

```
pop eax
```

is equivalent to:

```
mov eax, DWORD PTR [esp]
add esp, 4
```



# Calling a function

Instruction `call`:

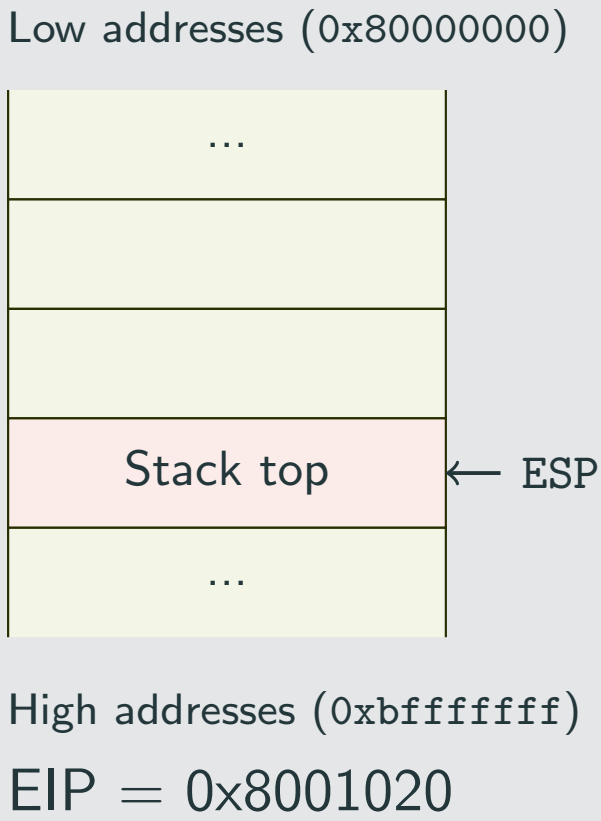
- Push to the stack the address of the next instruction
- Move the address of the first instruction of the callee into `EIP`

**Example: Let's call `func`, located at 0x800bff00**

Equivalent to:

- `push address(of the instruction after the call!)`
- `jmp func`

(reminder: we can't read or set `EIP` directly!)



# Calling a function

Instruction `call`:

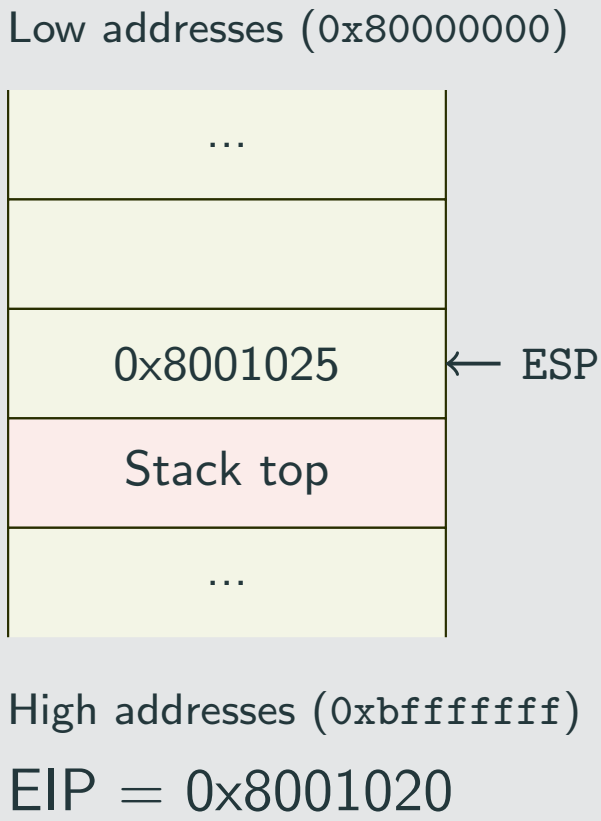
- Push to the stack the address of the next instruction
- Move the address of the first instruction of the callee into `EIP`

**Example: Let's call `func`, located at 0x800bff00**

Equivalent to:

- `push address(of the instruction after the call!)`
- `jmp func`

(reminder: we can't read or set `EIP` directly!)



# Calling a function

Instruction `call`:

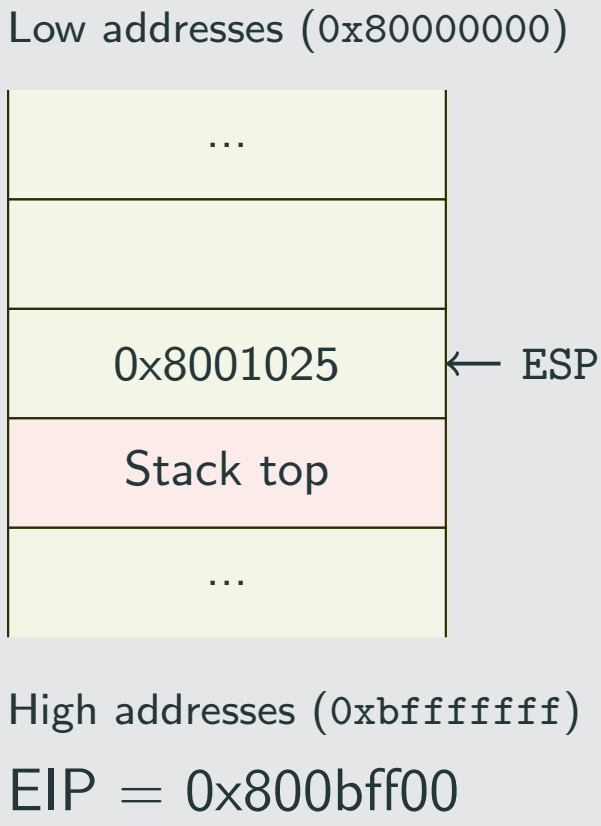
- Push to the stack the address of the next instruction
- Move the address of the first instruction of the callee into `EIP`

**Example: Let's call `func`, located at 0x800bff00**

Equivalent to:

- `push address(of the instruction after the call!)`
- `jmp func`

(reminder: we can't read or set `EIP` directly!)



# Returning from a function

Instruction `ret`:

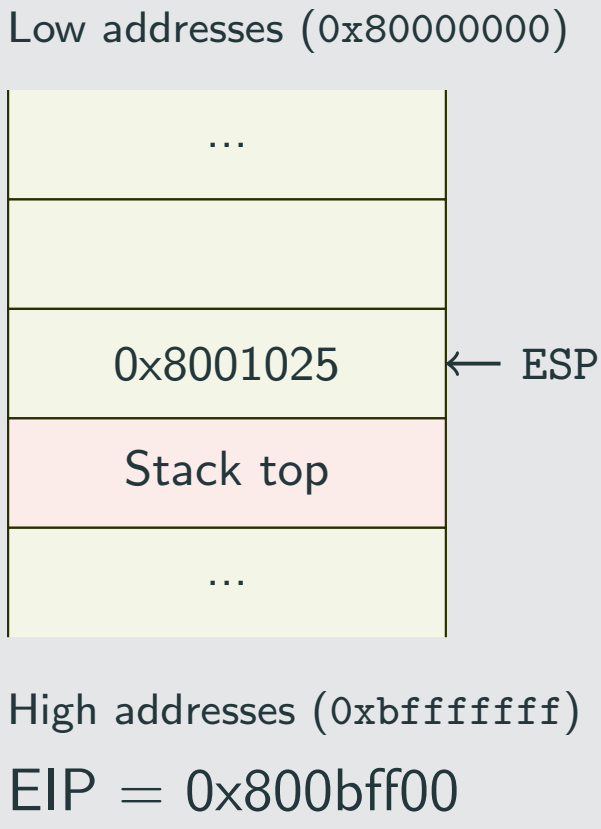
- Restores the return address saved by `call` from the top of the stack

**Example: let's return from `func`**

Equivalent to:

- `pop eip`

(reminde: we can't read or set `EIP` directly!)



# Returning from a function

Instruction `ret`:

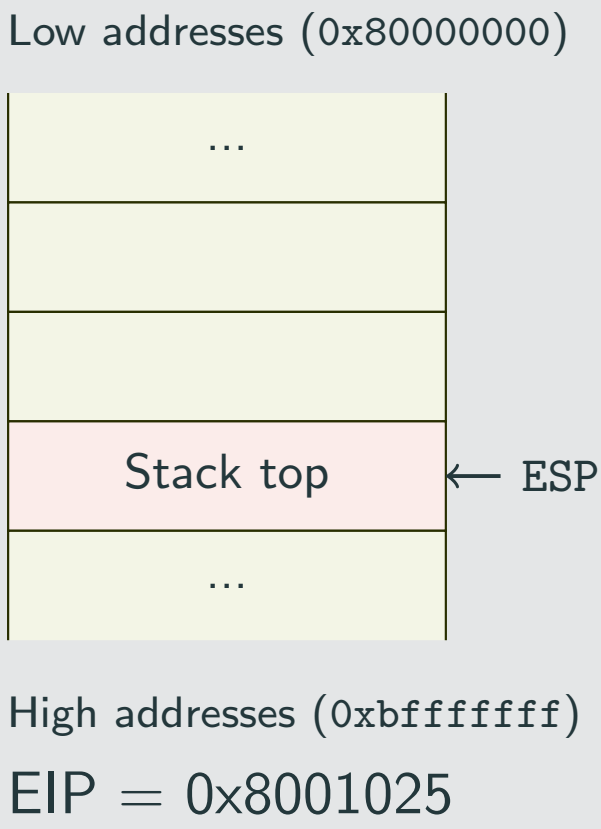
- Restores the return address saved by `call` from the top of the stack

**Example: let's return from `func`**

Equivalent to:

- `pop eip`

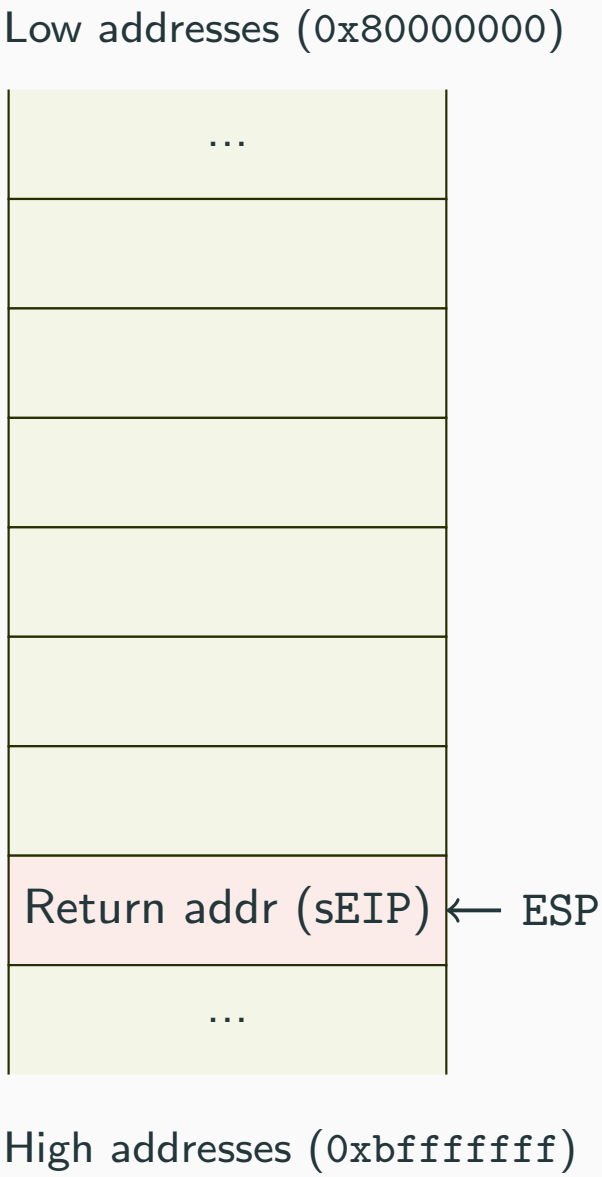
(reminde: we can't read or set `EIP` directly!)



# Functions and Stack Frames

At the beginning of a function, the function itself must reserve space for its local variables.

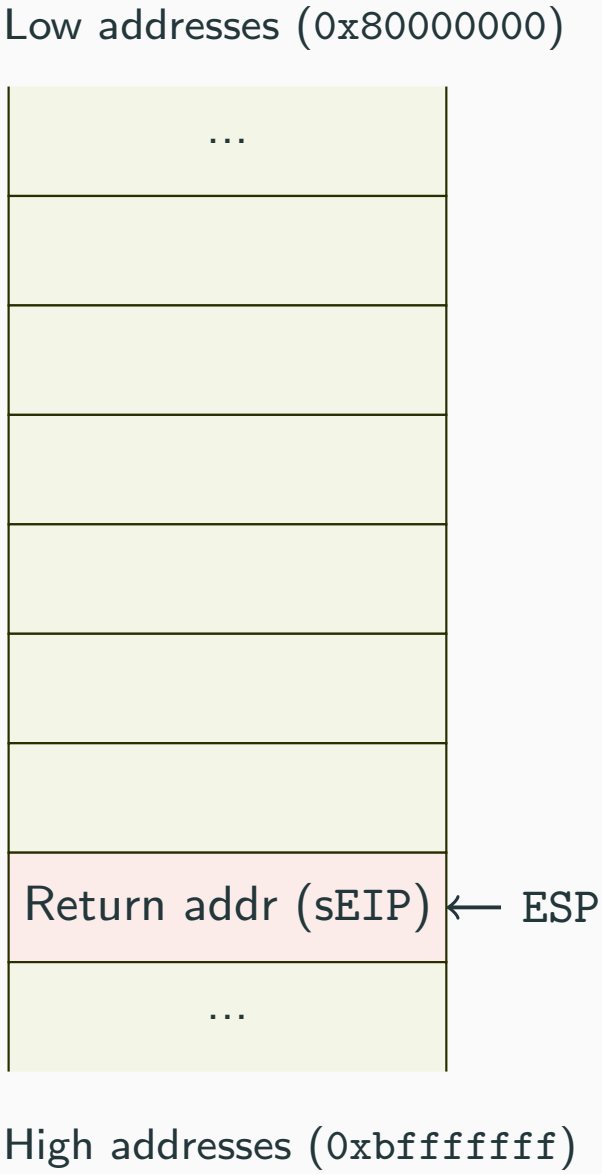
```
void foo() {
  int a;
  int b;
  int c;

  b = 0;
}
```



# Functions and Stack Frames

At the beginning of a function, the function itself must reserve space for its local variables.

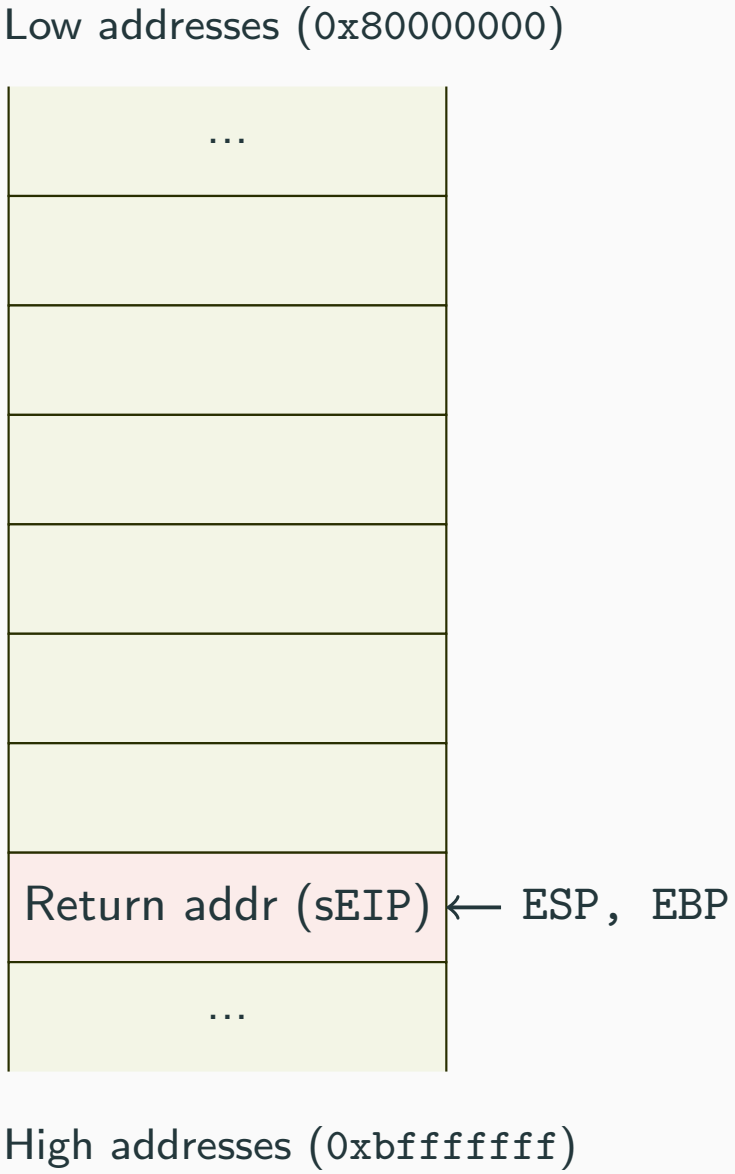
```
FOO:
    mov ebp, esp
    sub esp, 0xc
    mov [ebp - 0x8], 0x0
    add esp, 0xc
    ret
```



# Functions and Stack Frames

At the beginning of a function, the function itself must reserve space for its local variables.

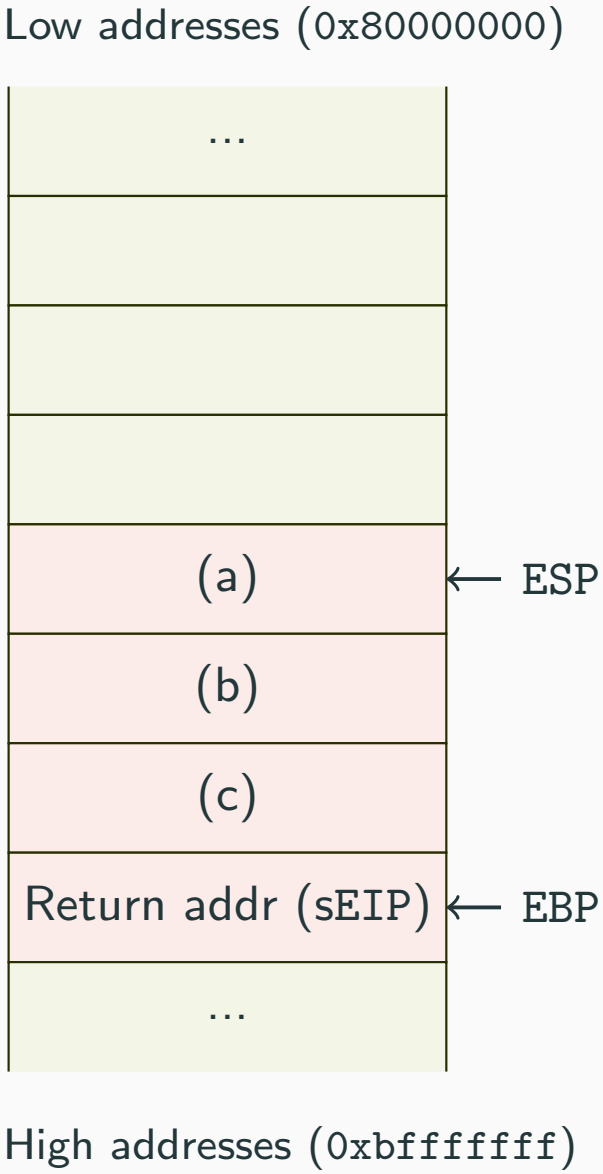
```
FOO:
    mov ebp, esp ←
    sub esp, 0xc
    mov [ebp - 0x8], 0x0
    add esp, 0xc
    ret
```



# Functions and Stack Frames

At the beginning of a function, the function itself must reserve space for its local variables.

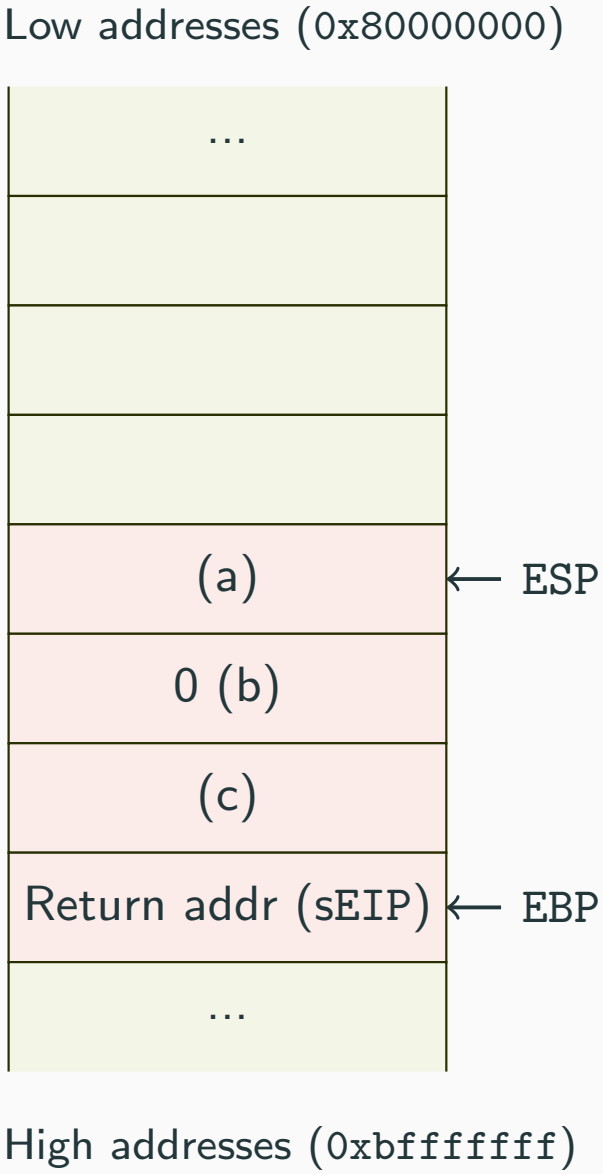
```
FOO:
    mov ebp, esp
    sub esp, 0xc ←
    mov [ebp - 0x8], 0x0
    add esp, 0xc
    ret
```



# Functions and Stack Frames

At the beginning of a function, the function itself must reserve space for its local variables.

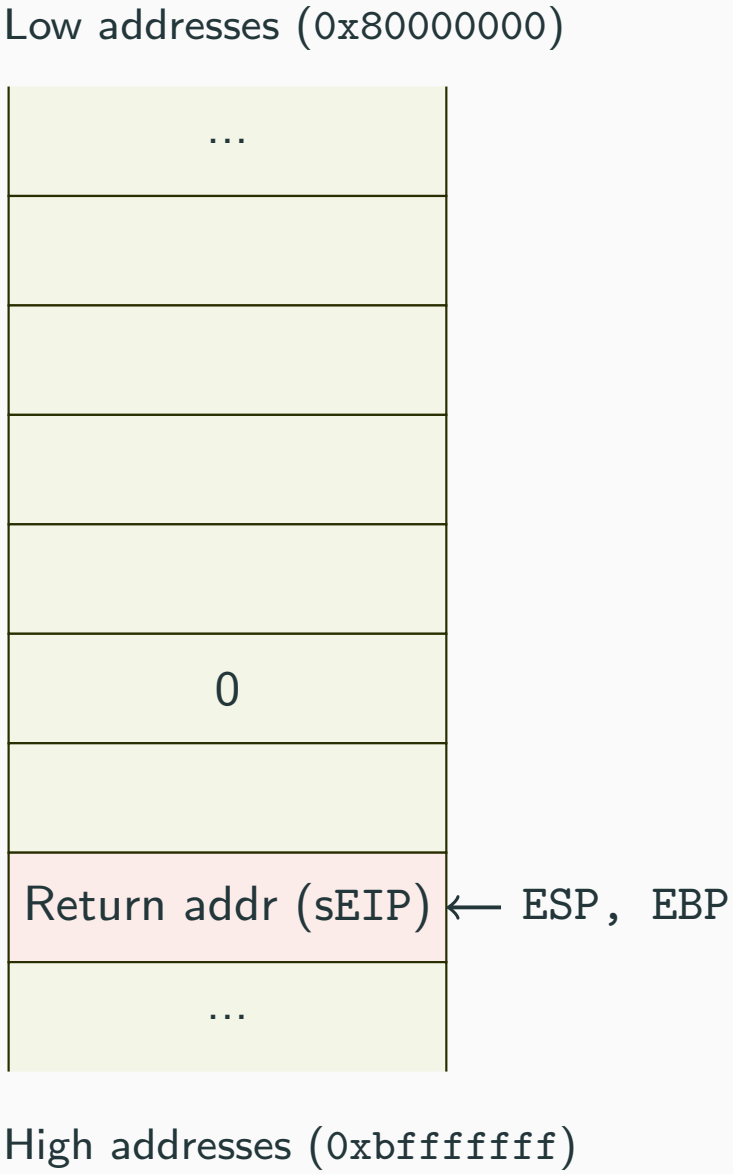
```
FOO:
    mov ebp, esp
    sub esp, 0xc
    mov [ebp - 0x8], 0x0 ←
    add esp, 0xc
    ret
```



# Functions and Stack Frames

At the beginning of a function, the function itself must reserve space for its local variables.

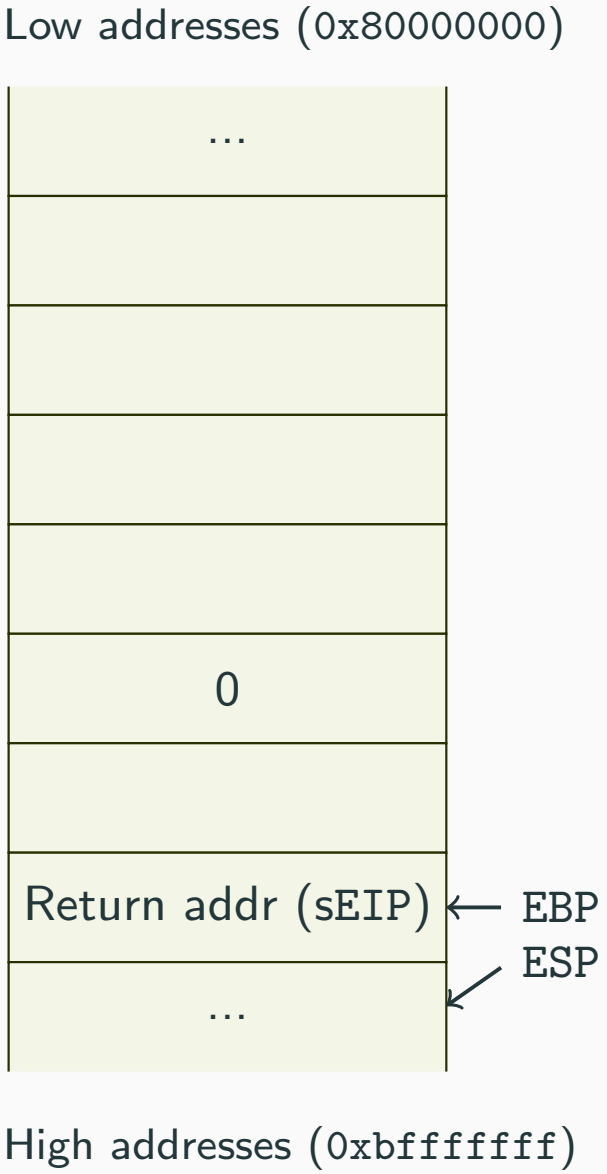
```
FOO:
    mov ebp, esp
    sub esp, 0xc
    mov [ebp - 0x8], 0x0
    add esp, 0xc ←
    ret
```



# Functions and Stack Frames

At the beginning of a function, the function itself must reserve space for its local variables.

```
FOO:
    mov ebp, esp
    sub esp, 0xc
    mov [ebp - 0x8], 0x0
    add esp, 0xc
    ret ←
```



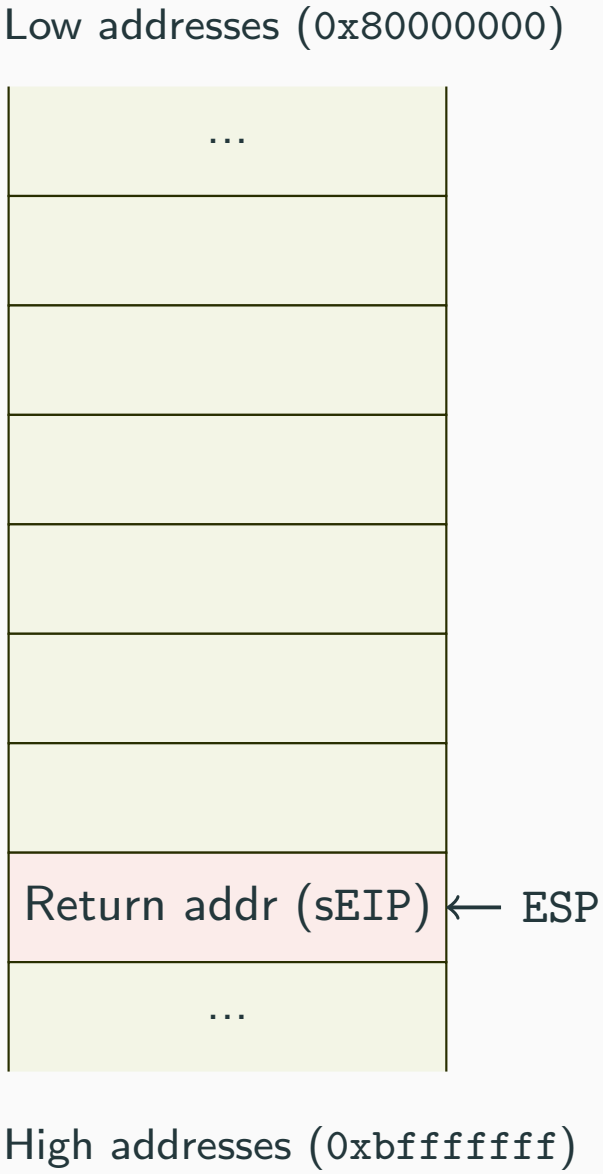
# Functions and Stack Frames

That works!!! But what if `foo` calls `bar`.

```
void foo() {
  int a;
  int b;
  int c;

  bar();
  b = 0;
}

void bar() {
  int d;

  d = 1;
}
```

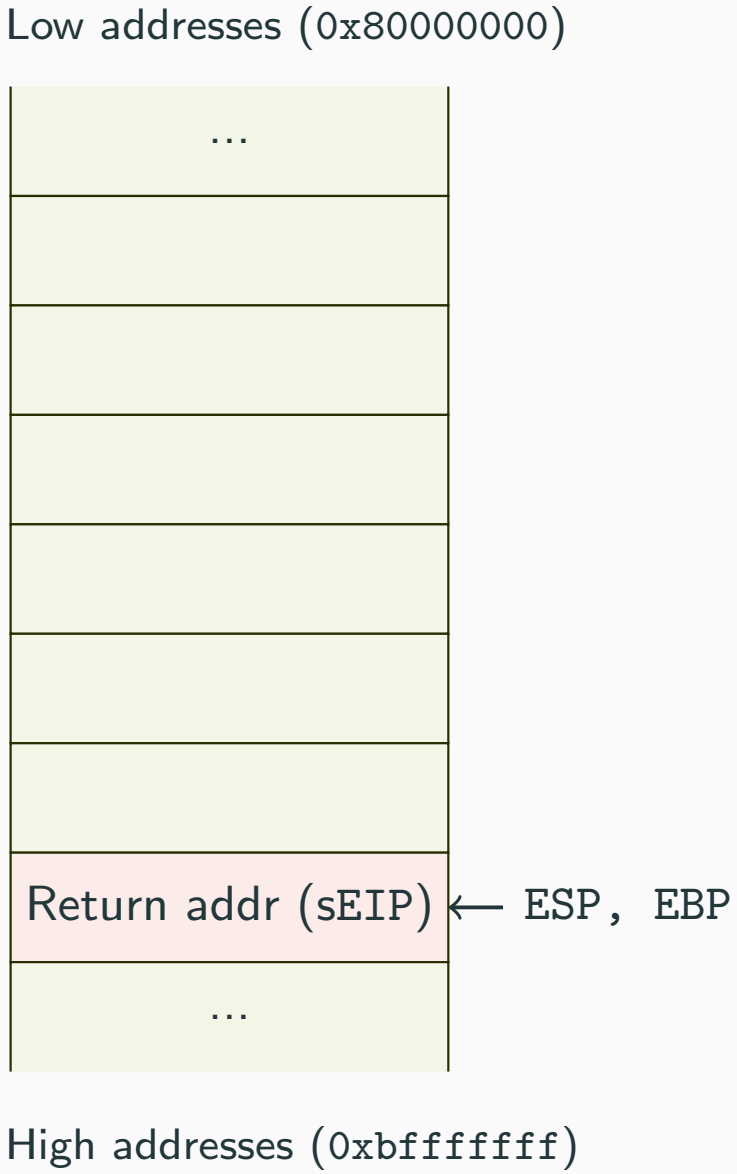
# Functions and Stack Frames

```
FOO:
    mov ebp, esp
    sub esp, 0xc
    call BAR
    mov [ebp - 0x8], 0x0
    add esp, 0xc
    ret

BAR:
    mov ebp, esp
    sub esp, 0x4
    mov [ebp - 0x4], 0x1
    add esp, 0x4
    ret
```



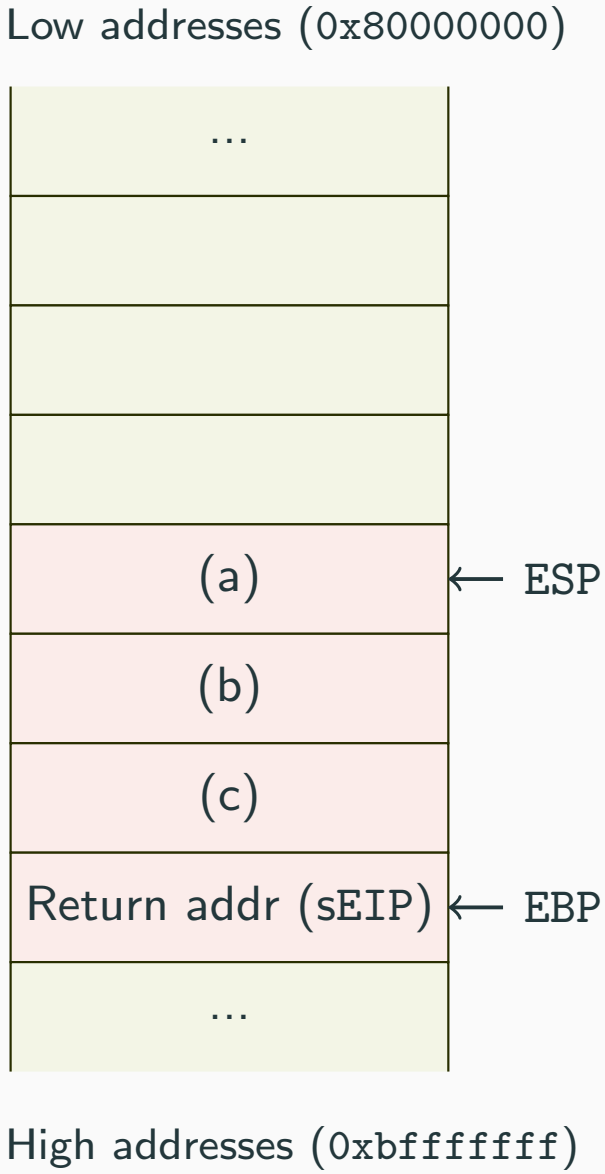
# Functions and Stack Frames

```
FOO:
    mov ebp, esp ←
    sub esp, 0xc
    call BAR
    mov [ebp - 0x8], 0x0
    add esp, 0xc
    ret

BAR:
    mov ebp, esp
    sub esp, 0x4
    mov [ebp - 0x4], 0x1
    add esp, 0x4
    ret
```



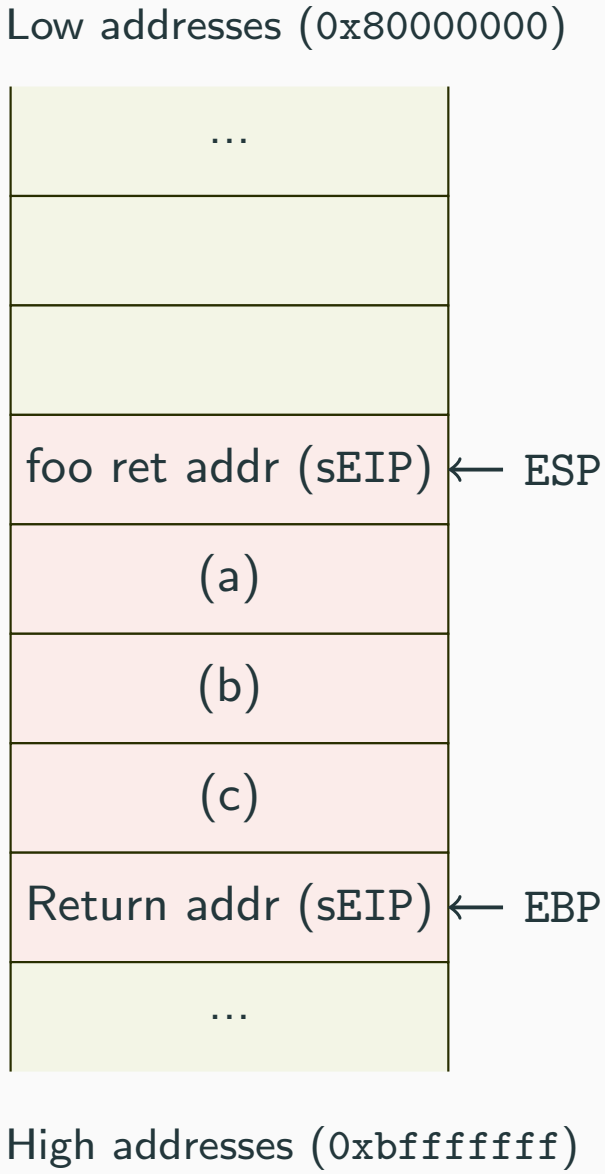
# Functions and Stack Frames

```
FOO:
    mov ebp, esp
    sub esp, 0xc ←
    call BAR
    mov [ebp - 0x8], 0x0
    add esp, 0xc
    ret

BAR:
    mov ebp, esp
    sub esp, 0x4
    mov [ebp - 0x4], 0x1
    add esp, 0x4
    ret
```



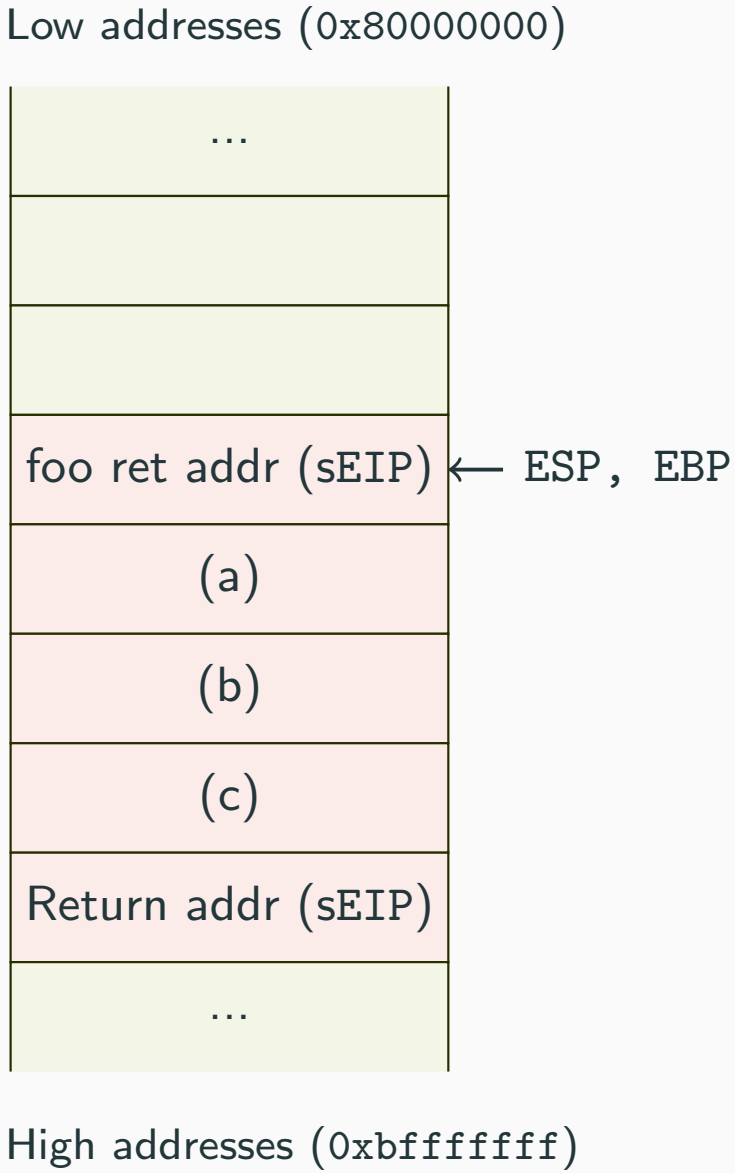
# Functions and Stack Frames

```
FOO:
    mov ebp, esp
    sub esp, 0xc
    call BAR ←
    mov [ebp - 0x8], 0x0
    add esp, 0xc
    ret

BAR:
    mov ebp, esp
    sub esp, 0x4
    mov [ebp - 0x4], 0x1
    add esp, 0x4
    ret
```



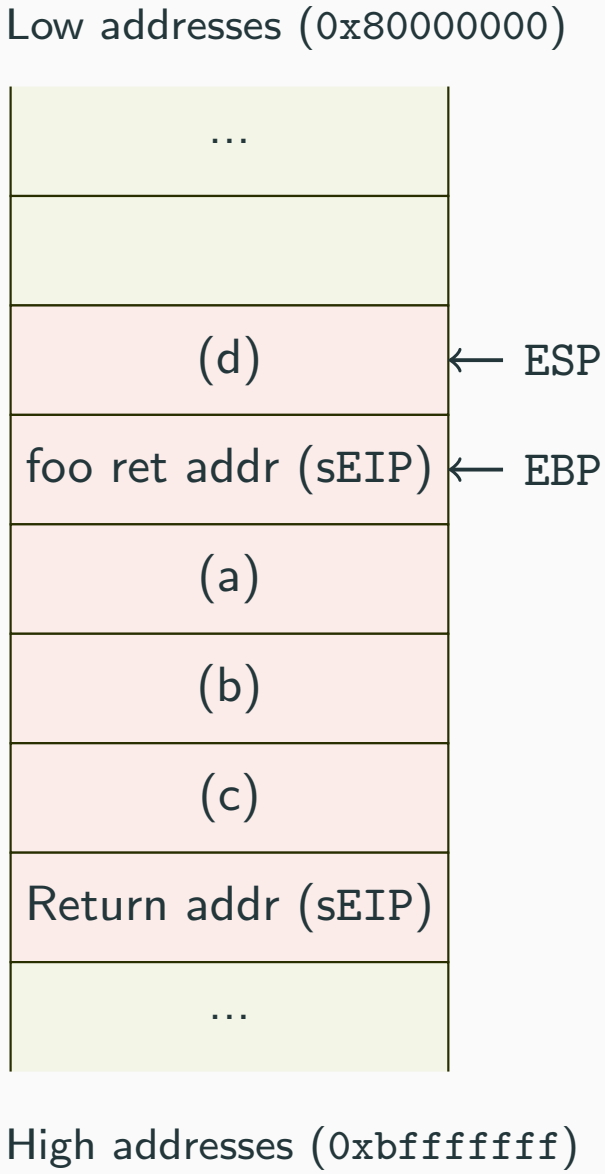
# Functions and Stack Frames

```
FOO:
    mov ebp, esp
    sub esp, 0xc
    call BAR
    mov [ebp - 0x8], 0x0
    add esp, 0xc
    ret

BAR:
    mov ebp, esp ←
    sub esp, 0x4
    mov [ebp - 0x4], 0x1
    add esp, 0x4
    ret
```



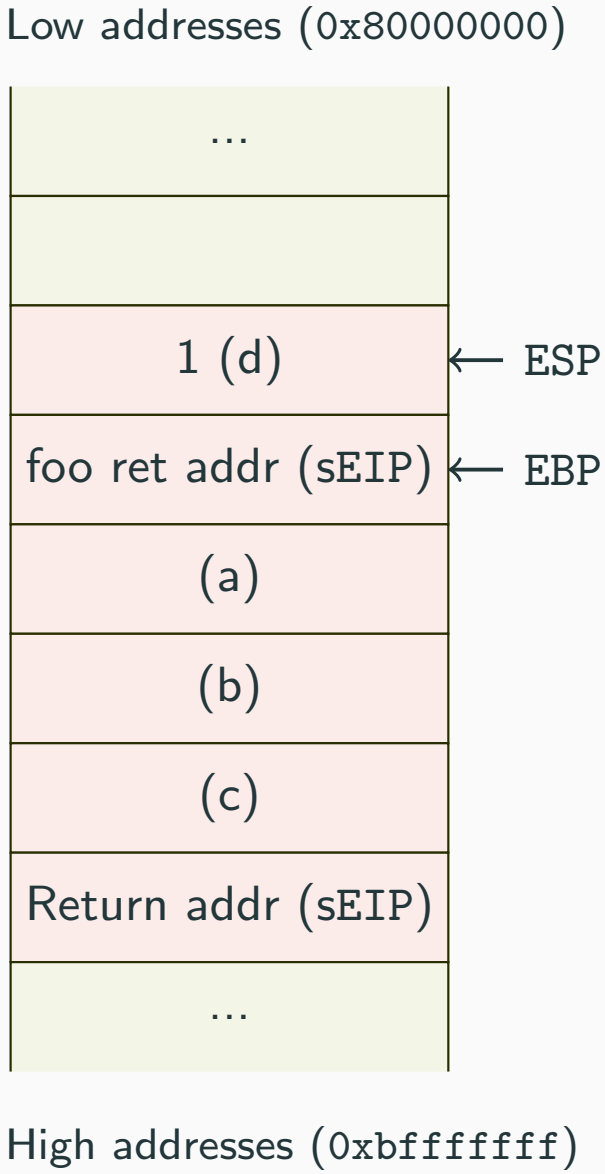
# Functions and Stack Frames

```
FOO:
    mov ebp, esp
    sub esp, 0xc
    call BAR
    mov [ebp - 0x8], 0x0
    add esp, 0xc
    ret

BAR:
    mov ebp, esp
    sub esp, 0x4 ←
    mov [ebp - 0x4], 0x1
    add esp, 0x4
    ret
```



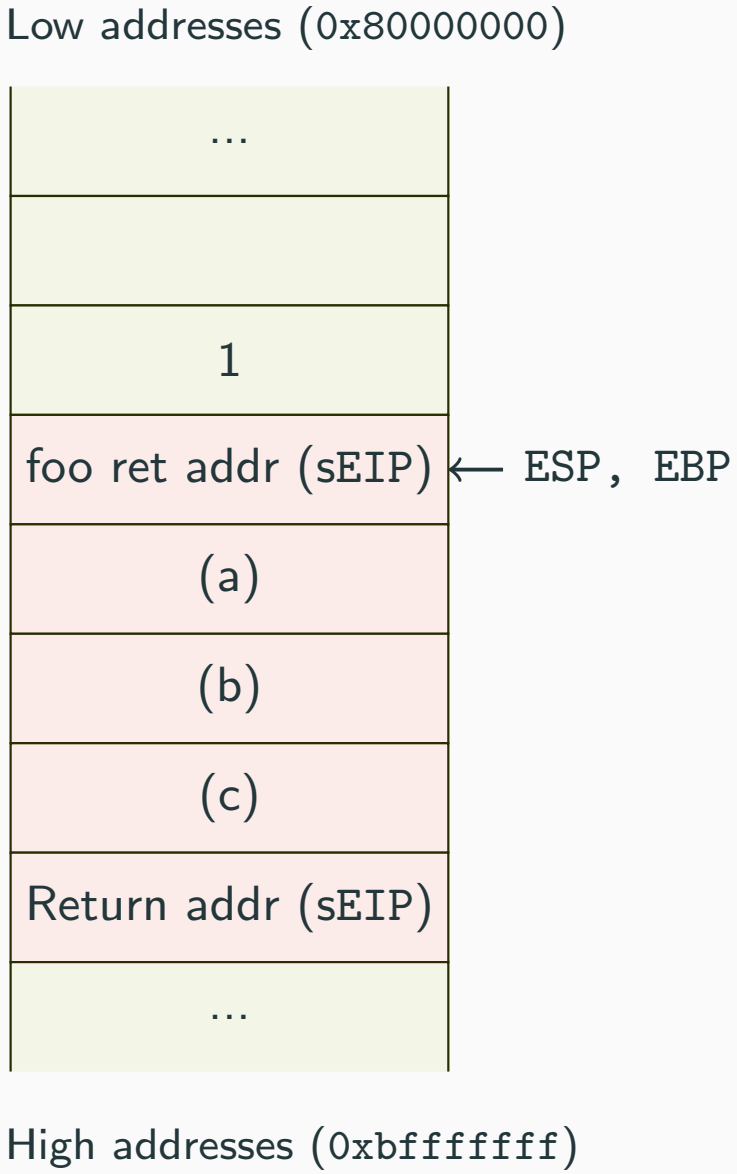
# Functions and Stack Frames

```
FOO:
    mov ebp, esp
    sub esp, 0xc
    call BAR
    mov [ebp - 0x8], 0x0
    add esp, 0xc
    ret

BAR:
    mov ebp, esp
    sub esp, 0x4
    mov [ebp - 0x4], 0x1 ←
    add esp, 0x4
    ret
```



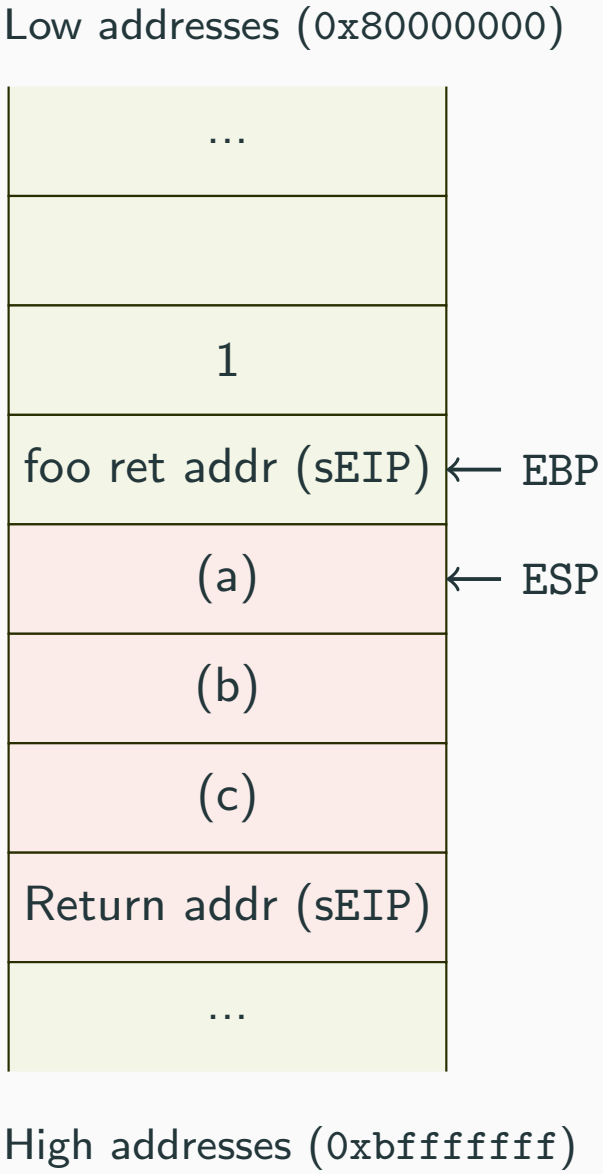
# Functions and Stack Frames

```
FOO:
    mov ebp, esp
    sub esp, 0xc
    call BAR
    mov [ebp - 0x8], 0x0
    add esp, 0xc
    ret

BAR:
    mov ebp, esp
    sub esp, 0x4
    mov [ebp - 0x4], 0x1
    add esp, 0x4 ←
    ret
```



# Functions and Stack Frames

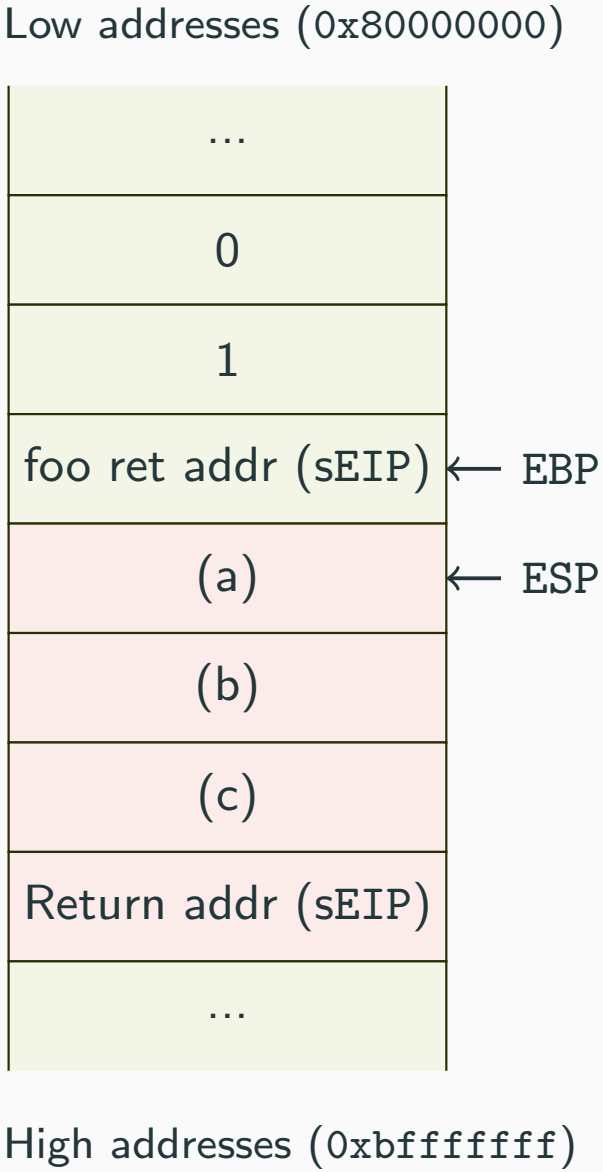
```
FOO:
    mov ebp, esp
    sub esp, 0xc
    call BAR
    mov [ebp - 0x8], 0x0
    add esp, 0xc
    ret

BAR:
    mov ebp, esp
    sub esp, 0x4
    mov [ebp - 0x4], 0x1
    add esp, 0x4
    ret ←
```



# Functions and Stack Frames

Wrong memory access!!! EBP changed and we lost the old value!

```
FOO:
    mov ebp, esp
    sub esp, 0xc
    call BAR
    mov [ebp - 0x8], 0x0 ←
    add esp, 0xc
    ret

BAR:
    mov ebp, esp
    sub esp, 0x4
    mov [ebp - 0x4], 0x1
    add esp, 0x4
    ret
```



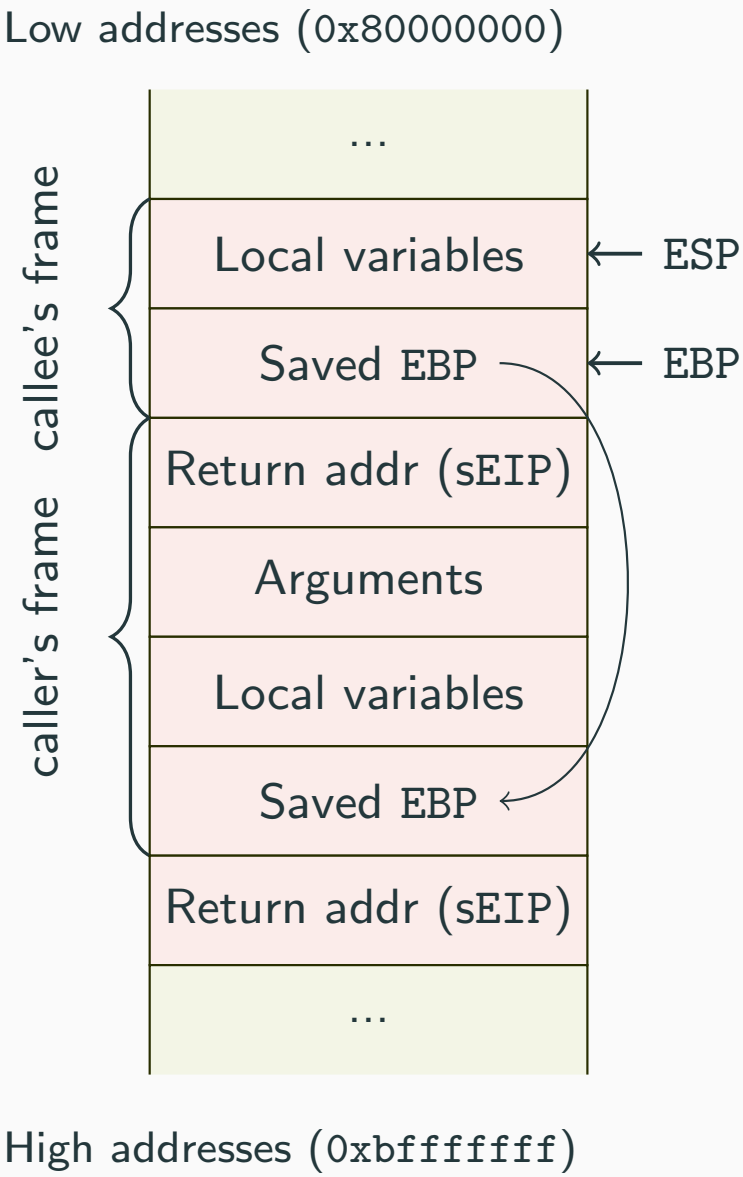
# Functions and Stack Frames

Redacted!

```
FOO:
    push ebp
    mov ebp, esp
    sub esp, 0xc
    call BAR
    mov [ebp - 0x8], 0x0
    leave
    ret

BAR:
    push ebp
    mov ebp, esp
    sub esp, 0x4
    mov [ebp - 0x4], 0x1
    leave
    ret
```

# Functions and Stack Frames

- Stack frame = stack area allocated to a function
- `EBP` register: pointer to the beginning (base) of a function's frame
- At the beginning of a function:
  - Save EBP to stack
  - Set `EBP` to the address of the function's frame

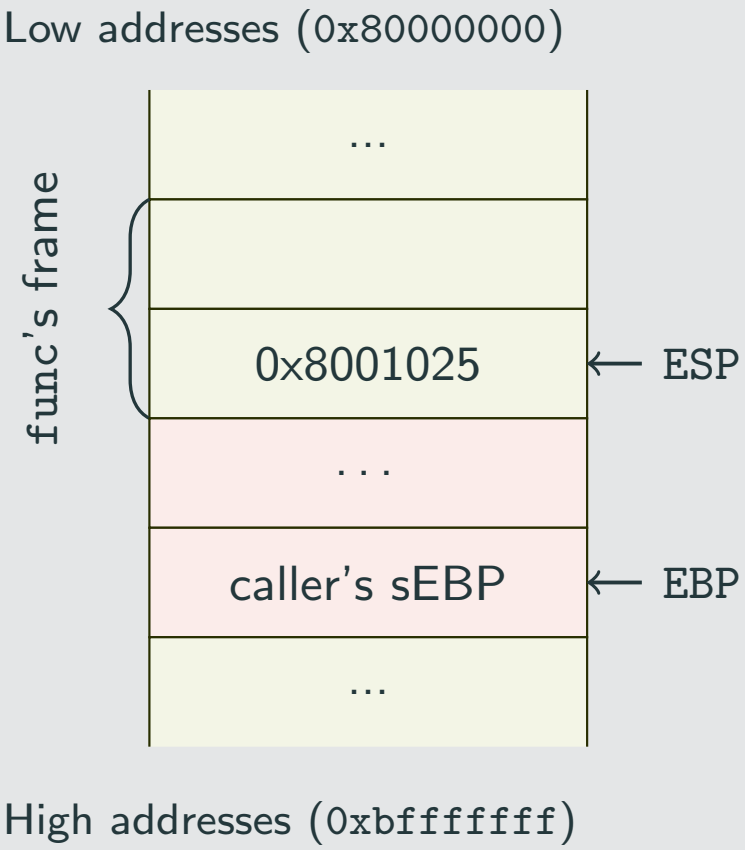


# Entering a function

**Example: We've just called `func`, located at 0x800bff00**

Setup the stack frame

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

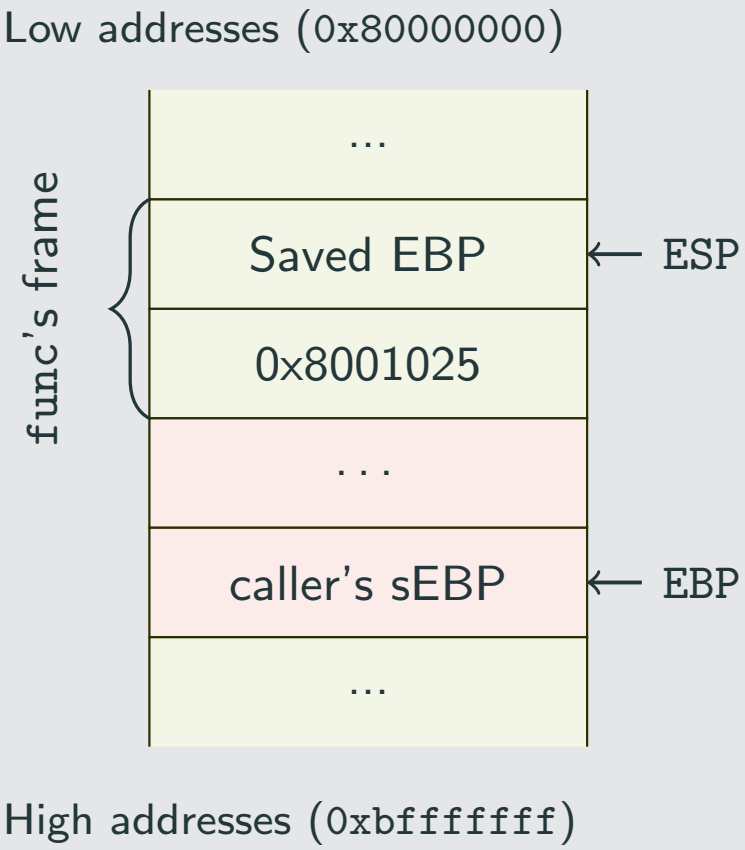


# Entering a function

**Example: We've just called `func`, located at 0x800bff00**

Setup the stack frame

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

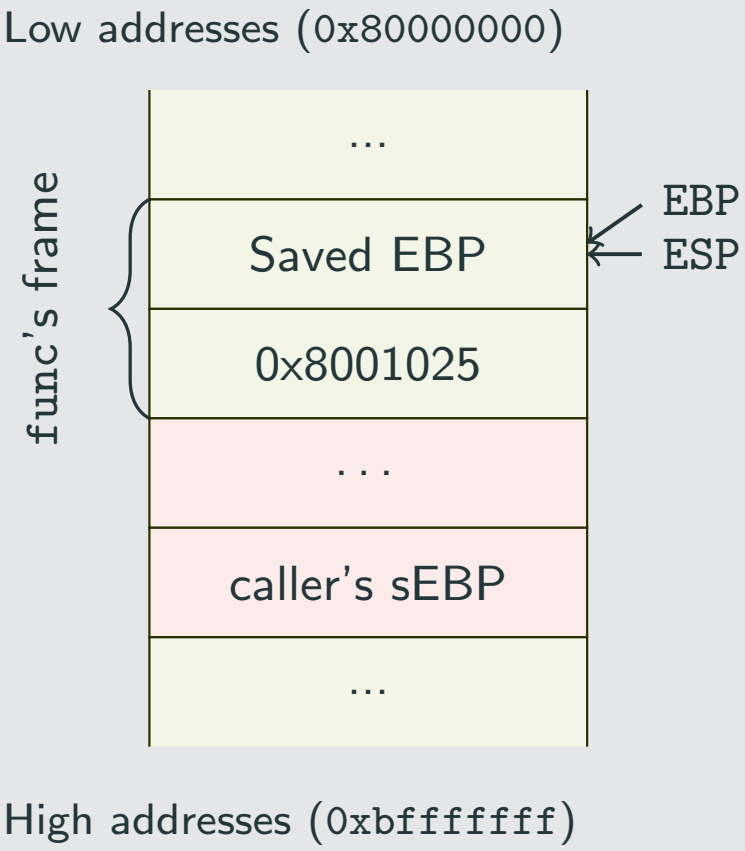


# Entering a function

**Example: We've just called `func`, located at 0x800bff00**

Setup the stack frame

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



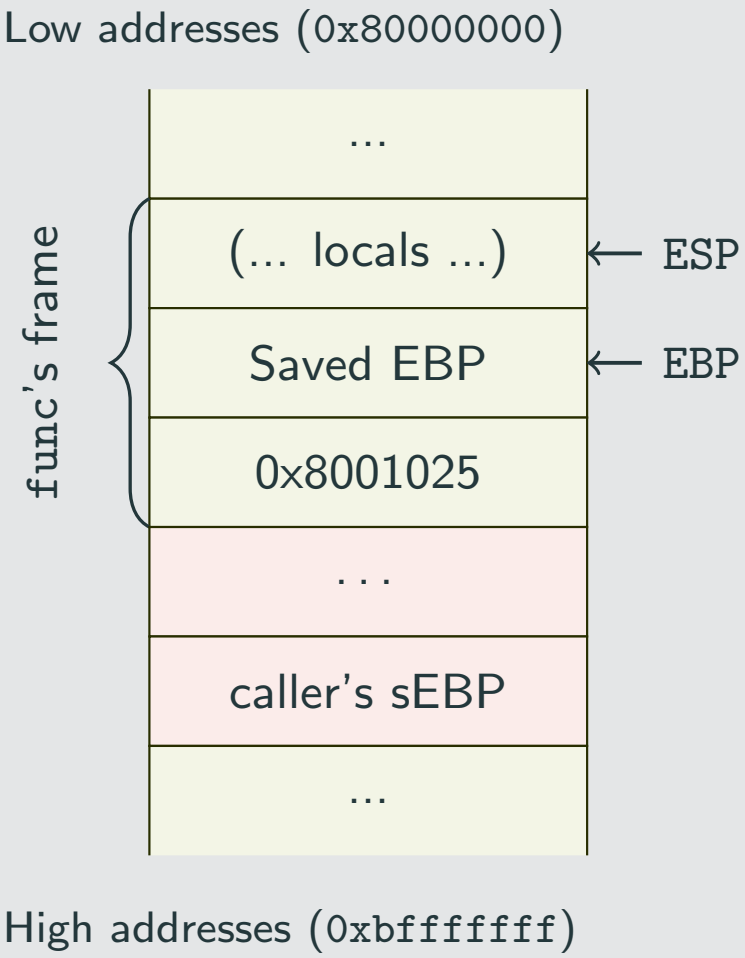
# Leaving a function

Instruction `leave`: restores the caller's base pointer

**Example: We're about to return from `func`**

Equivalent to:

- `mov esp, ebp`
- `pop ebp`



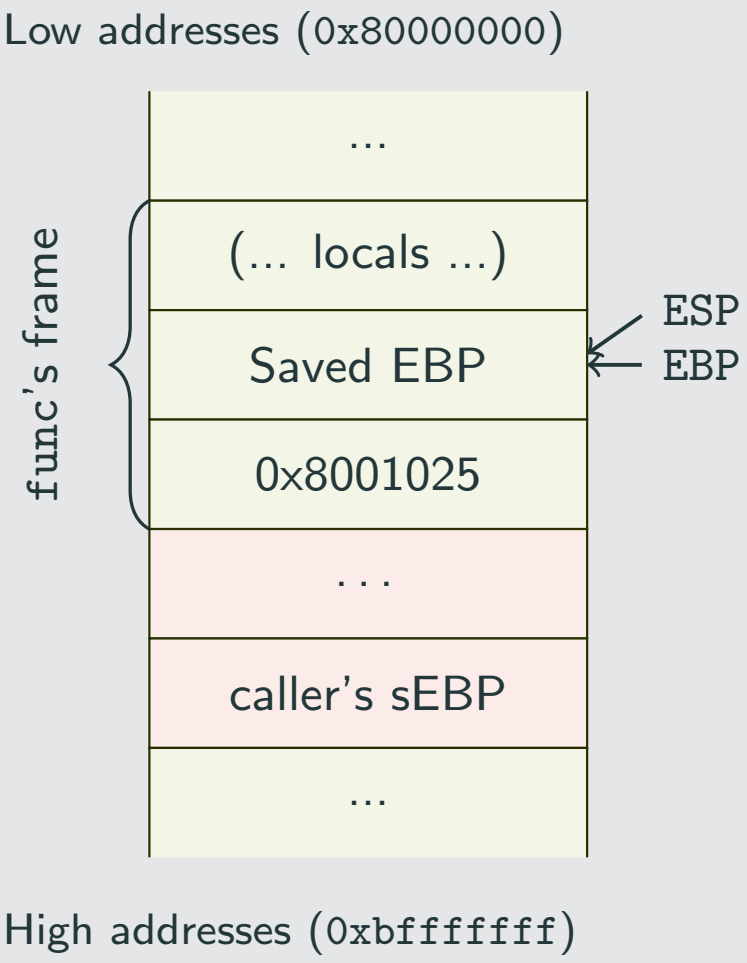
# Leaving a function

Instruction `leave`: restores the caller's base pointer

**Example: We're about to return from `func`**

Equivalent to:

- `mov esp, ebp`
- `pop ebp`



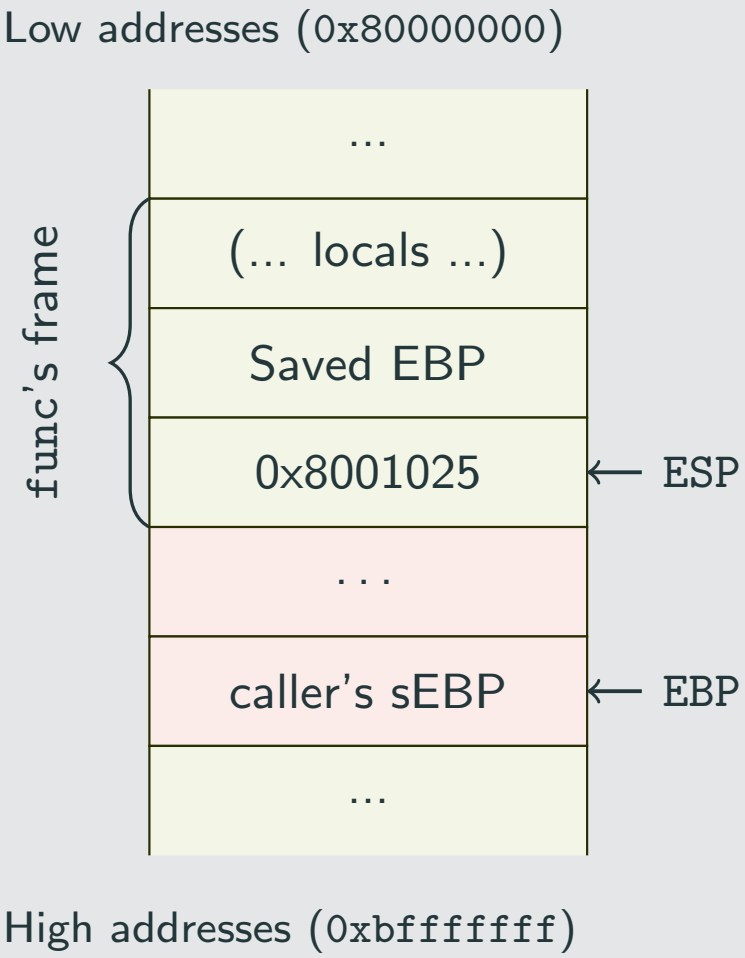
# Leaving a function

Instruction `leave`: restores the caller's base pointer

**Example: We're about to return from `func`**

Equivalent to:

- `mov esp, ebp`
- `pop ebp`



# Calling Conventions

- Defines
  - how to pass parameters (stack, registers or both, and who is responsible to clean them up)
  - how to return values
  - caller-saved or callee-saved registers
- The high-level language, the compiler, the OS, and the target architecture all together "implement" and "agree upon" a certain calling convention
  - it's part of the ABI, the Application Binary Interface

# Calling Conventions: `cdecl` (C declaration)

- Default calling convention used by most x86 C compilers
  - Can be forced with the modifier `_cdecl`
- Arguments: passed **through the stack**, right to left order
- Cleanup: the **caller removes** the parameters from the stack *after* the called function completes
- Return: register `EAX`
- Caller-saved registers: `EAX`, `ECX`, `EDX` (other are callee-saved)

# cdecl: Example

```
void demo_cdecl(int a, int b, int c, int z);

//...

demo_cdecl(1, 2, 3, 4); //calling
```

```
; ...
push 4 ; push last parameter value
push 3 ; push third parameter value
push 2 ; ...
push 1
call demo_cdecl ; call the subroutine
add esp, 16 ; clean up the stack
```

# Calling Conventions: `fastcall` (C declaration)

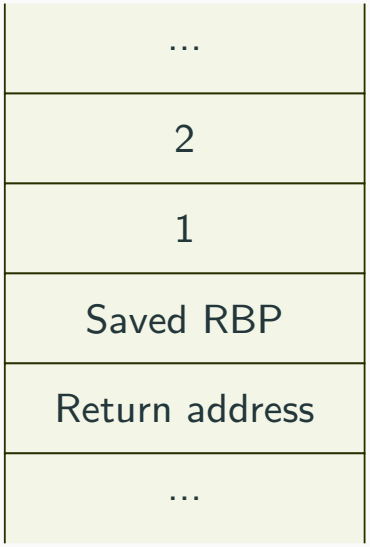
- Default calling convention used by most x86_64 C compilers
  - Can be forced with the modifier `_fastcall`
- Parameters passed **in registers**: `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`, subsequent ones on the stack (reverse order, caller cleanup)
- Callee-saved registers: `rbx`, `rsp`, `rbp`, `r12`, `r13`, `r14`, and `r15`
- Caller-saved registers (scratch): `rax`, `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`, `r10`, `r11`
- Return value: `rax` (if 128-bit: `rax` and `rdx`)

# Linux x86-64 calling convention: Example

```
main:
  push rbp
  mov rbp, rsp
  sub rsp, 16
  mov DWORD PTR [rbp-4], edi
  mov QWORD PTR [rbp-16], rsi
  mov esi, 2 ; Second parameter
  mov edi, 1 ; First parameter
  call function
  mov esi, eax ; Return value -> first param
  mov edi, OFFSET FLAT:.LC0 ; "The return ...
  mov eax, 0
  call printf
  leave
  ret

function:
  push rbp
  mov rbp, rsp
  mov DWORD PTR [rbp-4], edi
  mov DWORD PTR [rbp-8], esi
  mov edx, DWORD PTR [rbp-4]
  mov eax, DWORD PTR [rbp-8]
  add eax, edx
  pop rbp
  ret
```

```
int function (int a, int b) {
    return a + b;
}

int main (int argc, char** argv) {
    return printf("The return value is %d\n",
           function(1,2));
}
```

Low addresses




High addresses

# Tooling

# Shell for Dummies [3]

| | | |
|---|---|---|
| important paths | / | #root (first) directory |
| | ~/ | #your home directory |
| | . | #current directory |
| | .. | #parent directory |
| filesystem utils | pwd | #show current directory |
| | cd *path* | #change directory to *path* |
| | ls *path* | #list files in the directory at *path* |
| | cp *path_src path_dst* | #copy *path_src* to *path_dst, -r if copying directories* |
| | mv *path_src path_dst* | *#move path_src* to *path_dst* |
| basic text editor | nano *path/file* | #opens/create *file* in *path* (ctrl+x to exit) |
| | vim *path/file* | #opens/create *file* in *path* (i to edit; esc, :wq to save and quit) |
| remote actions | ssh *user@server_addr* | #ssh to *server_addr* as *user* |
| | scp [*u@s:*]*p_src* [*u@s:*]*p_dst* | #cp to/from remote server |

[3] `cmd --help` or `cmd -h` to get the aviable options

# Shell for Dummies [4]

| | | |
|---|---|---|
| file visualization | cat *file*<br>less *file*<br>hexdump -Cv *file* | #print *file* to stdout<br>#visualize *file* better, q to quit<br>#visualize raw bytes |
| redirections | command > file<br>command >> file<br><br>command < file<br>cmd1 \| cmd2 | #write stdout of command to *file*<br>#append stdout of command to *file*<br>#use *file* as stdin of command<br>#stdout of cmd1 as stdin cmd2 |
| argv from command | cmd `cmd2`<br><br>cmd $(cmd2) | #executes cmd2 first and uses the output to eval the next command |
| other useful commands | chmod +x *file*<br>grep *expression*<br>python -c 'cmd1;cmd2;...' | #give exec permission to *file*<br>#search for *expr* in stdin<br>#executes python commands |

[4] `cmd --help` or `cmd -h` to get the aviable options

# objdump

- `man objdump`
  objdump **displays information** about one or more **object files**.
- `-x` all-headers
- `-d` disassemble
- `-M intel` intel syntax (default is AT&T)

# Debugging: GDB

- **What is GDB?**
  GDB is GNU Project's Debugger: allows to follow, step by step, at assembler-level granularity, a running program, or what a program was doing right before it crashed.[5]

[5] http://www.gnu.org/software/gdb/

# Start, break and navigate the execution with gdb

- Suppose you have an executable binary and want run it
  - **gdb /path/to/executable** loads the binary in gdb
- Now you decide to start the program with two parameters
  - **run 1 "abc"** passes 1 via `argv[1]` and `"abc"` as `argv[2]`
  - **run ‘printf "AAAAAAAAAAAA"‘** (with the back ticks) we're passing the output of the print (very useful when you need to pass non printable characters such as raw bytes)
- Suppose you want to stop the execution at the address of a certain instruction
  - **break *0xDEADBEAF** places a break point at that address
  - **break *main+1** with debugging symbols this can be less painful
  - **catch syscall** block the execution when a syscall happens

# Start, break and navigate the execution with gdb

- Now the execution stops at our break point. Here we can do several things
- Examples:
  - **ni** allows to procede instruction per instruction
  - **next 4** moves 4 lines ahead (if you have the line-numbers information in the binary)
  - **si** step into function
  - **finish** run until the end of current function
  - **continue** runs until the next break point (if any)
- To see info about the execution state:
  - **info registers** to inspect the content of the registers
  - **info frame** to see the values of the stack frame related to the function where we are in
  - **info file** print the information about the sections of the binary

# Navigate the stack

- Suppose we're stopped somewhere in the code and want to inspect the stack
- Some useful view of the stack is achievable with:
  - **x/100wx $esp** prints 100 words of memory from the address found in the ESP to ESP+100 (x = hexadecimal formatting)
  - **x/10wo $ebp-100** prints 10 words of memory from EBP-100 to EBP-100+10 (o = octal formatting)
  - **x/s $eax** prints the elements pointed by EAX (s = string formatting)
- Do you have debug symbols? (i.e., `gcc -ggdb`)
  - **print args** prints info about the main parameters
  - **print a** prints the content of variable 'a'
  - **print *b** prints the value pointed by 'b'

# Our friend gdb

- **The '~/.gdbinit' file**
  Gdb is a command line tool and it supports the configuration script as almost all the *nix software.
  Some options that you may want to tune are:
  - **set history save on**
    To have the lastest commands always available also when we re-open gdb
  - **set follow-fork-mode child**
    Allows you, if the process spawns children, to follow them and not only wait their end.
  - **set disassembly-flavor [intel | att]**
    This option sets in which predefined syntax your disassembled will be showed up. The default one is at&t
- Highly recommended to install pwndbg
  `https://github.com/pwndbg/pwndbg`

# GDB - How to Survive [6]

| | | |
|---|---|---|
| start | gdb -q *program* | #starts gdb silently for program |
| disassemble | set disassembly-flavor intel<br>disass **address* (or *f-name*) | #sets intel syntax<br>#disassemble from given *address* |
| run program | run (r)<br>start<br>run *arg1*<br>run <<< *arg1* | #runs the program<br>#runs the program and imm. stops<br>#runs program with *arg1* in argv<br>#runs program with *arg1* in stdin |
| memory layout | vmmap | #show memory layout |

[6] CTRL + C to Break and Debug

# GDB - How to Survive [7]

| | | |
|---|---|---|
| **execution** | stepi (si)<br>nexti (ni)<br>finish (f)<br>continue (c) | #exec next inst - enters a function<br>#exec next inst - skips the function<br>#exec till next return statement<br>#continue exec till next break/ watch |
| **breakpoinits** | b **address*<br>b **address* if $*reg*==*val*<br>del *br_num* | #set software breakp at *address*<br>#set conditional breakp<br>#remove breakpoint *br_num* |
| **watchpoints** | w **address*<br>rw **address* | #set watch for write at *address*<br>#set watch for read at *address* |
| **examine** | x/*numF* **address*<br>search *string*<br>p *symbol* | #show *num* data of type *F*<br>(useful *F*s are bx, wx, gx, c, s, i)<br>#search for string in memory<br>#print address of symbol |

[7] CTRL + D to Exit

# strace

- Intercepts and records system calls and signals
- Dumps to standard error name, argument and return value of each system call

**Useful options**

- `-p <pid>` attach to existing process
- `-f` trace child process
- `-o <filename>` output to file
- `-e <expr>` modifies which events to trace (see manpage)

# ltrace

- Intercepts and records dynamic library calls
- Similar to strace, but at a different layer

# Questions?