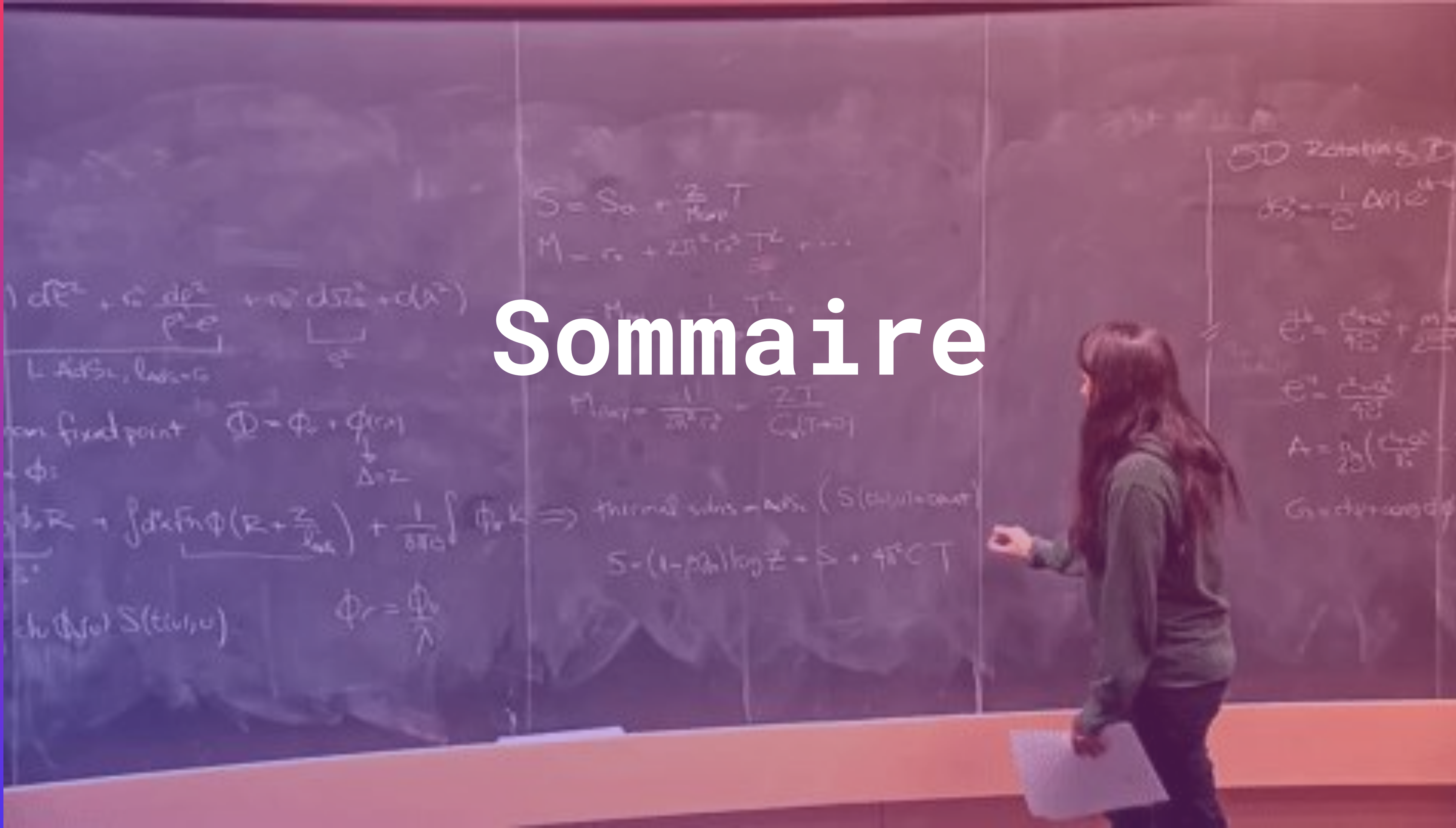
X86 ASM Introduction To Reverse Engineering

Simon RACAUD
With the the amazing work of Ophir Harpaz - www.begin.re and David Evans www.cs.virginia.edu

29/03/2021

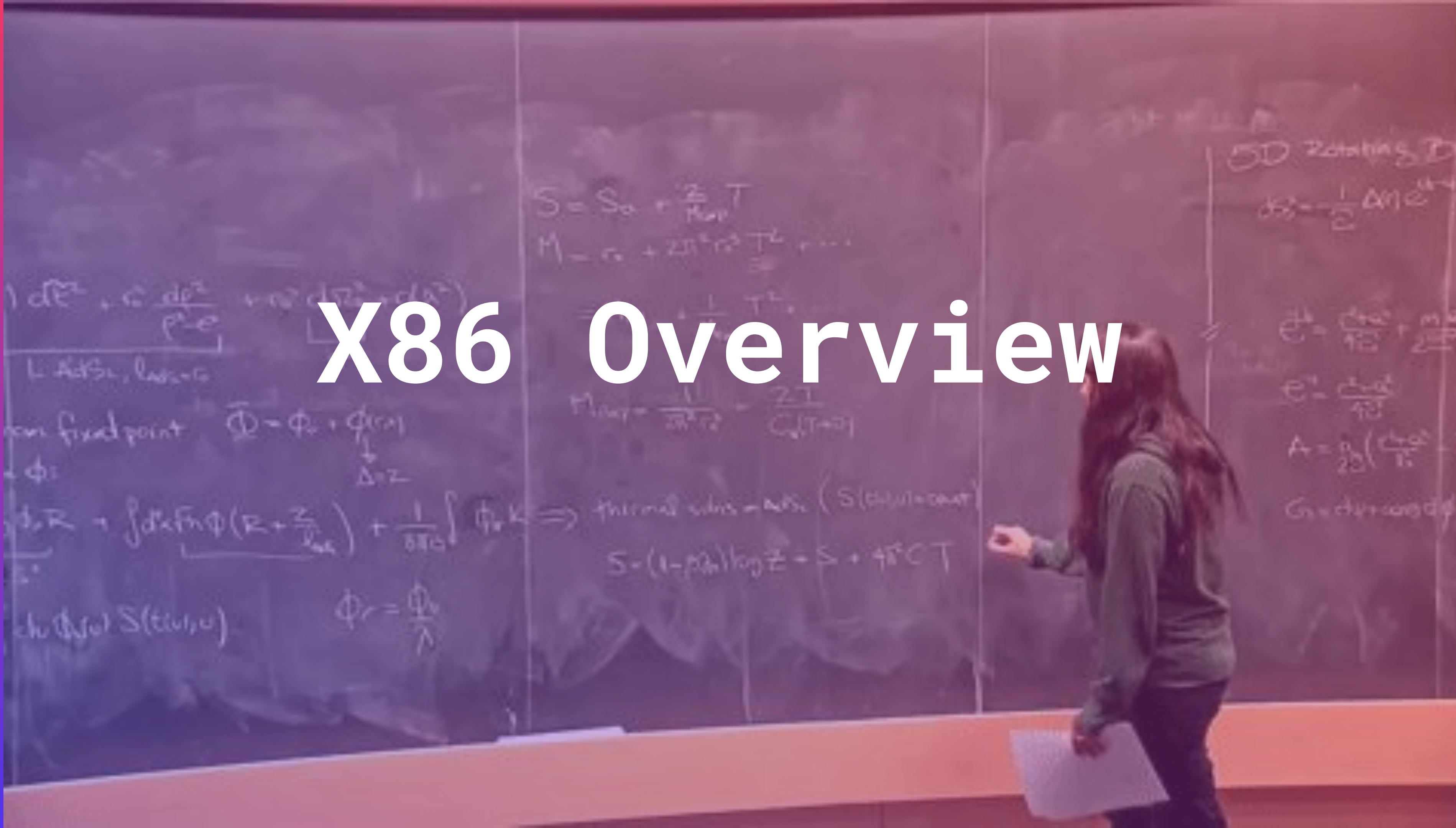
Sommaire



SOMMAIRE

- Assembly language introduction (here)
    - Compilation & Disassembly
    - x86 architecture
    - Instructions
    - Registers
    - Casts
    - Data
    - Branching
    - The Stack
    - C to x86
- Kahoot Quiz to test your new knowledges
- Introduction to IDA
- Two short TDs to begin
- TP : hack the minesweeper

X86 Overview



Why Assembly?

Assembly is the most popular low-level language* (more precisely, a class of languages)

Low-level language: a human-readable version of a computer architecture's instruction set.

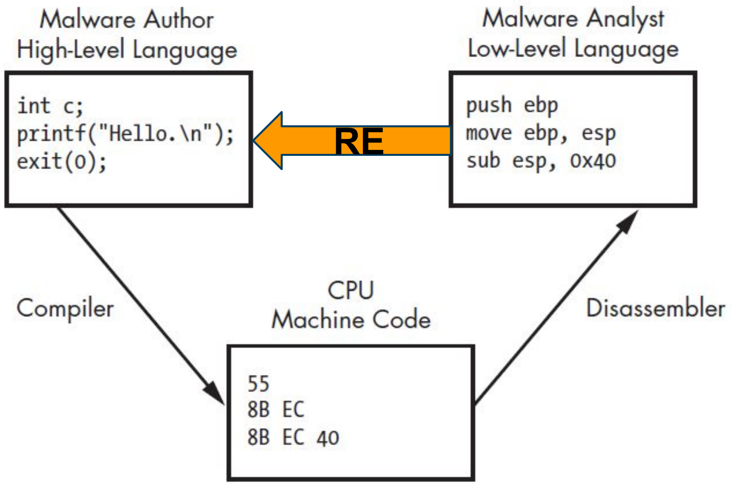
Compilation & Disassembly

The program author writes code in some high-level language, say C.

The C code is compiled into machine code - a series of bytes that the CPU understands.

The researcher usually has no access to the C source code, only to the bytes of machine code.

To make life easier, a disassembler translates these bytes into an easier-to-read textual representation.

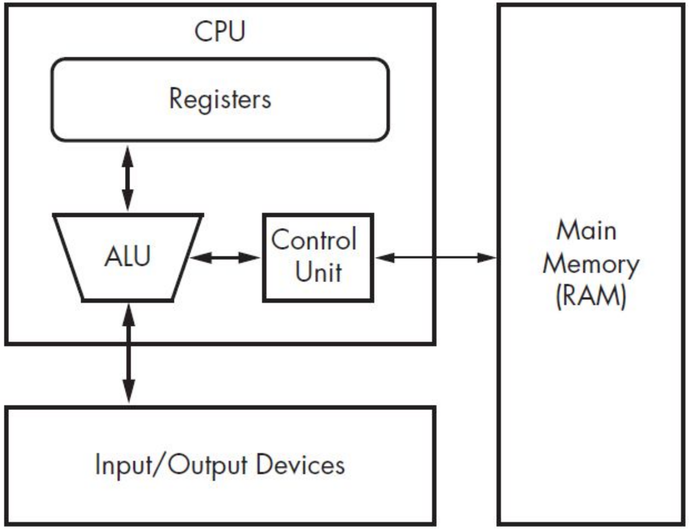


x86 Architecture

As a program runs, the following loop is executed:

1. A CPU instruction is read from the main memory by the Control Unit
2. The instruction is processed and executed by the Arithmetic-Logic Unit, along with input from the user or the registers
3. The operation's output is stored in the CPU's registers or sent to an output device

Let's understand what an x86 instruction looks like.

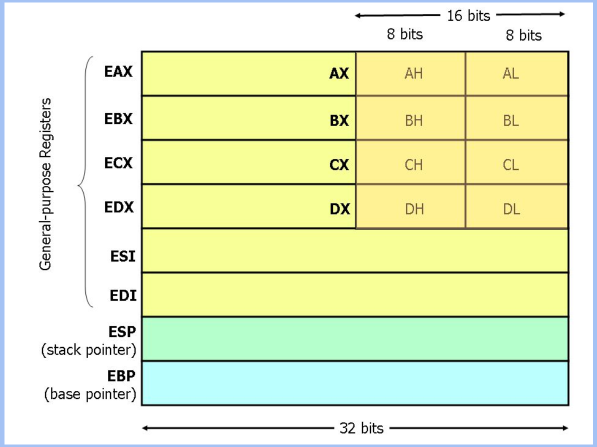


Registers

Registers

A register is the CPU’s basic data storage unit, whose access time is the fastest.

General Registers



Segment Registers

CS [code section]
SS [stack section]
DS [data section]
ES, FS, GS [general]

These point to the base address of different memory sections

EFLAGS

a register with 32 bit-flags that provide information on previous operations (TBC)

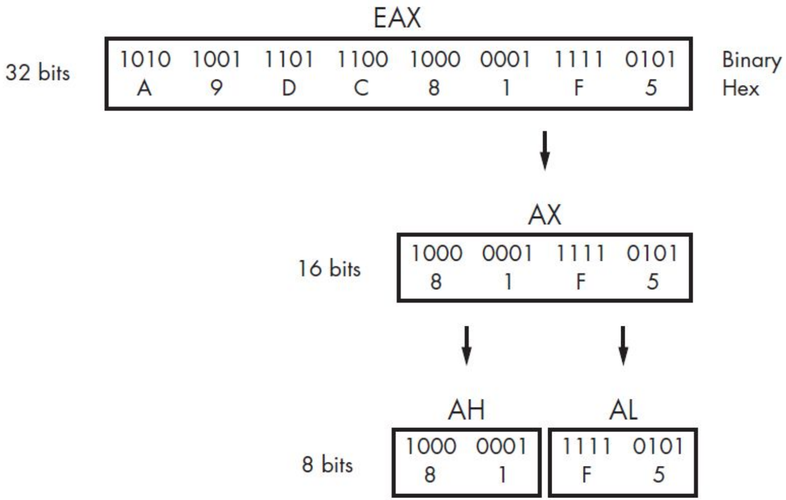
EIP

This register always holds the address of the next instruction to execute

Register Breakdown

EAX, EBX, ECX & EDX can be broken-down as follows (EAX is used as example):

- EAX - all 32 bits
- AX - 16 least-significant bits of EAX
- AH - 8 most-significant bits of AX
- AL - 8 least-significant bits of AX

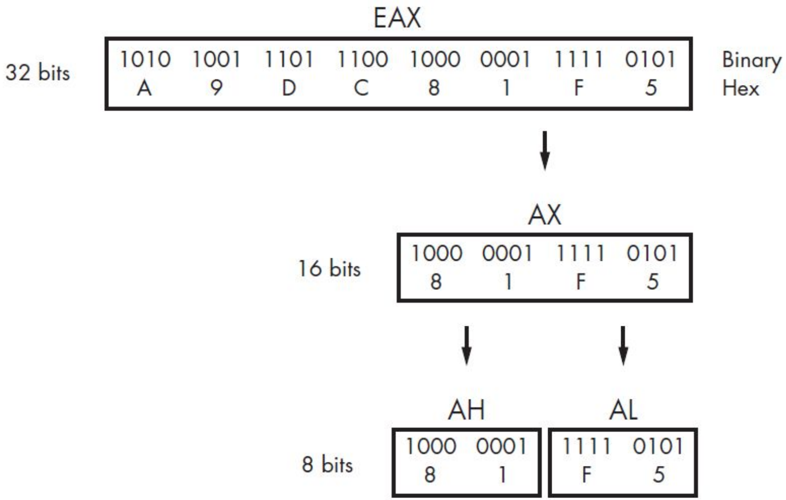


Can you think how the 16 most-significant bits of EAX can be accessed? (answer is in the next slide)

Register Breakdown

EAX, EBX, ECX & EDX can be broken-down as follows (EAX is used as example):

- EAX - all 32 bits
- AX - 16 least-significant bits of EAX
- AH - 8 most-significant bits of AX
- AL - 8 least-significant bits of AX



Can you think how the 16 most-significant bits of EAX can be accessed?

SHR EAX, 0x10 → Use AX

Register Conventions

- EAX - A function’s **return** value
- ECX - **Counters** (loop variables)
- EAX:EDX - **Quotient** and **remainder** in multiplication and division

EFLAGS Register

32 bit-flags that give information on the result of previous computation. The most common flags are:

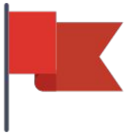
- **Zero Flag**: set if an operation result is 0; otherwise cleared.
- **Carry Flag**: set if an operation results is too large or too small for the destination operand; otherwise cleared
- **Sign Flag**: set if the MSB is set (namely, the result is negative); otherwise cleared

EFLAGS Quiz

```
mov     eax,  0x1

mov     ebx,  0x0

sub     ebx,  eax
```

What is the status of the flags (answers are in the next slides):

- zero-flag?
- carry-flag?
- sign-flag?

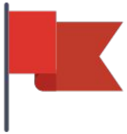


EFLAGS Quiz

```
mov     eax, 0x1
mov     ebx, 0x0
sub     ebx, eax
```

What is the status of the flags :

- Zero-flag? 0
- carry-flag?
- sign-flag?



EFLAGS Quiz

```
mov        eax,  0x1

mov        ebx,  0x0

sub        ebx,  eax
```

What is the status of the flags :

- Zero-flag?   0
- Carry-flag?   0
- Sign-flag?

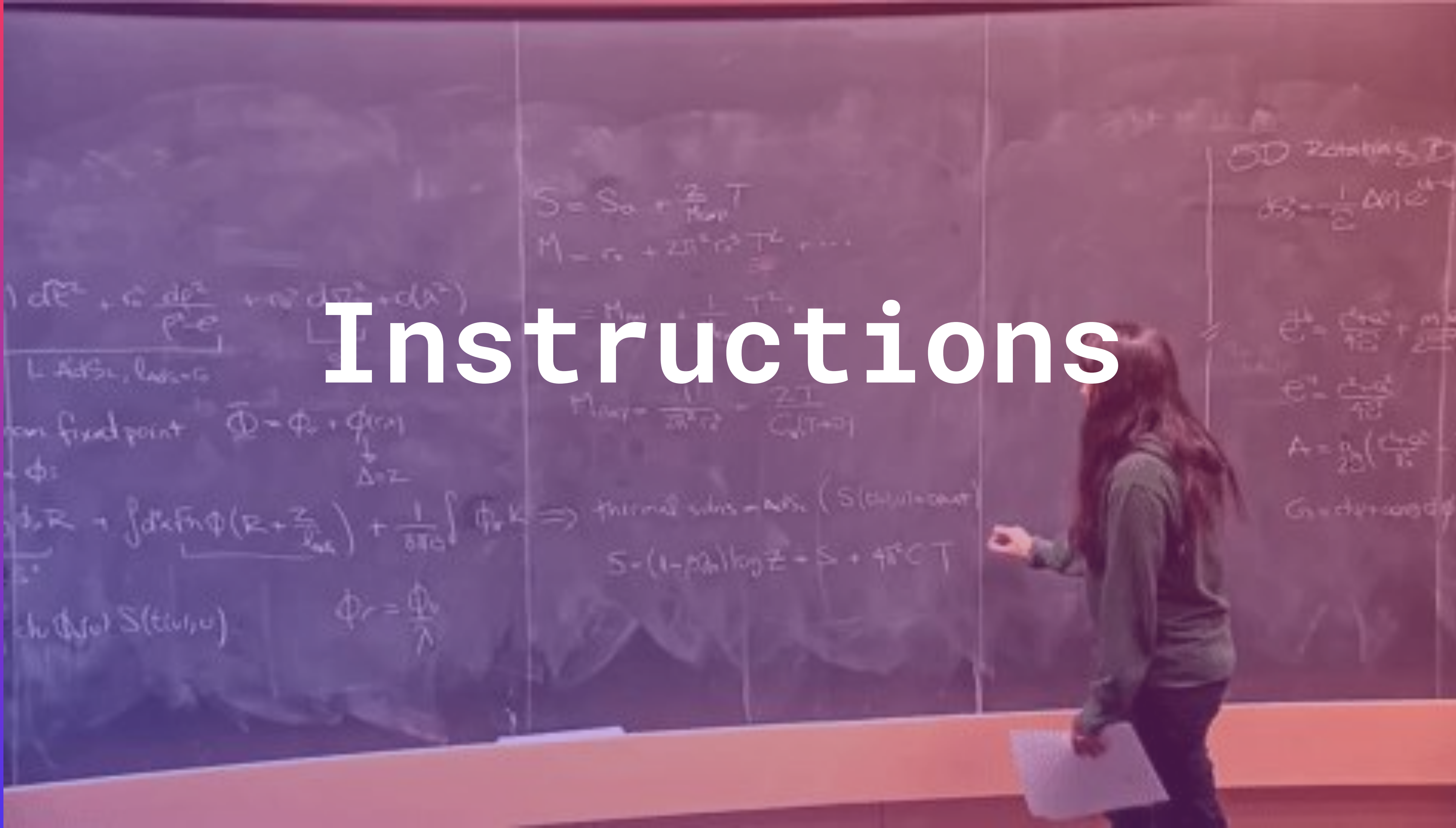
EFLAGS Quiz

```
mov         eax,  0x1

mov         ebx,  0x0

sub         ebx,  eax
```

What is the status of the flags :

- Zero-flag? 0
- Carry-flag? 0
- Sign-flag? 1

Instructions



Instructions

Assembly instruction = mnemonic + optional operand(s). For example:

    mov eax, 0xFF ~ B8 FF 00 00 00

An operand can be:

- an immediate - 0x3
- a register - eax
- a memory address - [0x400100 + 4]

Opcode: the bytes that correspond to the instruction and its operands

Popular Instructions

Data storage

- mov
- lea

Arithmetic

- add
- sub
- inc
- dec
- mul
- div

Logic

- or
- and
- xor
- shr
- shl

Stack

- push
- pop
- call
- ret

Control-Flow

- test
- cmp
- jmp
- jcc

Quiz

What does each of the following instructions do? (answers are in the next slides)

- mov eax, ebx
- mov eax, 0x42
- mov eax, [0x4037c4]
- mov eax, [ebx]
- mov eax, [ebx+esi*4]

Quiz

What does each of the following instructions do?

- mov eax, ebx — ***move what's in ebx to eax***
- mov eax, 0x42
- mov eax, [0x4037c4]
- mov eax, [ebx]
- mov eax, [ebx+esi*4]

Quiz

What does each of the following instructions do?

- mov eax, ebx — ***move what's in ebx to eax***
- mov eax, 0x42 — ***move 0x42 to eax***
- mov eax, [0x4037c4]
- mov eax, [ebx]
- mov eax, [ebx+esi*4]

Quiz

What does each of the following instructions do?

- mov eax, ebx ***move what's in ebx to eax***
- mov eax, 0x42 ***move 0x42 to eax***
- mov eax, [0x4037c4] ***move what's in address 0x4037c4 to eax***
- mov eax, [ebx]
- mov eax, [ebx+esi*4]

Quiz

What does each of the following instructions do?

- mov eax, ebx — ***move what's in ebx to eax***
- mov eax, 0x42 — ***move 0x42 to eax***
- mov eax, [0x4037c4] — ***move what's in address 0x4037c4 to eax***
- mov eax, [ebx] — ***move what's in the address in ebx to eax***
- mov eax, [ebx+esi*4]

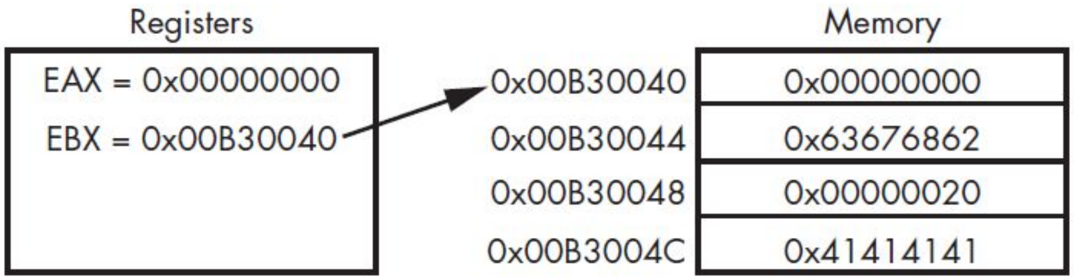
Quiz

What does each of the following instructions do?

• mov eax, ebx	***move what's in ebx to eax***
• mov eax, 0x42	***move 0x42 to eax***
• mov eax, [0x4037c4]	***move what's in address 0x4037c4 to eax***
• mov eax, [ebx]	***move what's in the address in ebx to eax***
• mov eax, [ebx+esi*4]	***move what's in address ebx+esi*4 to eax***

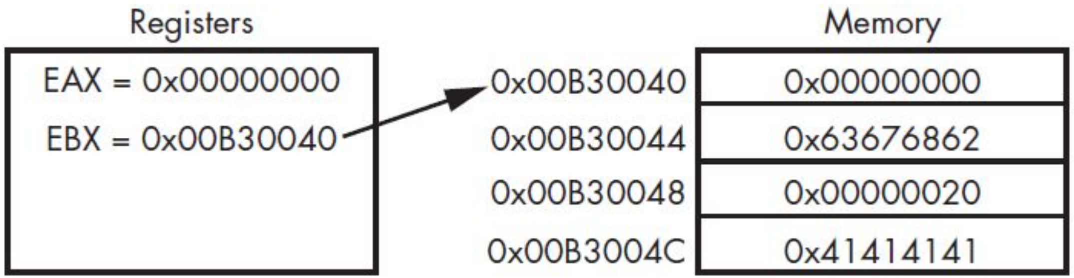
Quiz

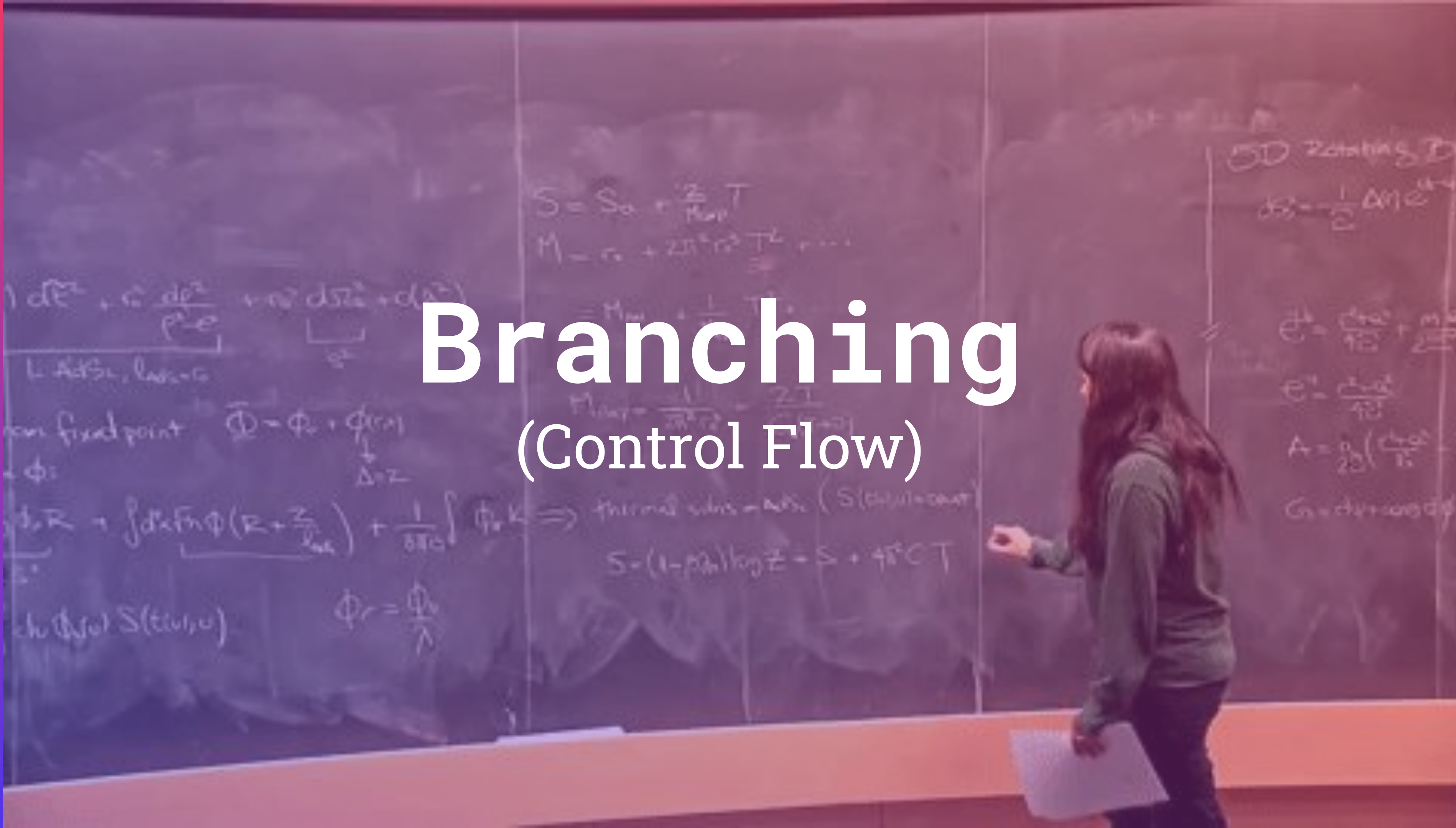
- What's the difference? (answer is in the next slide)
    - mov eax, [ebx + 8]
    - lea eax, [ebx + 8]



Quiz

- What's the difference?
  - mov eax, [ebx + 8]   ***move what's in address ebx+8 (0x20) to eax***
  - lea eax, [ebx + 8]   ***move the value ebx+8 (0xB30048) to eax***





Branching

(Control Flow)

Branching

- Two types of jumps in x86:
    - Unconditional - just jump to where I tell you
        - jmp 0x401072
    - Conditional - check the result of some computation, jump accordingly
        - cmp eax, 0x10 compare the value of eax with 0x10
        - jge 0x401072 jump to 0x401072 if eax is greater\equal than 0x10

Size Directives:

- 1 byte - 8 bits : BYTE PTR [...]
- 2 byte - 16 bits : WORD PTR [...]
- 4 byte - 32 bits : DWORD PTR [...]

```
mov BYTE PTR [ebx], 2   ; Move 2 into the single byte at the address stored in EBX.
mov WORD PTR [ebx], 2   ; Move the 16-bit integer representation of 2 into the 2 bytes starting at the address in EBX.
mov DWORD PTR [ebx], 2  ; Move the 32-bit integer representation of 2 into the 4 bytes starting at the address in EBX.

mov esi, DWORD PTR [rax*4+0x419260]
```

Static data regions (analogous to global variables)

- DB : 1 byte data location
- DW : 2 bytes
- DD : 4 bytes

```
.DATA
var    DB 64       ; Declare a byte, referred to as location var, containing the value 64.

var2   DB ?        ; Declare an uninitialized byte, referred to as location var2.

       DB 10       ; Declare a byte with no label, containing the value 10. Its location is var2 + 1

X      DW ?        ; Declare a 2-byte uninitialized value, referred to as location X.

Y      DD 30000    ; Declare a 4-byte value, referred to as location Y, initialized to 30000.
```

Static data regions (analogous to global variables)

Arrays in x86 assembly language are simply a number of cells located contiguously in memory.

* The **DUP** directive tells the assembler to duplicate an expression a given number of times. For example, 4 DUP(2) is equivalent to 2, 2, 2, 2.

```
Z     DD 1, 2, 3      ; Declare three 4-byte values, initialized to 1, 2, and 3. The value of location Z + 8
                        will be 3.

bytes DB 10 DUP(?)    ; Declare 10 uninitialized bytes starting at location bytes.

arr   DD 100 DUP(0)   ; Declare 100 4-byte words starting at location arr, all initialized to 0

str   DB 'hello',0    ; Declare 6 bytes starting at the address str, initialized to the ASCII character values
                        for hello and the null (0) byte.
```

Jumps

There are many types of **conditional** jumps. The set of conditional jump instructions is often referred to as jcc (where the j stands for jump and the c for condition).

Each jump instruction performs different checks on the **EFLAGS** register to determine whether the jump should be performed or not.

Instruction	Description
`jz loc`	Jump to specified location if ZF = 1.
`jnz loc`	Jump to specified location if ZF = 0.
`je loc`	Same as `jz`, but commonly used after a `cmp` instruction. Jump will occur if the destination operand equals the source operand.
`jne loc`	Same as `jnz`, but commonly used after a `cmp`. Jump will occur if the destination operand is not equal to the source operand.
`jg loc`	Performs signed comparison jump after a `cmp` if the destination operand is greater than the source operand.
`jge loc`	Performs signed comparison jump after a `cmp` if the destination operand is greater than or equal to the source operand.
`ja loc`	Same as `jg`, but an unsigned comparison is performed.
`jae loc`	Same as `jge`, but an unsigned comparison is performed.
`jl loc`	Performs signed comparison jump after a `cmp` if the destination operand is less than the source operand.
`jle loc`	Performs signed comparison jump after a `cmp` if the destination operand is less than or equal to the source operand.
`jb loc`	Same as `jl`, but an unsigned comparison is performed.
`jbe loc`	Same as `jle`, but an unsigned comparison is performed.
`jo loc`	Jump if the previous instruction set the overflow flag (OF = 1).
`js loc`	Jump if the sign flag is set (SF = 1).
`jecxz loc`	Jump to location if ECX = 0.

Conditionals

- Two operations are commonly used before conditional jumps:
    - **test** ~ perform a logical AND
    - **cmp** ~ perform subtraction

Both instructions do not store the result, but change the flags in **EFLAGS** as needed

An Example

Here's a pair of instructions you will see a lot (not necessarily with eax…)

```
test eax, eax

jz 0x400100
```

Since **test** is practically a logical AND, and since X AND X == X is always true, the result of the first line is either zero (if EAX == 0) or some non-zero value (otherwise).

Therefore, this is an efficient way of checking if EAX **equals zero or not.**

How would you say in x86...?

- if (a == b) goto 0x1000;
- if (a < b) goto 0x1000;
- if (a) goto 0x1000;

How would you say in x86...?

- if (a == b) goto 0x1000; **cmp a, b; jz 0x1000;**
- if (a < b) goto 0x1000;
- if (a) goto 0x1000;

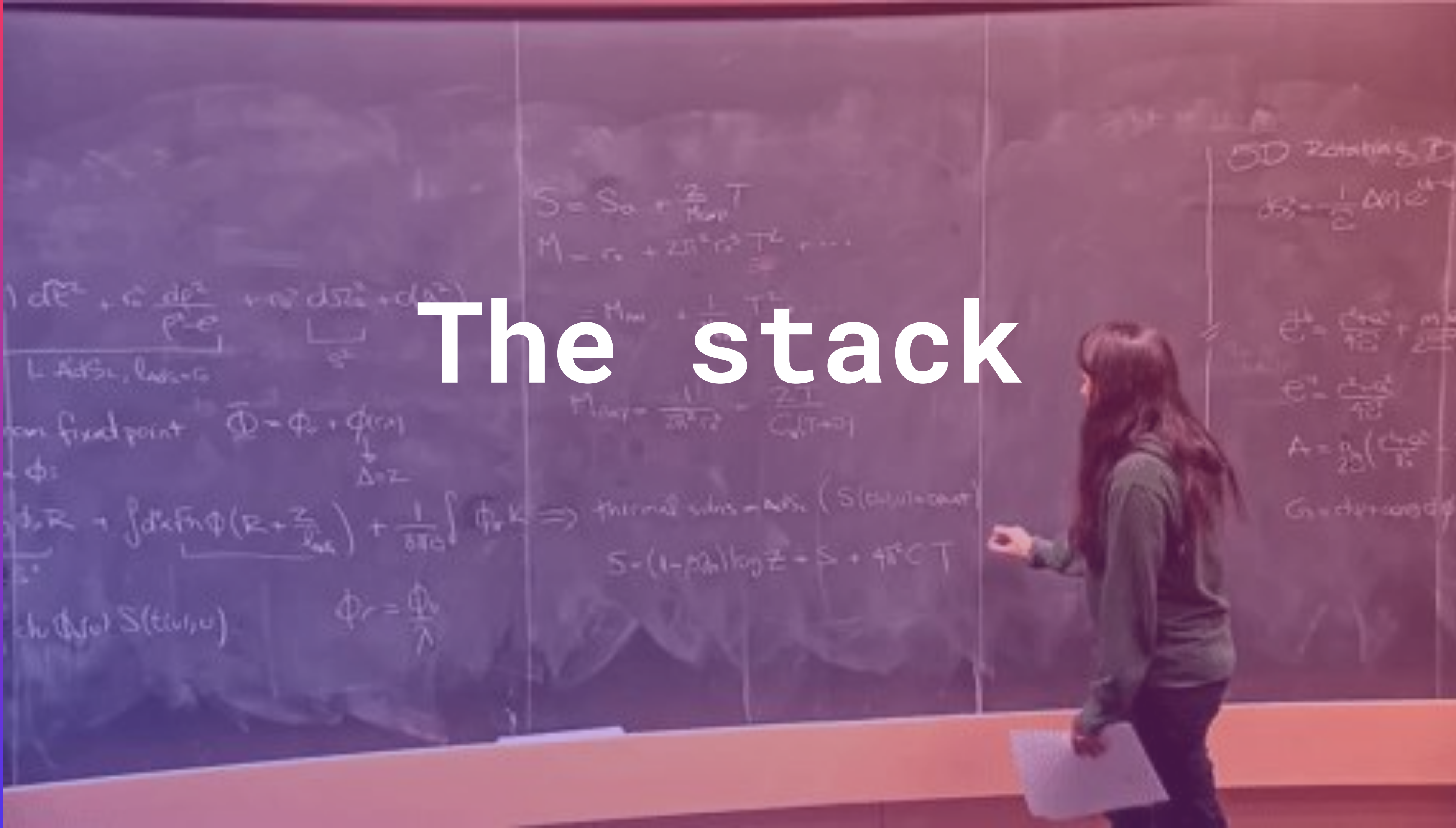
How would you say in x86…?

- if (a == b) goto 0x1000; **cmp a, b; jz 0x1000;**
- if (a < b) goto 0x1000; **cmp a,b; jl 0x1000;**
- if (a) goto 0x1000;

How would you say in x86…?

- if (a == b) goto 0x1000; **cmp a, b; jz 0x1000;**
- if (a < b) goto 0x1000; **cmp a,b; jl 0x1000;**
- if (a) goto 0x1000; **test a, a; jnz 0x1000;**

The stack

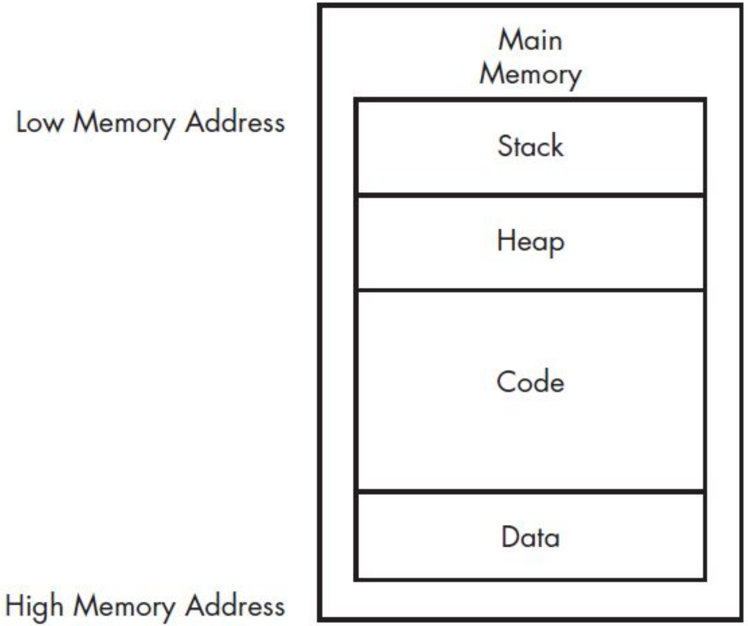


Main Memory

We have seen before how the CPU “talks” to the main memory (“RAM”).

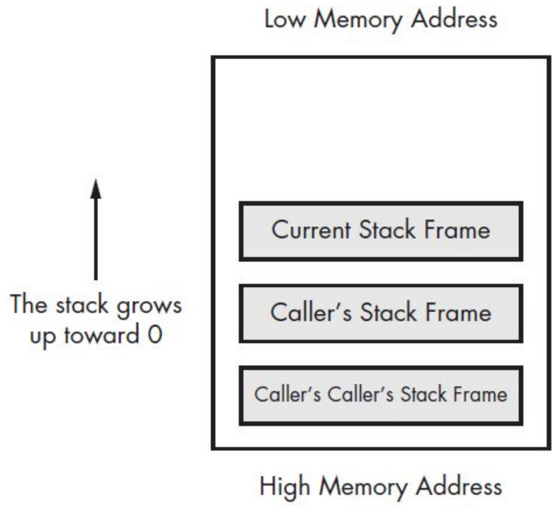
The following are (some of) the different **sections** of a process loaded into the RAM:

- **Data**: static / global variables, put in place when the program is loaded
- **Code**: the program’s CPU instructions
- **Heap**: dynamic memory, allocated and freed during runtime
- **Stack**: variables and arguments local to the program’s functions



The Stack

- A LIFO data structure with push & pop operations
- Stack-relevant registers:
    - **ESP - Stack Pointer**, always points to the top of the stack, therefore dynamic
    - **EBP - Base Pointer**, stays consistent within a function and is used to reference the function's local variables and parameters
- Stack-relevant instructions: pop, push, call, ret (and also enter, leave…)



Function Calls - Demo

In the following slides, x86 code will be presented along with the stack in its current state.

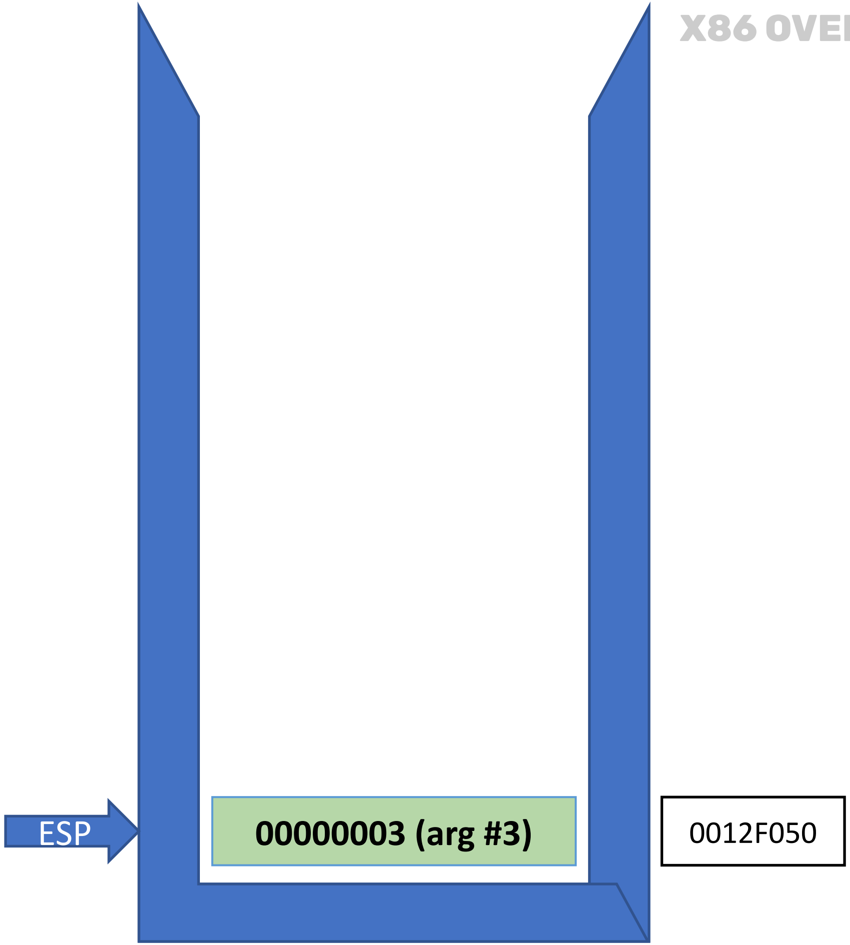
The code shows how a function named **caller** calls another function sum.

Each slide, look at the next instruction and try to predict the side-effects and changes to come.

Note: instructions in bold have been just executed and their side-effects are already presented on the slide.

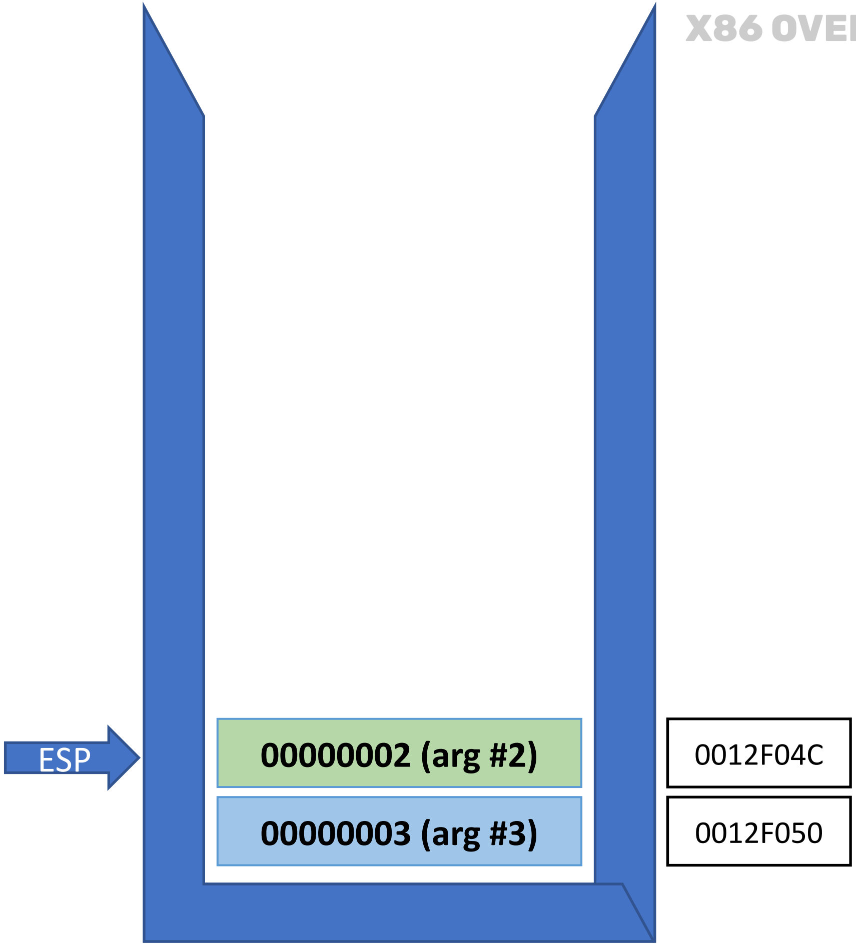
```
caller:
        …
00401296    push      3
00401298    push      2
0040129A    push      1
0040129C    call      sum
004012A1    add       esp, 0xC

sum:
00401300    push      ebp
00401301    mov       ebp, esp
        ; calculate sum
00401303    sub       esp, 0x8
00401306     …
00401320    mov       eax, 0x6
00401328    mov       esp, ebp
0040132A    pop       ebp
0040132B    ret
```

Registers	
EIP	**00401298**
EBP	0012F072
ESP	**0012F050**
EAX	00000000



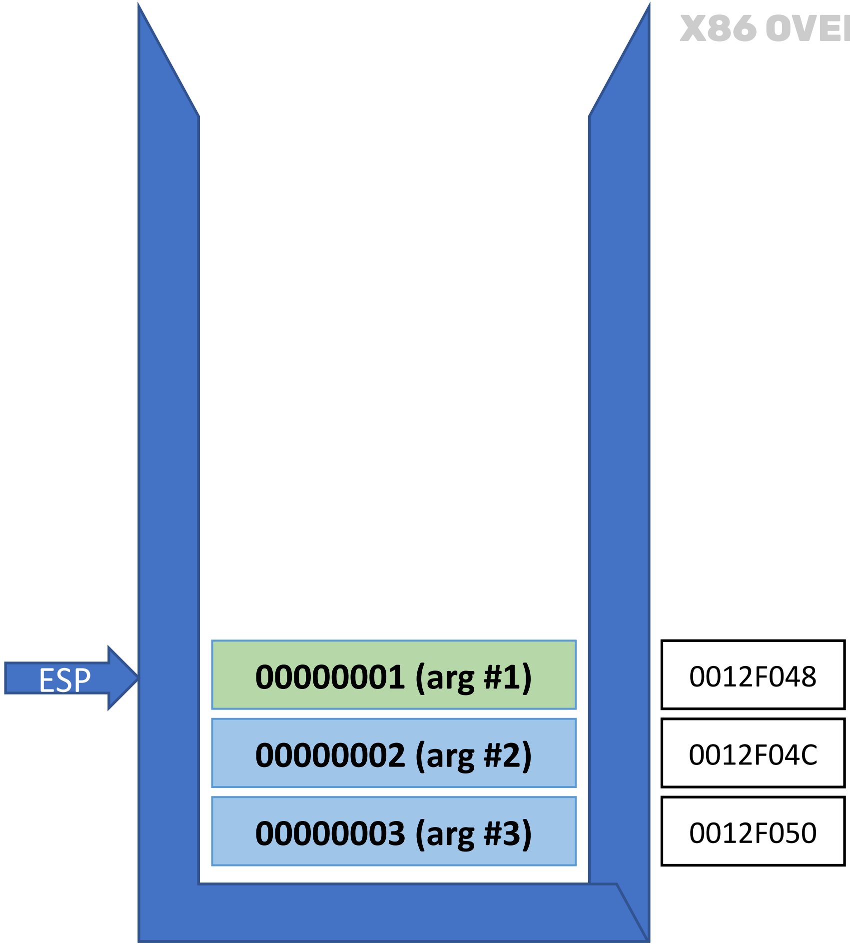
```
caller:
        …
00401296    push     3
00401298    push     2
0040129A    push     1
0040129C    call     sum
004012A1    add      esp, 0xC

sum:<<<
00401300    push     ebp
00401301    mov      ebp, esp
        ; calculate sum
00401303    sub      esp, 0x8
00401306     …
00401320    mov      eax, 0x6
00401328    mov      esp, ebp
0040132A    pop      ebp
0040132B    ret
```

Registers	
EIP	**0040129A**
EBP	0012F072
ESP	**0012F04C**
EAX	00000000



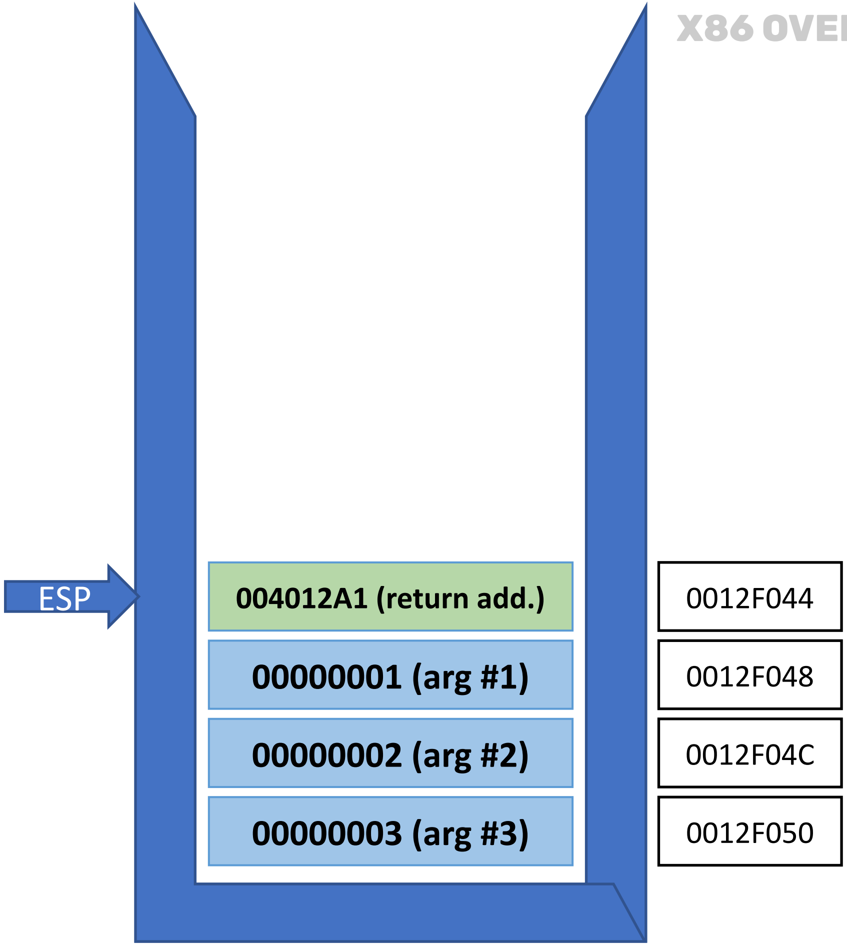
```
caller:
        …
00401296    push      3
00401298    push      2
0040129A    push      1
0040129C    call      sum
004012A1    add       esp, 0xC

sum:
00401300    push      ebp
00401301    mov       ebp, esp
        ; calculate sum
00401303    sub       esp, 0x8
00401306     …
00401320    mov       eax, 0x6
00401328    mov       esp, ebp
0040132A    pop       ebp
0040132B    ret
```

Registers	
EIP	**0040129C**
EBP	0012F072
ESP	**0012F048**
EAX	00000000



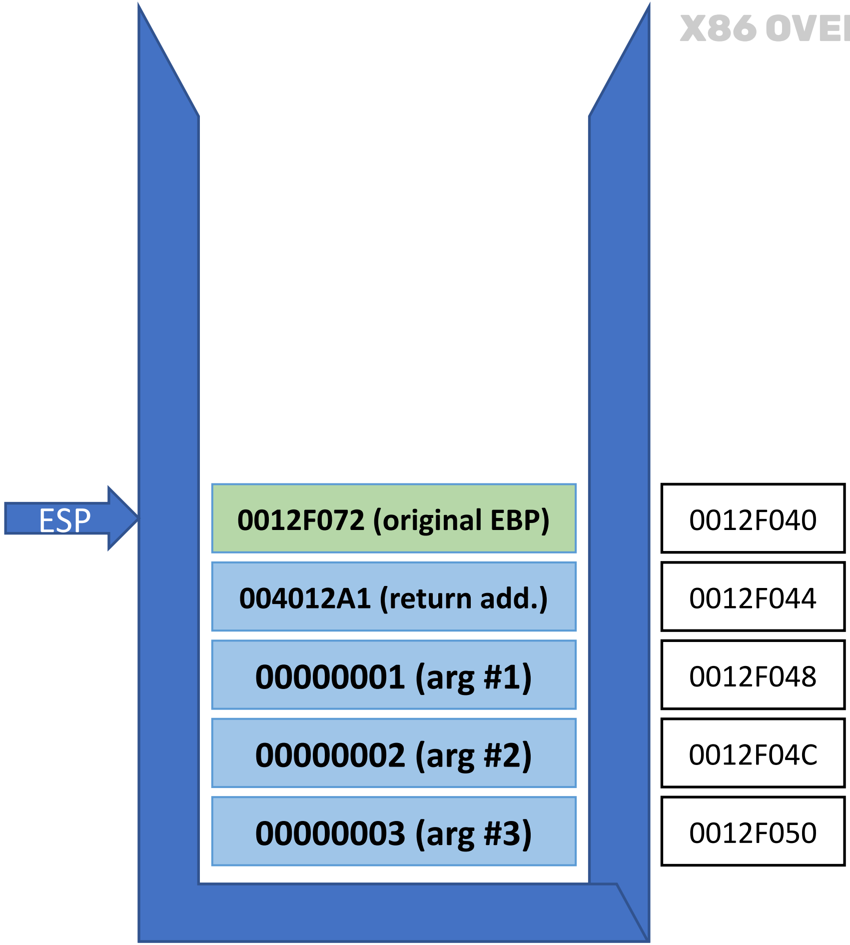
```
caller:
        …
00401296    push    3
00401298    push    2
0040129A    push    1
0040129C    call    sum
004012A1    add     esp, 0xC

sum:
00401300    push    ebp
00401301    mov     ebp, esp
        ; calculate sum
00401303    sub     esp, 0x8
00401306     …
00401320    mov     eax, 0x6
00401328    mov     esp, ebp
0040132A    pop     ebp
0040132B    ret
```

Registers	
EIP	**00401300**
EBP	0012F072
ESP	**0012F044**
EAX	00000000



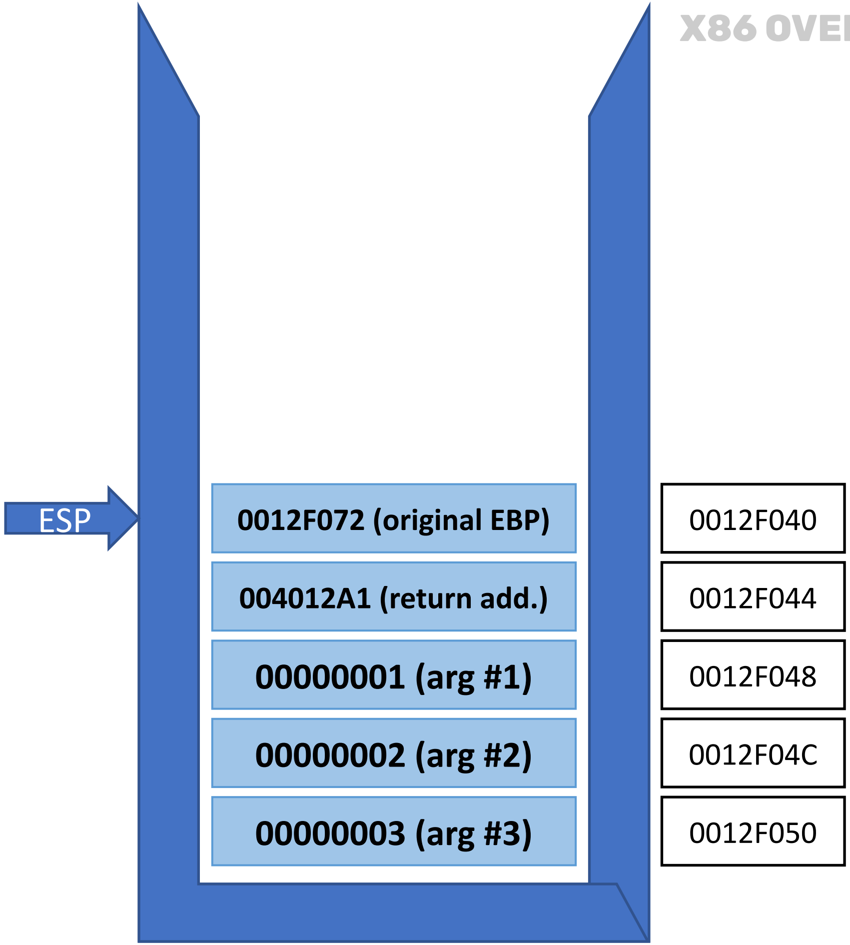
```
caller:
        …
00401296    push      3
00401298    push      2
0040129A    push      1
0040129C    call      sum
004012A1    add       esp, 0xC

sum:
00401300    push      ebp
00401301    mov       ebp, esp
        ; calculate sum
00401303    sub       esp, 0x8
00401306     …
00401320    mov       eax, 0x6
00401328    mov       esp, ebp
0040132A    pop       ebp
0040132B    ret
```

Registers	
EIP	**00401301**
EBP	0012F072
ESP	**0012F040**
EAX	00000000



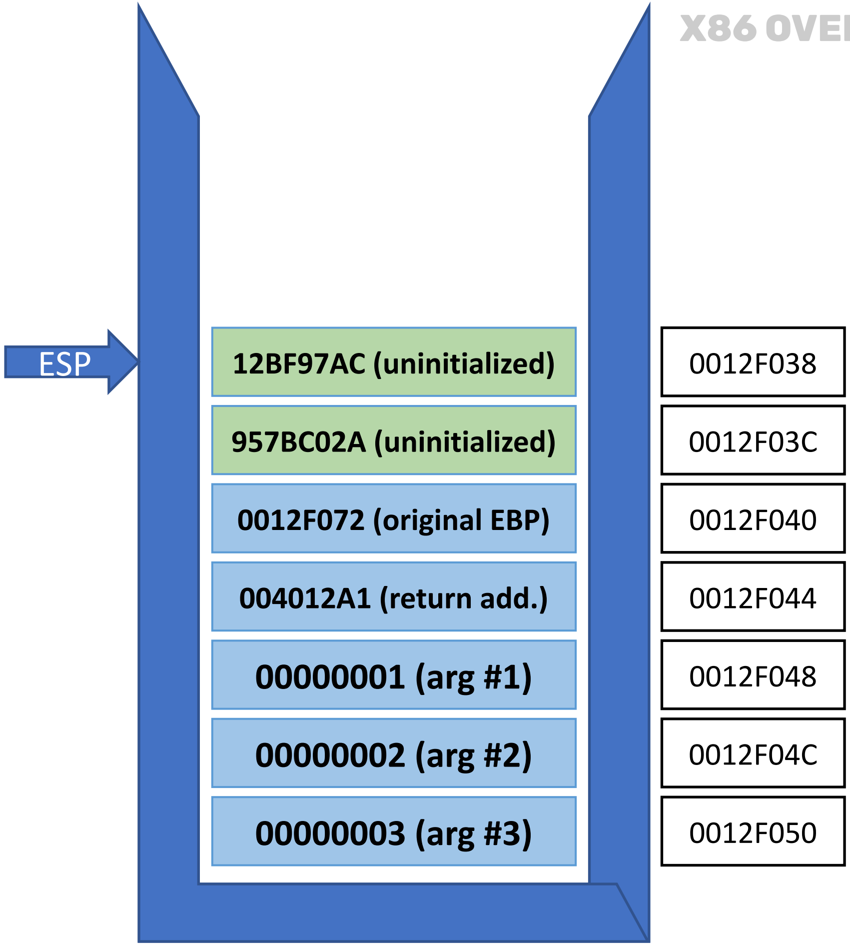
```
caller:
        …
00401296    push      3
00401298    push      2
0040129A    push      1
0040129C    call      sum
004012A1    add       esp, 0xC

sum:
00401300    push      ebp
00401301    mov       ebp, esp
        ; calculate sum
00401303    sub       esp, 0x8
00401306     …
00401320    mov       eax, 0x6
00401328    mov       esp, ebp
0040132A    pop       ebp
0040132B    ret
```

Registers	
EIP	**00401303**
EBP	**0012F040**
ESP	0012F040
EAX	00000000



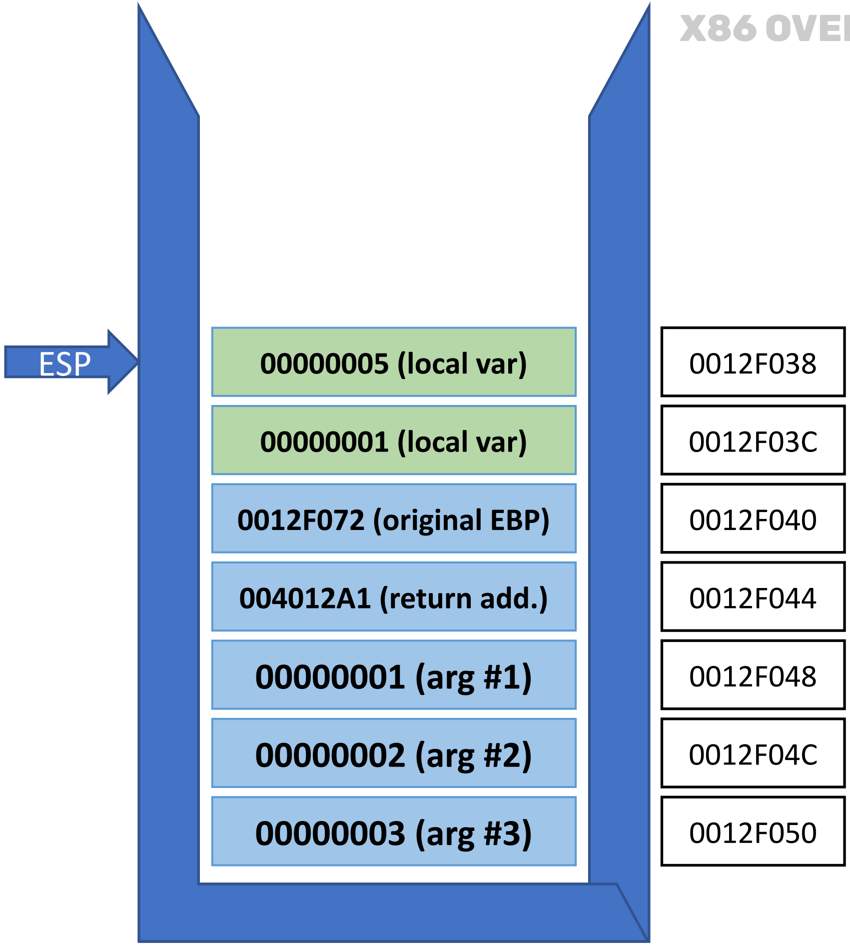
```
caller:
        …
00401296    push     3
00401298    push     2
0040129A    push     1
0040129C    call     sum
004012A1    add      esp, 0xC

sum:
00401300    push     ebp
00401301    mov      ebp, esp
       ; calculate sum
00401303    sub      esp, 0x8
00401306     …
00401320    mov      eax, 0x6
00401328    mov      esp, ebp
0040132A    pop      ebp
0040132B    ret
```

Registers	
EIP	**00401306**
EBP	0012F040
ESP	**0012F038**
EAX	00000000



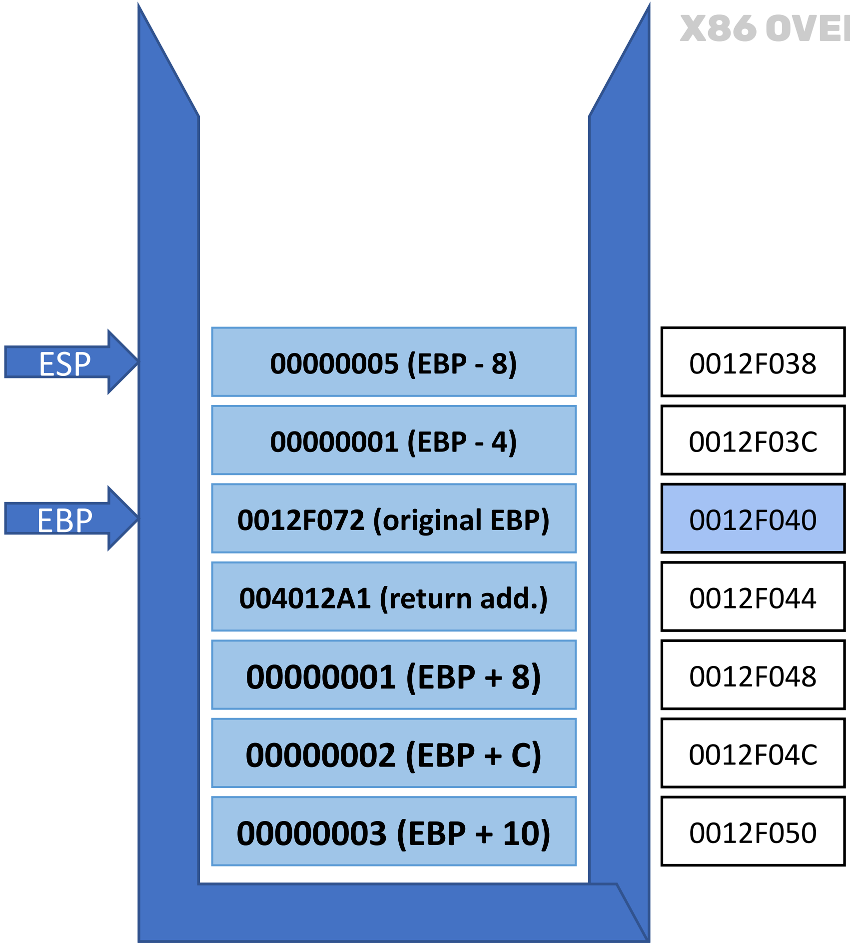
```
caller:
        …
00401296    push     3
00401298    push     2
0040129A    push     1
0040129C    call     sum
004012A1    add      esp, 0xC

sum:
00401300    push     ebp
00401301    mov      ebp, esp
        ; calculate sum
00401303    sub      esp, 0x8
00401306     …
00401320    mov      eax, 0x6
00401328    mov      esp, ebp
0040132A    pop      ebp
0040132B    ret
```

Registers	
EIP	**00401320**
EBP	0012F040
ESP	0012F038
EAX	00000000



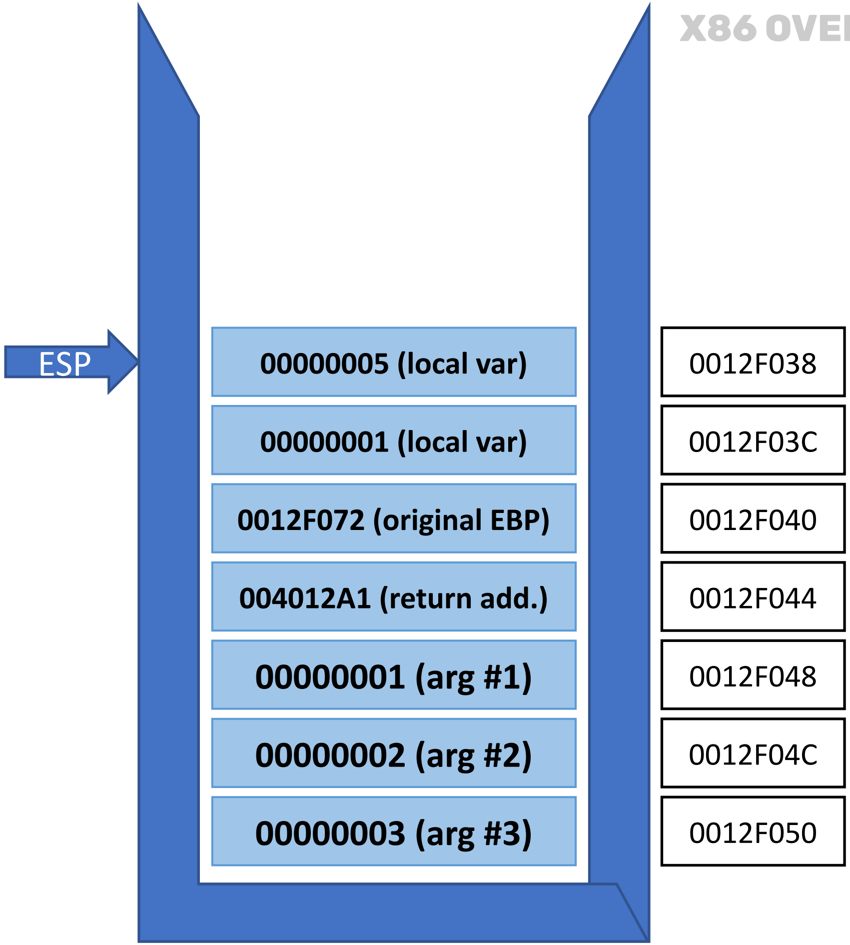
```
caller:
        …
00401296    push     3
00401298    push     2
0040129A    push     1
0040129C    call     sum
004012A1    add      esp, 0xC

sum:
00401300    push     ebp
00401301    mov      ebp, esp
       ; calculate sum
00401303    sub      esp, 0x8
00401306     …
00401320    mov      eax, 0x6
00401328    mov      esp, ebp
0040132A    pop      ebp
0040132B    ret
```

Registers	
EIP	**00401320**
EBP	0012F040
ESP	0012F038
EAX	00000000



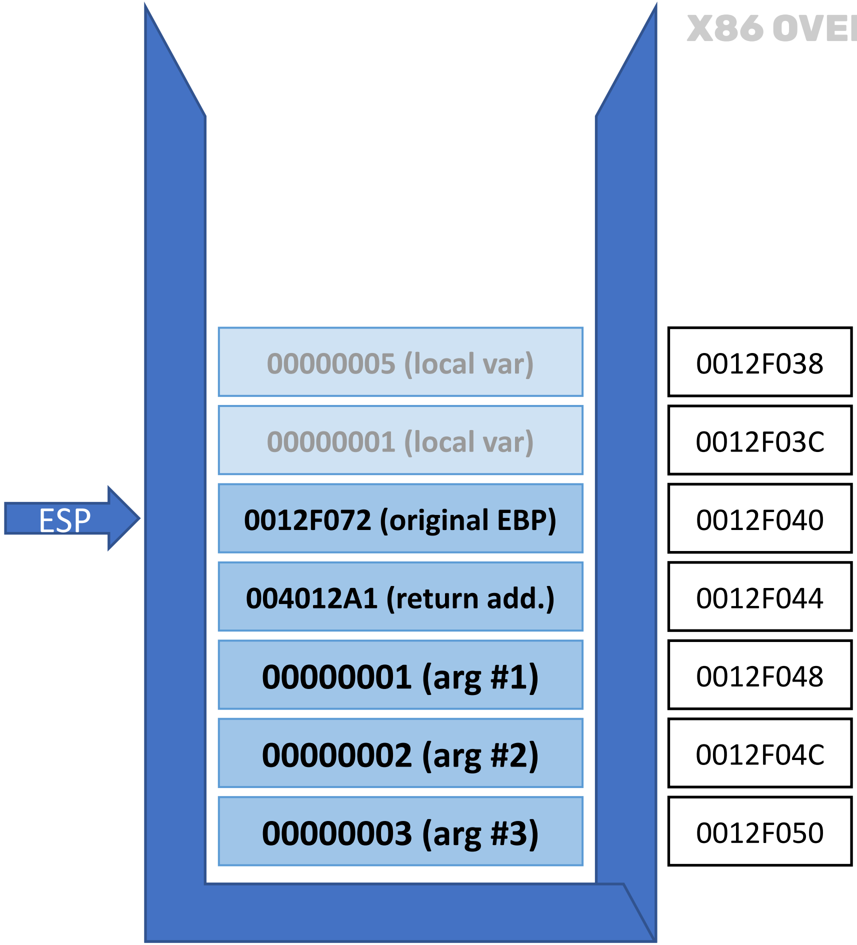
```
caller:
        …
00401296     push       3
00401298     push       2
0040129A     push       1
0040129C     call       sum
004012A1     add        esp, 0xC

sum:
00401300     push       ebp
00401301     mov        ebp, esp
       ; calculate sum
00401303     sub        esp, 0x8
00401306      …
00401320     mov        eax, 0x6
00401328     mov        esp, ebp
0040132A     pop        ebp
0040132B     ret
```

Registers	
EIP	**00401328**
EBP	0012F040
ESP	0012F038
EAX	**00000006**



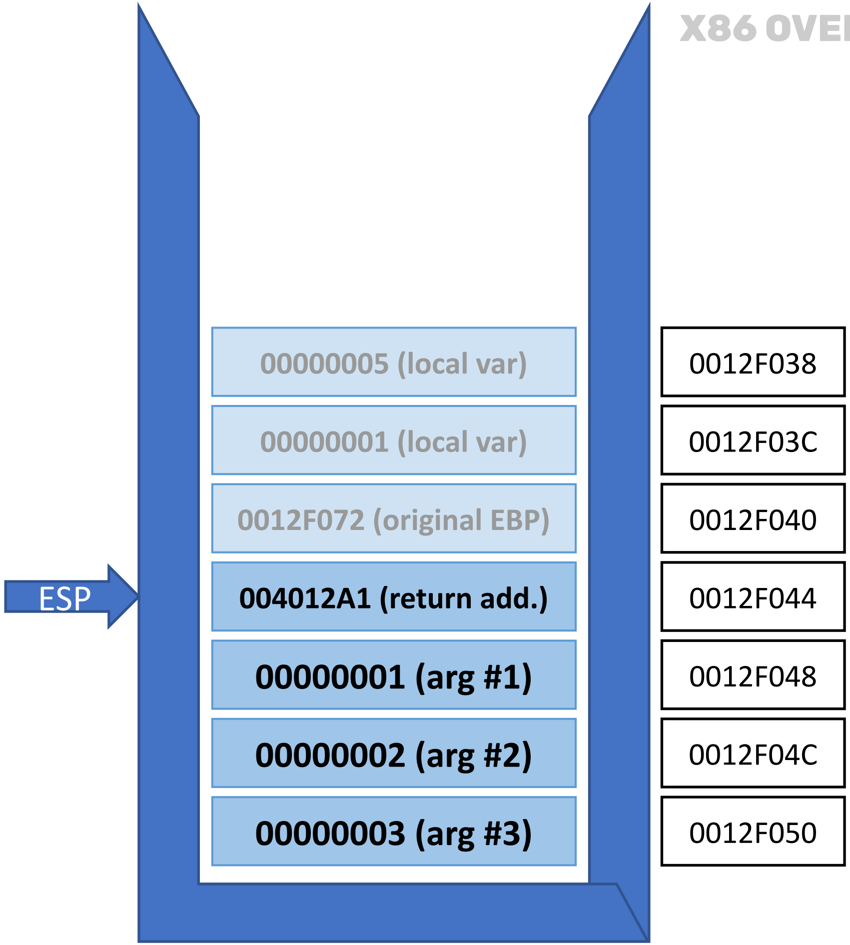
```
caller:
        …
00401296    push      3
00401298    push      2
0040129A    push      1
0040129C    call      sum
004012A1    add       esp, 0xC

sum:
00401300    push      ebp
00401301    mov       ebp, esp
        ; calculate sum
00401303    sub       esp, 0x8
00401306     …
00401320    mov       eax, 0x6
00401328    mov       esp, ebp
0040132A    pop       ebp
0040132B    ret
```

Registers	
EIP	**0040132A**
EBP	0012F040
ESP	**0012F040**
EAX	00000006



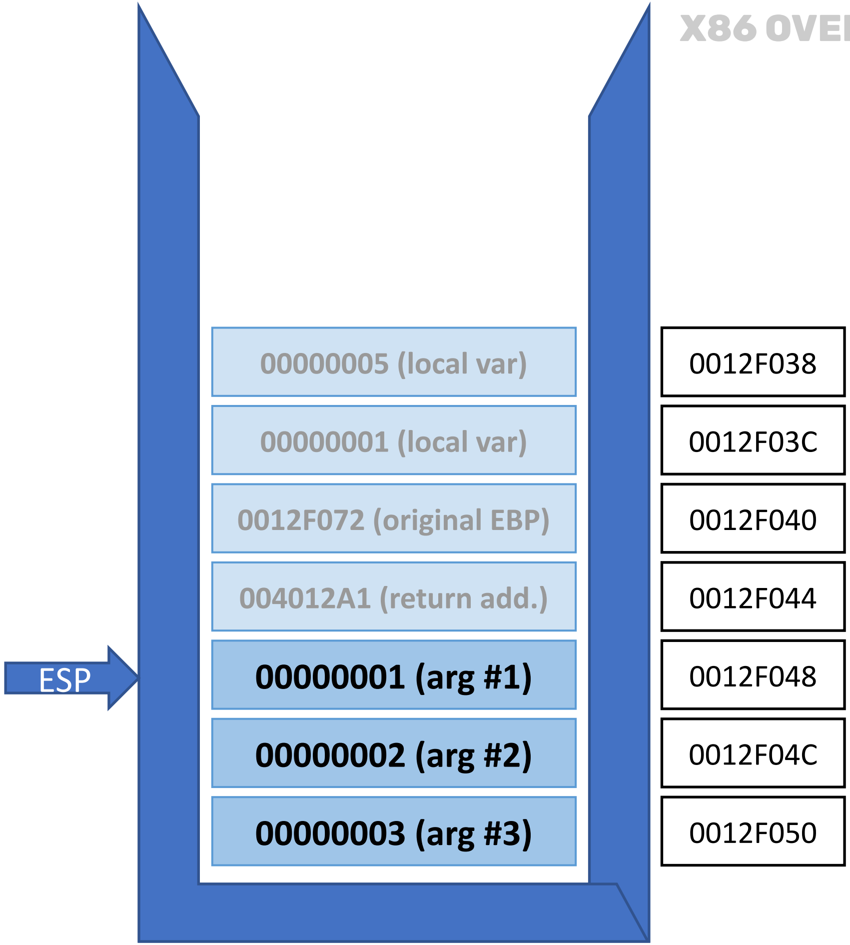
```
caller:
        …
00401296    push     3
00401298    push     2
0040129A    push     1
0040129C    call     sum
004012A1    add      esp, 0xC

sum:
00401300    push     ebp
00401301    mov      ebp, esp
        ; calculate sum
00401303    sub      esp, 0x8
00401306     …
00401320    mov      eax, 0x6
00401328    mov      esp, ebp
0040132A    pop      ebp
0040132B    ret
```

Registers	
EIP	**0040132B**
EBP	**0012F072**
ESP	**0012F044**
EAX	00000006



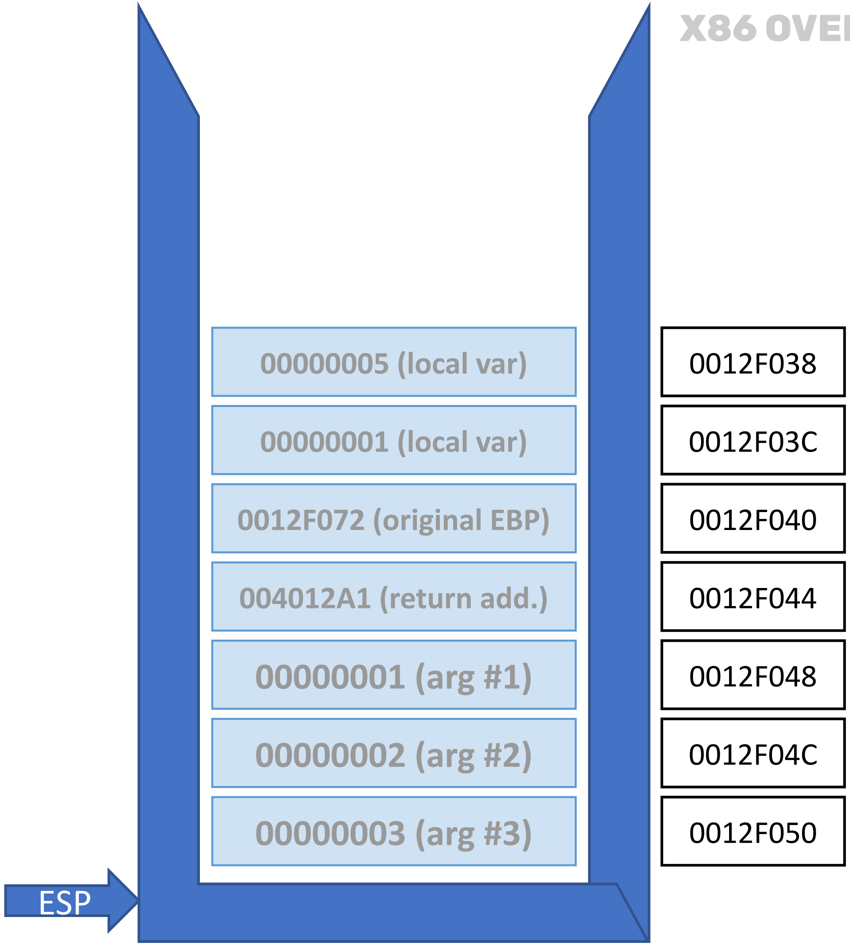
```
caller:
        …
00401296    push      3
00401298    push      2
0040129A    push      1
0040129C    call      sum
004012A1    add       esp, 0xC

sum:
00401300    push      ebp
00401301    mov       ebp, esp
        ; calculate sum
00401303    sub       esp, 0x8
00401306     …
00401320    mov       eax, 0x6
00401328    mov       esp, ebp
0040132A    pop       ebp
0040132B    ret
```

Registers	
EIP	**004012A1**
EBP	0012F072
ESP	**0012F048**
EAX	00000006



```
caller:
        …
00401296    push      3
00401298    push      2
0040129A    push      1
0040129C    call      sum
004012A1    add       esp, 0xC

sum:
00401300    push      ebp
00401301    mov       ebp, esp
        ; calculate sum
00401303    sub       esp, 0x8
00401306     …
00401320    mov       eax, 0x6
00401328    mov       esp, ebp
0040132A    pop       ebp
0040132B    ret
```

Registers	
EIP	**004012A4**
EBP	0012F072
ESP	**0012F054**
EAX	00000006

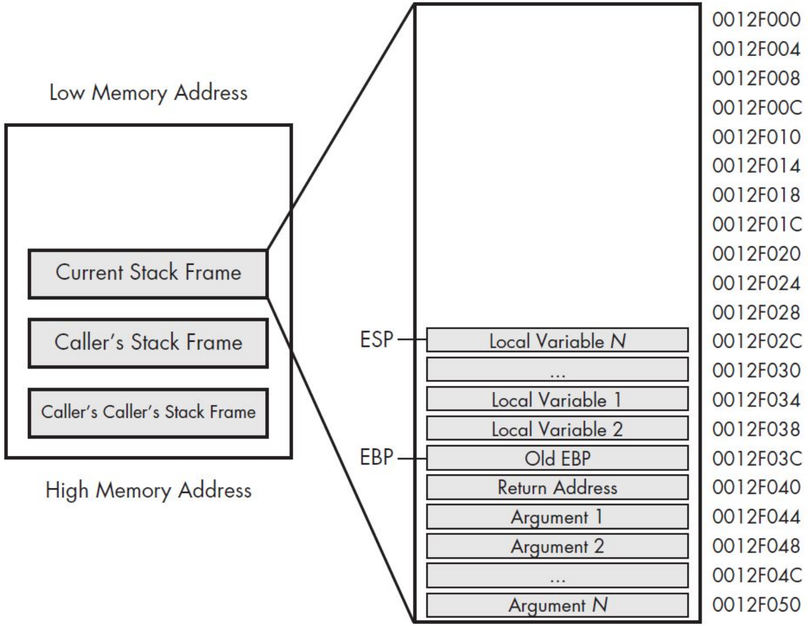


Function Calls - Zoom Out

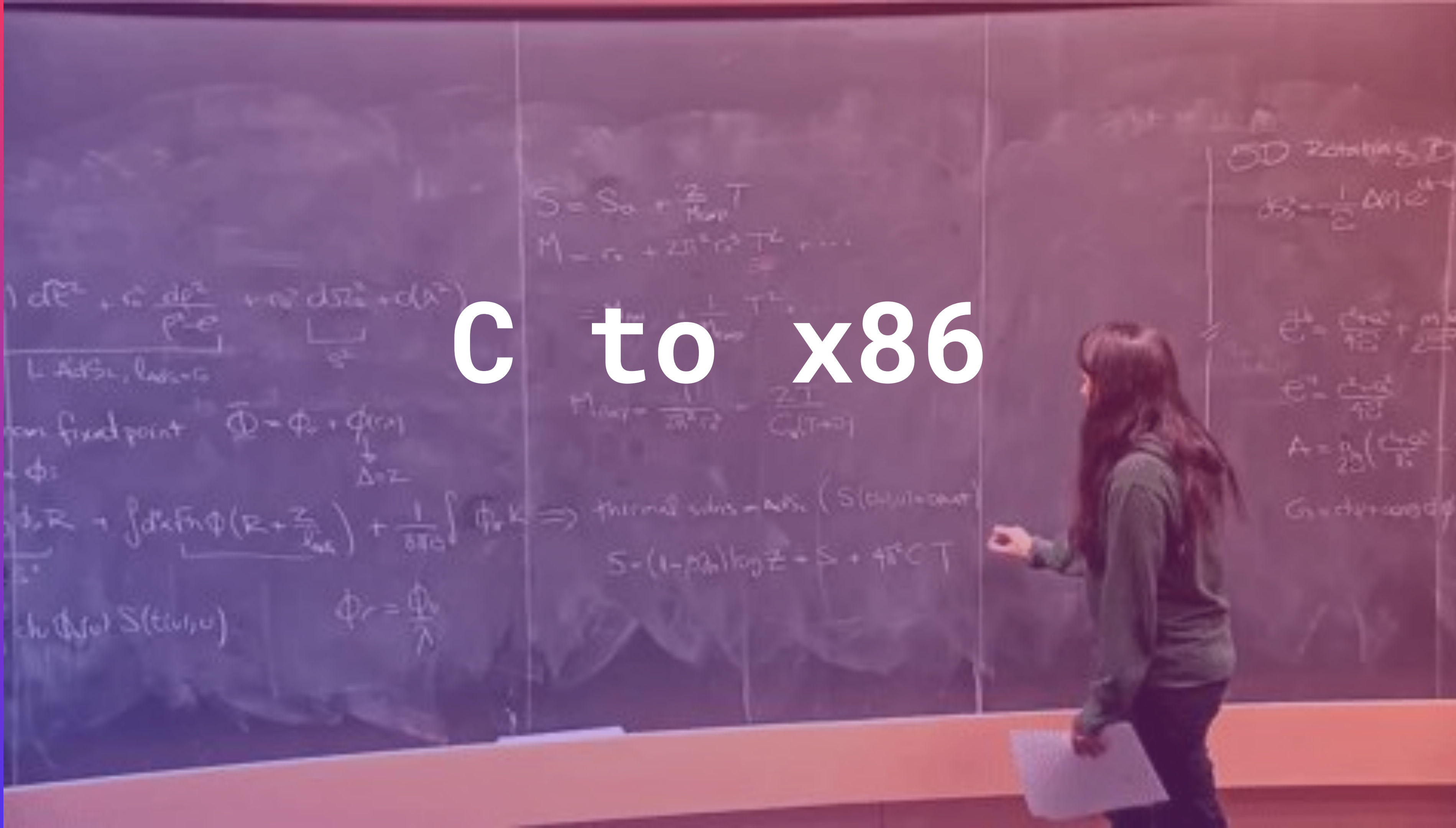
In the previous slides we saw how a single stack frame is generated.

In the illustration on the right we can see multiple stack frames on top of each other, each has its own parameters (arguments), return address, its caller's EBP and local variables.

Now that we know the architecture pretty well, let's see how high-level language is compiled into 0x86 assembly.



C to x86



C to x86

- A C program has two arguments for the main function:

```
int main(int argc, char** argv)
```

For example, a program which is executed from the command line like that:

awesome.exe -r path_to_file.txt

will have the following argc, argv values:

- `argc = 3`
- `argv = [ address of "awesome.exe" string in memory,`
  `address of "-r" string in memory,`
  `address of "path_to_file.txt" string in memory ]`

Remember #1

- `mov [ebp+my_var], some_value`

  What does it mean?

  - my_var is an **offset** (-4, -8, -C…) from EBP to the stack address where my_var is stored (thanks IDA for helping us and renaming it!)
  - The meaning of this expression is "assign the value some_value to my_var"

Remember #2

- function(a, b, c) → push c; push b; push a;

  Function parameters are pushed in **reverse** order.

Remember #3

```
mov eax, [ebp + argv]     ; argv is again an offset from EBP to where argv’s address
                          is stored

mov ecx, [eax + 4 * i]     ; This means “set ecx to equal argv[i]”
```

C to x86

Let's look at the following C program.

What does it do?

1. It verifies that the number of arguments passed to the program is 3 (including the program's path).
2. It checks if the second argument is a flag "-r".
3. If that's the case, it deletes the file provided in the third argument.

```
int main(int argc, char* argv[])
{
        if (argc != 3) {return 0;}

        if (strncmp(argv[1], "-r", 2) == 0){

                DeleteFileA(argv[2]);

        }
        return 0;
}
```

C to x86

First, we compare argc to 3.

We learned that cmp is practically a sub - so 3 is subtracted from the value of argc. If the result is 0, then argc is indeed 3 and we jump to 0x004113DA.

```
int main(int argc, char* argv[])
{
	if (argc != 3) {return 0;}

	if (strncmp(argv[1], "-r", 2) == 0){

		DeleteFileA(argv[2]);

	}
	return 0;
}
```

```
004113CE        cmp     [ebp+argc], 3 ❶
004113D2        jz      short loc_004113DA
004113D4        xor     eax, eax
004113D6        jmp     short loc_411414
```

C to x86

If the result is non zero, then argc is not equal to 3. We zeroize EAX by XORing it with itself and then jump to an address where the function returns (trust me with that).

```
int main(int argc, char* argv[])
{
      if (argc != 3) {return 0;}

      if (strncmp(argv[1], "-r", 2) == 0){

            DeleteFileA(argv[2]);

      }
      return 0;
}
```

```
004113CE        cmp     [ebp+argc], 3 ❶
004113D2        jz      short loc_004113DA
004113D4        xor     eax, eax
004113D6        jmp     short loc_411414
```

C to x86

Next, we can see a call to strncmp. The following parameters are pushed to it:

1. 2, the number of characters to compare
2. a literal string "-r", to which some other string should be compared
3. ECX - what does it hold?

```c
int main(int argc, char* argv[])
{
    if (argc != 3) {return 0;}

    if (strncmp(argv[1], "-r", 2) == 0){

        DeleteFileA(argv[2]);

    }
    return 0;
}
```

```
004113DA        push    2                   ; MaxCount
004113DC        push    offset Str2         ; "-r"
004113E1        mov     eax, [ebp+argv]
004113E4        mov     ecx, [eax+4]
004113E7        push    ecx                 ; Str1
004113E8        call    strncmp ❷
004113F8        test    eax, eax
004113FA        jnz     short loc_411412
```

C to x86

ECX stores whatever is in address eax+4.

EAX, in turn, stores whatever is in address ebp+argv, which is the address to the argv array.

Therefore, ECX = *(argv_address + 4) = argv[1]

```
int main(int argc, char* argv[])
{
    if (argc != 3) {return 0;}

    if (strncmp(argv[1], "-r", 2) == 0){

        DeleteFileA(argv[2]);

    }
    return 0;
}
```

```
004113DA        push    2                ; MaxCount
004113DC        push    offset Str2      ; "-r"
004113E1        mov     eax, [ebp+argv]
004113E4        mov     ecx, [eax+4]
004113E7        push    ecx              ; Str1
004113E8        call    strncmp ❷
004113F8        test    eax, eax
004113FA        jnz     short loc_411412
```

C to x86

Given that parameters are pushed in reverse, the actual call is:

strncmp(argv[1], “-r”, 2)

which is exactly what we see in our C code :)

```
int main(int argc, char* argv[])
{
      if (argc != 3) {return 0;}

      if (strncmp(argv[1], "-r", 2) == 0){

            DeleteFileA(argv[2]);

      }
      return 0;
}
```

```
004113DA    push    2               ; MaxCount
004113DC    push    offset Str2     ; "-r"
004113E1    mov     eax, [ebp+argv]
004113E4    mov     ecx, [eax+4]
004113E7    push    ecx             ; Str1
004113E8    call    strncmp ❷
004113F8    test    eax, eax
004113FA    jnz     short loc_411412
```

C to x86

If the comparison holds (namely, the strings are equal, and the user indeed passed “-r” as the second argument) - EAX will be equal to zero and there will <u>not</u> be a jump. Otherwise - if the strings are <u>not</u> equal - we jump and return zero (trust me once again...)

```
int main(int argc, char* argv[])
{
      if (argc != 3) {return 0;}

      if (strncmp(argv[1], "-r", 2) == 0){

            DeleteFileA(argv[2]);

      }
      return 0;
}
```

```
004113DA        push    2               ; MaxCount
004113DC        push    offset Str2     ; "-r"
004113E1        mov     eax, [ebp+argv]
004113E4        mov     ecx, [eax+4]
004113E7        push    ecx             ; Str1
004113E8        call    strncmp ❷
004113F8        test    eax, eax
004113FA        jnz     short loc_411412
```

C to x86

For the case in which the string are equal, we reach the following code. We push ECX, which (similarly to the previous block) hold *(argv + 8) = argv[2].

Then we call DeleteFileA to delete the file whose path was just pushed.

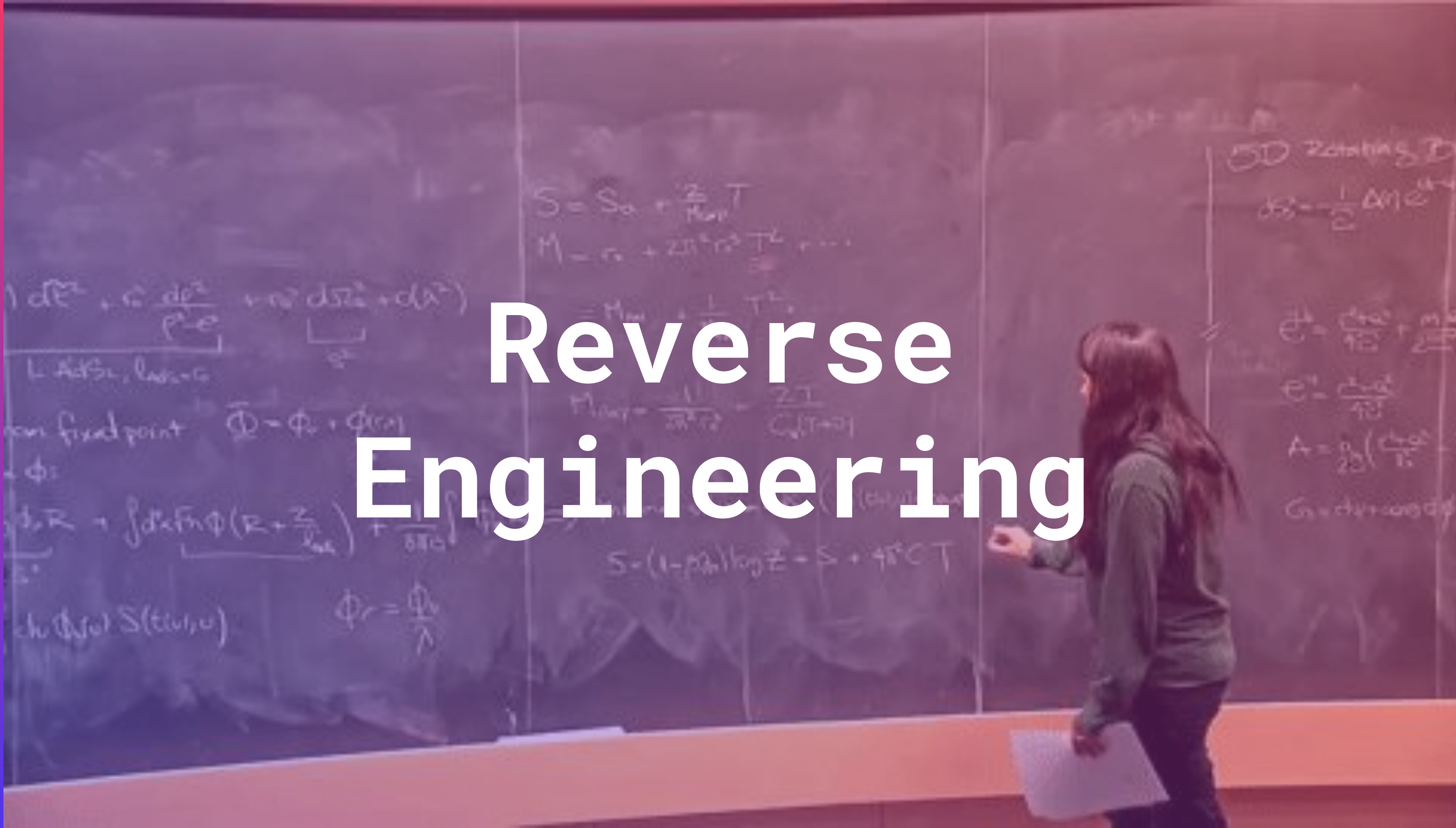
```
int main(int argc, char* argv[])
{
      if (argc != 3) {return 0;}

      if (strncmp(argv[1], "-r", 2) == 0){

            DeleteFileA(argv[2]);

      }
      return 0;
}
```

```
004113FE        mov     eax, [ebp+argv]
00411401        mov     ecx, [eax+8]
00411404        push    ecx             ; lpFileName
00411405        call    DeleteFileA
```

```c
int main(int argc, char* argv[])
{
     if (argc != 3) {return 0;}

     if (strncmp(argv[1], "-r", 2) == 0){

          DeleteFileA(argv[2]);

     }
     return 0;
}
```

```
004113CE          cmp     [ebp+argc], 3 ❶
004113D2          jz      short loc_004113DA
004113D4          xor     eax, eax
004113D6          jmp     short loc_411414

004113DA          push    2                ; MaxCount
004113DC          push    offset Str2      ; "-r"
004113E1          mov     eax, [ebp+argv]
004113E4          mov     ecx, [eax+4]
004113E7          push    ecx              ; Str1
004113E8          call    strncmp ❷
004113F8          test    eax, eax
004113FA          jnz     short loc_411412

004113FE          mov     eax, [ebp+argv]
00411401          mov     ecx, [eax+8]
00411404          push    ecx              ; lpFileName
00411405          call    DeleteFileA
```

Reverse Engineering



- What is Reverse Engineering?
  - Reverse engineering means to take some product and break it down in order to understand how it was produced
  - In hardware: slicing electrical components and analyzing their logical gates
  - In a restaurant: tasting an amazing dish and trying to reproduce it at home

Enigma was also RE'd ;)

REVERSE ENGINEERING

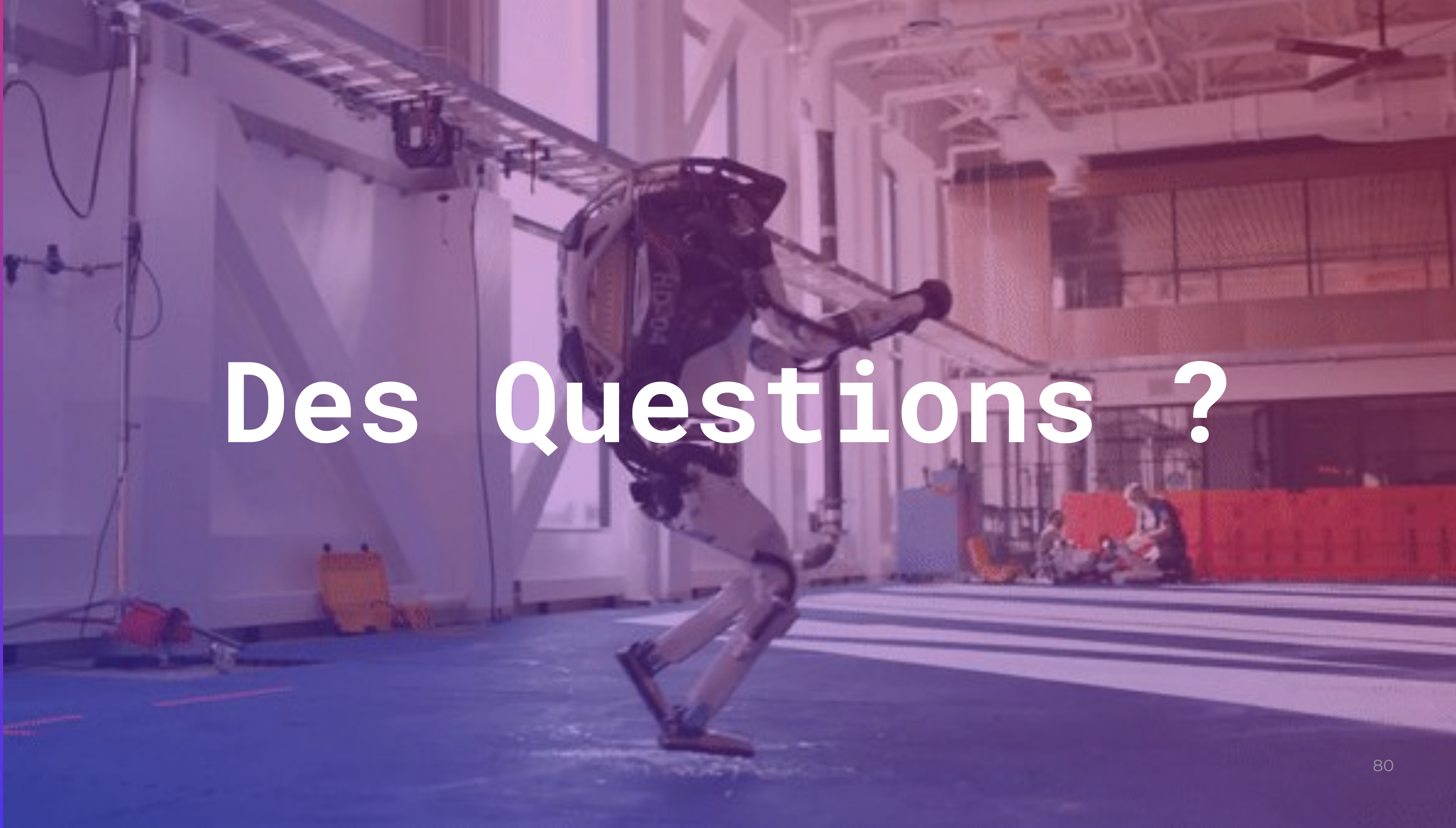
- In **software reverse engineering**:
  - The research object is a program - the machine code of an executable file
  - The goal is to understand what the program does and how

- Why RE ?
  - because sometimes you can’t run the program and you need to analyze it **statically**
  - because ‘basic’ analysis does not provide the whole picture:
    - knowing that a program X uses a function F is one thing, but why is F used? What are its parameters? What does it return?
    - knowing that a program X sends a packet P does not tell us exactly what is in P, how it’s parsed, etc.

Reverse Engineering is **Challenging**.

- Why ?
  - Real-life software is not easy to analyze:
    - it might be **obfuscated**
    - it might be **packed**
    - it will probably be huge
  - But one still has to start somewhere, right? :)

Des Questions ?



Let’s practice

Kahoot Quiz

Link

IDA Introduction

TDs

[Github link]

Three challenges:

- Password
- Good_luck
- Julia

Find the correct password !

TP - Hack Minesweeper

Github link

Your mission: Expose the mines on start-up by printing flags on the mined squares.

Methods:

1. Change the minefield
   Suppose we have a minefield somewhere is the memory of the Minesweeper process. This minefield must contain the information regarding mine locations, right?

   If we manage to somehow change this information, so that a square with a mine will be marked with a flag - we're done

2. Change the printing function
   As Minesweeper is a GUI application, it contains graphics and uses a function which prints those graphics to the screen.

   If we manage to find this function and pass it the argument corresponding to flag whenever there is a mine - we're done.

Original workshop

www.begin.re