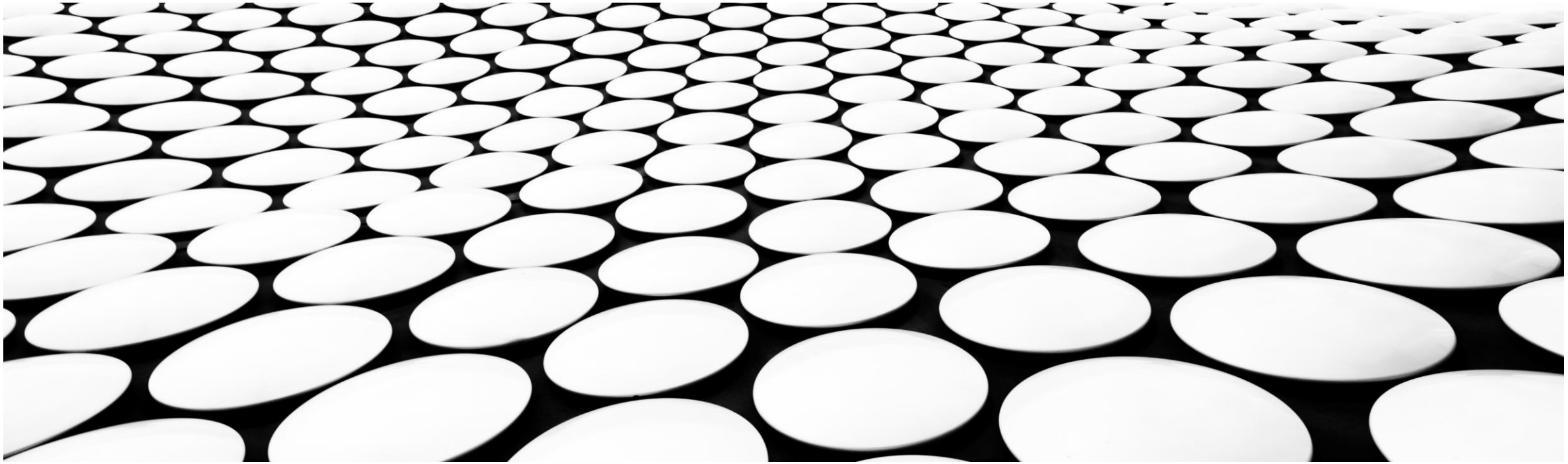
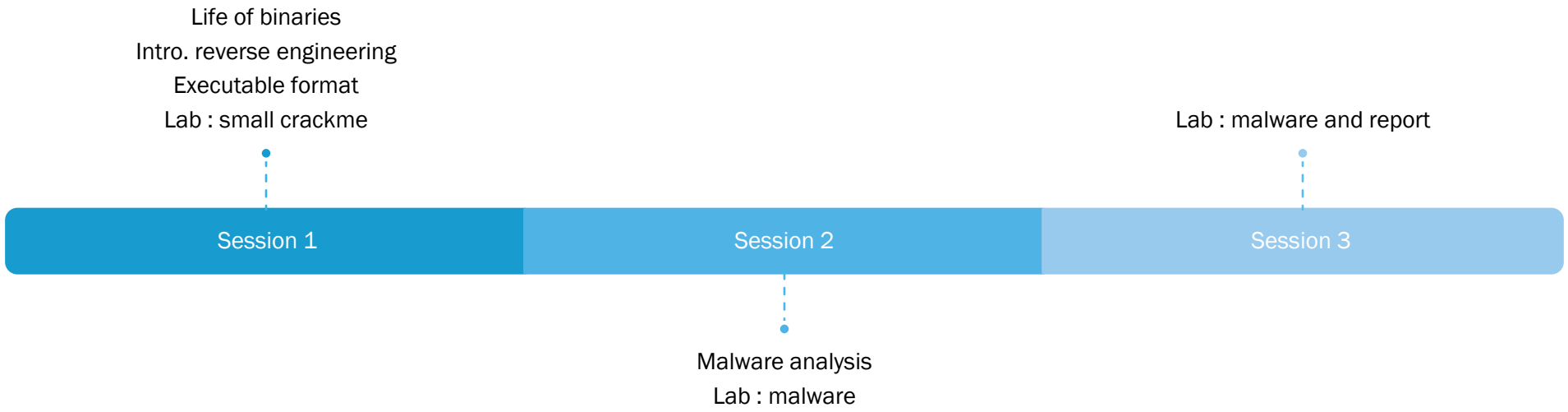
FORENSICS
REVERSE ENGINEERING
MALWARE ANALYSIS

ECE - 2025



AGENDA

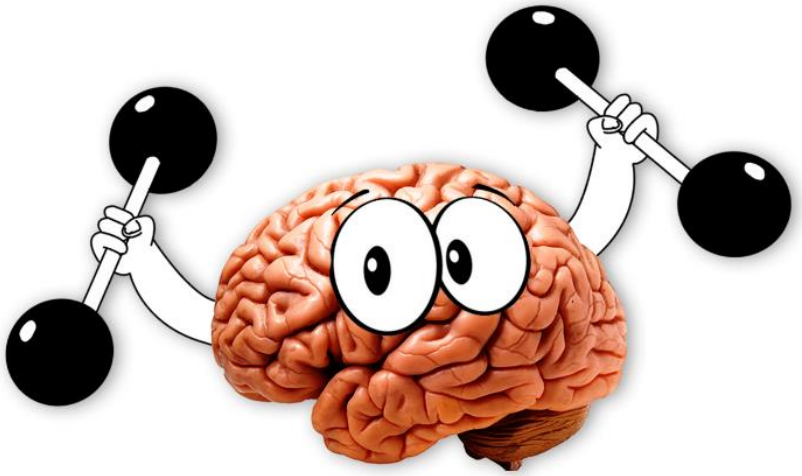

Life of binaries
Intro. reverse engineering
Executable format
Lab : small crackme
Lab : malware and report
Session 1
Session 2
Session 3
Malware analysis
Lab : malware

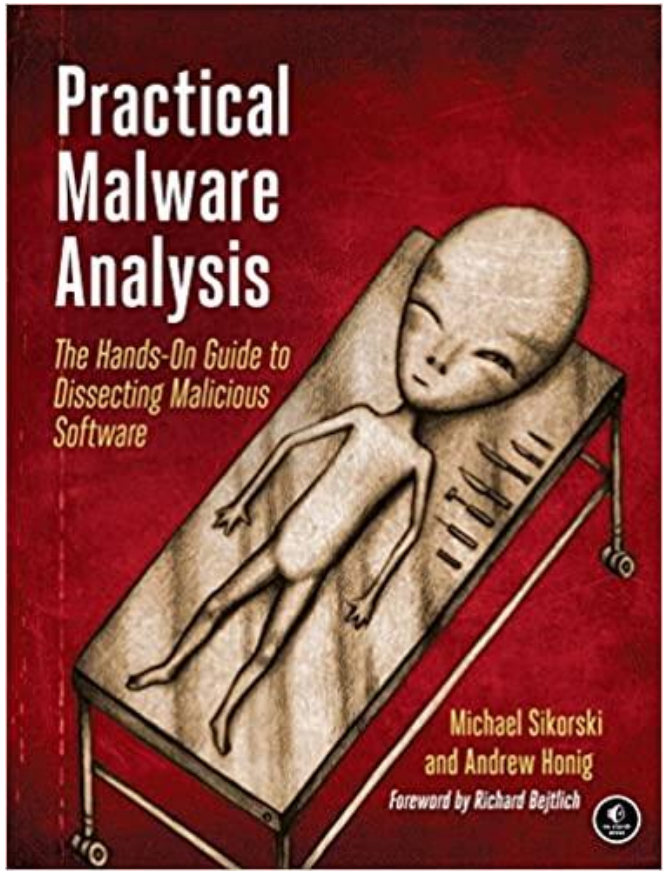
INTRODUCTION

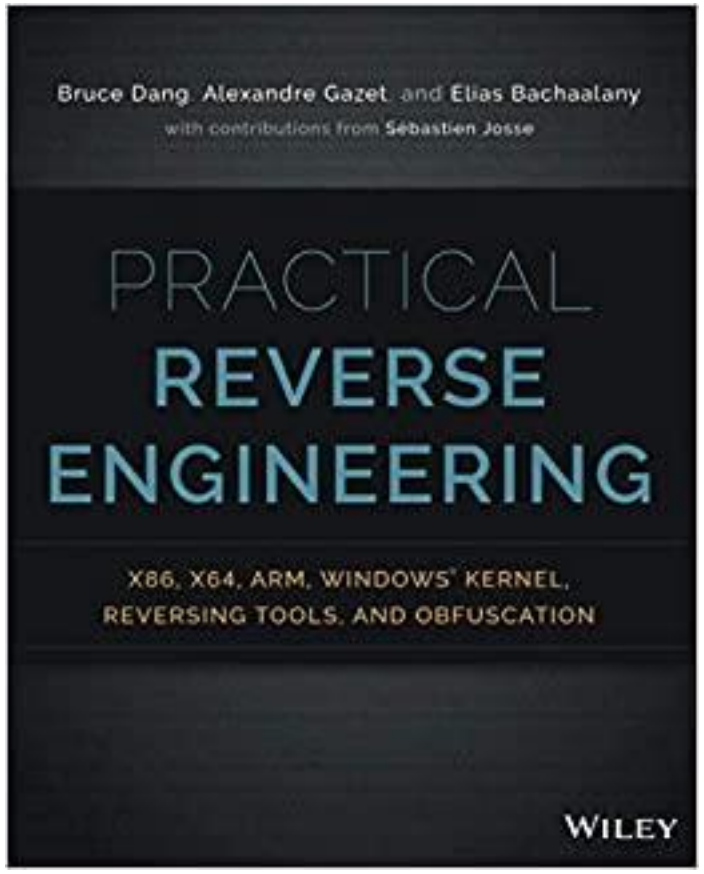
- Evaluation
  - Lab reports - 3 people max (taken into account during correction) → I need the groups by the end of the first session!
  - Final – 1 person (DS)

- Reverse engineering *report*
  - The approach & **methodology**
  - The problems &the findings

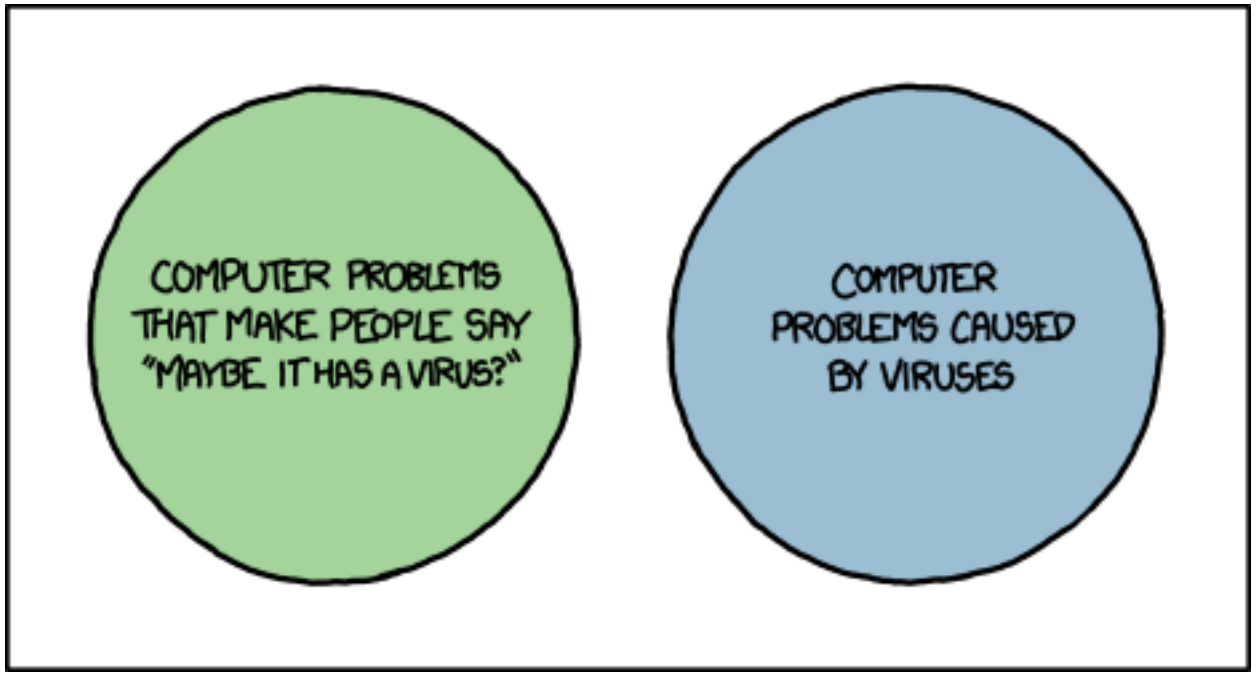
- Deadline – To be defined!
  - **Report on every lab**
  - **Everything on campus…**

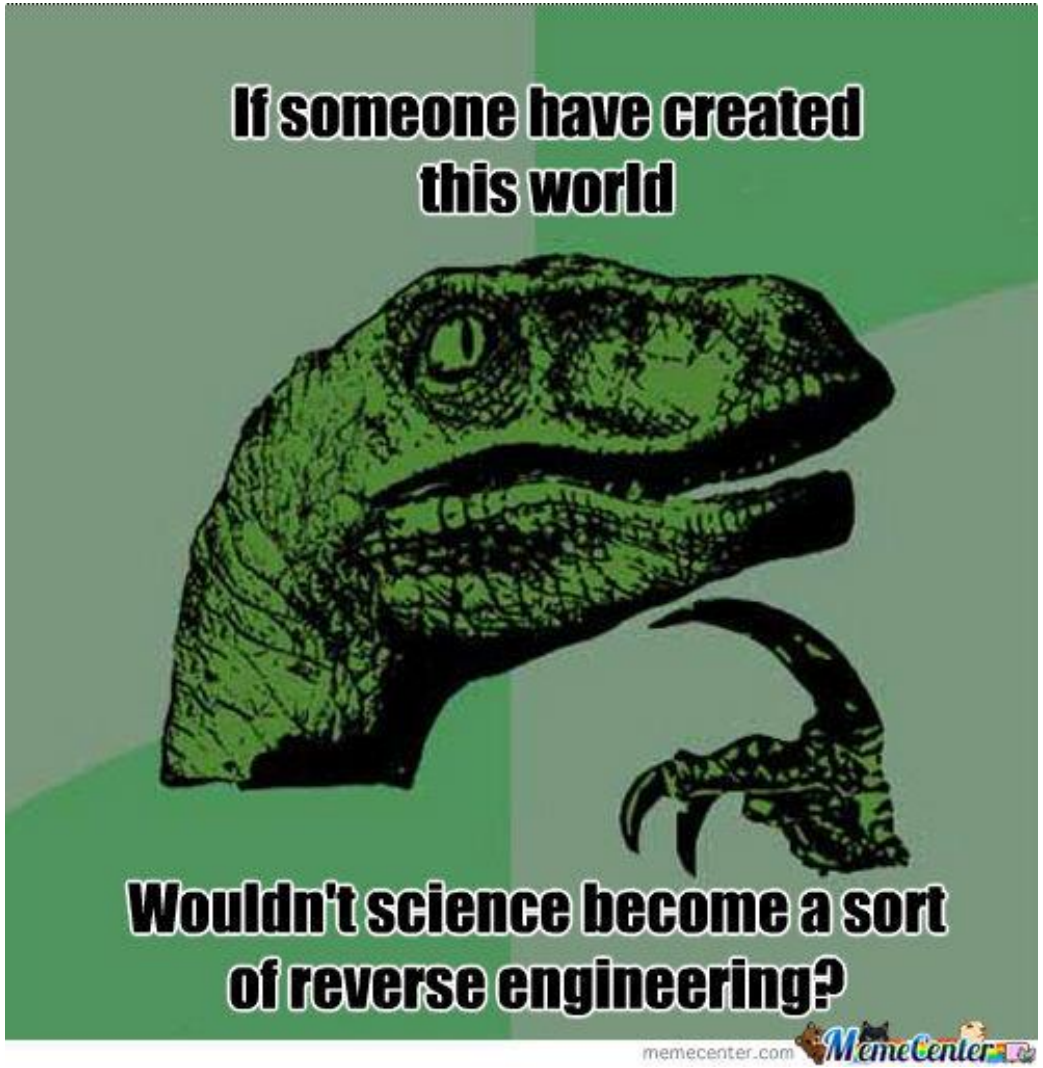



Practical
Malware
Analysis
The Hands-On Guide to
Dissecting Malicious
Software
Michael Sikorski
and Andrew Honig
Foreword by Richard Bejtlich


Bruce Dang, Alexandre Gazet, and Elias Bachaalany
with contributions from Sébastien Josse
PRACTICAL
REVERSE
ENGINEERING
X86, X64, ARM, WINDOWS' KERNEL,
REVERSING TOOLS, AND OBFUSCATION
WILEY

INCIDENT RESPONSE ?

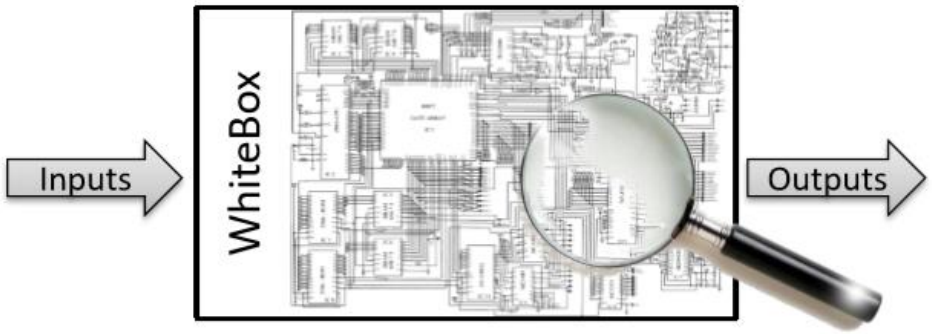


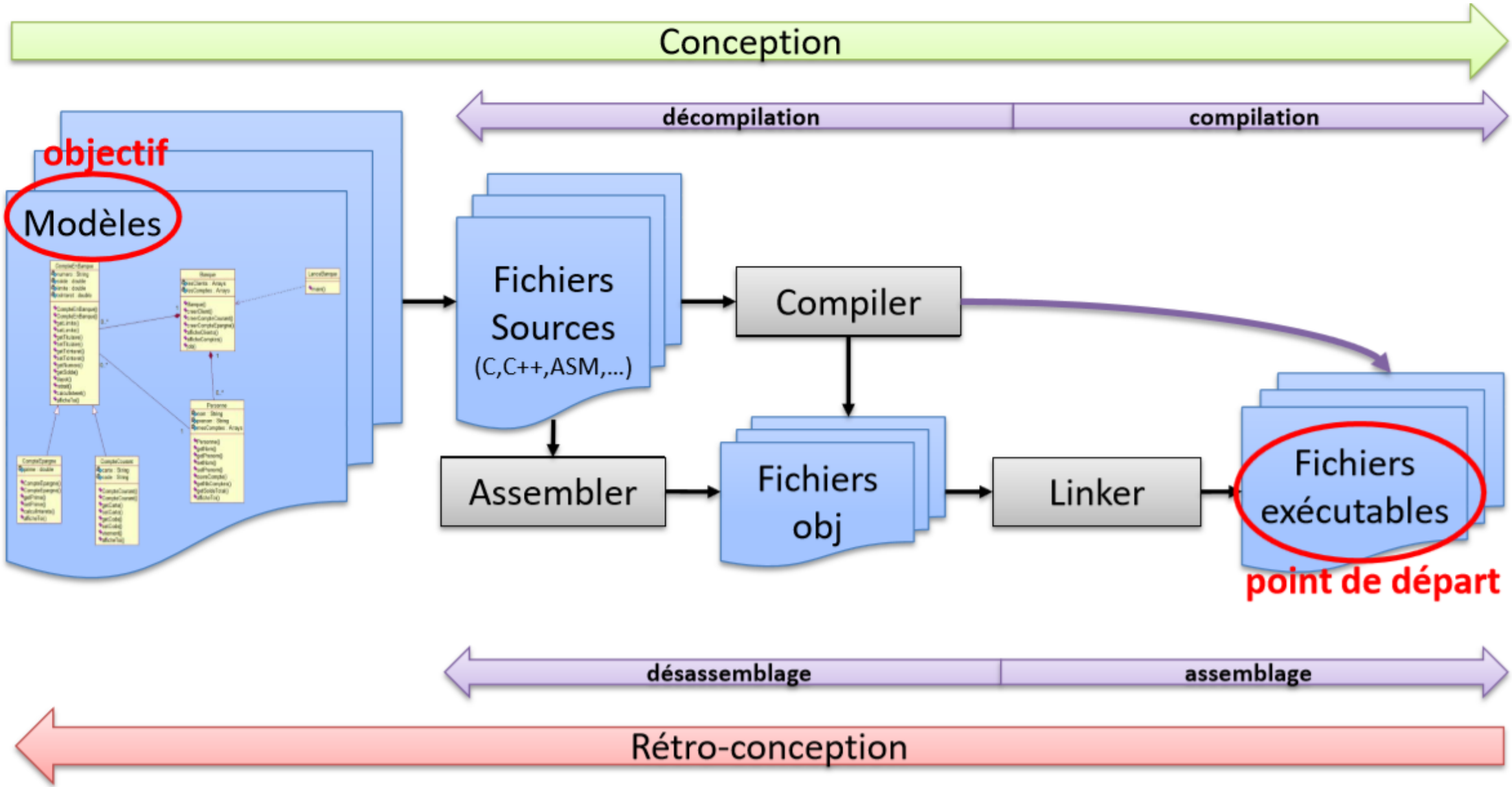

If someone have created this world
Wouldn't science become a sort of reverse engineering?
memecenter.com
MemeCenter

REVERSE ENGINEERING

- Black box
  - Only access to input and output of a program
    - Eg. f(x) = y ➔ access to x and y

- White box
  - Access to source files of the program
  - **Access to binary of the program ➔ our context !**




Conception
décompilation
compilation
objectif
Modèles
Fichiers Sources
(C,C++,ASM,...)
Compiler
Assembler
Fichiers obj
Linker
Fichiers exécutables
point de départ
désassemblage
assemblage
Rétro-conception

```
; int __fastcall sub_401230(UINT uValue)
sub_401230      proc near
                push    ebx
                push    esi
                mov     eax, ecx
                mov     esi, 2
                xor     ebx, ebx
                mov     ecx, 1

loc_401240:
                xor     edx, edx
                div     esi
                test    edx, edx
                jz      short loc_40124A
                or      ebx, ecx

loc_40124A:
                shl     ecx, 1
                jnb     short loc_401240
                mov     eax, ebx
                pop     esi
                pop     ebx
                retn
sub_401230      endp
```

```
int __fastcall sub_401230(unsigned int a1)
{
  unsigned int v1; // eax@1
  int v2; // ebx@1
  signed int v3; // ecx@1
  unsigned int v4; // ett@2
  unsigned __int8 v5; // cf@4

  v1 = a1;
  v2 = 0;
  v3 = 1;
  do
  {
    v4 = v1;
    v1 /= 2u;
    if ( v4 % 2 )
      v2 |= v3;
    v5 = __CFSHL__(v3, 1);
    v3 *= 2;
  }
  while ( !v5 );
  return v2;                                    return a1;
}
```

BABY LAB

1. Install IDA (hexrays)
2. Prepare a small C program that adds 2 number from user input
3. Build it
4. Open it in IDA

REVERSE ENGINEERING

- Compilation
  - Loss of information
    - Symbols
    - Distinction between code and data
    - Type
    - Control flow high level (for, while, do, switch, etc.)

- Assembly
  - Depends on the behaviour of the compilation
    - Call convention
    - Usage of registers
    - Inline functions

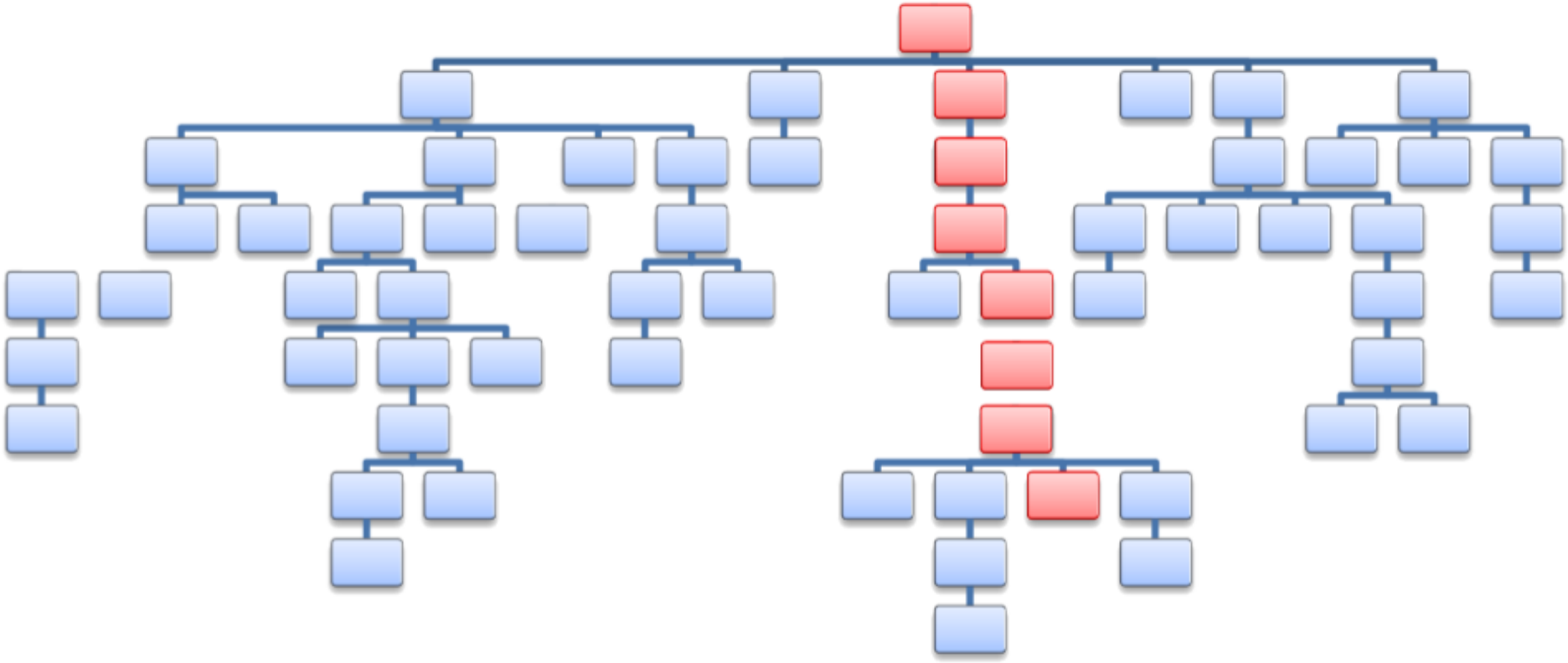
STRATEGY?
TACTICS?

STRATEGY

- Static analysis
  - Recover information from a software *without* its execution
  - *Multipath* analysis
  - To build a *Model* (MVC pattern)

- Dynamic analysis
  - Recover information from a software *through* its execution
  - *Unique path* analysis
  - To build an *Instance* of the Model (MVC pattern)

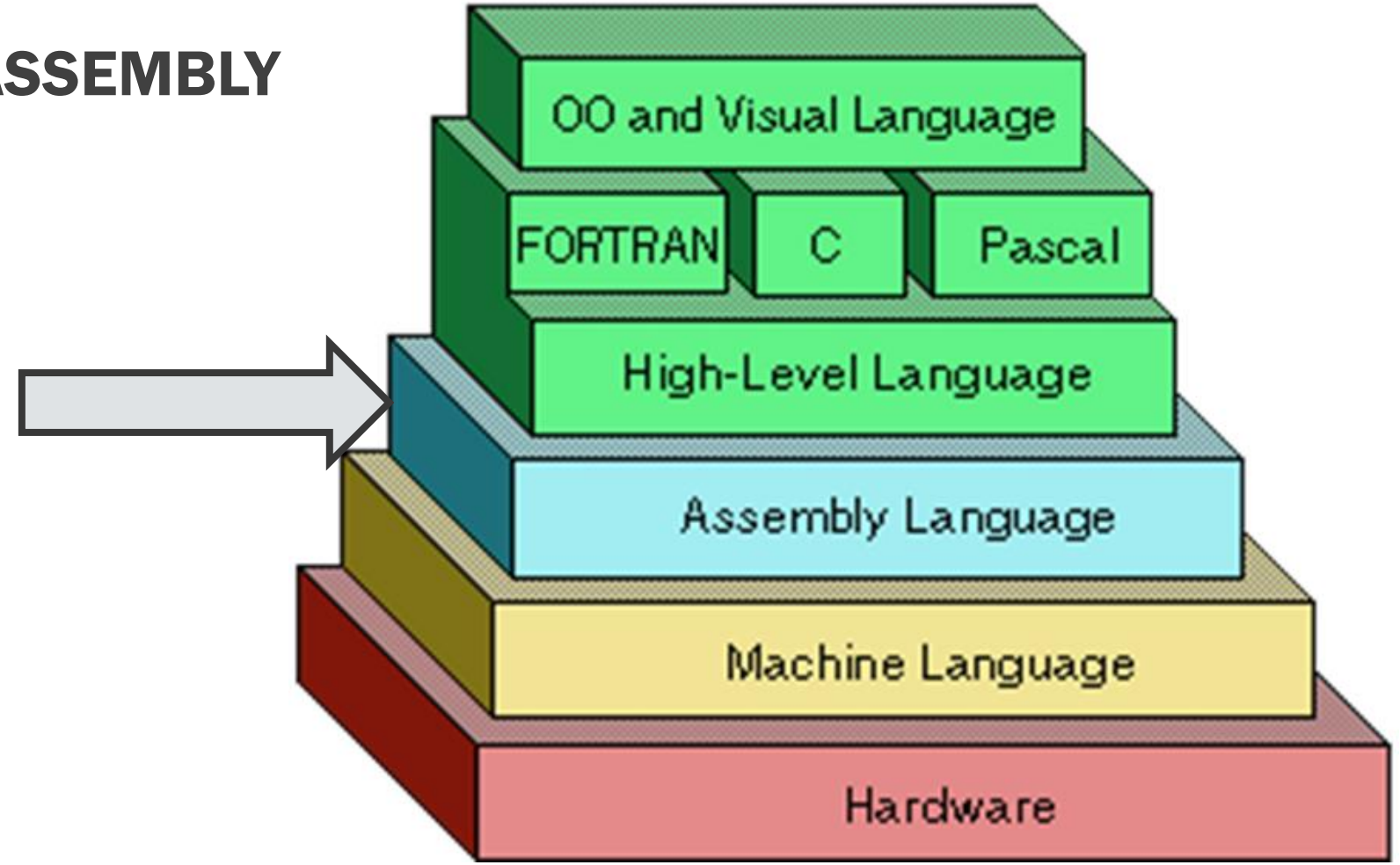
STRATEGY?



STRATEGY

- Pros and cons
  - Static analysis
    - Disassemble -> Propagate data -> Propagate types -> Detect signatures
    - The tools can do a lot ***BUT*** the “manual” part remains big
  - Dynamic analysis
    - Can determine the exact behavior of a program with a given input
    - The analyst does not know the remaining work to do (one path)
  - <u>The two are complementary</u>

ASSEMBLY


OO and Visual Language
FORTRAN
C
Pascal
High-Level Language
Assembly Language
Machine Language
Hardware

INSTRUCTIONS & OPERANDS

- Each instruction corresponds to opcodes that tell the CPU what operation to perform
  - ex. mov ecx, 0x42 → B9 42 00 00 00
- Operands are used to identify the data used by an instruction
  - Immediate operands - mov eax, 0x42
    - Fixed values, such as 0x42
  - Register operands - mov eax, ecx
    - Registers, such as ecx
  - Memory Addresses - mov eax, [eax]
    - Denoted by brackets, [eax]

REGISTERS

- General Registers
  - Used by the CPU during execution
- Status Flags
  - Used to make decisions
- Instruction Pointer (eip)
  - Used to keep track of the next instruction

ASSEMBLY LANGUAGE (INTEL X86) - GENERAL REGISTERS

- The general registers of 32bits **EAX**, **EBX**, **ECX**, **EDX**, **ESI**, **EDI**, **EBP** and **ESP** :
  - Operands for logical and arithmetical operations ;
  - Operands for addresses calculation ;
  - Pointers.

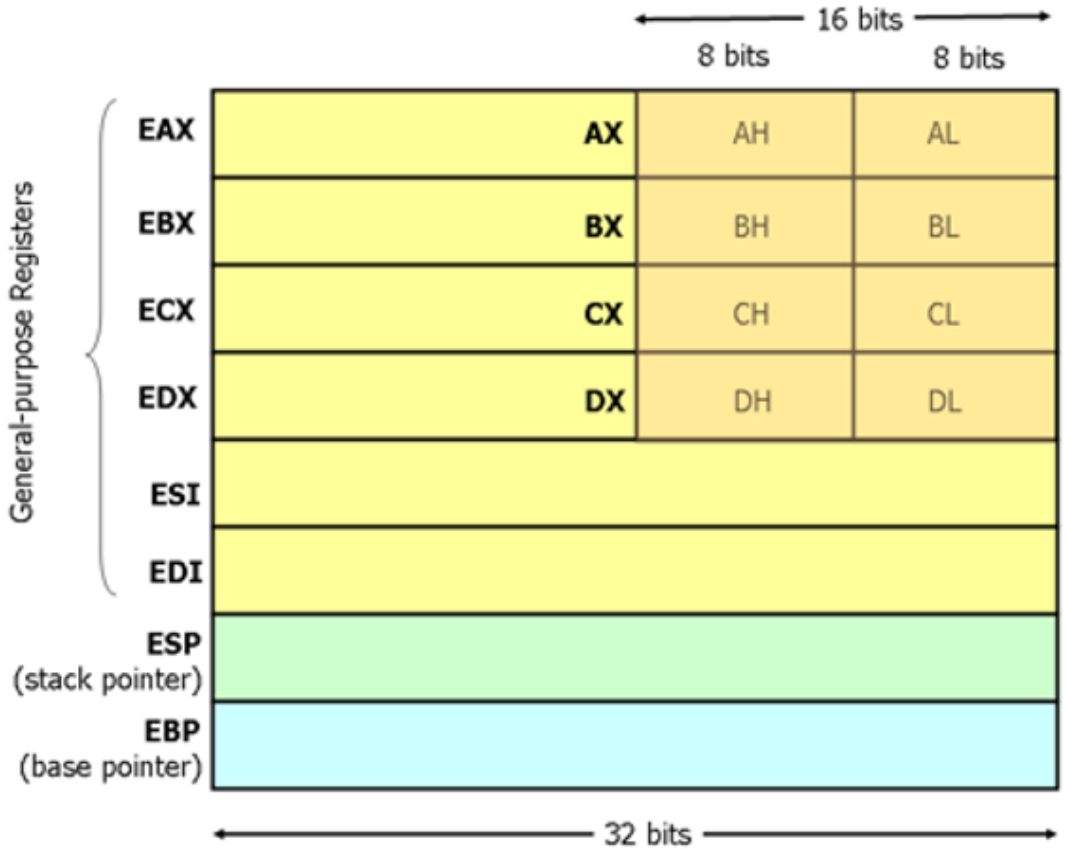
- ESP maintains the stack pointer
- String related instructions use the content of ECX, ESI and EDI registers as operands.

ASSEMBLY LANGUAGE (INTEL X86)

- The main registers on x86 architecture
  - **EAX** (*Accumulator register*) : It is used for I/O port access, arithmetic, interrupt calls,.
  - **EBX** (*Base register*) : It is used as a base pointer for memory access.
  - **ECX** (*Counter register*) : It is used as a loop counter and for shifts.
  - **EDX** (*Data register*) : It is used for I/O port access, arithmetic, some interrupt .
  - **ESI** (*Source index register*) : Used for string and memory array copying.
  - **EDI** (*Destination index register*) : Used for string, memory array copying.
  - **ESP** (*Stack pointer register*) : Holds the top address of the stack.
  - **EBP** (*Stack Base pointer register*) : Holds the base address of the stack.

- *It is a convention, not a reality*
  - It depends on the compiler!

ASSEMBLY LANGUAGE (INTEL X86)



ASSEMBLY LANGUAGE (INTEL X86) - FLAGS REGISTER (EFLAGS)

- The **EFLAGS** register is a 32bits register containing a set of state flags, a control flag and a set of system flags.
- The state flags are modified by specific instructions: **arithmetical**, **logical** and **comparison** instructions.
- Conditional jump instructions **test the state of these flags** to make a branch or not:
  - CF (Carry flag – bit 0) : This bit is set to 1 if an operation causes a carry.
  - PF (Parity flag – bit 2) : This bit is set to 1 if the result of an arithmetical operation is composed by an even number of bit set to 1.
  - ZF (Zero flag – bit 6) : This bit, when set to 1, indicates that the result of the last arithmetical operation has resulted on 0.
  - SF (Sign flag – bit 7) : In signed arithmetic, this bit indicates that the result of an instruction gives a negative number (most significant bit set to 1).

ASSEMBLY LANGUAGE (INTEL X86)

- Main assembly instructions
  - Data manipulation
    - MOV, PUSH, POP, PUSHA/PUSHAD, POPA/POPAD, CWD/CDQ, MOVSX, MOVZX
  - Binary arithmetic
    - ADD, ADC, SUB, SBB, IMUL, MUL, IDIV, DIV, INC, DEC, NEG, CMP
  - Decimal arithmetic
    - DAA, DAS, AAA, AAS, AAM, AAD
  - Logical arithmetic
    - AND, OR, XOR, NOT

ASSEMBLY LANGUAGE (INTEL X86)

- Main assembly instructions
  - Shift and rotation
    - SAR, SHR, SAL/SHL, SHRD, SHLD, ROR, ROL, RCR, RCL
  - Flow control
    - JMP, JE/JZ, JNE/JNZ, JA/JNBE, JAE/JNB, JB/JNAE, JBE/JNA, JG/JNLE, JGE/JNL, JL/JNGE, JLE/JNG, JC, JNC, JO, JNO, JS, JNS, JPO/JNP, JPE/JP, JCXZ/JECXZ, LOOP, LOOPZ/LOOPE, LOOPNZ/LOOPNE, CALL, RET, IRET, INT, INTO, BOUND, ENTER, LEAVE
  - Misc. instructions
    - LEA, NOP, XLAT/XLATB, CPUID, MOVBE

ASSEMBLY LANGUAGE (INTEL X86)

- Example

MOV EAX, 1	EAX = 1
ADD EDX, 5	EDX = EDX + 5
SUB EBX, 2	EBX = EBX - 2
AND ECX, 0	ECX = ECX & 0
XOR EDX, 4	EDX = EDX ^ 4
INC ECX	ECX++

- lea eax, [ebx+8]
- mov eax, [ebx]

MOV

- Mov instruction
  - it… moves things…
  - mov destination, source

  - mov eax, [ebx+8] puts the value at ebx+8 into eax
    - mov eax, ebx+8 is the same as lea eax, [ebx+8], but is invalid

LEA

- lea - Load Effective Address
  - lea destination, source
  - lea eax, [ebx+8] puts ebx+8 into eax

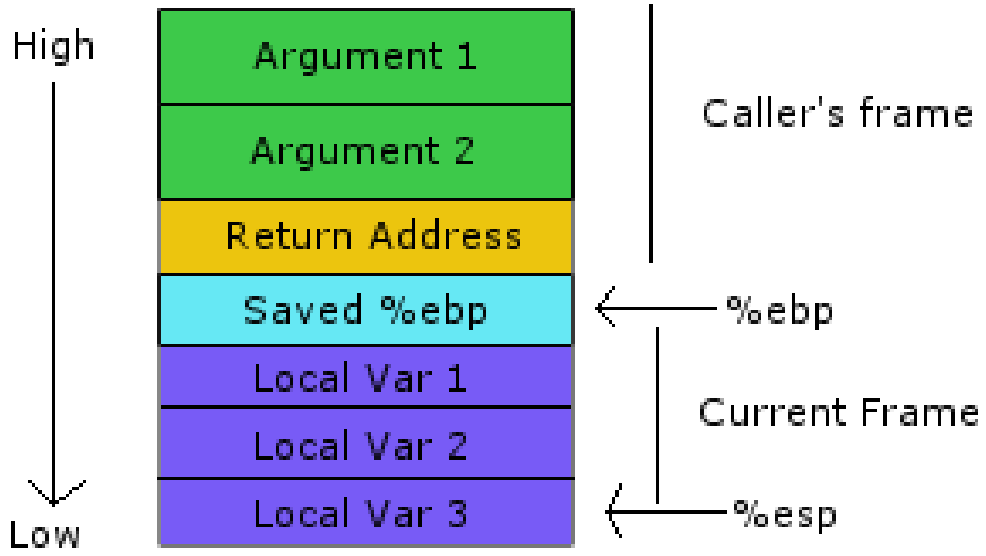
MEMORY ALLOCATIONS

- Allocation on the heap
  - malloc
- Allocation on the stack
  - The stack is a data structure based on the "last in, first out" principle (LIFO or Last In, First Out)
  - In the x86 32bit, **the ESP register contains the top of the stack address**.

  - "**push**" and "**pop**" opcodes allows to stack and unstack of data.
  - "**call**" and "**ret**" opcodes use the stack to call functions and leave later.

  - On this architecture the stack **grows down** (ESP ≤ EBP).

MEMORY ALLOCATIONS



CALLING CONVENTIONS ?

- Example with the following code :

```
int substract( int a, int b, int c )
{
    return a - b - c;
}

int main( int argc, char **argv )
{
    int nb1, nb2, nb3, resultat;
    [...]
    resul = substract(nb1, nb2, nb3);
    [...]
}
```

CALLING CONVENTIONS

- __cdecl
  - It is the convention that supports the C-language syntax and, particularly, the existence of function with a variable number of args (e.g. printf()).
  - This convention is the standard.
  - The cleaning of the stack is executed by the caller
  - The args are pushed from right to the left to the stack
  - Header of the function « substract » with Microsoft Visual C++ :
    - int __cdecl substract (int a, int b, int c);

CALLING CONVENTIONS

```
substract:
    push ebp
    mov  ebp, esp

    mov  eax, [ebp + 8]  ; a
    sub  eax, [ebp + 12] ; a = a - b
    sub  eax, [ebp + 16] ; a = a - c

    leave
    ret
```

- Example: « __cdecl » in assembly

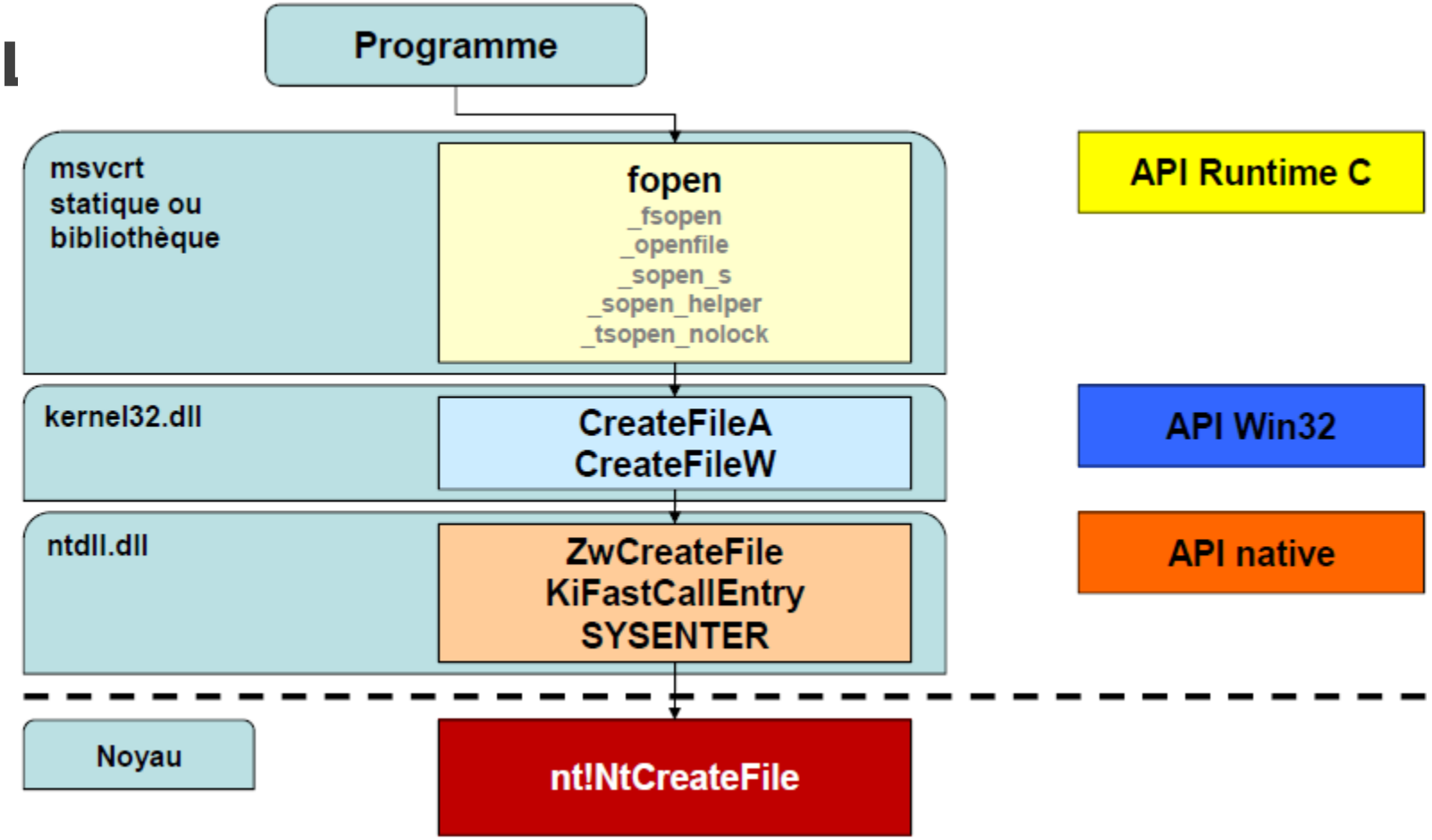
```
    ....
   ; Pass the args from the right to the left
   ; First arg at the top of the stack
    push dword [ebp - 12] ; param c
    push dword [ebp - 8]  ; param b
    push dword [ebp - 4]  ; param a

    call substract
    ; The caller cleans the stack
    ; (here 12 bytes : 3 int * 4 bytes)
```

CALLING CONVENTIONS

- SUPER IMPORTANT
  - Within a given function's stack frame:
    - Local variables will be at a negative offset from ebp
    - Function Arguments will be at a positive offset

USER CALL


Programme
msvcrt
statique ou
bibliothèque
fopen
_fsopen
_openfile
_sopen_s
_sopen_helper
_tsopen_nolock
kernel32.dll
CreateFileA
CreateFileW
ntdll.dll
ZwCreateFile
KiFastCallEntry
SYSENTER
Noyau
nt!NtCreateFile
API Runtime C
API Win32
API native

PE FORMAT

- Portable Executable
  - Export
    - A binary can export symbols related to functions or variables.
    - **IMAGE_EXPORT_DIRECTORY**
  - Import
    - It is possible to import symbols (variables or functions) from external libraires.
    - **IMAGE_IMPORT_DIRECTORY**

BINARY ANALYSIS?

- Suspicious binary analysis
  - What components or elements

BINARY ANALYSIS?

- Suspicious binary analysis
  - Quick analysis (triage)
    - Nature (ELF, PE, PE64, Kernel components)
    - Strings : piece of URL, IP address, registry keys, filenames, ...
    - Imports : external functions used (network, encryption, ...)
  - Extensive qualification
    - Reverse engineering
  - IOC definition
    - Set of characteristic elements of a threat.

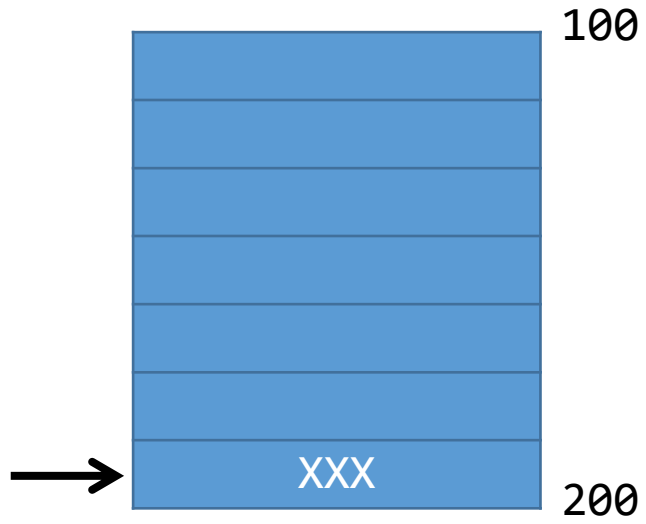
SMALL CPU SIMULATION

WARM UP

```
@0: push 10
@1: push 20
@2: call add
@3: ….
```

```
add:
@20: push ebp
@21: mov ebp, esp
@22: sub esp, 8
@23: mov eax, [ebp+8]
@24: add eax, [ebp+12]
@25: mov esp, ebp
@26: pop ebp
@27: ret
```

```
eip : 0
esp : 200
```

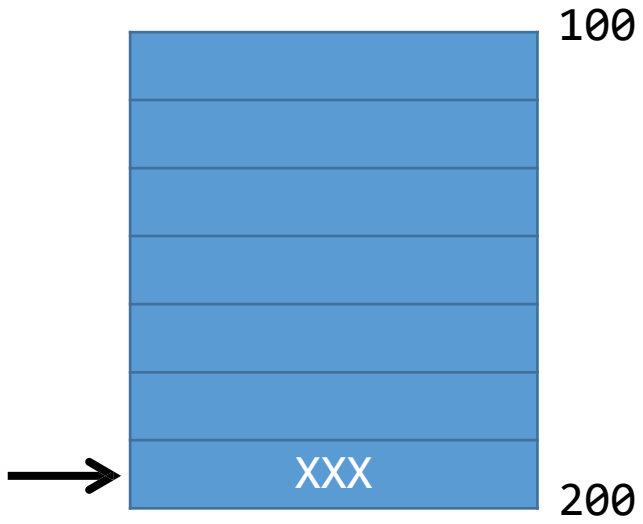


```
@0: push 10
@1: push 20
@2: call add
@3: ….



add:
@20: push ebp
@21: mov ebp, esp
@22: sub esp, 8
@23: mov eax, [ebp+8]
@24: add eax, [ebp+12]
@25: mov esp, ebp
@26: pop ebp
@27: ret
```

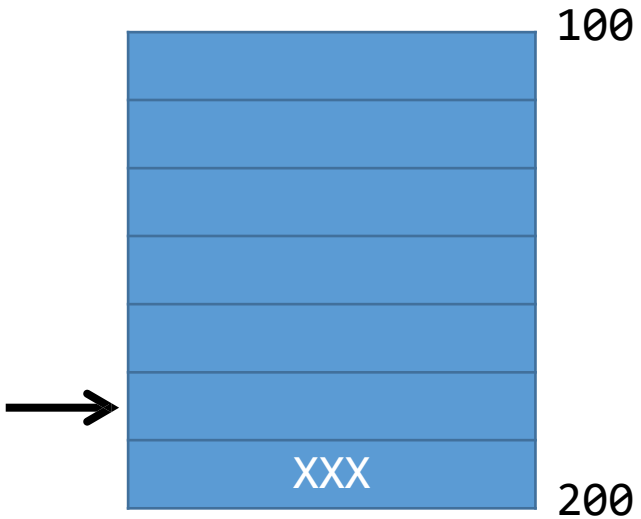
```
eip : 0
esp : 200
```



```
@0: push 10
@1: push 20
@2: call add
@3: ….
```

```
add:
@20: push ebp
@21: mov ebp, esp
@22: sub esp, 8
@23: mov eax, [ebp+8]
@24: add eax, [ebp+12]
@25: mov esp, ebp
@26: pop ebp
@27: ret
```

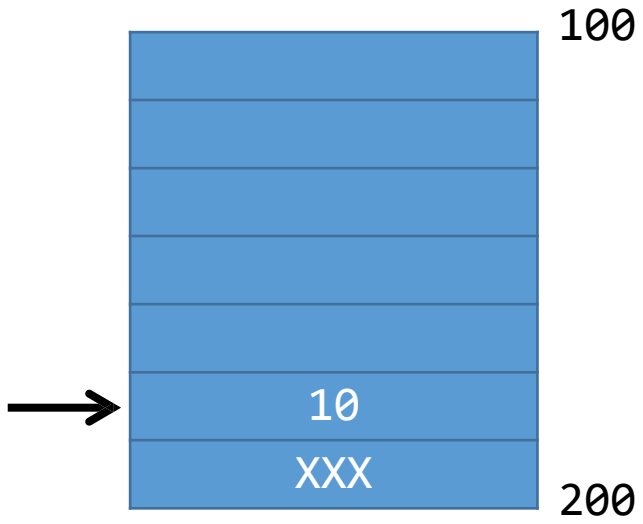
```
eip : 0
esp : 196
```



```
@0: push 10
@1: push 20
@2: call add
@3: ….
```

```
add:
@20: push ebp
@21: mov ebp, esp
@22: sub esp, 8
@23: mov eax, [ebp+8]
@24: add eax, [ebp+12]
@25: mov esp, ebp
@26: pop ebp
@27: ret
```

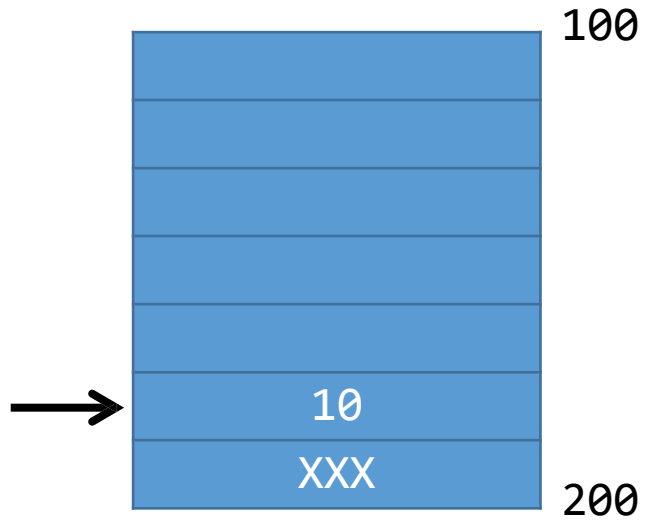
```
eip : 0
esp : 196
```



```
@0: push 10
@1: push 20
@2: call add
@3: ….
```

```
add:
@20: push ebp
@21: mov ebp, esp
@22: sub esp, 8
@23: mov eax, [ebp+8]
@24: add eax, [ebp+12]
@25: mov esp, ebp
@26: pop ebp
@27: ret
```

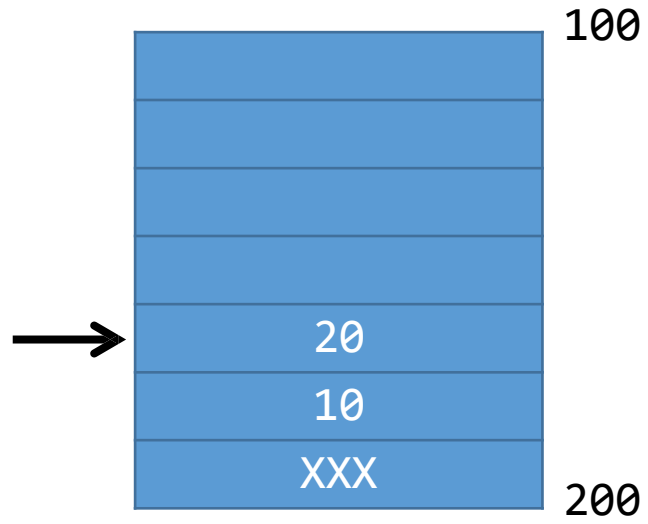
```
eip : 1
esp : 196
```



```
@0: push 10
@1: push 20
@2: call add
@3: ….
```

```
add:
@20: push ebp
@21: mov ebp, esp
@22: sub esp, 8
@23: mov eax, [ebp+8]
@24: add eax, [ebp+12]
@25: mov esp, ebp
@26: pop ebp
@27: ret
```

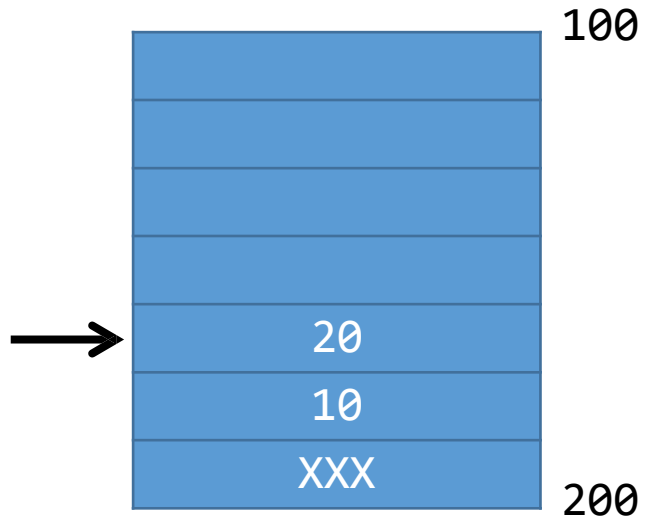
```
eip : 2
esp : 192
```

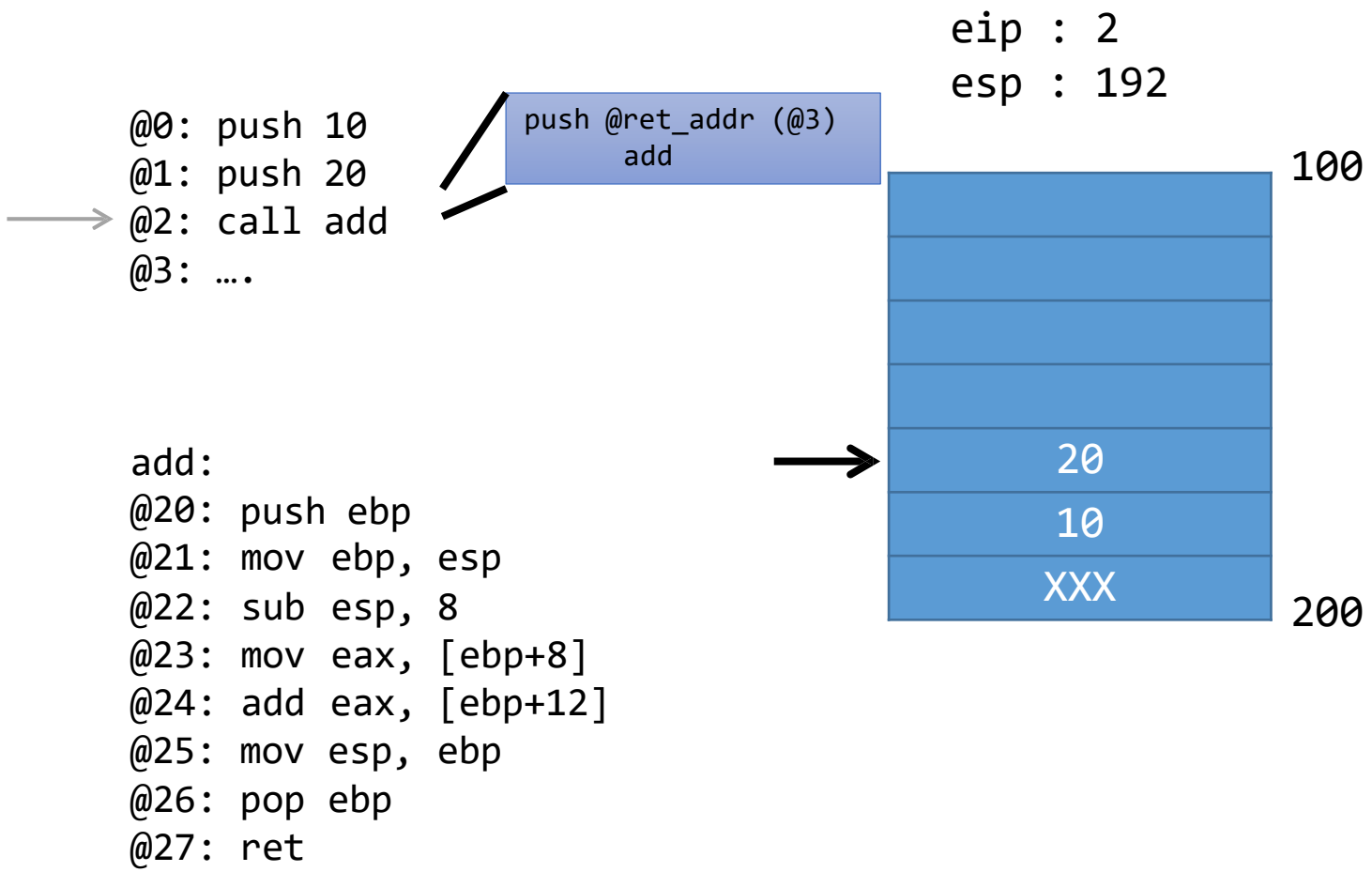


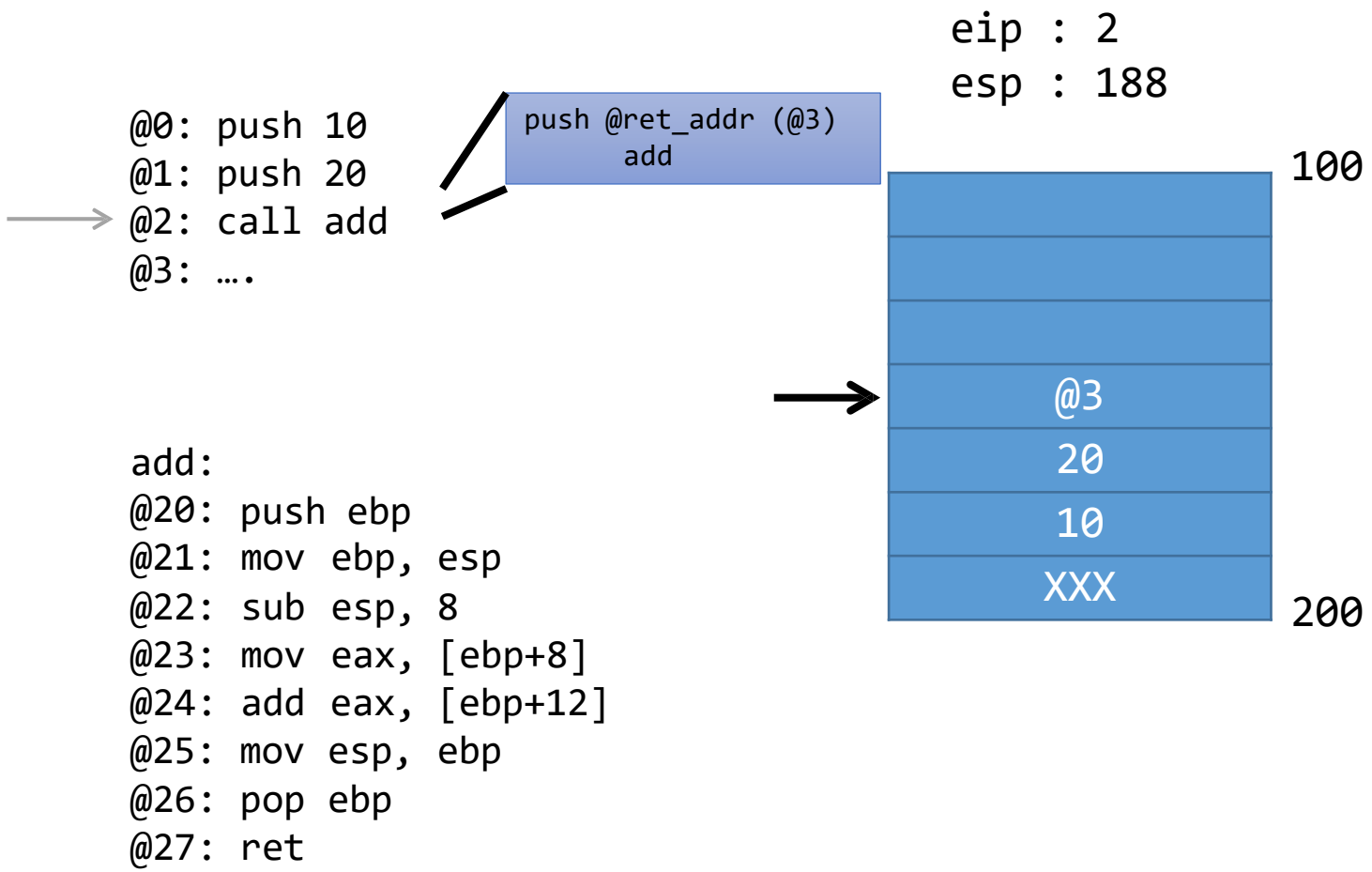
```
@0: push 10
@1: push 20
@2: call add
@3: ….
```

```
add:
@20: push ebp
@21: mov ebp, esp
@22: sub esp, 8
@23: mov eax, [ebp+8]
@24: add eax, [ebp+12]
@25: mov esp, ebp
@26: pop ebp
@27: ret
```

```
eip : 2
esp : 192
```

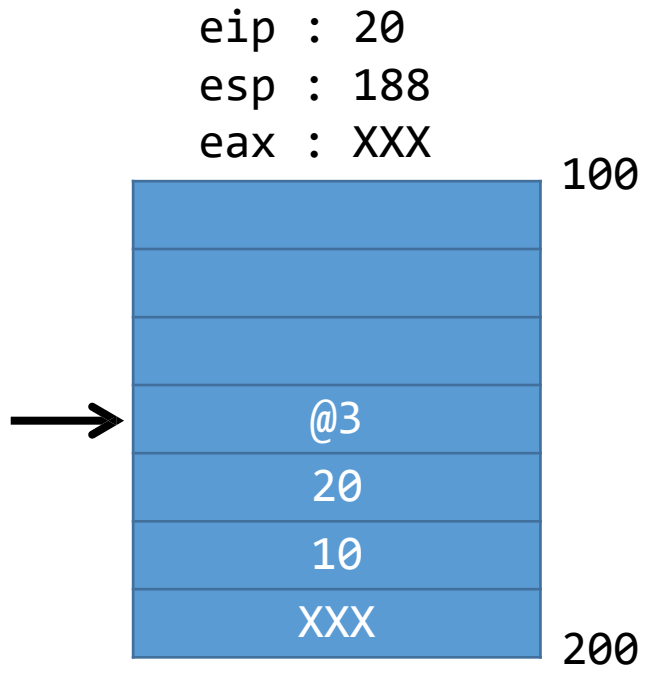



eip : 2
esp : 192
@0: push 10
@1: push 20
@2: call add
@3: ….
push @ret_addr (@3)
add
100
20
10
XXX
200
add:
@20: push ebp
@21: mov ebp, esp
@22: sub esp, 8
@23: mov eax, [ebp+8]
@24: add eax, [ebp+12]
@25: mov esp, ebp
@26: pop ebp
@27: ret


eip : 2
esp : 188
@0: push 10
@1: push 20
@2: call add
@3: ….
push @ret_addr (@3)
add
100
@3
20
10
XXX
200
add:
@20: push ebp
@21: mov ebp, esp
@22: sub esp, 8
@23: mov eax, [ebp+8]
@24: add eax, [ebp+12]
@25: mov esp, ebp
@26: pop ebp
@27: ret

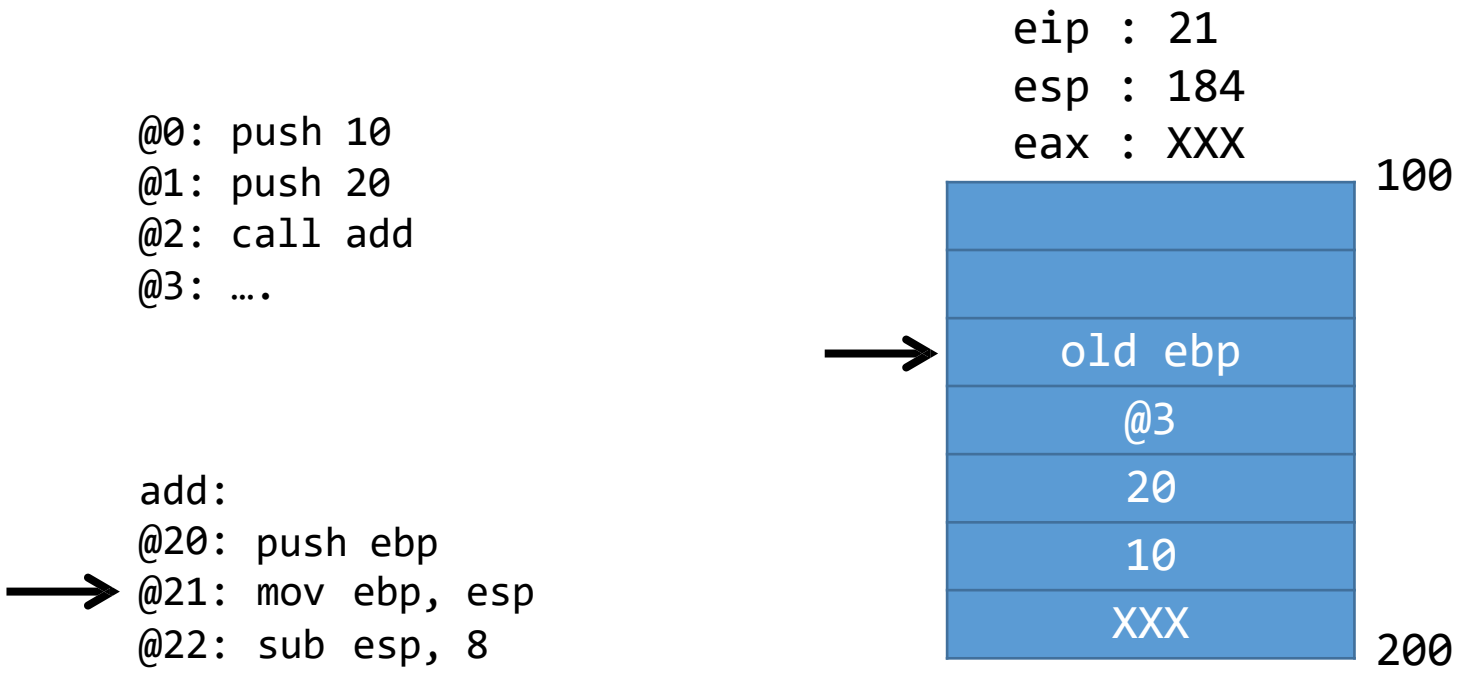
```
@0: push 10
@1: push 20
@2: call add
@3: ….
```

```
add:
@20: push ebp
@21: mov ebp, esp
@22: sub esp, 8
@23: mov eax, [ebp+8]
@24: add eax, [ebp+12]
@25: mov esp, ebp
@26: pop ebp
@27: ret
```



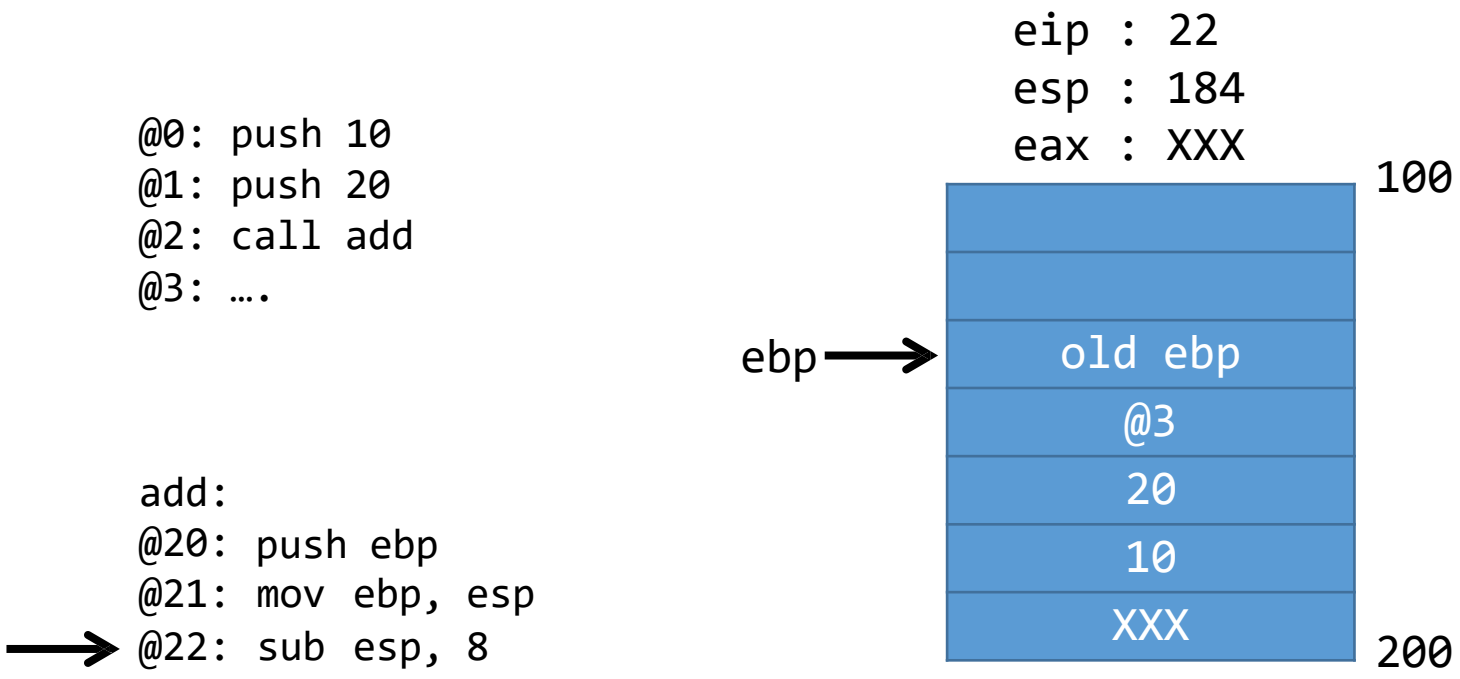
```
@0: push 10
@1: push 20
@2: call add
@3: ….
```

```
add:
@20: push ebp
@21: mov ebp, esp
@22: sub esp, 8
@23: mov eax, [ebp+8]
@24: add eax, [ebp+12]
@25: mov esp, ebp
@26: pop ebp
@27: ret
```



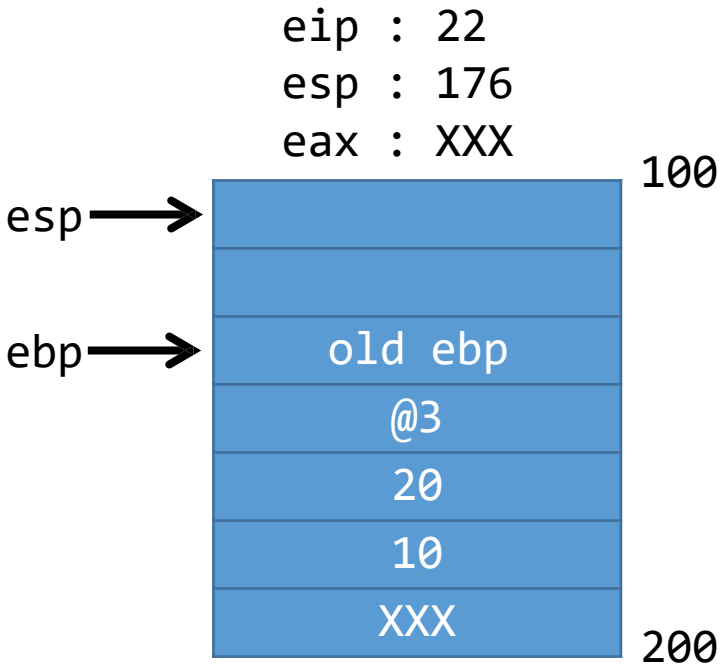
```
@0: push 10
@1: push 20
@2: call add
@3: ….
```

```
add:
@20: push ebp
@21: mov ebp, esp
@22: sub esp, 8
@23: mov eax, [ebp+8]
@24: add eax, [ebp+12]
@25: mov esp, ebp
@26: pop ebp
@27: ret
```



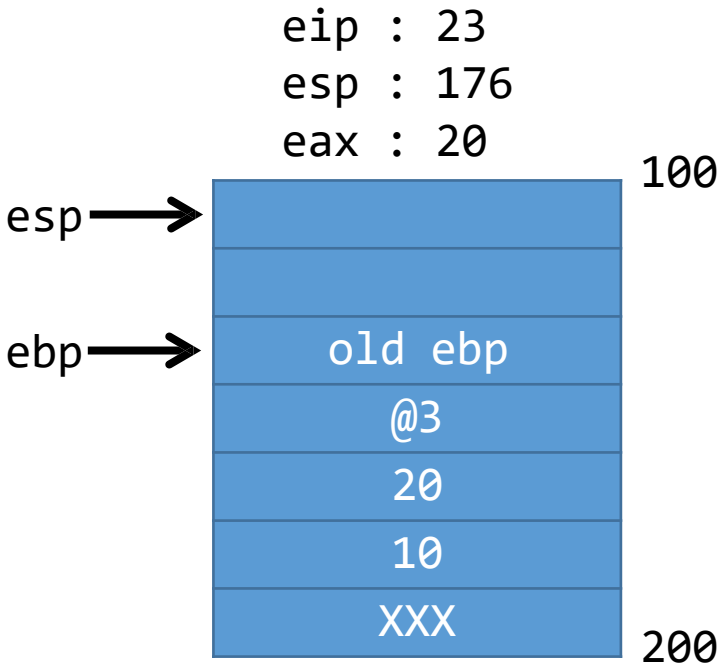
```
@0: push 10
@1: push 20
@2: call add
@3: ….
```

```
add:
@20: push ebp
@21: mov ebp, esp
@22: sub esp, 8
@23: mov eax, [ebp+8]
@24: add eax, [ebp+12]
@25: mov esp, ebp
@26: pop ebp
@27: ret
```



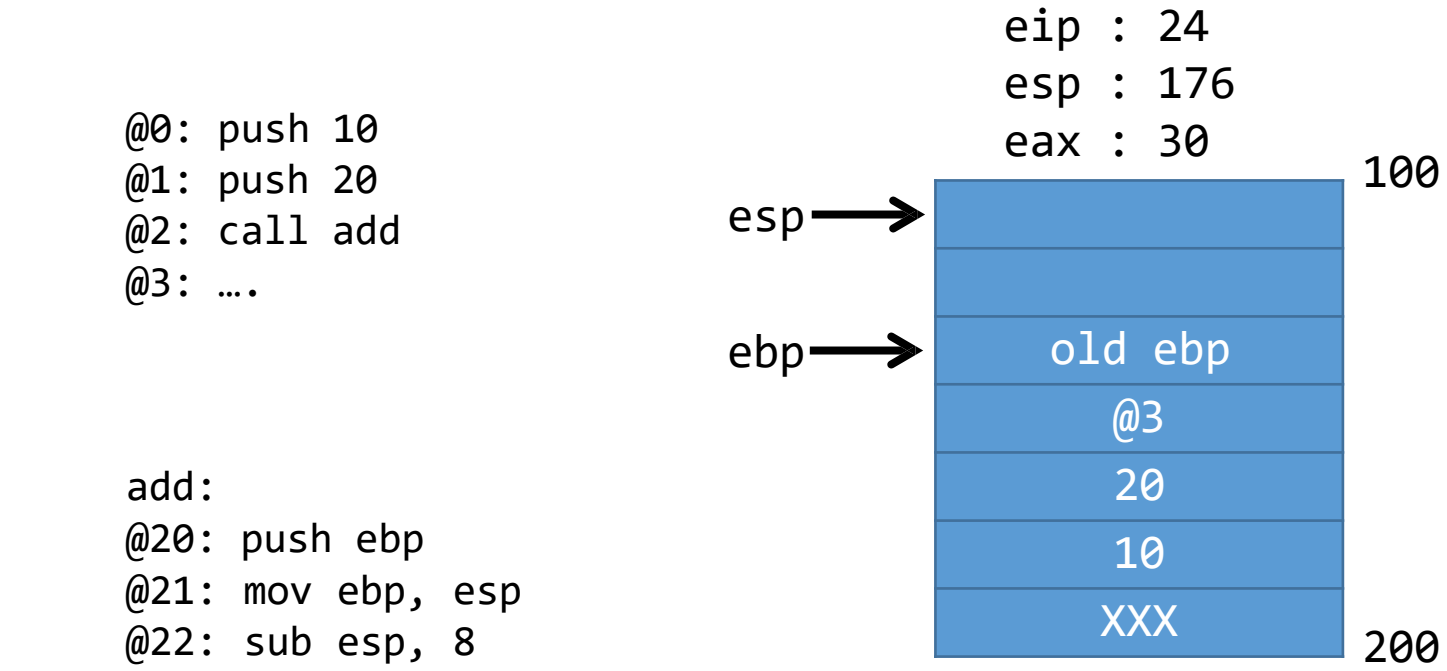
```
@0: push 10
@1: push 20
@2: call add
@3: ….
```

```
add:
@20: push ebp
@21: mov ebp, esp
@22: sub esp, 8
@23: mov eax, [ebp+8]
@24: add eax, [ebp+12]
@25: mov esp, ebp
@26: pop ebp
@27: ret
```



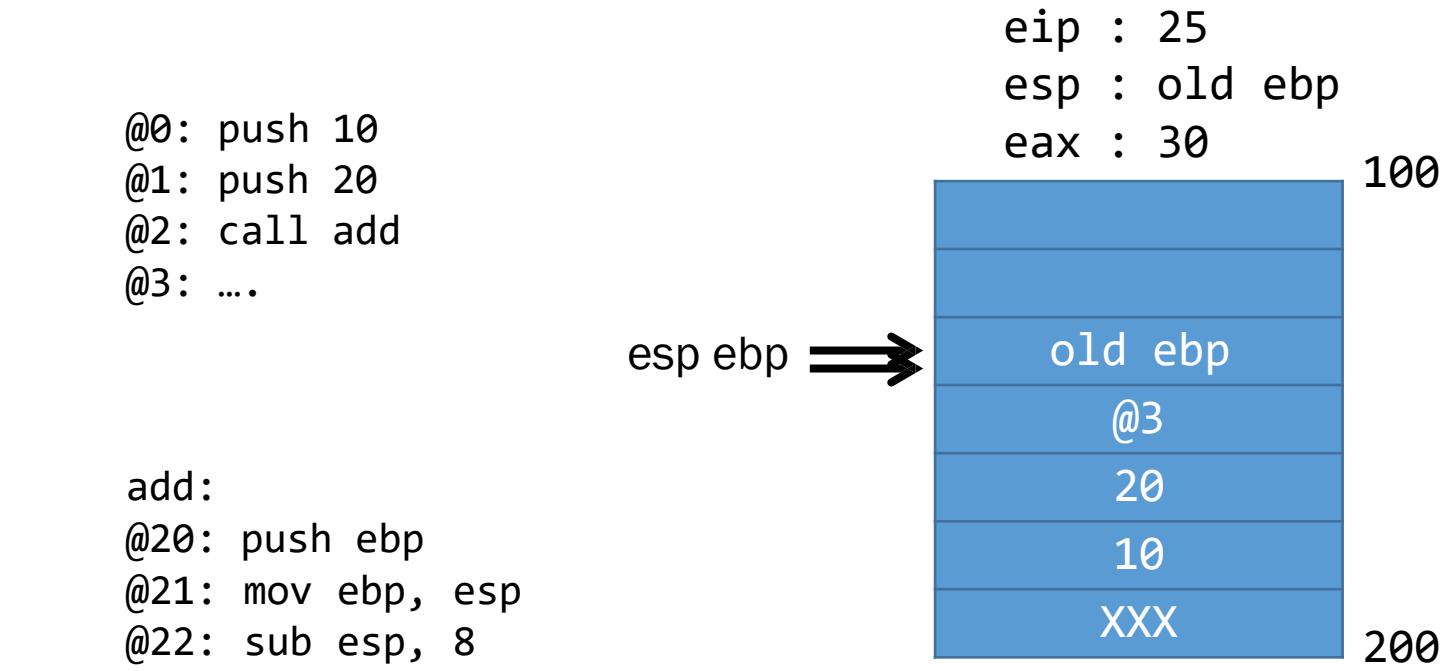
```
@0: push 10
@1: push 20
@2: call add
@3: ….
```

```
add:
@20: push ebp
@21: mov ebp, esp
@22: sub esp, 8
@23: mov eax, [ebp+8]
@24: add eax, [ebp+12]
@25: mov esp, ebp
@26: pop ebp
@27: ret
```



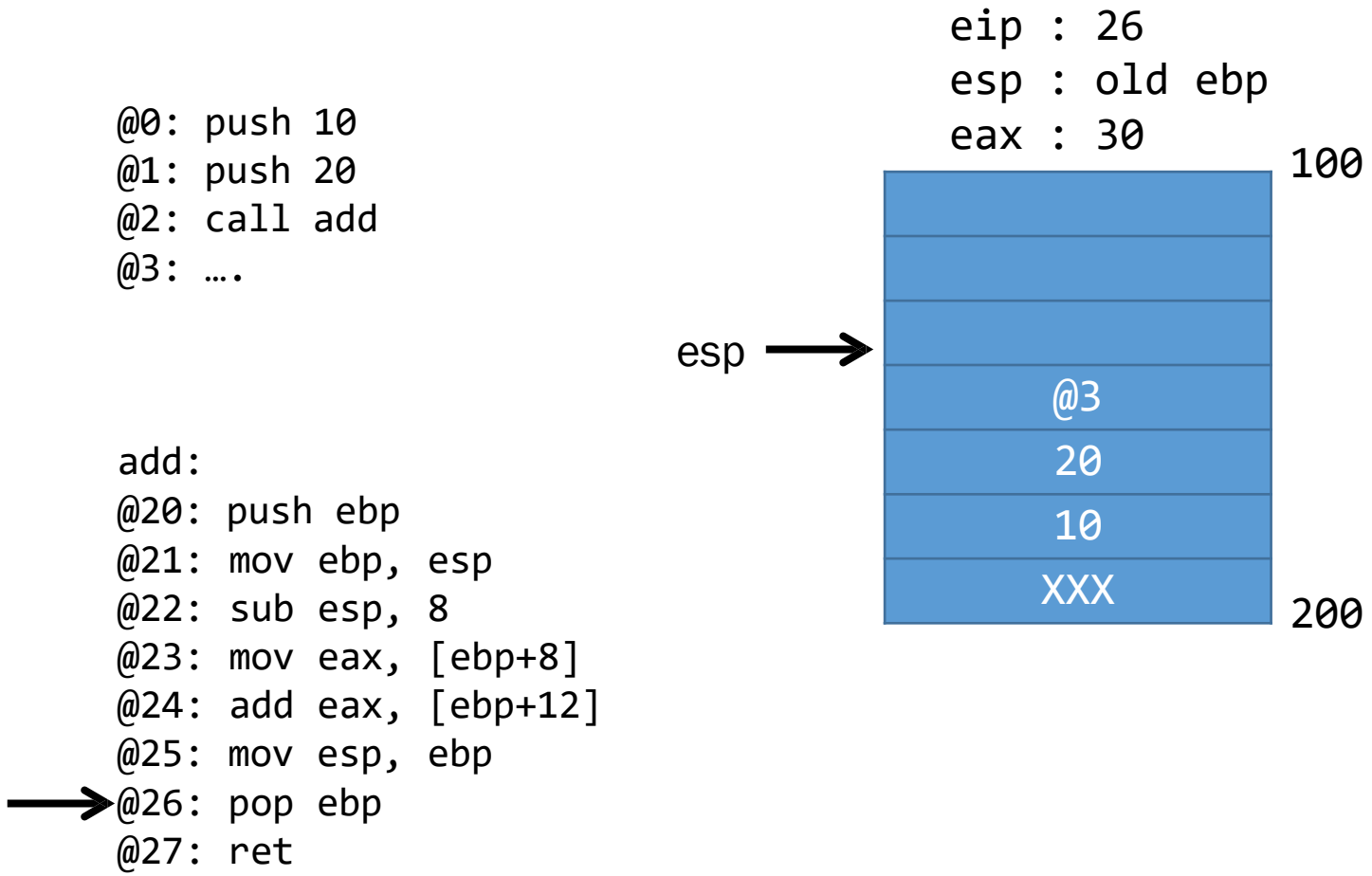
```
@0: push 10
@1: push 20
@2: call add
@3: ….
```

```
add:
@20: push ebp
@21: mov ebp, esp
@22: sub esp, 8
@23: mov eax, [ebp+8]
@24: add eax, [ebp+12]
@25: mov esp, ebp
@26: pop ebp
@27: ret
```



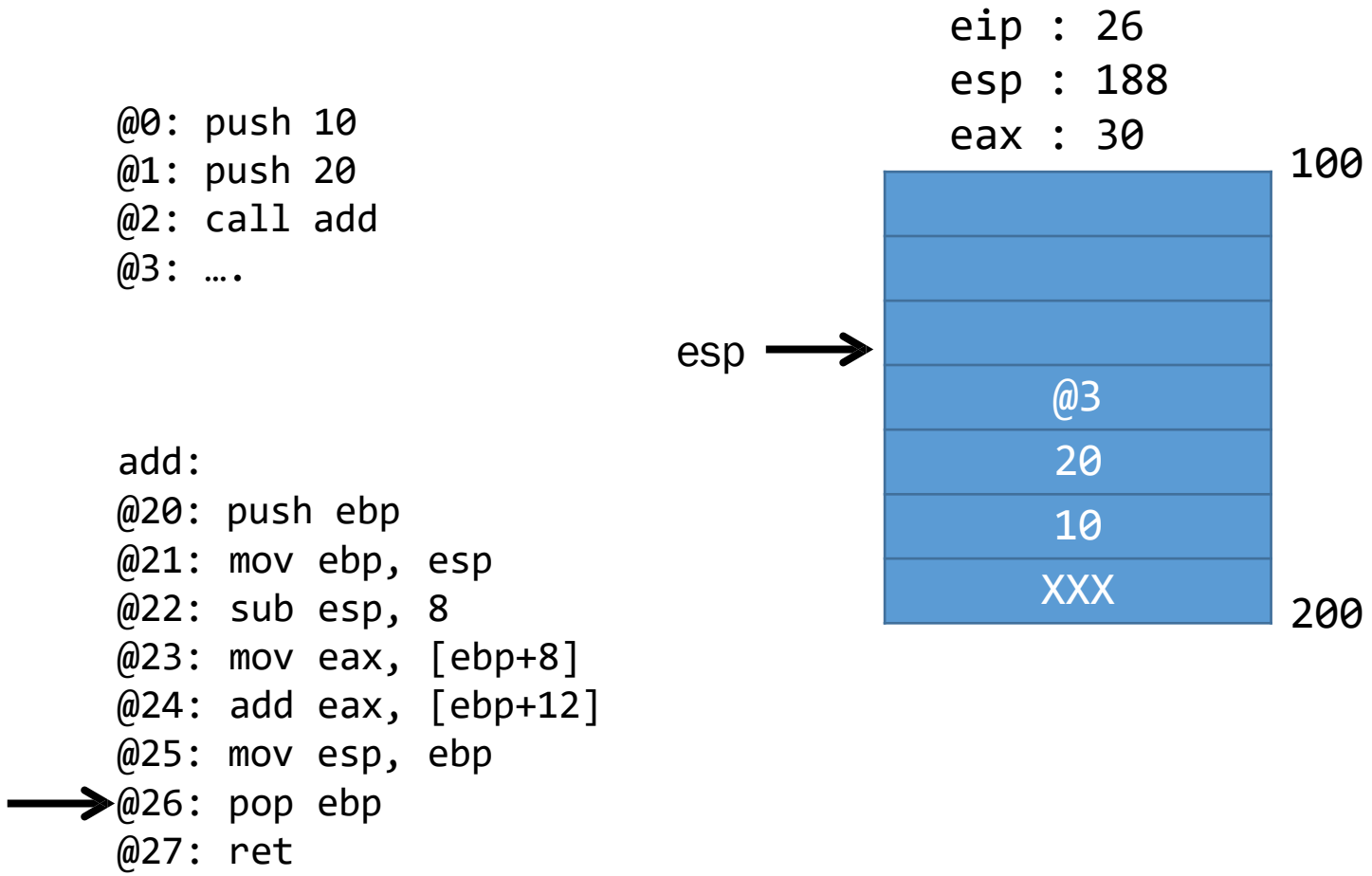
```
@0: push 10
@1: push 20
@2: call add
@3: ….
```

```
add:
@20: push ebp
@21: mov ebp, esp
@22: sub esp, 8
@23: mov eax, [ebp+8]
@24: add eax, [ebp+12]
@25: mov esp, ebp
@26: pop ebp
@27: ret
```



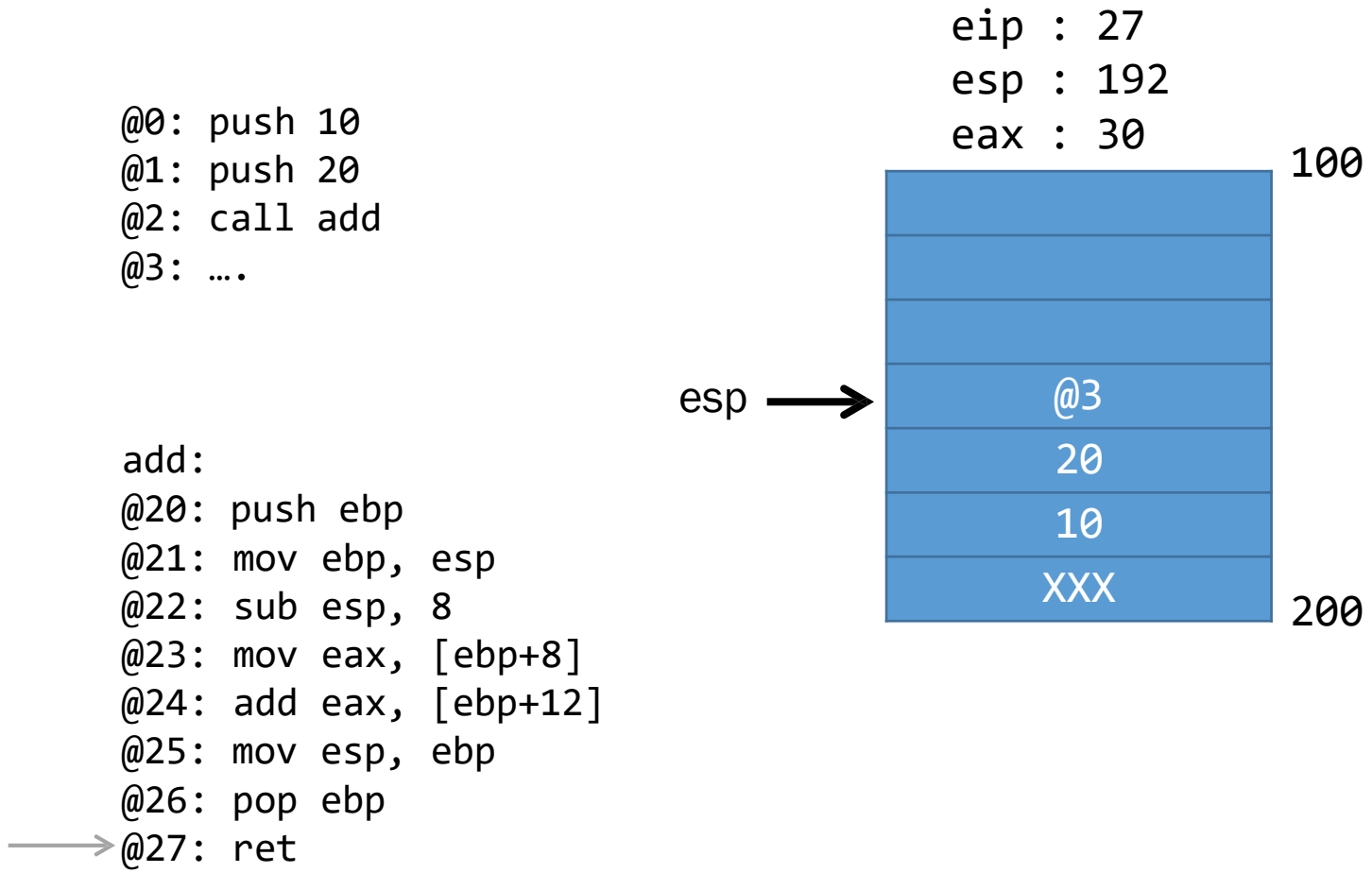
```
@0: push 10
@1: push 20
@2: call add
@3: ….
```

```
add:
@20: push ebp
@21: mov ebp, esp
@22: sub esp, 8
@23: mov eax, [ebp+8]
@24: add eax, [ebp+12]
@25: mov esp, ebp
@26: pop ebp
@27: ret
```



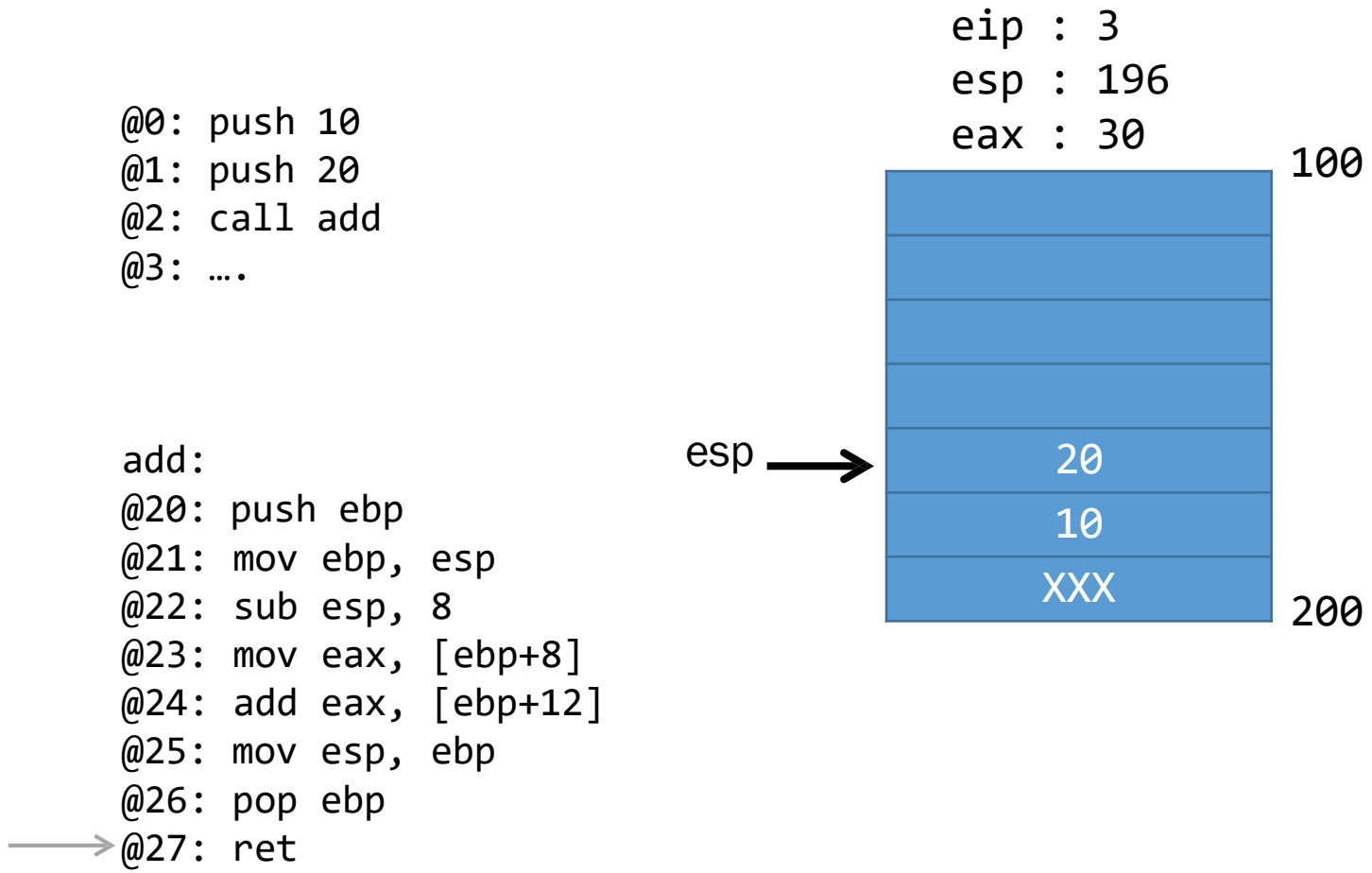
```
@0: push 10
@1: push 20
@2: call add
@3: ….
```

```
add:
@20: push ebp
@21: mov ebp, esp
@22: sub esp, 8
@23: mov eax, [ebp+8]
@24: add eax, [ebp+12]
@25: mov esp, ebp
@26: pop ebp
@27: ret
```



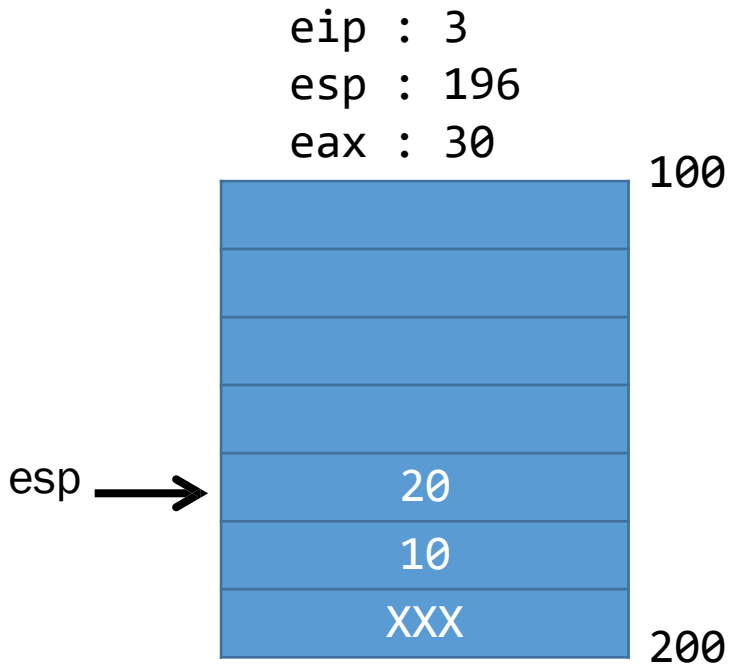
```
@0: push 10
@1: push 20
@2: call add
@3: ….
```

```
add:
@20: push ebp
@21: mov ebp, esp
@22: sub esp, 8
@23: mov eax, [ebp+8]
@24: add eax, [ebp+12]
@25: mov esp, ebp
@26: pop ebp
@27: ret
```



```
@0: push 10
@1: push 20
@2: call add
@3: ….
```

```
add:
@20: push ebp
@21: mov ebp, esp
@22: sub esp, 8
@23: mov eax, [ebp+8]
@24: add eax, [ebp+12]
@25: mov esp, ebp
@26: pop ebp
@27: ret
```



CRACKME

WARM UP

EXERCISES

- Crackme
  1. Find the key
  2. Document your analysis

The crackmes are on campus (yet?)

Take the zip file and wait for us to give you the archive password

EXERCISES

- Crackme_ECE_*
  - 1 -> Very easy
  - 2 -> Easy
  - 3 -> Easy
  - 4 -> Some may find it hard…
- Group
  - 1 person

TAKE AWAYS AND EXERCISES

WARM UP

CONTROL FLOW?

CONTROL FLOW?

```
demo_stackframe(int, int, int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        mov     DWORD PTR [rbp-12], edx
        mov     edx, DWORD PTR [rbp-8]
        mov     eax, DWORD PTR [rbp-12]
        add     eax, edx
        cmp     eax, DWORD PTR [rbp-4]
        jge     .L3
        mov     DWORD PTR [rbp-4], 10
.L3:
        nop
        pop     rbp
        ret
```

CONTROL FLOW?

```
demo_stackframe(int, int, int):
        push    ebp
        mov     ebp, esp
        mov     DWORD PTR [ebp-20], edi
        mov     DWORD PTR [ebp-24], esi
        mov     DWORD PTR [ebp-28], edx
        mov     DWORD PTR [ebp-4], 0
.L3:
        cmp     DWORD PTR [ebp-4], 9
        jg      .L4
        mov     eax, DWORD PTR [ebp-24]
        add     DWORD PTR [ebp-20], eax
        mov     eax, DWORD PTR [ebp-28]
        add     DWORD PTR [ebp-24], eax
        add     DWORD PTR [ebp-4], 1
        jmp     .L3
.L4:
        nop
        pop     ebp
        ret
```

CONTROL FLOW?

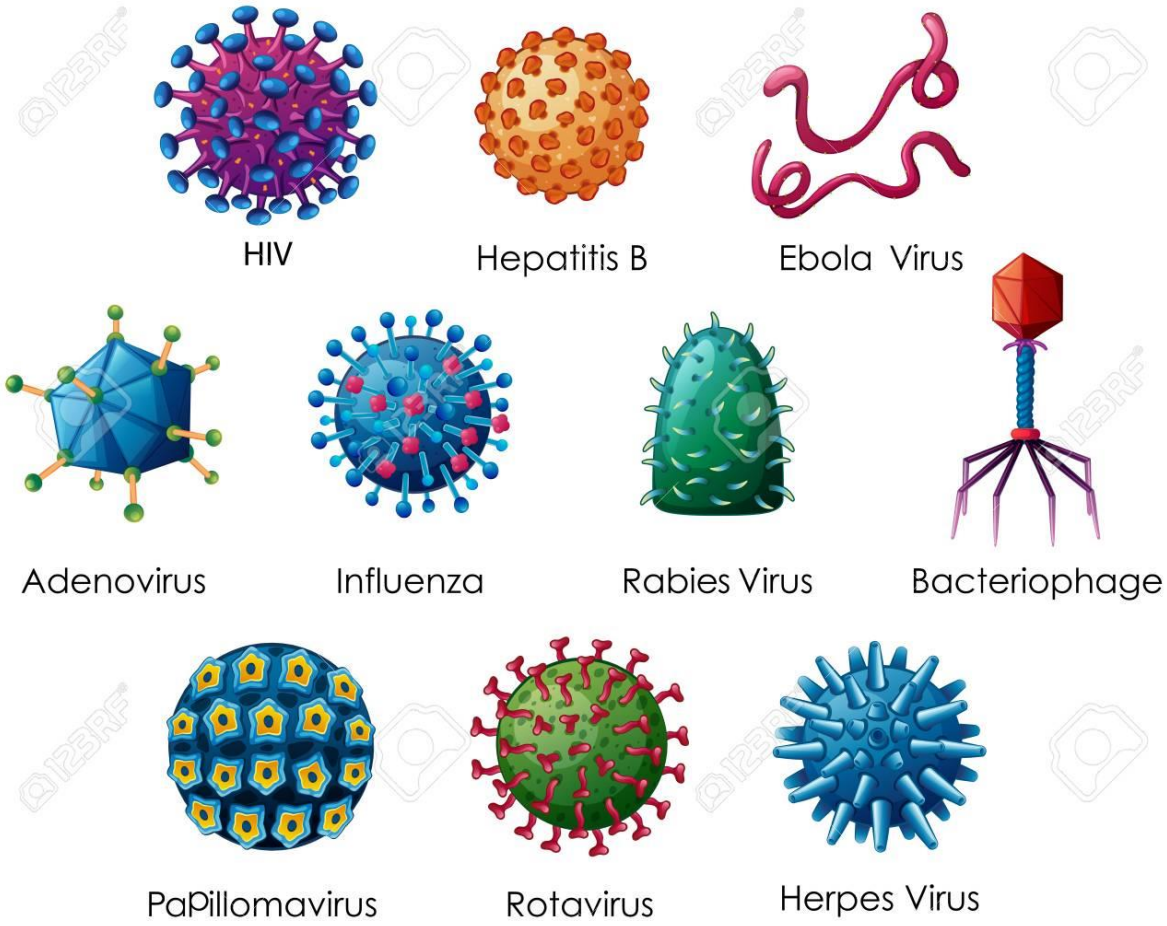
```
demo_stackframe(int, int, int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        mov     DWORD PTR [rbp-12], edx
.L3:
        mov     eax, DWORD PTR [rbp-12]
        add     DWORD PTR [rbp-8], eax
        sub     DWORD PTR [rbp-4], 1
        cmp     DWORD PTR [rbp-4], 0
        jle     .L4
        jmp     .L3
.L4:
        nop
        pop     rbp
        ret
```

CONTROL FLOW?

```
demo_stackframe(int, int, int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        mov     DWORD PTR [rbp-12], edx
.L3:
        mov     eax, DWORD PTR [rbp-4]
        and     eax, DWORD PTR [rbp-8]
        test    eax, eax
        je      .L4
        mov     eax, DWORD PTR [rbp-4]
        and     eax, DWORD PTR [rbp-12]
        mov     DWORD PTR [rbp-8], eax
        add     DWORD PTR [rbp-4], 1
        jmp     .L3
.L4:
        nop
        pop     rbp
        ret
```

MALWARE ANALYSIS

DIFFERENT TYPES?



THREE OBJECTIVES, SEVERAL MEANS

- Money & intelligence & war
  - Ransomware & Stuxnet & Estonia 2007

- Money : preventing people from working (ransom)
- Money : stealing information (dataleak)
- Intelligence : acting on nuclear plants
- Intelligence : stealing weapon information
- War : acting on satellite orbit
- War : destroy hospital network

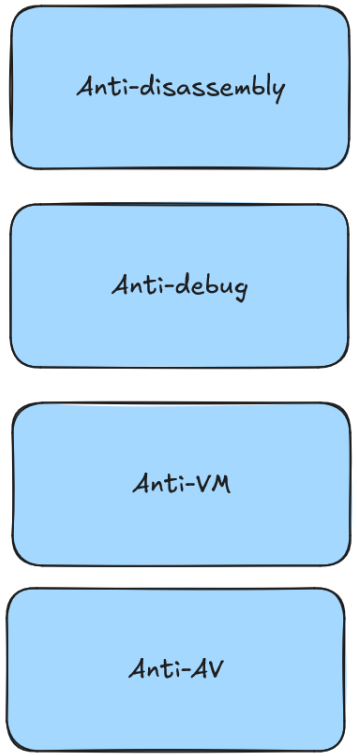
MALWARE TECHNICAL ATTRIBUTES

- Persistence
  - Stay on place even after reboot
- Lateral/Vertical movement
  - Lateral : Look for other interesting system on network
  - Vertical : Gain privilege on the system (administrator)
  - Exploit (Remote Code Execution or Local Privilege Escalation)
- Payload
  - After all of that, what is the main purpose of the malware?

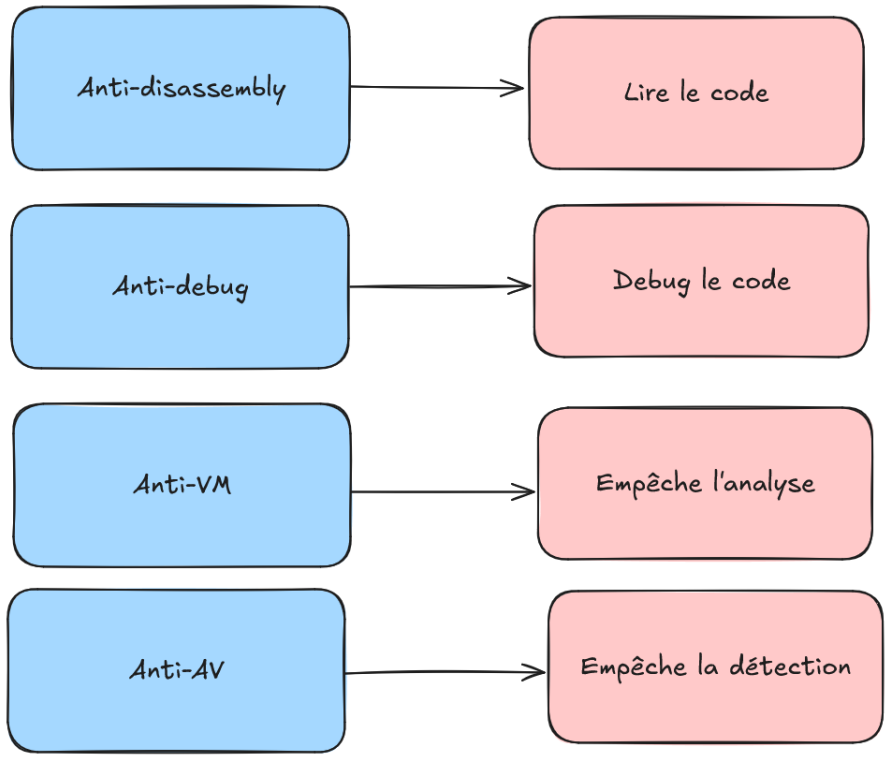
MALWARE ANTI-ANALYSIS TECHNIQUES

- Anti-disassembly
  - Counter static analysis
  - Eg. packer ➔ encrypt payload
- Anti-debug
  - Counter dynamic analysis
- Anti-vm
  - Counter automatic analysis (?)
  - Not really a thing as most of the data is in virtual machines

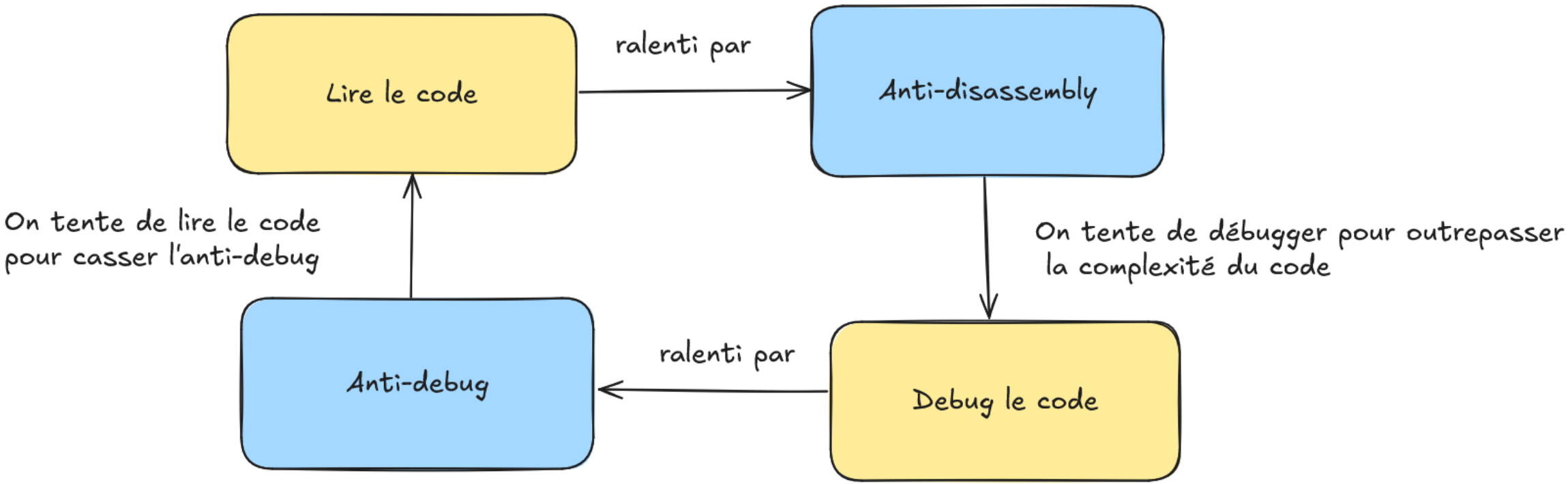
ANTI-ANALYSE



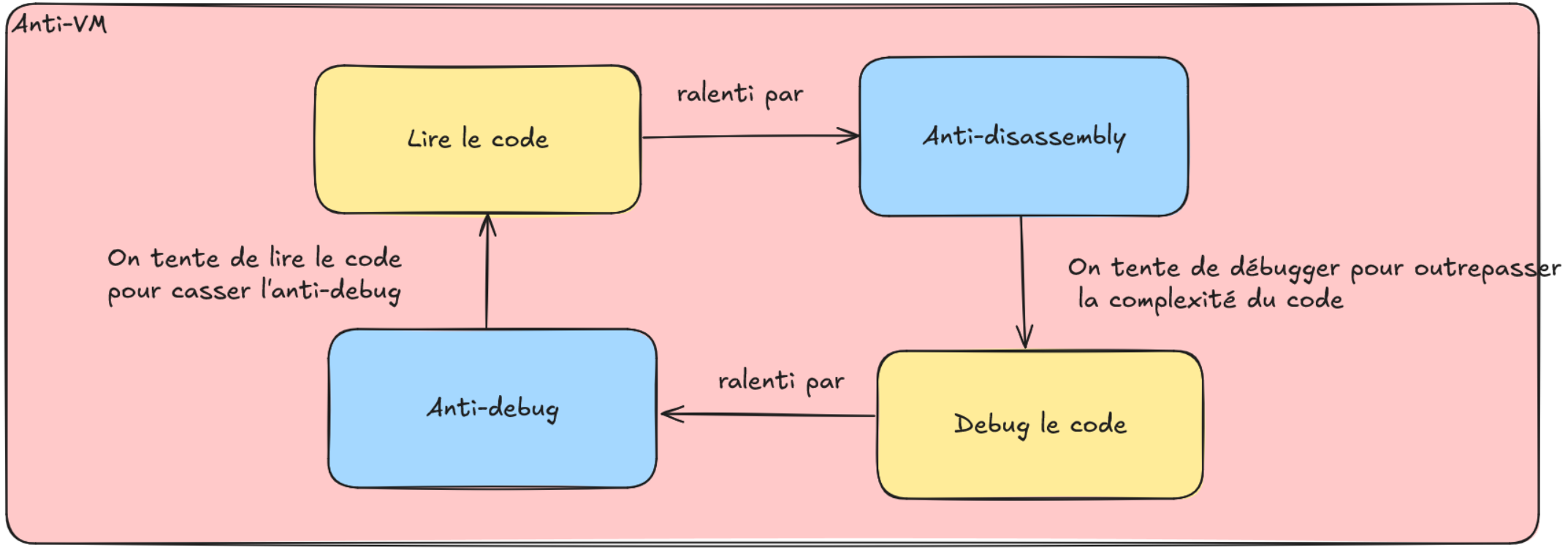
ANTI-ANALYSE



ANTI-ANALYSE



ANTI-ANALYSE



HOW TO PREVENT INFECTION?

- Apply in depth defence!

1. Coworker education
2. System updates
3. Privilege segmentation
4. Network segmentation

MALWARE ANALYSIS

- Report
  - What does the malware do on the system?
  - How does it persist?
  - How does it propagate?
  - Does it use exploit?
    - What vulnerability does it exploit?
  - How can i detect it automatically?
  - Does it leave artifacts?
- Example : https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2018/03/07205202/The_Mystery_of_Duqu_2_0_a_sophisticated_cyberespionage_actor_returns.pdf

BYE BYE!

END

FEEDBACKS?