CSE467: Computer Security

12. Introduction & Assembly (x86)

Seongil Wi

Department of Computer Science and Engineering

The slide is based on Prof. Sang Kil Cha's lecture slide

HW2: Web Hacking Competition

- Hacking practice: Capture the Flag (CTF)
- Challenge open (competition start): 4/14 (Mon)
- Due date (writeup report): 5/2 (Fri)

Introduction to Software Security

Secure Software?

Can we say a program is secure if it considers the CIA properties?

Where there is engineering, there is a ***security problem***

Why?

Humans always make ***mistakes***

Software Security is About Software Bugs

- Find software bugs
- Exploit software bugs
- Patch software bugs

Software Bug

- Software bug is an *error/fault/mistake* in the code that produces **an unexpected result**

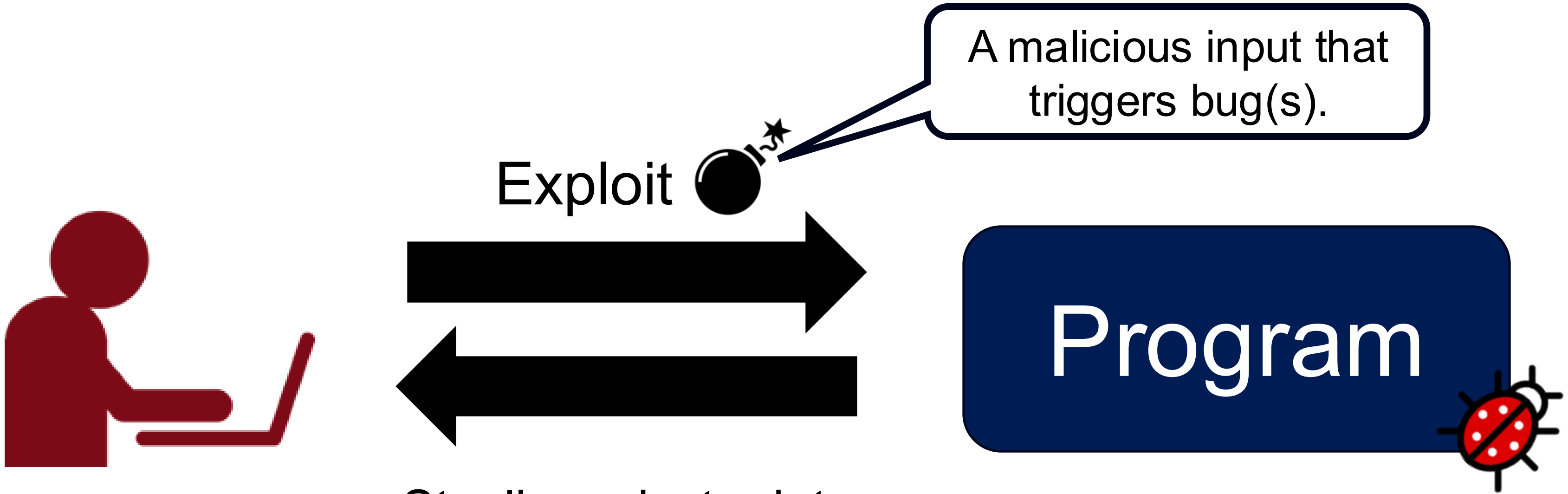
Thinking like an adversary is important for security

Software Bugs is the Key

- ***Root cause*** of many security problems including malware, hacking incidents, phishing, privacy leakage, etc.

Not Every Bug is Security Critical

- Bug is an error in program that makes it malfunction
  - Ex) Compute wrong outputs for corner case inputs
- Vulnerability is a bug that causes security issues



- Stealing private data
- Executing arbitrary commands
- ...

Example #1

- Consider the python code below for *bank application*
  - Takes in the amount of money you want to transfer
  - Your balance and the recipient's balance will be updated

```
my_balance = 1000
def send(recipient):
    print("How much do you want to send?")
    val = read_int()
    if (val <= my_balance):
        my_balance = my_balance – val
        ... # Increase the balance of recipient
```

Input : **100** → my_balance: **1000 → 900**

What can go wrong with this code?

Example #2

- Next, consider the following C code
  - Reads in a string input and prints it back

```
int main(void) {
    char buf[32];
    printf("Input your name: ");
    scanf("%s", buf);
    printf("Your name: %s\n", buf);
    return 0;
}
```

Input : “**Seongil**” → Printed output: **"Your name: Seongil"**

What can go wrong with this code?

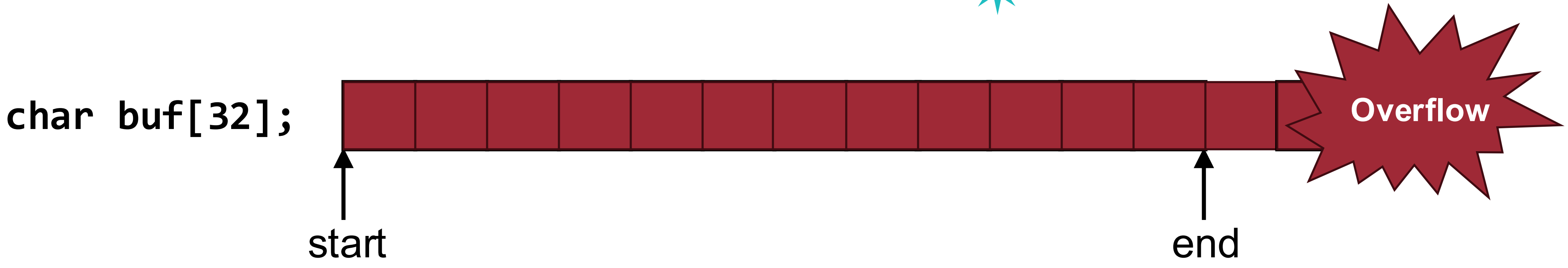
Example #2

- Next, consider the following C code
  - Reads in a string input and prints it back

```
int main(void) {
    char buf[32];
    printf("Input your name: ");
    scanf("%s", buf);
    printf("Your name: %s\n", buf);
    return 0;
}
```

This is infamous *buffer overflow* vulnerability

Buffer Overflow & Memory Corruption



- C has no automatic check on array index and boundary
  - Allows writing past the end of an array
  - This is call buffer overflow, or BOF in short
  - Such write will corrupt other variables and data in the memory

Q. How can a hacker do bad things (e.g., code execution, privilege escalation) with this?

To answer this, we must learn assembly

Our Goal in Software Security

- Find out whether a program is secure or not

- To do so, we need to see how the ***binary code*** (= executable code) executes on a machine!

Assembly

Compilation

- Converting a <u>high-level language</u> into a <u>machine language</u> that the computer can understand

```
int test (int a){
    return 32;
}
```

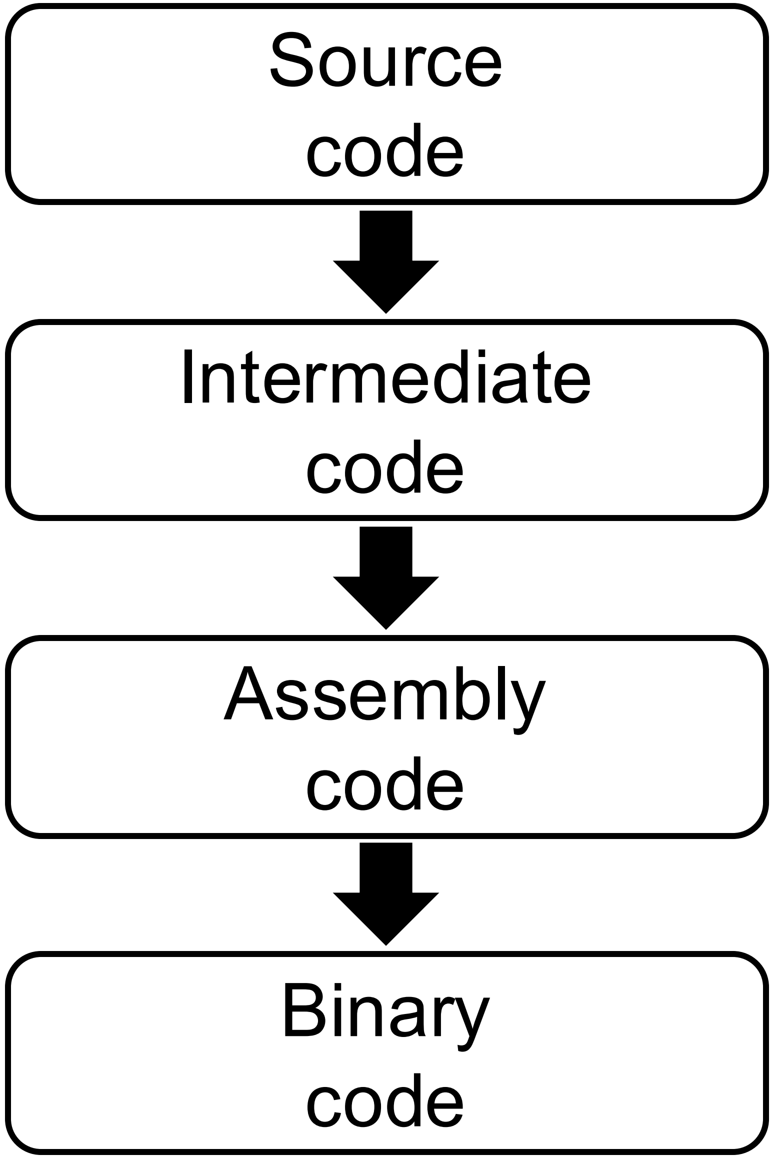
High-level language

Compile

```
01000101010100100101
01001000100000001010
11100011010101010100
10101000001010100010
11100101010100101110
```

Machine language

Compilation Process



Compilation Process

Source code

Intermediate code

Assembly code

Binary code

test.c

```c
int test (int a){
    return 32;
}
```

$clang -emit-llvm -S test.c

test.ll

```
define dso_local i32 @test(i32)
#0 {
  %2 = alloca i32, align 4
  store i32 %0, i32* %2, align 4
  ret i32 32
}
```

Compilation Process

Source code

↓

Intermediate code

↓

Assembly code

↓

Binary code

test.ll

```
define dso_local i32 @test(i32) #0
{
  %2 = alloca i32, align 4
  store i32 %0, i32* %2, align 4
  ret i32 32
}
```

```
$llc --x86-asm-syntax=intel test.ll
```

test.s

```
test:                    # @test
# %bb.0:
push ebp
mov ebp, esp
mov eax, dword ptr [ebp + 8]
mov eax, 32
pop ebp
ret
```

Compilation Process

Source code

Intermediate code

Assembly code

Binary code

test.ll

```
define dso_local i32 @test(i32) #0
{
  %2 = alloca i32, align 4
  store i32 %0, i32* %2, align 4
  ret i32 32
}
```

Intel syntax

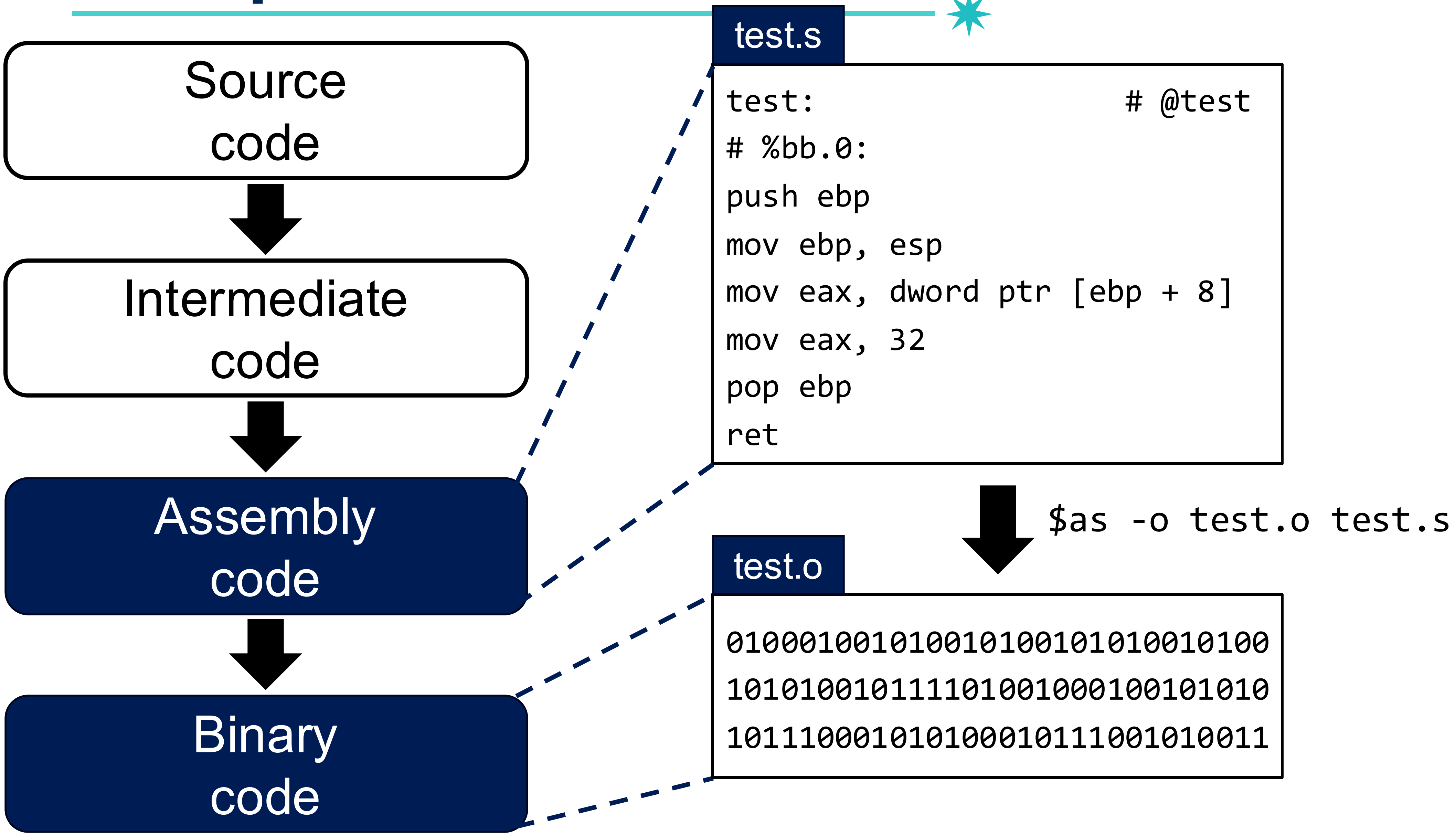
```
$llc --x86-asm-syntax=intel test.ll
```

test.s

```
test:                    # @test
# %bb.0:
push ebp
mov ebp, esp
mov eax, dword ptr [ebp + 8]
mov eax, 32
pop ebp
ret
```

The last human-readable format

Compilation Process

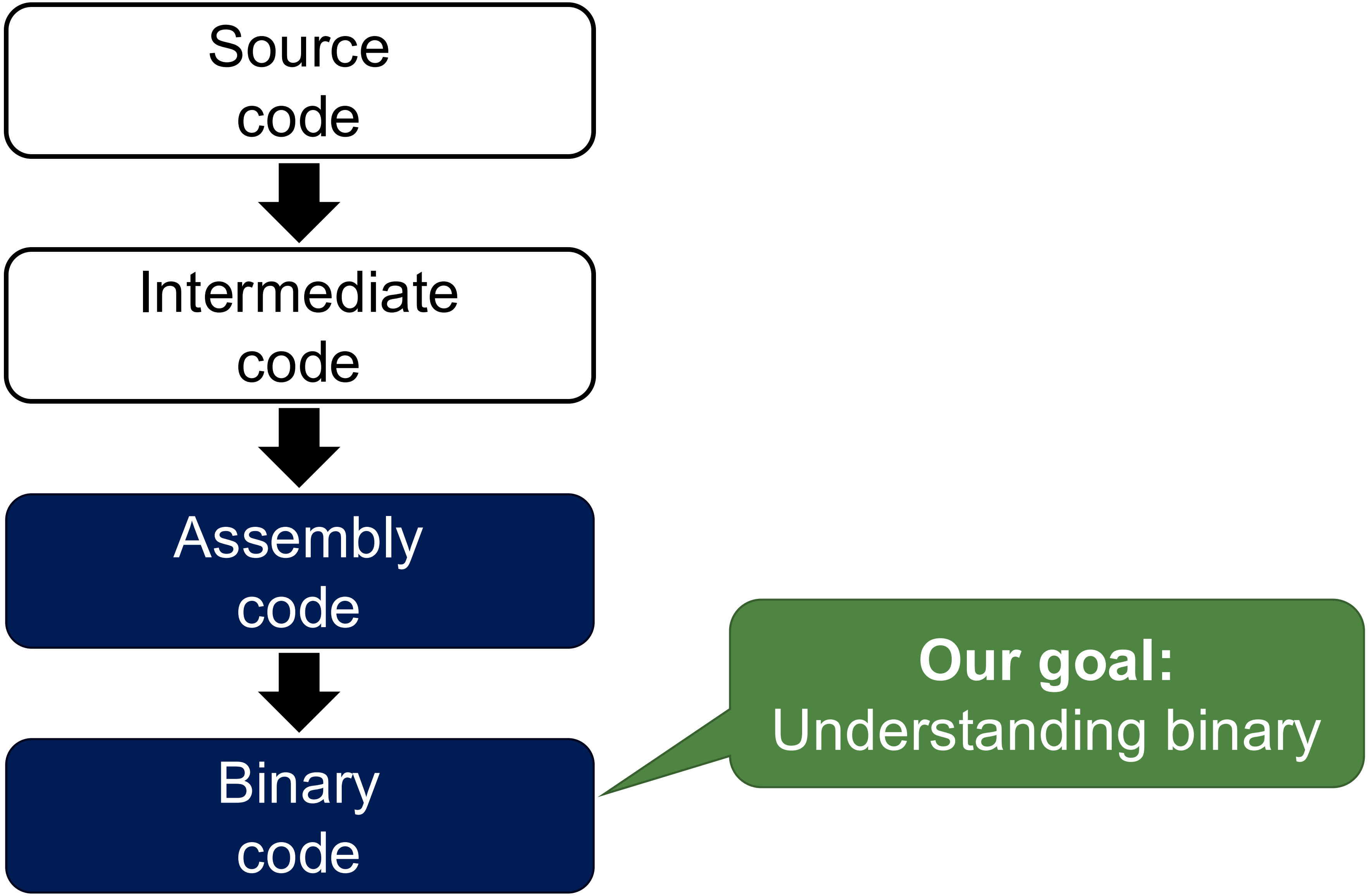


GNU AS (Assembler)

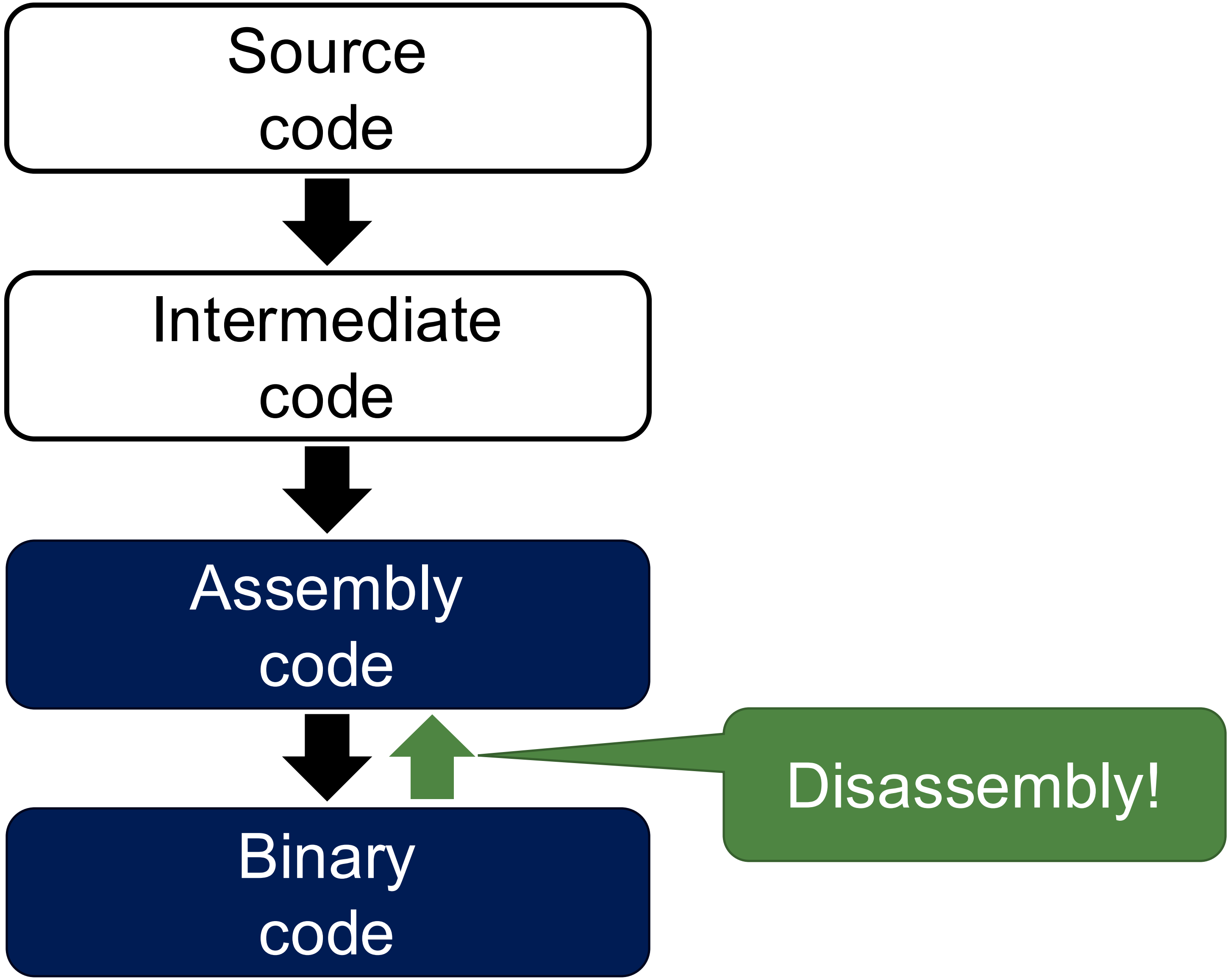
```
$as -o test.o test.s
$ls test.o
```

```
.intel_syntax noprefix

mov eax, ebx
…
```

Compilation Process



Disassembling Binary Code



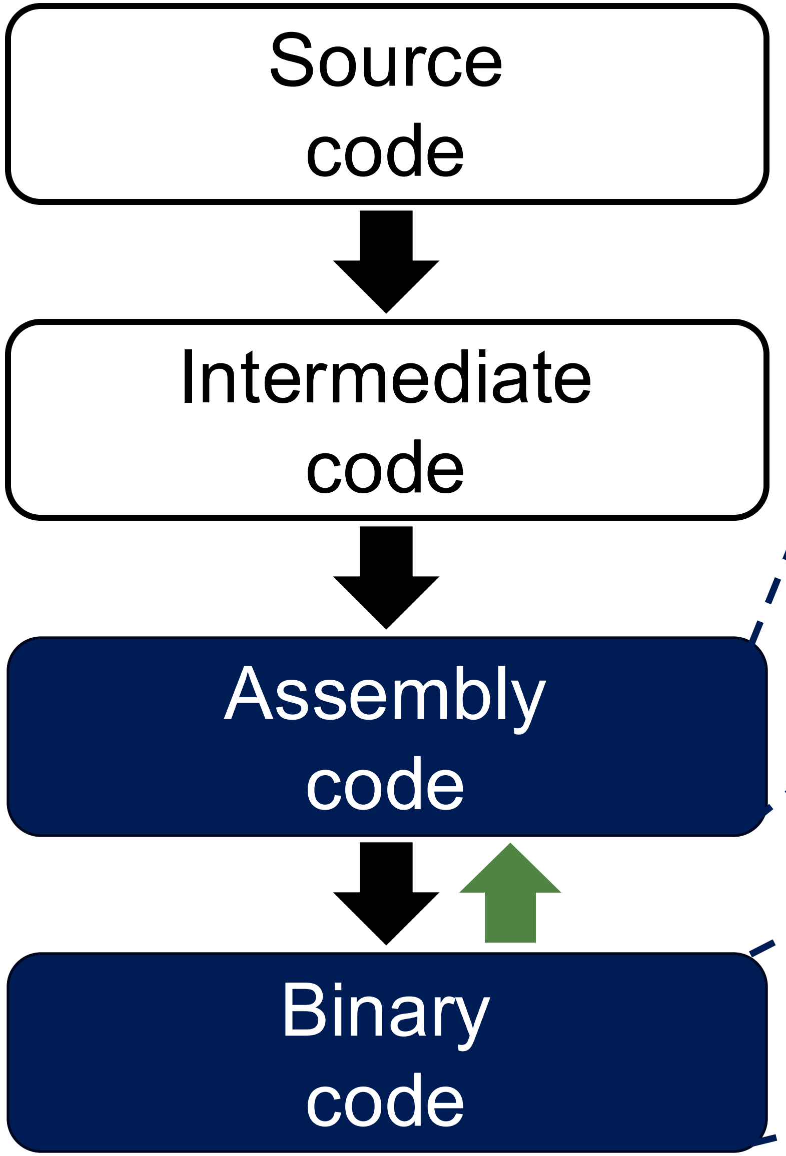
Disassembler:

- Objdump
- IDA
- B2R2
- …

GNU `objdump`

- One of the GNU Binutils
- Perform disassembly on the file

GNU objdump



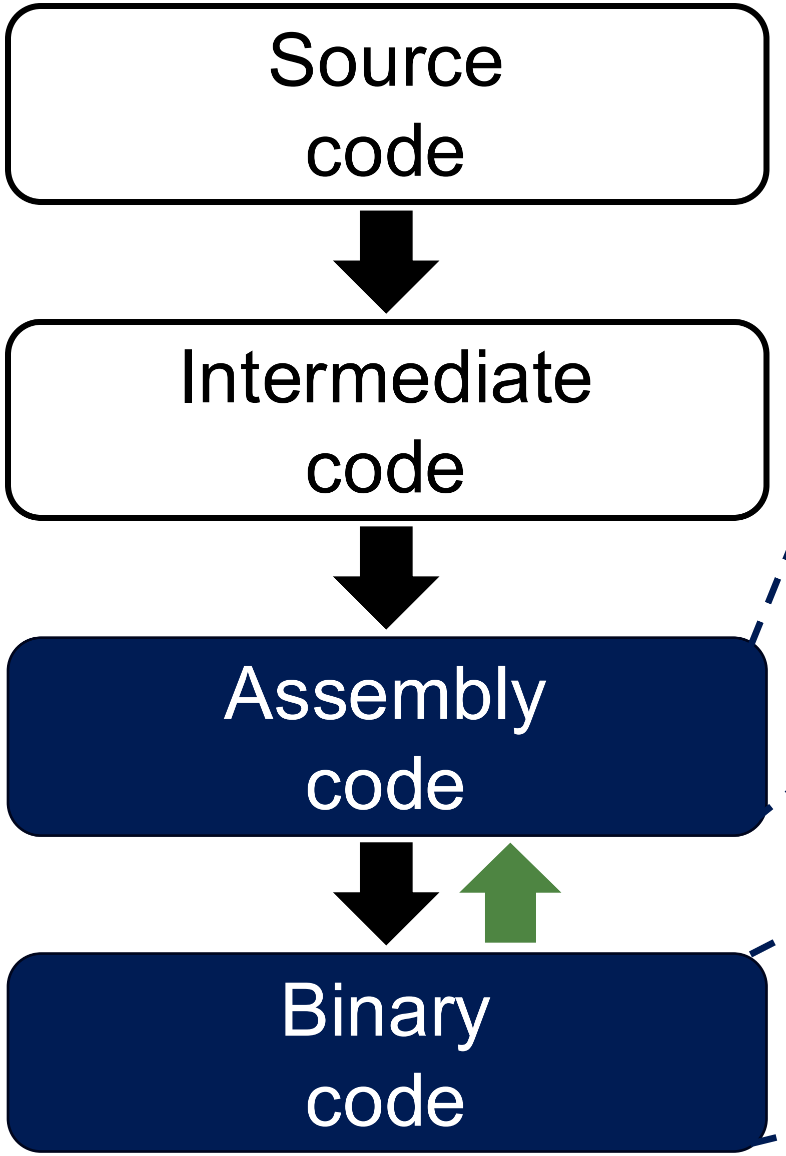

```
00000000 <test>:
 0: 55                push ebp
 1: 89 e5             mov  ebp,esp
 3: 8b 45 08          mov  eax,DWORD PTR [ebp+0x8]
 6: b8 20 00 00 00    mov  eax,0x20
 b: 5d                pop  ebp
 c: c3                ret
```

```
$ objdump -M intel -d test.o
```

test.o

```
010001001010010100101010010100
101010010111101001000100101010
101110001010100010111001010011
```

GNU objdump



Address | Binary code | Disassembled assembly code

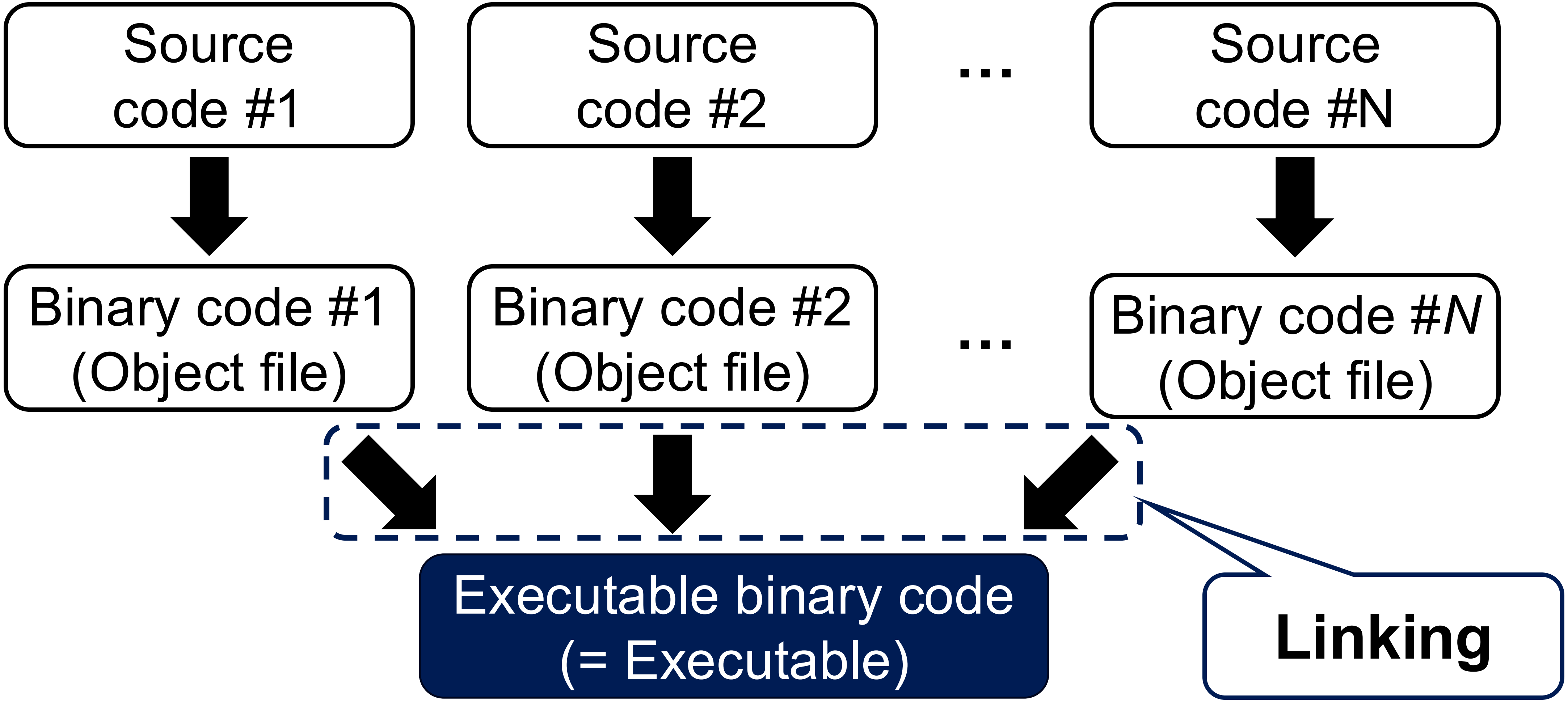
```
00000000 <test>:
   0:  55                   push  ebp
   1:  89 e5                mov   ebp,esp
   3:  8b 45 08             mov   eax,DWORD PTR [ebp+0x8]
   6:  b8 20 00 00 00       mov   eax,0x20
   b:  5d                   pop   ebp
   c:  c3                   ret
```

```
$ objdump -M intel -d test.o
```

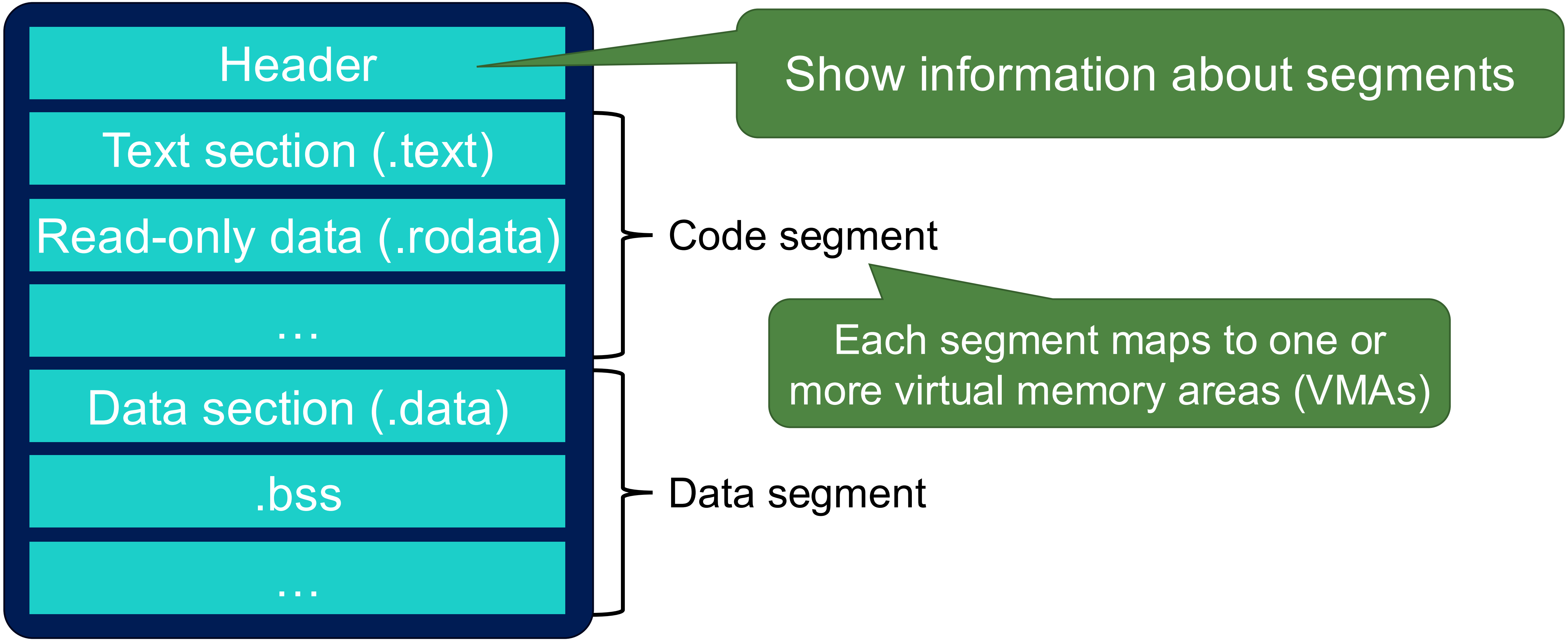
test.o

```
010001001010010100101010010100
101010010111101001000100101010
101110001010100010111001010011
```

Linking

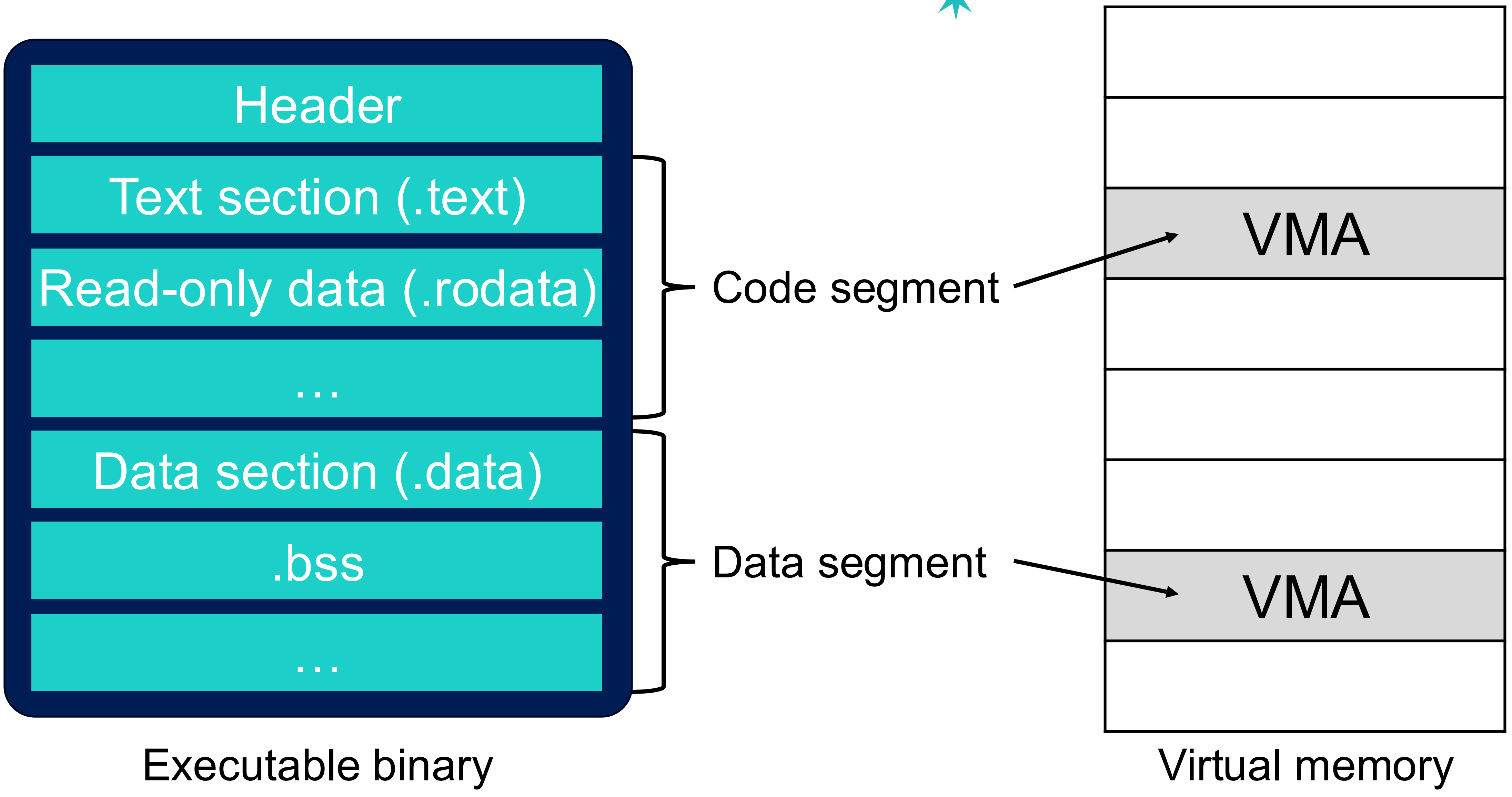

Source code #1
Source code #2
…
Source code #N
Binary code #1 (Object file)
Binary code #2 (Object file)
…
Binary code #N (Object file)
Executable binary code (= Executable)
Linking

Executable Binary (=Executable, Binary)



Executable binary

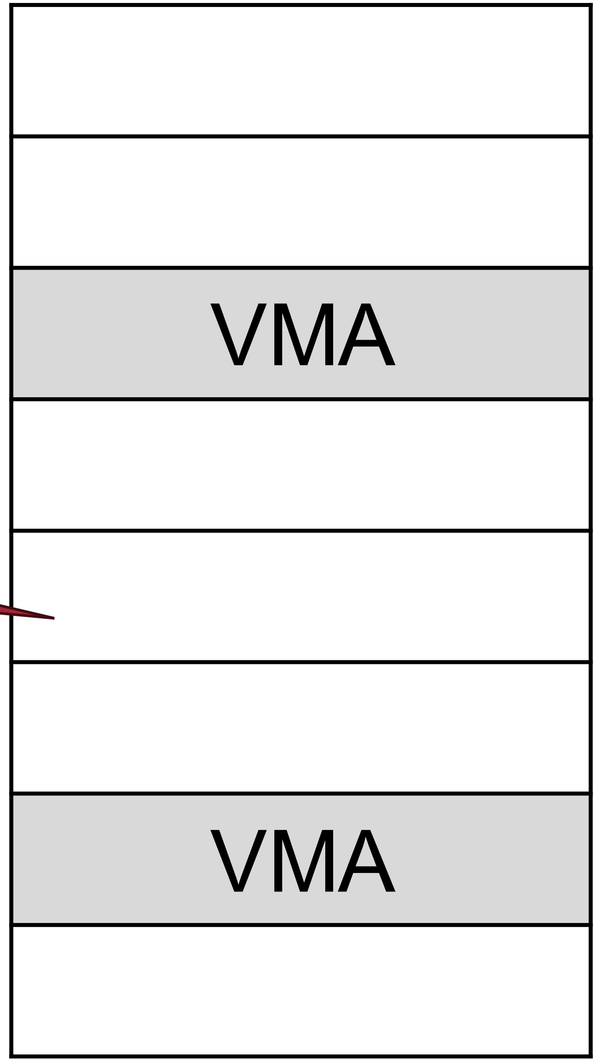
Executable Binary (=Executable, Binary)



Segmentation Fault

- a.k.a., SegFault or Access violation

Happens when we reference an unmapped memory address



Virtual memory

x86 (IA-32) Architecture

x86 Instruction Set Architecture

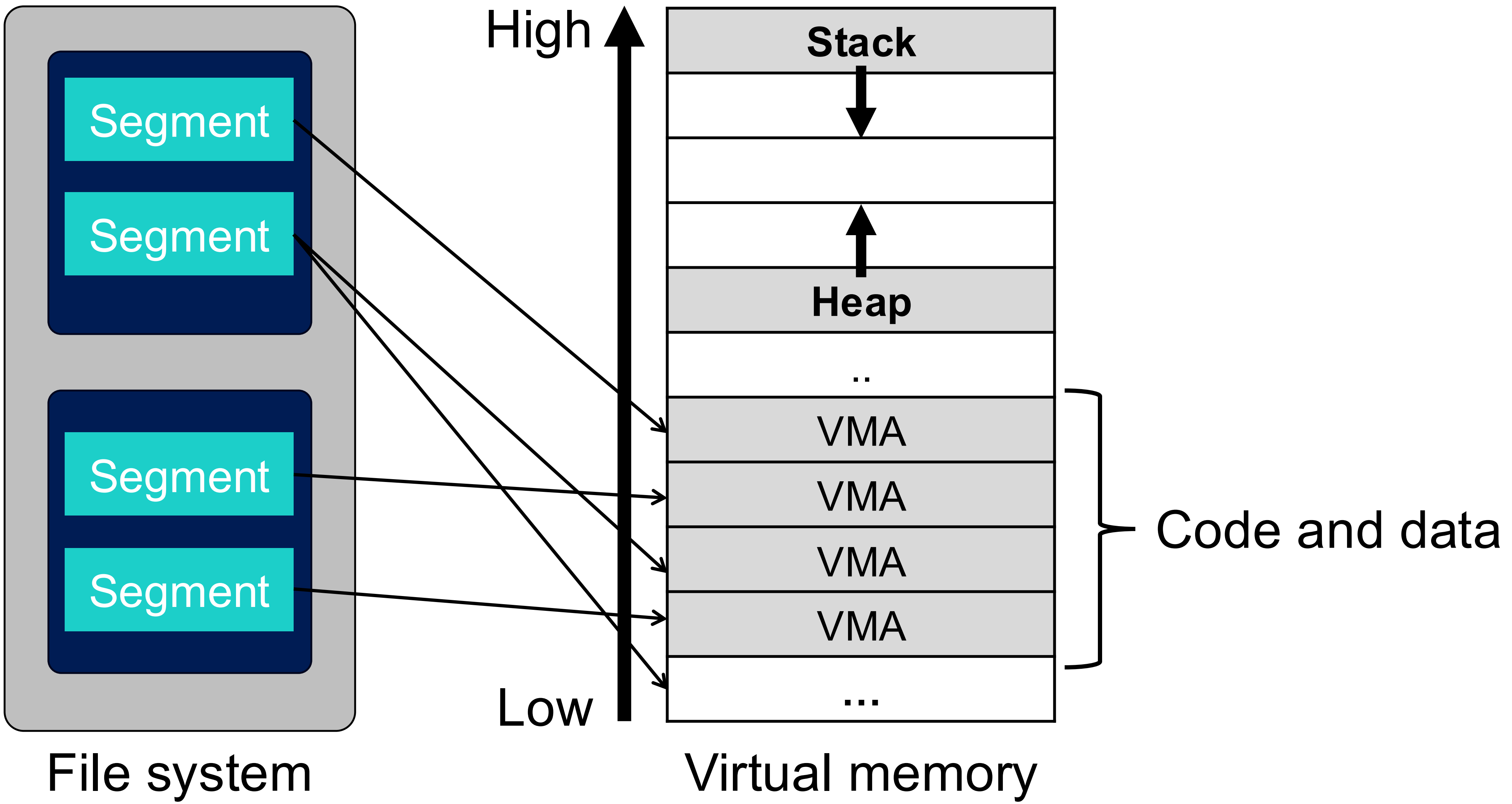
- Developed by Intel in 1985
- CISC (Complex Instruction Set Computer) architecture
- **32-bit** address space
- One of the most common architecture

History of x86 ISA

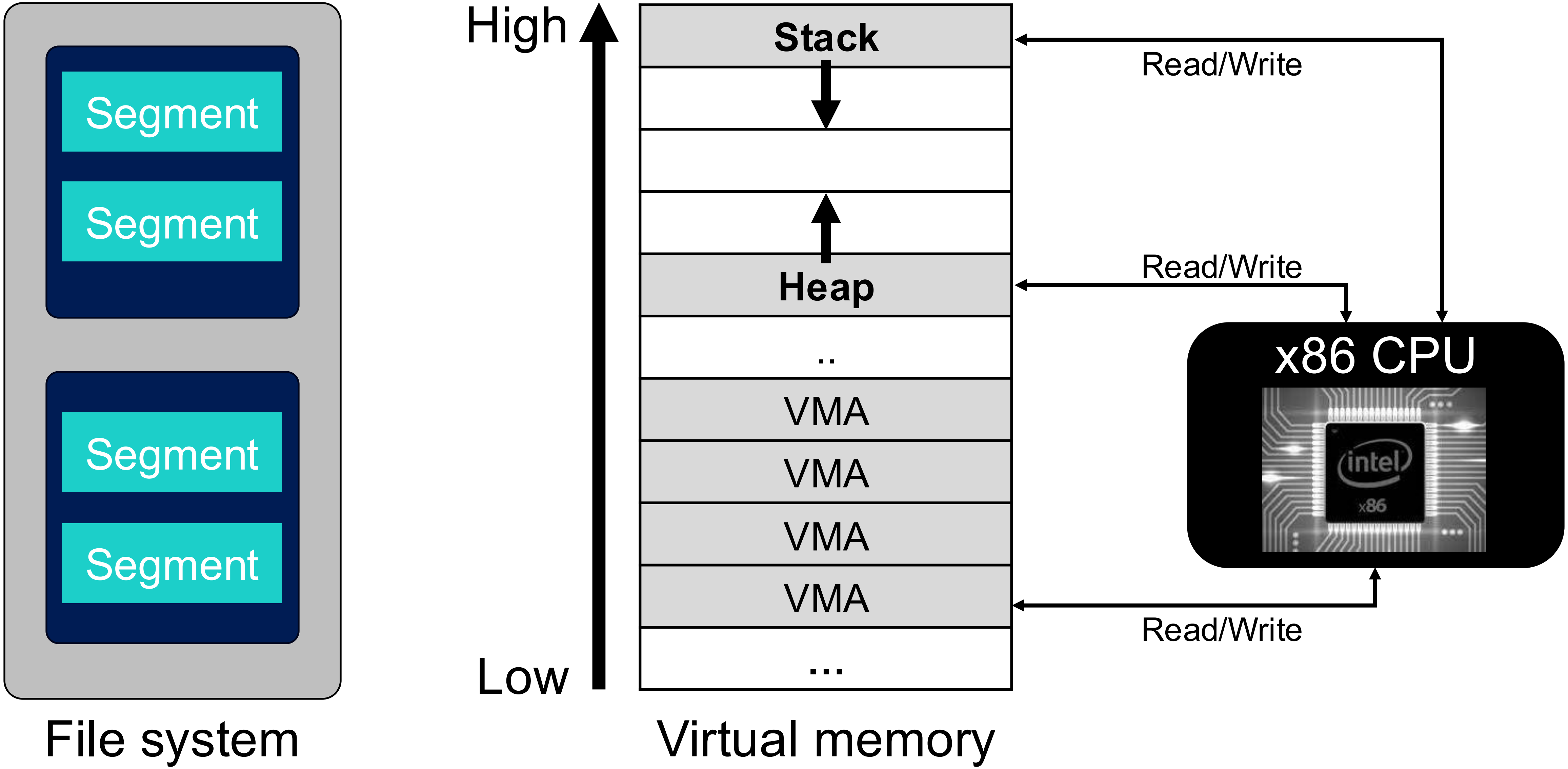
- (**8086**) 16-bit address space (in 1978)
- (**x86 or IA-32**) 32-bit address space (in 1985)
- (**x86-64 or x64 or AMD64**) 64-bit address space (in 2003)

This course will focus on x86 (IA-32) architecture

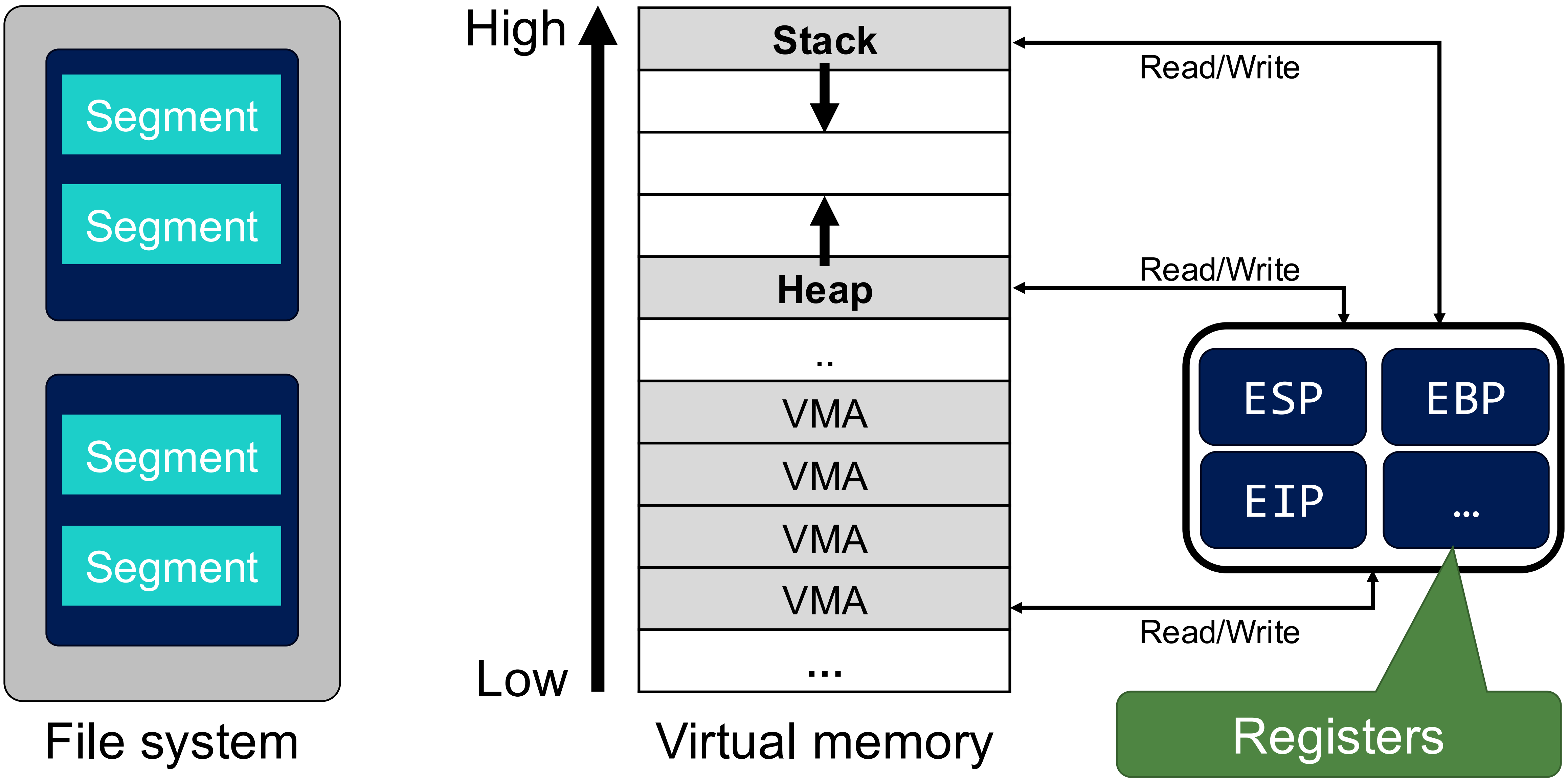
Memory Layout and CPU Registers



Memory Layout and CPU Registers



Memory Layout and CPU Registers



Registers in x86

- Program counter (instruction pointer)
  - **`EIP:`** points to the instruction to execute

- Stack pointers
  - **`ESP:`** points to the top of the stack
  - **`EBP:`** points to the base of the current stack frame

- Status register (FLAGS register)
  - **`EFLAGS:`** contains the current condition flags

- Other general purpose registers
  - **`EAX`**, **`EBX`**, **`ECX`**, **`EDX`**, **`ESI`**, **`EDI`**

All of them have a size of a ***double word*** (=32 bit)

Size of Registers

- x86 registers are 32-bit
- A word is the natural unit of data used by a processor
  - Typically, a word size is 32 bits on a 32-bit machine

- However, Intel defines a word is 16 bits on x86 (32-bit machine)

History of Intel/AMD Processors

- 1978: 8086
- 1982: 80286

16-bit processor,
Registers (`SP, BP, IP, ...`)

- 1985: 80386
- 1989: 80486
- ...

32-bit processor,
Registers (ESP, EBP, EIP, ...)

- 2003: Opteron
- 2005: Prescott
- 2006: Core 2
- 2008: Core i7
- …

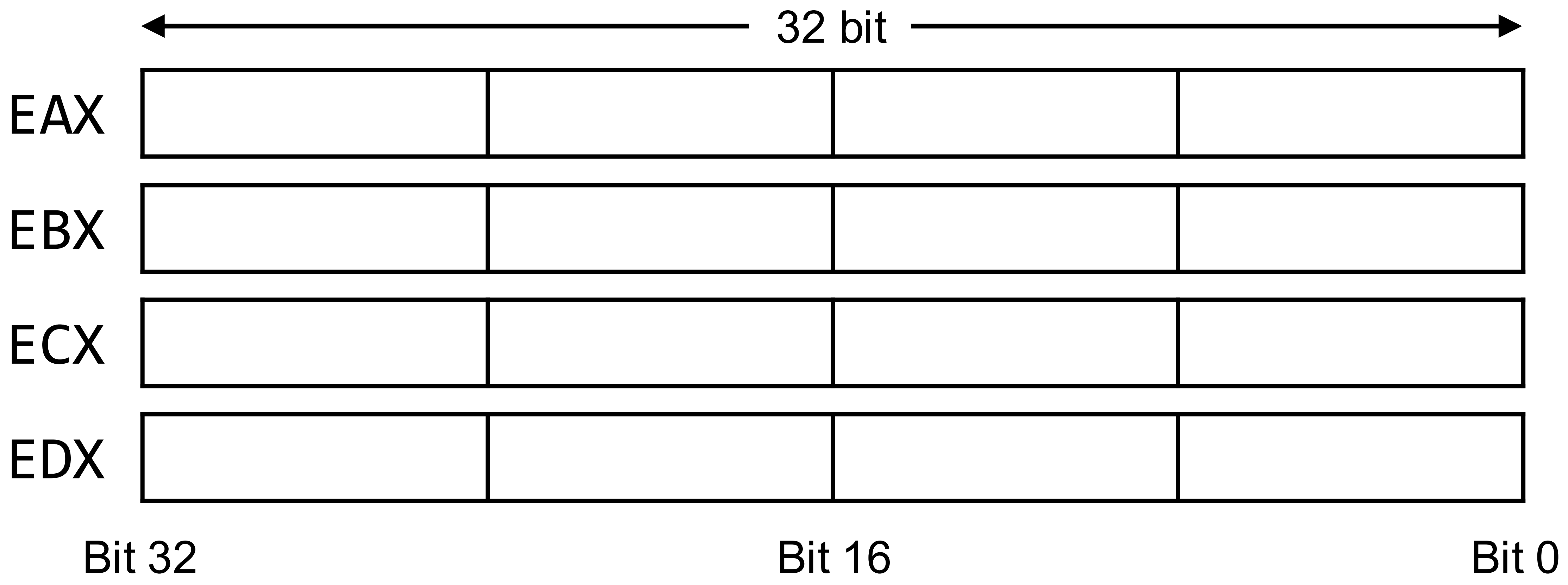
64-bit processor,
Registers (RSP, RBP, RIP, ...)

x86 Convention

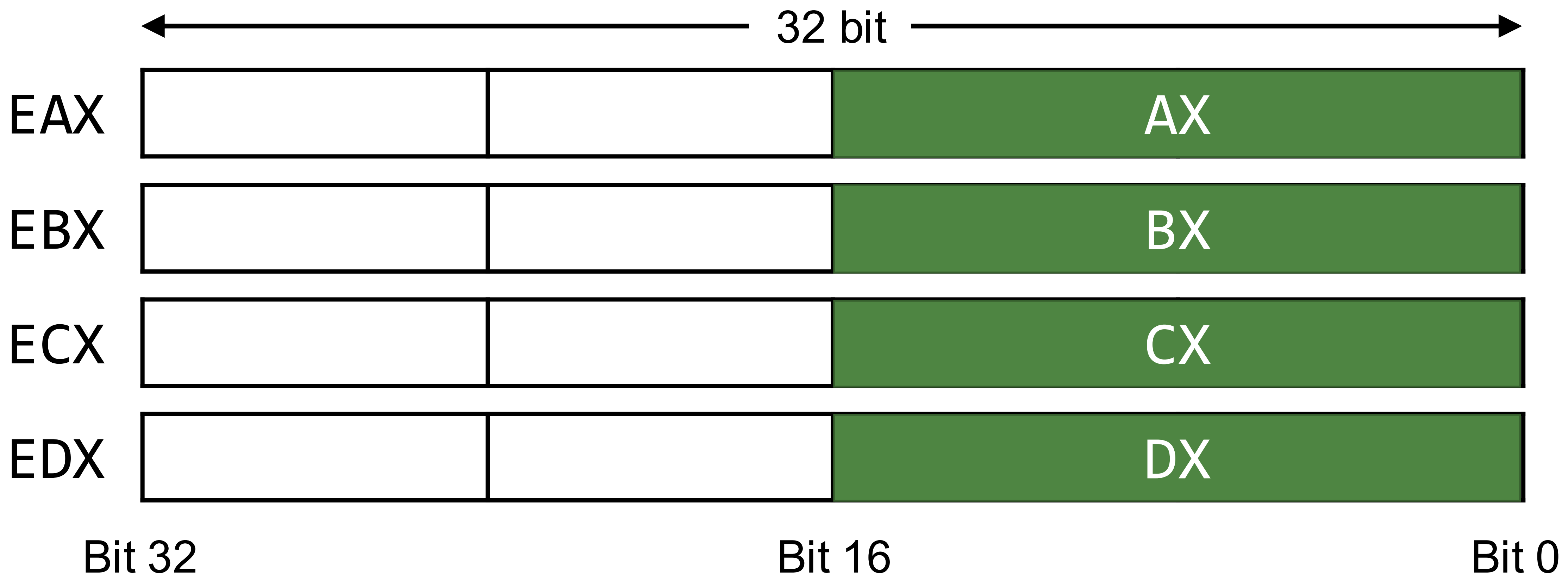
- Word = 16 bits
- Double Word (DWORD) = 32 bits

- Linear address space = 0 ~ 2^{32} bits

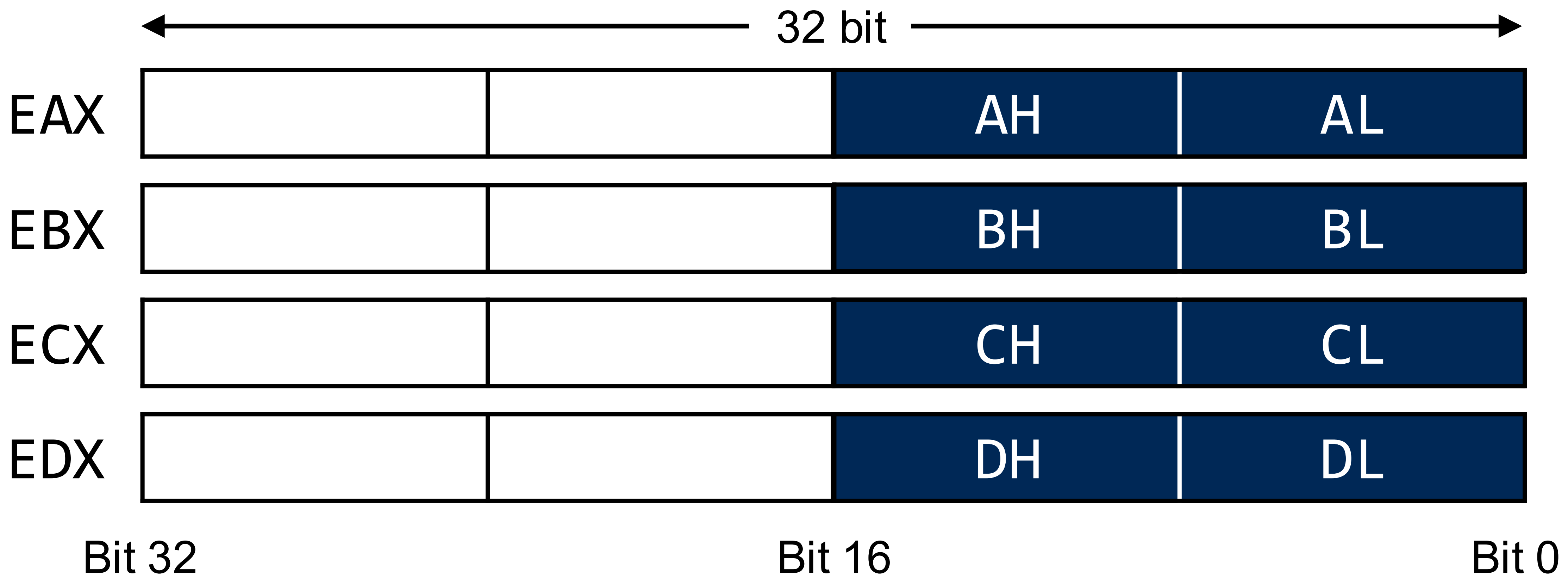
x86 Register Access



x86 Register Access



x86 Register Access



x86 Register Access

32 bit

EAX	11110000	00000000	00000000	00000111

AX	00000000	00000111
AH	00000000	
AL		00000111

x86 Memory Access = Byte Addressing

x86 Assembly Basics

Basic Formats

- Three formats of Instructions
  - 2 operands
  - 1 operands
  - 0 operands

Basic Format #1: Instructions with 2 Operands

```
mov eax, ebx
```

Opcode

Operand 1

Operand 2

Basic Format #2: Instructions with 1 Operand


```
inc eax
```

Opcode

Operand

Basic Format #3: Instructions with 0 Operand

```
ret
```

Opcode

Intel vs AT&T Format

- There are two ways to represent x86 assembly code

AT&T

```
mov %eax, %ebx
```

Intel

```
mov ebx, eax
```

We will use the Intel syntax

Recap: Compilation Process

Source code

Intermediate code

Assembly code

Binary code

test.ll

```
define dso_local i32 @test(i32) #0
{
  %2 = alloca i32, align 4
  store i32 %0, i32* %2, align 4
  ret i32 32
}
```

Intel syntax

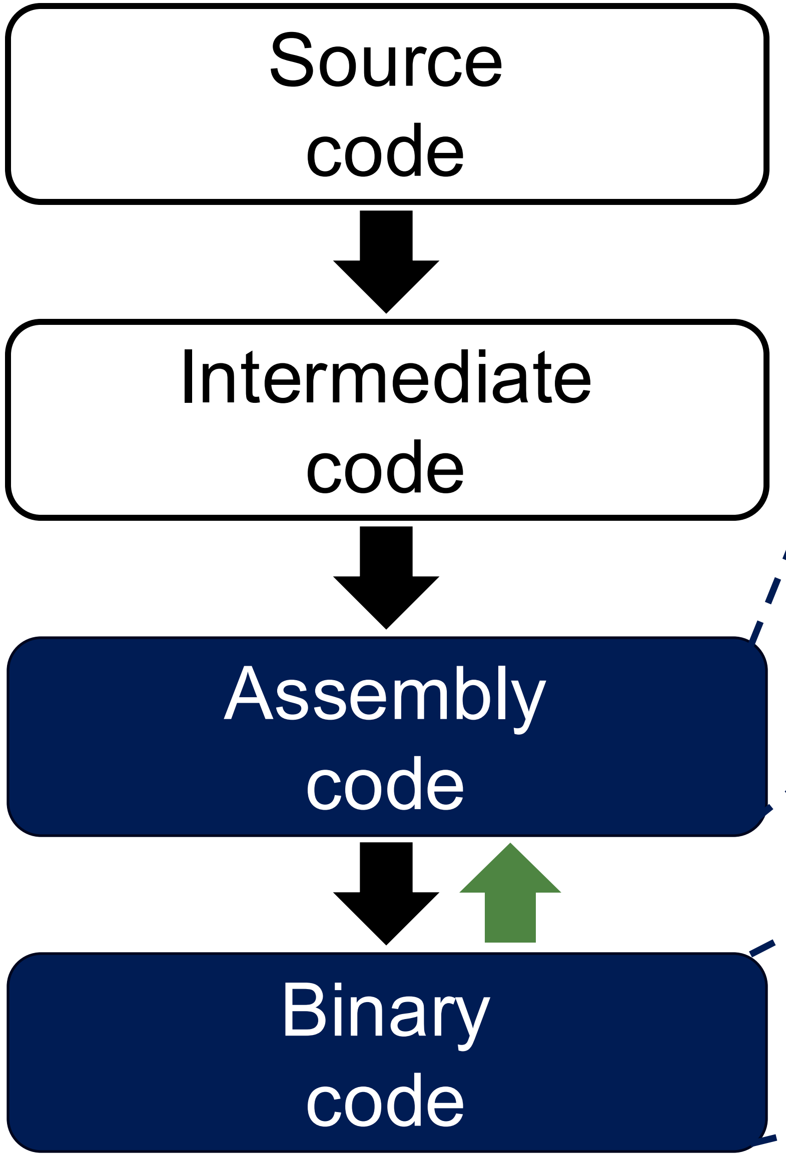
```
$llc --x86-asm-syntax=intel test.ll
```

test.s

```
test:                    # @test
# %bb.0:
push ebp
mov ebp, esp
mov eax, dword ptr [ebp + 8]
mov eax, 32
pop ebp
ret
```

The last human-readable format

Recap: GNU objdump




```
00000000 <test>:
 0: 55                push ebp
 1: 89 e5             mov  ebp,esp
 3: 8b 45 08          mov  eax,DWORD PTR [ebp+0x8]
 6: b8 20 00 00 00    mov  eax,0x20
 b: 5d                pop  ebp
 c: c3                ret
```

```
$ objdump -M intel -d test.o
```

Intel syntax

test.o

```
01000100101001010010101001010
10101001011110100100010010101
10111000101010001011100101001
```

Opcode Decides Semantics


```
mov eax, ebx
sub esp, 0x8
inc eax
```

`mov eax, ebx`	eax ← ebx
`sub esp, 0x8`	esp ← esp – 0x8
`inc eax`	eax ← eax + 1

Operand Types

Memory pointed by ebx

```
mov eax, [ebx]
```

Register

...	
0xc	...
0x8	0x4122a13c
0x4	0x273faace
0x0	0xbffa0220

Registers

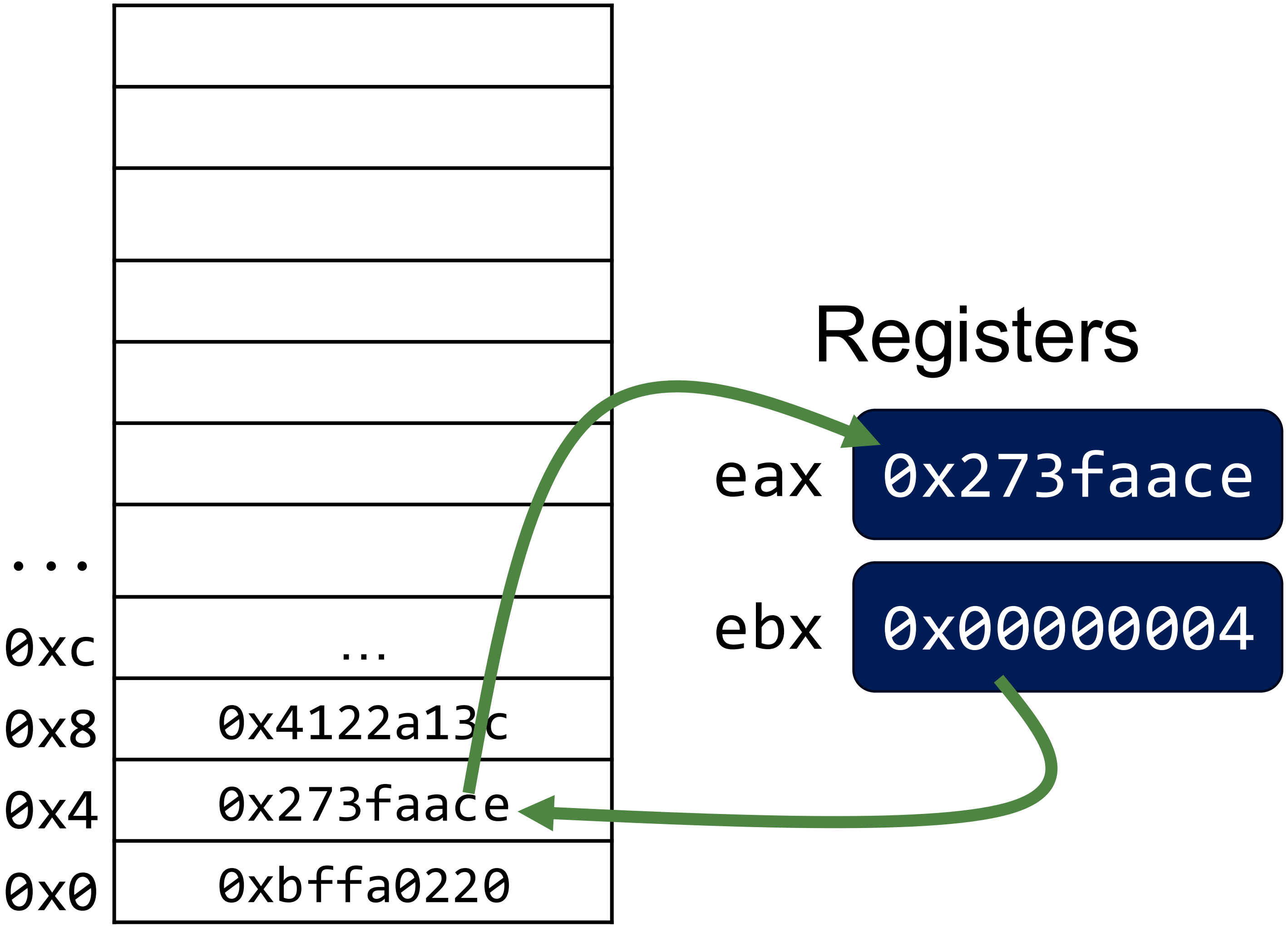
eax	
ebx	0x00000004

Operand Types

Memory pointed by ebx

```
mov eax, [ebx]
```

Register



Operand Types

```
sub esp, 0x8
```

Constant integer

esp `0x0000000a`

↓

esp `0x00000002`

Operand Types


```
mov cl, BYTE ptr [eax]
```

Pointer directive

Pointer Directive


```
mov [esi], al    ; ok
mov [esi], 1     ; error
```

Error: ambiguous operand size for 'mov'

Because it could be any of the followings

- `mov BYTE PTR [esi], 1`
- `mov WORD PTR [esi], 1`
- `mov DWORD PTR [esi], 1`
- `mov QWORD PTR [esi], 1`

Therefore, we need pointer directive ☺

Moving Data Around (mov)

- `mov eax, ebx`
- `mov al, bl`

Register to Register

- `mov [eax], ebx`

Register to Memory

- `mov eax, [ebx]`
- `mov eax, [ebx + edx * 4]`
- `mov al, BYTE PTR [esi]`

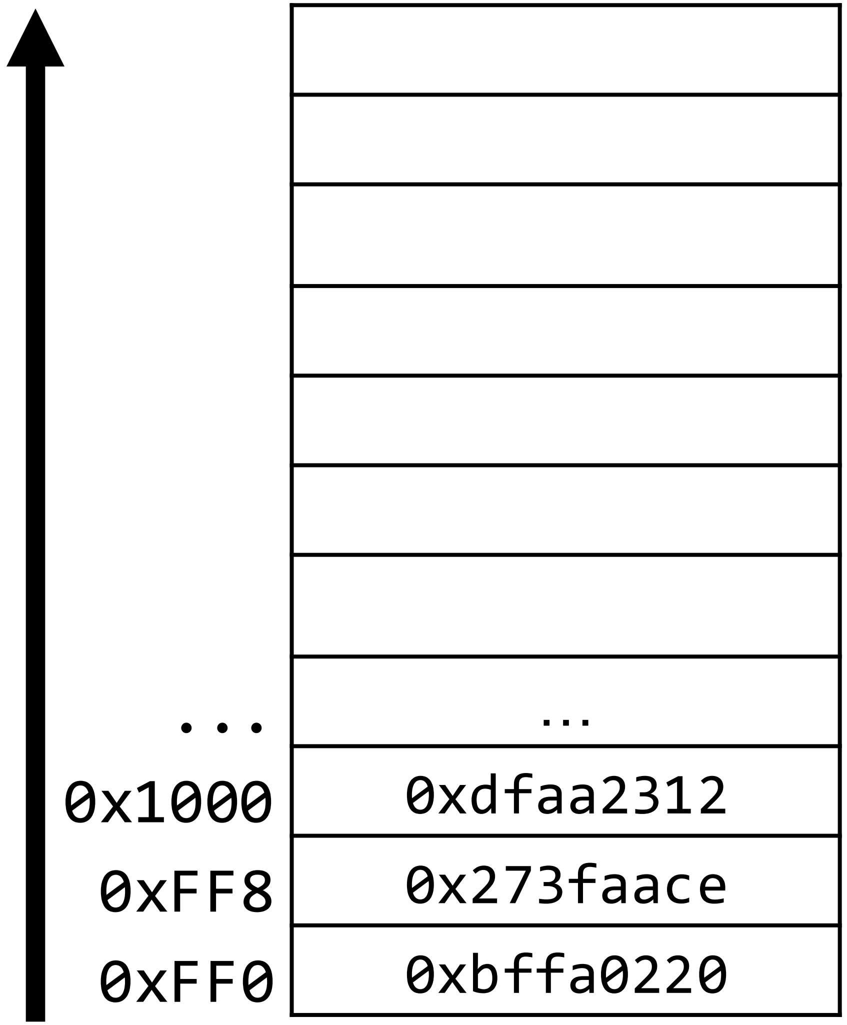
Memory to Register

- `mov eax, 42`
- `mov BYTE PTR [eax], 42`

Constant to Memory/Register

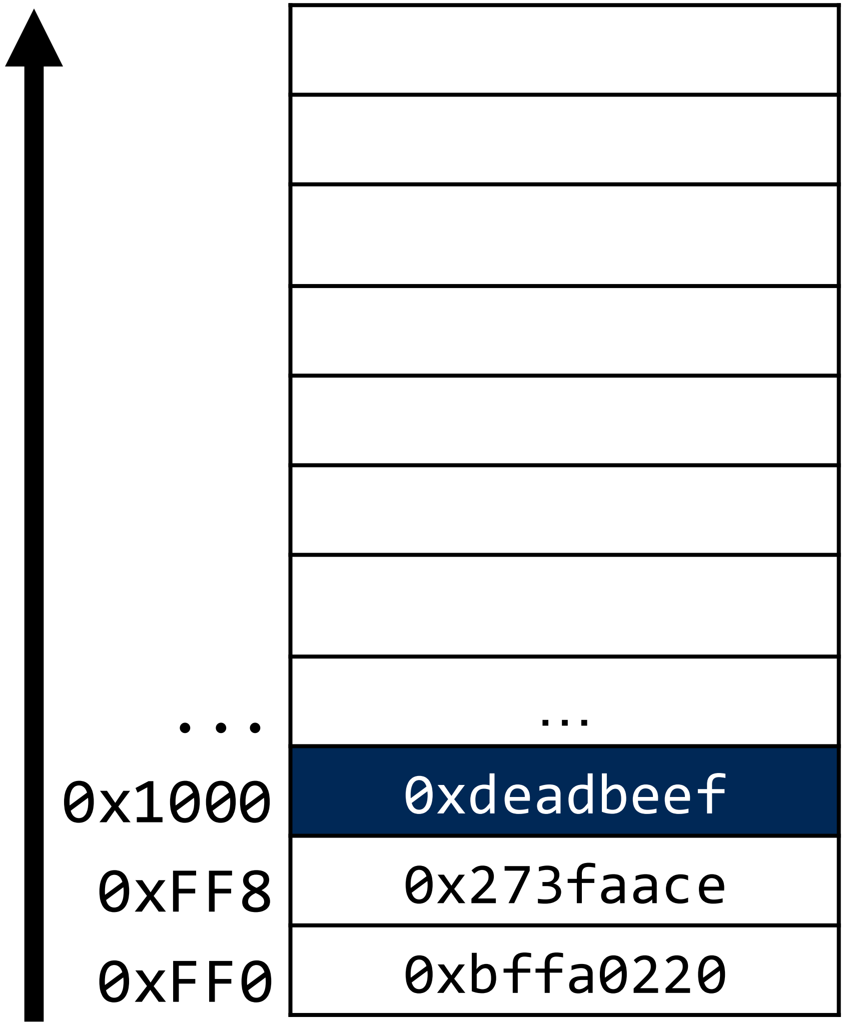
Example: Storing a DWORD in Memory


```
mov [eax], 0xdeadbeef      ; eax = 0x1000
```



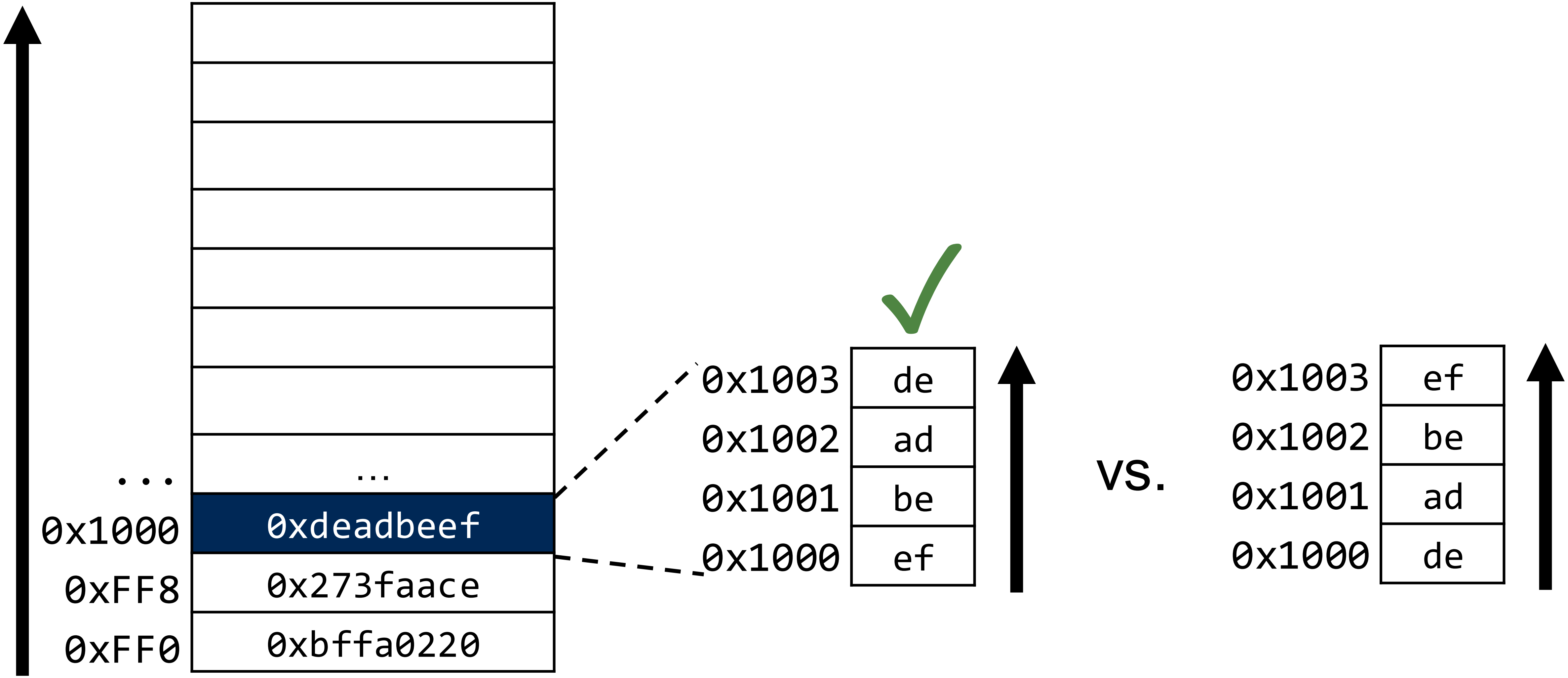
Example: Storing a DWORD in Memory

```
mov [eax], 0xdeadbeef      ; eax = 0x1000
```



Example: Storing a DWORD in Memory

```
mov [eax], 0xdeadbeef     ; eax = 0x1000
```



Endianness

- The order in which a sequence of bytes are stored in memory

- Big Endian = The MSB goes to the lowest address
- Little Endian = The LSB goes to the lowest address

x86 uses Little Endian

Exercise

```
mov cl, BYTE ptr [eax]
```

Pointer directive

...	
0xc	...
0x8	0x4122a13c
0x4	0x273faace
0x0	0xbffa0220

Registers

eax	0x00000009
ecx	0x13ef0c2e

What value will be after the instruction?

Addressing Modes

Specify how a memory operand is interpreted to derive an effective address

- **register**
  - ✓`mov eax, [eax]`
- **register + register**
  - ✓`mov eax, [eax + ebx]`
- **displacement**
  - ✓`mov eax, [0x1000]`
- **register + register × scale + displacement**
  - ✓`mov eax, [eax + ebx * 4 + 0x1000]`

Addressing Modes

Specify how a memory operand is interpreted to derive an effective address

$$\left[\left\{ \begin{matrix} \texttt{eax} \\ \texttt{ebx} \\ \texttt{ecx} \\ \texttt{edx} \\ \texttt{esp} \\ \texttt{ebp} \\ \texttt{esi} \\ \texttt{edi} \end{matrix} \right\} + \left\{ \begin{matrix} \texttt{eax} \\ \texttt{ebx} \\ \texttt{ecx} \\ \texttt{edx} \\ \texttt{ebp} \\ \texttt{esi} \\ \texttt{edi} \end{matrix} \right\} \times \left\{ \begin{matrix} 1 \\ 2 \\ 4 \\ 8 \end{matrix} \right\} + \begin{matrix} \texttt{displacement} \\ \text{(32-bit integer)} \end{matrix} \right]$$

Loading Address (lea)

```
lea eax, [ebx]
lea eax, [ebp-0x8]
```

Memory address to Register

What is the Difference?

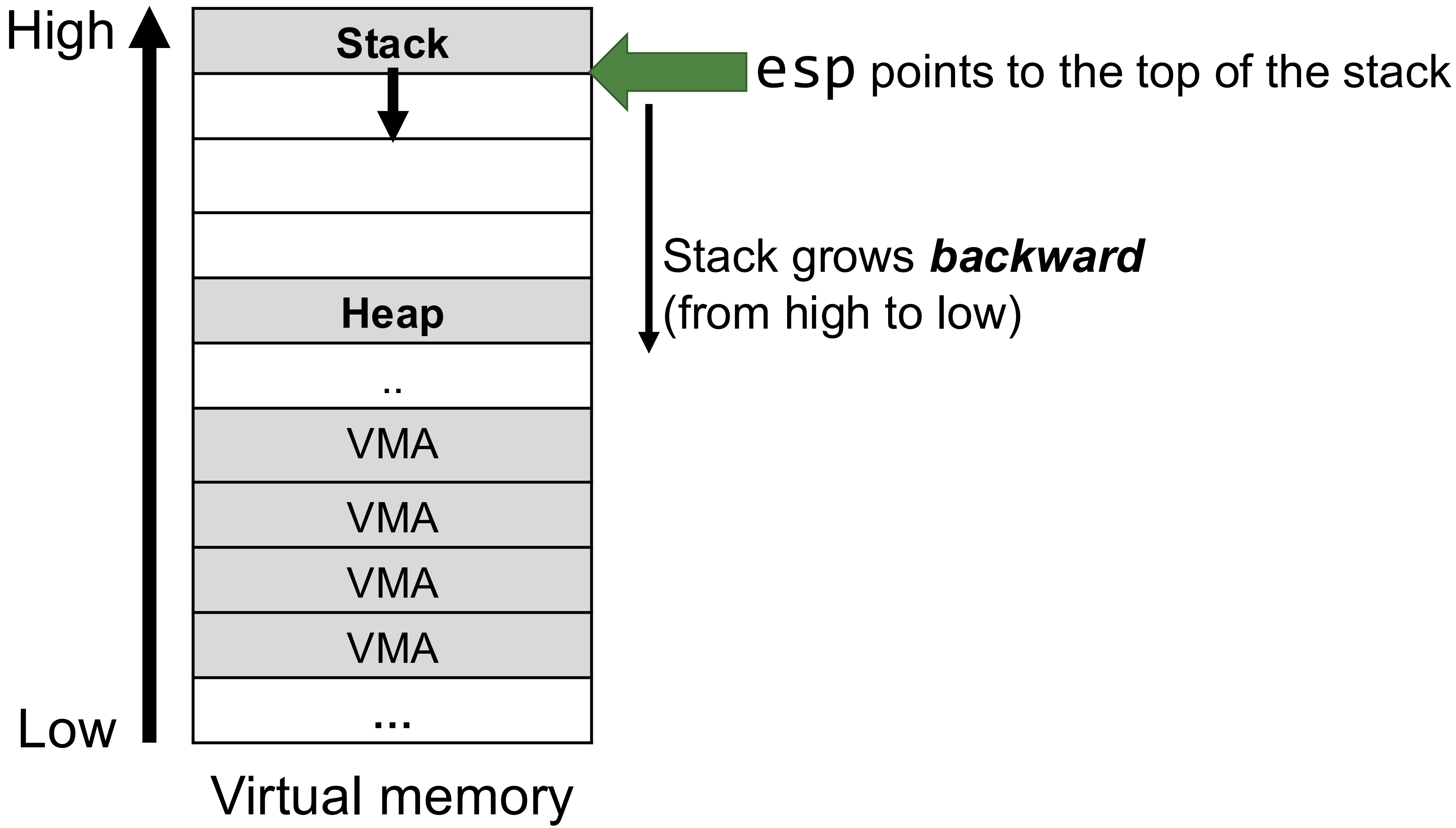

```
mov eax, [ebp + 0x10]
         vs.
lea eax, [ebp + 0x10]
```

```
eax ← *(ebp + 0x10)
        vs.
eax ← (ebp + 0x10)
```

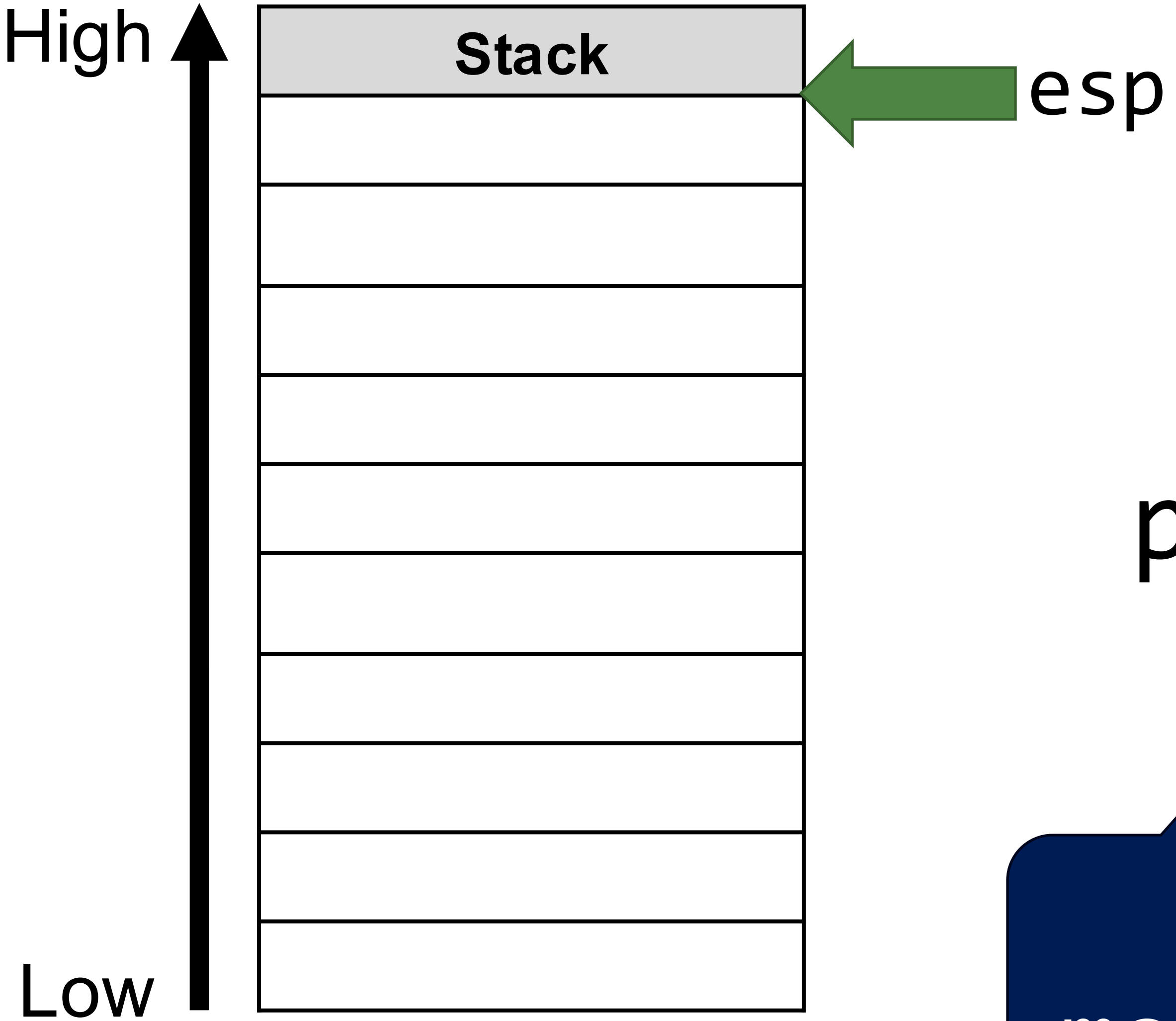
Stack Memory

- Stack stores data in a LIFO (Last-In-First-Out) fashion. When a function is invoked, a new ***stack frame*** is allocated at the top of the stack memory

Stack Operations



Stack Operations – Push and Pop

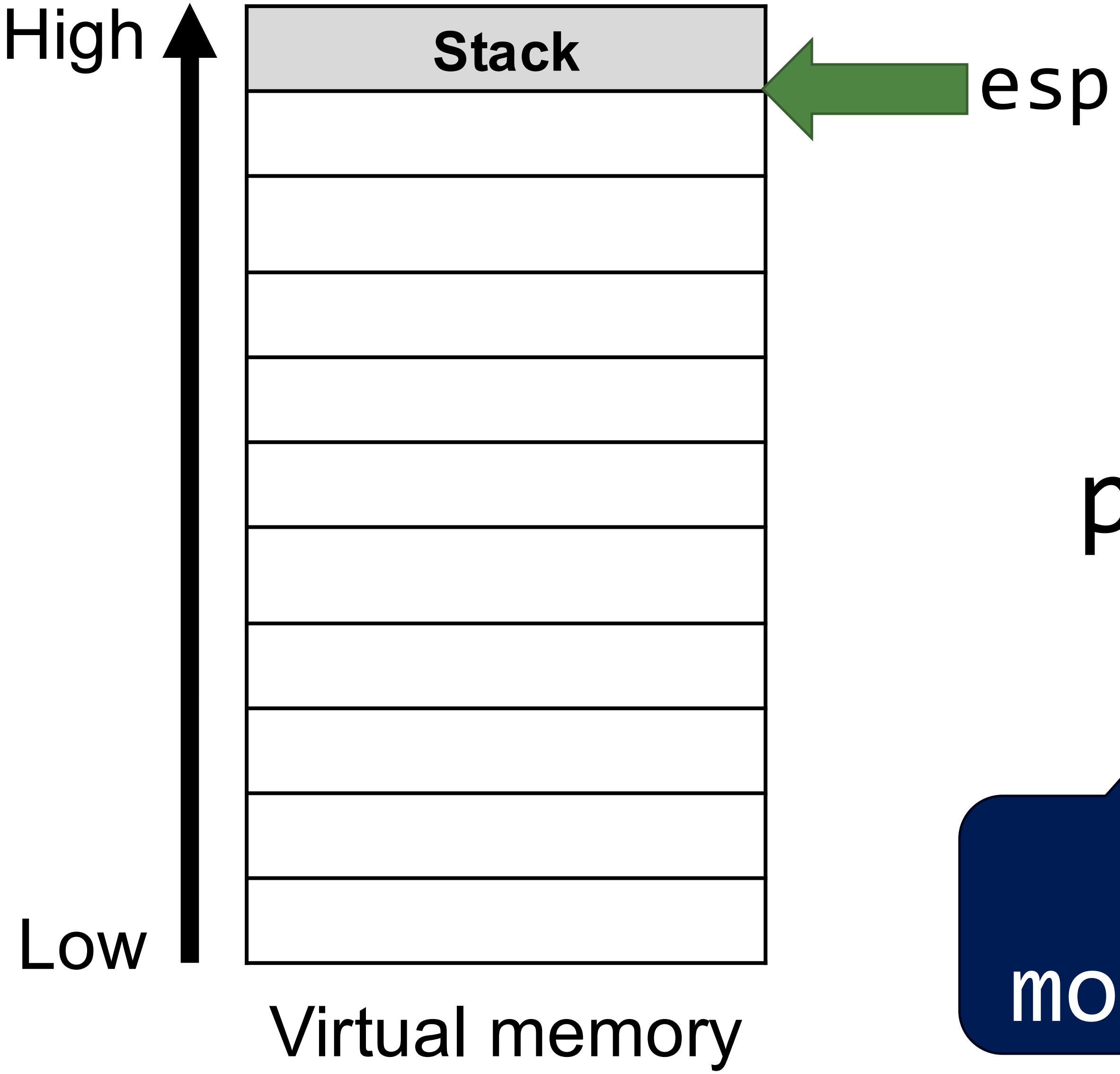


Virtual memory

```
push eax
```

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

Stack Operations – Push and Pop



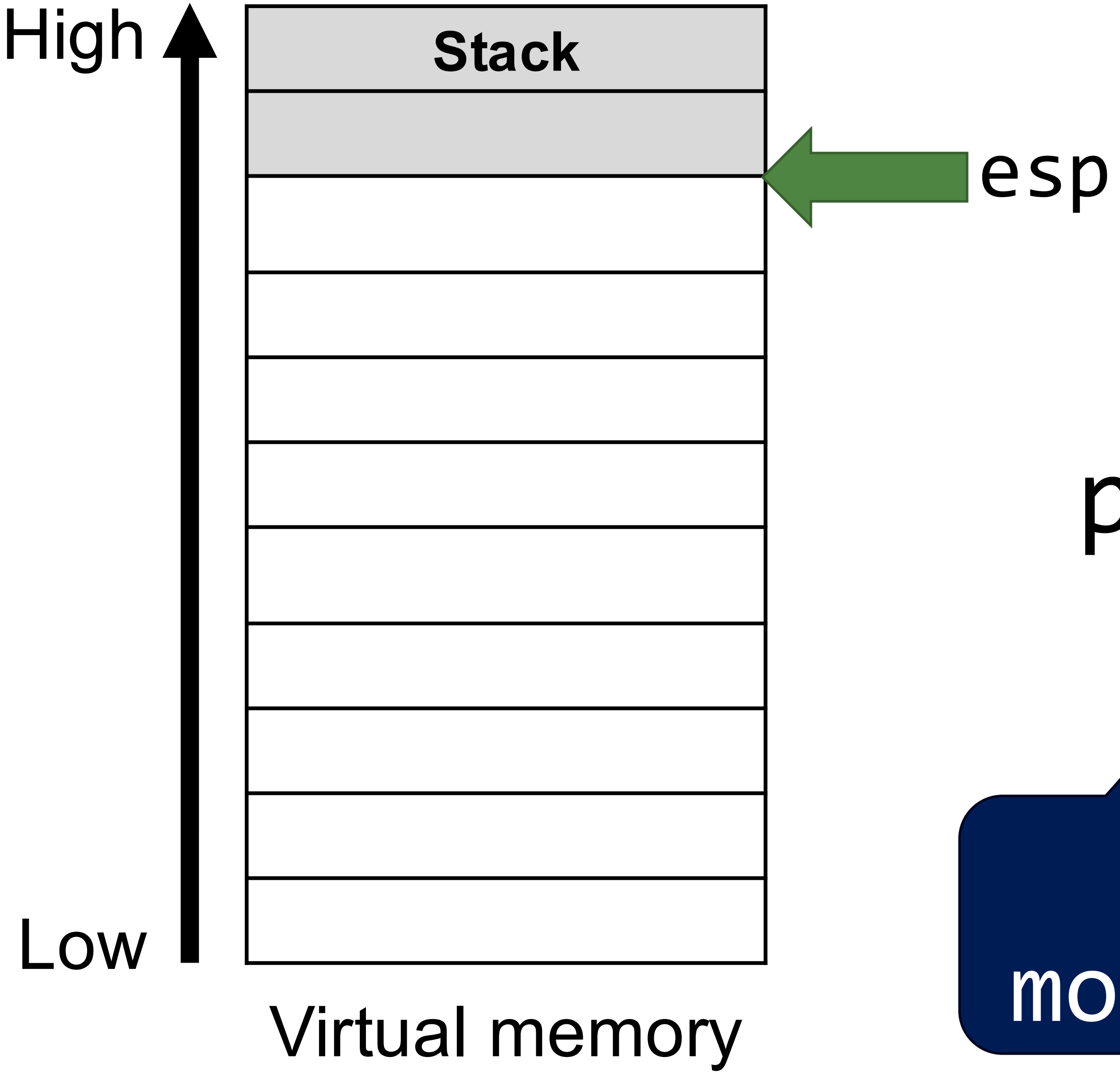

```
push eax
```

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

Registers

eax	0x00000009

Stack Operations – Push and Pop



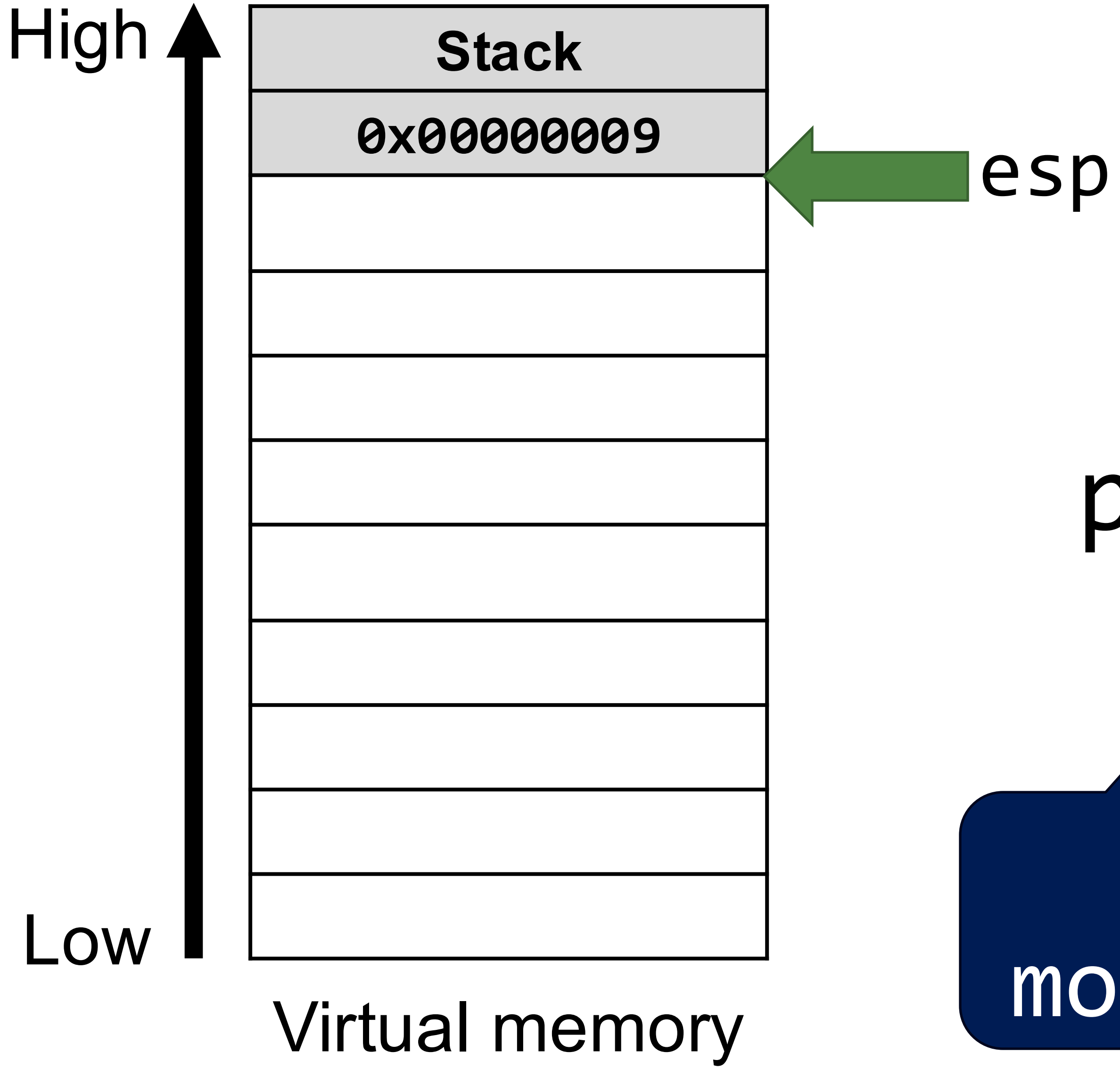

```
push eax
```

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

Registers

eax	0x00000009

Stack Operations – Push and Pop



push eax

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

Registers

eax 0x00000009

Stack Operations (push)

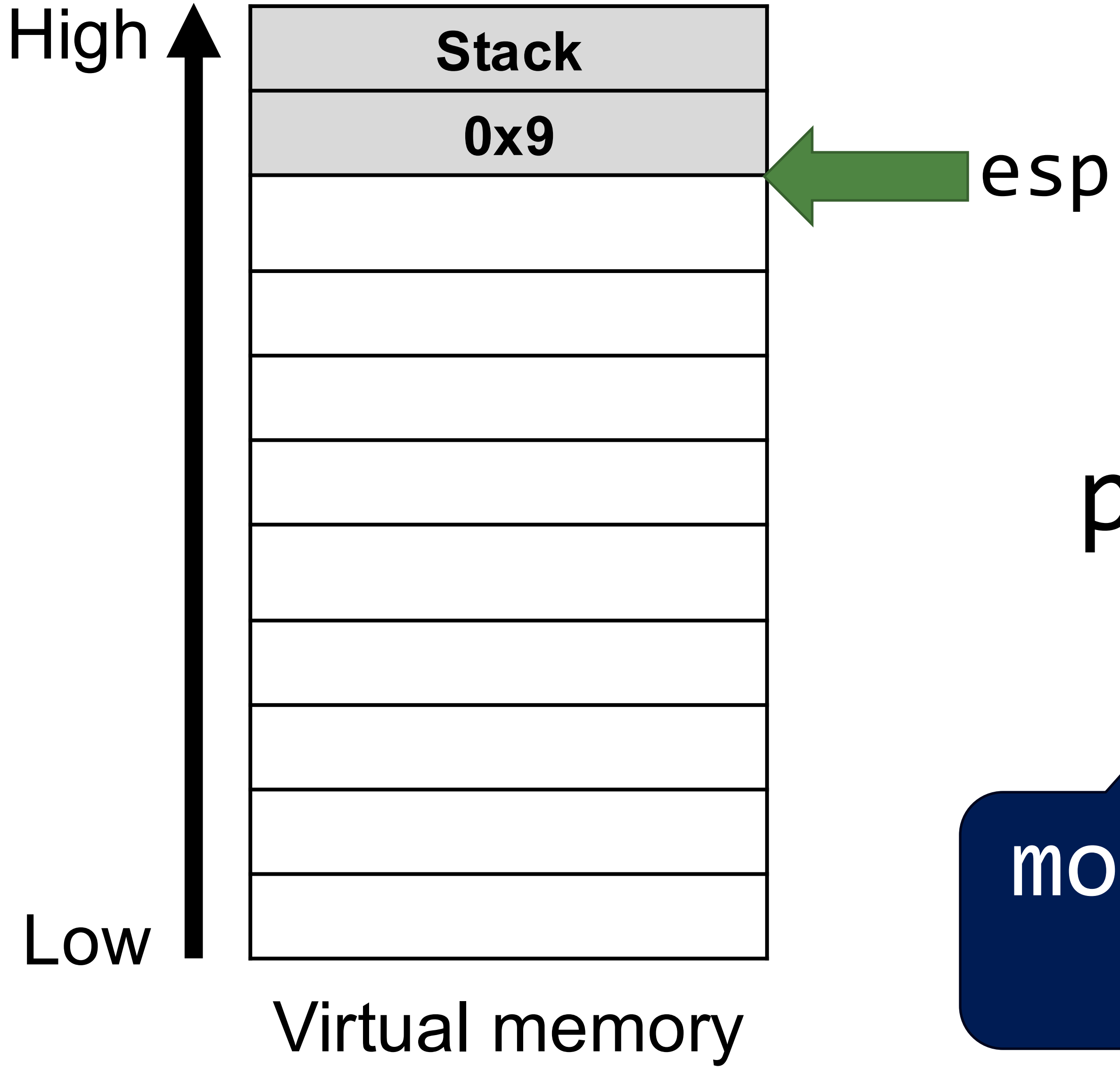
`push eax`	Push register on the stack
`push 0x42`	Push constant on the stack
`push [eax]`	Push a value at the memory address on the stack

```
push x
```

=

```
sub esp, 4
mov [esp], x
```

Stack Operations – Push and Pop




```
pop ecx
```

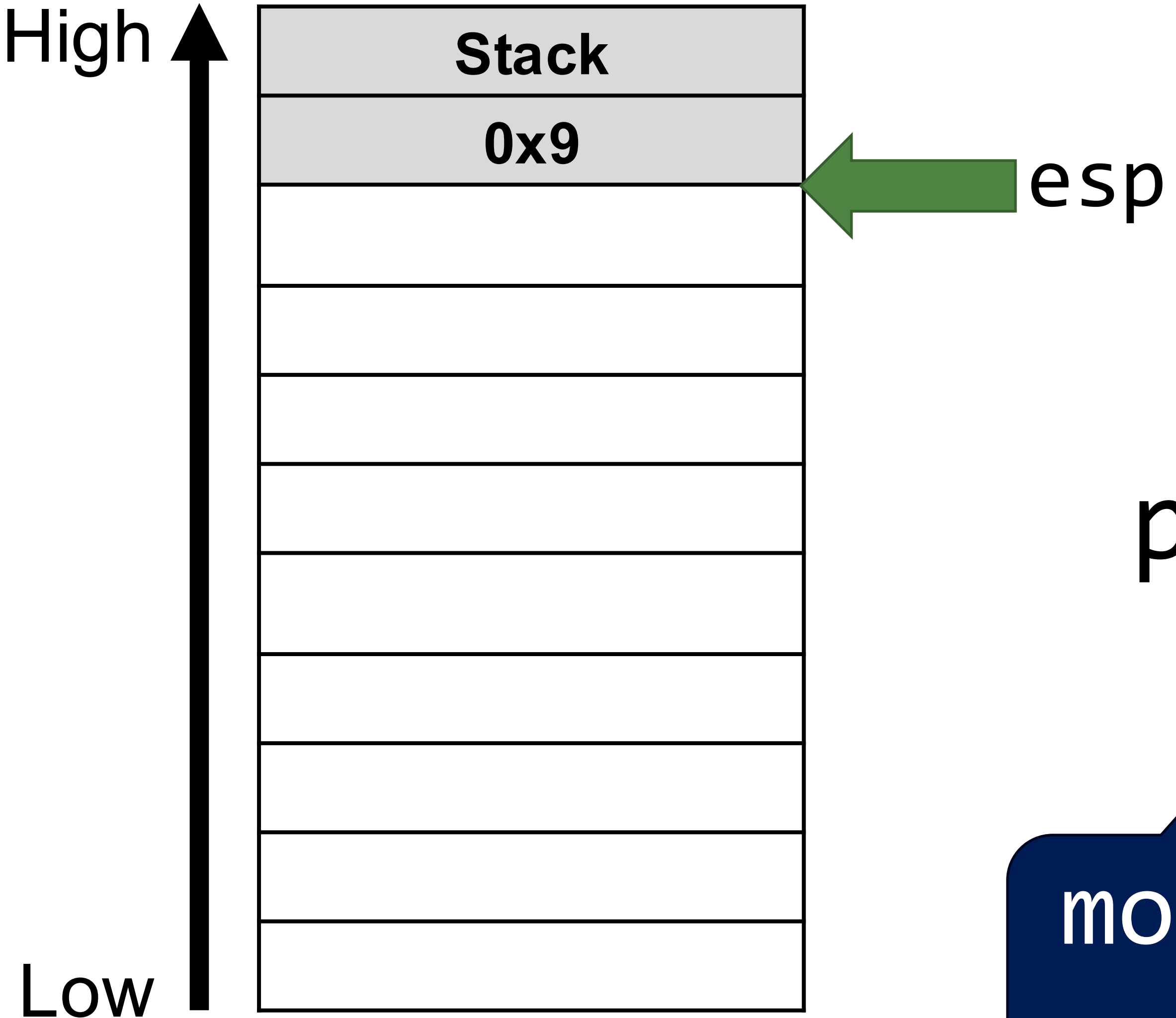
```
mov ecx, [esp]
add esp, 4
```

Registers

ecx `0x13ef0ca1`

(before)

Stack Operations – Push and Pop



Virtual memory

```
pop ecx
```

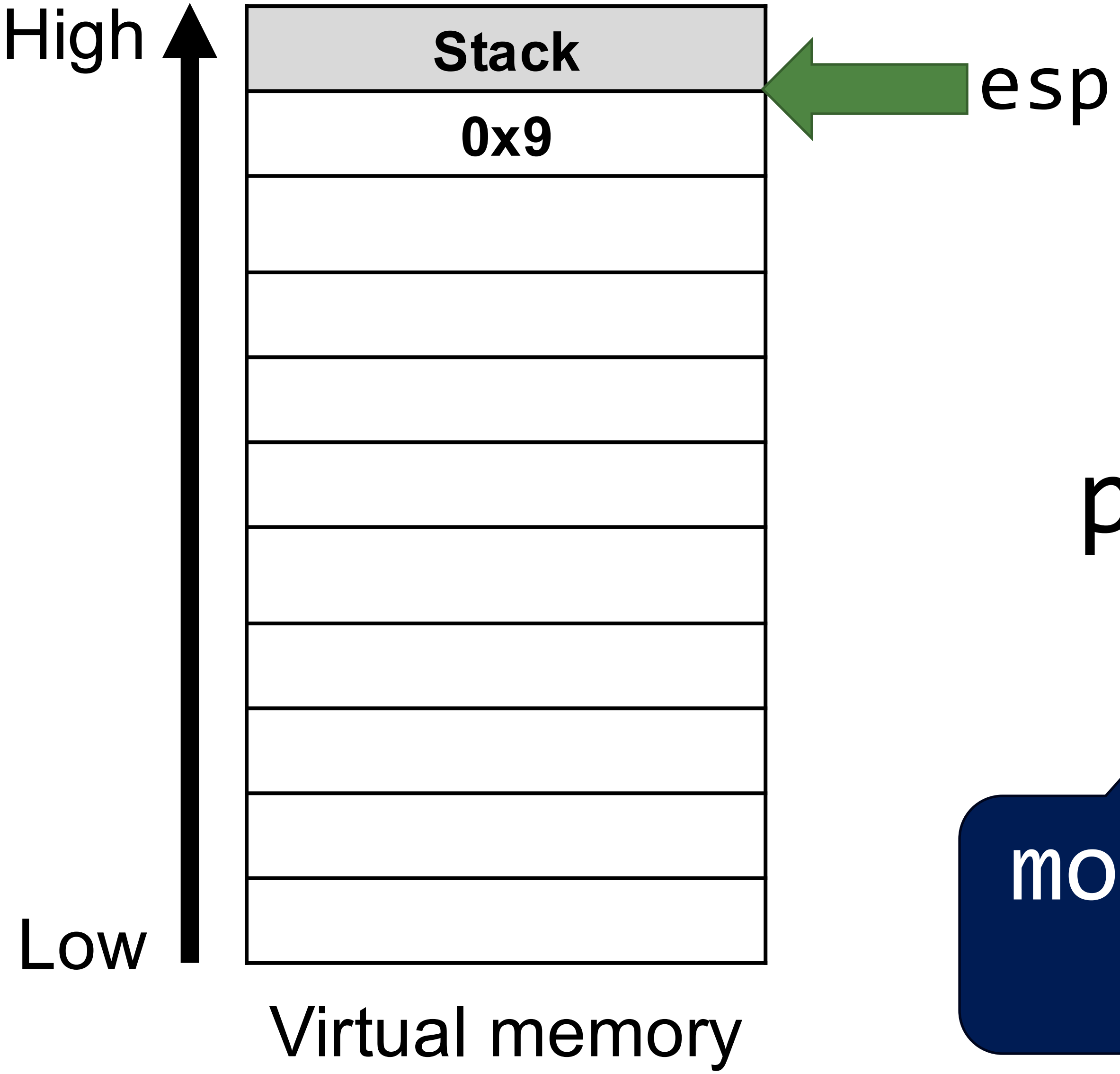
```
mov ecx, [esp]
add esp, 4
```

Registers

ecx `0x00000009`

(after)

Stack Operations – Push and Pop




```
pop ecx
```

```
mov ecx, [esp]
add esp, 4
```

Registers

Stack Operations (pop)

`pop eax`	Pop the top element of the stack into register
`pop [eax]`	Pop the top element of the stack into the memory address

```
pop x
```

=

```
mov x, [esp]
add esp, 4
```

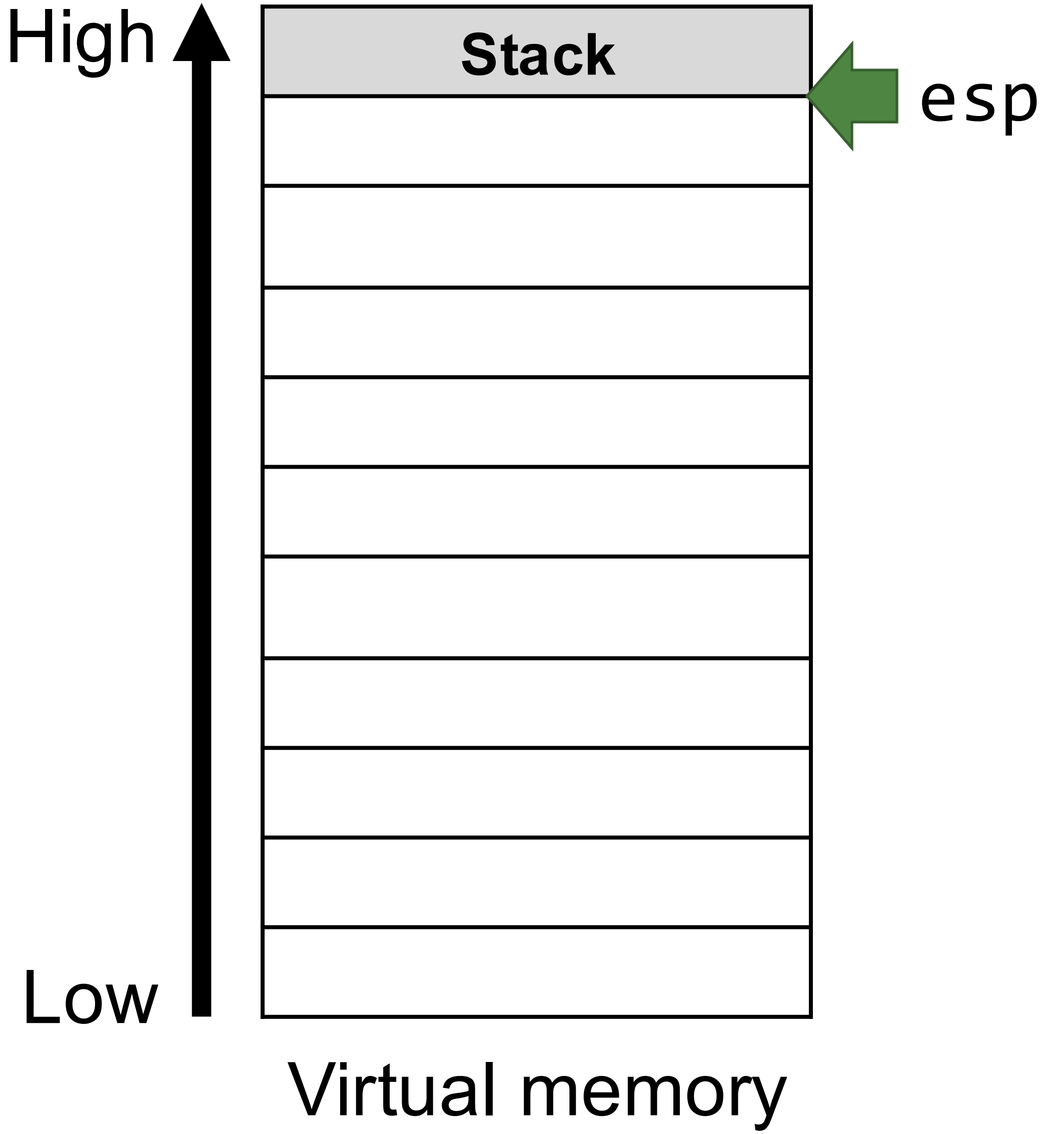
Stack Operations (leave)

```
leave = mov esp, ebp
        pop ebp
```

Function Call (call)


```
    …
    call foo
nextret:
    nop
    nop
    ...
foo:
    nop
    nop
```

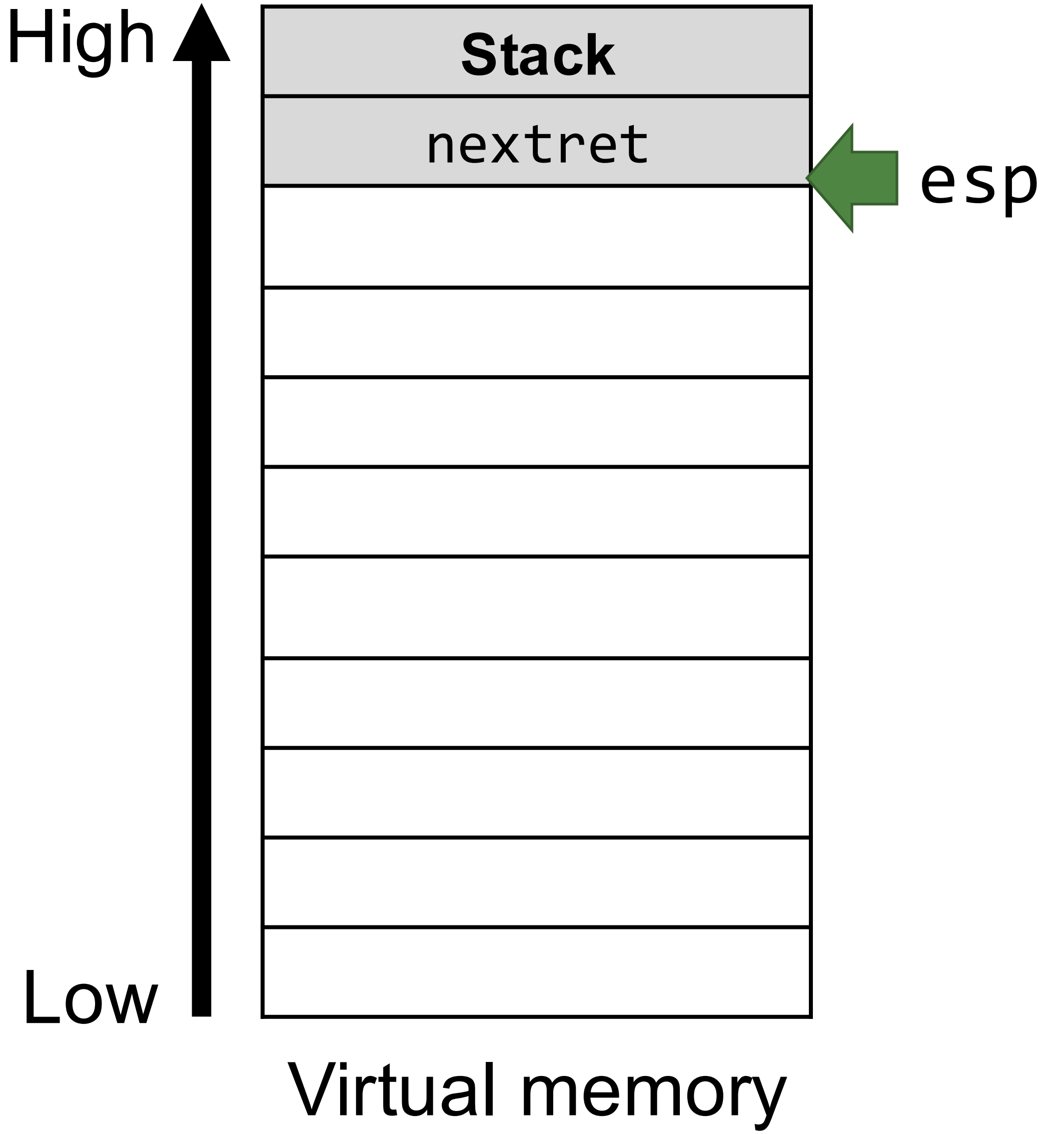
```
push nextret
jmp foo
```



Function Call (call)

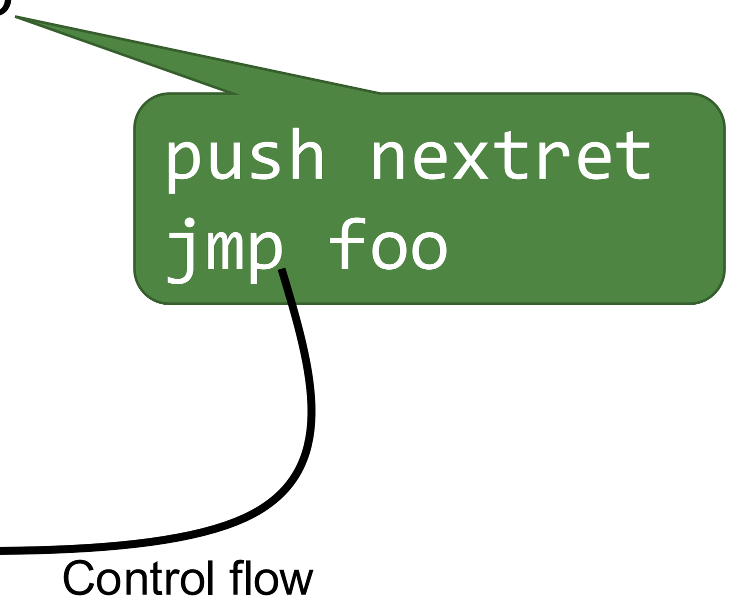
```
    …
    call foo
nextret:
    nop
    nop
    ...
foo:
    nop
    nop
```

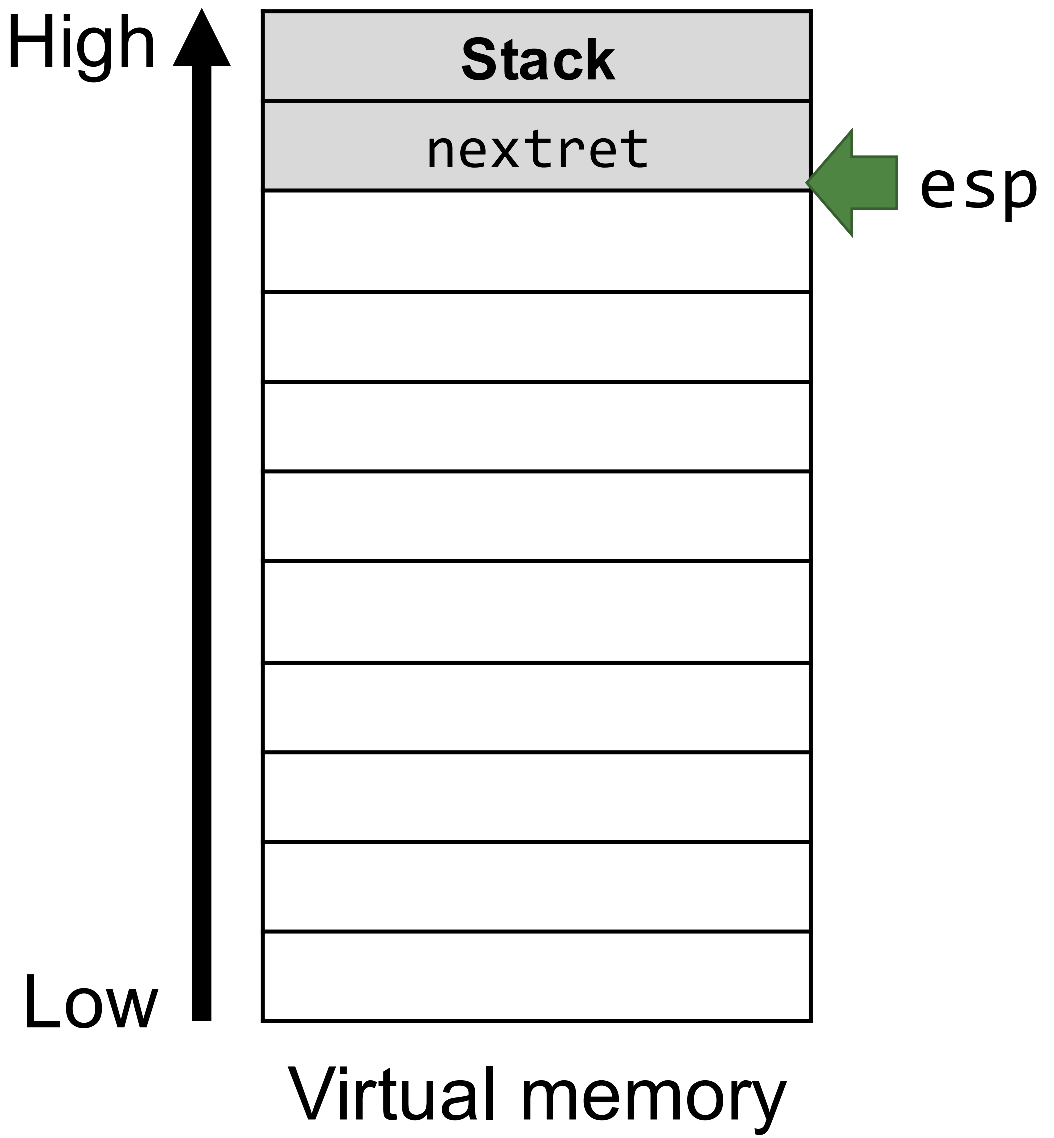
```
push nextret
jmp foo
```



Function Call (call)

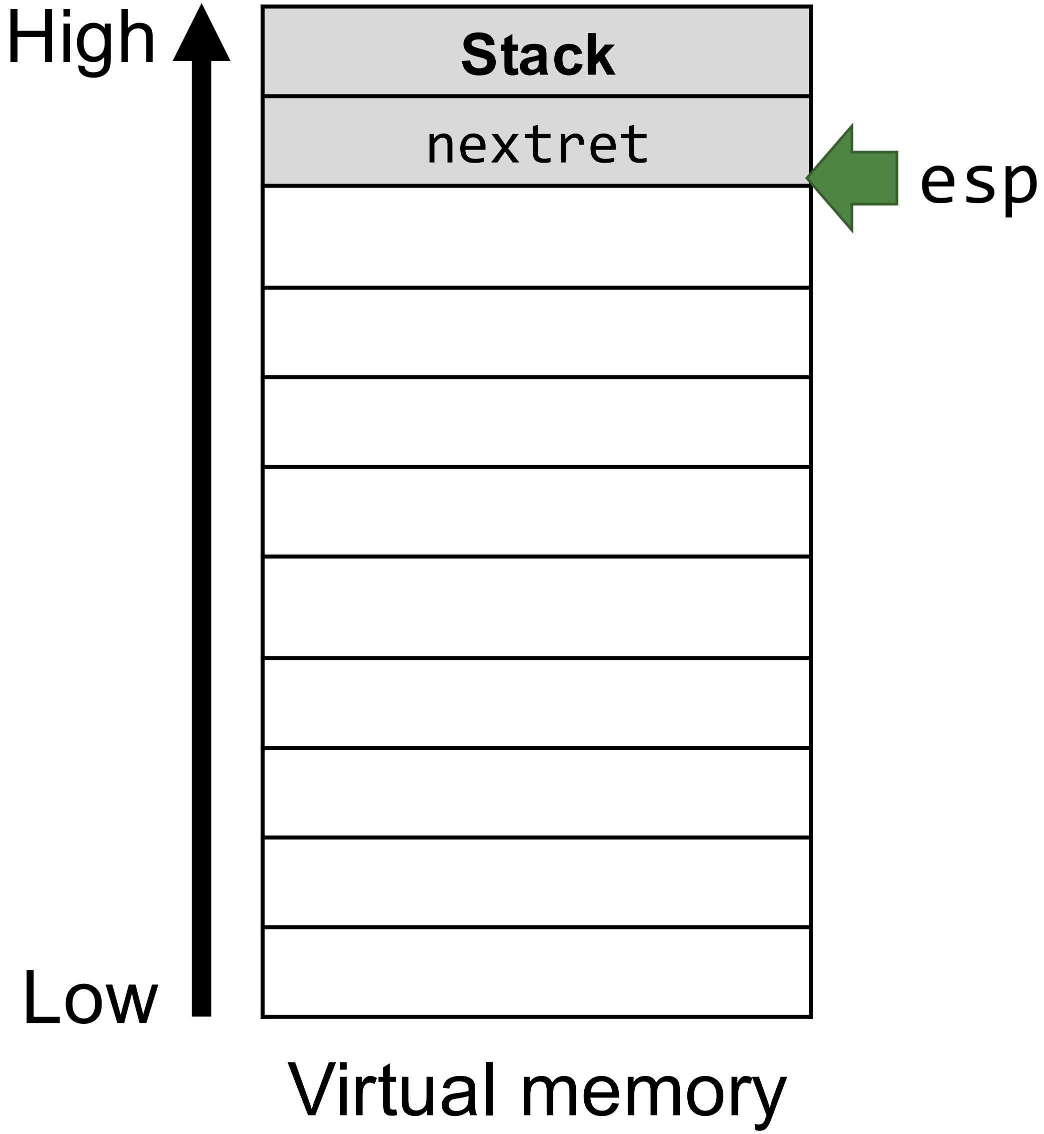

```
    …
    call foo
nextret:
    nop
    nop
    ...
foo:
    nop
    nop
```





Function Return (ret)

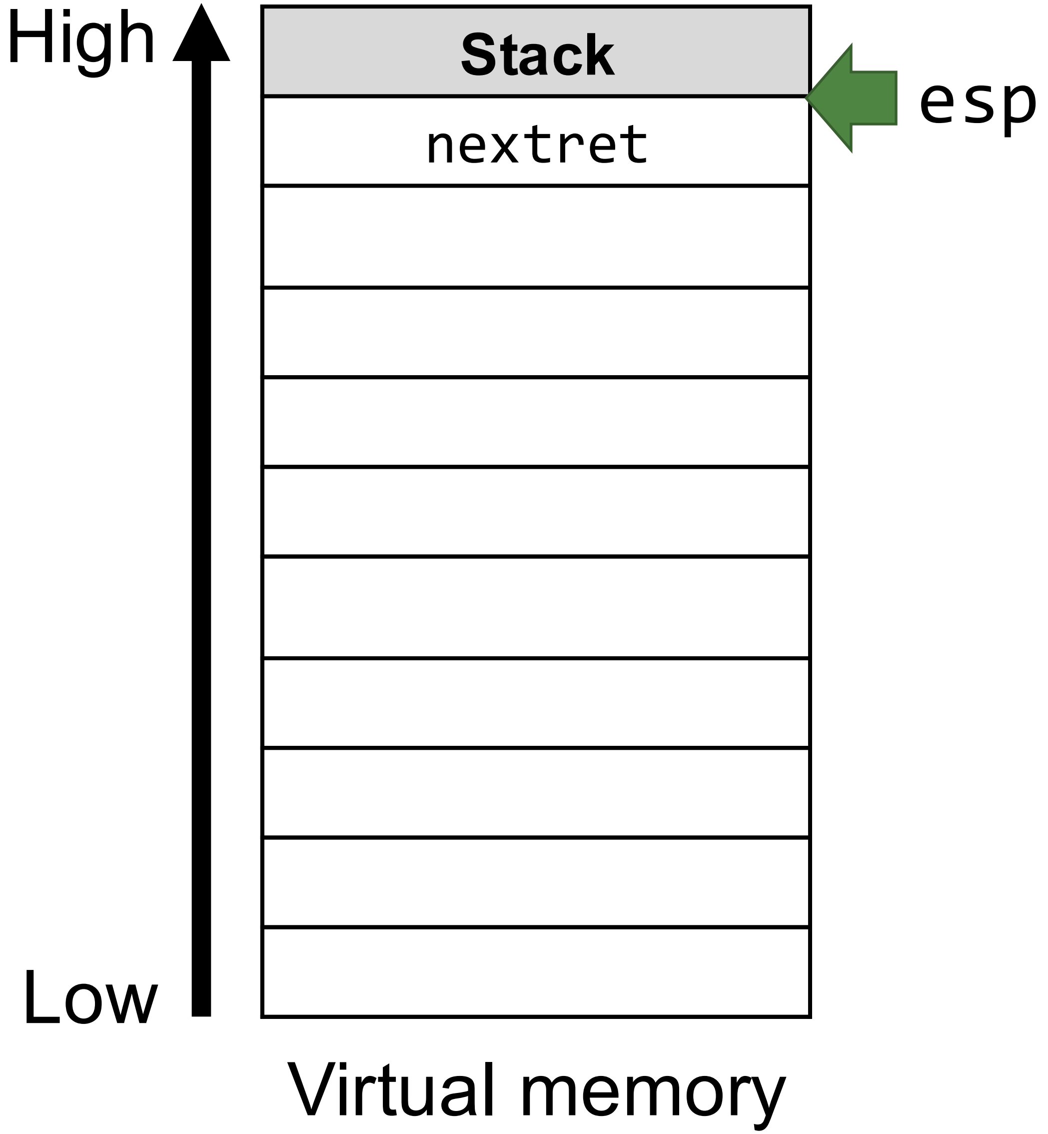
`ret` = `pop eip`



Function Return (ret)

ret = pop eip

eip nextret



Arithmetic and Logical Operations

- `add eax, [ebx]`
- `sub esp, 0x40`
- `inc ecx`
- `dec edx`
- `and [eax + ecx], ebx`
- `xor edx, ebx`
- `shl eax, 1`
- ...

Control Flows


```
if ( x ) {
   /* A */
}
else {
   /* B */
}
```

```
while (x) {
}
```

```
for (i = 0; i < n ; i++) {
}
```

How to represent in assembly?

Control Flows in Assembly (1)

- There are only "if" and "goto" (no "else")

```
if ( x ) {
  /* A */
}
else {
  /* B */
}
```

How assembly looks like…

```
  if (!x) goto F;
  /* A */
  goto DONE;
F:
  /* B */
DONE:
```

Control Flows in Assembly (2)

- There are only “if” and “goto” (no “else”)

```
while (x) {
    /* body */
}
```

How assembly looks like…

```
WHILE:
   if (!x) goto DONE;
   /* body */
   goto WHILE;
DONE:
```

Control Flows in Assembly (3)

- There are only “if” and “goto” (no “else”)

```
for (i = 0; i < n ; i++) {
  /* body */
}
```

How assembly looks like…

```
   i = 0;
LOOP:
   if (i >= n) goto DONE;
   /* body */
   i++;
   goto LOOP;
DONE:
```

Control Flows in Assembly (Example)

```
if (!x) goto F;
```

```
cmp x, 0
je F
```

Test if x is zero

If x=zero then goto F

Control Flows in Assembly (Example)

```
if (!x) goto F;
```

→

```
cmp x, 0
je F
```

```
if (i >= n) goto F;
```

→

```
cmp i, n
jge F
```

Test if `i >= n`

If `i >= n` then goto F

Control Flows in Assembly (Example)

Where do we store the result of comparison (cmp)?

```
if (!x) goto F;
```

→

```
cmp x, 0
je F
```

```
if (i >= n) goto F;
```

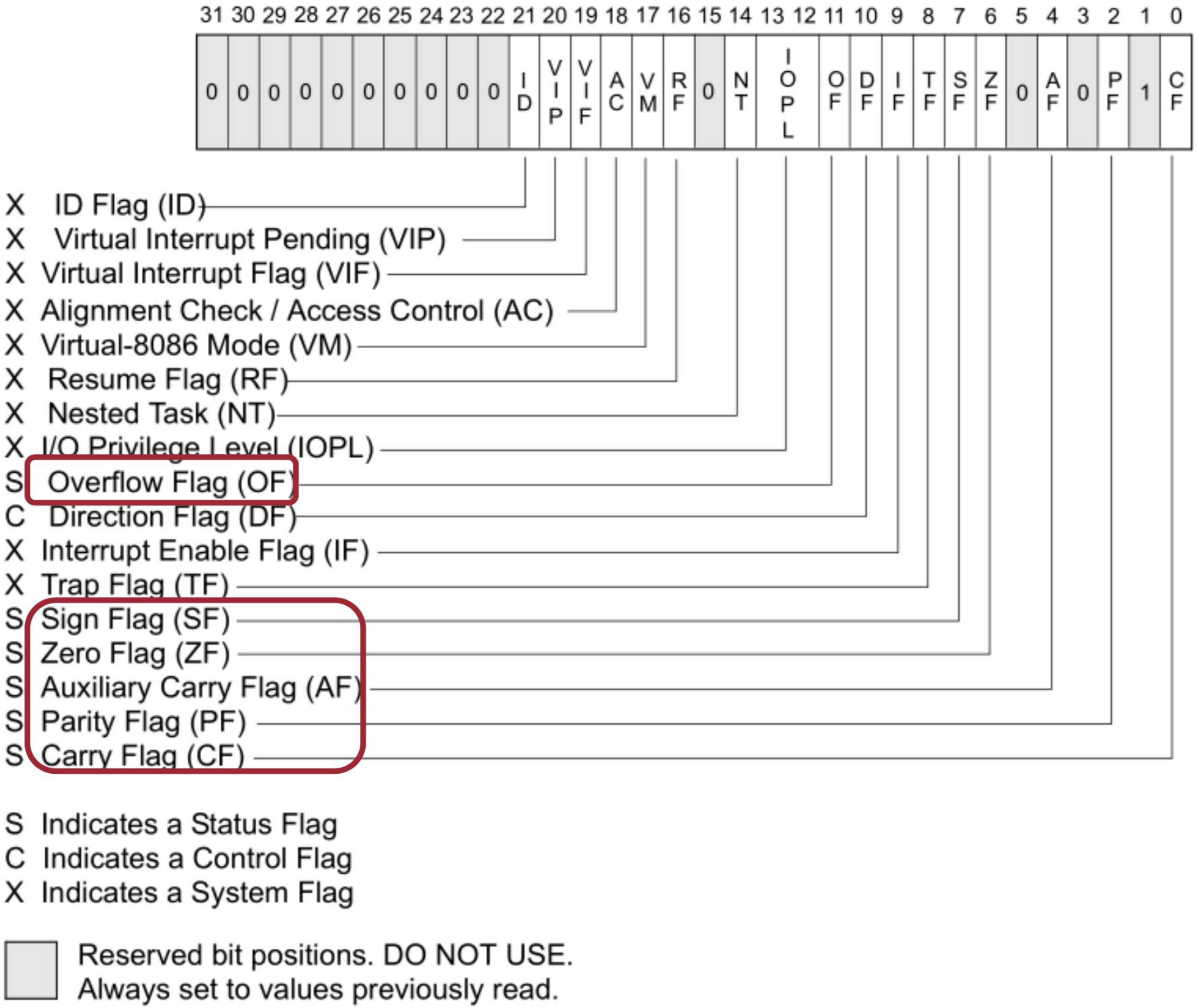
→

```
cmp i, n
jge F
```

EFLAGS: Storing the Processor State

- EFLAGS is a status register used in x86, which is essentially a collection of status flag bits

- There are approximately 20 different flag bits used in x86, but we are mainly interested in 6 condition flags:
  - OF: Overflow flag
  - SF: Sign flag
  - ZF: Zero flag
  - AF: Auxiliary carry flag
  - PF: Parity flag
  - CF: Carry flag


31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 ID VIP VIF AC VM RF 0 NT IOPL OF DF IF TF SF ZF 0 AF 0 PF 1 CF
X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check / Access Control (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
S Overflow Flag (OF)
C Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)
S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag
Reserved bit positions. DO NOT USE.
Always set to values previously read.

`cmp` Only Affects EFLAGS

- `cmp` is the same as `sub`, except that it only affects EFLAGS, but not the destination operand. For example, `cmp eax, ebx` will not change the `eax` register

Branch Instructions

Assume that a comparison instruction (`cmp`) precedes the branch instruction

Branch Instruction	Condition	Description
`ja`	`CF = 0 and ZF = 0`	Jump if above
`jb`	`CF = 1`	Jump if below
`je`	`ZF = 1`	Jump if equal
`jl`	`SF ≠ F`	Jump if less
`jle`	`ZF = 1 or SF ≠ F`	Jump if less or equal
`jna`	`CF = 1 or ZF = 1`	Jump if not above
`jnb`	`CF = 0`	Jump if not below
`jz`	`ZF = 1`	Jump if zero
… (many more)		

Summary So Far

- We learned how to move around data
  - `mov`, `lea`, `push`, `pop`, etc.
- We learned how to perform arithmetic and logical operations
  - `add`, `sub`, `and`, `or`, etc.
- We also learned how to control program flows
  - `cmp`, `jmp`, `ja`, `jz`, etc.

Already Turing Complete!

x86 Execution Model

Recap: Stack Frame

- When a function is invoked, a new ***stack frame*** is allocated at the top of the stack memory

- Also, called as procedure frame or activation record

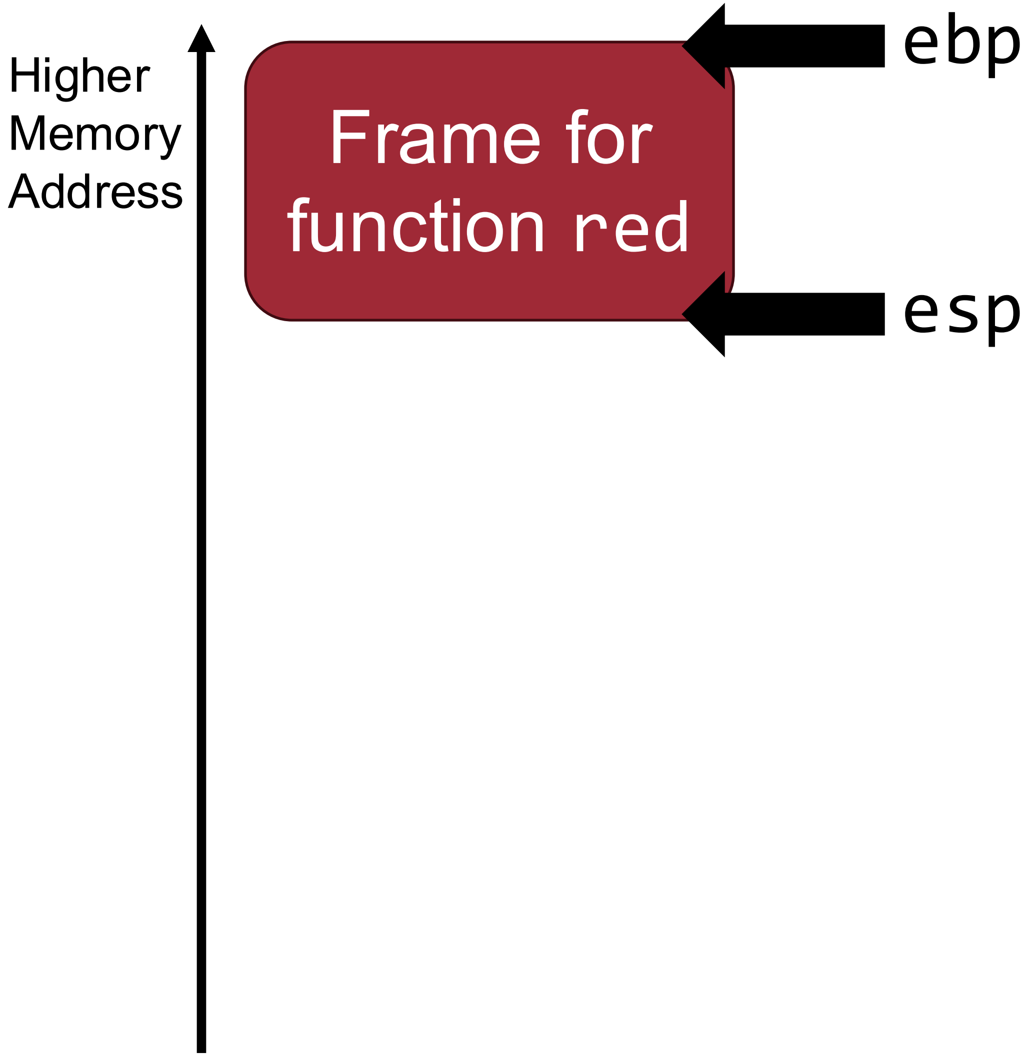
Our Example


```
int purple(int a1, int a2) {
    return 2 + a1 - a2;
}

int blue(int a1) {
    return 1 + purple(a1, b);
}

int red(int a1) {
    return blue(a1 - 42);
}
```

Our Example – Stack


```
int purple(int a1, int a2) {
    return 2 + a1 - a2;
}

int blue(int a1) {
    return 1 + purple(a1, b);
}

int red(int a1) {
    return blue(a1 - 42);
}
```

Start

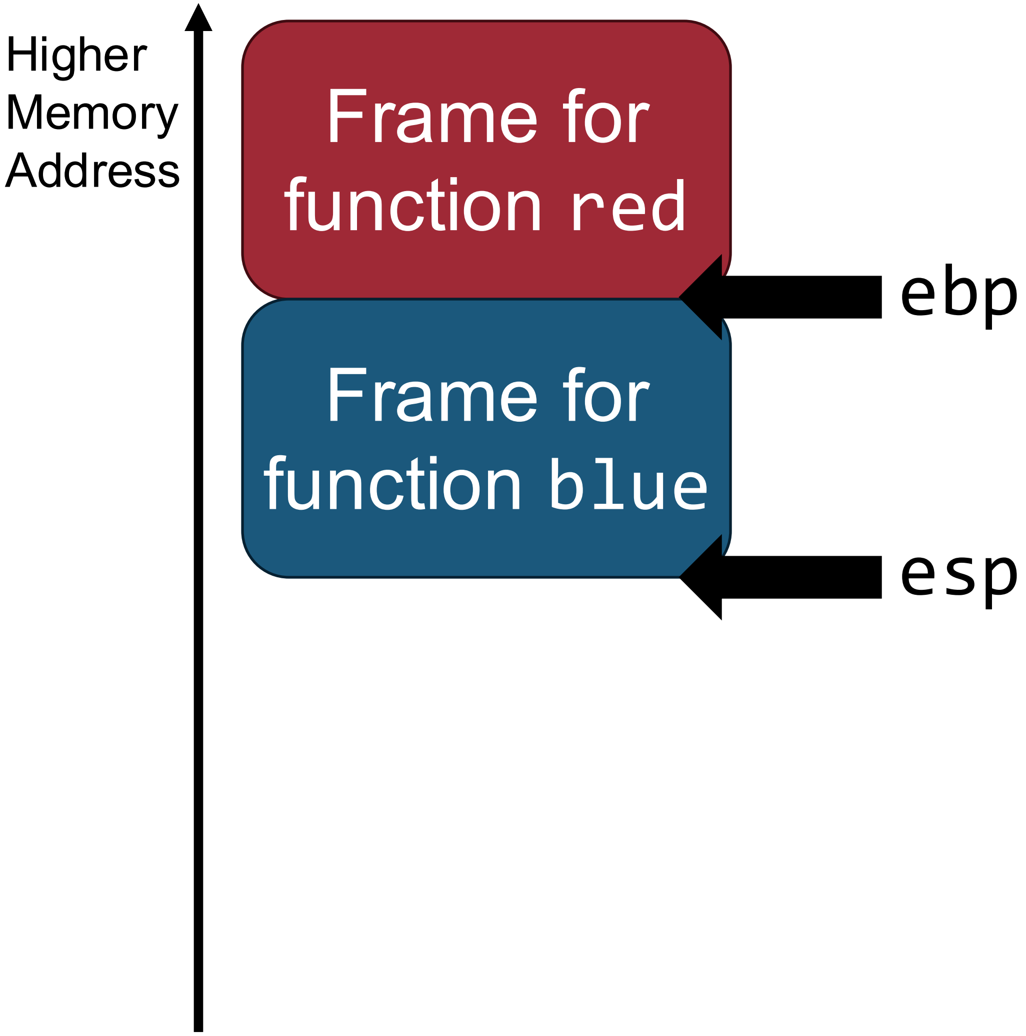


Recap: Registers in x86

- Program counter (instruction pointer)
  - `EIP`: points to the instruction to execute
- Stack pointers
  - `ESP`: points to the top of the stack
  - `EBP`: points to the base of the current stack frame
- Status register (FLAGS register)
  - `EFLAGS`: contains the current condition flags
- Other general purpose registers
  - `EAX`, `EBX`, `ECX`, `EDX`, `ESI`, `EDI`

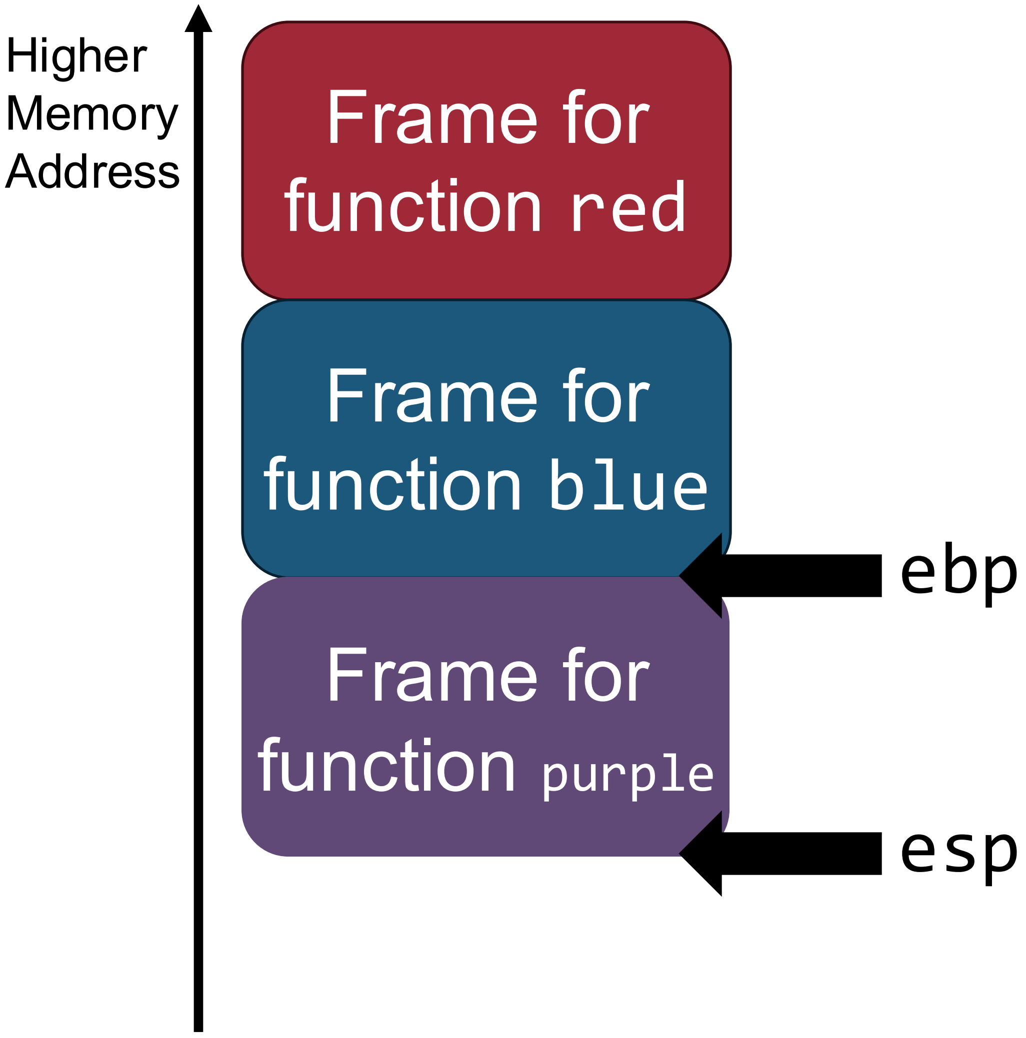
Our Example – Stack

```
int purple(int a1, int a2) {
    return 2 + a1 - a2;
}

int blue(int a1) {
    return 1 + purple(a1, b);
}

int red(int a1) {
    return blue(a1 - 42);
}
```



Our Example – Stack

```
int purple(int a1, int a2) {
    return 2 + a1 - a2;
}

int blue(int a1) {
    return 1 + purple(a1, b);
}

int red(int a1) {
    return blue(a1 - 42);
}
```



Questions

red: `blue (a1 - 42)` → blue: `1 + purple(a1, b)` → purple: `2 + a1 - a2`

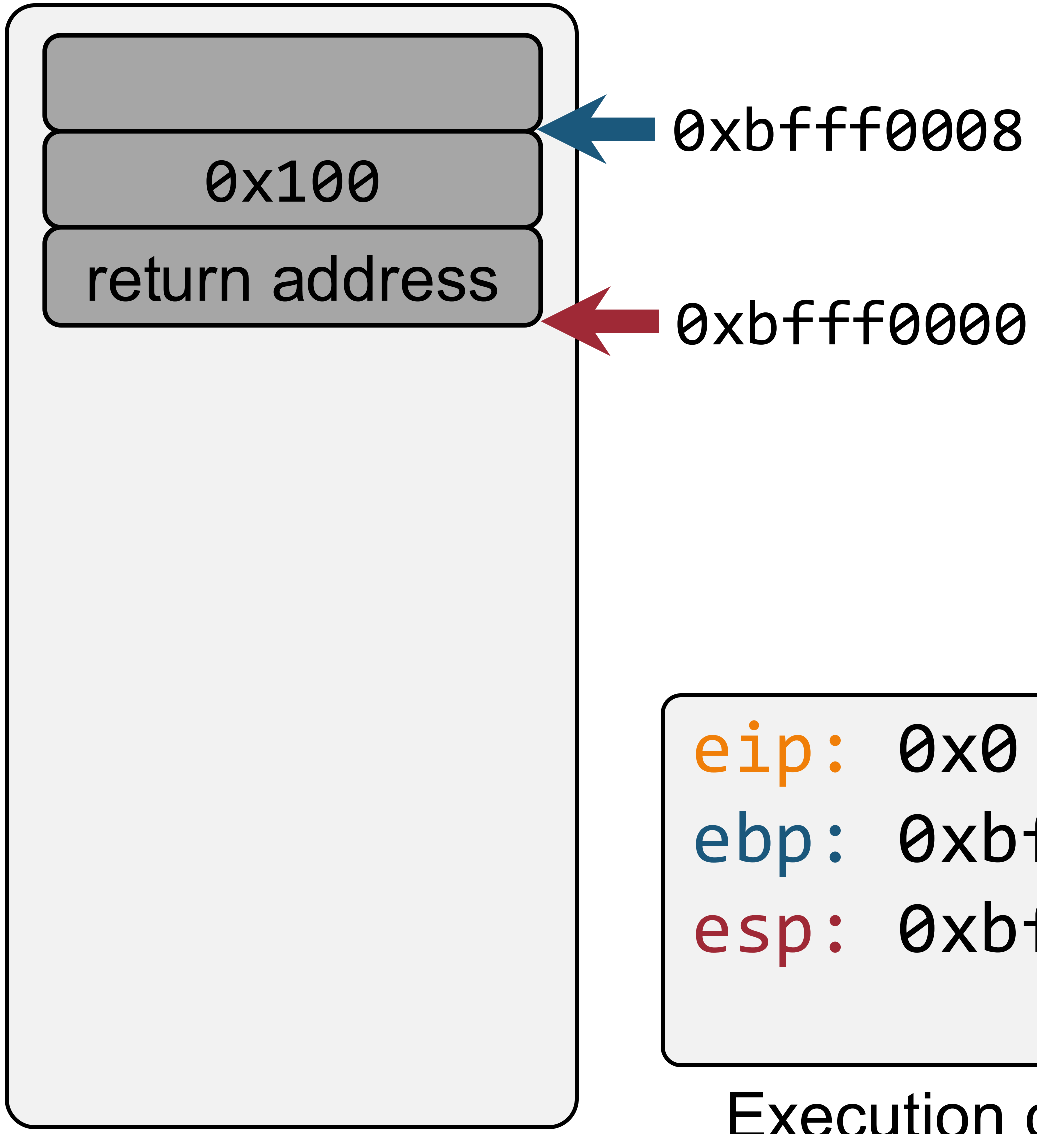
- How do we pass function parameters?
- When a function returns, how do we restore the register values of the caller
- Where do we store local variables?

We can easily get the answer by compiling the example program and disassembling the resulting binary

Disassembled Code (x86)

```
<red>:
   0:   push   ebp
   1:   mov    ebp,esp
   3:   sub    esp,0x28
   6:   mov    DWORD PTR [ebp-0xc],0x0
   d:   mov    eax,DWORD PTR [ebp+0x8]
  10:   sub    eax,0x2a
  13:   mov    DWORD PTR [esp],eax
  16:   call   Blue
  1b:   mov    edx,DWORD PTR [ebp-0xc]
  1e:   add    eax,edx
  20:   leave
  21:   ret

<blue>:
  22:   push   ebp
  23:   mov    ebp,esp
  25:   sub    esp,0x28
  28:   mov    DWORD PTR [ebp-0xc],0x1
  2f:   mov    eax, DWORD PTR [ebp-0xc]
  32:   mov    DWORD PTR [esp+0x4], eax
  36:   mov    eax, DWORD PTR [ebp+0x8]
  39:   mov    DWORD PTR [esp], eax
  3c:   call   purple
  41:   mov    edx, DWORD PTR [ebp-0xc]
  44:   add    eax, edx
  46:   leave
  47:   ret

<purple>:
  48:   push   ebp
  49:   mov    ebp,esp
  4b:   sub    esp,0x10
  4e:   mov    DWORD PTR [ebp -0x4],0x2
  55:   mov    eax,DWORD PTR [ebp+0x8]
  58:   mov    eax,DWORD PTR [ebp-0x4]
  5b:   add    eax,edx
  5d:   sub    eax,DWORD PTR [ebp+0xc]
  60:   leave
  61:   ret
```

Execution Example



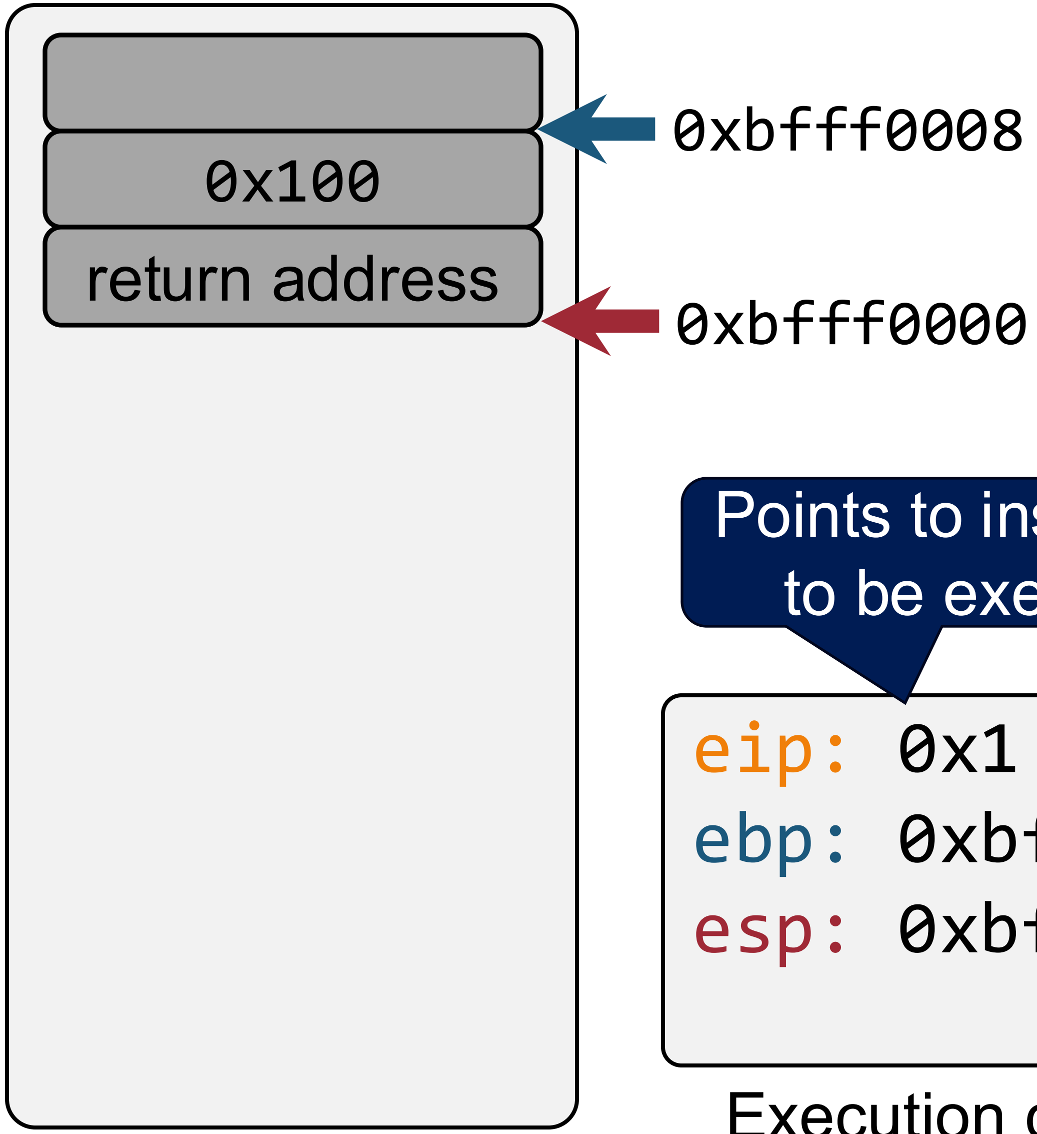
Virtual memory

```
eip: 0x0
ebp: 0xbfff0008
esp: 0xbfff0000
```

Execution context

```
<red>:
   0:  push   ebp
   1:  mov    ebp,esp
   3:  sub    esp,0x28
   6:  mov    DWORD PTR [ebp-0xc],0x0
   d:  mov    eax,DWORD PTR [ebp+0x8]
  10:  sub    eax,0x2a
  13:  mov    DWORD PTR [esp],eax
  16:  call   blue
  1b:  mov    edx,DWORD PTR [ebp-0xc]
  1e:  add    eax,edx
  20:  leave
  21:  ret

<blue>:
  22:  push    ebp
  23:  mov     ebp,esp
  ...
  46:   leave
  47:   ret
```

Execution Example



Virtual memory

Points to instruction to be executed

```
eip:  0x1
ebp:  0xbfff0008
esp:  0xbfff0000
```

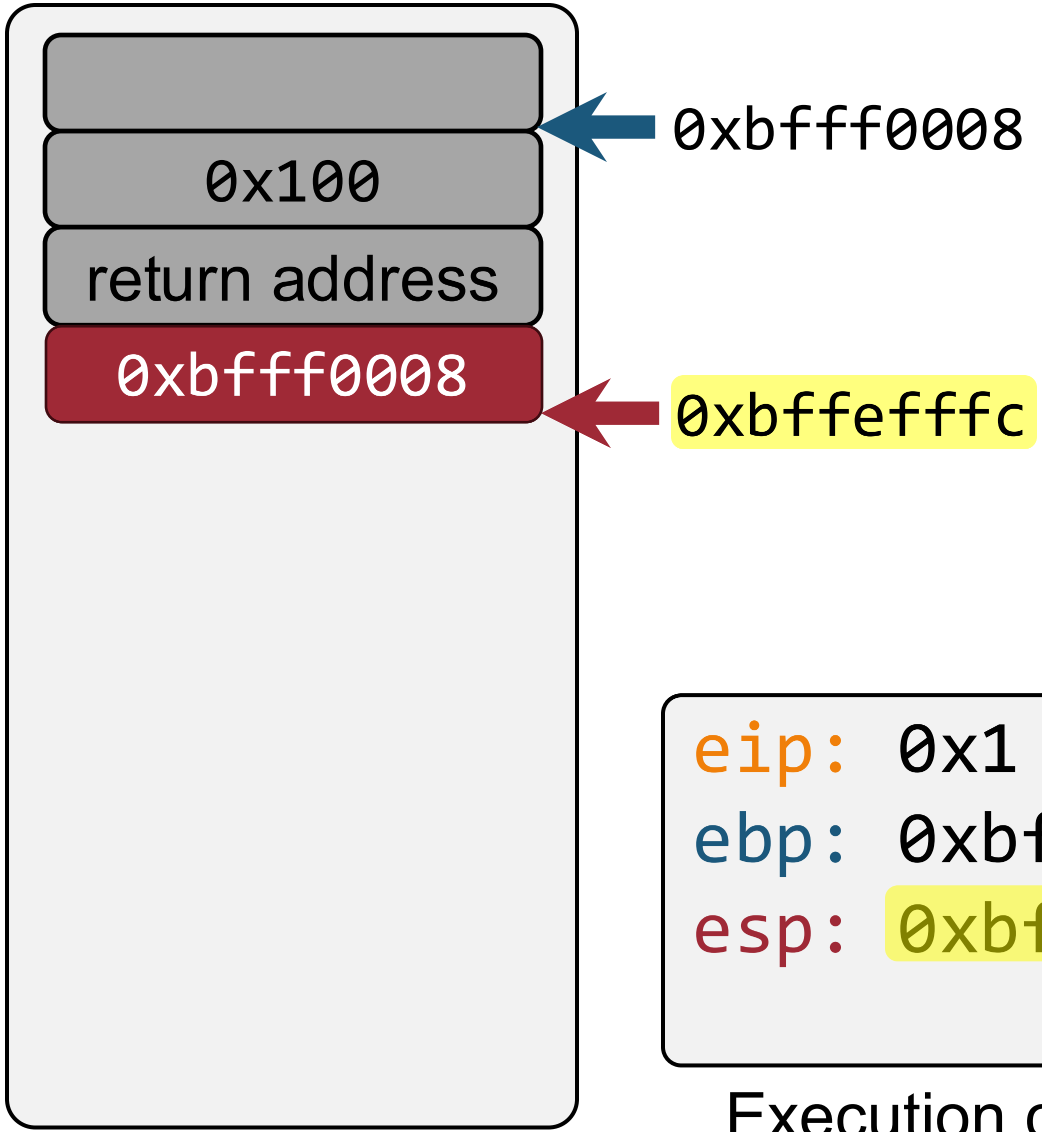
Execution context

Currently executed instruction

```
<red>:
   0:  push   ebp
   1:  mov    ebp,esp
   3:  sub    esp,0x28
   6:  mov    DWORD PTR [ebp-0xc],0x0
   d:  mov    eax,DWORD PTR [ebp+0x8]
  10:  sub    eax,0x2a
  13:  mov    DWORD PTR [esp],eax
  16:  call   blue
  1b:  mov    edx,DWORD PTR [ebp-0xc]
  1e:  add    eax,edx
  20:  leave
  21:  ret

<blue>:
  22:  push    ebp
  23:  mov     ebp,esp
  ...
  46:   leave
  47:   ret
```

Execution Example

Save the base address of the previous function's stack frame



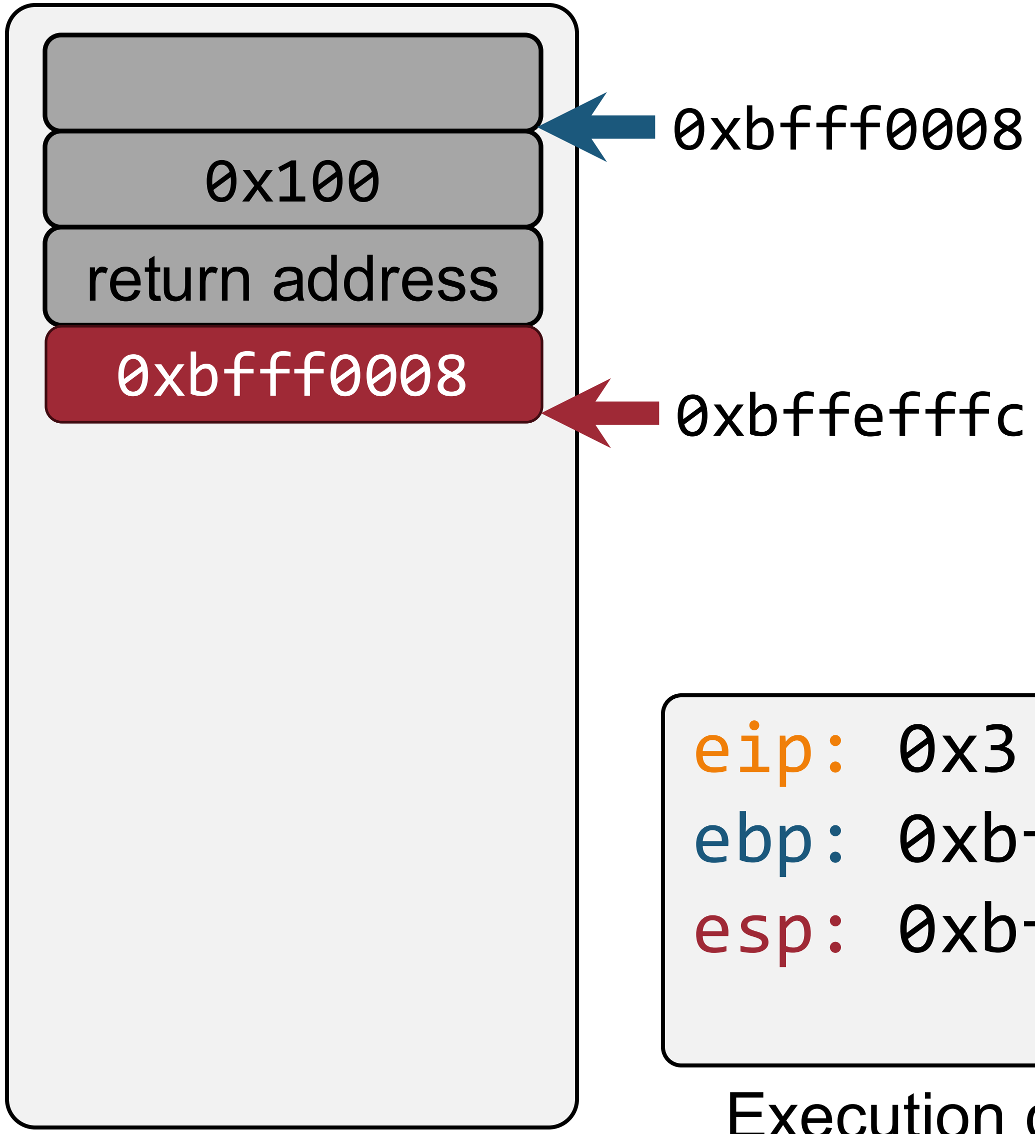
Virtual memory

```
eip: 0x1
ebp: 0xbfff0008
esp: 0xbffefffc
```

Execution context

```
<red>:
   0:  push   ebp
   1:  mov    ebp,esp
   3:  sub    esp,0x28
   6:  mov    DWORD PTR [ebp-0xc],0x0
   d:  mov    eax,DWORD PTR [ebp+0x8]
  10:  sub    eax,0x2a
  13:  mov    DWORD PTR [esp],eax
  16:  call   blue
  1b:  mov    edx,DWORD PTR [ebp-0xc]
  1e:  add    eax,edx
  20:  leave
  21:  ret

<blue>:
  22:  push    ebp
  23:  mov     ebp,esp
  ...
  46:   leave
  47:   ret
```

Execution Example



Virtual memory

```
eip: 0x3
ebp: 0xbfff0008
esp: 0xbffefffc
```

Execution context

```
<red>:
   0:  push   ebp
   1:  mov    ebp,esp
   3:  sub    esp,0x28
   6:  mov    DWORD PTR [ebp-0xc],0x0
   d:  mov    eax,DWORD PTR [ebp+0x8]
  10:  sub    eax,0x2a
  13:  mov    DWORD PTR [esp],eax
  16:  call   blue
  1b:  mov    edx,DWORD PTR [ebp-0xc]
  1e:  add    eax,edx
  20:  leave
  21:  ret

<blue>:
  22:  push    ebp
  23:  mov     ebp,esp
  ...
  46:   leave
  47:   ret
```

Execution Example

Now, ebp points to the base of the current stack frame

Virtual memory	
	0xbfff0008
0x100	
return address	
0xbfff0008	0xbffefffc

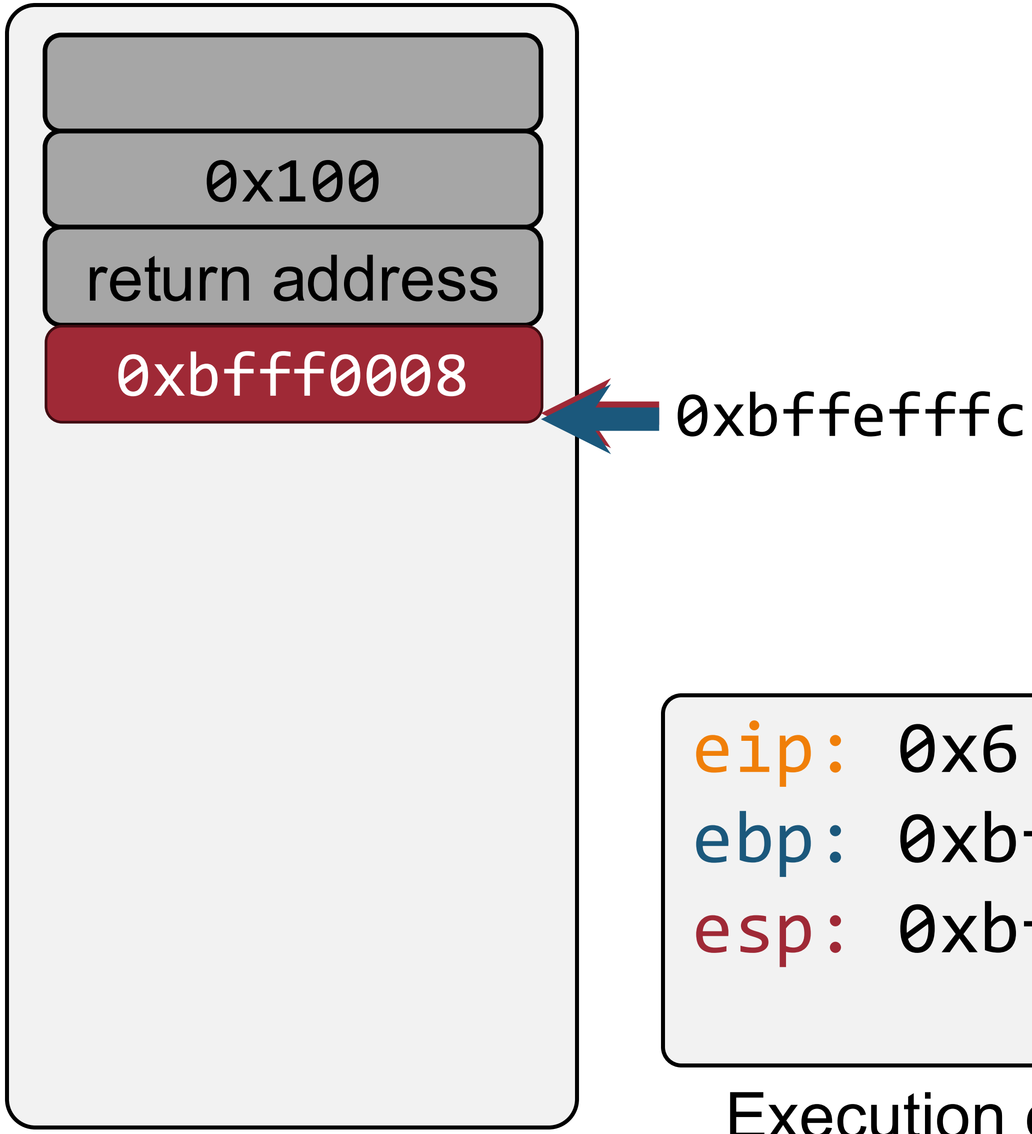
Virtual memory

```
eip: 0x3
ebp: 0xbffefffc
esp: 0xbffefffc
```

Execution context

```
<red>:
   0:  push   ebp
   1:  mov    ebp,esp
   3:  sub    esp,0x28
   6:  mov    DWORD PTR [ebp-0xc],0x0
   d:  mov    eax,DWORD PTR [ebp+0x8]
  10:  sub    eax,0x2a
  13:  mov    DWORD PTR [esp],eax
  16:  call   blue
  1b:  mov    edx,DWORD PTR [ebp-0xc]
  1e:  add    eax,edx
  20:  leave
  21:  ret

<blue>:
  22:  push    ebp
  23:  mov     ebp,esp
  ...
  46:   leave
  47:   ret
```

Execution Example



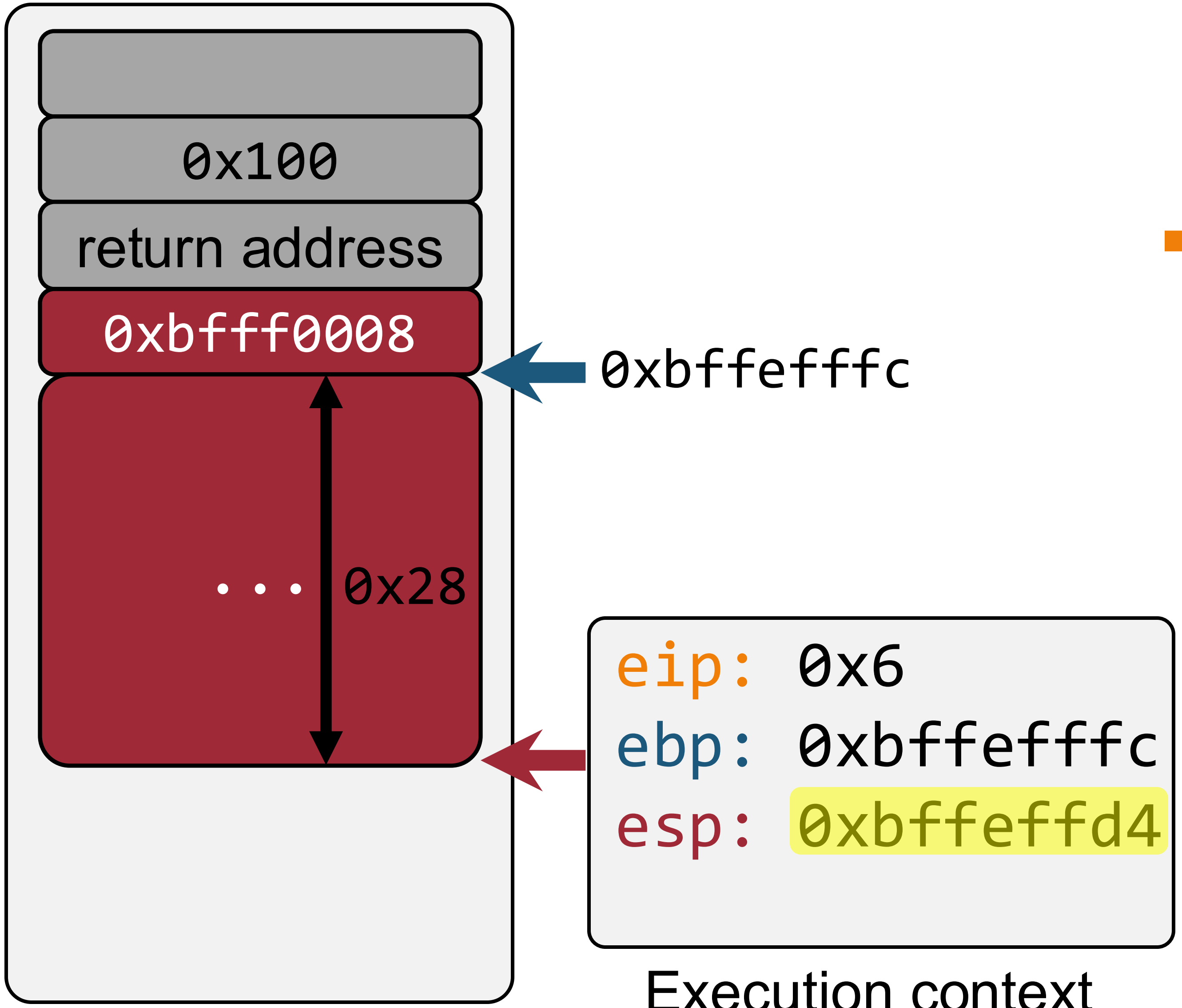
Virtual memory

```
eip: 0x6
ebp: 0xbffefffc
esp: 0xbffefffc
```

Execution context

```
<red>:
    0:  push   ebp
    1:  mov    ebp,esp
    3:  sub    esp,0x28
    6:  mov    DWORD PTR [ebp-0xc],0x0
    d:  mov    eax,DWORD PTR [ebp+0x8]
   10:  sub    eax,0x2a
   13:  mov    DWORD PTR [esp],eax
   16:  call   blue
   1b:  mov    edx,DWORD PTR [ebp-0xc]
   1e:  add    eax,edx
   20:  leave
   21:  ret

<blue>:
   22:  push    ebp
   23:  mov     ebp,esp
   ...
   46:  leave
   47:  ret
```

Execution Example

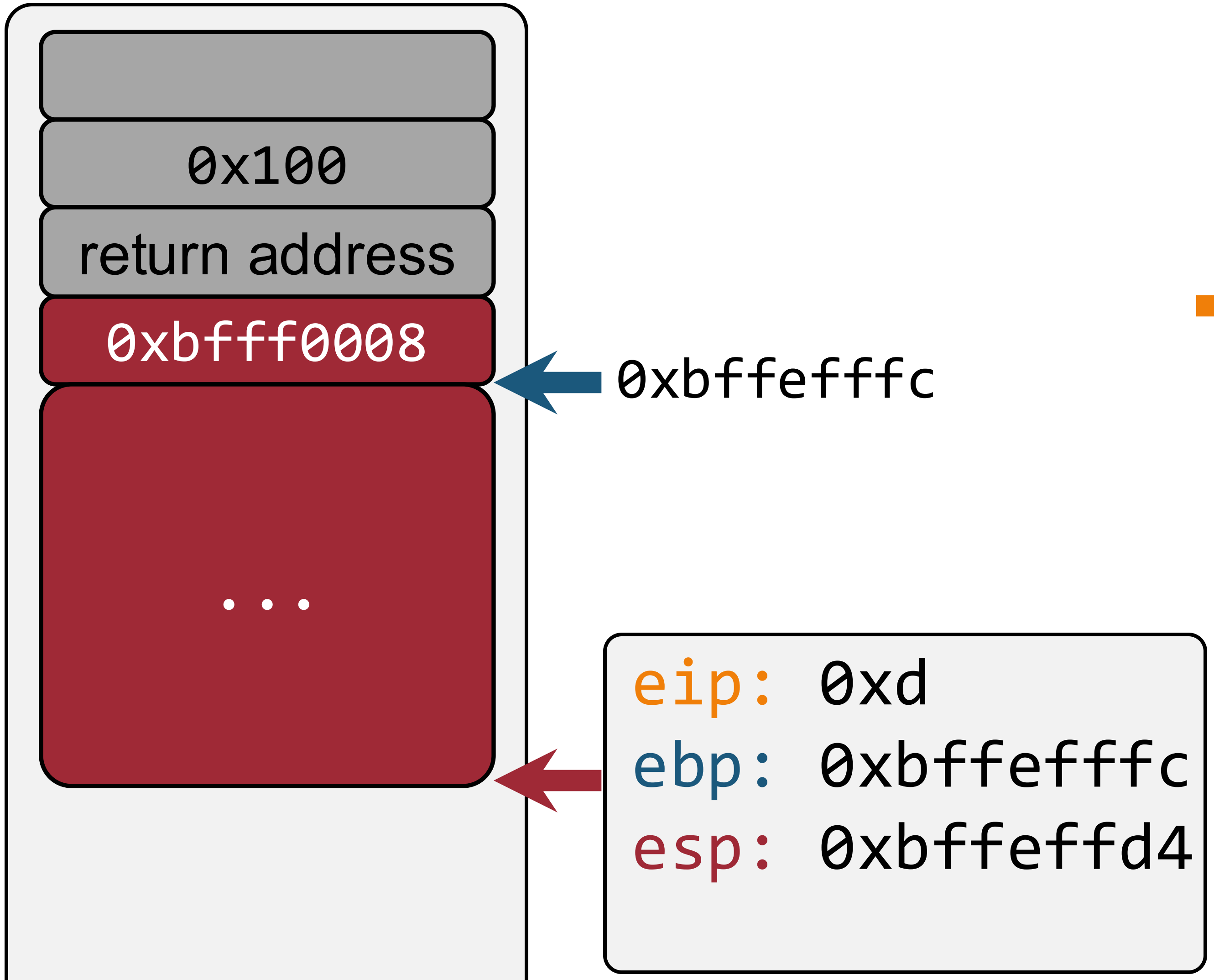


Virtual memory

Execution context

```
<red>:
   0:  push   ebp
   1:  mov    ebp,esp
   3:  sub    esp,0x28
   6:  mov    DWORD PTR [ebp-0xc],0x0
   d:  mov    eax,DWORD PTR [ebp+0x8]
  10:  sub    eax,0x2a
  13:  mov    DWORD PTR [esp],eax
  16:  call   blue
  1b:  mov    edx,DWORD PTR [ebp-0xc]
  1e:  add    eax,edx
  20:  leave
  21:  ret

<blue>:
  22:  push    ebp
  23:  mov     ebp,esp
  ...
  46:   leave
  47:   ret
```

Execution Example

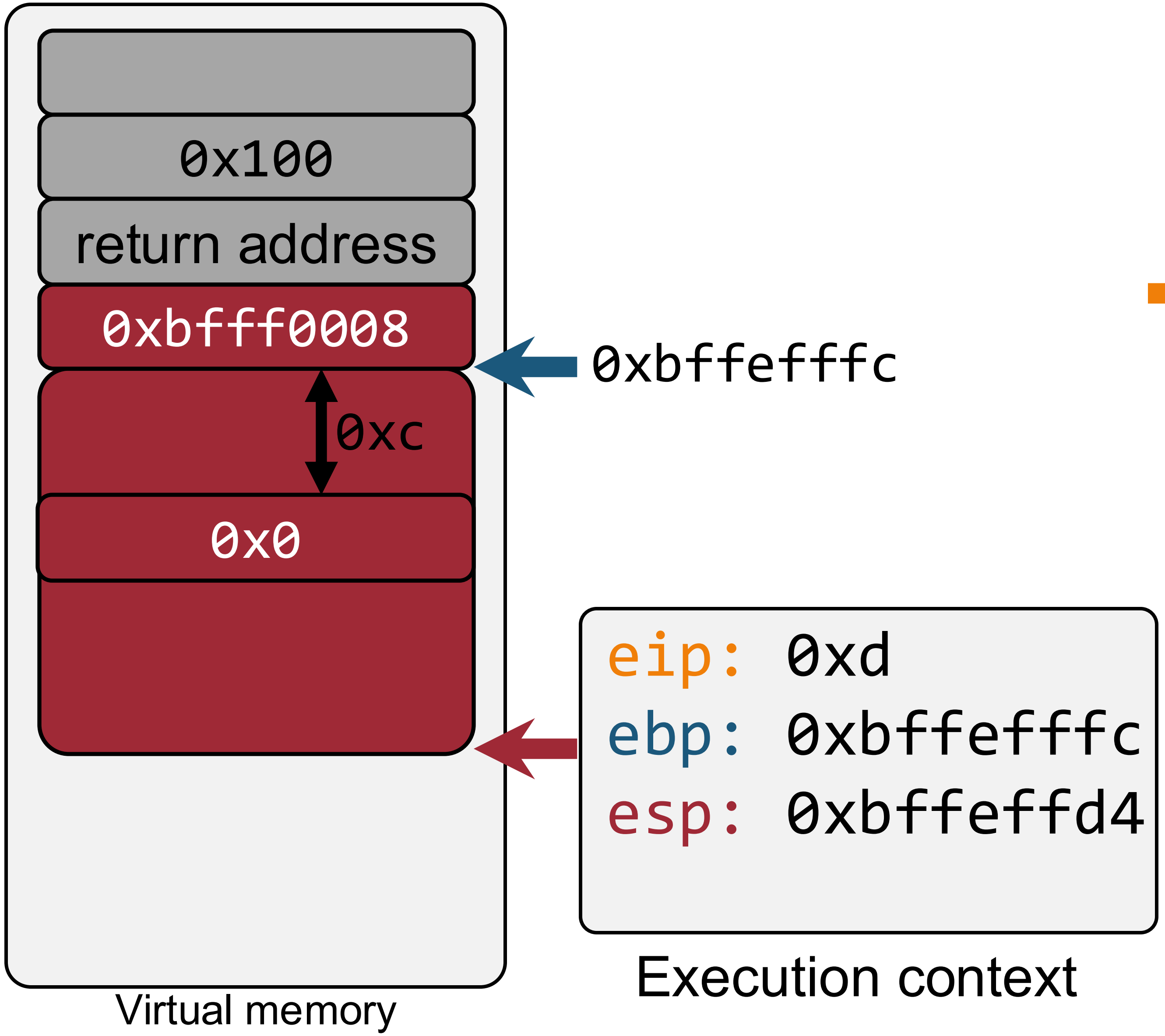


Virtual memory

Execution context

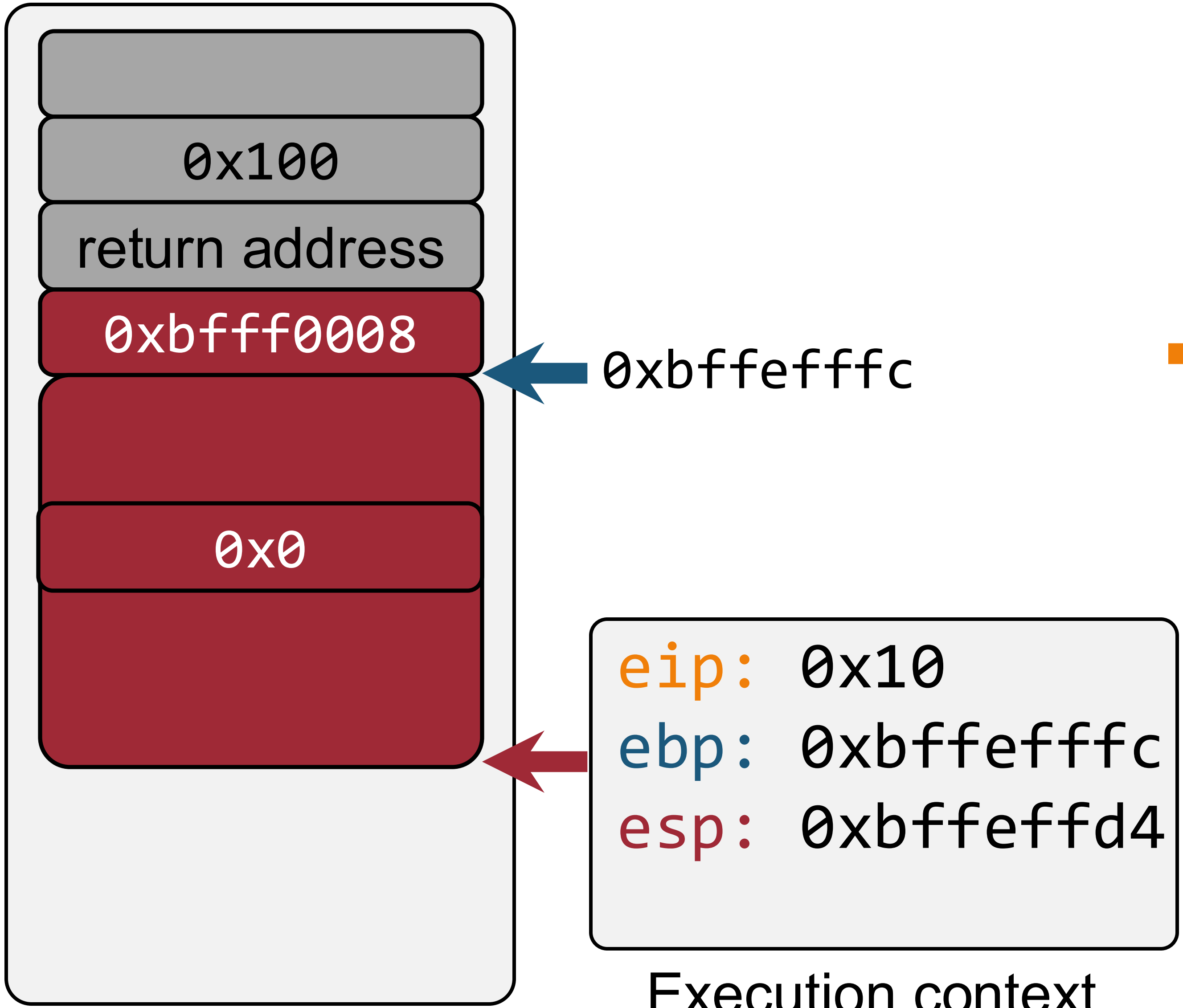
```
<red>:
   0:  push   ebp
   1:  mov    ebp,esp
   3:  sub    esp,0x28
   6:  mov    DWORD PTR [ebp-0xc],0x0
   d:  mov    eax,DWORD PTR [ebp+0x8]
  10:  sub    eax,0x2a
  13:  mov    DWORD PTR [esp],eax
  16:  call   blue
  1b:  mov    edx,DWORD PTR [ebp-0xc]
  1e:  add    eax,edx
  20:  leave
  21:  ret

<blue>:
  22:  push    ebp
  23:  mov     ebp,esp
  ...
  46:   leave
  47:   ret
```

Execution Example




```
<red>:
   0:  push   ebp
   1:  mov    ebp,esp
   3:  sub    esp,0x28
   6:  mov    DWORD PTR [ebp-0xc],0x0
   d:  mov    eax,DWORD PTR [ebp+0x8]
  10:  sub    eax,0x2a
  13:  mov    DWORD PTR [esp],eax
  16:  call   blue
  1b:  mov    edx,DWORD PTR [ebp-0xc]
  1e:  add    eax,edx
  20:  leave
  21:  ret

<blue>:
  22:  push    ebp
  23:  mov     ebp,esp
  ...
  46:   leave
  47:   ret
```

Execution Example

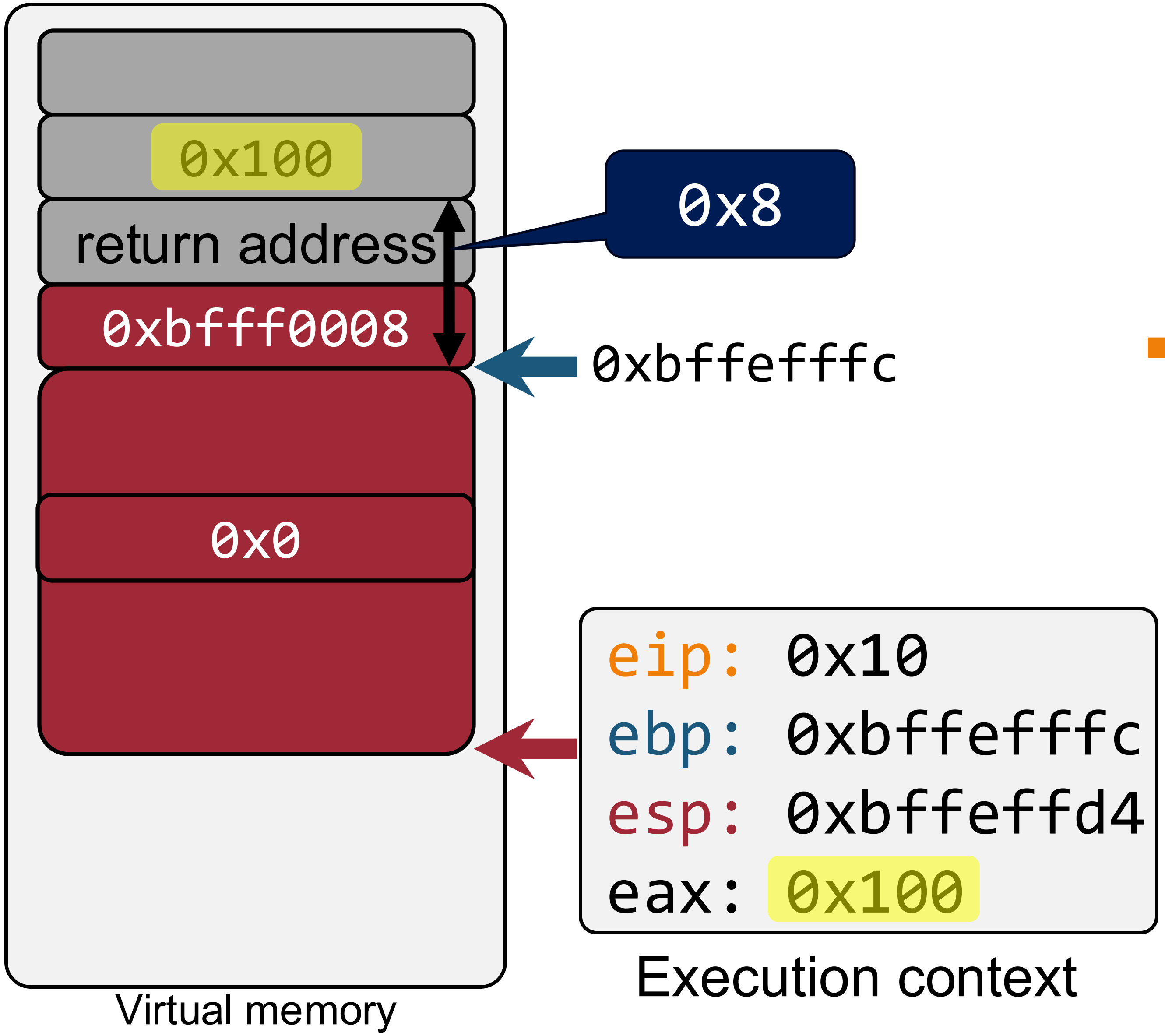


Virtual memory

Execution context

```
<red>:
   0:  push   ebp
   1:  mov    ebp,esp
   3:  sub    esp,0x28
   6:  mov    DWORD PTR [ebp-0xc],0x0
   d:  mov    eax,DWORD PTR [ebp+0x8]
  10:  sub    eax,0x2a
  13:  mov    DWORD PTR [esp],eax
  16:  call   blue
  1b:  mov    edx,DWORD PTR [ebp-0xc]
  1e:  add    eax,edx
  20:  leave
  21:  ret

<blue>:
  22:  push    ebp
  23:  mov     ebp,esp
  ...
  46:   leave
  47:   ret
```

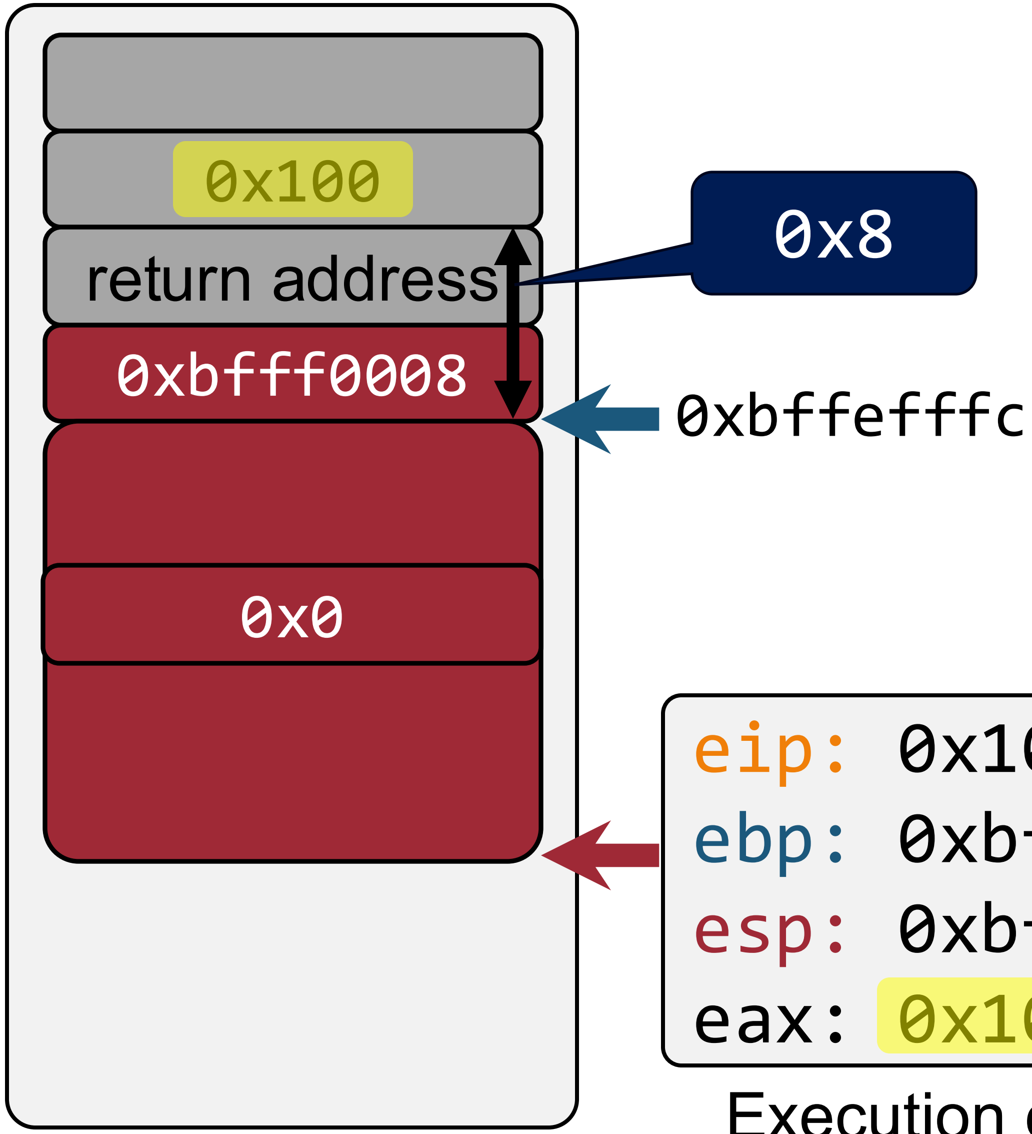
Execution Example




```
<red>:
   0:  push   ebp
   1:  mov    ebp,esp
   3:  sub    esp,0x28
   6:  mov    DWORD PTR [ebp-0xc],0x0
   d:  mov    eax,DWORD PTR [ebp+0x8]
  10:  sub    eax,0x2a
  13:  mov    DWORD PTR [esp],eax
  16:  call   blue
  1b:  mov    edx,DWORD PTR [ebp-0xc]
  1e:  add    eax,edx
  20:  leave
  21:  ret

<blue>:
  22:  push    ebp
  23:  mov     ebp,esp
  ...
  46:   leave
  47:   ret
```

Execution Example

```
int red(int a1) {
    return blue(a1 - 42);
}
```



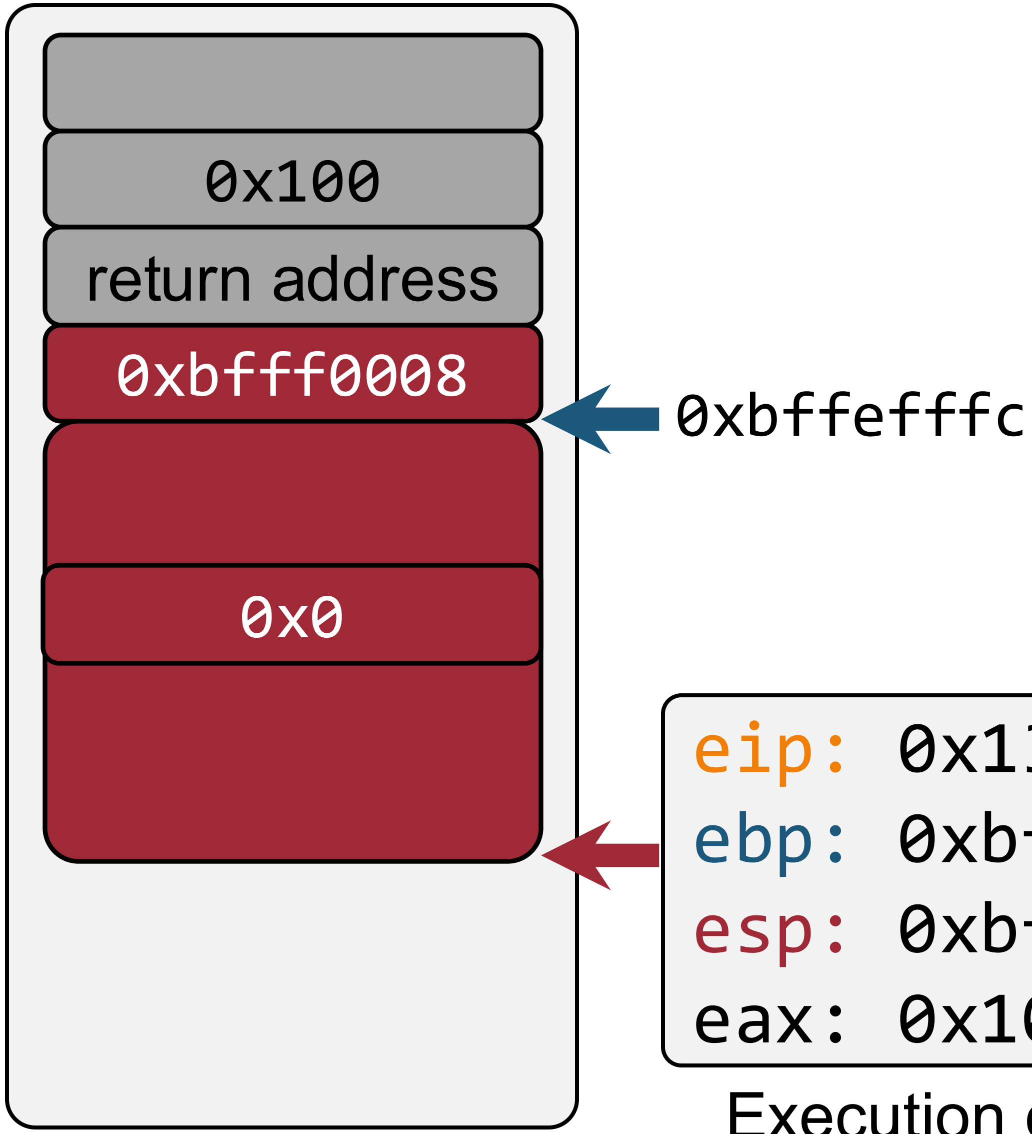
Virtual memory

```
eip: 0x10
ebp: 0xbffefffc
esp: 0xbffeffd4
eax: 0x100
```

Execution context

```
<red>:
   0:  push   ebp
   1:  mov    ebp,esp
   3:  sub    esp,0x28
   6:  mov    DWORD PTR [ebp-0xc],0x0
   d:  mov    eax,DWORD PTR [ebp+0x8]
  10:  sub    eax,0x2a
  13:  mov    DWORD PTR [esp],eax
  16:  call   blue
  1b:  mov    edx,DWORD PTR [ebp-0xc]
  1e:  add    eax,edx
  20:  leave
  21:  ret

<blue>:
  22:  push    ebp
  23:  mov     ebp,esp
  ...
  46:   leave
  47:   ret
```

Execution Example



Virtual memory

```
eip: 0x13
ebp: 0xbffefffc
esp: 0xbffeffd4
eax: 0x100
```

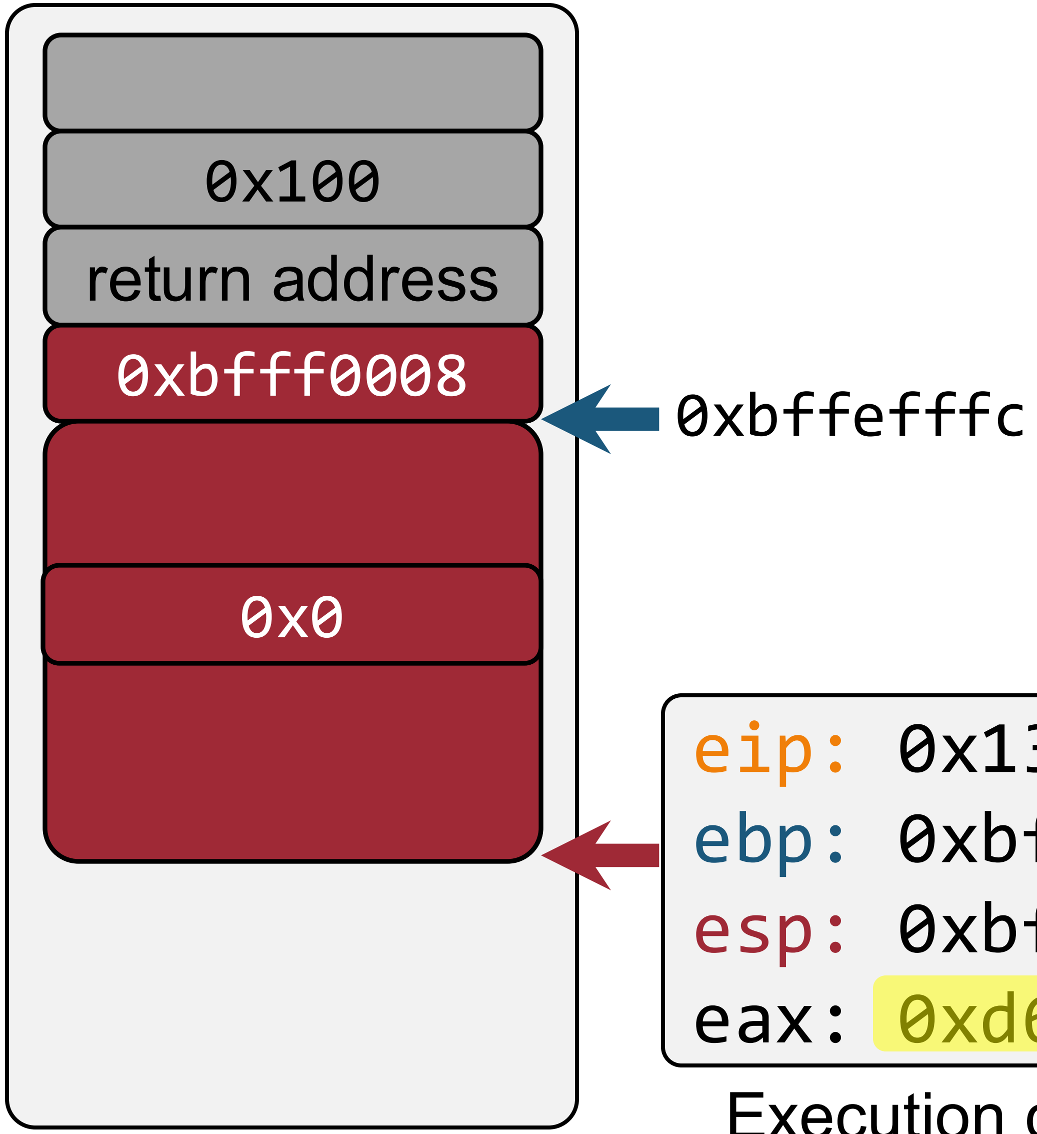
Execution context

```
<red>:
   0:  push   ebp
   1:  mov    ebp,esp
   3:  sub    esp,0x28
   6:  mov    DWORD PTR [ebp-0xc],0x0
   d:  mov    eax,DWORD PTR [ebp+0x8]
  10:  sub    eax,0x2a
  13:  mov    DWORD PTR [esp],eax
  16:  call   blue
  1b:  mov    edx,DWORD PTR [ebp-0xc]
  1e:  add    eax,edx
  20:  leave
  21:  ret

<blue>:
  22:  push    ebp
  23:  mov     ebp,esp
  ...
  46:   leave
  47:   ret
```

Execution Example

```
int red(int a1) {
    return blue(a1 - 42);
}
```

```
<red>:
   0:  push   ebp
   1:  mov    ebp,esp
   3:  sub    esp,0x28
   6:  mov    DWORD PTR [ebp-0xc],0x0
   d:  mov    eax,DWORD PTR [ebp+0x8]
  10:  sub    eax,0x2a
  13:  mov    DWORD PTR [esp],eax
  16:  call   blue
  1b:  mov    edx,DWORD PTR [ebp-0xc]
  1e:  add    eax,edx
  20:  leave
  21:  ret

<blue>:
  22:  push   ebp
  23:  mov    ebp,esp
  ...
  46:  leave
  47:  ret
```

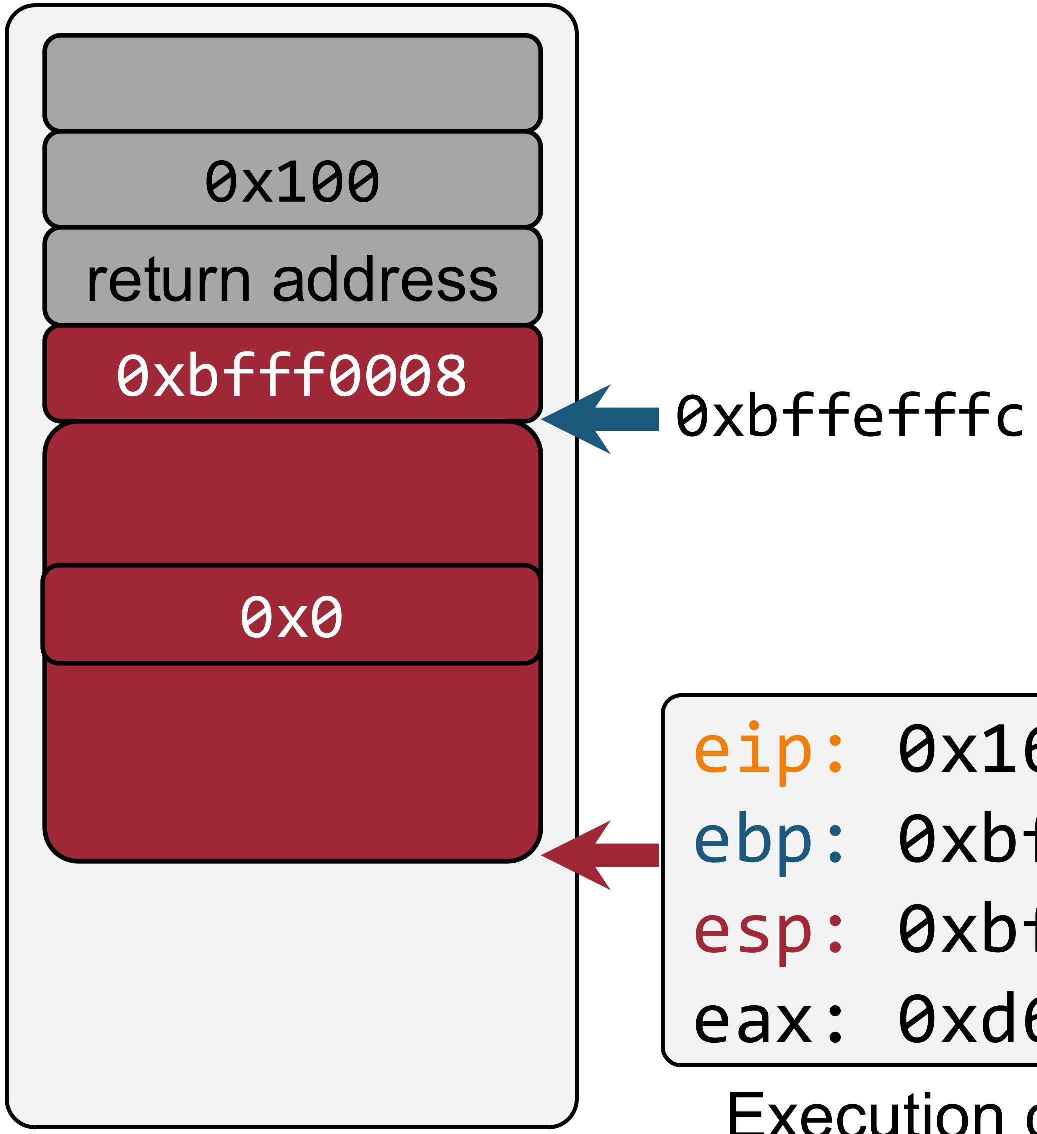


Virtual memory

```
eip: 0x13
ebp: 0xbffefffc
esp: 0xbffeffd4
eax: 0xd6
```

Execution context

Execution Example



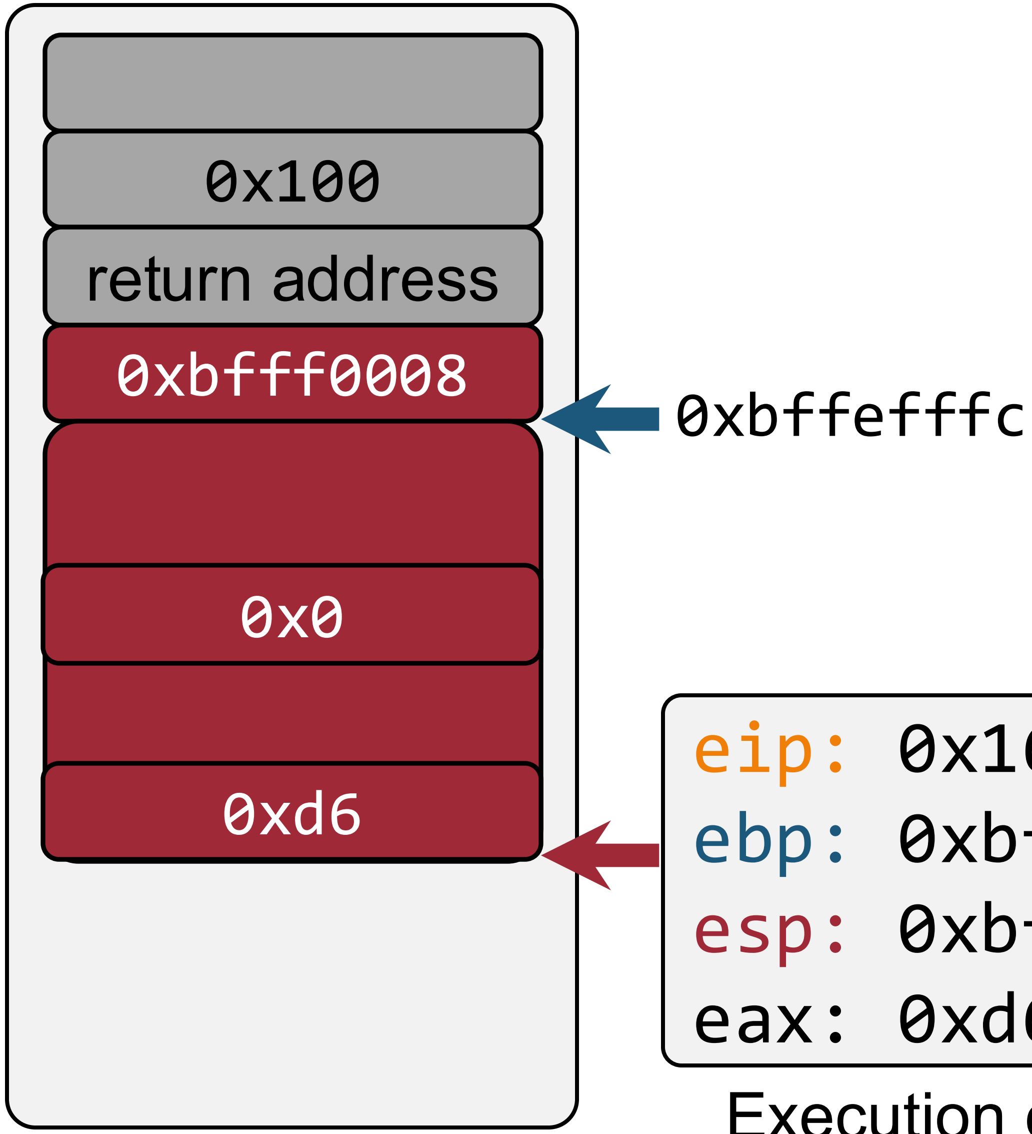
Virtual memory

```
eip: 0x16
ebp: 0xbffefffc
esp: 0xbffeffd4
eax: 0xd6
```

Execution context

```
<red>:
   0:  push   ebp
   1:  mov    ebp,esp
   3:  sub    esp,0x28
   6:  mov    DWORD PTR [ebp-0xc],0x0
   d:  mov    eax,DWORD PTR [ebp+0x8]
  10:  sub    eax,0x2a
  13:  mov    DWORD PTR [esp],eax
  16:  call   blue
  1b:  mov    edx,DWORD PTR [ebp-0xc]
  1e:  add    eax,edx
  20:  leave
  21:  ret

<blue>:
  22:  push    ebp
  23:  mov     ebp,esp
  ...
  46:   leave
  47:   ret
```

Execution Example



Virtual memory

```
eip: 0x16
ebp: 0xbffefffc
esp: 0xbffeffd4
eax: 0xd6
```

Execution context

```
<red>:
   0:  push   ebp
   1:  mov    ebp,esp
   3:  sub    esp,0x28
   6:  mov    DWORD PTR [ebp-0xc],0x0
   d:  mov    eax,DWORD PTR [ebp+0x8]
  10:  sub    eax,0x2a
  13:  mov    DWORD PTR [esp],eax
  16:  call   blue
  1b:  mov    edx,DWORD PTR [ebp-0xc]
  1e:  add    eax,edx
  20:  leave
  21:  ret

<blue>:
  22:  push    ebp
  23:  mov     ebp,esp
  ...
  46:   leave
  47:   ret
```

Execution Example

0x100
return address
0xbfff0008
0x0
0xd6

Virtual memory

0xbffefffc

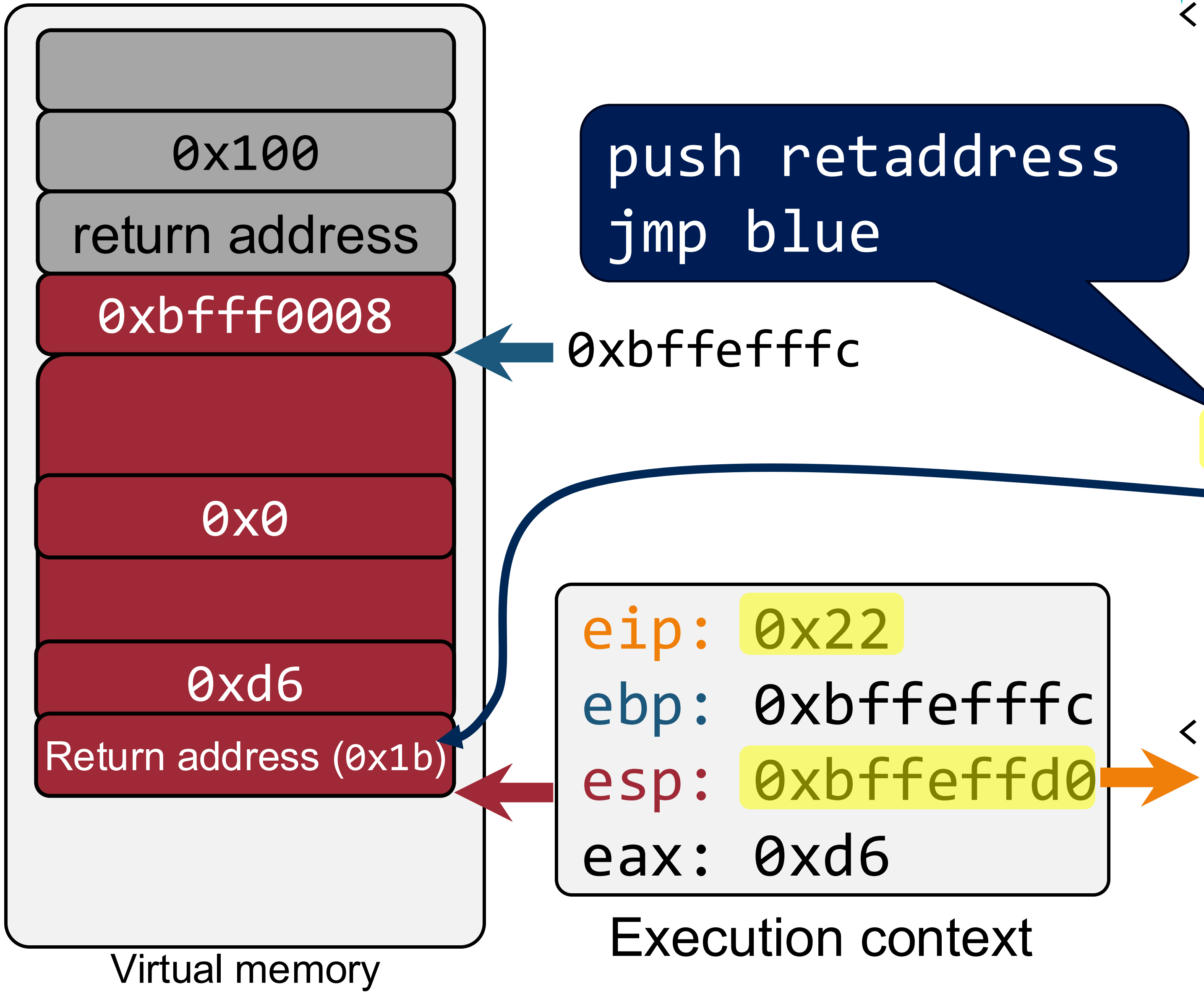
```
push retaddress
jmp blue
```

```
eip: 0x1b
ebp: 0xbffefffc
esp: 0xbffeffd4
eax: 0xd6
```

Execution context

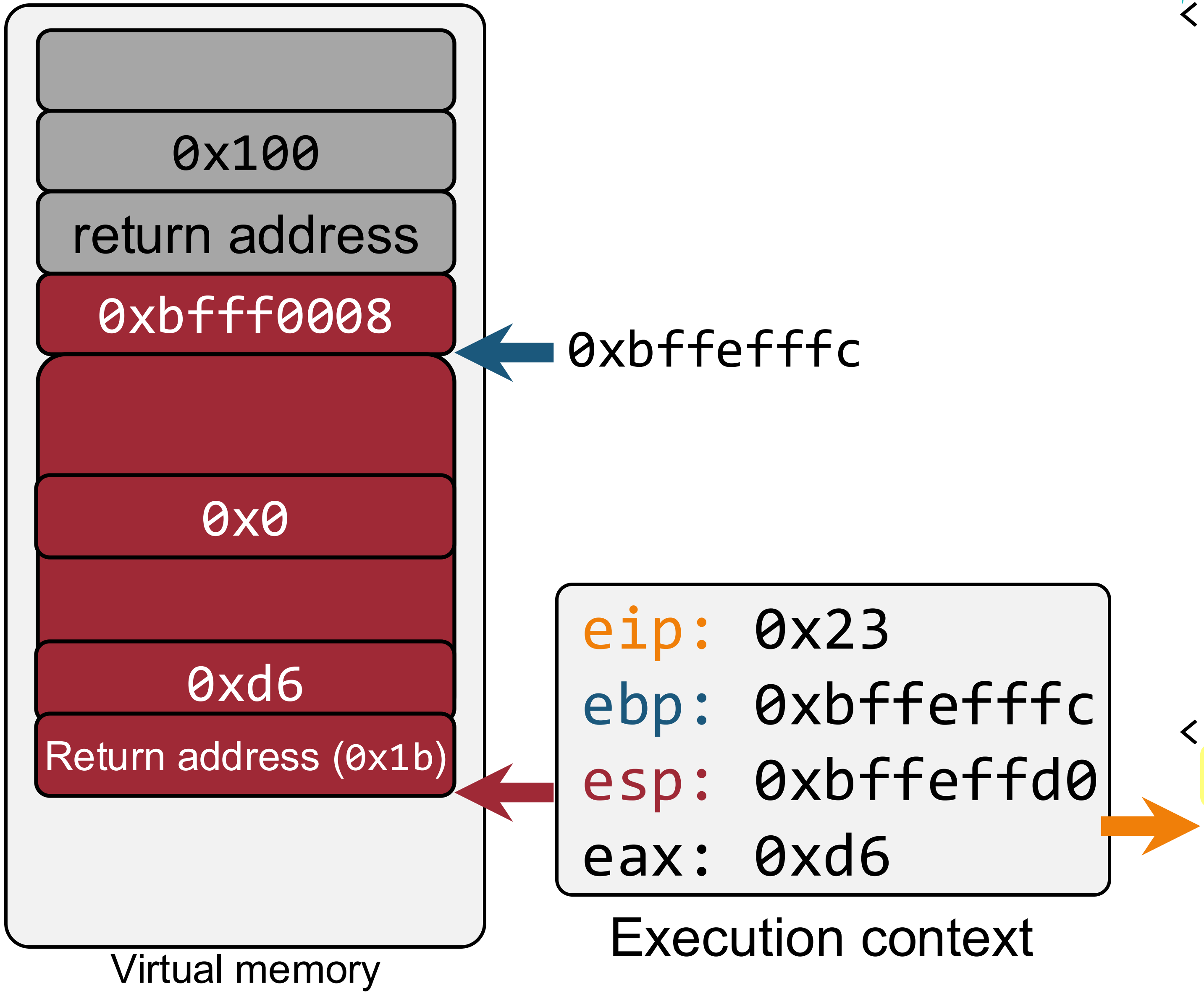
```
<red>:
   0:  push   ebp
   1:  mov    ebp,esp
   3:  sub    esp,0x28
   6:  mov    DWORD PTR [ebp-0xc],0x0
   d:  mov    eax,DWORD PTR [ebp+0x8]
  10:  sub    eax,0x2a
  13:  mov    DWORD PTR [esp],eax
  16:  call   blue
  1b:  mov    edx,DWORD PTR [ebp-0xc]
  1e:  add    eax,edx
  20:  leave
  21:  ret

<blue>:
  22:  push   ebp
  23:  mov    ebp,esp
  ...
  46:  leave
  47:  ret
```

Execution Example



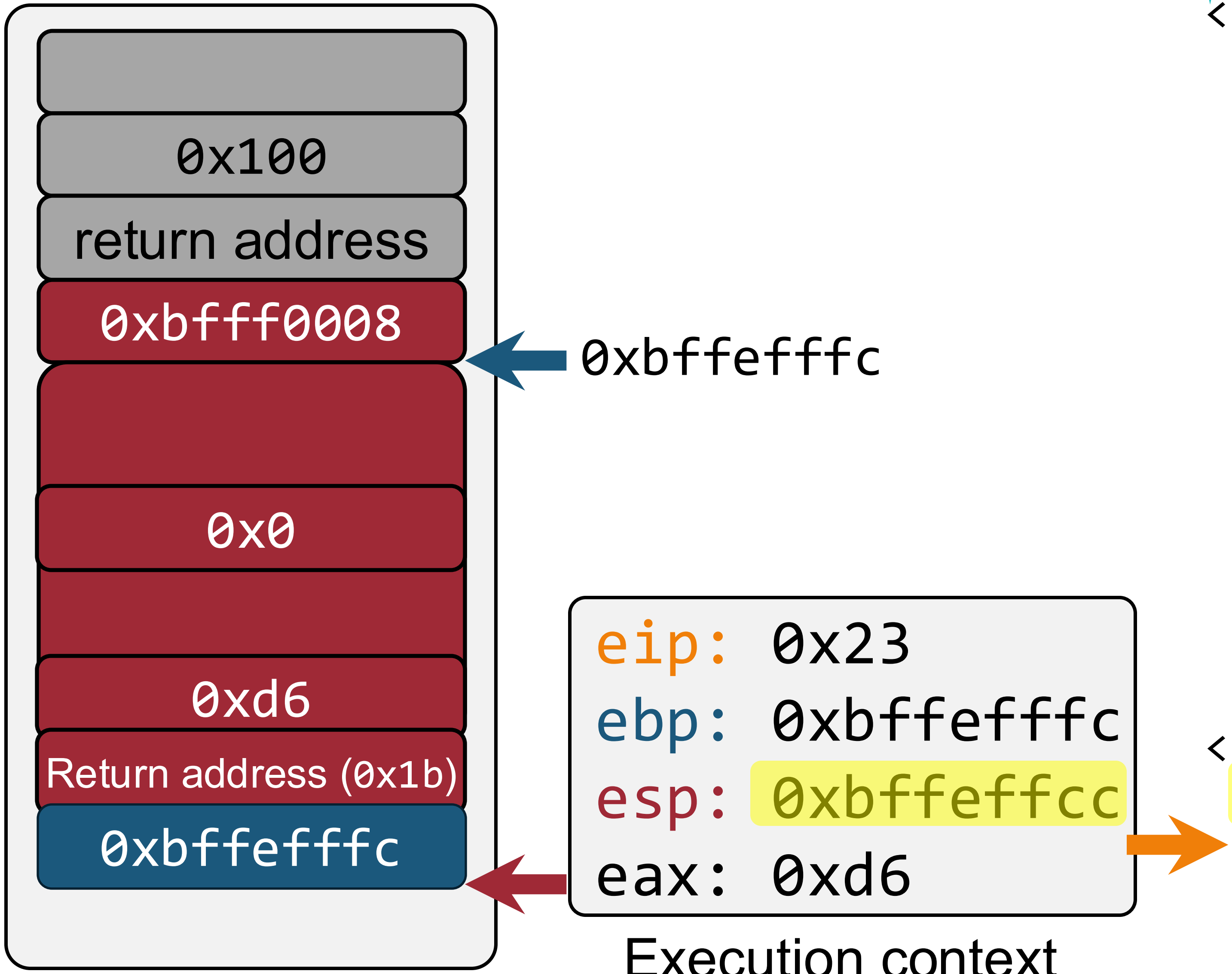

```
<red>:
   0:  push   ebp
   1:  mov    ebp,esp
   3:  sub    esp,0x28
   6:  mov    DWORD PTR [ebp-0xc],0x0
   d:  mov    eax,DWORD PTR [ebp+0x8]
  10:  sub    eax,0x2a
  13:  mov    DWORD PTR [esp],eax
  16:  call   blue
  1b:  mov    edx,DWORD PTR [ebp-0xc]
  1e:  add    eax,edx
  20:  leave
  21:  ret

<blue>:
  22:  push    ebp
  23:  mov     ebp,esp
  ...
  46:   leave
  47:   ret
```

Execution Example




```
<red>:
   0:  push   ebp
   1:  mov    ebp,esp
   3:  sub    esp,0x28
   6:  mov    DWORD PTR [ebp-0xc],0x0
   d:  mov    eax,DWORD PTR [ebp+0x8]
  10:  sub    eax,0x2a
  13:  mov    DWORD PTR [esp],eax
  16:  call   blue
  1b:  mov    edx,DWORD PTR [ebp-0xc]
  1e:  add    eax,edx
  20:  leave
  21:  ret

<blue>:
  22:  push    ebp
  23:  mov     ebp,esp
  ...
  46:   leave
  47:   ret
```

Execution Example

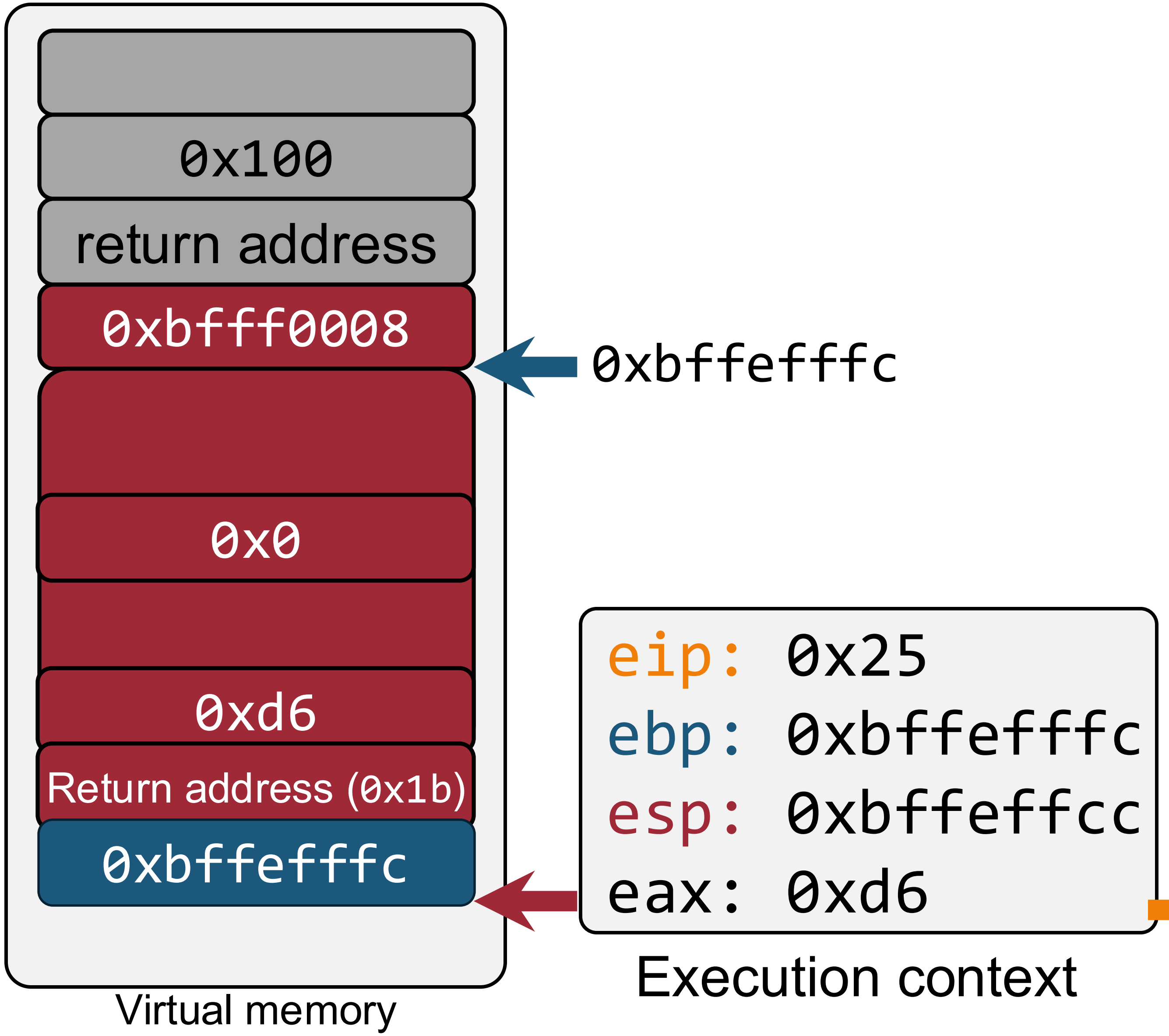


Virtual memory

Execution context

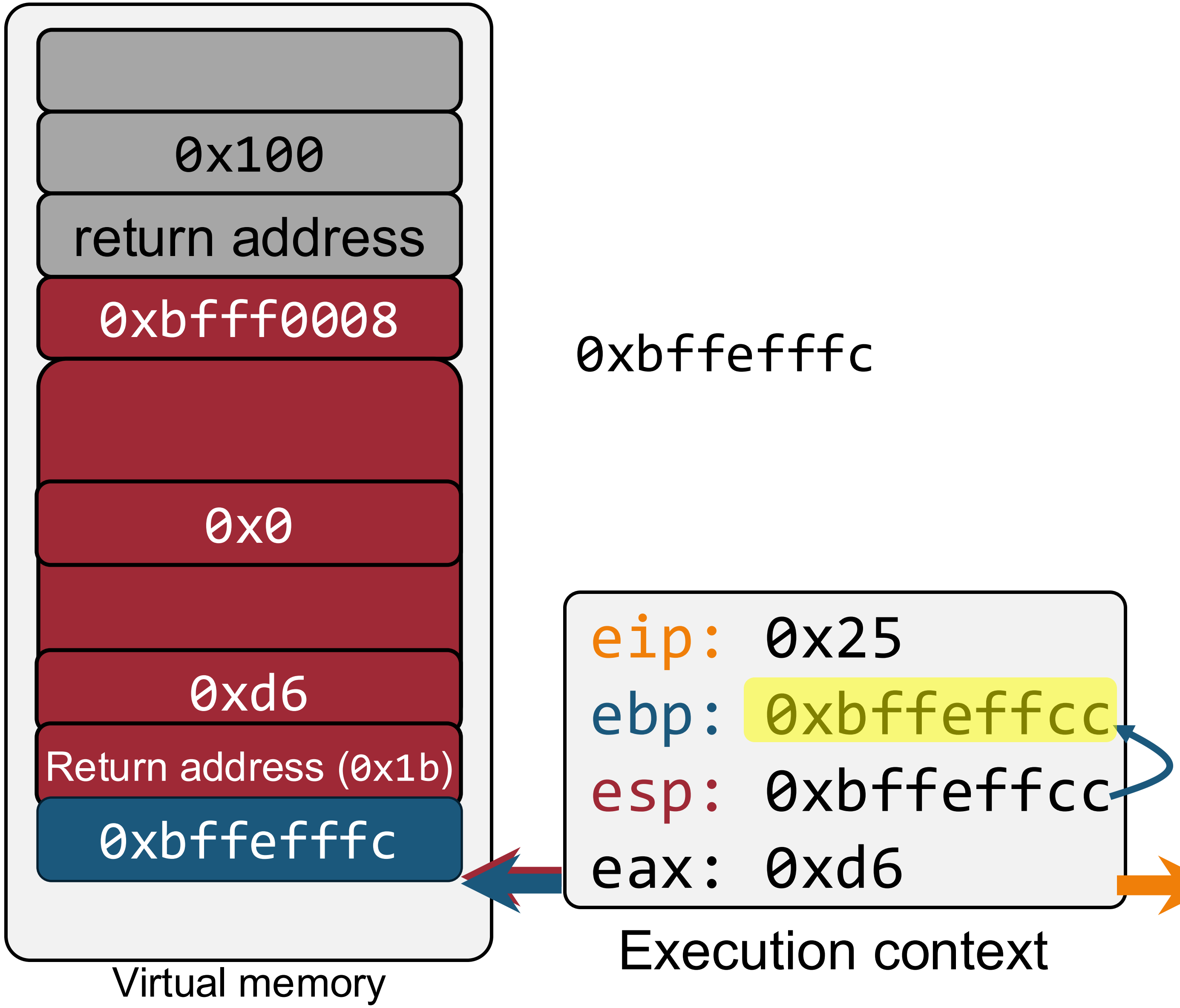
```
<red>:
   0:  push   ebp
   1:  mov    ebp,esp
   3:  sub    esp,0x28
   6:  mov    DWORD PTR [ebp-0xc],0x0
   d:  mov    eax,DWORD PTR [ebp+0x8]
  10:  sub    eax,0x2a
  13:  mov    DWORD PTR [esp],eax
  16:  call   blue
  1b:  mov    edx,DWORD PTR [ebp-0xc]
  1e:  add    eax,edx
  20:  leave
  21:  ret

<blue>:
  22:  push    ebp
  23:  mov     ebp,esp
  ...
  46:   leave
  47:   ret
```

Execution Example



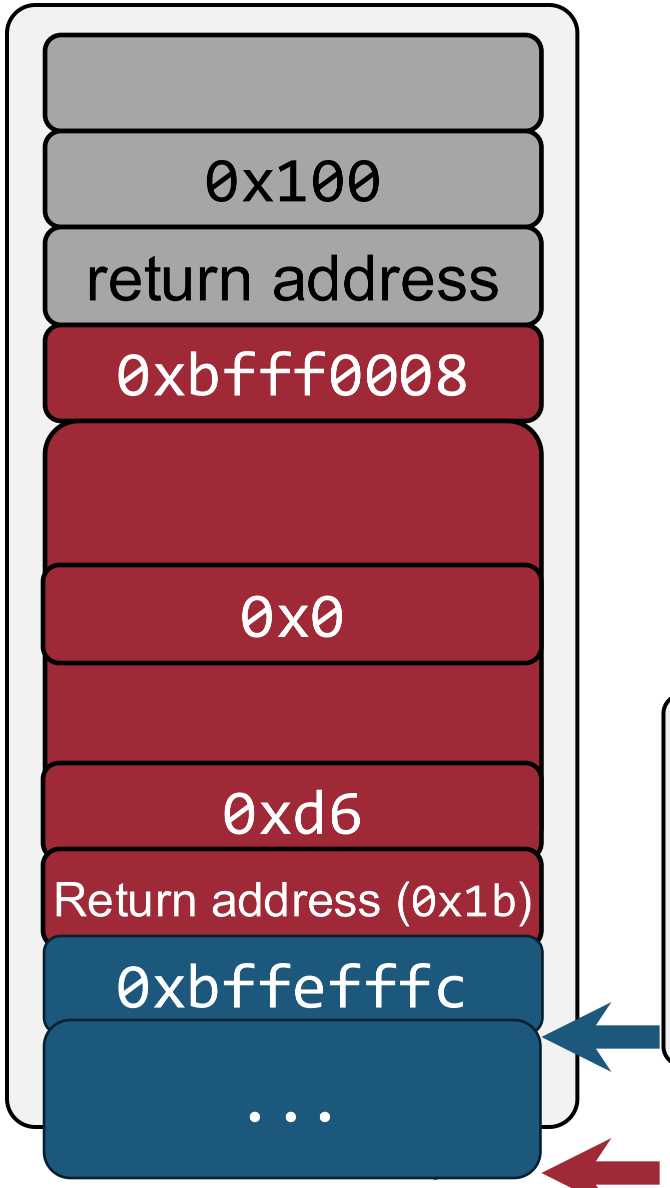

```
<red>:
   0:  push   ebp
   1:  mov    ebp,esp
   3:  sub    esp,0x28
   6:  mov    DWORD PTR [ebp-0xc],0x0
   d:  mov    eax,DWORD PTR [ebp+0x8]
  10:  sub    eax,0x2a
  13:  mov    DWORD PTR [esp],eax
  16:  call   blue
  1b:  mov    edx,DWORD PTR [ebp-0xc]
  1e:  add    eax,edx
  20:  leave
  21:  ret

<blue>:
  22:  push    ebp
  23:  mov     ebp,esp
  ...
  46:   leave
  47:   ret
```

Execution Example




```
<red>:
   0:  push   ebp
   1:  mov    ebp,esp
   3:  sub    esp,0x28
   6:  mov    DWORD PTR [ebp-0xc],0x0
   d:  mov    eax,DWORD PTR [ebp+0x8]
  10:  sub    eax,0x2a
  13:  mov    DWORD PTR [esp],eax
  16:  call   blue
  1b:  mov    edx,DWORD PTR [ebp-0xc]
  1e:  add    eax,edx
  20:  leave
  21:  ret

<blue>:
  22:  push    ebp
  23:  mov     ebp,esp
  ...
  46:   leave
  47:   ret
```

Execution Example



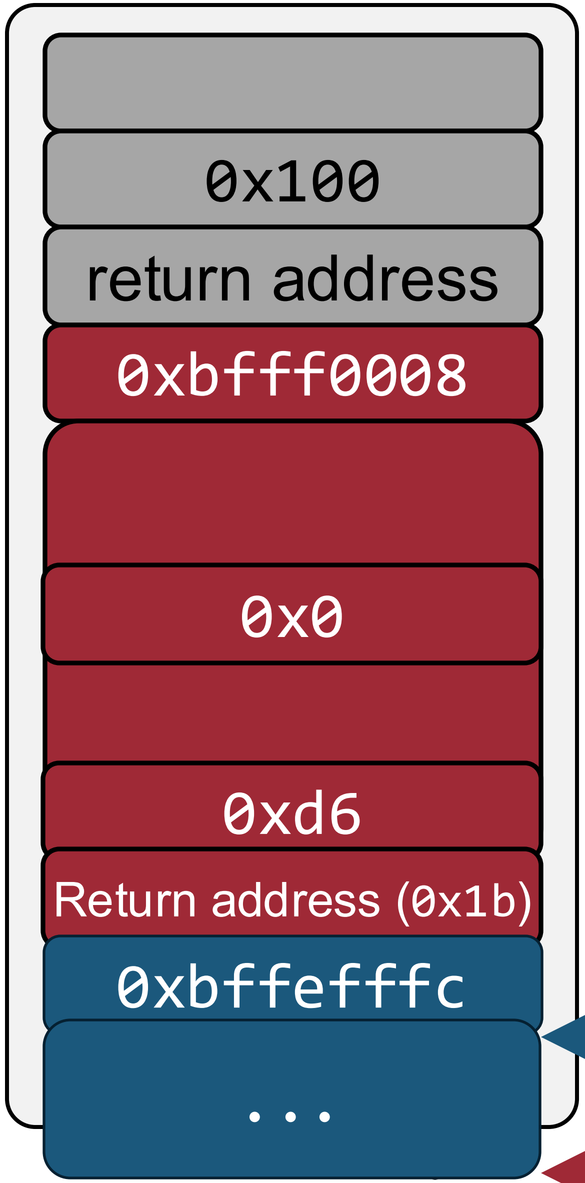
0xbffefffc

```
eip: 0x46
ebp: 0xbffeffcc
esp: 0xbffeffac
```

Execution context

```
<red>:
   0:  push   ebp
   1:  mov    ebp,esp
   3:  sub    esp,0x28
   6:  mov    DWORD PTR [ebp-0xc],0x0
   d:  mov    eax,DWORD PTR [ebp+0x8]
  10:  sub    eax,0x2a
  13:  mov    DWORD PTR [esp],eax
  16:  call   blue
  1b:  mov    edx,DWORD PTR [ebp-0xc]
  1e:  add    eax,edx
  20:  leave
  21:  ret

<blue>:
  22:  push    ebp
  23:  mov     ebp,esp
  ...                      Skip!
  46:   leave
  47:   ret
```

Execution Example



0xbffefffc

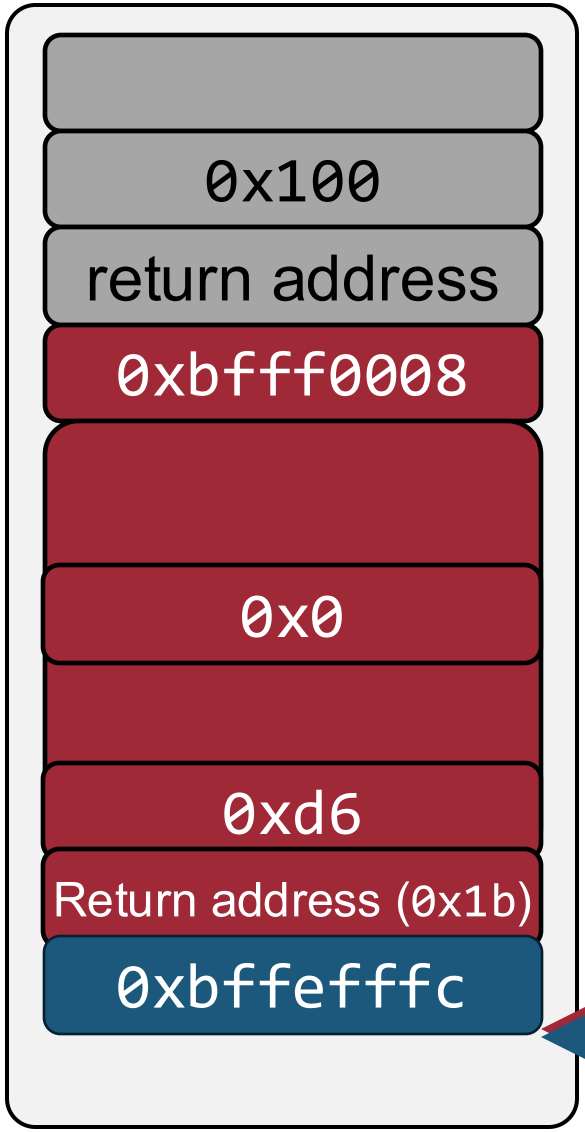
```
eip: 0x47
ebp: 0xbffeffcc
esp: 0xbffeffac
```

Execution context

```
<red>:
   0:  push   ebp
   1:  mov    ebp,esp
   3:  sub    esp,0x28
   6:  mov    DWORD PTR [ebp-0xc],0x0
   d:  mov    eax,DWORD PTR [ebp+0x8]
  10:  sub    eax,0x2a
  13:  mov    DWORD PTR [esp],eax
  16:  call   blue
  1b:  mov    edx,DWORD PTR [ebp-0xc]
  1e:  add    eax,edx
  20:  leave
  21:  ret

<blue>:
  22:  push    ebp
  23:  mov     ebp,esp
  ...
  46:  leave
  47:  ret
```

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

Execution Example



Virtual memory

0xbffefffc

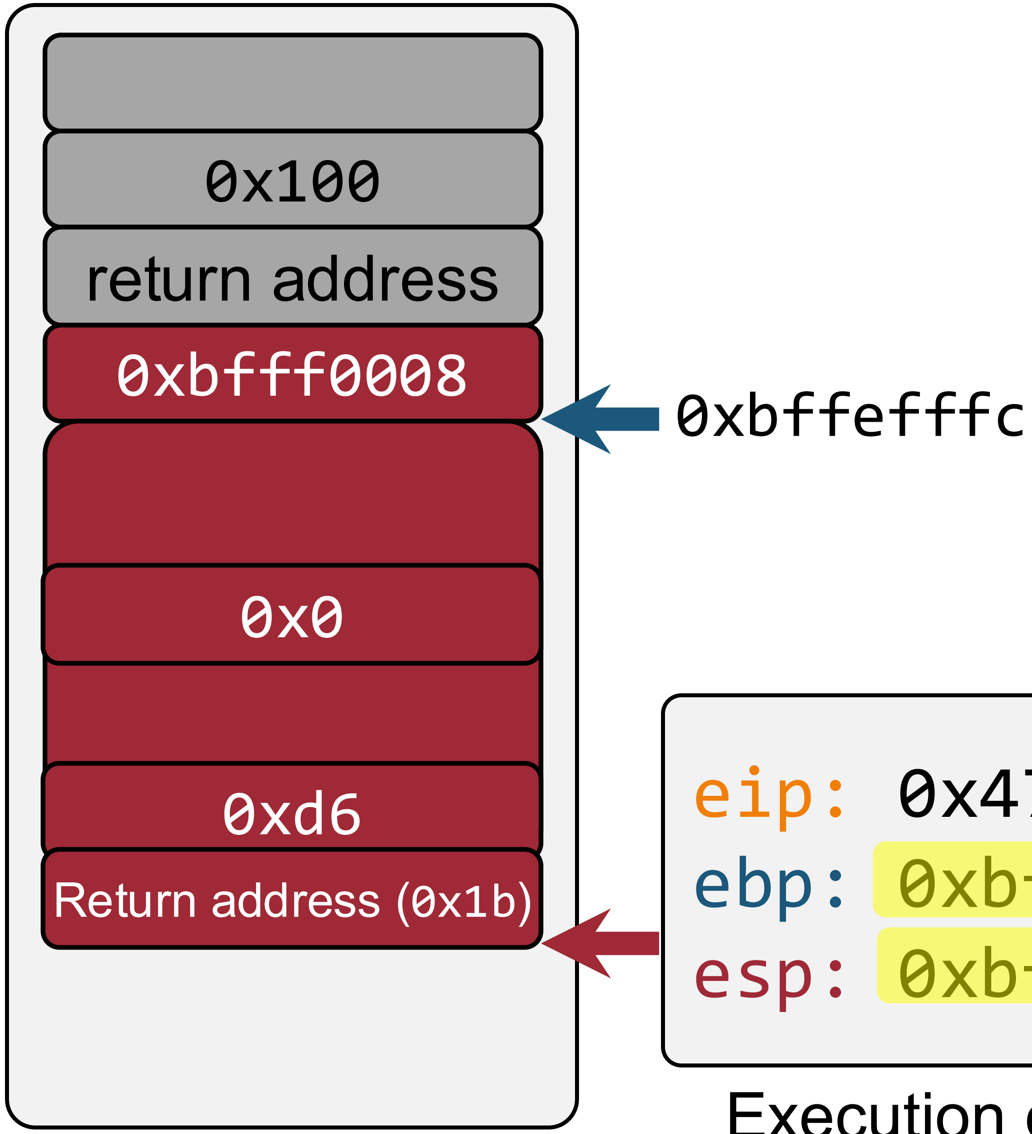
eip: 0x47
ebp: 0xbffeffcc
esp: 0xbffeffcc

Execution context

```
<red>:
   0:  push   ebp
   1:  mov    ebp,esp
   3:  sub    esp,0x28
   6:  mov    DWORD PTR [ebp-0xc],0x0
   d:  mov    eax,DWORD PTR [ebp+0x8]
  10:  sub    eax,0x2a
  13:  mov    DWORD PTR [esp],eax
  16:  call   blue
  1b:  mov    edx,DWORD PTR [ebp-0xc]
  1e:  add    eax,edx
  20:  leave
  21:  ret

<blue>:
  22:  push    ebp
  23:  mov     ebp,esp
  ...
  46:   leave
  47:   ret
```

mov esp, ebp
pop ebp

Execution Example



Virtual memory

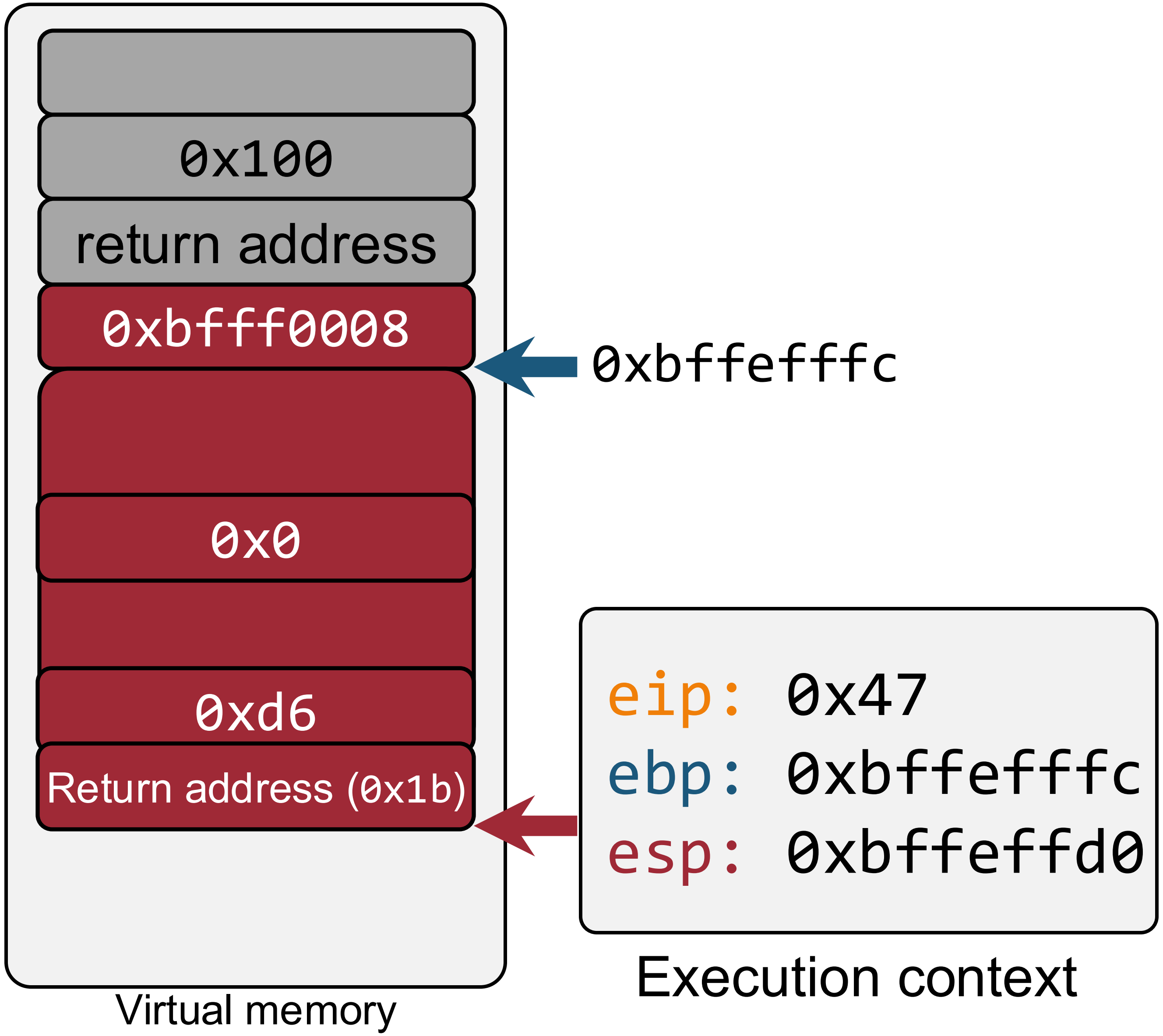
eip: 0x47
ebp: 0xbffefffc
esp: 0xbffeffd0

Execution context

```
<red>:
   0:  push   ebp
   1:  mov    ebp,esp
   3:  sub    esp,0x28
   6:  mov    DWORD PTR [ebp-0xc],0x0
   d:  mov    eax,DWORD PTR [ebp+0x8]
  10:  sub    eax,0x2a
  13:  mov    DWORD PTR [esp],eax
  16:  call   blue
  1b:  mov    edx,DWORD PTR [ebp-0xc]
  1e:  add    eax,edx
  20:  leave
  21:  ret

<blue>:
  22:  push    ebp
  23:  mov     ebp,esp
  ...
  46:  leave
  47:  ret
```

mov esp, ebp
pop ebp

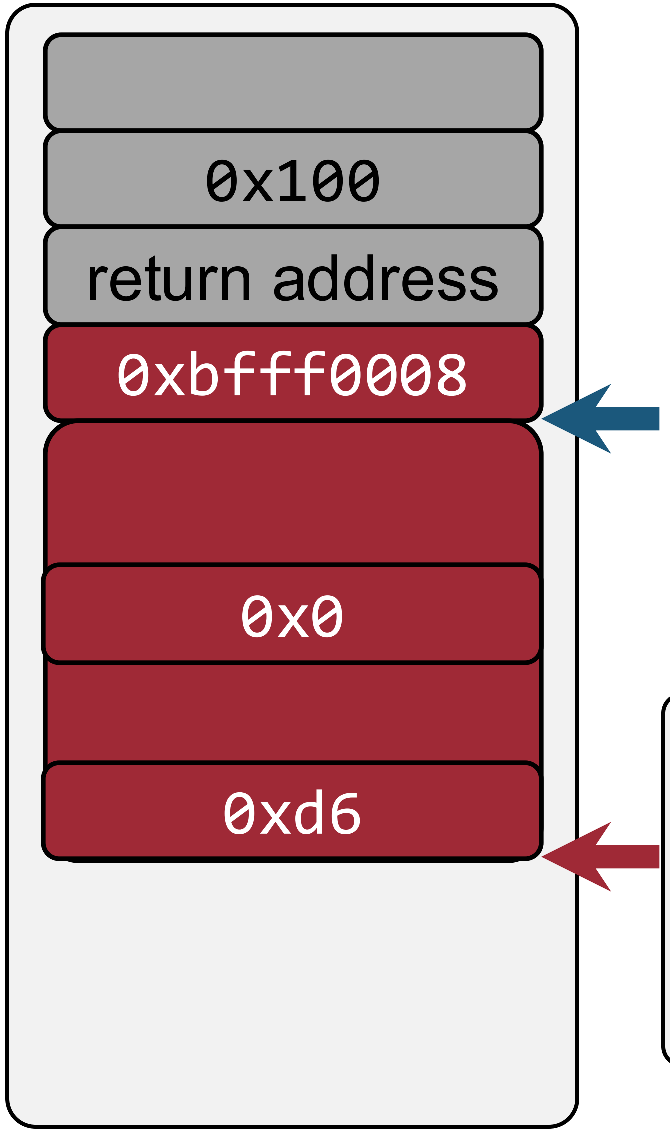
Execution Example




```
<red>:
   0:  push   ebp
   1:  mov    ebp,esp
   3:  sub    esp,0x28
   6:  mov    DWORD PTR [ebp-0xc],0x0
   d:  mov    eax,DWORD PTR [ebp+0x8]
  10:  sub    eax,0x2a
  13:  mov    DWORD PTR [esp],eax
  16:  call   blue
  1b:  mov    edx,DWORD PTR [ebp-0xc]
  1e:  add    eax,edx
  20:  leave
  21:  ret

<blue>:
  22:  push    ebp
  23:  mov     ebp,esp
  ...
  46:   leave
  47:   ret
```

pop eip

Execution Example



Virtual memory

eip: 0x1b
ebp: 0xbffefffc
esp: 0xbffeffd0

Execution context

```
<red>:
   0:  push   ebp
   1:  mov    ebp,esp
   3:  sub    esp,0x28
   6:  mov    DWORD PTR [ebp-0xc],0x0
   d:  mov    eax,DWORD PTR [ebp+0x8]
  10:  sub    eax,0x2a
  13:  mov    DWORD PTR [esp],eax
  16:  call   blue
  1b:  mov    edx,DWORD PTR [ebp-0xc]
  1e:  add    eax,edx
  20:  leave
  21:  ret

<blue>:
  22:  push    ebp
  23:  mov     ebp,esp
  ...
  46:   leave
  47:   ret
```

pop eip

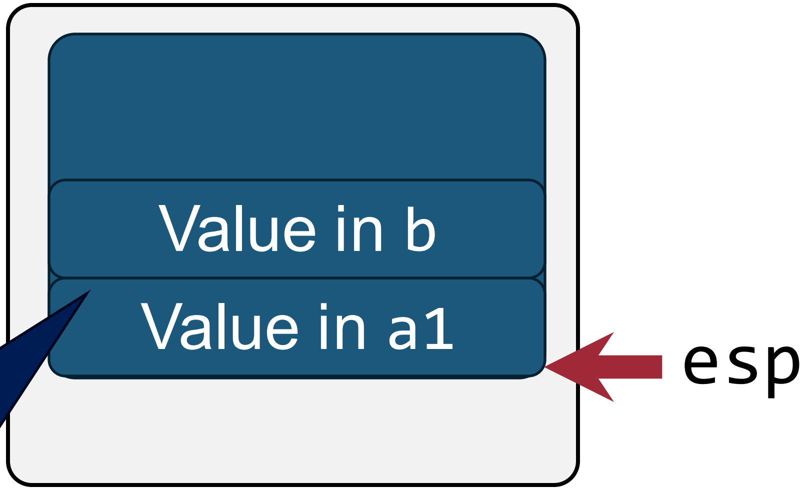
Calling Convention

```
<blue>:
  22:   push    ebp
  23:   mov     ebp,esp
  25:   sub     esp,0x28
  28:   mov     DWORD PTR [ebp-0xc],0x1
  2f:   mov     eax, DWORD PTR [ebp-0xc]
  32:   mov     DWORD PTR [esp+0x4], eax
  36:   mov     eax, DWORD PTR [ebp+0x8]
  39:   mov     DWORD PTR [esp], eax
  3c:   call    purple
  41:   mov     edx, DWORD PTR [ebp-0xc]
  44:   add     eax, edx
  46:   leave
  47:   ret
```

b

a1

```
int blue(int a1) {
    return 1 + purple(a1, b);
}
```



Virtual memory

Passing parameter values in a reverse order

Calling Convention

```
<blue>:
  22:   push    ebp
  23:   mov     ebp,esp
  25:   sub     esp,0x28
  28:   mov     DWORD PTR [ebp-0xc],0x1
  2f:   mov     eax, DWORD PTR [ebp-0xc]
  32:   mov     DWORD PTR [esp+0x4], eax
  36:   mov     eax, DWORD PTR [ebp+0x8]
  39:   mov     DWORD PTR [esp], eax
  3c:   call    purple
  41:   mov     edx, DWORD PTR [ebp-0xc]
  44:   add     eax, edx
  46:   leave
  47:   ret
```

b

a1

```c
int blue(int a1) {
    return 1 + purple(a1, b);
}
```

Storing a return value in eax

Key Concepts

- Compilation pipeline
- x86 architecture
- Assembly
- Disassembly

Question?