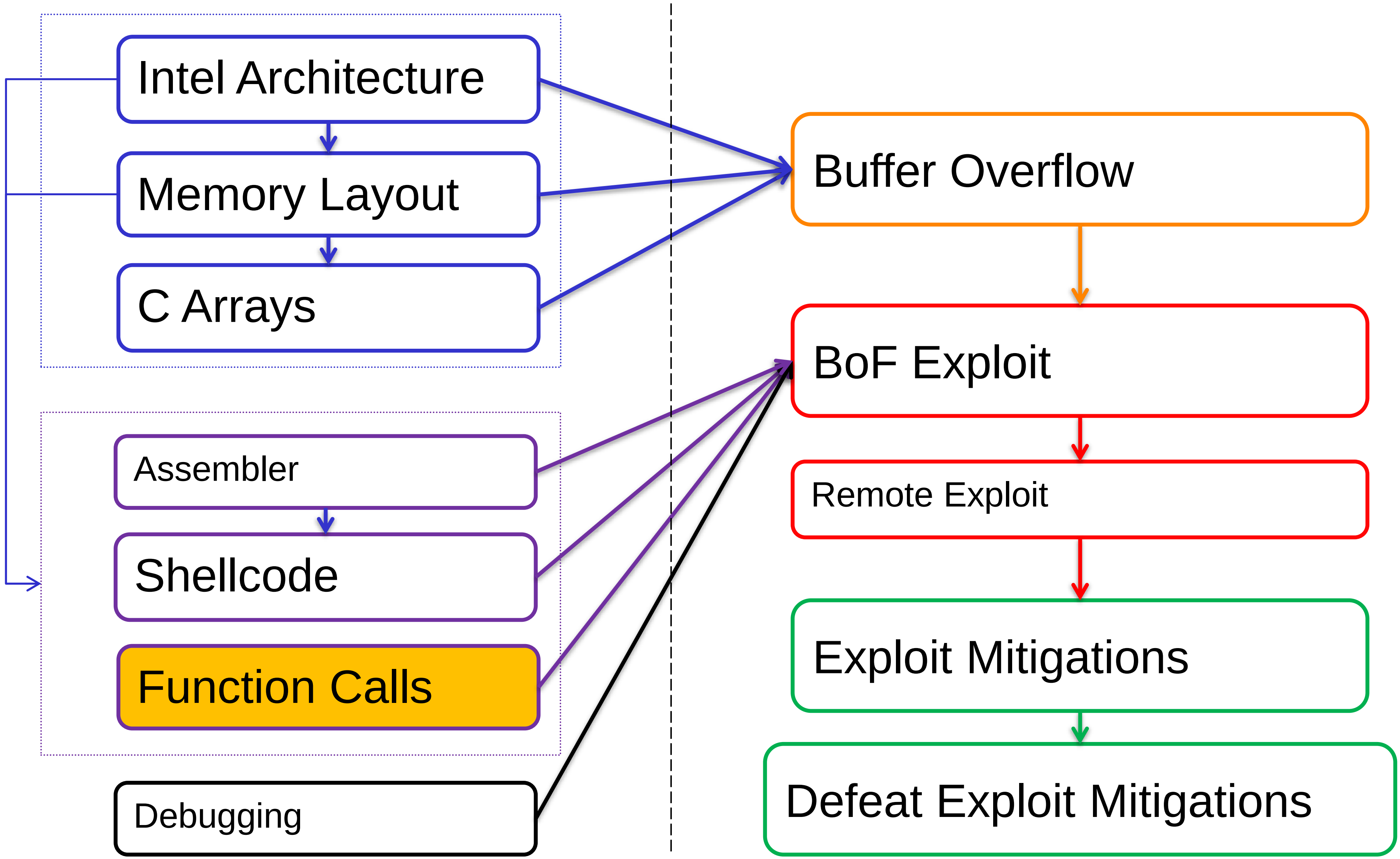
Function Call Convention

Content


Intel Architecture
Memory Layout
C Arrays
Assembler
Shellcode
Function Calls
Debugging
Buffer Overflow
BoF Exploit
Remote Exploit
Exploit Mitigations
Defeat Exploit Mitigations

Function Call Convention

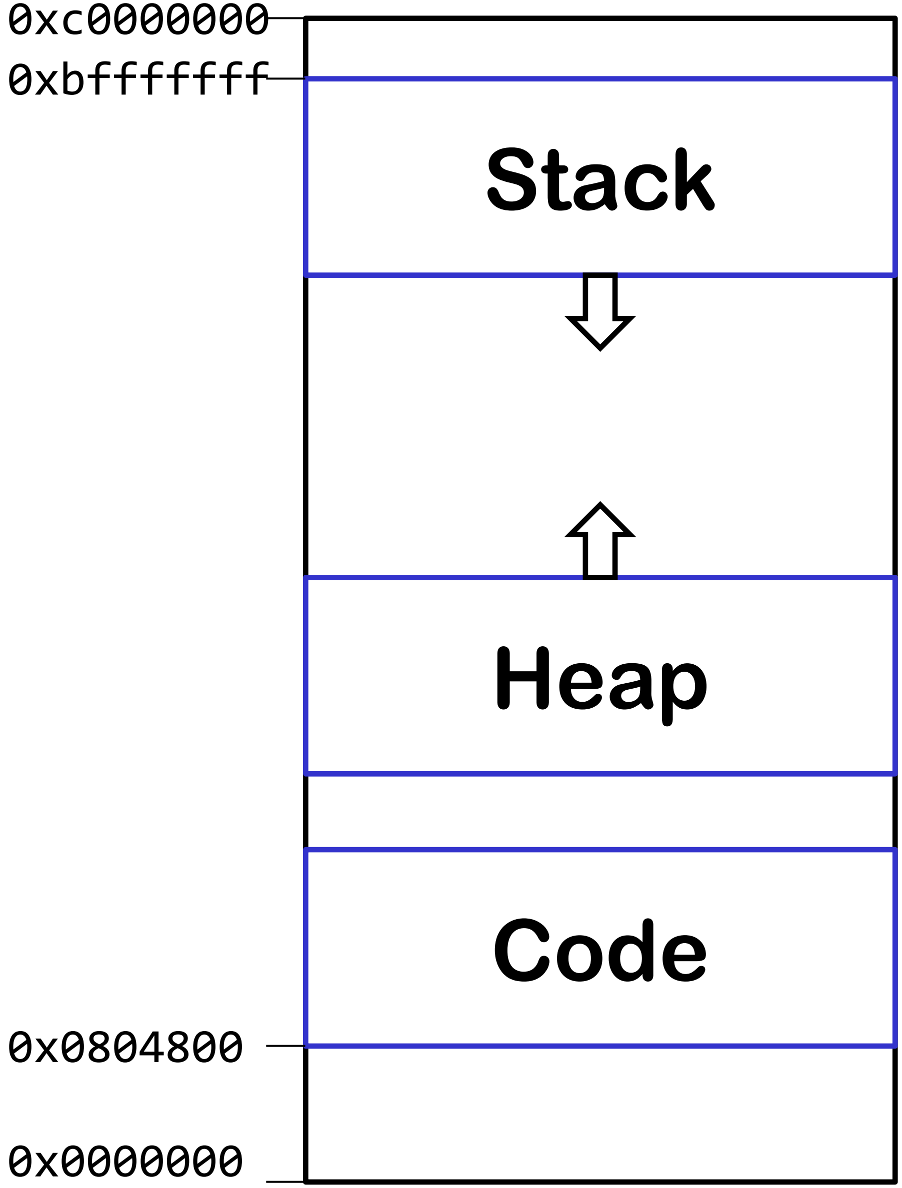
Function call convention:

- How functions work
- Program-metadata on the stack

Stack based buffer overflow:

- Overwrite program-metadata on the stack

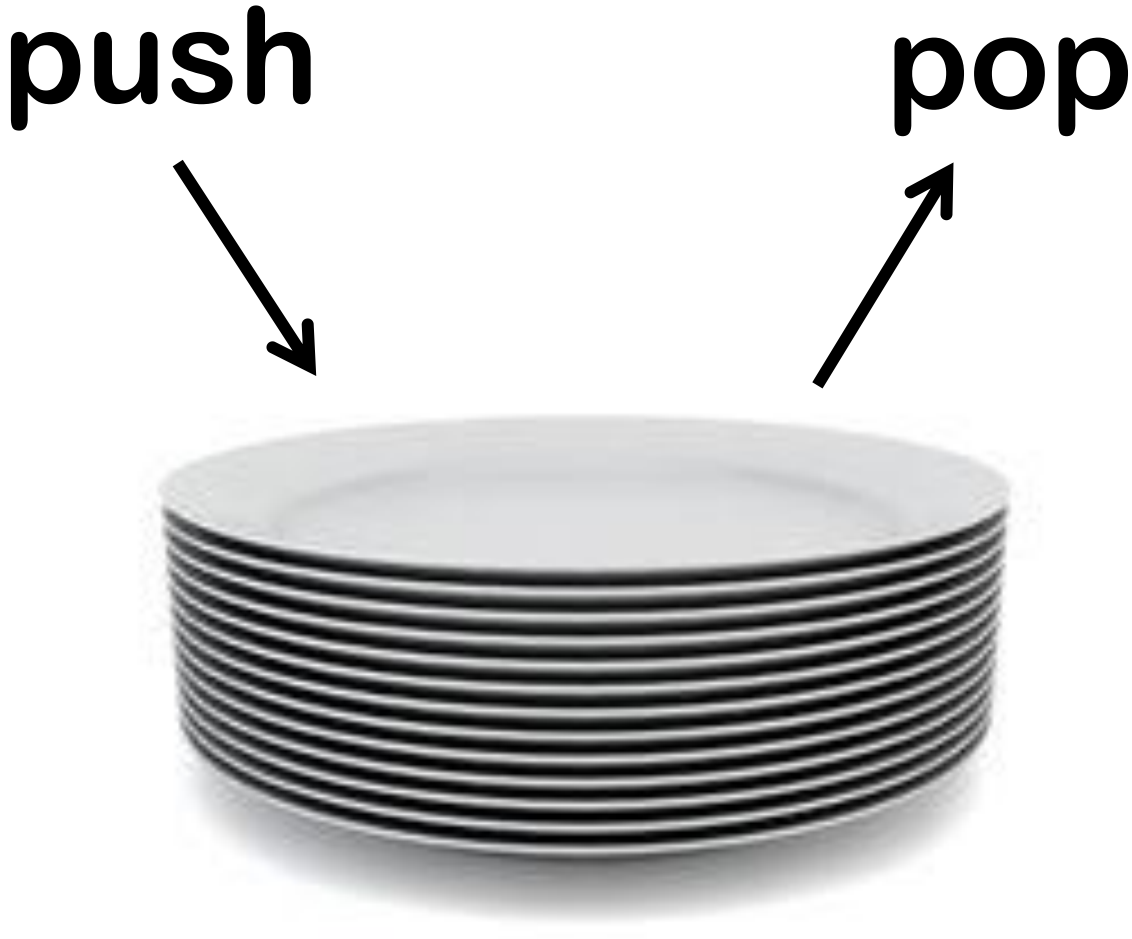
x32 Memory Layout



Stacks

How do they work?

Stack



Stack

0x10000

0x00010

push

pop

Stack

```
push 0x1
push 0x2
push 0x3
pop
push 0x4
```

```
push 0x1
push 0x2
push 0x3
pop
push 0x4
```

0x01

```
push 0x1
push 0x2
push 0x3
pop
push 0x4
```

0x01
0x02

Stack

push 0x1

push 0x2

push 0x3

pop

push 0x4

0x01
0x02
0x03

```
push 0x1
push 0x2
push 0x3
pop
push 0x4
```

0x01
0x02

Stack

```
push 0x1
push 0x2
push 0x3
pop
push 0x4
```

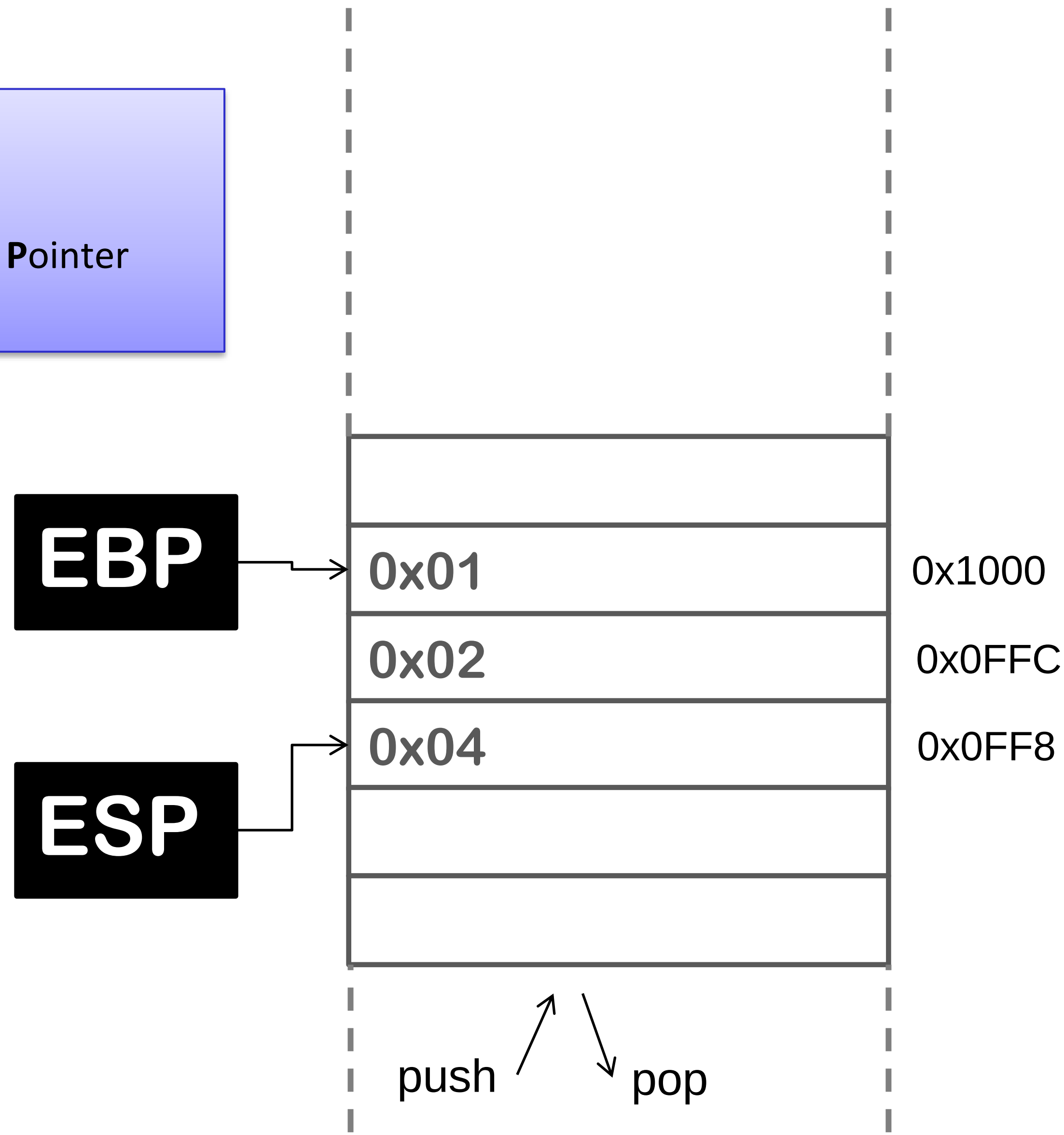
0x01
0x02
0x04

Stack on intel

Intel stack registers:

- **ESP**: **St**ack **P**ointer
- **EBP**: (Stack-) **B**ase **P**ointer

EBP = 0x1000
ESP = 0x0FF8



Stack in computers

Stack is using process memory as basis

CPU instruction support (because stack is so useful)

Note:

- CPU instructions like push/pop are just for ease of use
- The “stack values” can be accessed (read, write) like every other memory address
- You can point the stack (ebp, esp) to wherever in the memory you want
- There’s usually just ONE stack per process (thread)

x32 Call Convention

Functions and the Stack

What is a function?

- Self contained subroutine
- Re-usable
- Can be called from anywhere
- After function is finished: Jump to the calling function (calee)

```
void main(void) {
  int blubb = 0;
  foobar(blubb);
  return;
}

void foobar (int arg1) {
  char compass1[];
  char compass2[];
}
```

What does the function foobar() need?

- Function Argument:
  - blubb
- Local variables
  - Compass1
  - Compass2
- And: Address of next instruction in main()
  - &return

x32 Call Convention

Saved IP (&__libc_start)

Saved Frame Pointer

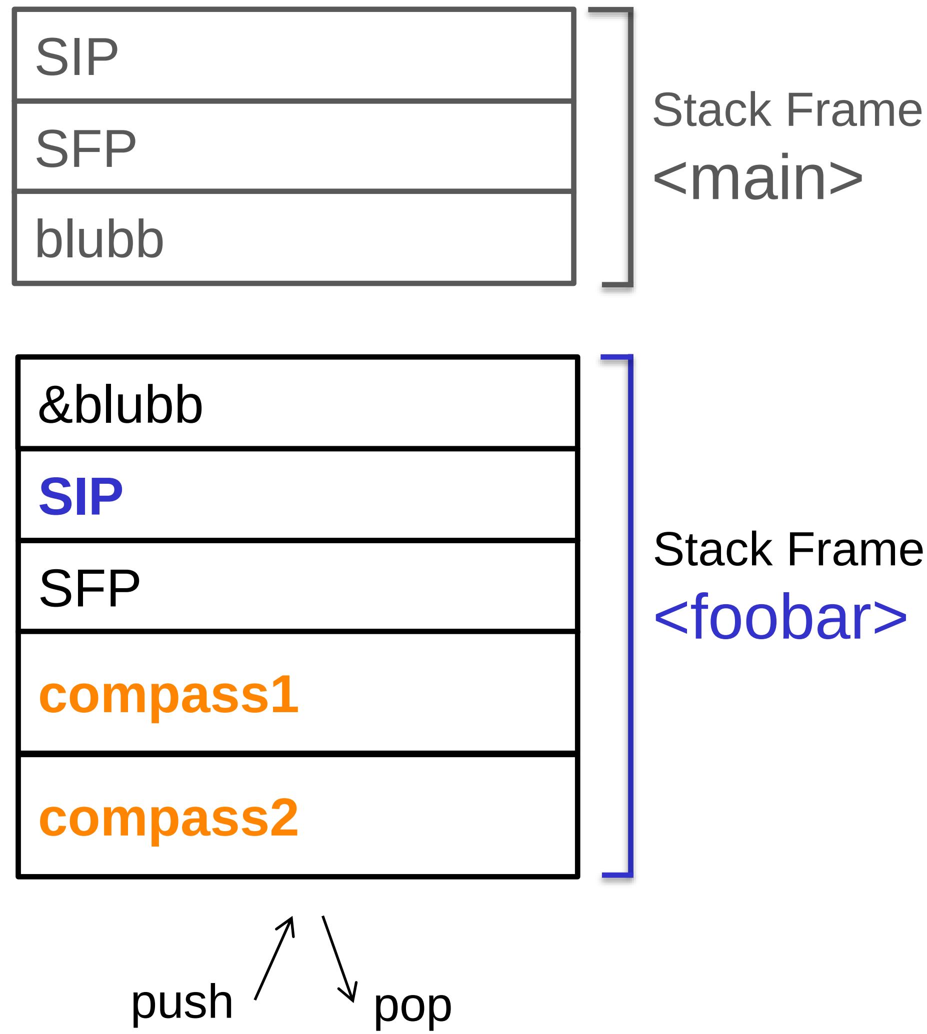
Local Variables <main>

Argument for <foobar>

Saved IP (&return)

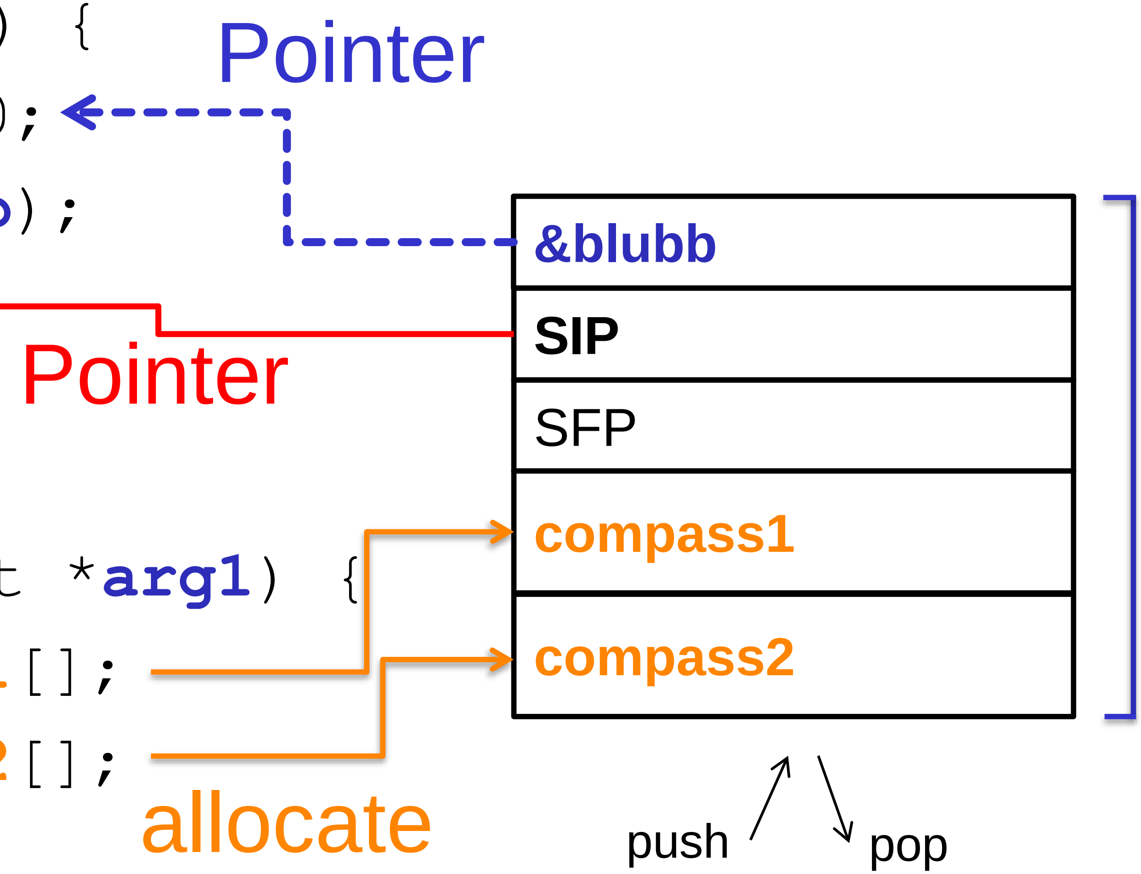
Saved Frame Pointer

Local Variables <foobar>

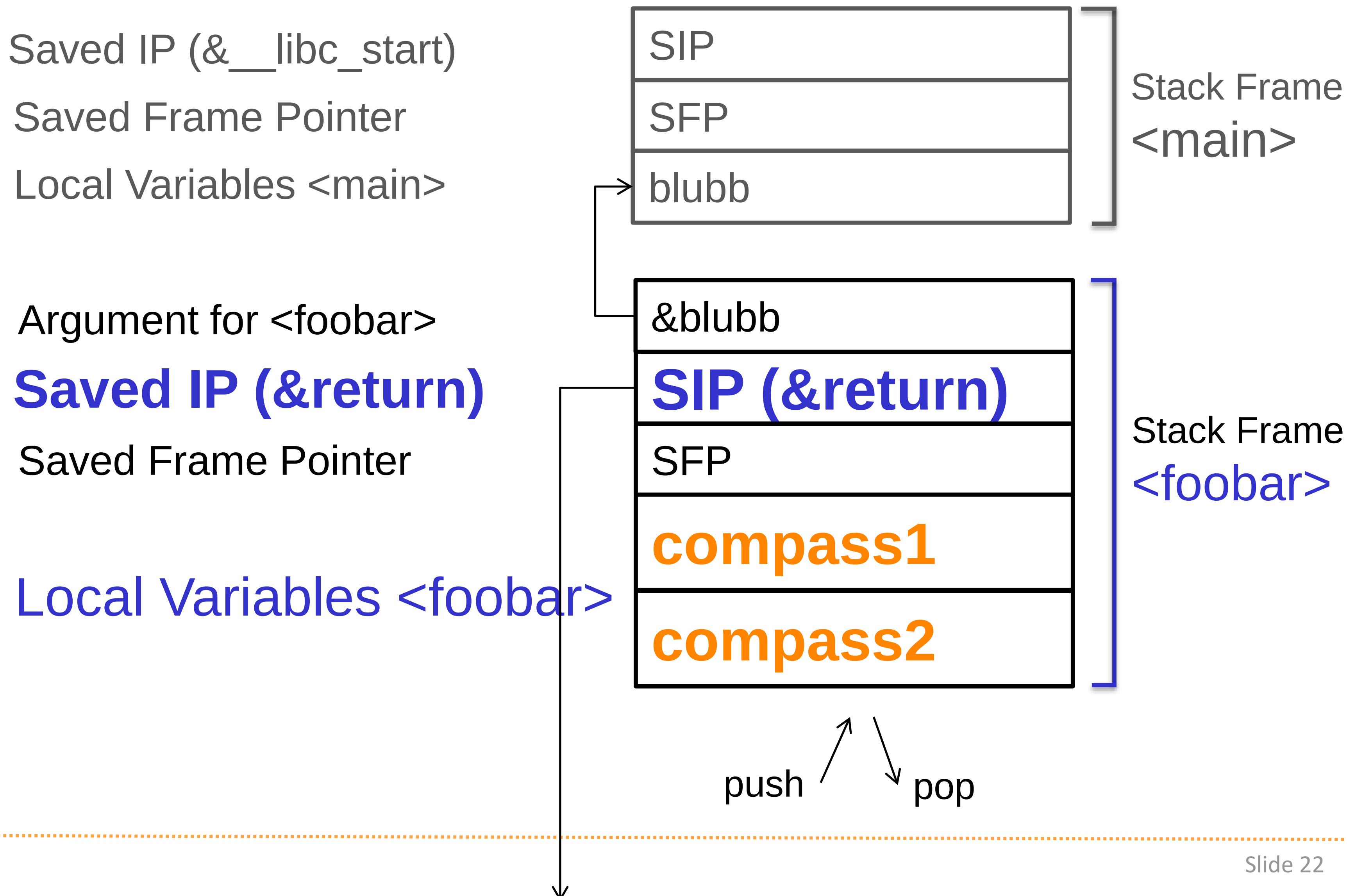


x32 Call Convention

```
void main(void) {
  int blubb = 0;
  foobar(&blubb);
  return;
}

void foobar(int *arg1) {
  char compass1[];
  char compass2[];
}
```



x32 Call Convention



SIP: Stored Instruction Pointer

- Copy of EIP
- Points to the address where control flow continues after end of function
  - (return, ret)
- Usually points into the code section

SBP: Stored Base Pointer

- Copy of EBP
- Every function has its own little stack frame
- Stack frame is where local variables, function arguments etc. are
- A function should only access its own stack frame
- Most of the function epilogue and prologue handle setting up and removing the stack frame
- Note: It is not 100% necessary to completely understand it - but you will see it in the disassembly of every function
- Note: You can compile programs without using SBP (ASM will be a bit harder to read)

x32 Call Convention

Attention! Assembler ahead!

- AT&T vs Intel syntax

Intel syntax:

```
mov     eax,1
mov     ebx,0ffh
int     80h
```

AT&T syntax:

```
movl    $1,%eax
movl    $0xff,%ebx
int     $0x80
```

Don’t hang me if I messed this up somewhere

In ASM:

call 0x11223344 <&foobar>

push EIP

jmp 0x11223344

<function code> (0x11223344)

ret

pop eip

x32 Call Convention

In ASM:

```
call 0x11223344 <&foobar>
        push EIP
        jmp 0x11223344
mov ebp, esp
<function code>
mov esp, ebp
ret
        pop eip
```

x32 Call Convention

In ASM:

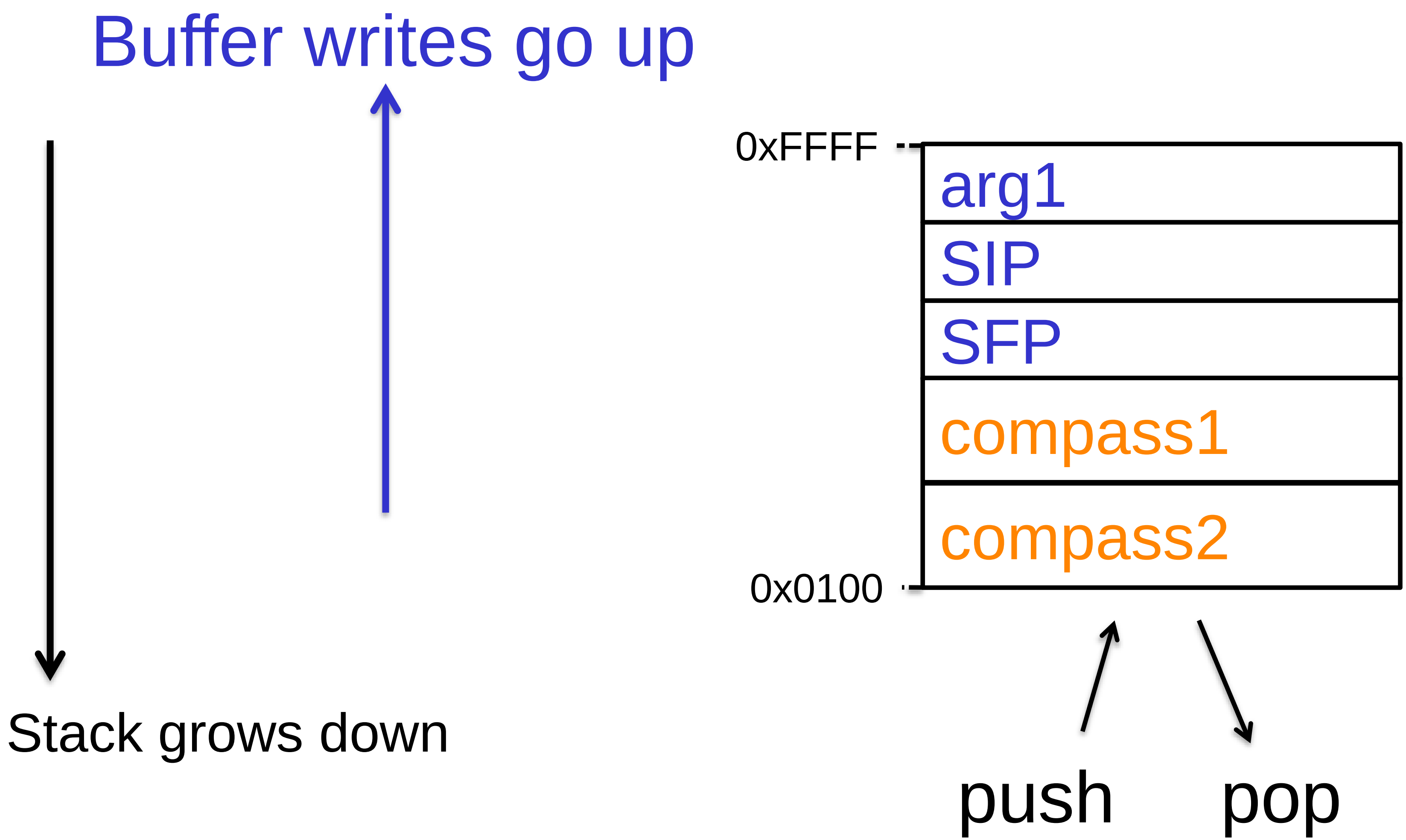
```
call 0x11223344 <&foobar>
     push EIP
     jmp 0x11223344
mov ebp, esp
<function code>
mov esp, ebp
ret
     pop eip
```

Prolog

Function

Epilog

x32 Call Convention



x32 Call Convention

Recap:

- User data is on the stack
- Also: important stuff is on the stack (Instruction Pointer, SIP)
- Stack grows down ⇩
- Writes go up ⇧

x32 Call Convention Details

x32 Call Convention Details

```
int add(int x, int y) {
  int sum;
  sum = x + y;
  return sum;
}
```

x32 Call Convention Details

```
c = add(3, 4)
```

C

```
push 4

push 3

call add
```

ASM

```
push 4

push 3

push EIP

jmp add
```

ASM, detailed

x32 Call Convention Details

add():

```
push 4
push 3
push EIP
jmp add
```

```
push ebp
mov ebp, esp,
sub esp, 0x10

mov eax, DWORD PTR [ebp + 0xc]
mov edx, DWORD PTR [ebp + 0x8]
add eax, edx
mov DWORD PTR [ebp – 0x04], eax
mov eax, DWORD PTR [ebp – 0x04]

leave
ret
```

x32 Call Convention Details

```
push 4
push 3
push EIP
jmp add
```

add():

```
push ebp
mov ebp, esp,
sub esp, 0x10

mov eax, DWORD PTR [ebp + 0xc]
mov edx, DWORD PTR [ebp + 0x8]
add eax, edx
mov DWORD PTR [ebp – 0x04], eax
mov eax, DWORD PTR [ebp – 0x04]

mov esp, ebp        ; leave
pop ebp             ; leave
ret
```

x32 Call Convention Details

add():

```
push 4
push 3
push EIP
jmp add
```

```
push ebp
mov ebp, esp,
sub esp, 0x10

mov eax, DWORD PTR [ebp + 0xc]
mov edx, DWORD PTR [ebp + 0x8]
add eax, edx
mov DWORD PTR [ebp – 0x04], eax
mov eax, DWORD PTR [ebp – 0x04]

mov esp, ebp        ; leave
pop ebp             ; leave
pop eip             ; ret
```

x32 Call Convention Details

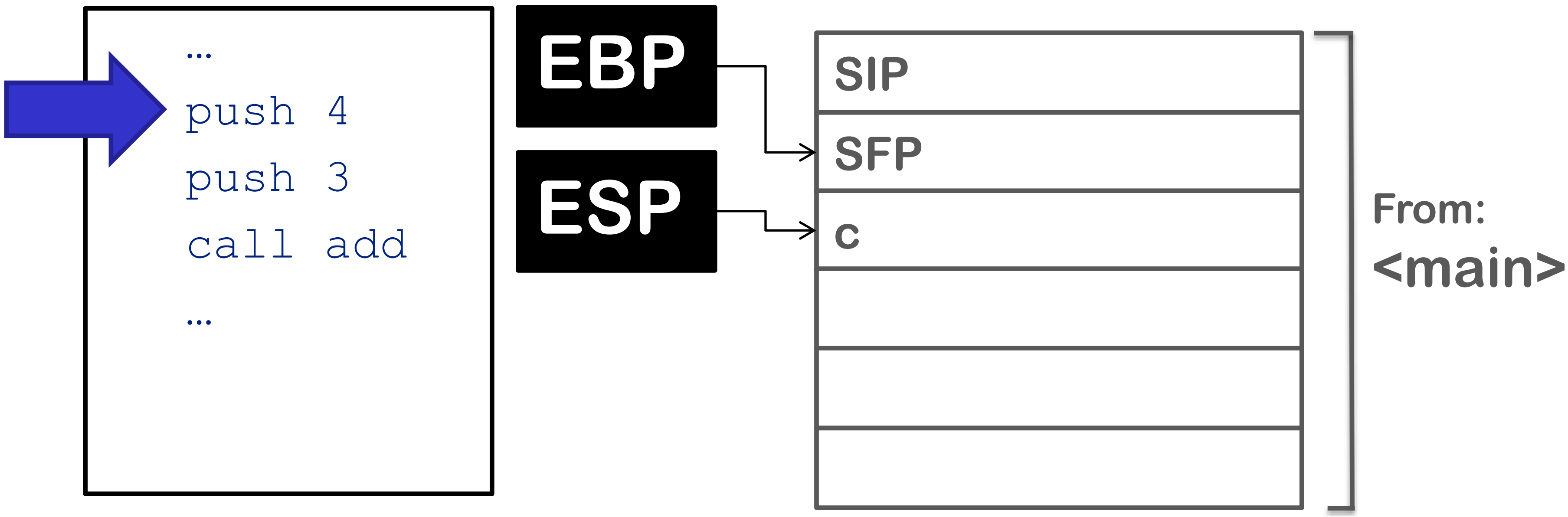
add():

```
push 4
push 3
push EIP
jmp add
```

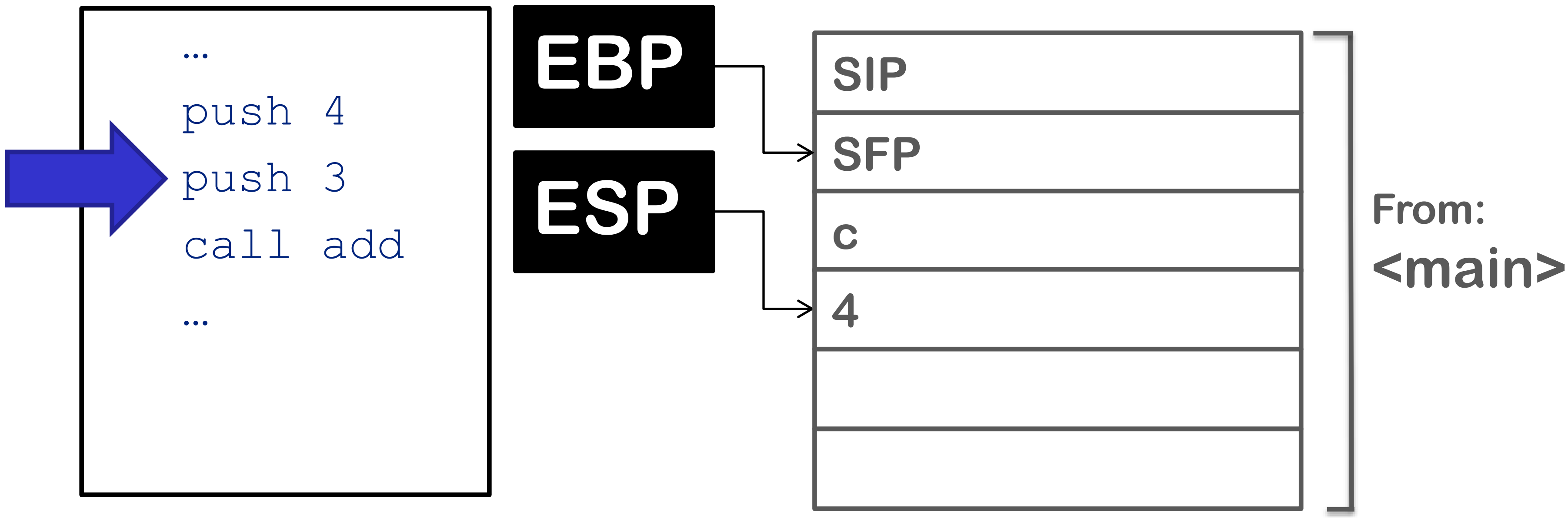
```
push ebp
mov ebp, esp,
sub esp, 0x10


mov esp, ebp        ; leave
pop ebp             ; leave
pop eip             ; ret
```

Function Prolog

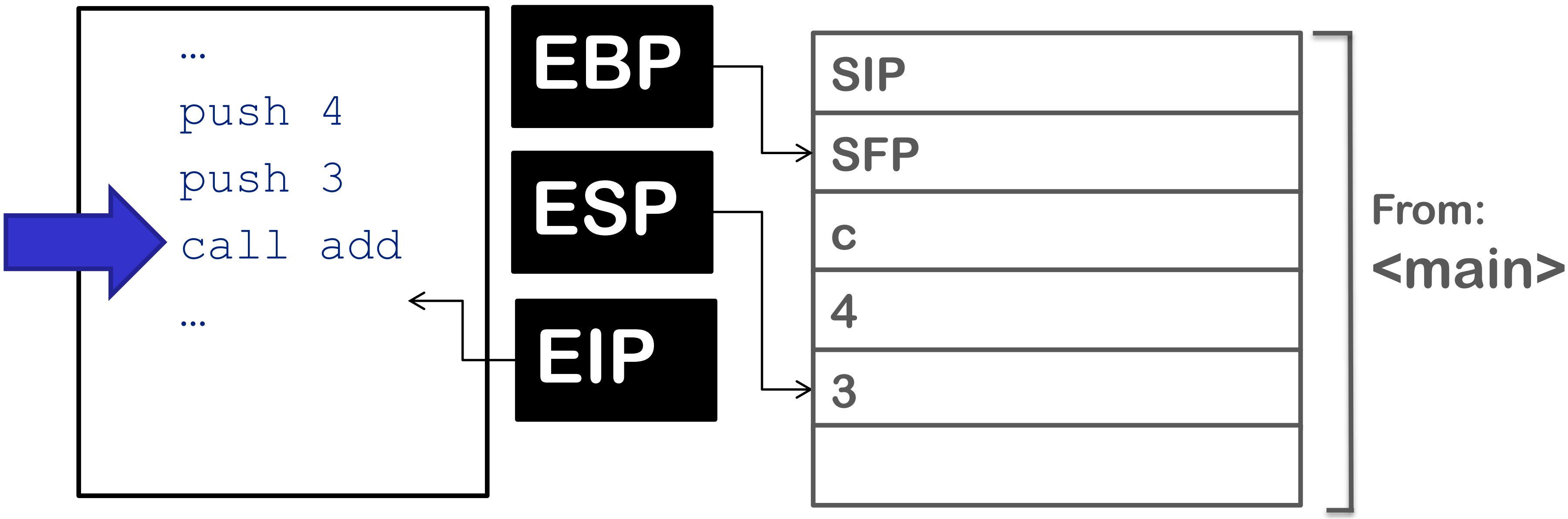
x32 Call Convention - Function Prolog



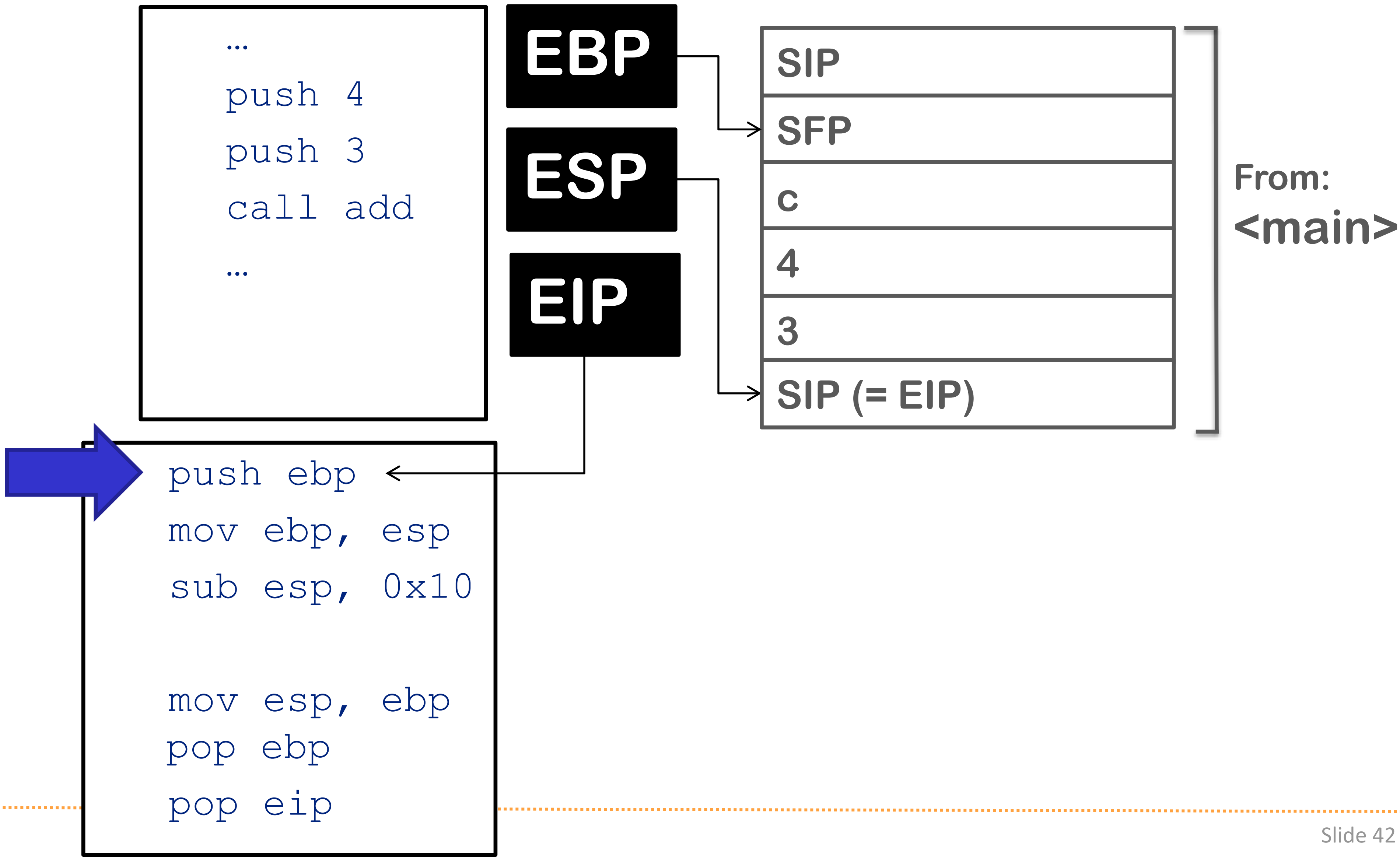
x32 Call Convention - Function Prolog



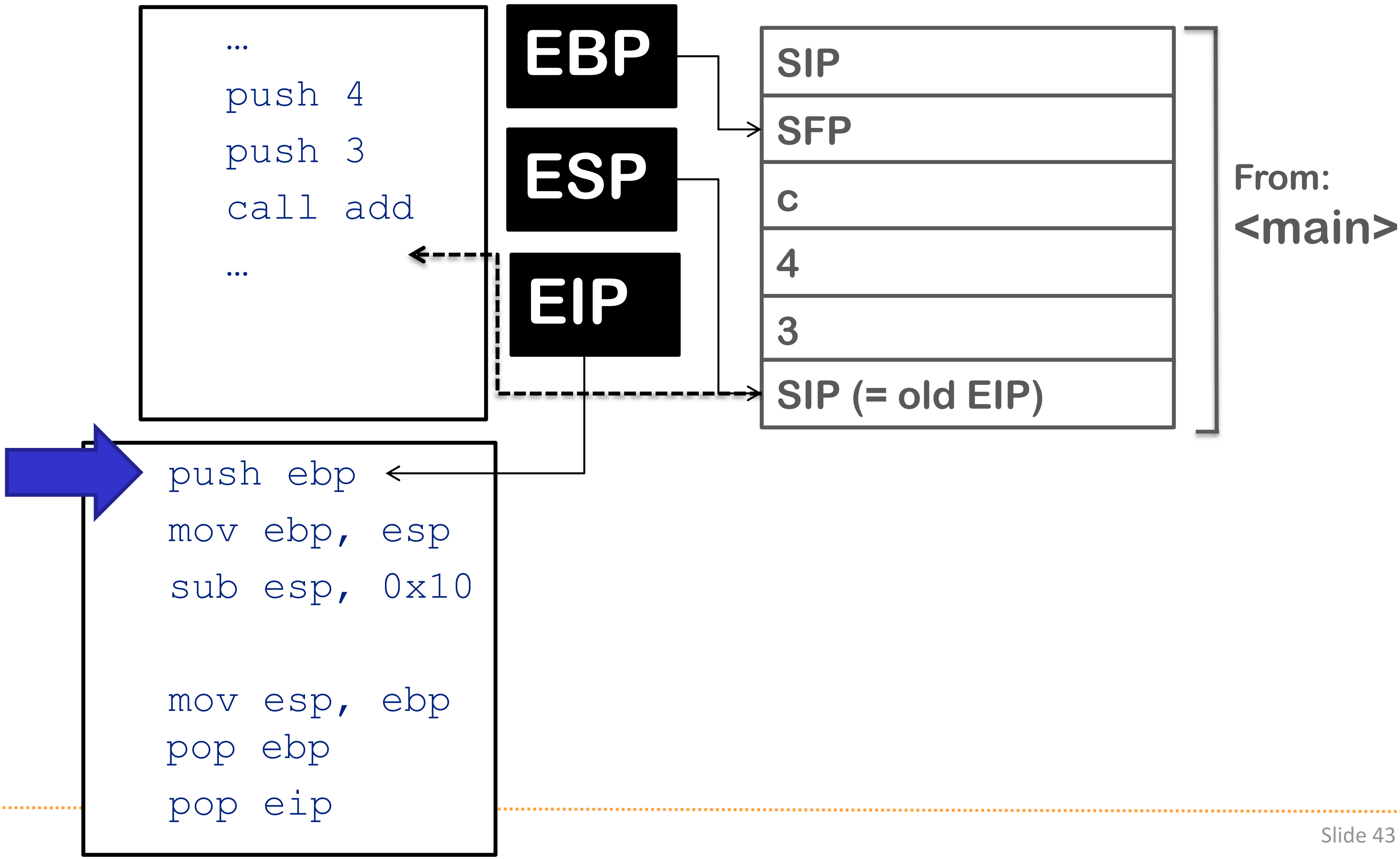
x32 Call Convention - Function Prolog



x32 Call Convention - Function Prolog

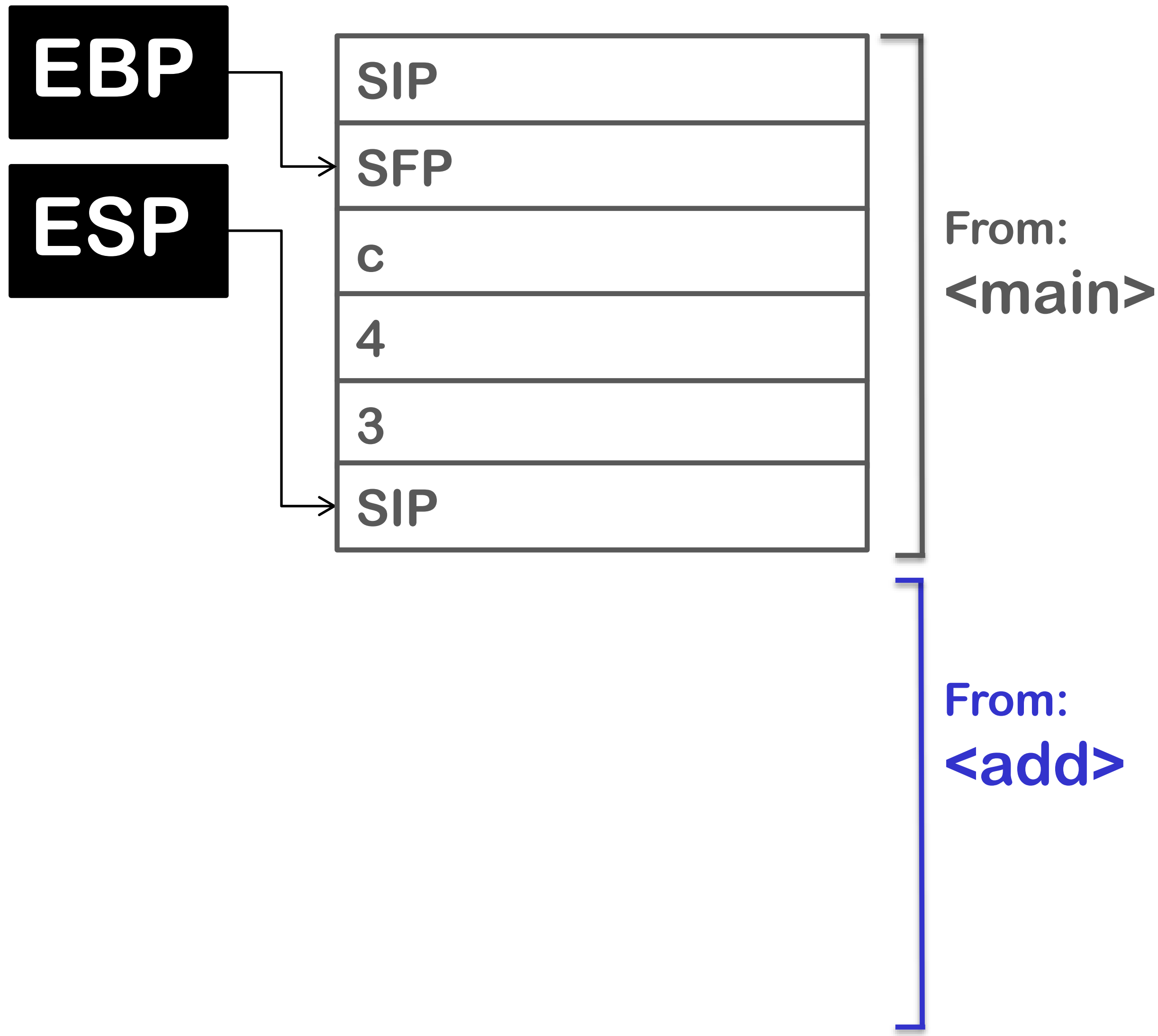


x32 Call Convention - Function Prolog



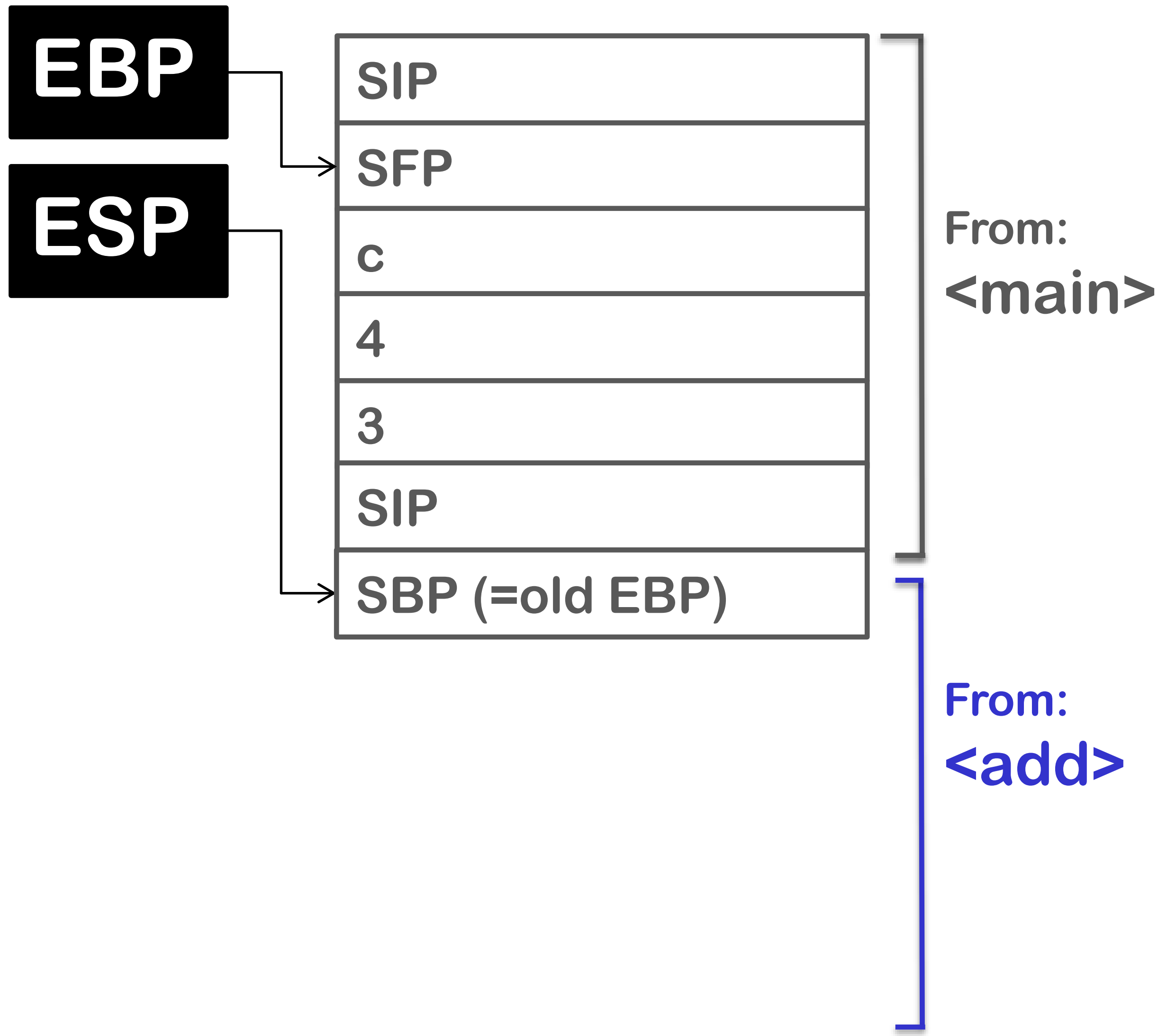
x32 Call Convention - Function Prolog

```
push ebp
mov ebp, esp
sub esp, 0x10

mov esp, ebp
pop ebp
pop eip
```



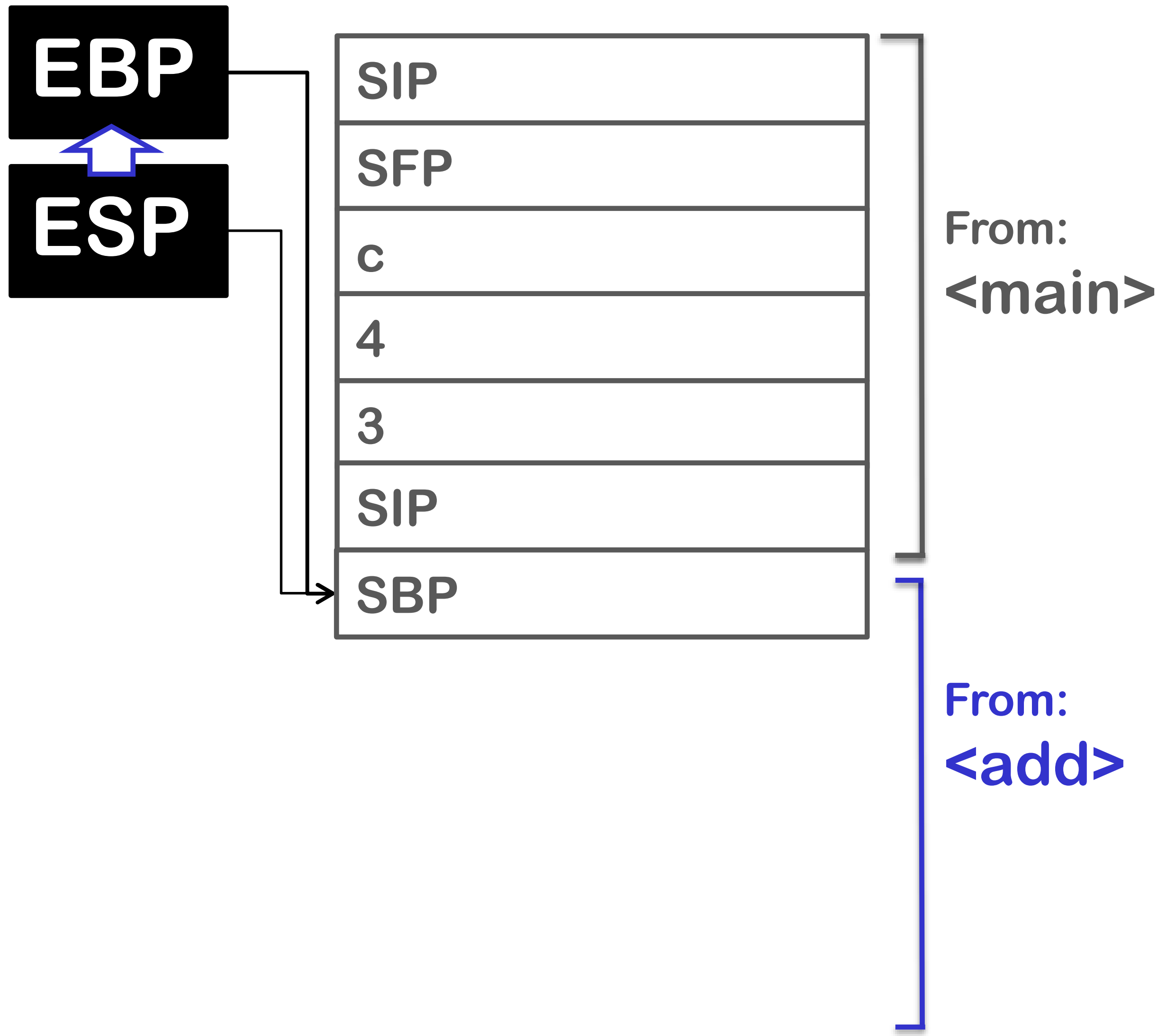
x32 Call Convention - Function Prolog

```
push ebp
mov ebp, esp
sub esp, 0x10

mov esp, ebp
pop ebp
pop eip
```



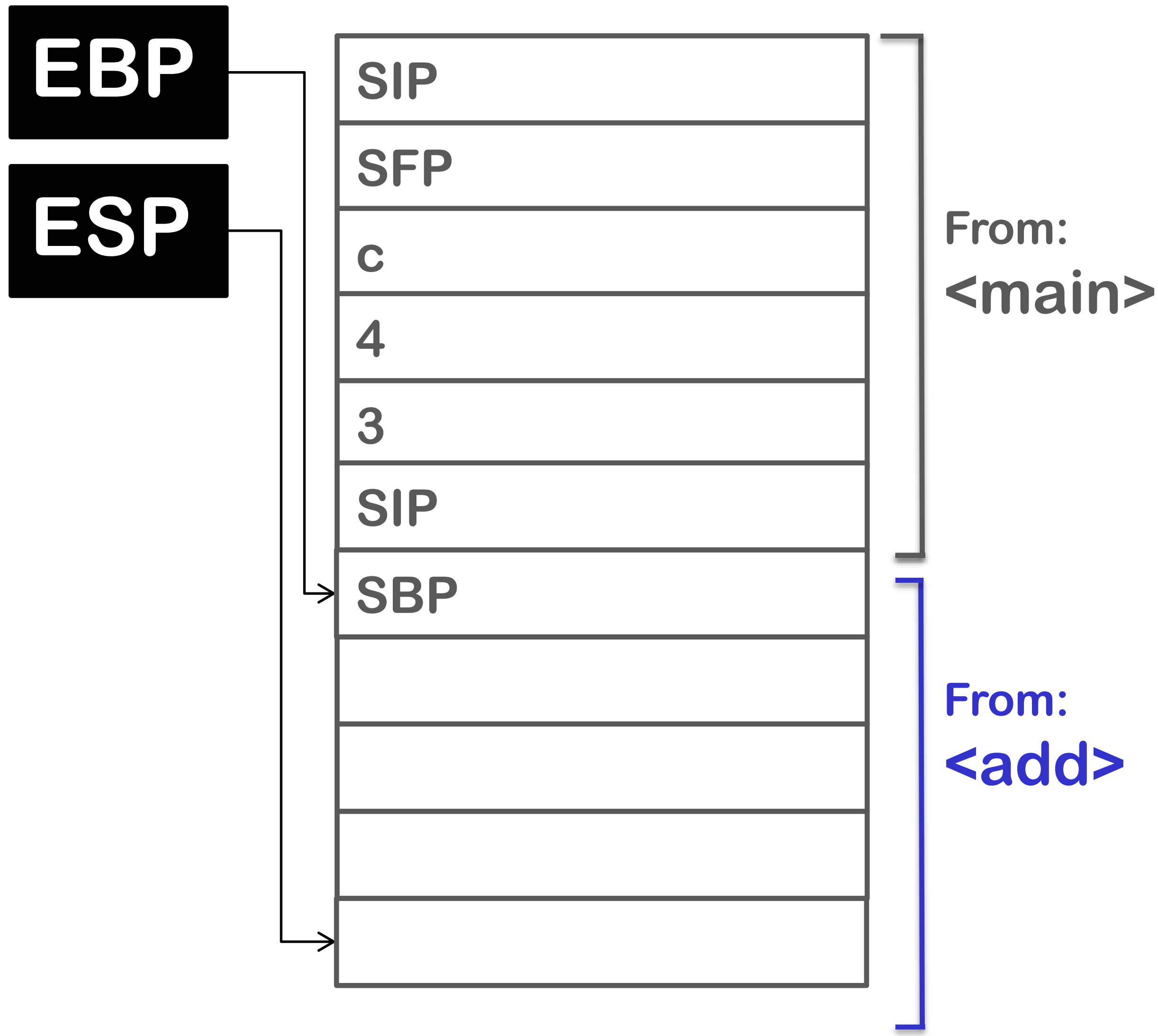
x32 Call Convention - Function Prolog

```
push ebp
mov ebp, esp
sub esp, 0x10

mov esp, ebp
pop ebp
pop eip
```



x32 Call Convention - Function Prolog

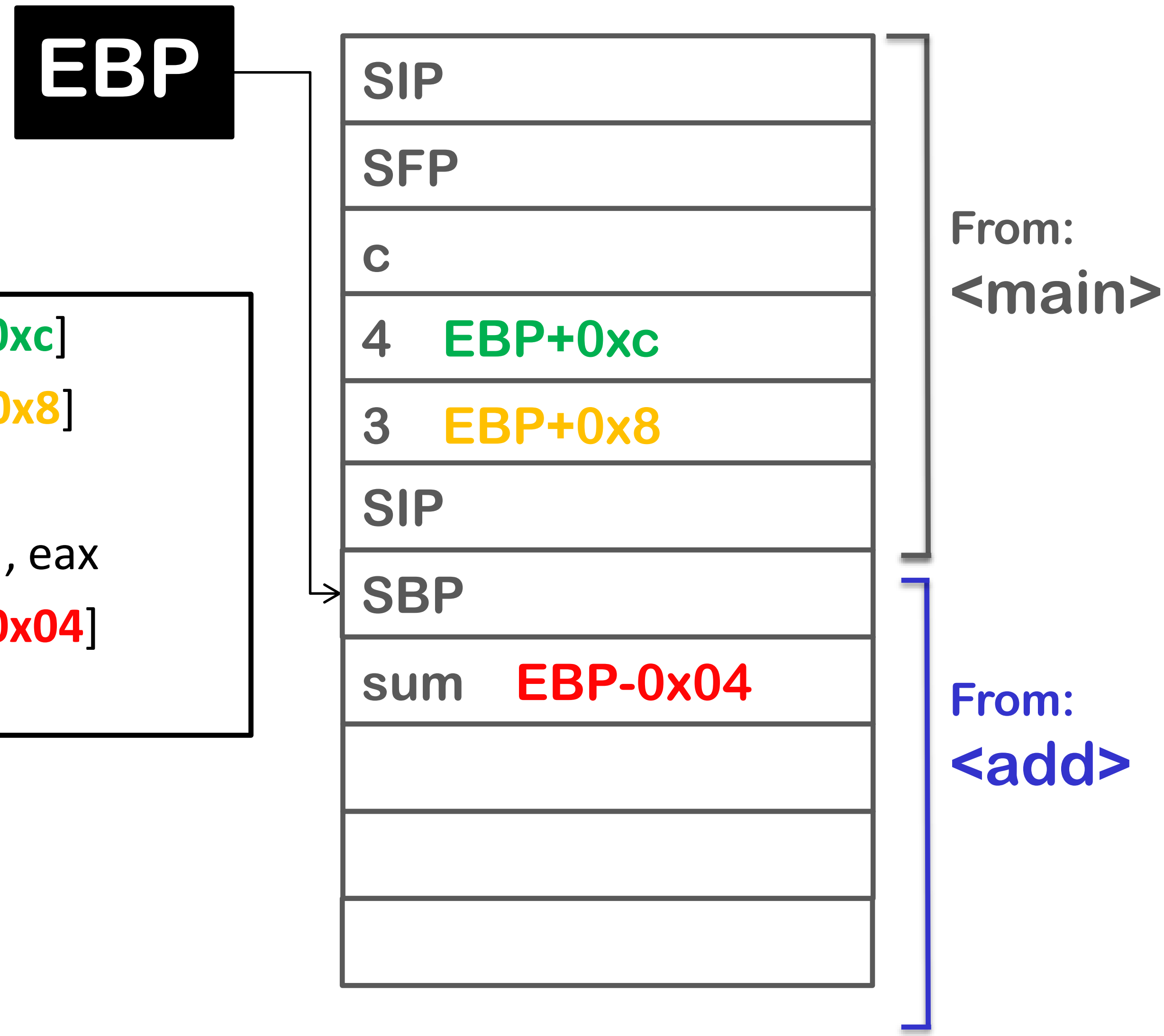
```
push ebp
mov ebp, esp
sub esp, 0x10

mov esp, ebp
pop ebp
pop eip
```



Execute Function

x32 Call Convention - Execute Function

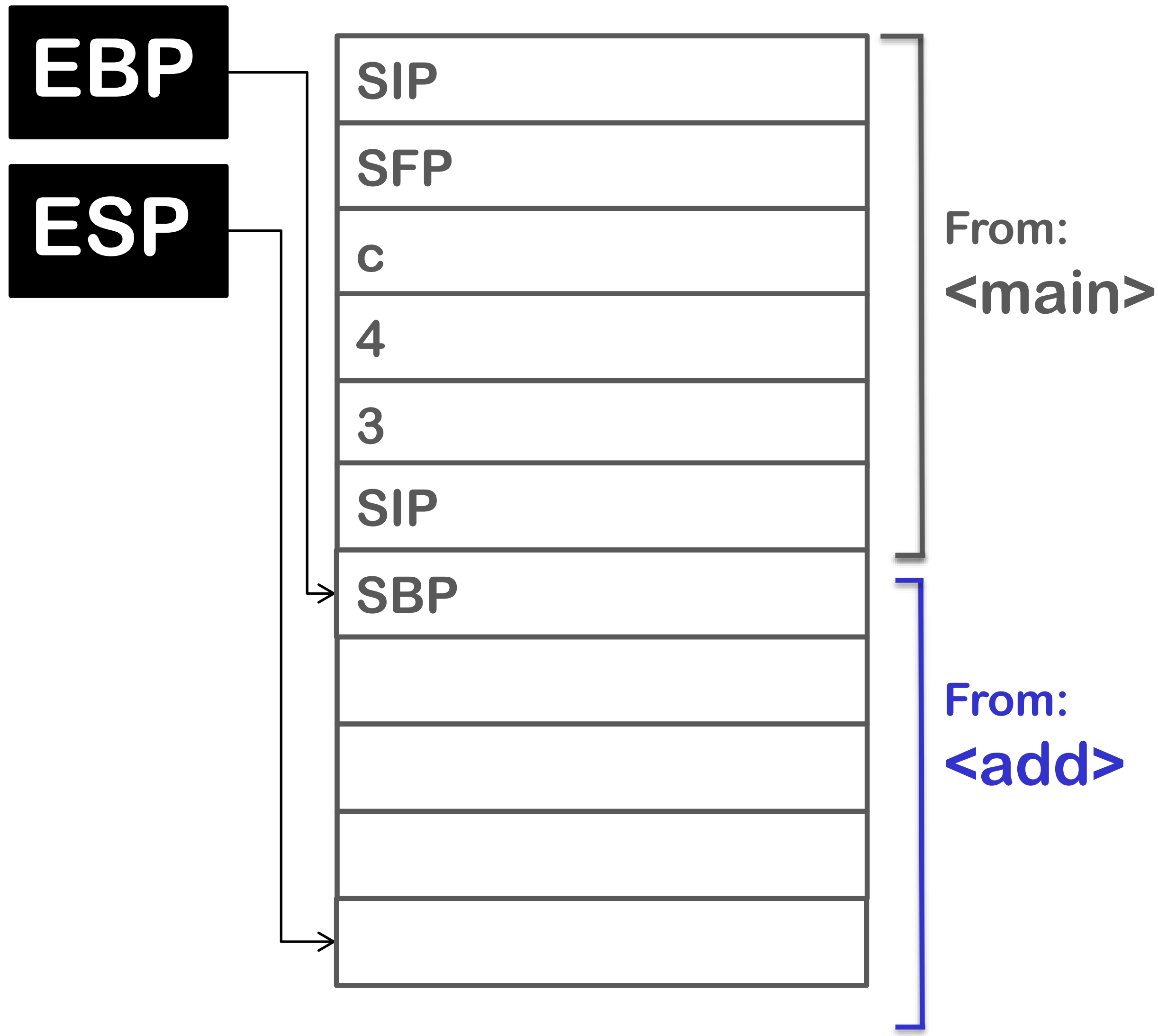
```
mov eax, DWORD PTR [ebp + 0xc]
mov edx, DWORD PTR [ebp + 0x8]
add eax, edx
mov DWORD PTR [ebp – 0x04], eax
mov eax, DWORD PTR [ebp – 0x04]
```



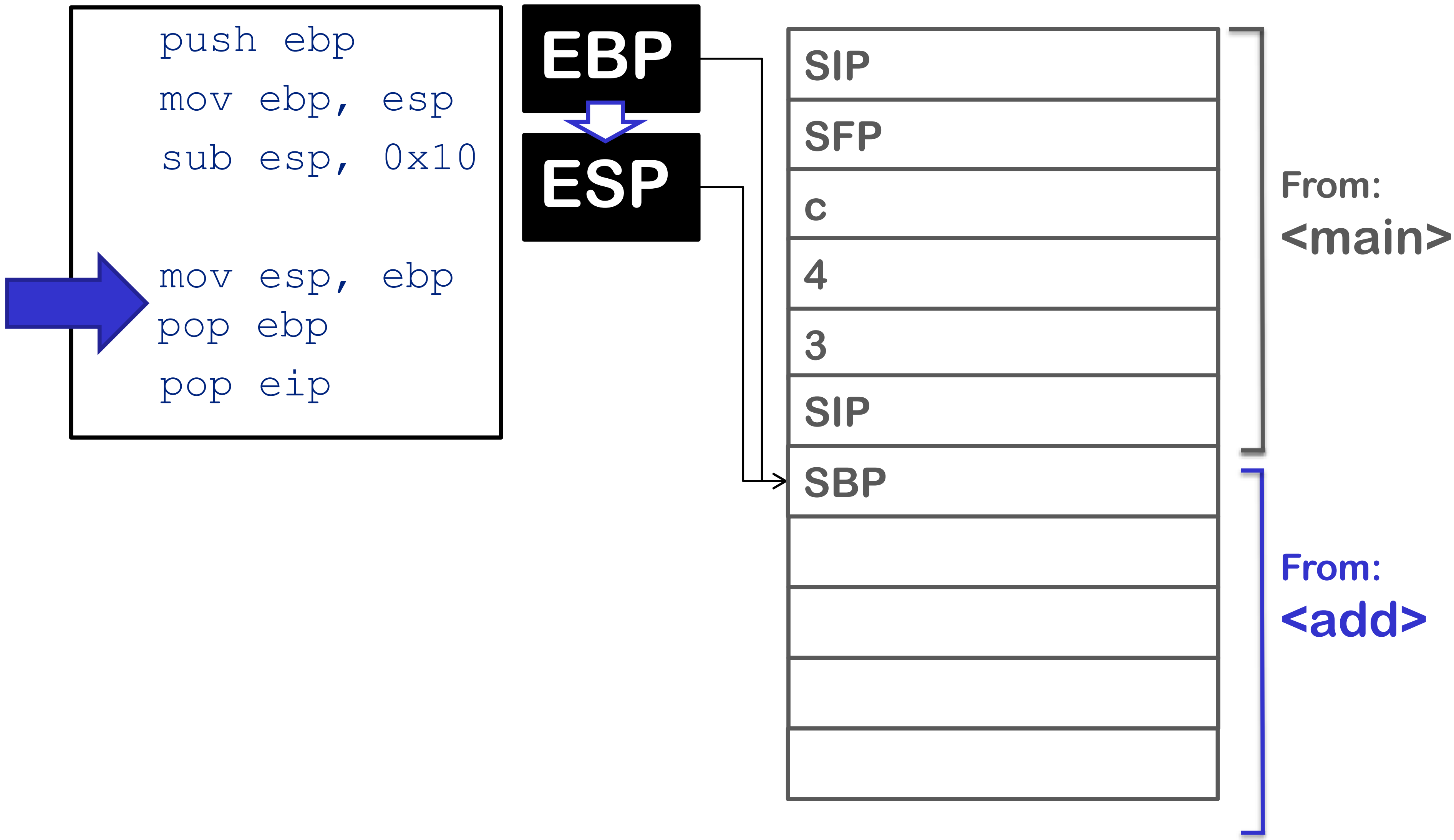
Function Epilog

x32 Call Convention - Function Epilog

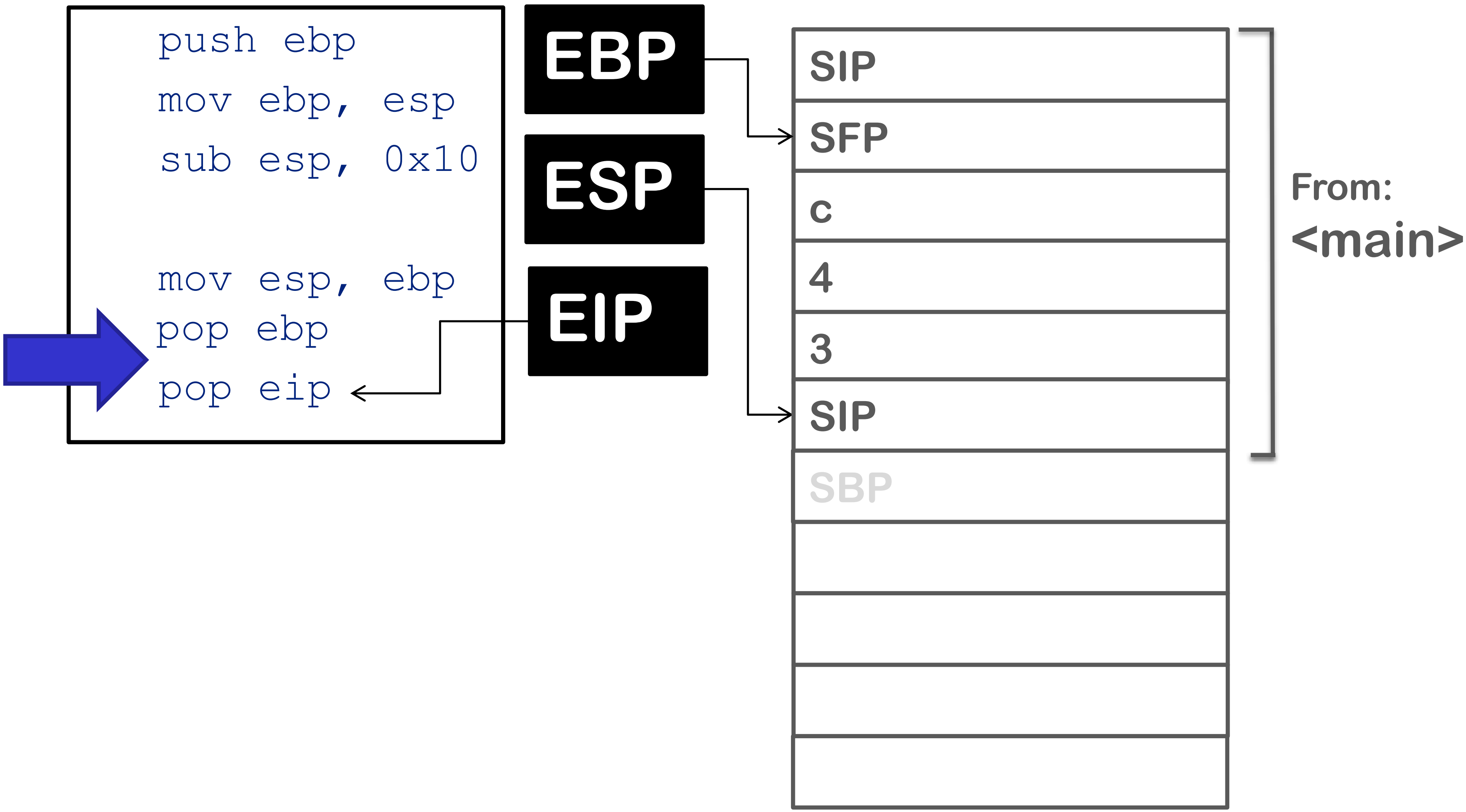
```
push ebp
mov ebp, esp
sub esp, 0x10

mov esp, ebp
pop ebp
pop eip
```



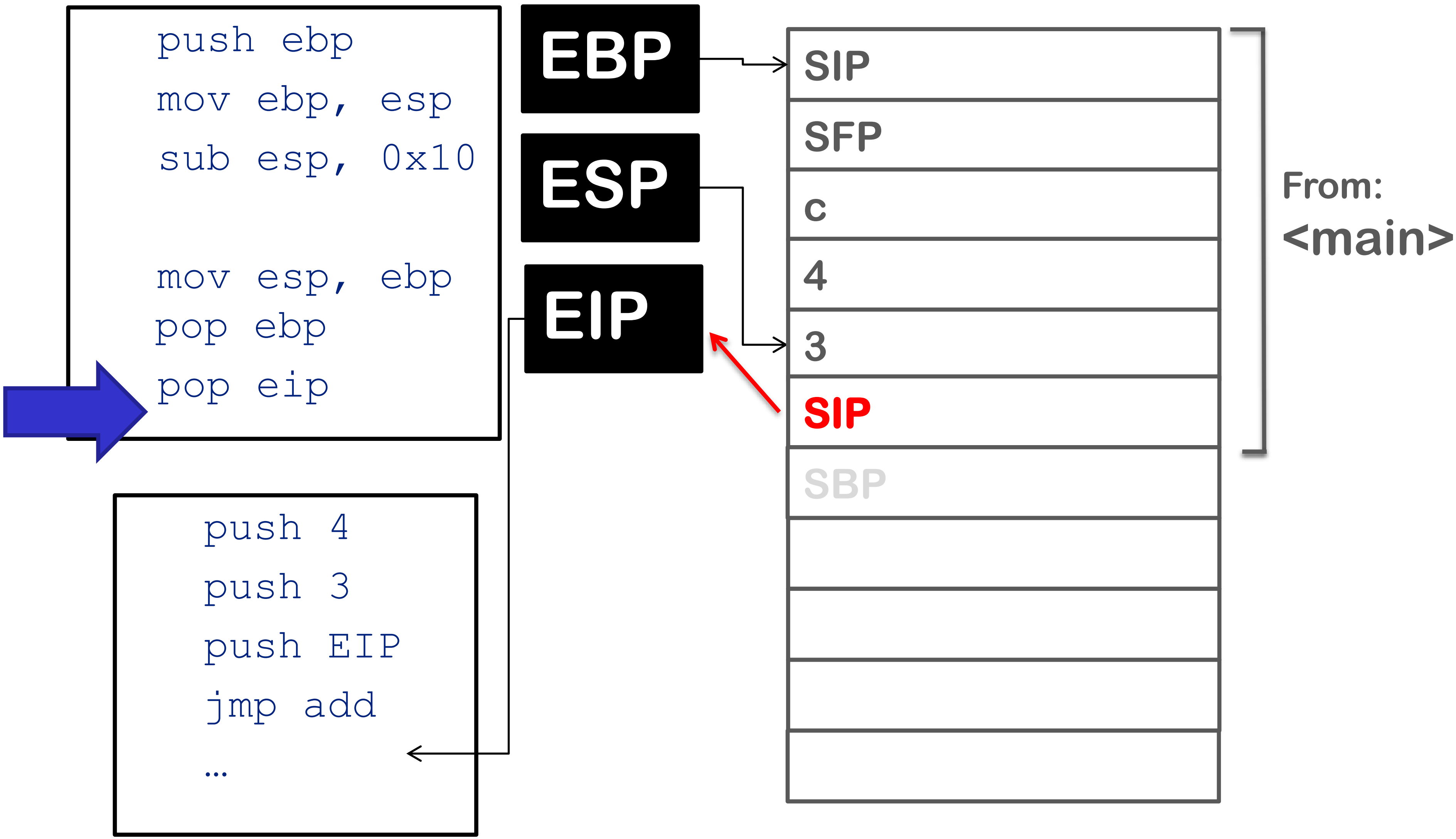
x32 Call Convention - Function Epilog



x32 Call Convention - Function Epilog



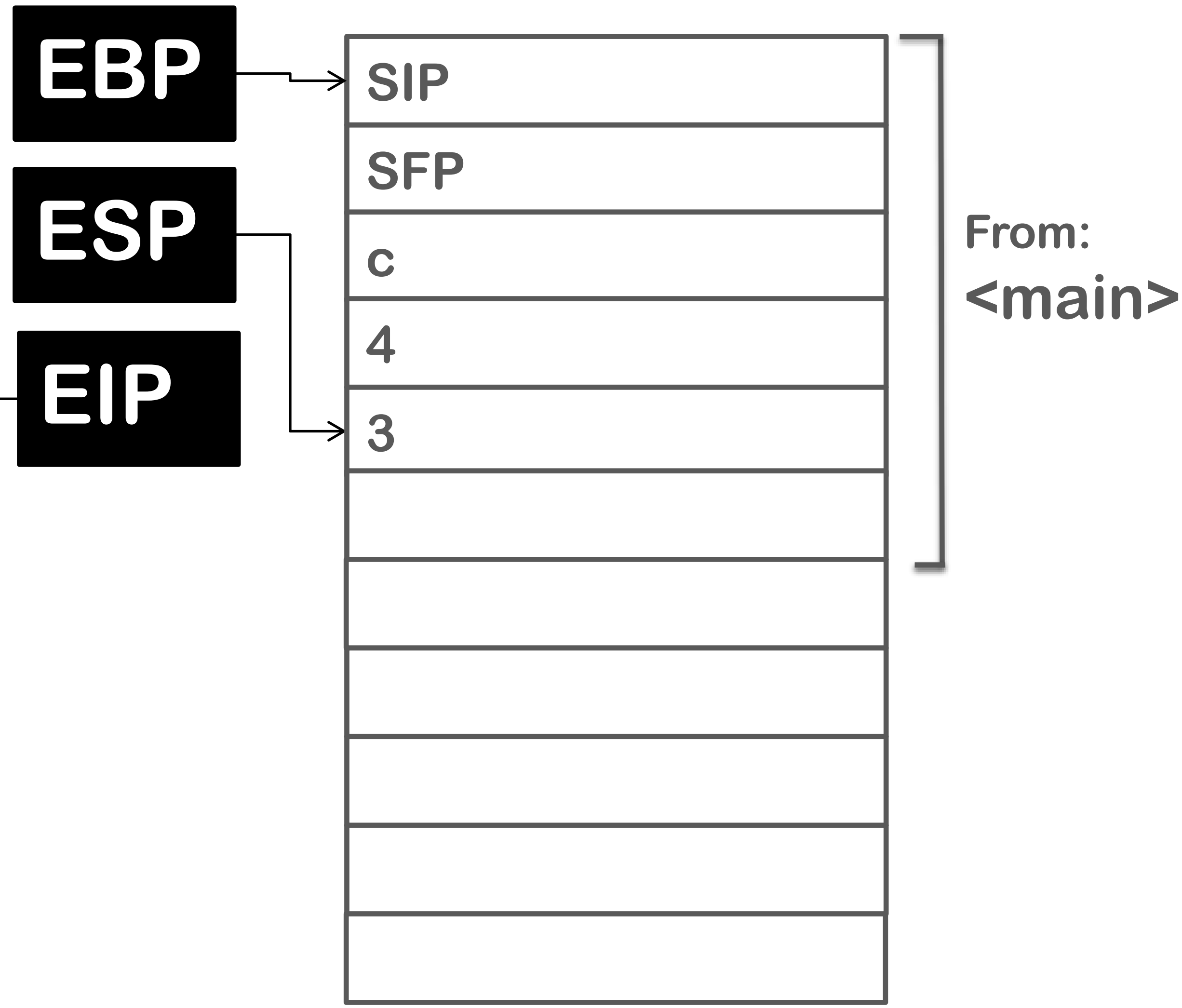
x32 Call Convention - Function Epilog



x32 Call Convention - Function Epilog

```
push ebp
mov ebp, esp
sub esp, 0x10

mov esp, ebp
pop ebp
pop eip
```

```
push 4
push 3
push EIP
jmp add
…
```



x32 Call Convention - Function Calling

```
call <addr> =
  push EIP
  jmp <addr>

leave =
  mov esp, ebp
  pop ebp

ret =
  pop eip
```

x32 Call Convention - Function Calling

Why “leave”?

- Opposite of “enter”

“enter”:

```
push ebp
mov ebp, esp
sub esp, imm
```

Why no “enter” used?

- enter:
  - 8 cycle latency
  - 10-20 micro ops
- call <addr>; mov ebp, esp; sub esp, imm:
  - 3 cycles latency
  - 4-6 micro ops

x32 Call Convention - Function Calling

Recap:

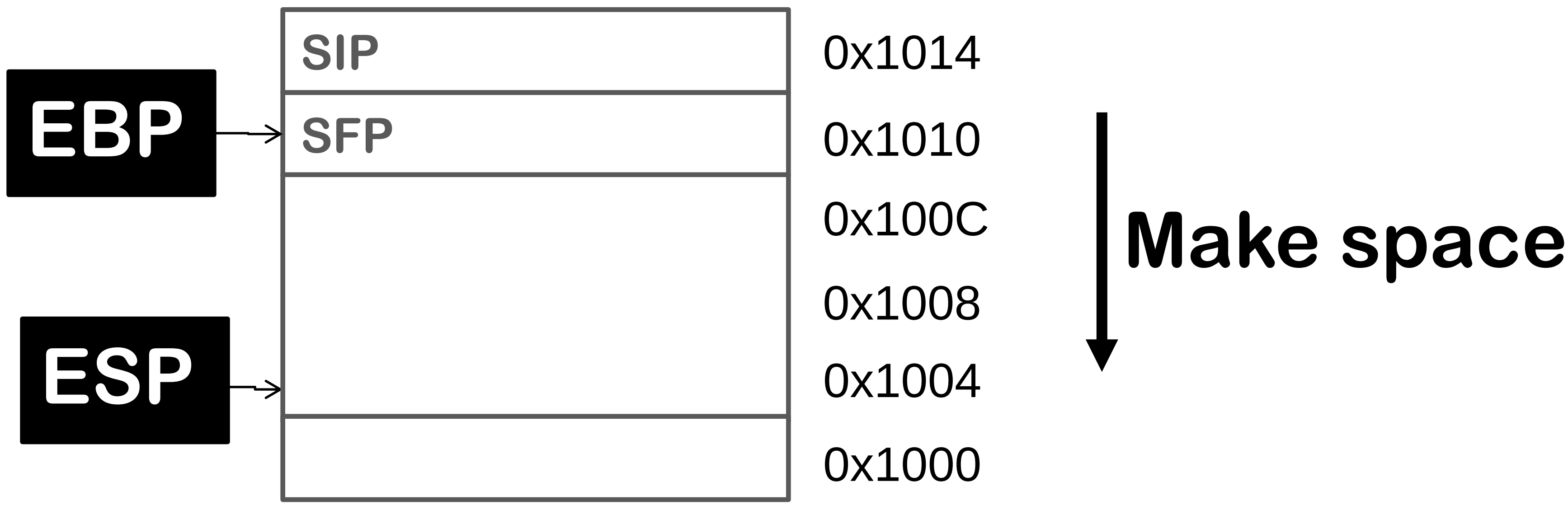
- When a function is called:
  - EIP is pushed on the stack (=SIP)
  - (“call” is doing implicit “push EIP”)

- At the end of the function:
  - SIP is recovered into EIP
  - (“ret” is doing implicit “pop EIP”)

Accessing the Stack

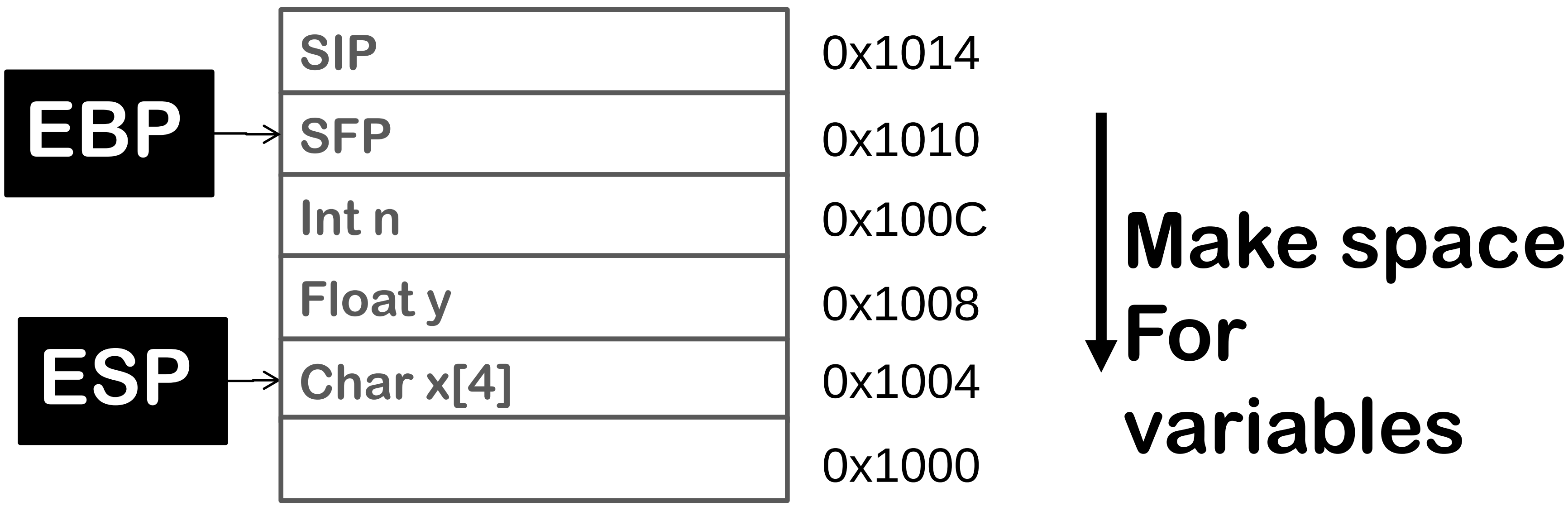
Accessing the stack

- Push/Pops are rarely used nowadays
- Each function makes some space in its stack frame for local variables

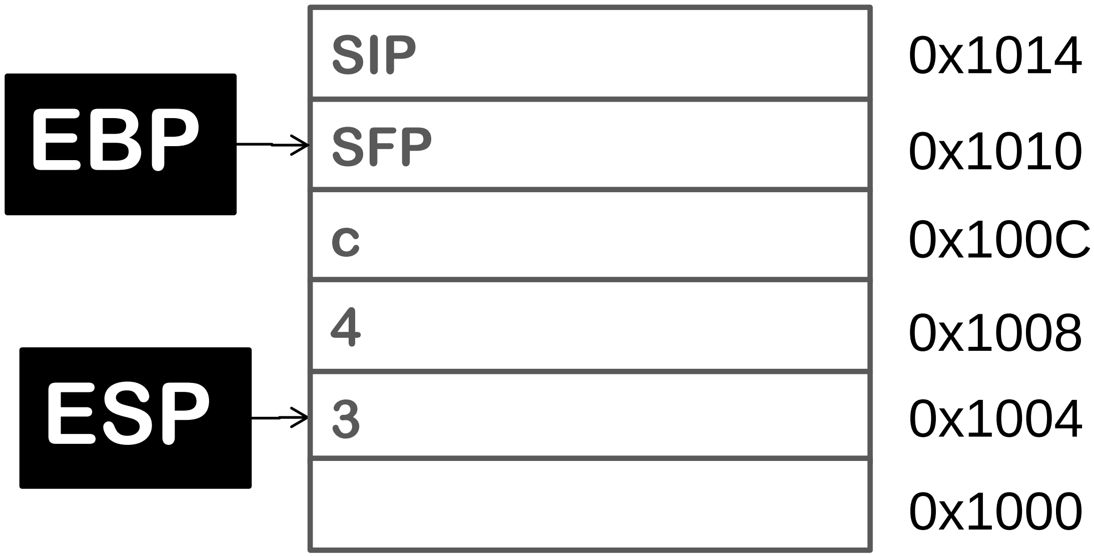


Accessing the stack

- Push/Pops are rarely used nowadays
- Each function makes some space in its stack frame for local variables



Accessing the stack: triple view




```
A) push 0x1
B) mov [ebp-0x10], 0x1
C) mov eax, 0x1000
   mov [eax], 0x1
```

Function Calls in x64

x32 Call Convention - Function Call in x64

Differences between x32 and x64 function calls:

Arguments are in registers (not on stack)

```
RDI, RSI, RDX, R8, R9
```

x32 Call Convention - Function Call in x64

Differences between x32 and x64 function calls

Different ASM commands doing the same thing

```
callq (call)
leaveq (leave)
retq (ret)
```

x32 Call Convention - Function Call in x64

Some random x64 architecture facts:

The stack should stay **8-byte aligned** at all times

An n-byte item should start at an **address divisible by n**

- E.g. 64 bit number: 8 bytes, can be at 0x00, 0x08, 0x10, 0x18, …

%rsp points to the lowest **occupied** stack location

- not the next one to use!

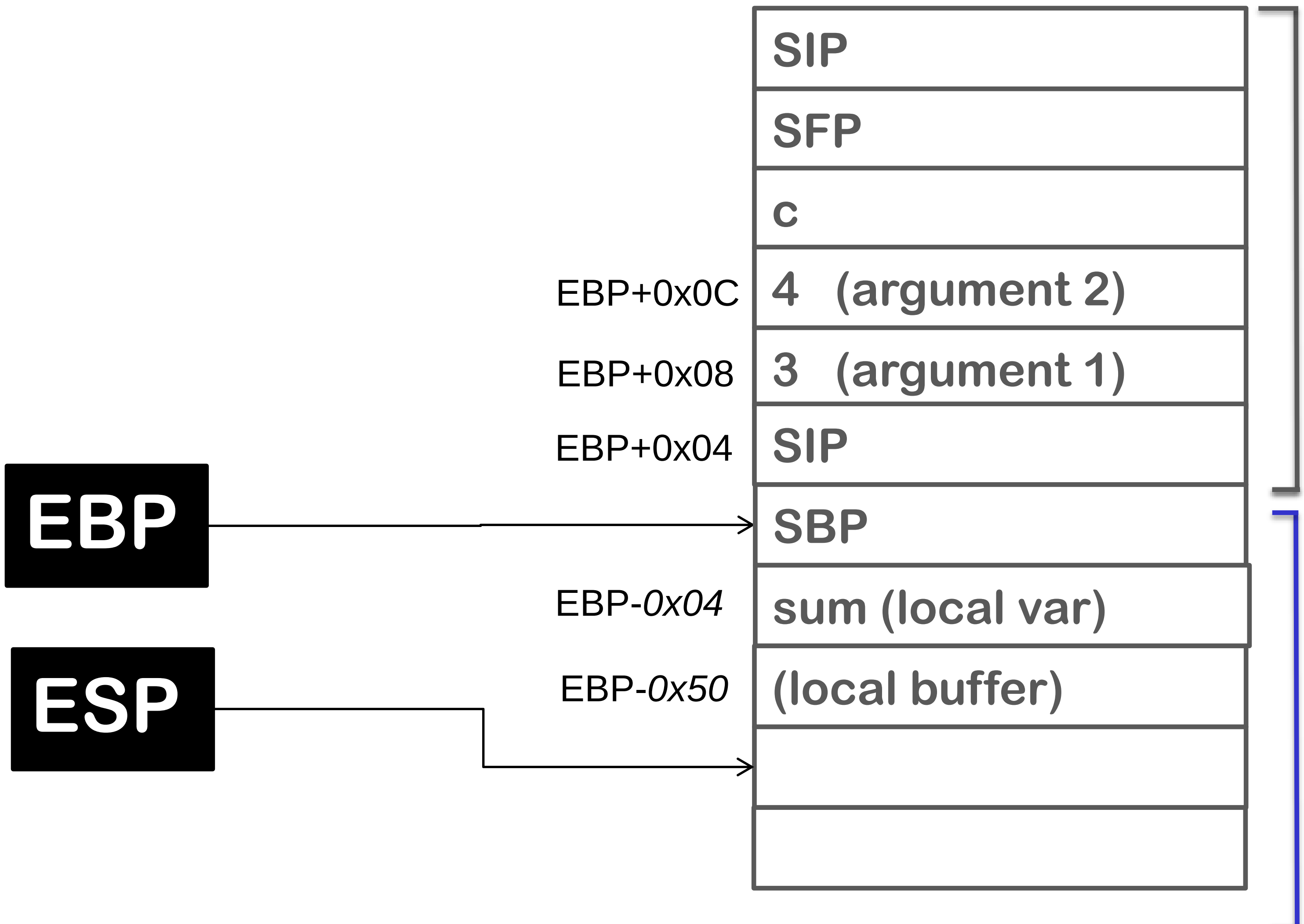
Function Call Convention Cheat Sheet

x32	Parameter	Syscall nr in
x32 userspace	`stack`	
x32 syscalls	`ebx, ecx, edx, esi, edi, ebp`	eax

x64	Parameter	Syscall nr in
x64 userspace	`rdi, rsi, rdx, rcx, r8, r9`	
x64 syscall	`rdi, rsi, rdx, r10, r8, r9`	rax

http://stackoverflow.com/questions/2535989/what-are-the-calling-conventions-for-unix-linux-system-calls-on-x86-64

EBP Cheat Sheet



EBP Cheat Sheet

```
:    :
|  2 | [ebp + 16] (3rd function argument)
|  5 | [ebp + 12] (2nd argument)
| 10 | [ebp + 8]  (1st argument)
| RA | [ebp + 4]  (return address)
| FP | [ebp]      (old ebp value)
|    | [ebp - 4]  (1st local variable)
:    :
:    :
|    | [ebp - X]  (esp - the current stack pointer. The use of push / pop is valid now)
```

Outro

Further questions

Can you implement push/pop in ASM? (without actually using push/pop)

Answers

Pseudocode:

```
# EAX is the new ESP
push <data>:
     sub eax, 4
     mov (%eax), <data>

pop <register>:
     mov <register>, (%eax)
     add eax, 4
```