CS354: Machine Organization and Programming

Lecture 16
Friday the October 09th 2015

Section 2
Instructor: Leo Arulraj

Class Announcements

1. Midterm 1 grades should be available by Monday next week.
2. Programming Assignment 1 will also be likely graded before early next week.

Lecture Overview

1. Demo of function calls using gdb along with slides that show how the stack changes during a simple function call.

2. Calling Conventions

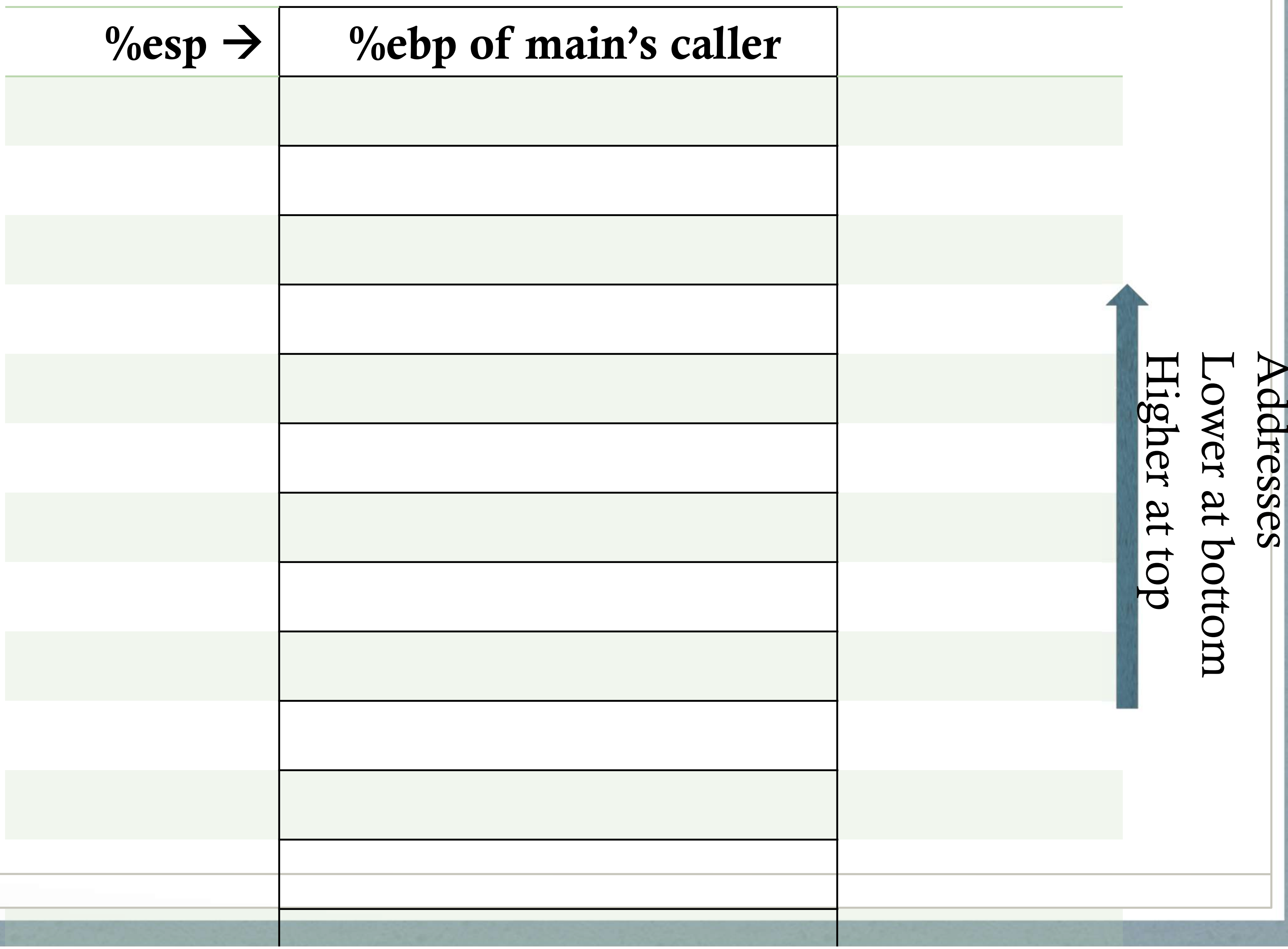
3. Overview of Function calls

Demo

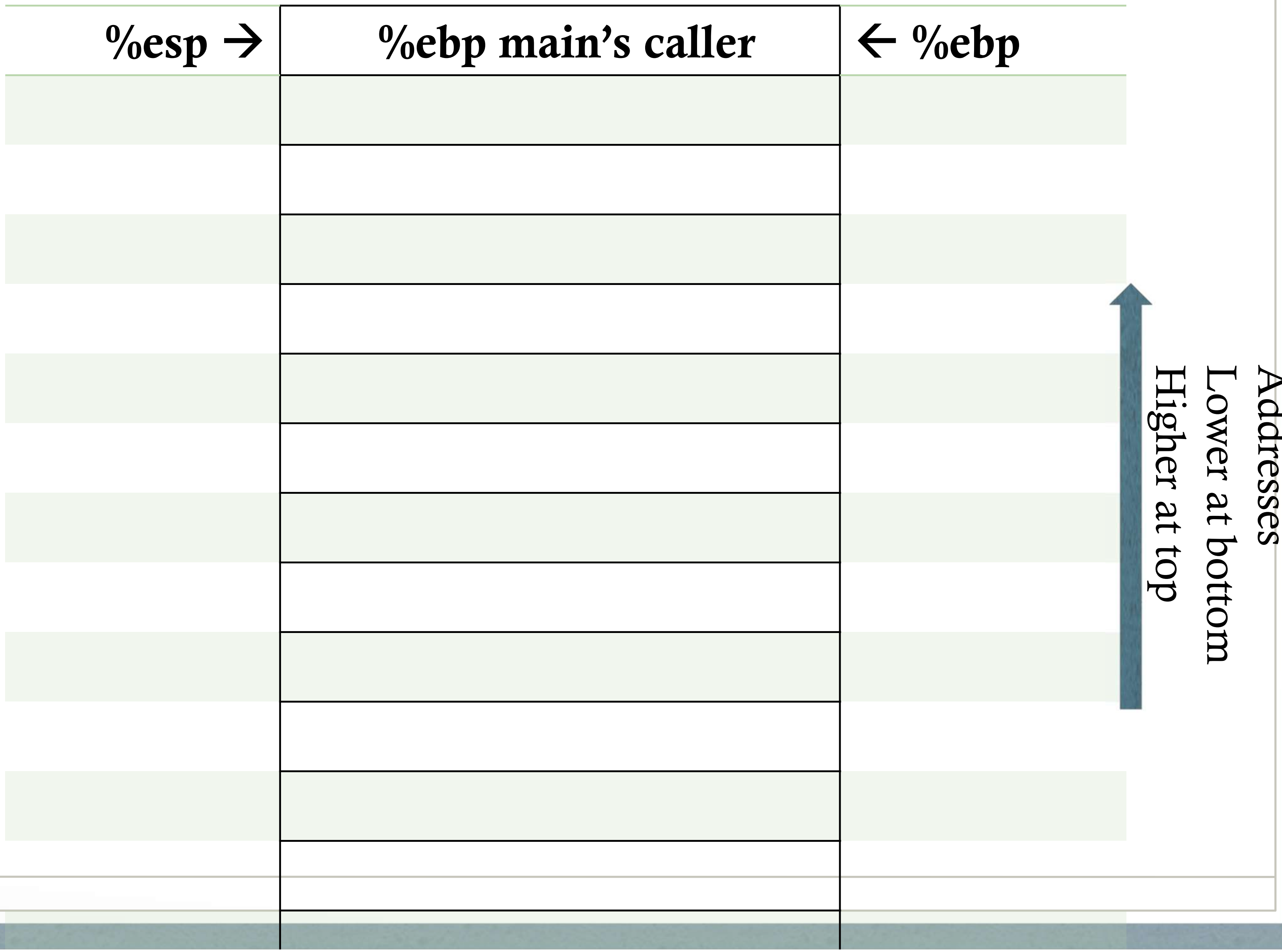
1. The following slides step through the assembly instructions for the program simplefunctions1.c from Lecture 16 and show how the stack changes.

2. Keep the files simplefunctions1.c and simplefunctions1.objdump open while going over the following slides that show the stack layout.

Prologue: After executing Instruction : 0x80483be: push %ebp



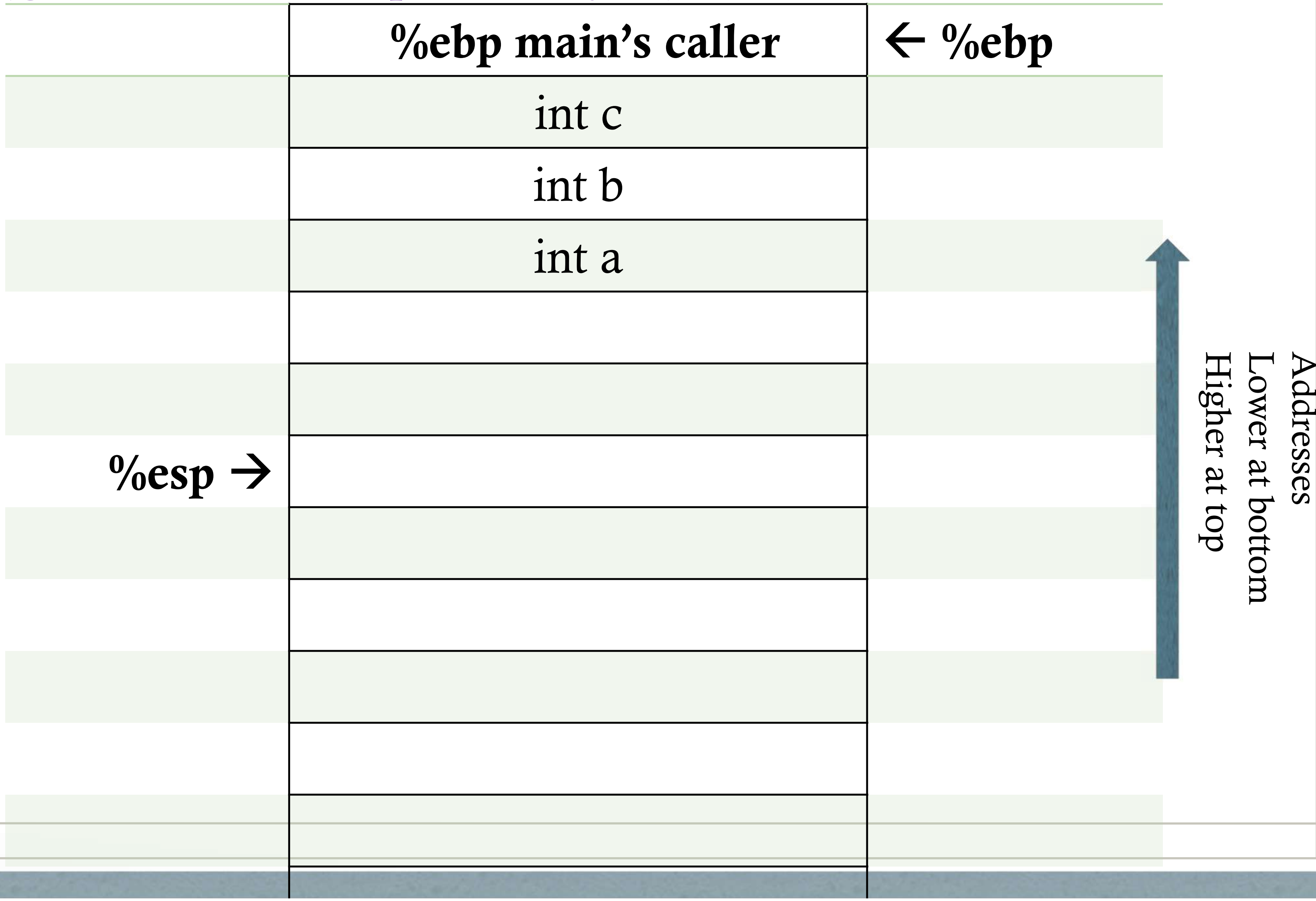
Prologue: After executing Instruction : 0x80483bf: mov %esp,%ebp



Prologue: After executing Instruction : 0x80483c1: sub $0x18,%esp

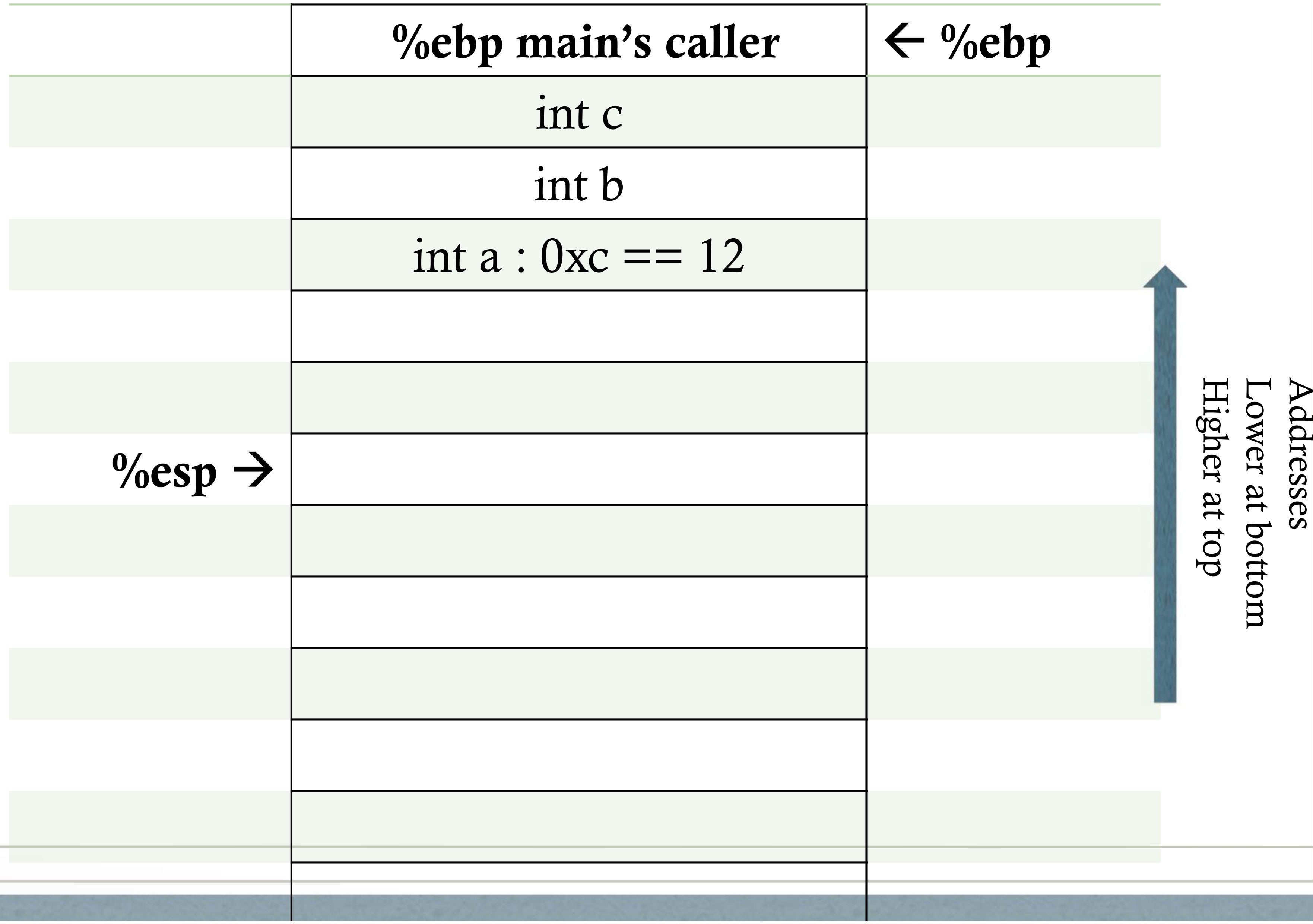
Allocating Space for local variables : a, b, c and parameters to func1

(gcc allocates in multiples of 16 bytes)



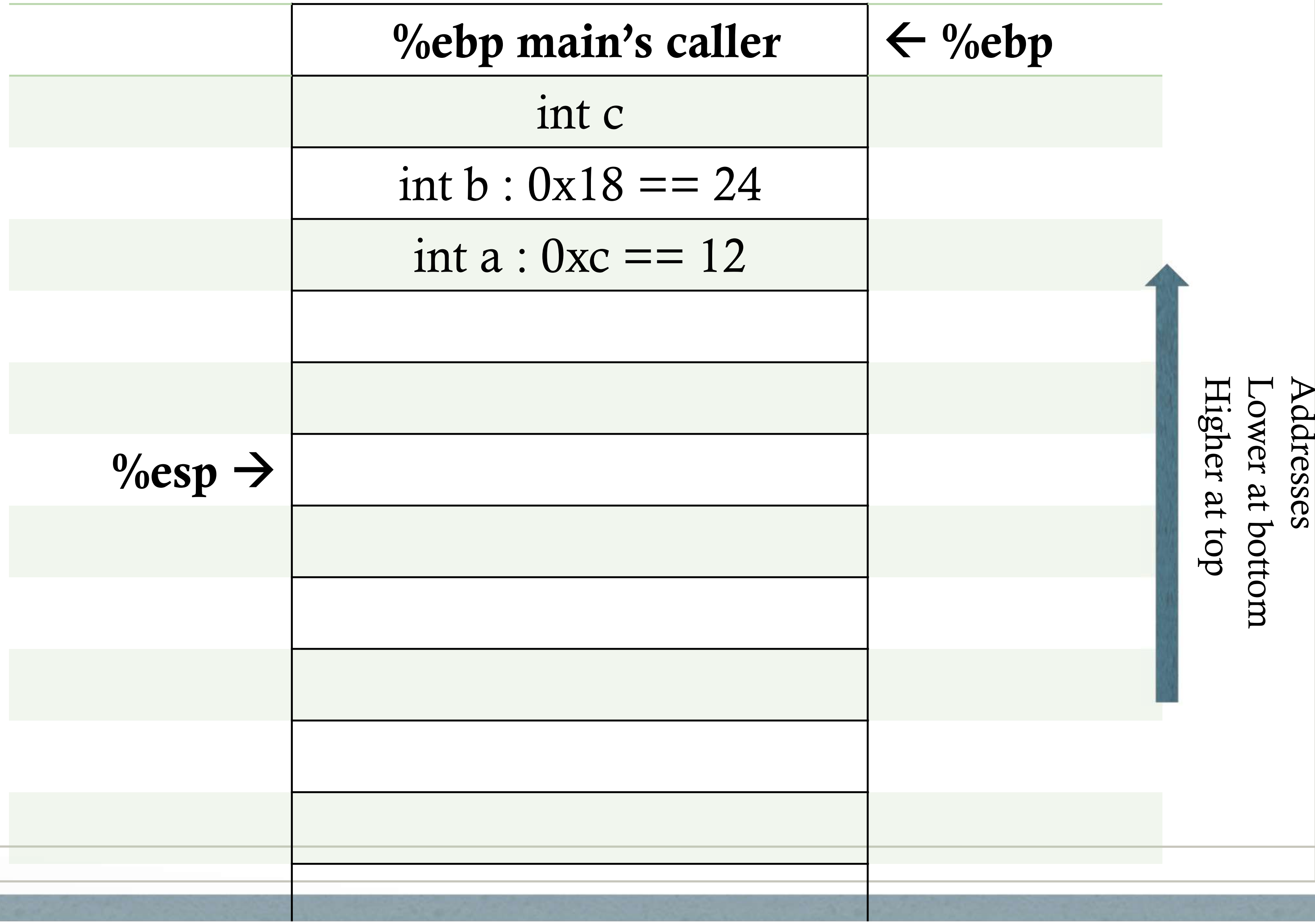
After executing Instruction : 0x80483c4: movl $0xc,-0xc(%ebp)

Initializing local variable a;



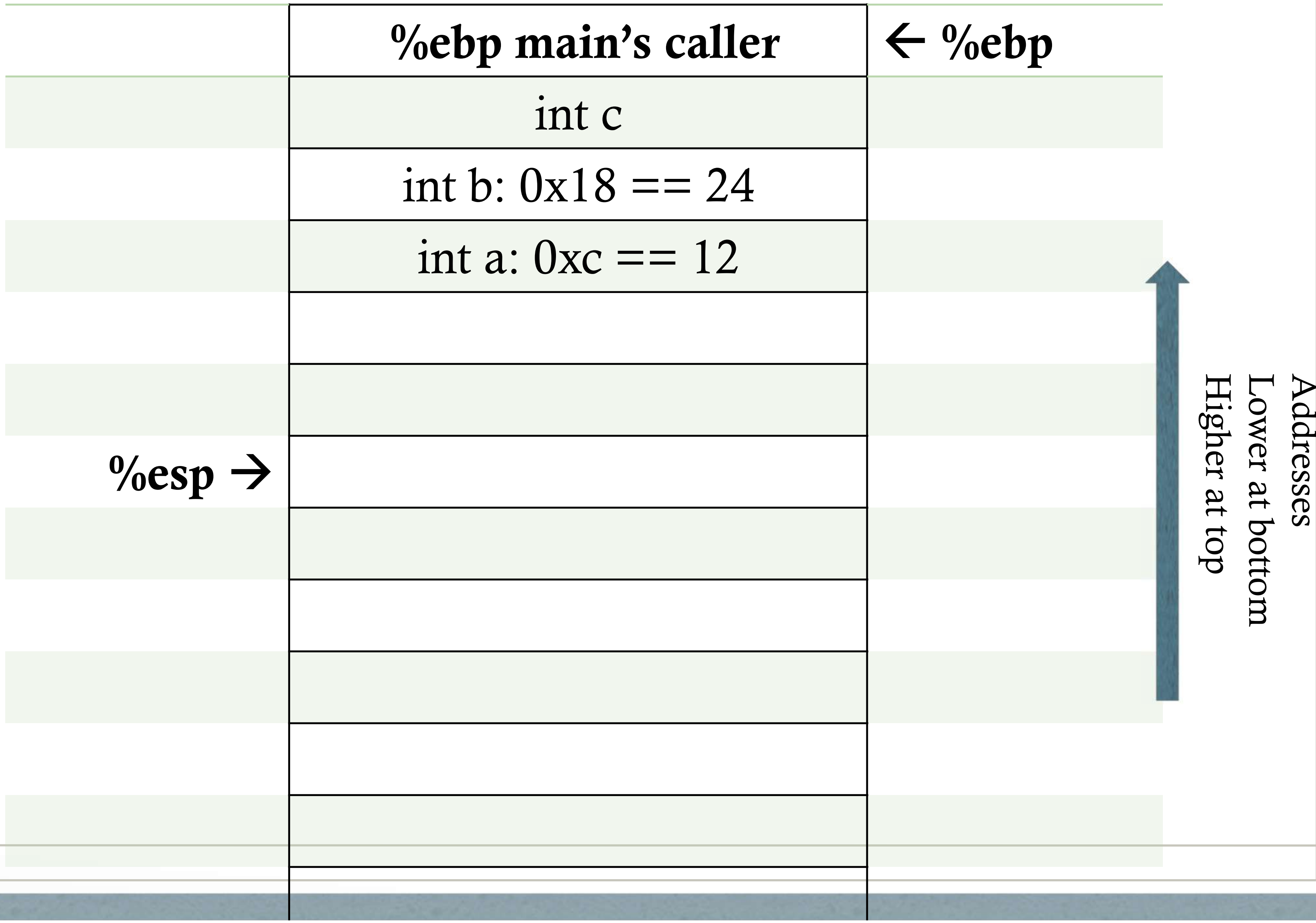
After executing Instruction : 0x80483cb: movl $0x18,-0x8(%ebp)

Initializing local variable b;



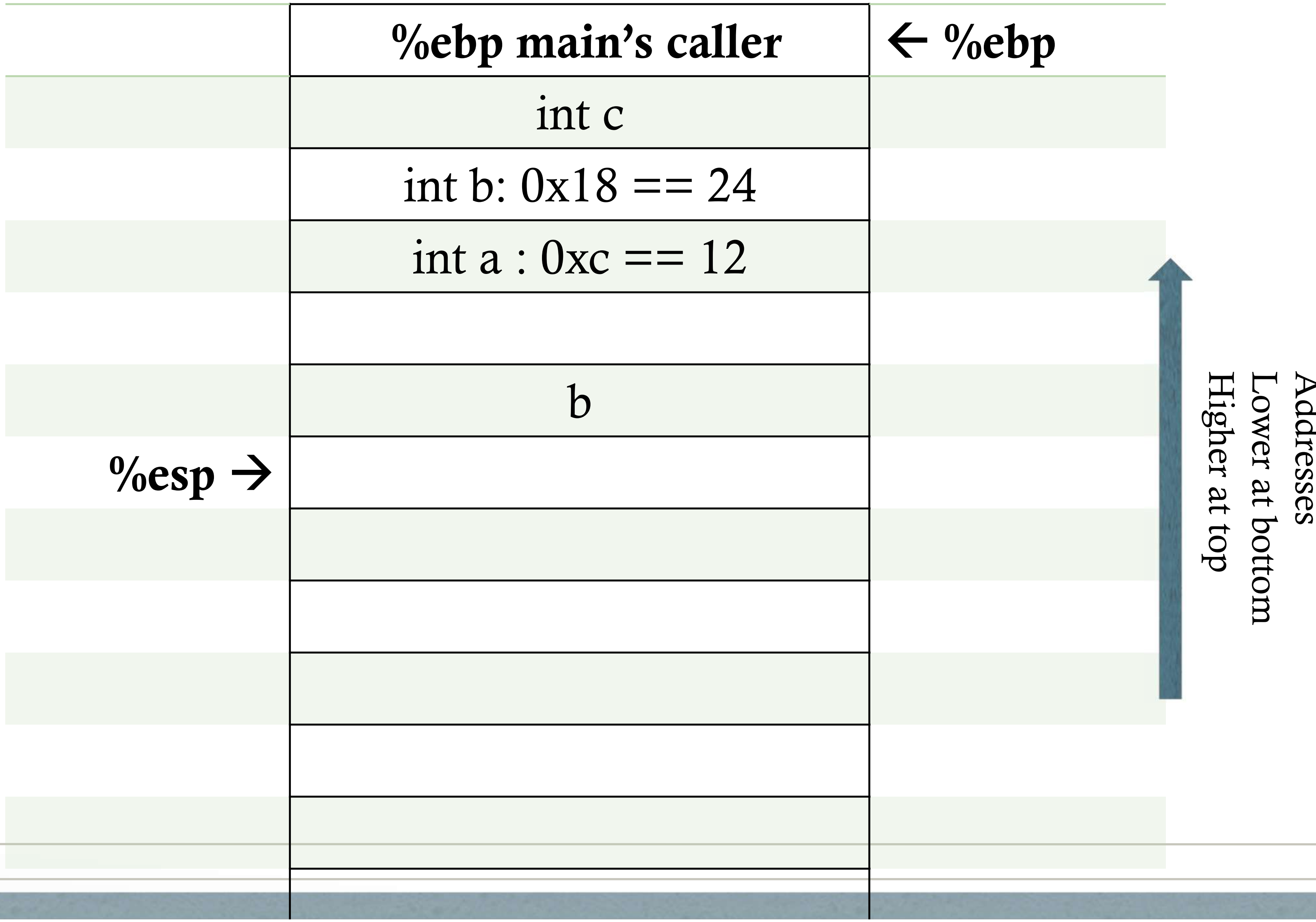
After executing Instruction : 0x80483d2: mov -0x8(%ebp),%eax

Fetch b in %eax;



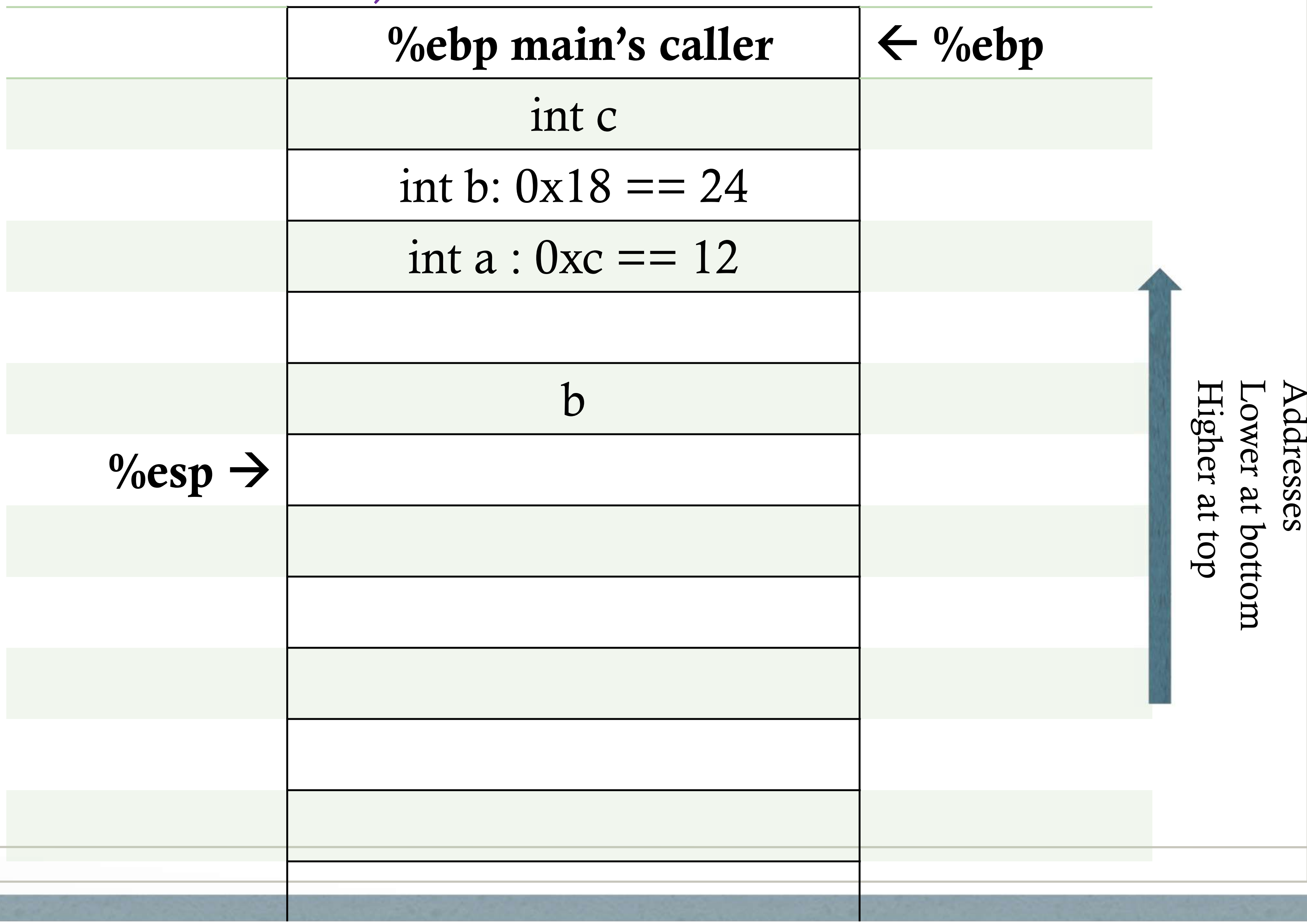
After executing Instruction : 0x80483d5: mov %eax,0x4(%esp)

Set up parameter b;



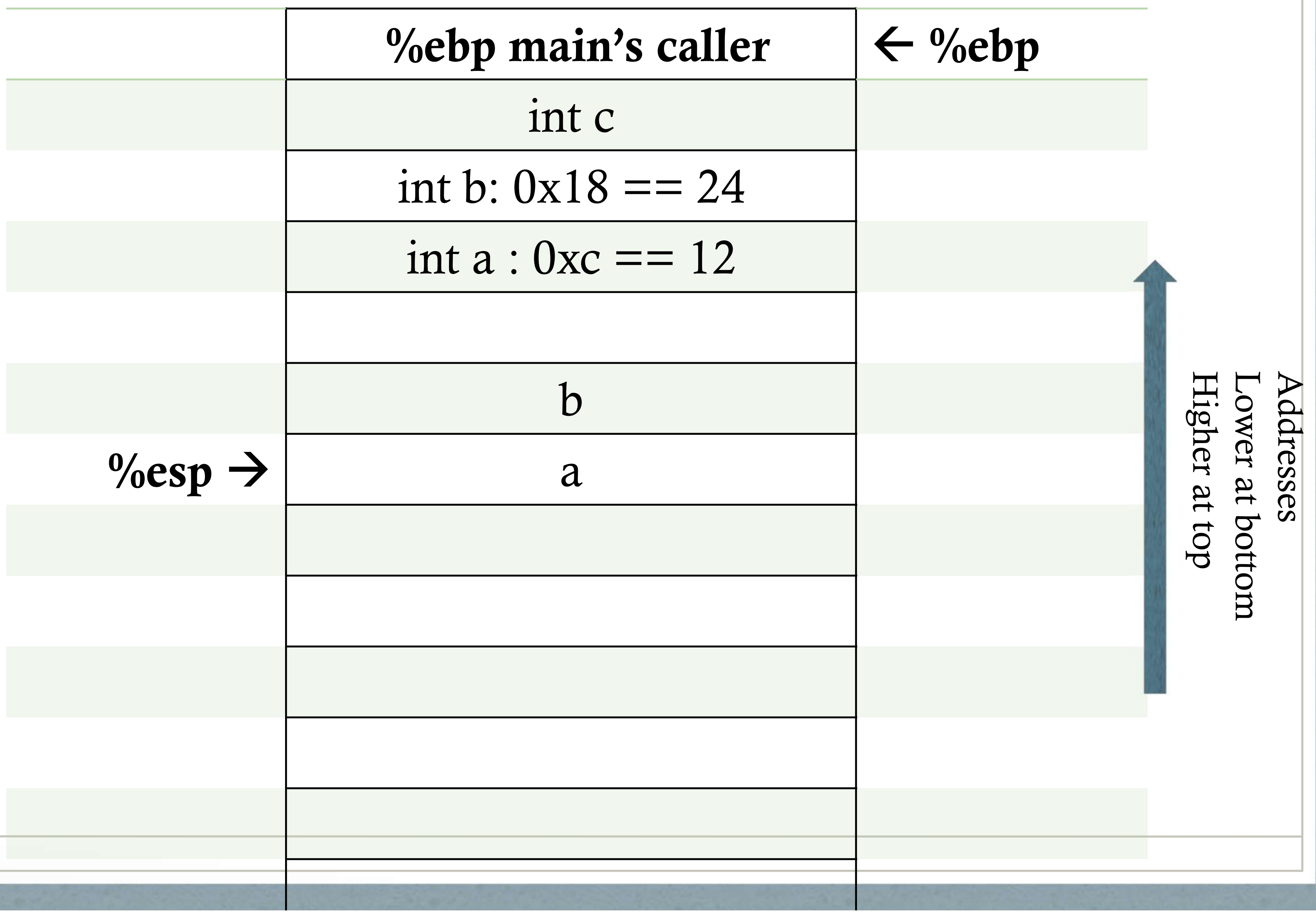
After executing Instruction : 0x80483d9: mov -0xc(%ebp),%eax

Fetch a into %eax;



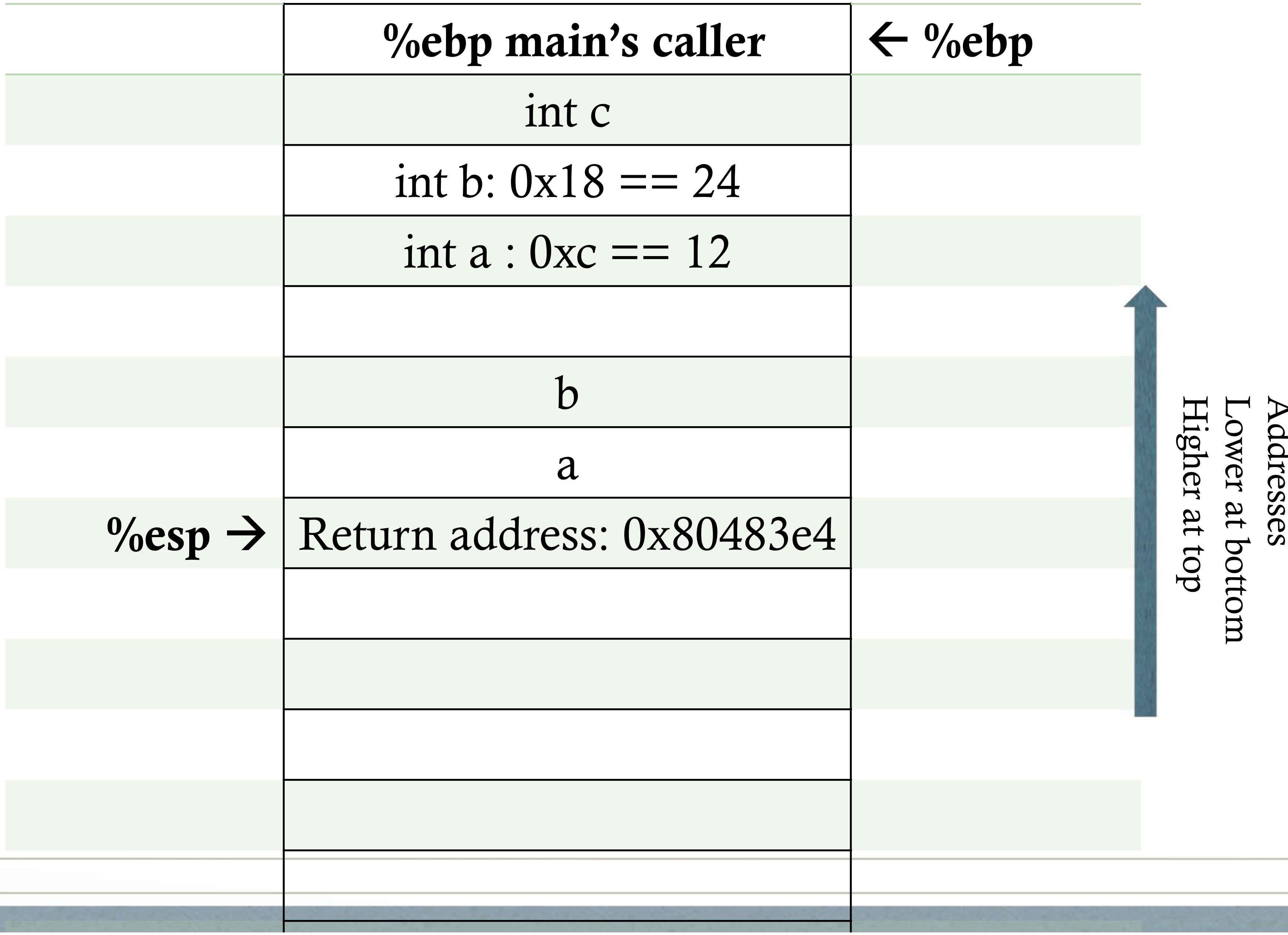
After executing Instruction : 0x80483dc: mov %eax,(%esp)

Set up parameter a;



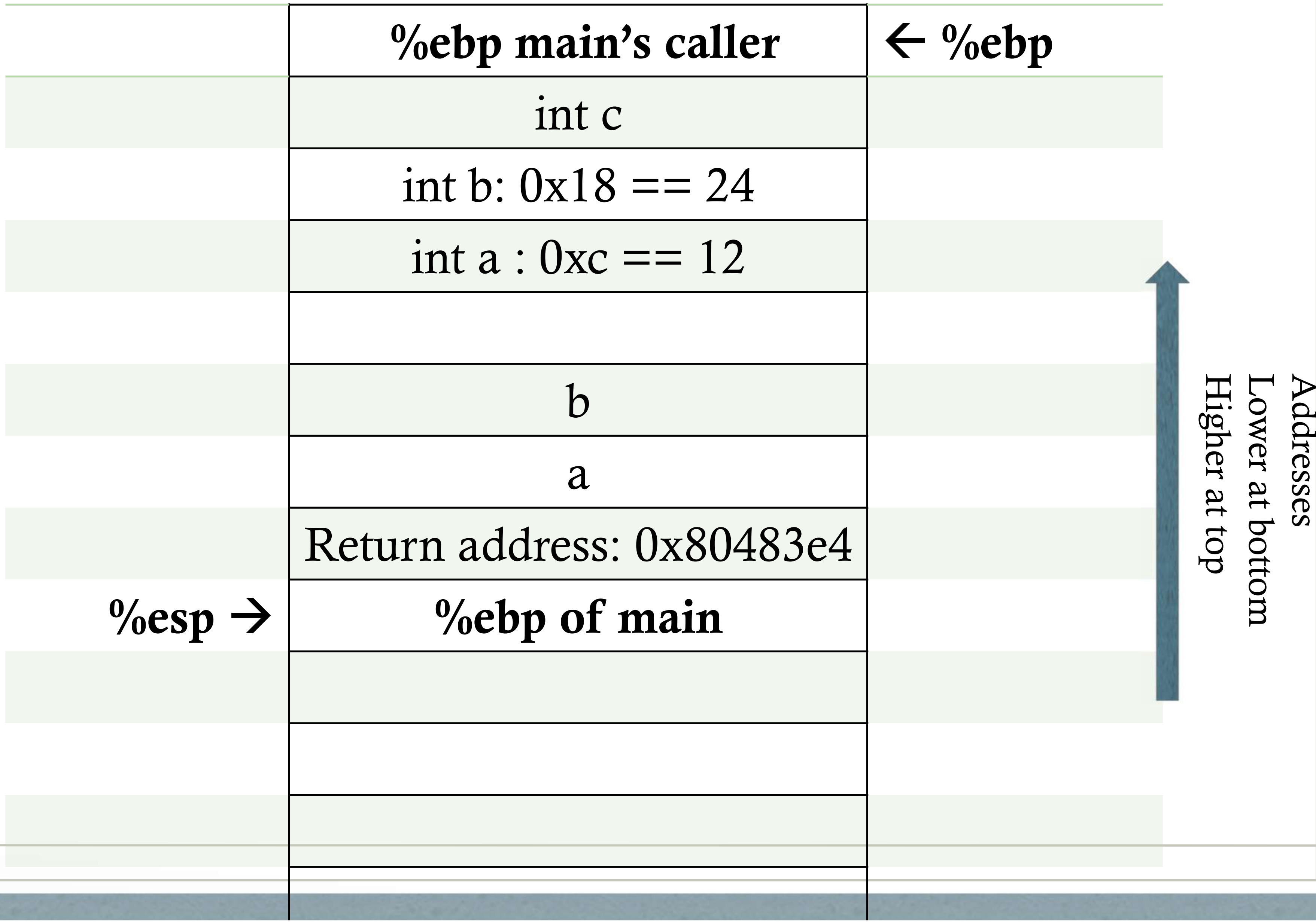
After executing Instruction : 0x80483df: call 8048394 <func1>

Call function func1: which pushes return address on stack and jumps to func1;



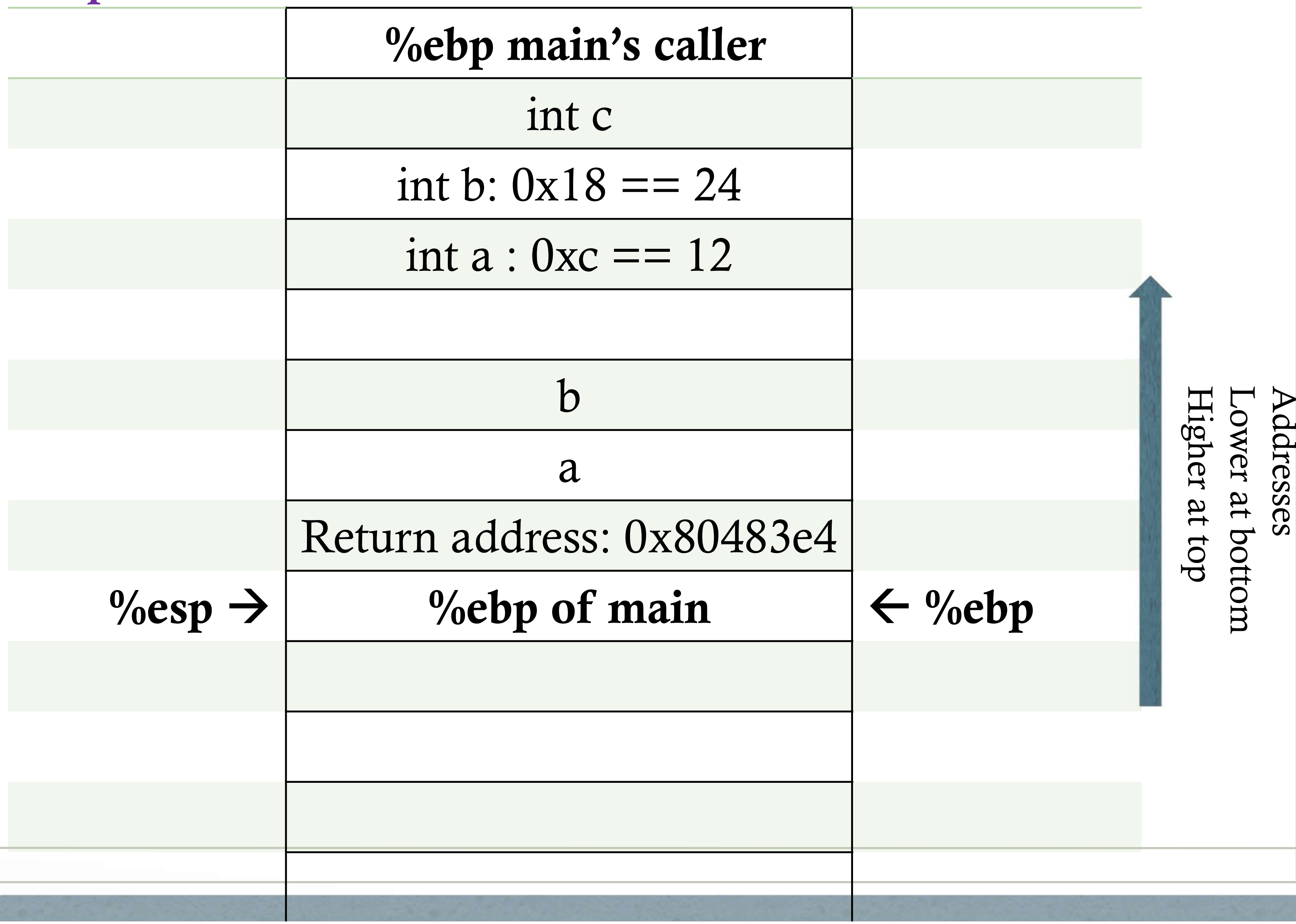
Prologue: After executing Instruction : 0x8048394: push %ebp

Push %ebp of main into stack



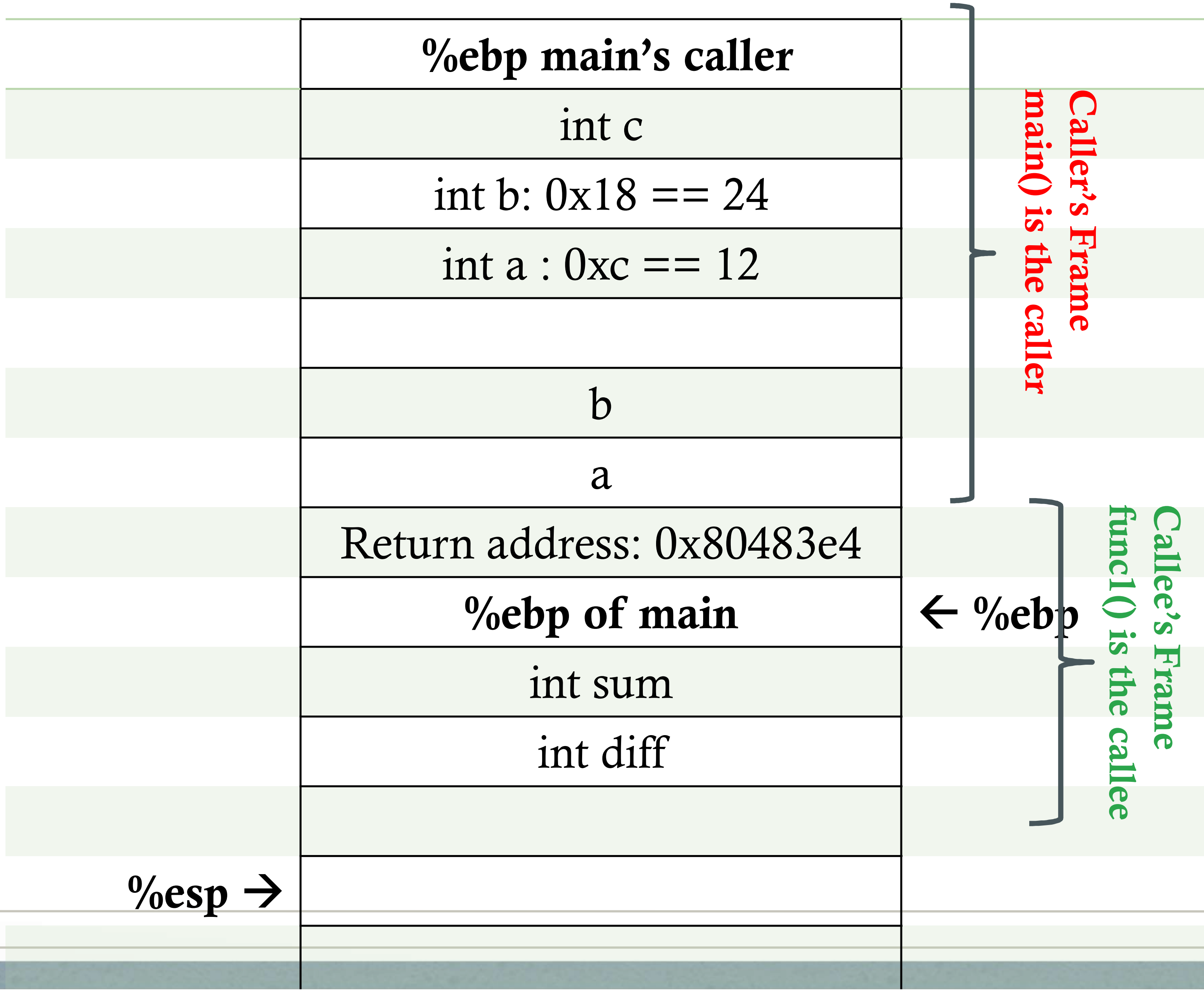
Prologue: After executing Instruction : 0x8048395: mov %esp,%ebp

Setup frame for func1



Prologue: After executing Instruction : 0x8048397: sub $0x10,%esp

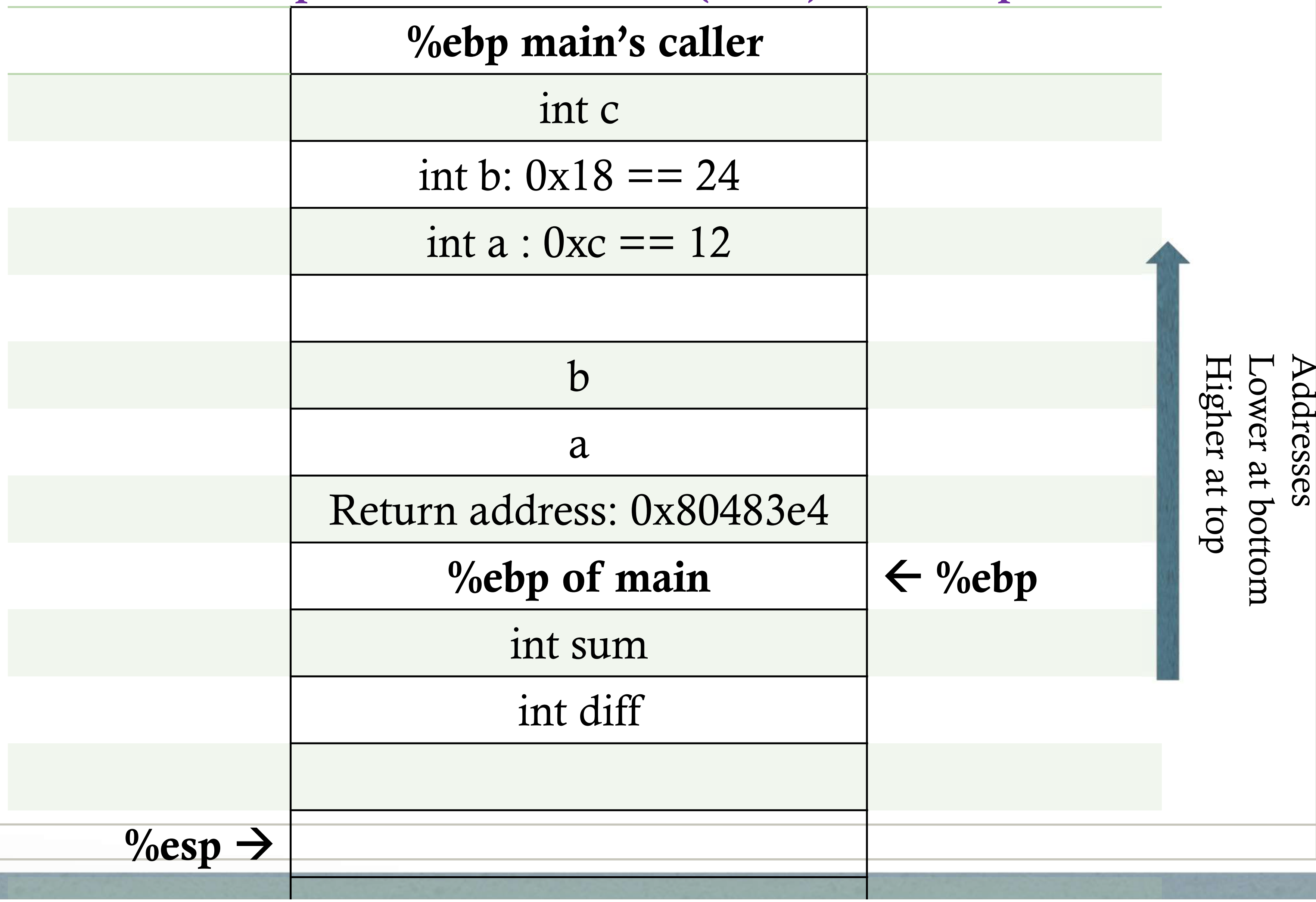
Allocate space for local variables: diff, sum (gcc allocates in multiples of 16 bytes)



After executing Instruction : 0x804839a: mov 0xc(%ebp),%eax

Fetch second parameter into %eax

General rule is : parameter i is at offset (4+4*i) from %ebp



After executing Instruction : 0x804839d: mov 0x8(%ebp),%edx

Fetch first parameter into %edx

General rule is : parameter i is at offset (4+4*i) from %ebp

	Stack	
	%ebp main's caller	
	int c	
	int b: 0x18 == 24	
	int a : 0xc == 12	
	b	
	a	
	Return address: 0x80483e4	
	%ebp of main	**← %ebp**
	int sum	
	int diff	
%esp →		

Addresses
Lower at bottom
Higher at top

After executing Instructions :

0x80483a0: mov %edx,%ecx

0x80483a0: mov %edx,%ecx

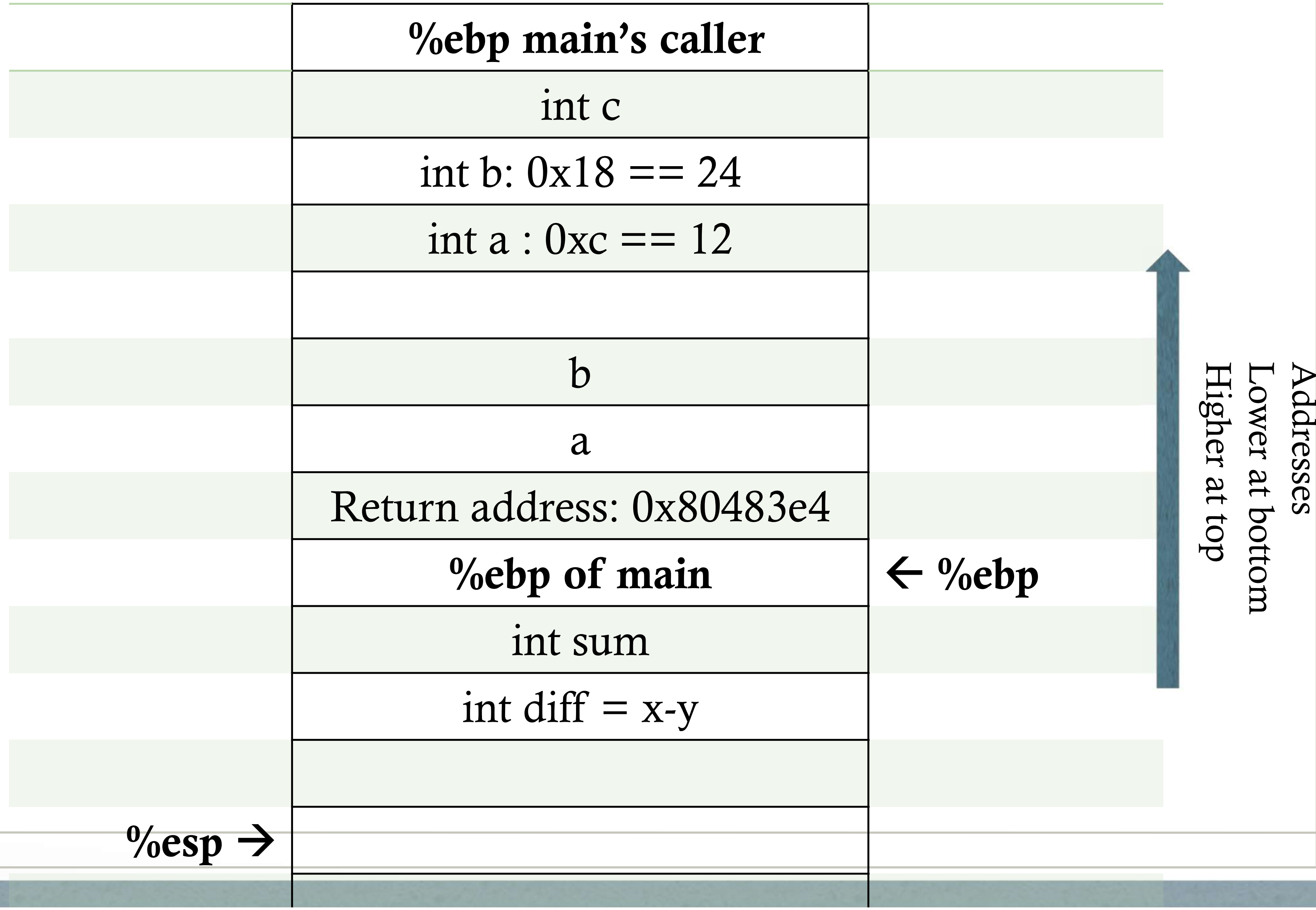
0x80483a2: sub %eax,%ecx

0x80483a4: mov %ecx,%eax

These instruction calculate x-y and store it in %eax

After executing Instruction : 0x80483a6: mov %eax,-0x8(%ebp)

Store result in diff



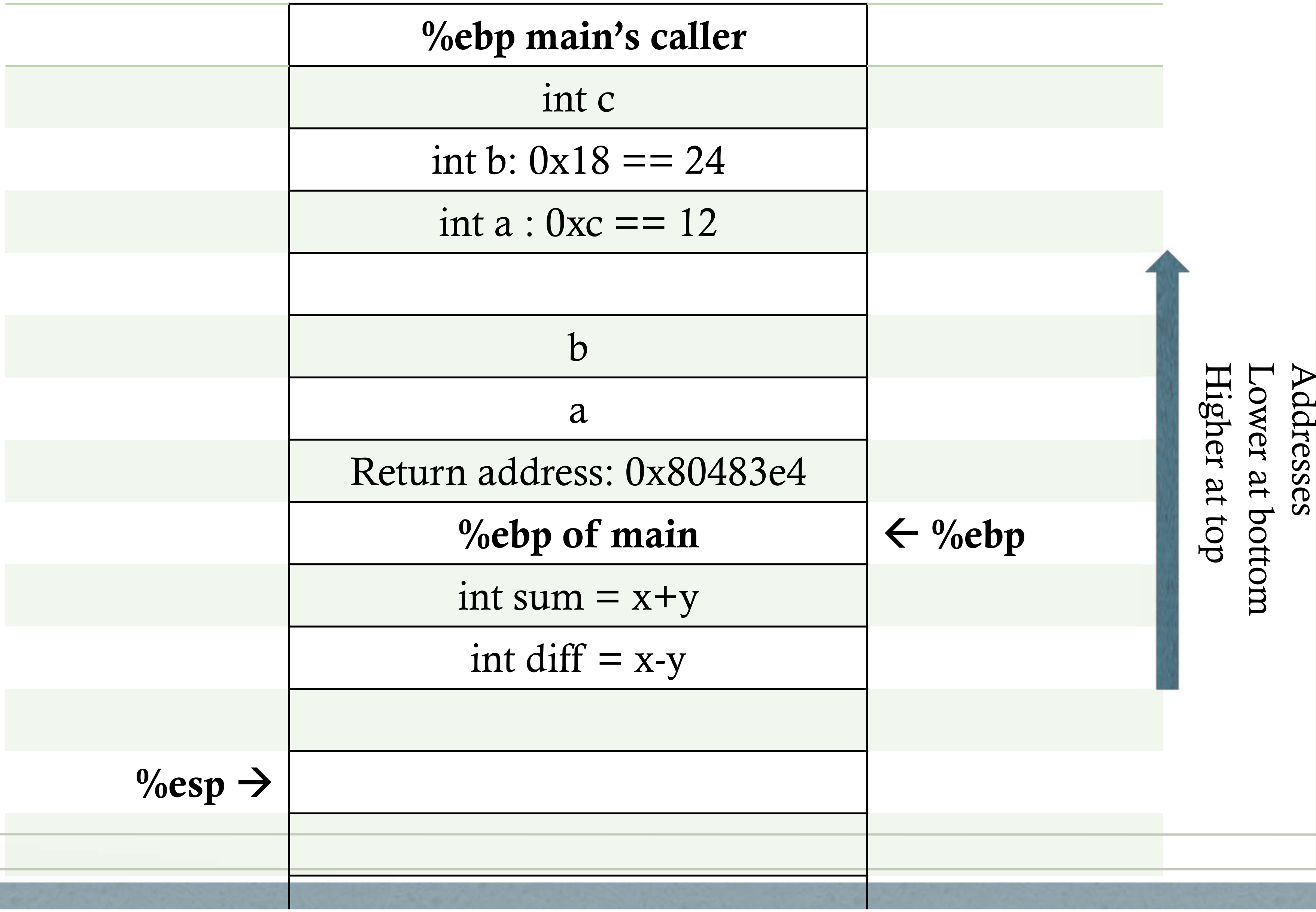
After executing Instructions :

0x80483a9: mov 0xc(%ebp),%eax
0x80483ac: mov 0x8(%ebp),%edx
0x80483af: lea (%edx,%eax,1),%eax

These instruction fetch parameters x, y into temporary registers, calculate x+y into register %eax

After executing Instruction : 0x80483b2: mov %eax,-0x4(%ebp)

Store result in sum



After executing Instructions :

0x80483b5: mov -0x4(%ebp),%eax
0x80483b8: imul -0x8(%ebp),%eax

These instructions fetch sum into %eax, and then calculate product of sum and diff into register %eax

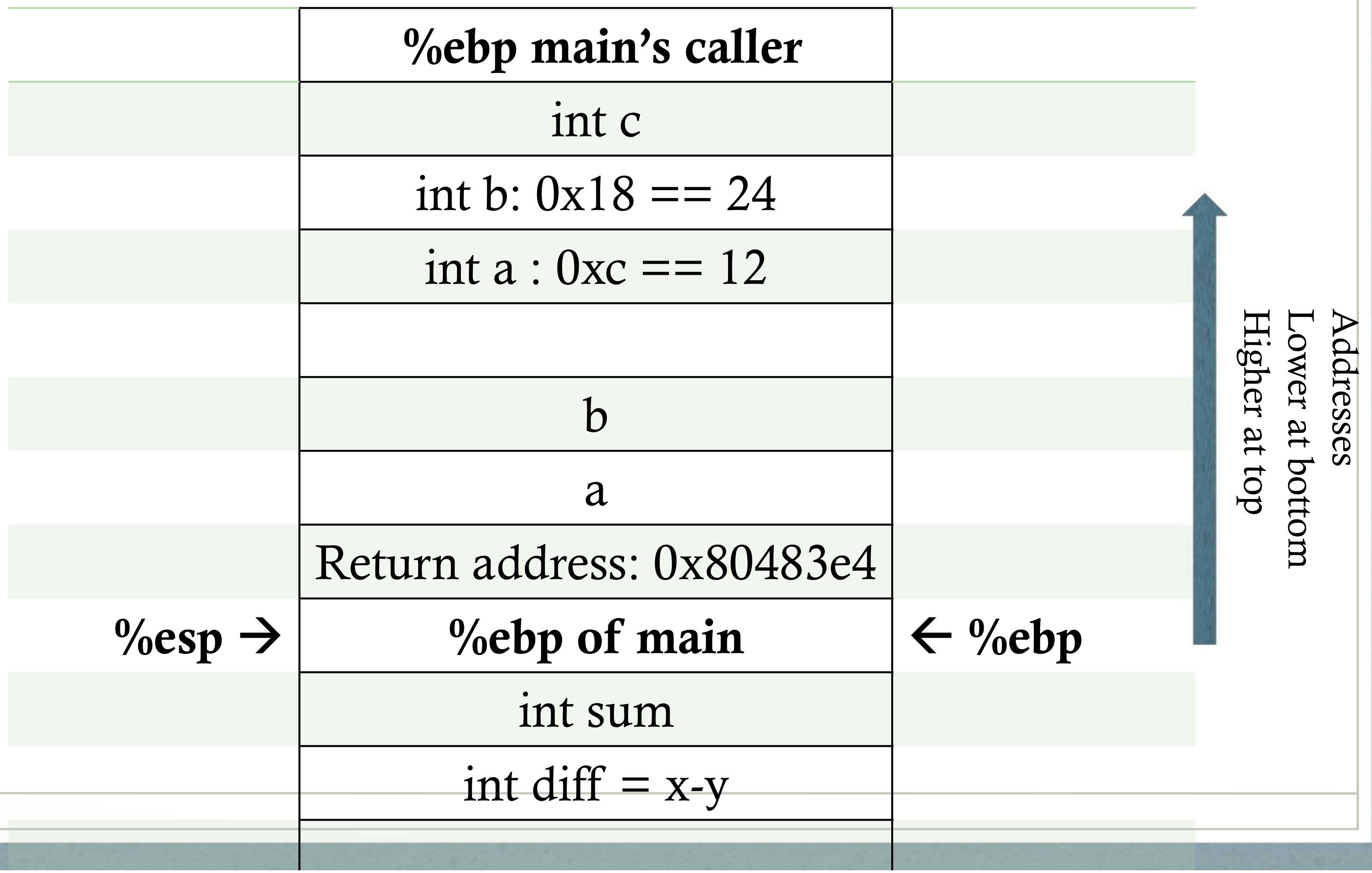
Since by x86 conventions, the result of a function is left in %eax, we do not need to anything further.

After executing First part of Instruction : 0x80483bc: leave

Set up stack for returning to main.

Part 1: moves %ebp into %esp

Part 2: pops from stack into %ebp.



After executing Second part of Instruction : 0x80483bc: leave

Set up stack for returning to main.

Part 1: moves %ebp into %esp

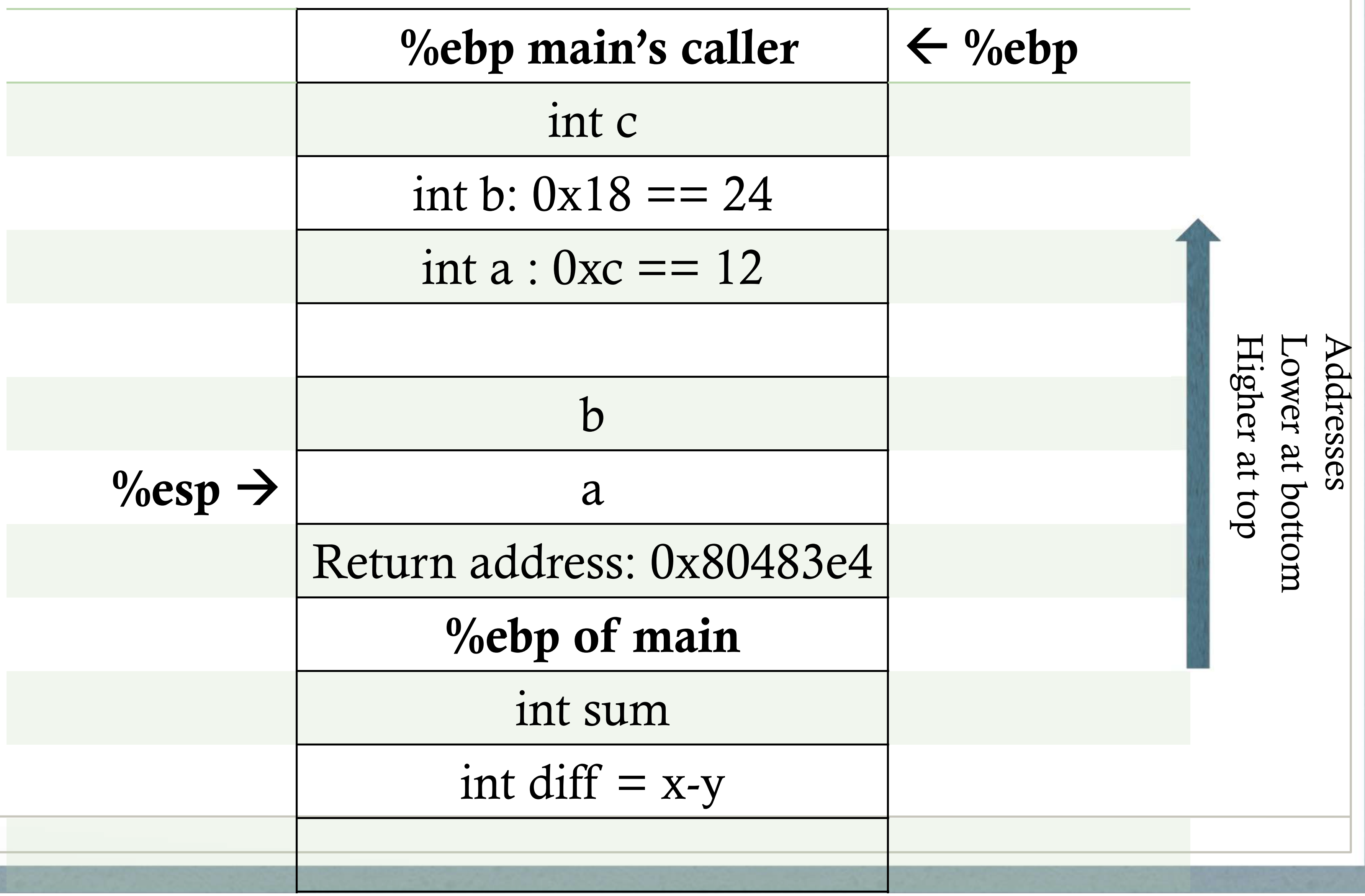
Part 2: pops from stack into %ebp.

	%ebp main's caller	← **%ebp**
	int c	
	int b: 0x18 == 24	
	int a : 0xc == 12	
	b	
	a	
%esp →	Return address: 0x80483e4	
	%ebp of main	
	int sum	
	int diff = x-y	

Addresses
Lower at bottom
Higher at top

After executing Instruction : 0x80483bd: ret

Return to main by poping into %eip



After executing Instruction : 0x80483e4: mov %eax,-0x4(%ebp)

Store result into local variable c

	%ebp main's caller	← **%ebp**
	int c = return value of func1()	
	int b: 0x18 == 24	
	int a : 0xc == 12	
	b	
%esp →	a	
	Return address: 0x80483e4	
	%ebp of main	
	int sum	
	int diff = x-y	

Addresses
Lower at bottom
Higher at top

After executing Instruction : 0x80483e7: mov $0x0,%eax

Store result of main (value 0) into %eax by x86 convention

	%ebp main's caller	← **%ebp**
	int c = return value of func1()	
	int b: 0x18 == 24	
	int a : 0xc == 12	
	b	
%esp →	a	
	Return address: 0x80483e4	
	%ebp of main	
	int sum	
	int diff = x-y	

Addresses
Lower at bottom
Higher at top

After executing Part 1 of Instruction : 0x80483ec: leave

Set up stack for returning to main.

Part 1: moves %ebp into %esp

Part 2: pops from stack into %ebp.

%esp →	**%ebp main's caller**	**← %ebp**
	int c = return value of func1()	
	int b: 0x18 == 24	
	int a : 0xc == 12	
	b	
	a	
	Return address: 0x80483e4	
	%ebp of main	
	int sum	
	int diff = x-y	

Addresses
Lower at bottom
Higher at top

After executing Part 2 of Instruction : 0x80483ec: leave

Set up stack for returning to main.

Part 1: moves %ebp into %esp

Part 2: pops from stack into %ebp.

%ebp main's caller
int c = return value of func1()
int b: 0x18 == 24
int a : 0xc == 12
b
a
Return address: 0x80483e4
%ebp of main
int sum
int diff = x-y

Addresses
Lower at bottom
Higher at top

Register Saving x86 conventions

What if caller does

```
a:   add    %edx, %ecx
     call   b
     add    %ecx, %eax
```

Terminology: %ecx is *live* across the call to b

We need a *convention* that the compiler can implement for

1. responsibility for saving/restoring register contents
2. location of saved register contents

IA 32 convention

caller save

`%eax`   `%edx`   `%ecx`

callee save

`%ebx`   `%esi`   `%edi`

Which is `%ebp` ?

With the convention, a's code becomes

```
a:   addl   %edx, %ecx
     pushl   %ecx
     call    b
     popl   %ecx
     add   %ecx, %eax
```

using caller save register

How a caller/parent uses a callee save register

```
parent: pushl  %ebp
        movl   %esp, %ebp
        subl   $16, %esp       includes space for callee save regs
        movl   %ebx, -4(%ebp)   save callee save registers
        movl   %edi, -8(%ebp)
           ( use %ebx and %edi )
        call   child
           ( use %ebx and %edi some more )
        movl   -4(%ebp), %ebx   restore callee save registers
        movl   -8(%ebp), %edi
        leave
        ret
```

Example Program on Register Calling Conventions