CS 461 / ECE 422
Discussion #1

x86 Assembly Review

Paul Murley – pmurley2@illinois.edu

8/28/19

Overview

- Introductions
- Discussion Structure
- MPs - What to expect
- Review – Assembly Programming and Memory Structure

Discussion Sections

- Attendance recommended but not required
- Time will mostly be spent on topics that will help with the MPs
- Questions encouraged*
  - If you have a longer, specific question (e.g., your code is not working for the MP and you are not sure why), you should come to office hours rather than staying after discussion
- No screens policy does not apply in discussion section (but please be courteous)

Structure of an MP (Project)

- 5 MPs, each MP lasts about 2 – 2.5 weeks
- Split into two parts (Checkpoint 1 and Checkpoint 2)
- Scored out of 120 points (C1: 20pts, C2: 100pts)
- Checkpoint 1 will be ***significantly*** easier/faster to complete when compared Checkpoint 2
    - What does this mean?
- Emphasis: **NO LATE SUBMISSIONS ACCEPTED**

MP Handout and Submission: Git

MP Handout and Submission: Git

- Create your personal Git repository for this course (if you haven’t already):
  1. Follow link (also found on either Piazza or course website):

     https://edu.cs.illinois.edu/create-ghe-repo/cs461-fa19/

  2. Log in with your netid/password
  3. That’s it! Your repository is now available at:

     https://github-dev.cs.Illinois.edu/cs461-fa19/<netid>

MP Submission/Handout Workflow

- Empty/skeleton files for each MP will be pushed to your repository as a new branch
  - Branch will be named for the MP topic (e.g., “AppSec”)
  - You should immediately merge this branch into your master branch

- You will submit MPs by committing and pushing your completed files to master
  - The autograder will read from the master branch only!

MP1: Application Security

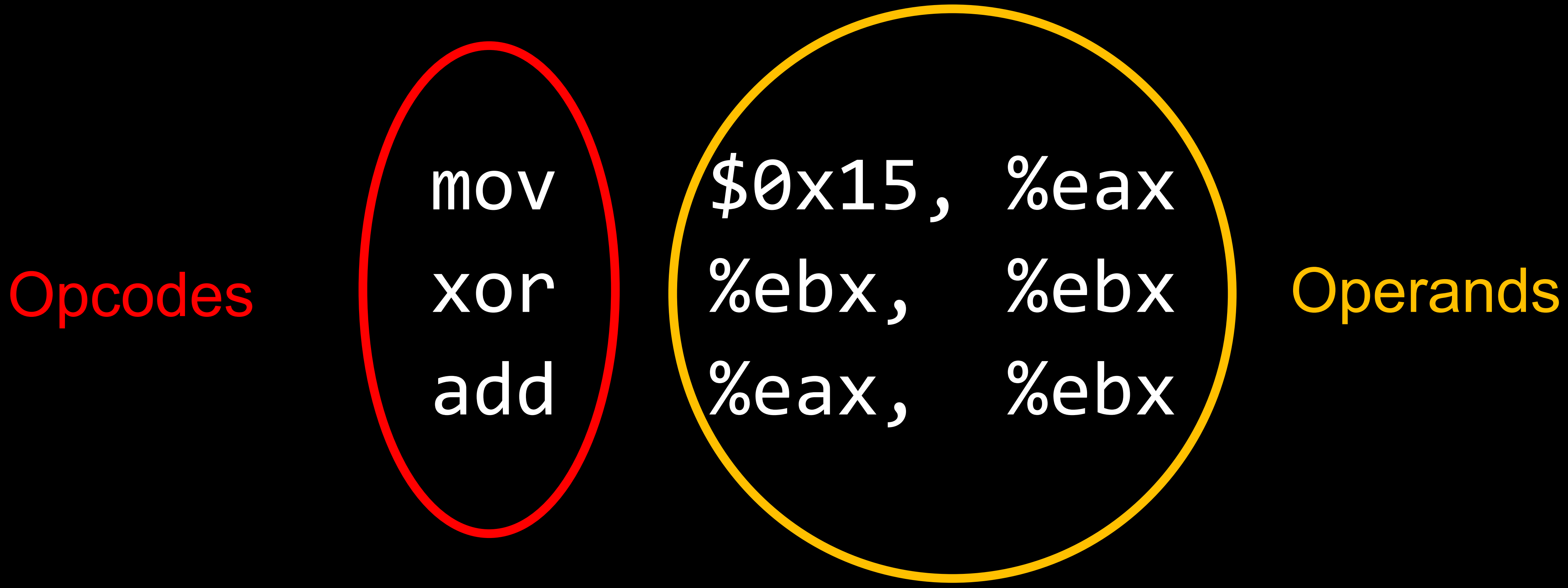
- Released: Monday, 9/2 @ 6:00pm
- Involves understanding and exploiting programs using buffer overflows, integer overflows, format string vulnerabilities, etc.
- The entire MP will be done inside of a virtual machine that we will provide to you
  - You will be expected to install virtualization software (VirtualBox) on your own computer in order to run this VM. If you don’t have a computer that can handle this, contact course staff and we will help provide a working environment.

Questions before we move on?

Review: x86 Assembly

```
mov   $0x15, %eax
xor   %ebx,  %ebx
add   %eax,  %ebx
```

Review: x86 Assembly

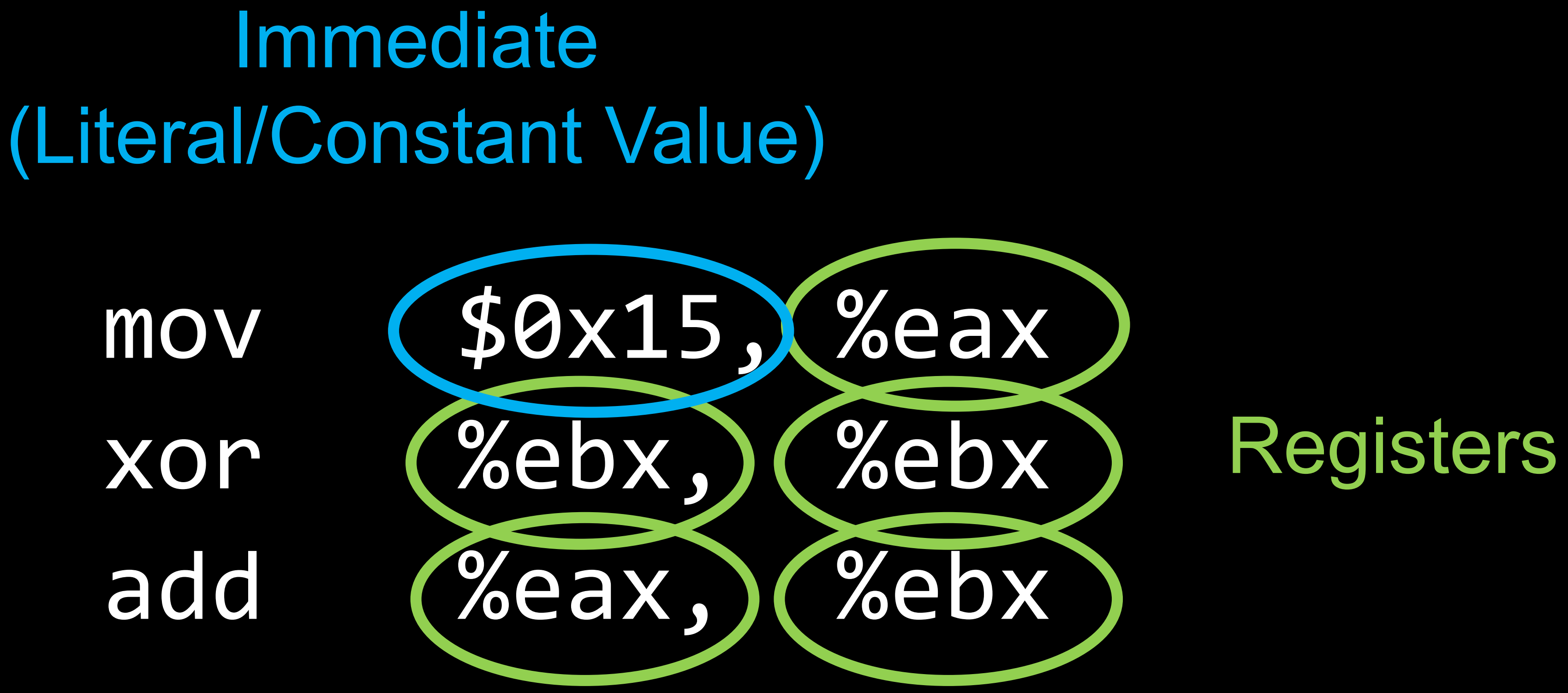


```
mov  $0x15, %eax
xor  %ebx,  %ebx
add  %eax,  %ebx
```

Opcodes

Operands

Review: x86 Assembly



Immediate
(Literal/Constant Value)

```
mov   $0x15, %eax
xor   %ebx,  %ebx
add   %eax,  %ebx
```

Registers

Commonly Used x86 Registers

General purpose registers

- EAX - Return value
- EBX
- ECX - Loop counter
- EDX
- EDI - Repeated destination
- ESI - Repeated source

Special Registers

- EBP – Frame pointer/Base pointer
- ESP - Stack pointer
- EIP - Program counter
- EFLAGS - Status of previous operations (used in conditionals)

x86 Assembly Syntax

There are two main variants of x86 syntax:

Intel

- `add eax, [ebx+4]`
- Mnemonic, then operands
- Destination operand first, then source
- Brackets indicate memory access

AT&T (GAS)

- `add 4(%ebx), %eax`
- Mnemonic, then operands
- Source operand first, then destination
- Parentheses indicate memory access

x86 Assembly Syntax

There are two main variants of x86 syntax:

Intel

- `add eax, [ebx+4]`
- Mnemonic, then operands
- Destination operand first, then source
- Brackets indicate memory access

AT&T (GAS)

- `add 4(%ebx), %eax`
- Mnemonic, then operands
- Source operand first, then destination
- Parentheses indicate memory access

In this course, we use AT&T (GAS) syntax exclusively.

AT&T Memory Address Calculation

Write it:

```
displacement (base_reg, offset_reg, multiplier)
```

Calculate it:

```
base_reg + displacement + (offset_reg*multiplier)
```

```
mov   8 (%ebp), %eax          # Mem[EBP+8] to eax
mov  12 (,%edx,4), %eax       # Mem[EDX*4+12] to eax
```

Notice that not all fields are required!

Common x86 Instructions (Opcodes) (1)

Arithmetic Operations

- `add`, `sub` - add/subtract data in first operand to/from second
- `inc`, `dec` - increment/decrement operand
- `neg` - change sign of operand

Logical Operations

- `and`, `or`, `xor` - bitwise and/or/xor
- `not` - flip all of the bit values
- `shl`, `shr` - shift bits left/right

Common x86 Instructions (Opcodes) (2)

Transfer Instructions

- `mov` - copy data from first operand to second
- `lea` - compute address and store it in second operand (does NOT access memory)
- `push` - Push the operand onto the stack (see later slides)
- `pop` - Pop a value off the top of the stack into the operand

Common x86 Instructions (Opcodes) (3)

Transfer Instructions

- `jmp` – jump to label or address specified by operand
- `je` - jump if equal
- `jne` - jump if not equal
- `jz` - jump if zero
- `jg` - jump if greater than
- `jl` - jump if less than
- `jle/jge` - jump if equal or less than/greater than

For conditional jumps, EFLAGS is used. EFLAGS is a register set by the CMP and TEST instructions (and all other arithmetic instructions)

Common Instructions – Exercise 1

```
xor %eax, %eax              # eax = eax ^ eax

add $3, %eax                # eax = eax + 3

dec %eax                    # eax = eax - 1

shl $3, %eax                # eax = eax << 3

lea 4(, %eax, 4), %eax      # eax = 4 * eax + 4
```

What is in EAX?

Common Instructions – Exercise 2

```
mov   $11,%eax           11 -> eax

mov   $12,%ebx           12 -> ebx

mov   $8,%ecx            8 -> ecx

add   %ecx,%ebx          ebx = ebx + ecx

sub   %ecx,%eax          eax = eax - ecx
```

What is in EAX?

Common Instructions – Exercise 3

9 -> EAX

0 -> EBX

ECX -> EDX

M[ECX] -> EBX

M[EDX+4] -> EAX

Remember:

Opcode-Source-Destination

Address calculation:

displacement(base_reg, offset_reg, multiplier)

Translate into valid AT&T Assembly

Common Instructions – Exercise 3

9 -> EAX	mov	$9, %eax
0 -> EBX	xor	%ebx, %ebx
ECX -> EDX	mov	%ecx, %edx
M[ECX] -> EBX	mov	(%ecx), %ebx
M[EDX+4] -> EAX	mov	4(%edx), %eax

Translate into valid AT&T Assembly

GAS/AT&T Memory Syntax Example

```
typedef struct {
  int a, b, c, d;
} foo_t;
foo_t my_foos[10];
```

```
my_foos[5].c = 461;
```

```
mov my_foos, %ebx
mov $5, %ecx
mov $461, 8(%ebx, %ecx, 16)
```

32-bit x86 ISA

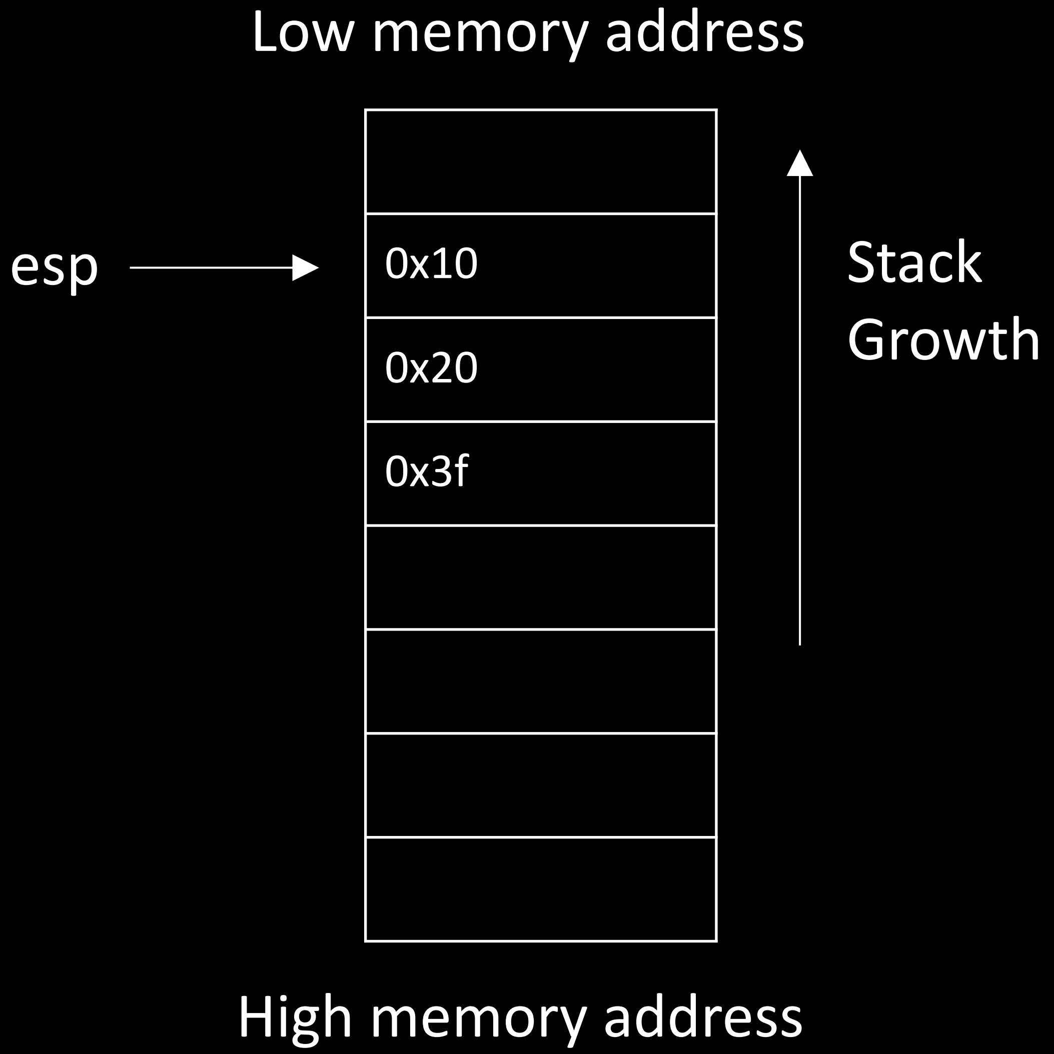
- 1 byte = 8 bits
- char -> 1 byte
- integer -> 4 bytes
- word -> 2 bytes (in gdb, word -> 4 bytes)
- Memory address -> 4 bytes
- Pointer -> 4 bytes
- Registers -> 4 bytes
- Each memory location -> 1 byte

The Stack

- Stores working data (local variables, function arguments, return addresses, etc)
- Last-in First-out (LIFO) structure
- Grows downwards (towards lower memory addresses)
- Manipulated with `push` and `pop` instructions

The Stack

- ESP (stack pointer) points to the top of the stack



Low memory address

esp

0x10

0x20

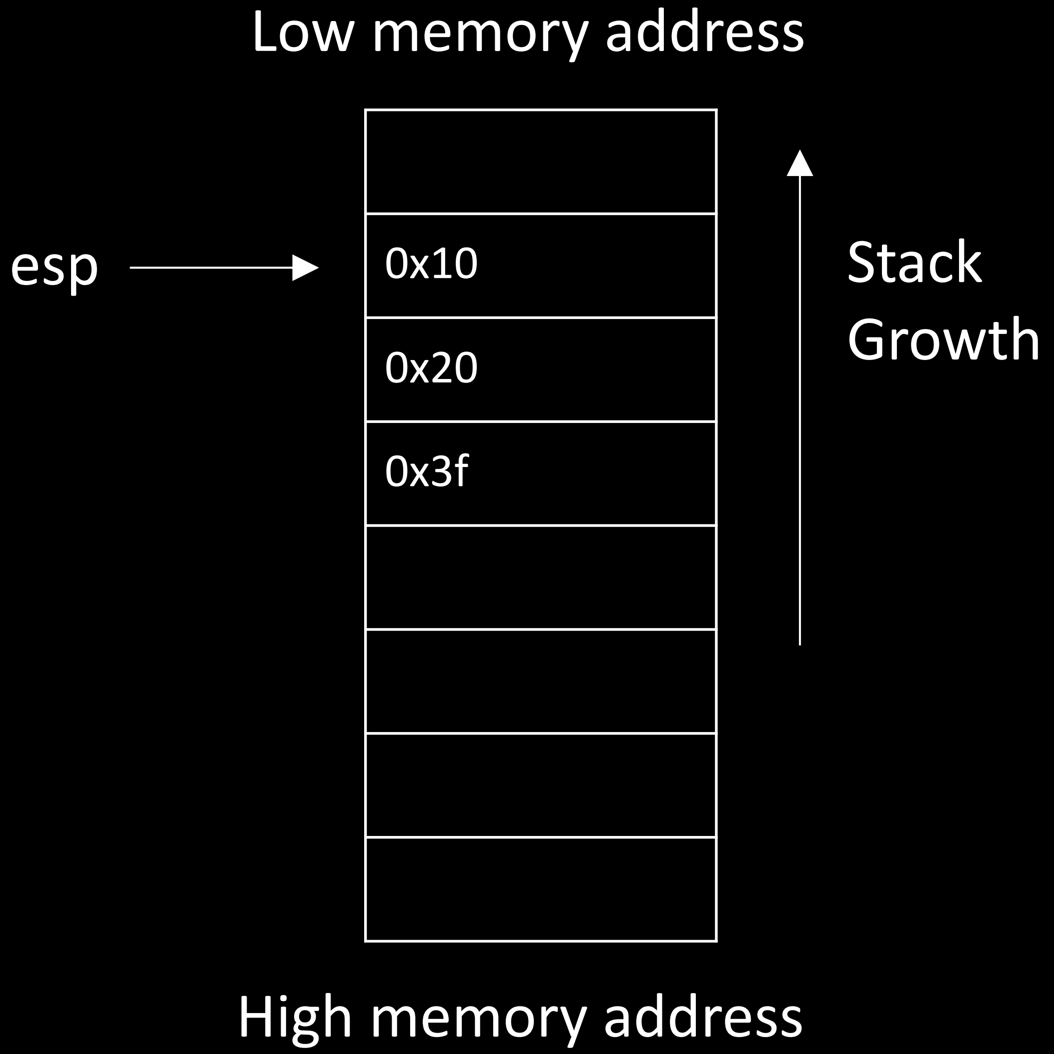
0x3f

Stack Growth

High memory address

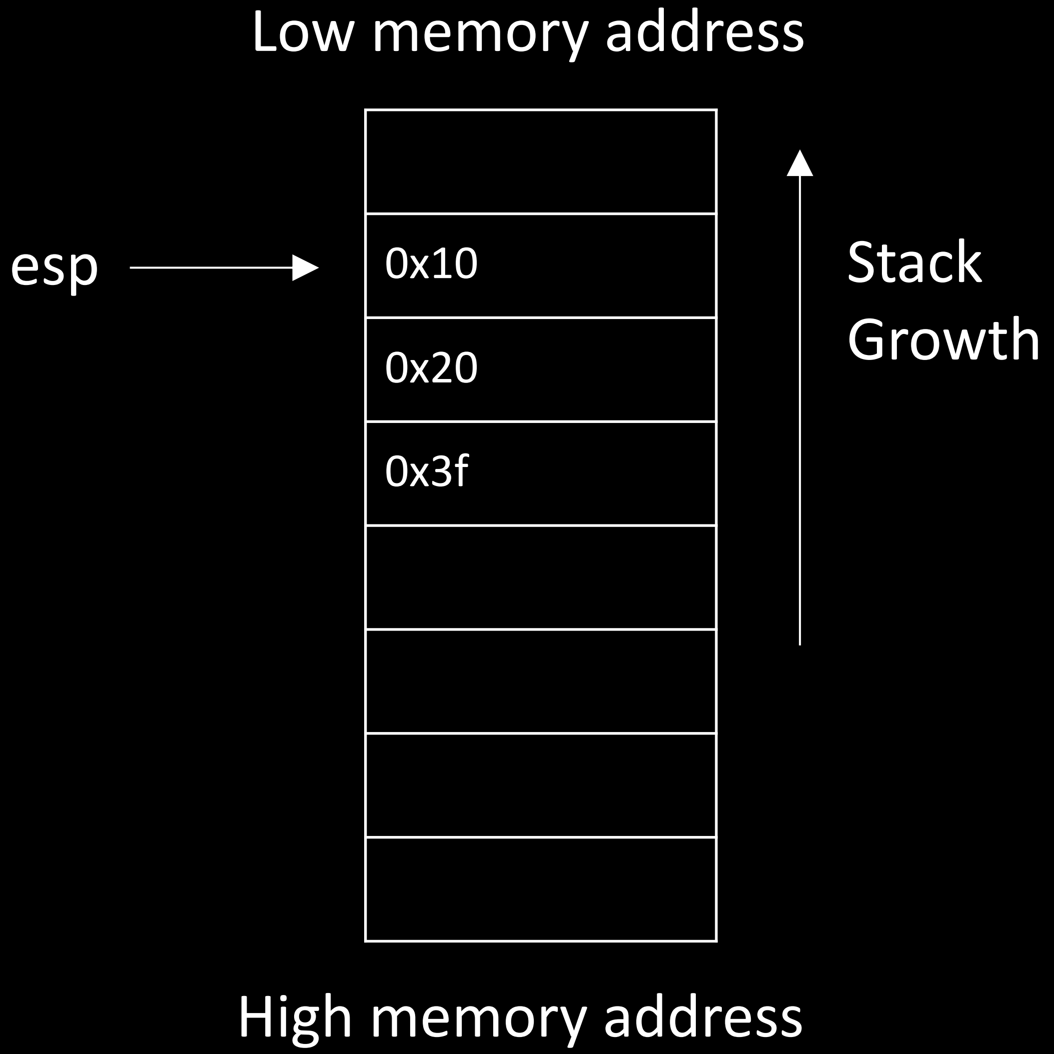
The Stack

- ESP (stack pointer) points to the top of the stack
- push instruction decrements ESP (subtracts), and then writes to the top of the stack



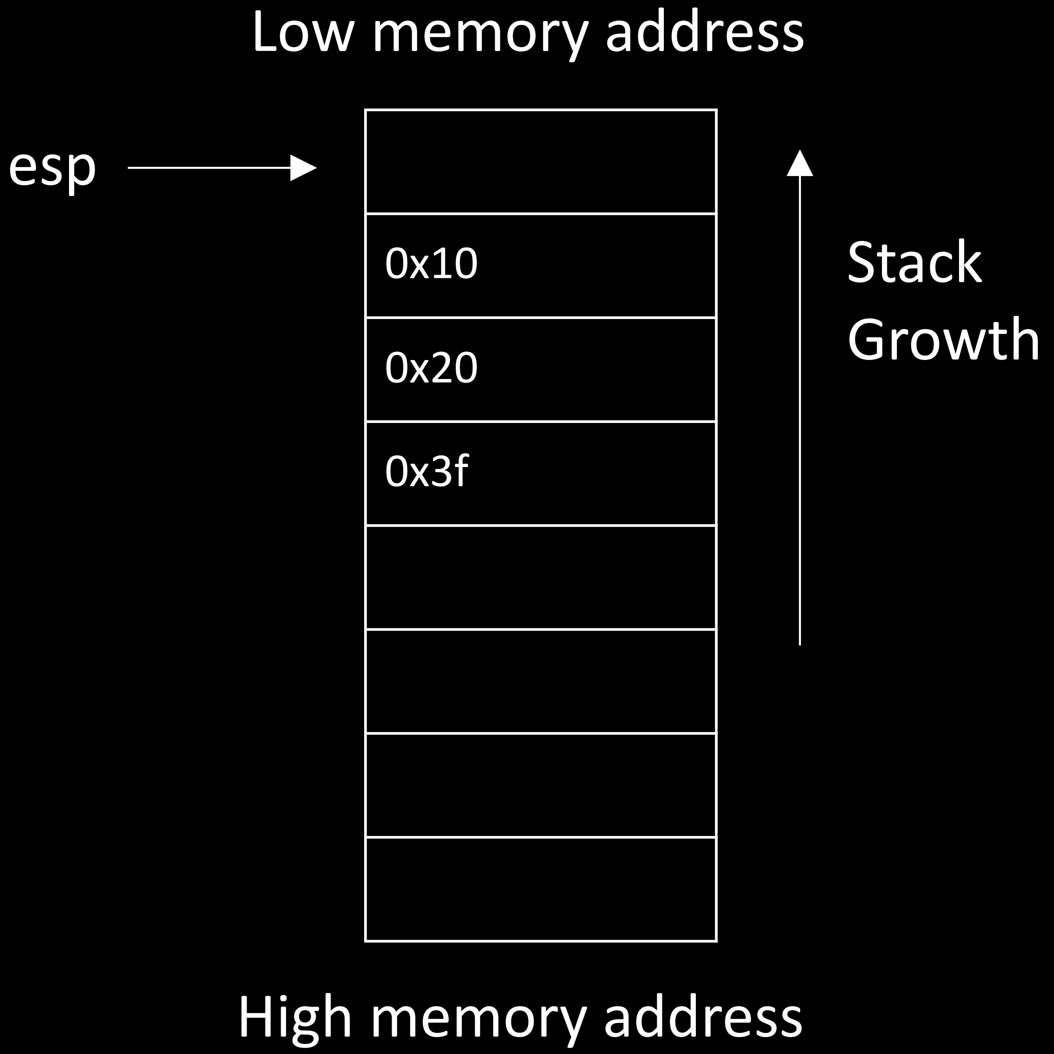
The Stack

- ESP (stack pointer) points to the top of the stack
- `push` instruction decrements ESP (subtracts), and then writes to the top of the stack
    - Example: `push 0x4e`



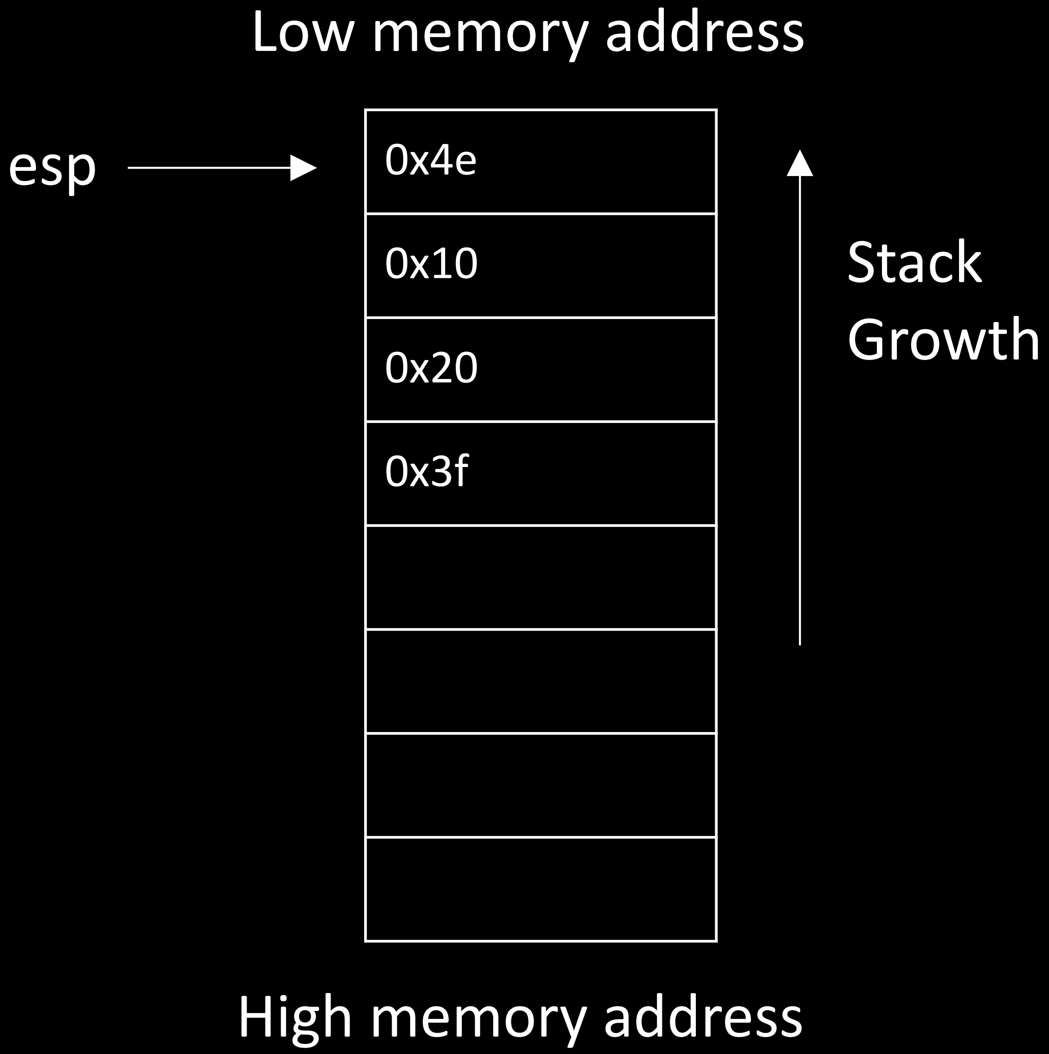
The Stack

- ESP (stack pointer) points to the top of the stack
- `push` instruction decrements ESP (subtracts), and then writes to the top of the stack
  - Example: `push 0x4e`



The Stack

- ESP (stack pointer) points to the top of the stack
- push instruction decrements ESP (subtracts 4), and then writes to the top of the stack
  - Example: `push 0x4e`



Low memory address

esp

0x4e

0x10

0x20

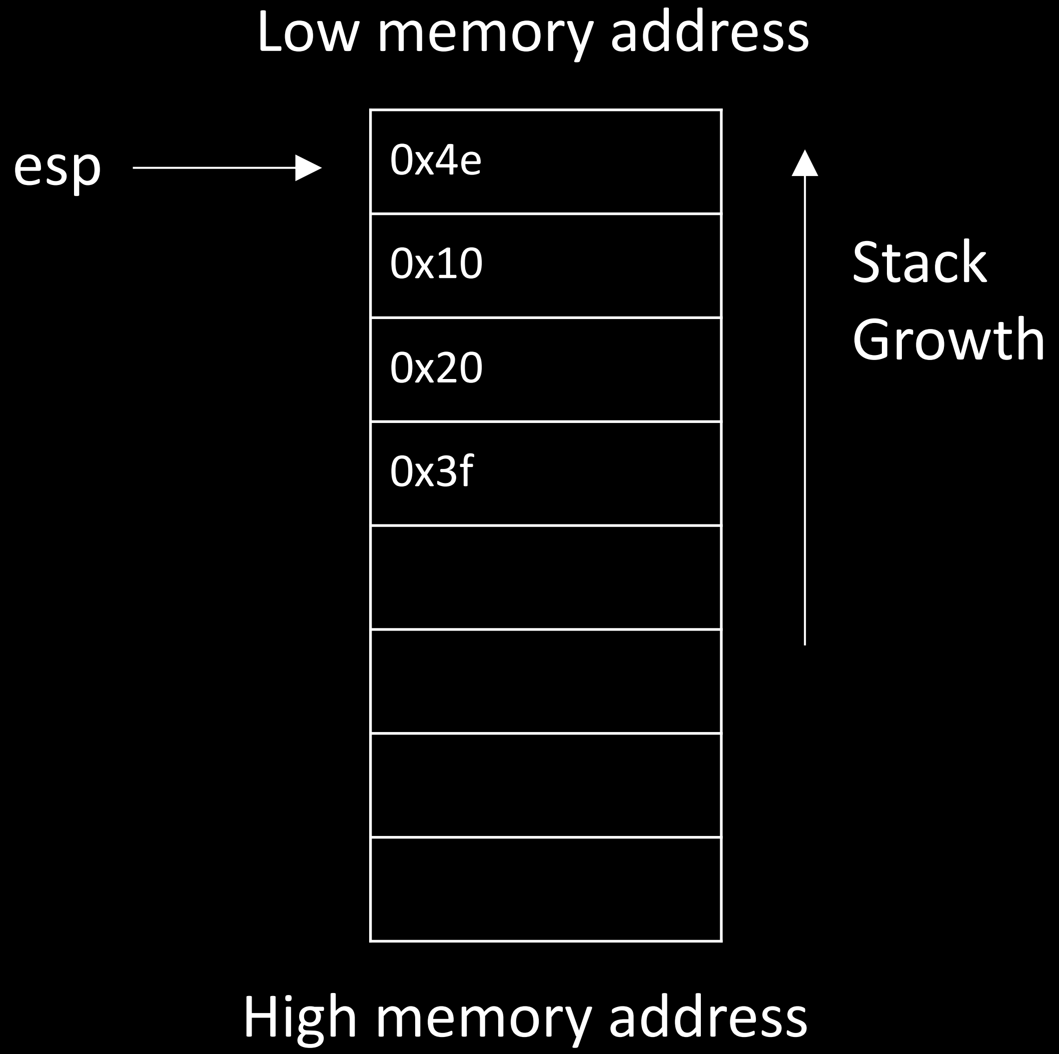
0x3f

Stack Growth

High memory address

The Stack

- ESP (stack pointer) points to the top of the stack
- `push` instruction decrements ESP (subtracts), and then writes to the top of the stack
  - Example: `push 0x4e`
- `pop` instruction reads the value on top of the stack and then decrements ESP



Low memory address

esp

0x4e

0x10

0x20

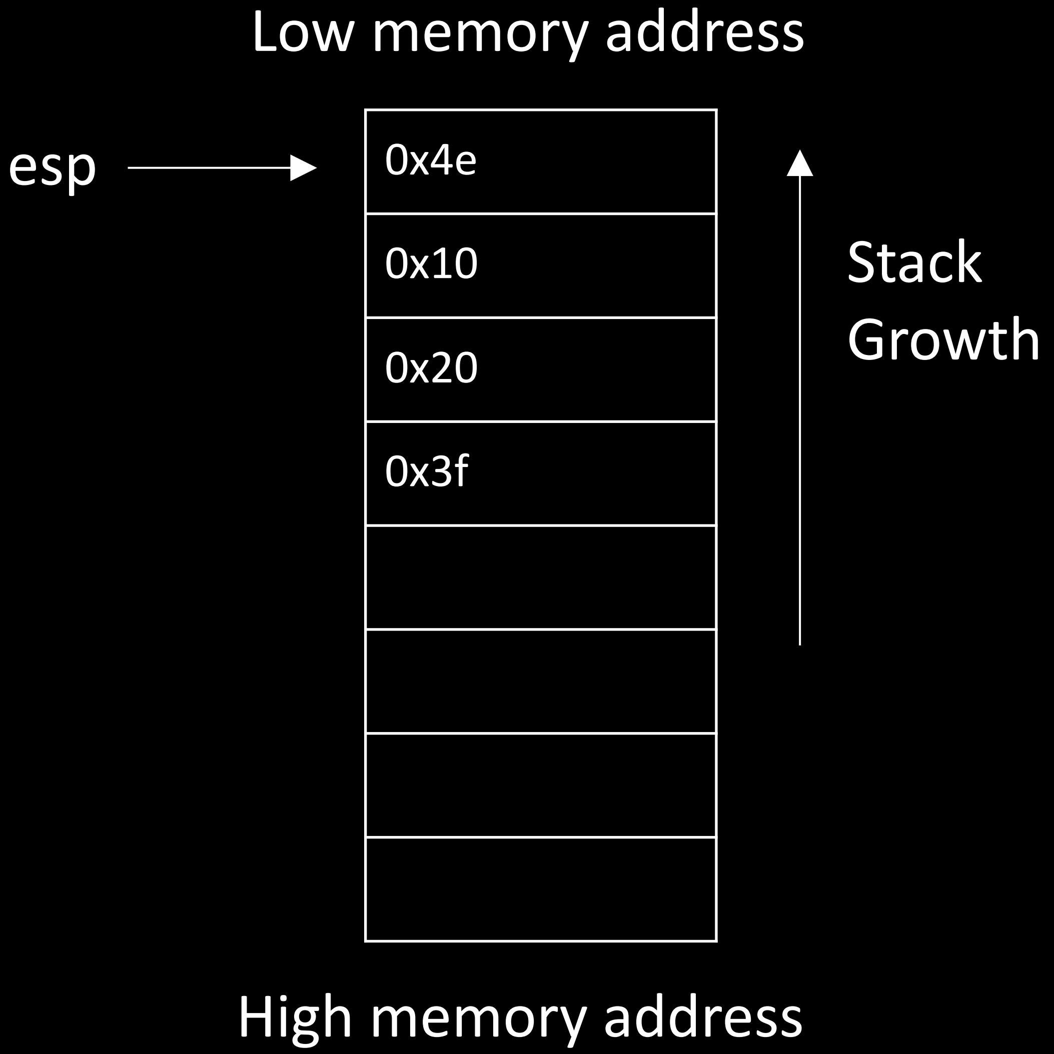
0x3f

Stack Growth

High memory address

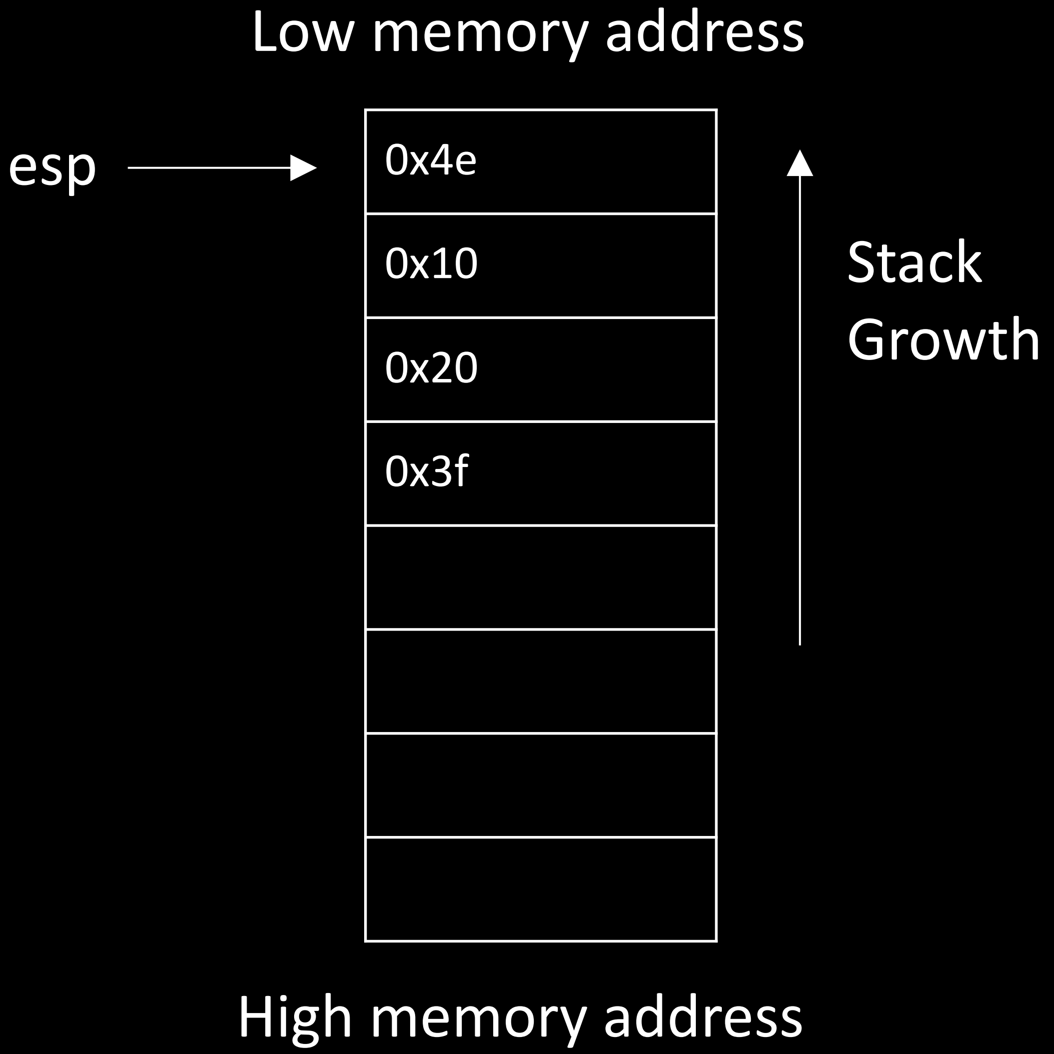
The Stack

- ESP (stack pointer) points to the top of the stack
- push instruction decrements ESP (subtracts), and then writes to the top of the stack
  - Example: `push 0x4e`
- pop instruction reads the value on top of the stack and then decrements ESP
  - Example: `pop %eax`



The Stack

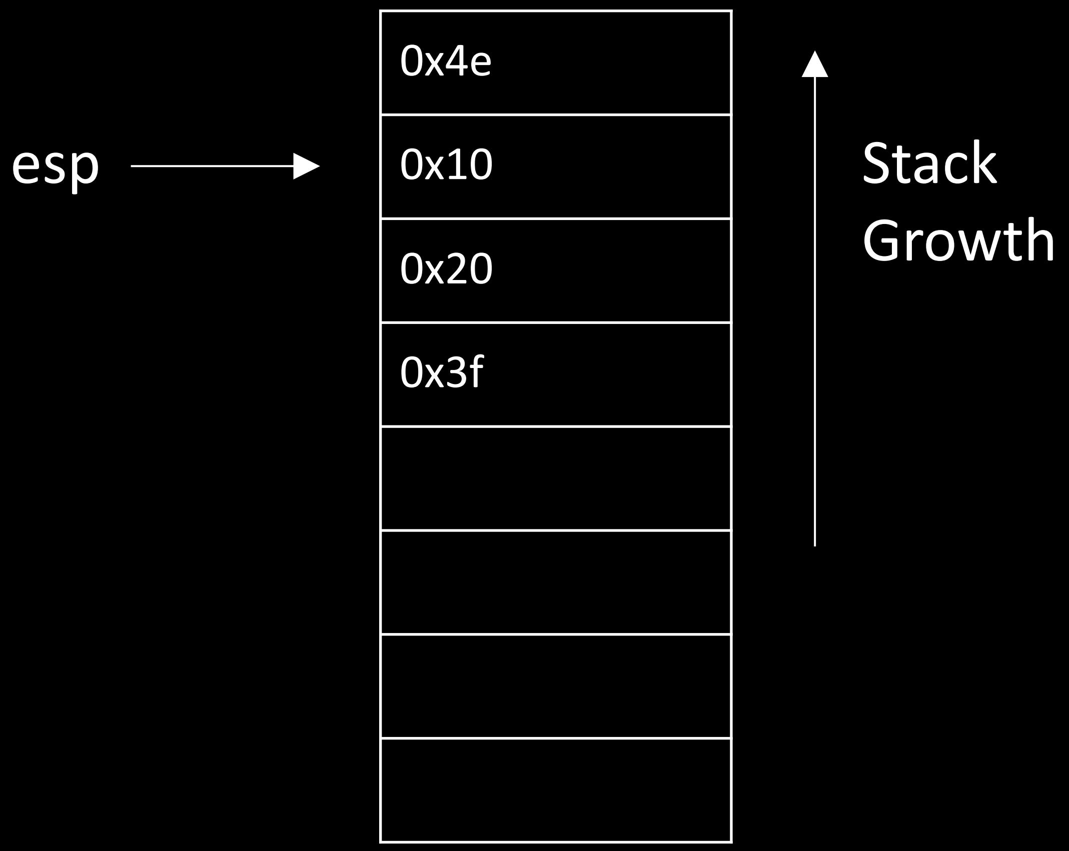
- ESP (stack pointer) points to the top of the stack
- `push` instruction decrements ESP (subtracts), and then writes to the top of the stack
  - Example: `push 0x4e`
- `pop` instruction reads the value on top of the stack and then decrements ESP
  - Example: `pop %eax` (%eax ⇐ 4)



The Stack

- ESP (stack pointer) points to the top of the stack
- `push` instruction decrements ESP (subtracts), and then writes to the top of the stack
  - Example: `push 0x4e`
- `pop` instruction reads the value on top of the stack and then decrements ESP
  - Example: `pop %eax` (%eax ⇐ 4)

Low memory address



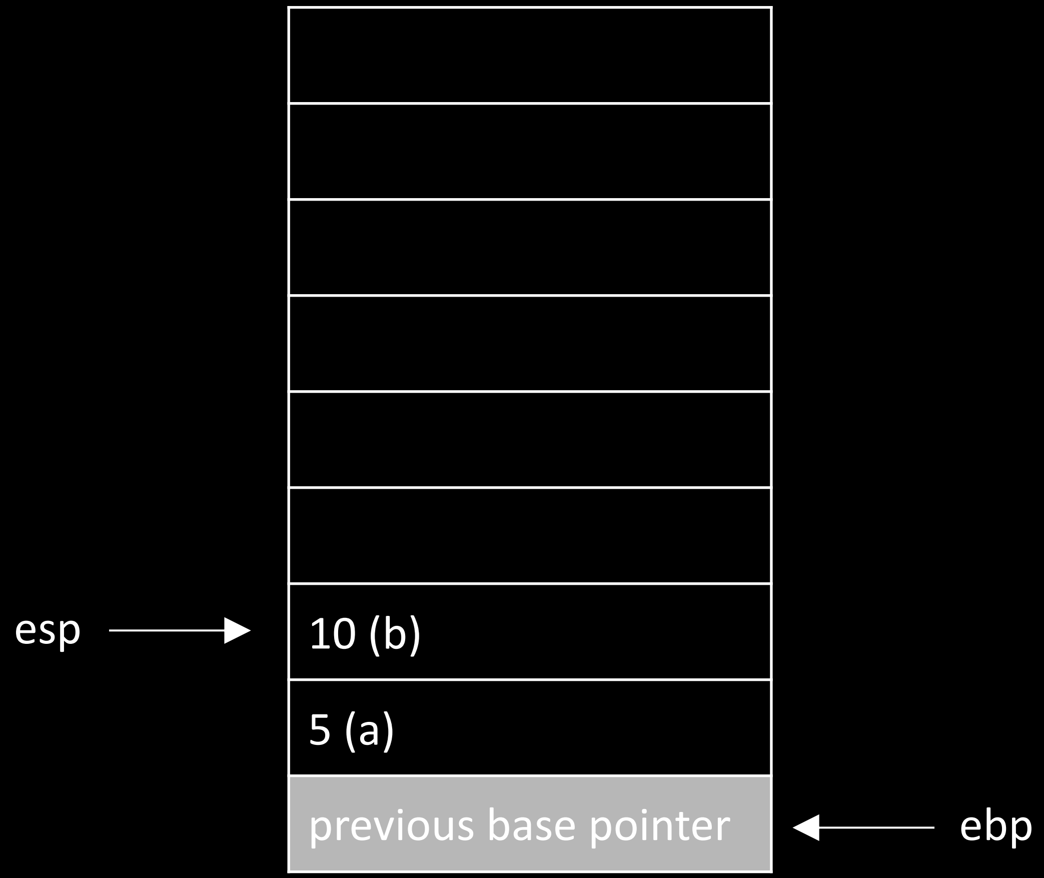
High memory address

Stack Frames

```
void bar {
     int a = 5;  // (push $5)
     int b = 10; //(push $10)
     foo(11,12);
}
```

Stack Frames

```
void bar {
     int a = 5;  // (push $5)
     int b = 10; //(push $10)
     foo(11,12);
}
```



esp

10 (b)

5 (a)

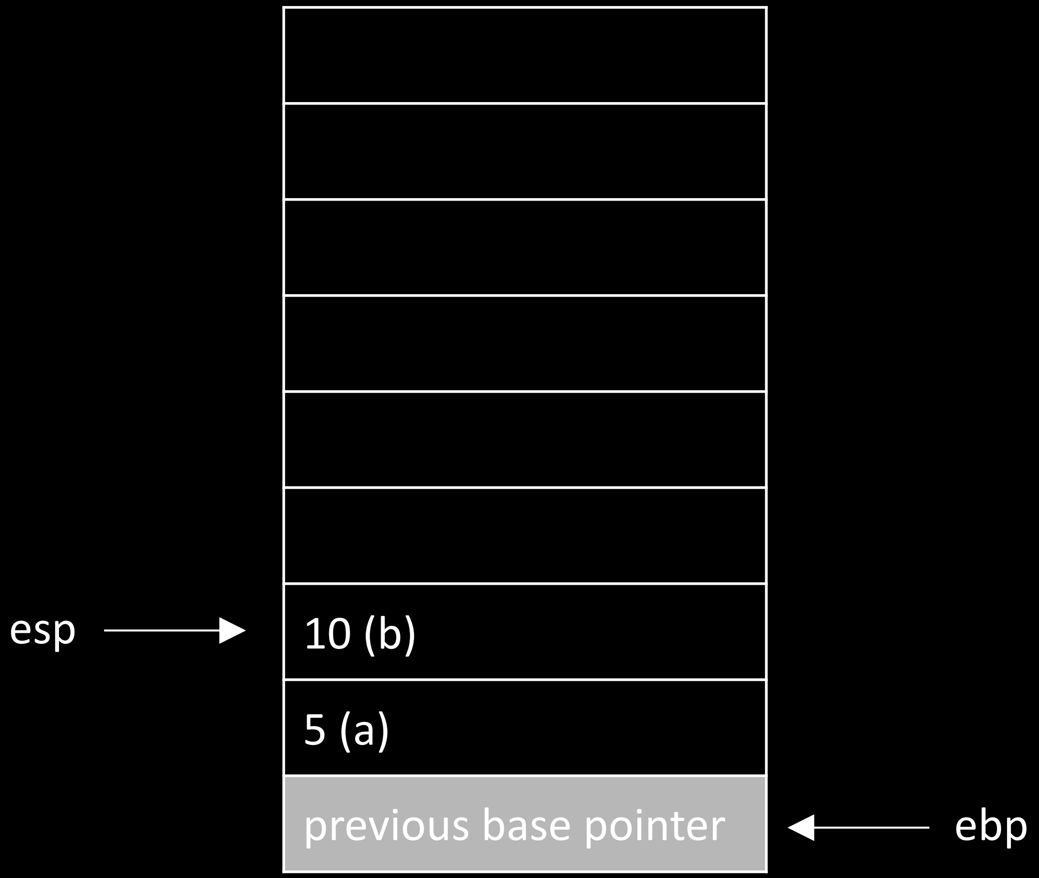
previous base pointer

ebp

Stack Frames

```
void bar {
     int a = 5;  // (push $5)
     int b = 10; //(push $10)
     foo(11,12);
}
```

1. Do stuff in bar()

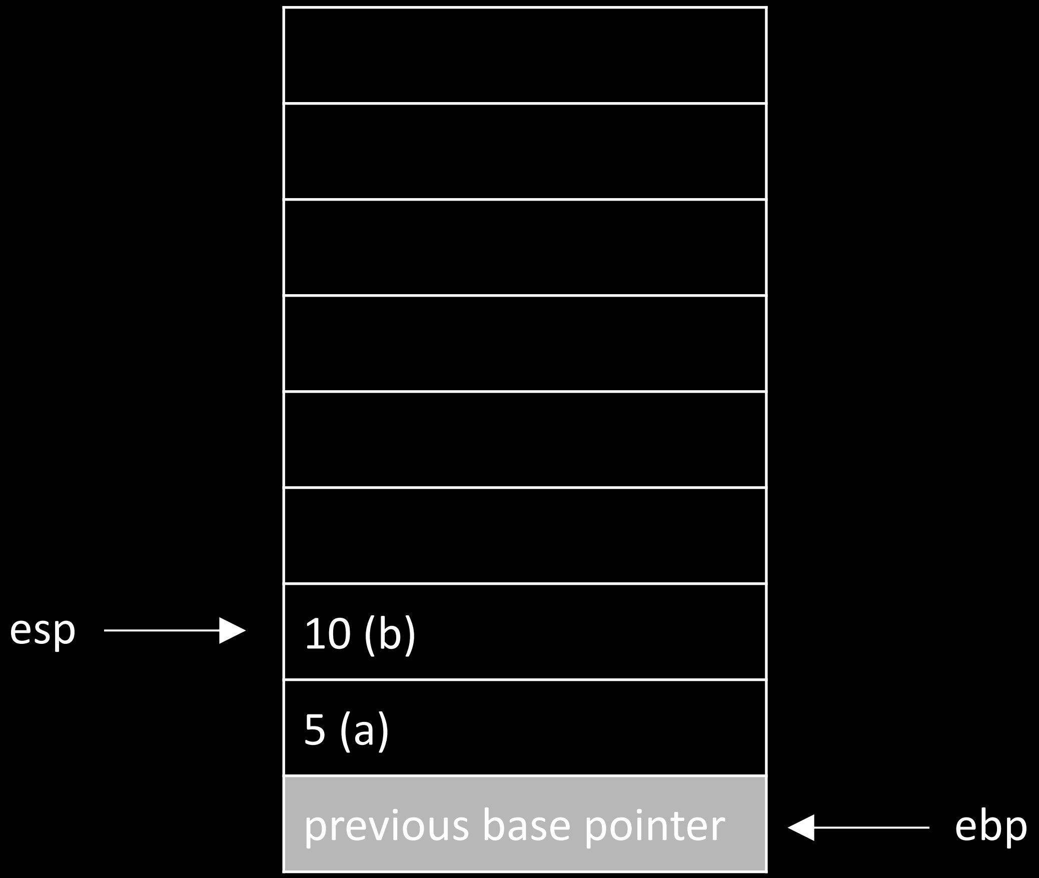


Stack Frames

```
void bar {
     int a = 5;  // (push $5)
     int b = 10; //(push $10)
     foo(11,12);
}
```

1. Do stuff in bar()
2. Set up arguments for foo()
   - Example: foo() takes 2 arguments, so we need to:

     `push $11, push $12`



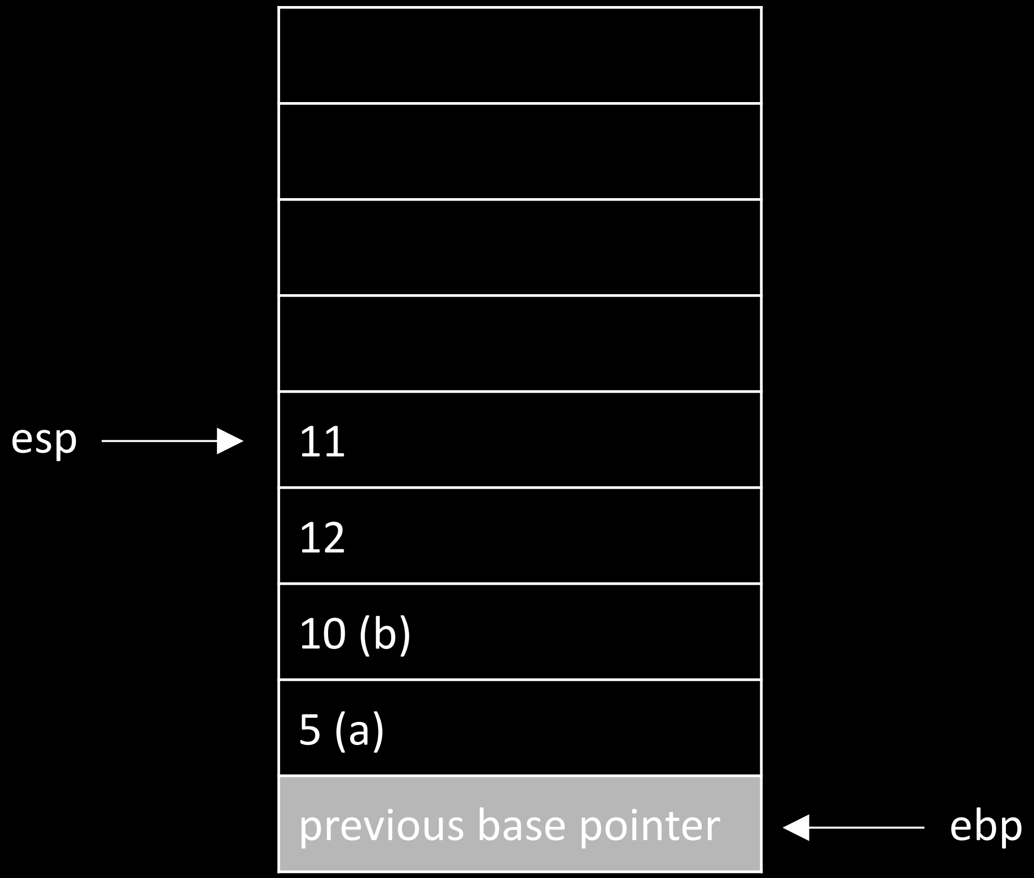
Stack Frames

```
void bar {
     int a = 5;  // (push $5)
     int b = 10; //(push $10)
     foo(11,12);
}
```

1. Do stuff in bar()
2. Set up arguments for foo()

   - Example: foo() takes 2 arguments, so we need to:

     `push $12, push $11`



esp → 11

12

10 (b)

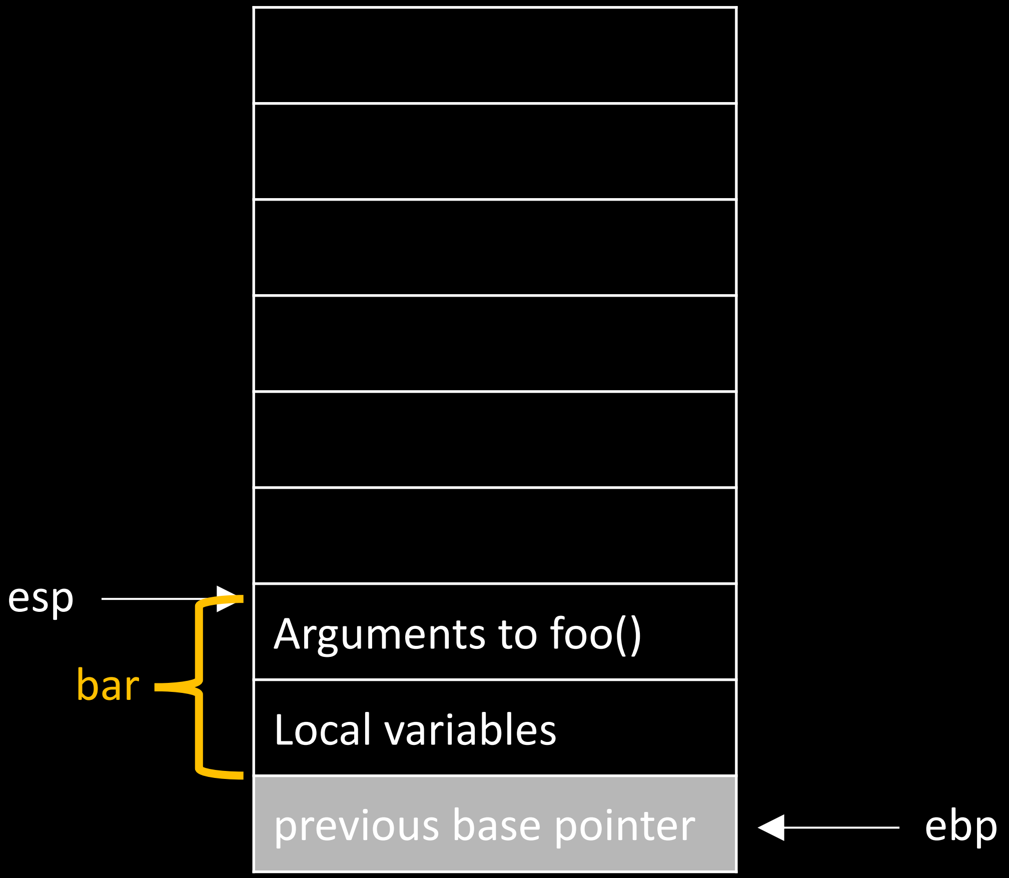
5 (a)

previous base pointer ← ebp

Stack Frames

```
void bar {
	int a = 5;  // (push $5)
	int b = 10; //(push $10)
	foo(11,12);
}
```

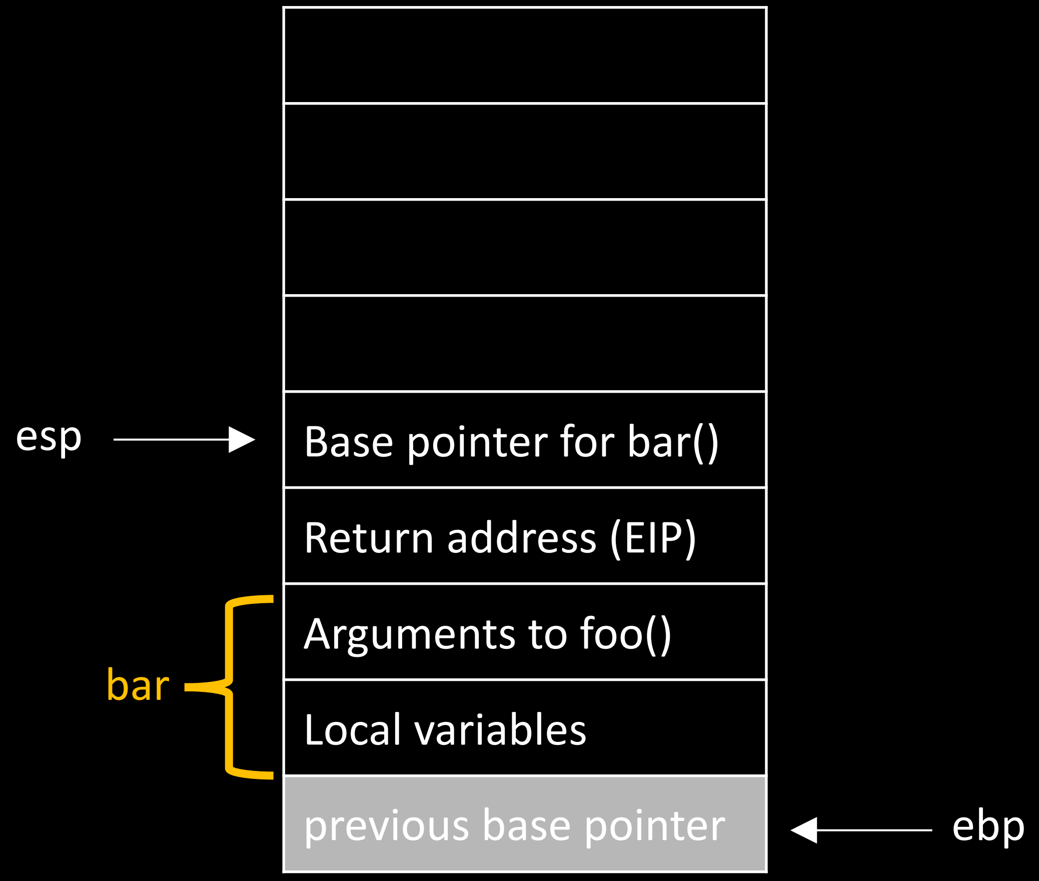
1. Do stuff in bar()
2. Set up arguments for foo()
3. Set up stack frame for foo()



Stack Frames

```
void bar {
     int a = 5;  // (push $5)
     int b = 10; //(push $10)
     foo(11,12);
}
```

1. Do stuff in bar()
2. Set up arguments for foo()
3. Set up stack frame for foo()



esp

Base pointer for bar()

Return address (EIP)

Arguments to foo()

Local variables

bar

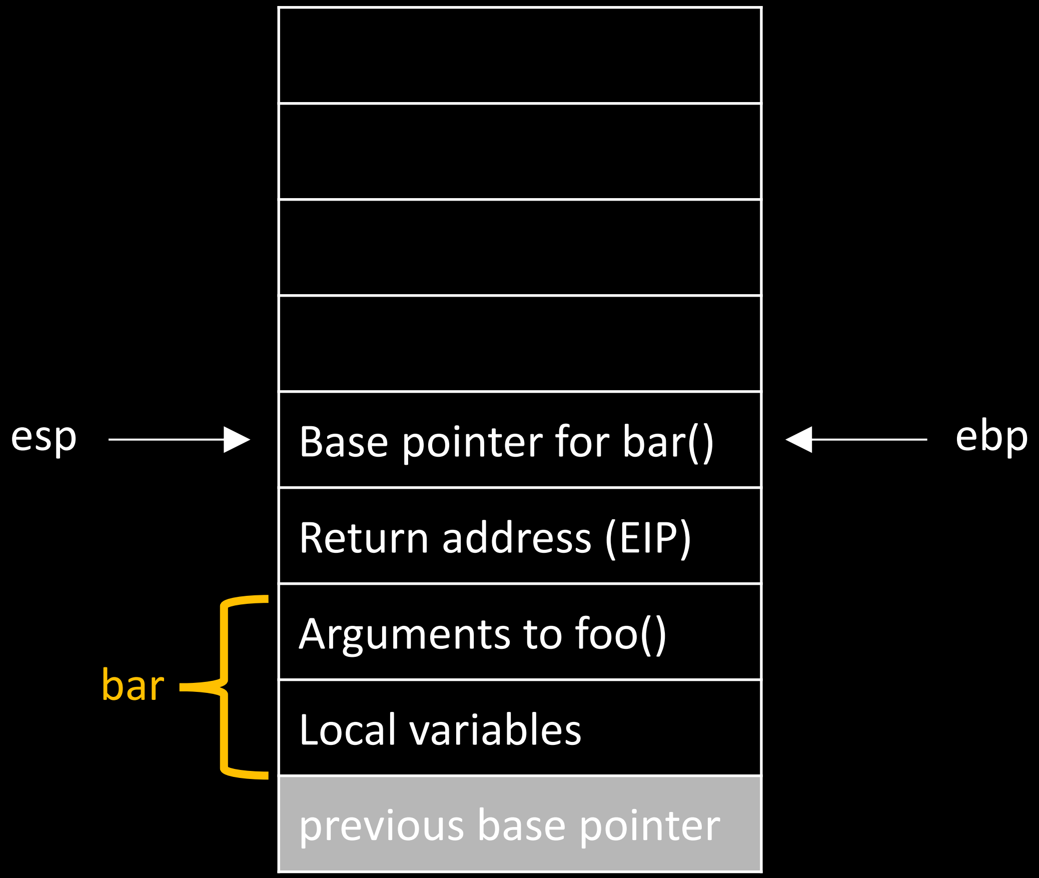
previous base pointer

ebp

Stack Frames

```
void bar {
	int a = 5;  // (push $5)
	int b = 10; //(push $10)
	foo(11,12);
}
```

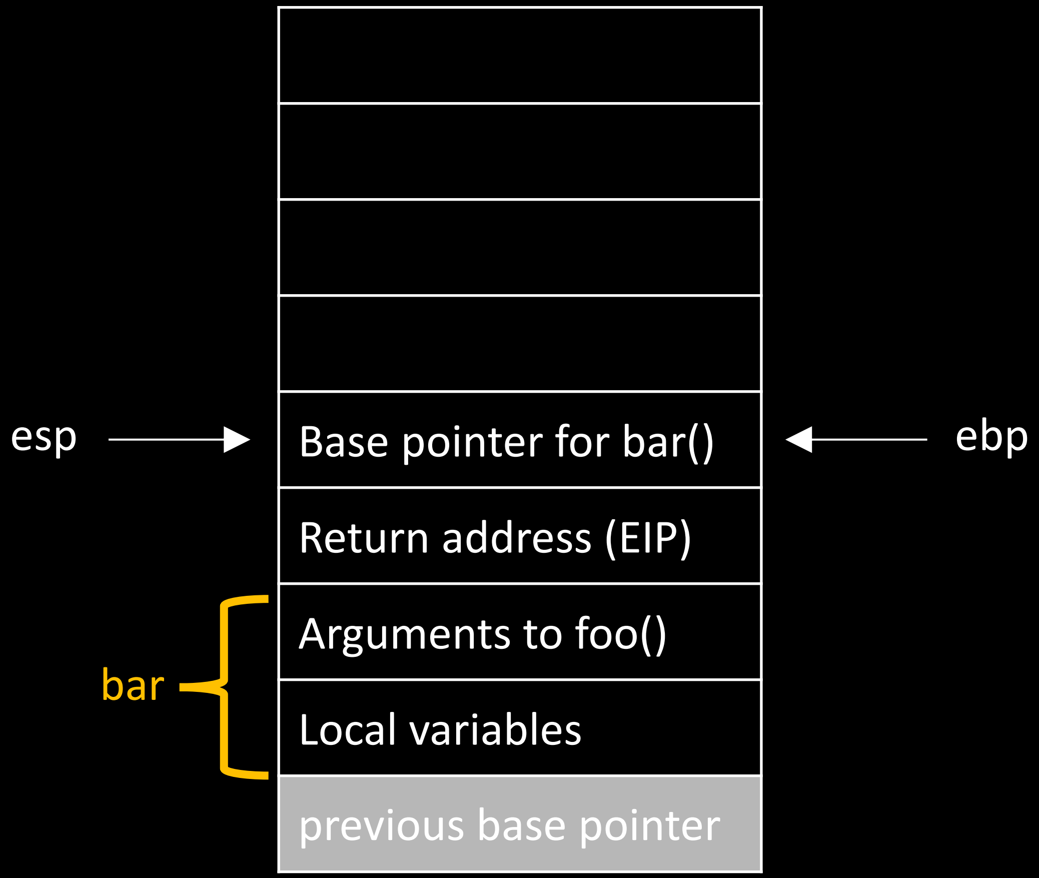
1. Do stuff in bar()
2. Set up arguments for foo()
3. Set up stack frame for foo()



Stack Frames

```
void bar {
        int a = 5;  // (push $5)
        int b = 10; //(push $10)
        foo(11,12);
}
```

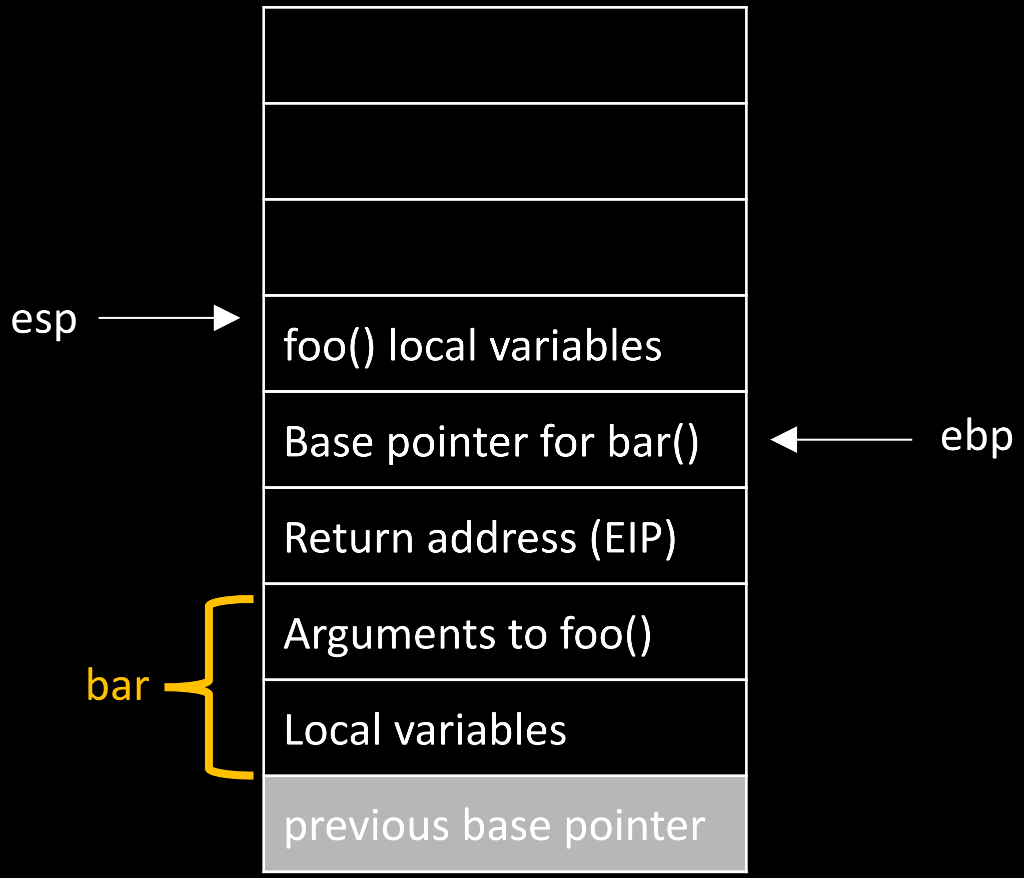
1. Do stuff in bar()
2. Set up arguments for foo()
3. Set up stack frame for foo()
4. Do stuff in foo()



Stack Frames

```
void bar {
     int a = 5;  // (push $5)
     int b = 10; //(push $10)
     foo(11,12);
}
```

1. Do stuff in bar()
2. Set up arguments for foo()
3. Set up stack frame for foo()
4. Do stuff in foo()



esp

foo() local variables

Base pointer for bar()

ebp

Return address (EIP)

Arguments to foo()

bar

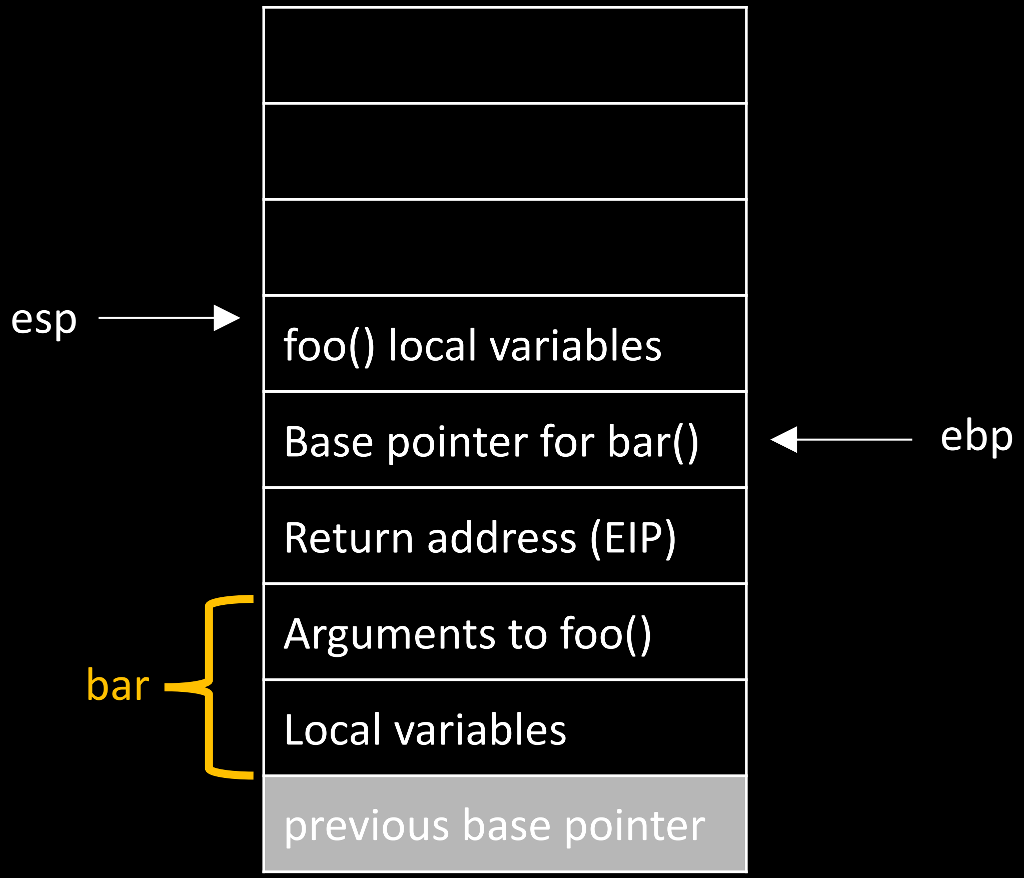
Local variables

previous base pointer

Stack Frames

```
void bar {
        int a = 5;  // (push $5)
        int b = 10; //(push $10)
        foo(11,12);
}
```

1. Do stuff in bar()
2. Set up arguments for foo()
3. Set up stack frame for foo()
4. Do stuff in foo()
5. Return to bar()

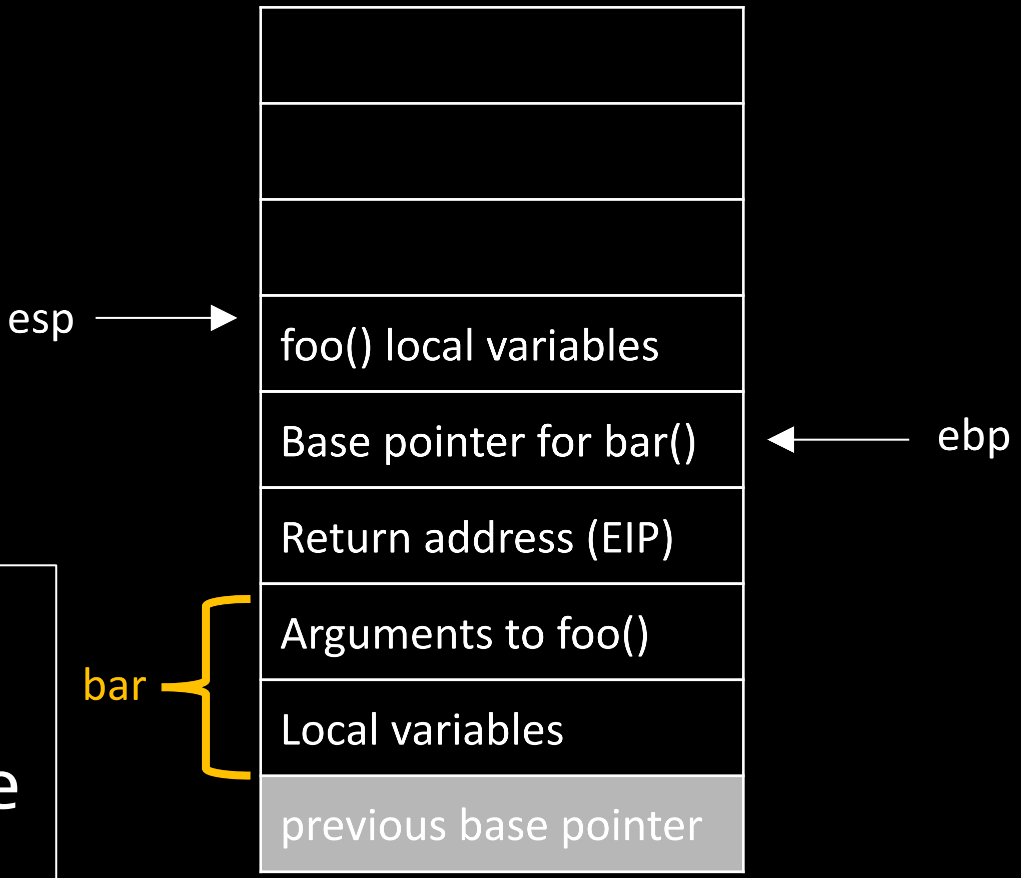


Stack Frames

```
void bar {
        int a = 5;  // (push $5)
        int b = 10; //(push $10)
        foo(11,12);
}
```

1. Do stuff in bar()
2. Set up arguments for foo()
3. Set up stack frame for foo()
4. Do stuff in foo()
5. Return to bar()

```
foo:
    …
    leave
    ret
```

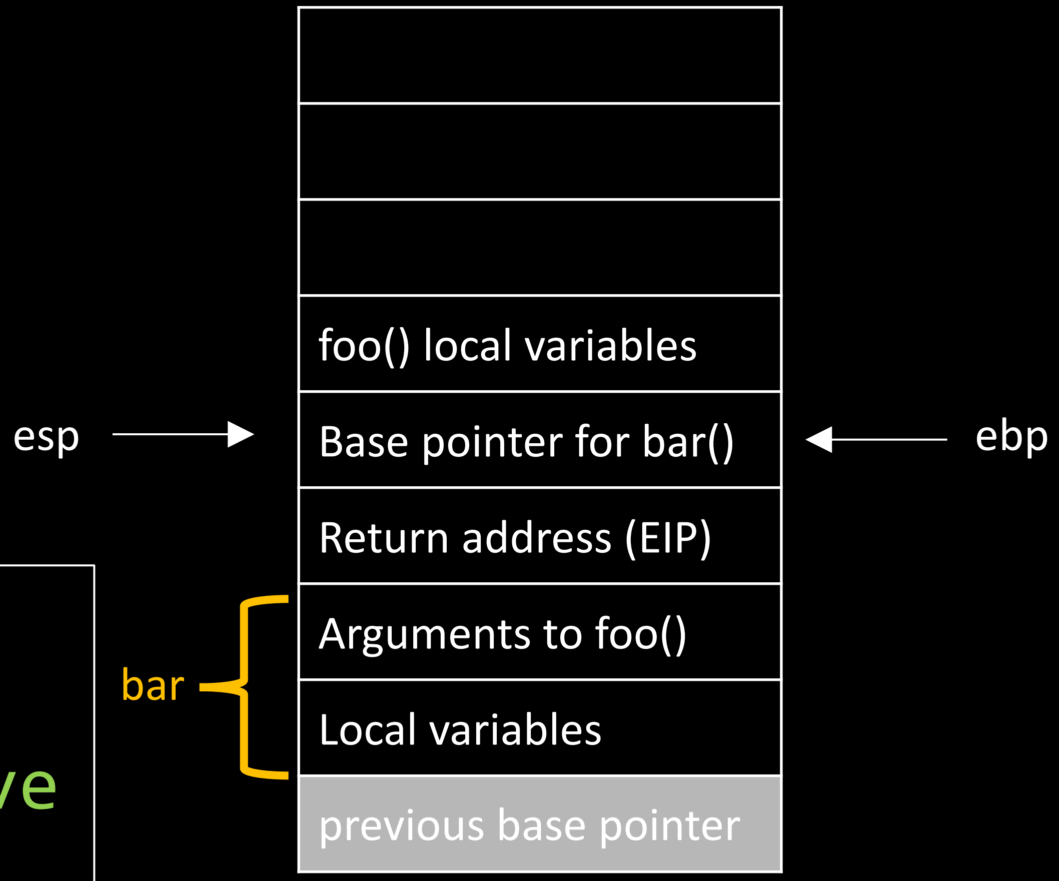


Stack Frames

```
void bar {
        int a = 5;  // (push $5)
        int b = 10; //(push $10)
        foo(11,12);
}
```

1. Do stuff in bar()
2. Set up arguments for foo()
3. Set up stack frame for foo()
4. Do stuff in foo()
5. Return to bar()

```
foo:
    …
    leave
    ret
```



foo() local variables

Base pointer for bar()

Return address (EIP)

Arguments to foo()

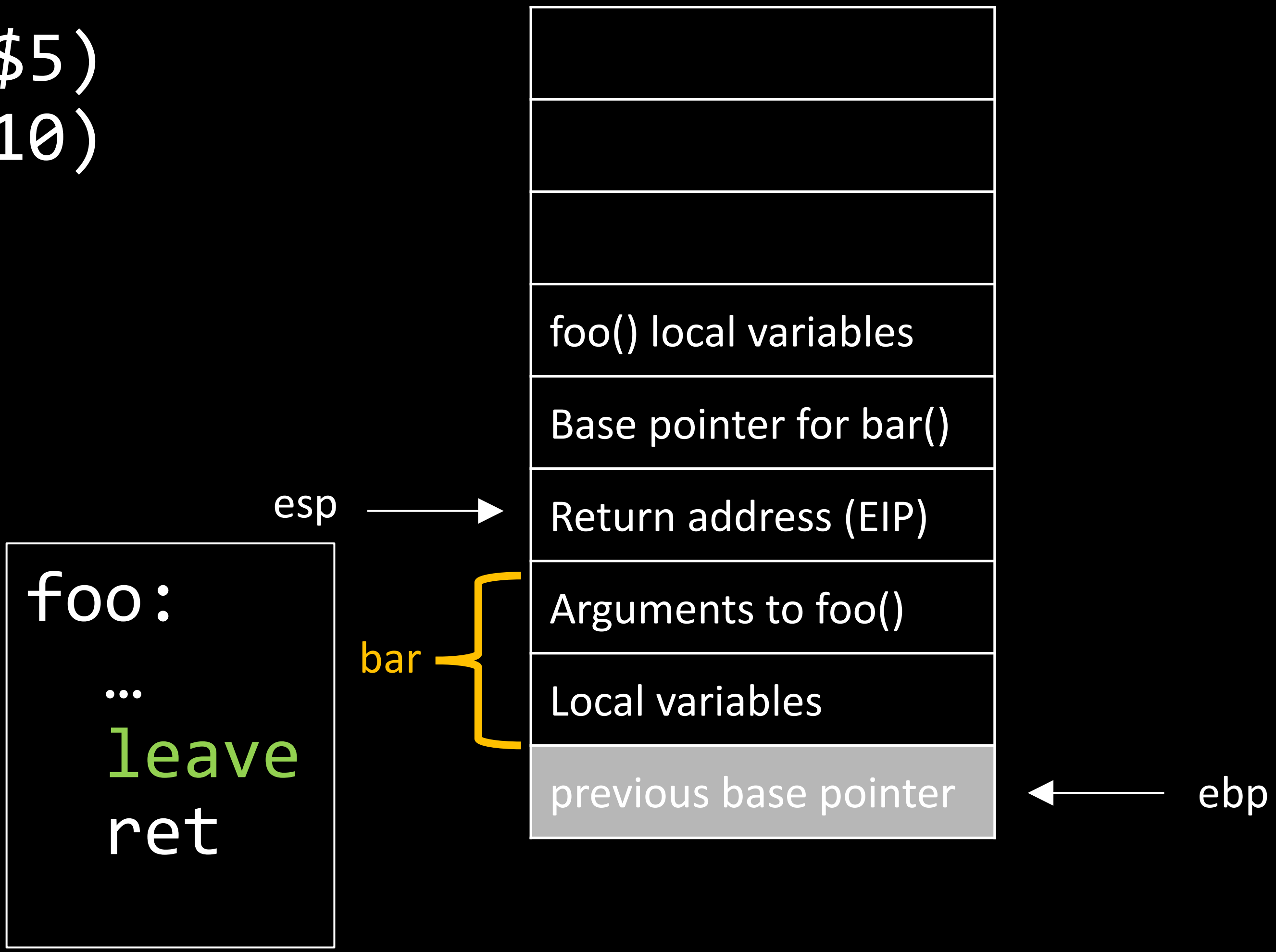
Local variables

previous base pointer

Stack Frames

```
void bar {
        int a = 5;  // (push $5)
        int b = 10; //(push $10)
        foo(11,12);
}
```

1. Do stuff in bar()
2. Set up arguments for foo()
3. Set up stack frame for foo()
4. Do stuff in foo()
5. Return to bar()



```
foo:
    …
    leave
    ret
```

foo() local variables

Base pointer for bar()

Return address (EIP)

Arguments to foo()

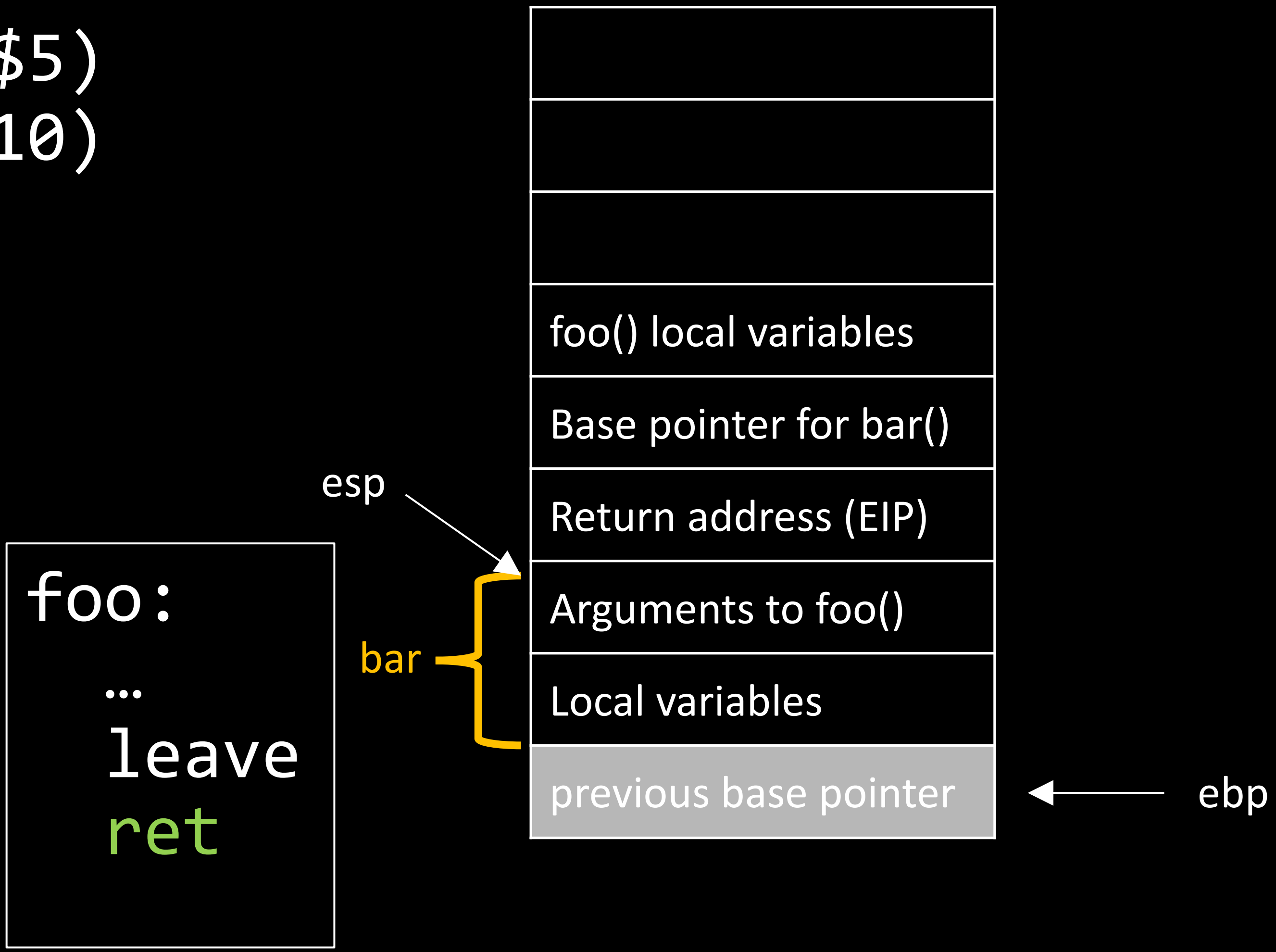
Local variables

previous base pointer

Stack Frames

```
void bar {
     int a = 5;  // (push $5)
     int b = 10; //(push $10)
     foo(11,12);
}
```

1. Do stuff in bar()
2. Set up arguments for foo()
3. Set up stack frame for foo()
4. Do stuff in foo()
5. Return to bar()



```
foo:
    …
    leave
    ret
```

Exercise: Translate to x86 Assembly

```
int func() {
    int a = 3;
    addnumbers(2, 6);
}

int addnumbers(int x, int y) {
    int b = 1;
    b = x + y;
    return b;
}
```

Possible Solution

```
func:
  push %ebp
  mov %esp,%ebp
  push $3 //int a = 3;
  push $6 //addnumbers(2,6);
  push $2
  call addnumbers
  leave
  ret
```

```
addnumbers:
  push %ebp
  mov %esp, %ebp
  push $1   //int b = 1;
  mov 8(%ebp), %eax
  add 12(%ebp),%eax
  mov %eax,(%esp)
  leave
  ret
```

Final Reminders

- MP1 Release: Monday, 9/2 @ 6:00pm
- Office Hours
  - Every weeknight from 5:00pm-7:00pm (starting 9/3)
  - Room: Siebel 4405
- Discussion Next Week
  - gdb tutorial
  - MP1 Checkpoint 1 Tips
- Contact
  - Paul Murley
  - pmurley2@illinois.edu
  - CSL 445