CSE 127: Intro to Computer Security

WI24

Lecture 2 - Security Mindset/X86

Please sit in the center rows.

Slide Credit: Kirill Levchenko, Stefan Savage, Deian Stefan, Nadia Heninger, Earlence Fernandes, Imani Munyaka

Announcements

HW 1 due next week Friday
Discussion on 1/15

We follow the University calendar. OH cancelled on Holidays

Reminder: Check google calendar for OH

Webclicker Check

What is your favorite ice cream flavor?

A - who cares
B- this is just a check
C - always hit C
D - ice cream is good
E - this is the last option

Secure Design

Common mistake:

Convince yourself that the system is secure

Better approach:

Identify *weaknesses* of design, focus on correcting them

Formally prove that design is secure (soon)

Secure design is a **process**

- Must be practiced continuously
- Retrofitting security is super hard

Where to focus defenses

- *Trusted components*
  Parts that must function correctly for the system to be secure.
  Attack surface
- Parts of the system exposed to the attacker

Security Principles

- Simplicity, open design, and maintainability
- Privilege separation and least privilege
- Defense-in-depth and diversity
- Complete mediation and fail-safe

Review of Security Foundations

- **Threat Model defines Security Goals:** What are we trying to protect from whom?
  - – Assets and Attackers
- Let's look at one definition of a secure system
  - – "System that remains dependable in the face of malice" -Ross Anderson
- What does "remains dependable" mean?
  - – Confidentiality, Integrity, and/or Availability (C.I.A.) of particular system, component, or data are preserved
  - – These are the Assets we need to protect
- "An attacker must not be able to compromise the [Confidentiality | Integrity | Availability] of…"
  - – Assets have value: Must understand value in order to decide how much to spend on protection
- What does "malice" mean?
  - – Malice from whom? Curious roommate? Criminal? Law enforcement? Foreign military?
  - – These are potential Attackers
- Defined by Capability and Intent

Review of Security Foundations

Confidentiality – obligation to prevent asset content from unauthorized users

Integrity – assurance that the asset is trustworthy and accurate

Availability – guarantee of reliable access to the asset by authorized people

Review of Security Foundations

Confidentiality – obligation to prevent asset content from unauthorized users

Integrity – assurance that the asset is trustworthy and accurate

Availability – guarantee of reliable access to the asset by authorized people

- Which security property is violated if someone …

– unplugs your alarm clock while you're sleeping?

Review of Security Foundations

Confidentiality – obligation to prevent asset content from unauthorized users

Integrity – assurance that the asset is trustworthy and accurate

Availability – guarantee of reliable access to the asset by authorized people

- Which security property is violated if someone …

– unplugs your alarm clock while you're sleeping?

– changes the time on your alarm clock?

Review of Security Foundations

Confidentiality – obligation to prevent asset content from unauthorized users

Integrity – assurance that the asset is trustworthy and accurate

Availability – guarantee of reliable access to the asset by authorized people

- Which security property is violated if someone …

– unplugs your alarm clock while you're sleeping?

– changes the time on your alarm clock?

– installs a camera in your room?

What is a software vulnerability?

- A bug in program that allows an unprivileged user capabilities that should be denied to them

What is a software vulnerability?

- A bug in program that allows an unprivileged user capabilities that should be denied to them

- There are many types of vulnerabilities

- Today: bugs that violate "control *flow* integrity"

  - Why? This lets an attacker run code on your computer!

What is a software vulnerability?

- A bug in program that allows an unprivileged user capabilities that should be denied to them

- There are many types of vulnerabilities
- Today: bugs that violate "control *fl*ow integrity"
  - Why? This lets an attacker run code on your computer!

Typically these involve violating *assumptions* of the programming language or its runtime

Control Flow Integrity

"security mechanism that disallows changes to the original control flow graph of a compiled binary, making it significantly harder to perform such attacks"

A defense strategy that protects the direction of execution in a program

Exploiting Vulnerabilities (the start)

- Dive into low-level details of how exploits work
  - How can a remote attacker get a victim program to execute their code?
- Threat model: Victim code is handling input that comes from across a security boundary
  - What are some examples of this?
- Security policy: Want to protect integrity of execution and confidentiality of data from being compromised by malicious and highly skilled users of our system.

Today: Stack buffer overflows

Lecture objectives:

- Understand how buffer overflow vulns can be exploited
- Identify buffer overflow and assess their impact
- Avoid introducing buffer overflow vulnerabilities
- Correctly fix buffer overflow activities

X86 Review

Another Linux Kernel Bug Surfaces, Allowing Root Access

Author:
Tara Seals
September 28, 2018
/ 2:11 pm

October 7, 2019

Kernel privilege escalati
exploited in Android d

Bradley Barth
Follow @bbb1216bbb

THANKS!

Low Level Software Security

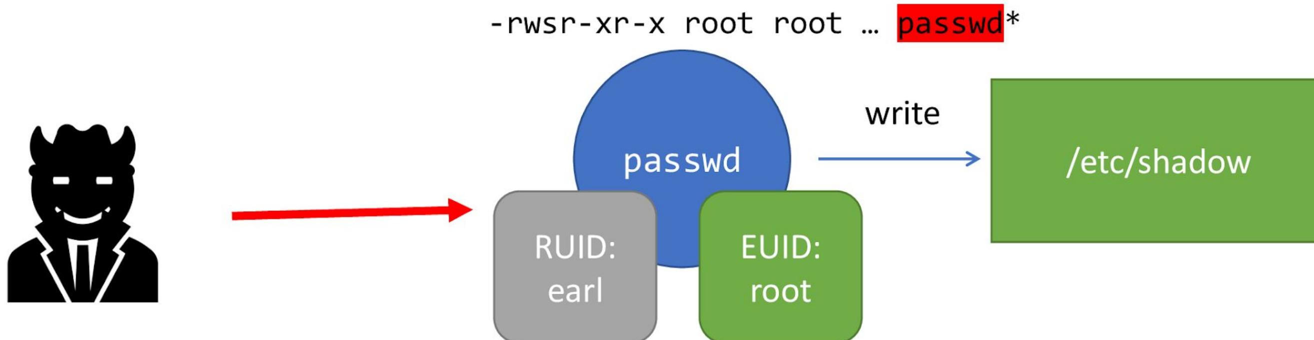
Introduction to Security
(CSE 127)

Earlence Fernandes
efernandes@ucsd.edu

* Slides borrowed/adapted from R. Chatterjee, D. Davidson, T. Ristenpart

Processes are the front line of system security

- Control a process and you get the privileges of its UID
- So how do you control a process?
  - Send specially formed input to process



X86: The De Facto Standard

- Extremely popular for desktop computers
- Alternatives
  - ARM: popular on mobile
  - MIPS: very simple
  - Itanium: ahead of its time
- CISC (complex instruction set computing)
  - Over 100 distinct opcodes in the set
- Register poor
  - Only 8 registers of 32-bits, only 6 are general-purpose
- Variable-length instructions
- Built of many backwards-compatible revisions
  - Many security problems preventable… in hindsight

Why do we need to look at assembly?

WYSINWYX: What You See Is Not What You eXecute

G. Balakrishnan[1], T. Reps[1,2], D. Melski[2], and T. Teitelbaum[2]

[1] Comp. Sci. Dept., University of Wisconsin; {bgogul,reps}@cs.wisc.edu
[2] GrammaTech, Inc.; {melski,tt}@grammatech.com

We understand code in this form

```
int foo(){
        int a = 0;
        return a + 7;
}
```

Vulnerabilities exploited in this form

```
pushl %ebp
movl  %esp, %ebp
subl  $16, %esp
movl  $0, -4(%ebp)
movl  -4(%ebp), %eax
addl  $7, %eax
leave
ret
```

Tools: GCC

```
gcc –O0 –S program.c –o program.S –m32
```

Generate Assembly Code

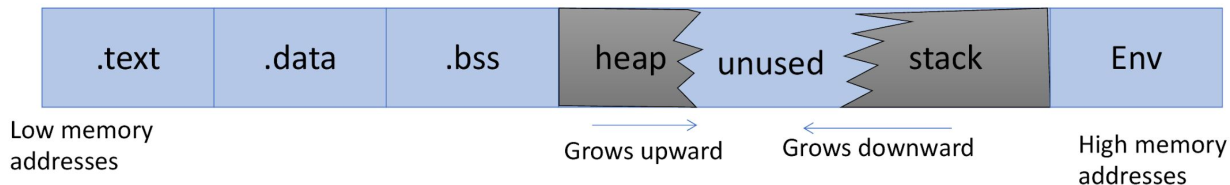
```
gcc –O0 –g program.c –o program –m32
```

Generate Debugging Information

Tools: GDB

```
gdb program
(gdb) run
(gdb) list   /* show the high-level code */
(gdb)
(gdb) disas foo  /* show assembly of foo */
(gdb) disas main
(gdb) quit
```

Process memory layout



.text
- Machine code of executable

.data
- Global initialized variables

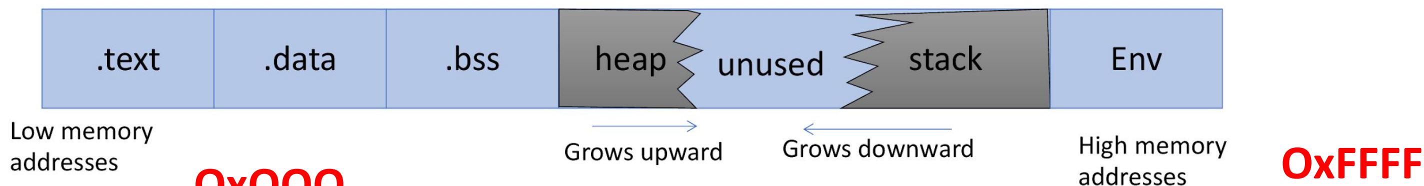
.bss
- Below Stack Section
  global uninitialized variables

heap
- Dynamic variables

stack
- Local variables
- Function call data

Env
- Environment variables
- Program arguments

Process memory layout



OxOOO

OxFFFF

.text

- Machine code of executable

.data

- Global initialized variables

.bss

- Below Stack Section
  global uninitialized variables

heap

- Dynamic variables

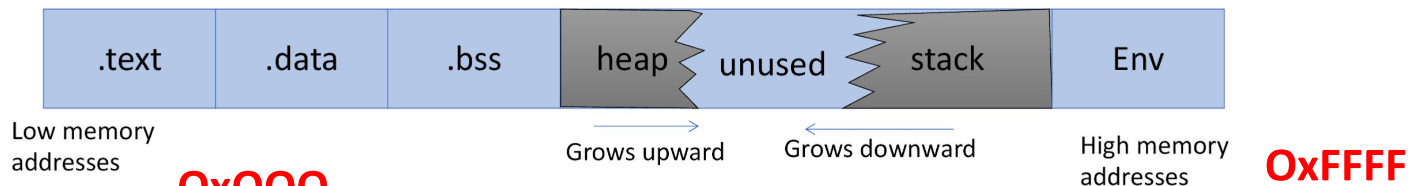
stack

- Local variables
- Function call data

Env

- Environment variables
- Program arguments

Process memory layout



OxOOO

OxFFFF

.text

- Machine code of executable

.data

- Global initialized variables

.bss

- Below Stack Section
  global uninitialized variables

heap

- Dynamic variables

stack

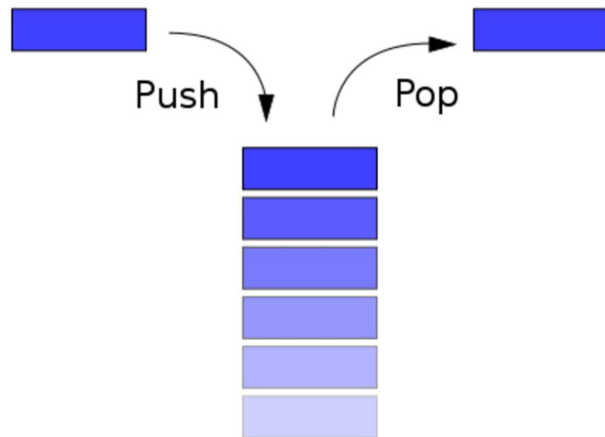
- Local variables
- Function call data

Env

- Environment variables
- Program arguments

The Stack

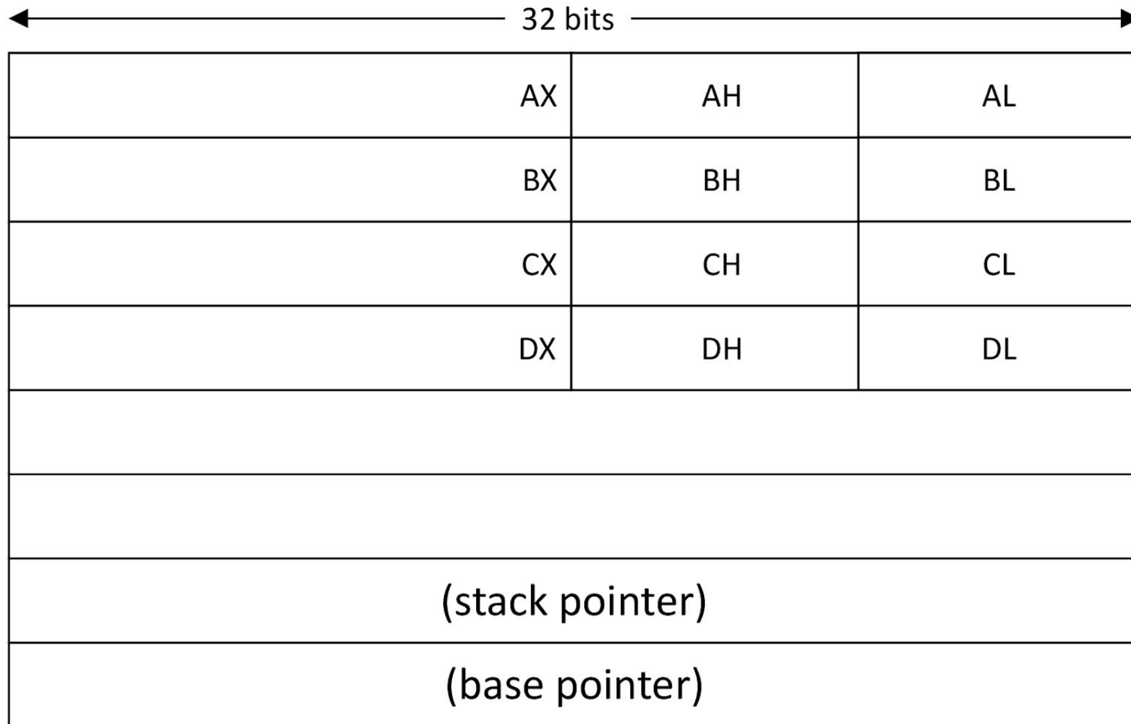
- Local storage
  - Good place to keep data that doesn't fit into registers
- Grows from high addresses towards low addresses



Reminder: These are conventions

- Dictated by compiler
- Only instruction support by processor
  - Almost no structural notion of memory safety
    - Use of uninitialized memory
    - Use of freed memory
    - Memory leaks
- So how are they actually implemented?

Registers

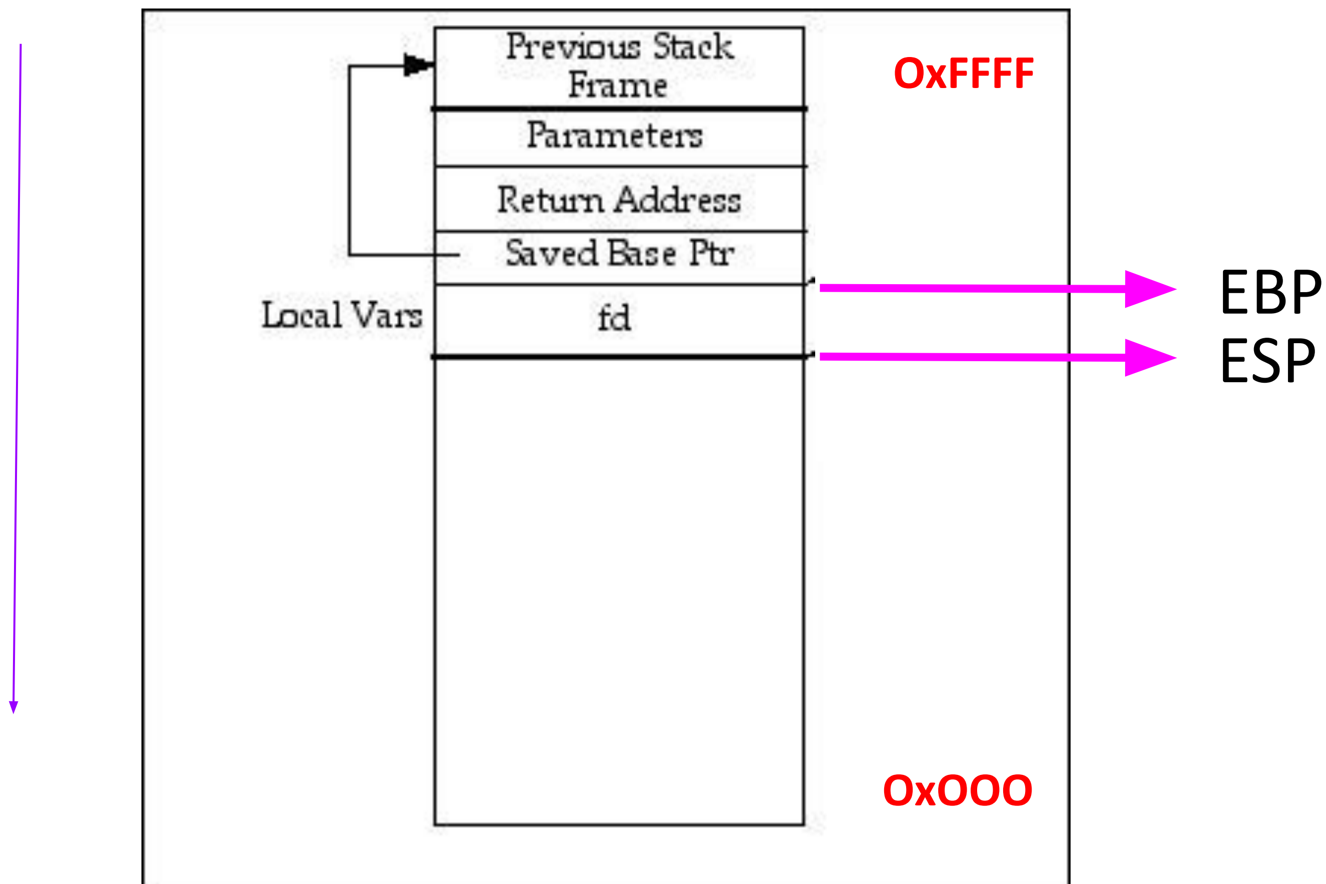


Registers

- **eip** (instruction pointer) - address of current machine instruction
- **ebp** (base pointer) - address of the top of the current stack frame.
- **esp** (stack pointer) - address of the bottom of the current stack frame.

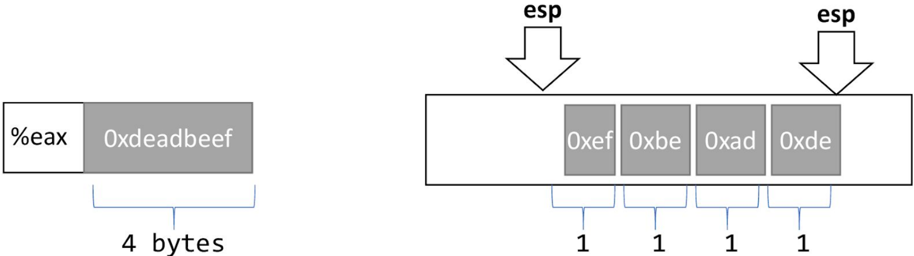
Registers

Stack
Growth



Data types / Endianness

- x86 is a little-endian architecture



Instruction Syntax

Examples:

```
subl $16, %ebx

movl (%eax), %ebx
```

```
opcode src, dst
```

- Constants preceded by $
- Registers preceded by %
- Indirection uses ( )

Instruction Syntax

Examples:

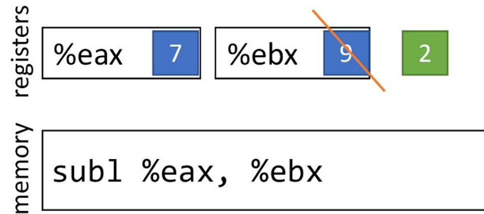
```
subl $16, %ebx

movl (%eax), %ebx
```

```
opcode src, dst
```

- Constants preceded by $
- Registers preceded by %
- Indirection uses ( )

```
opcode src, dst
```

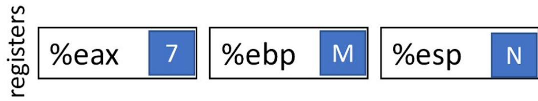
Register Instructions: `sub`

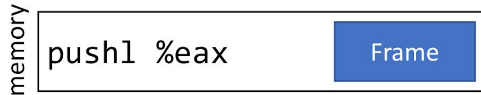


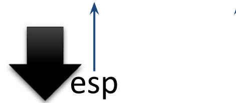
- Subtract from a register value

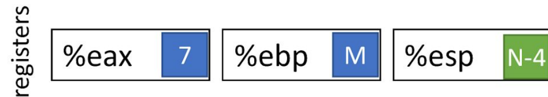
opcode dst

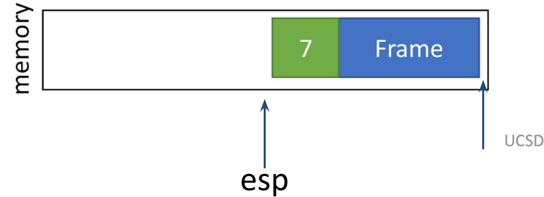
Frame Instructions: **push**









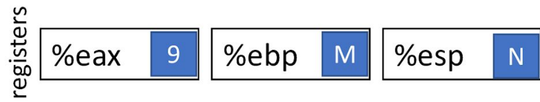


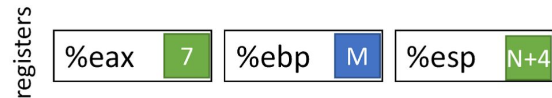
- Put a value on the stack
  - Pull from register
  - Value goes to `%esp`
  - Subtract from `%esp`
- Example:

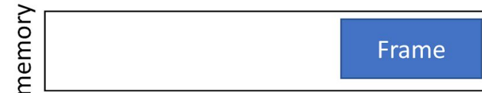
`pushl %eax`

opcode dst

Frame Instructions: **pop**

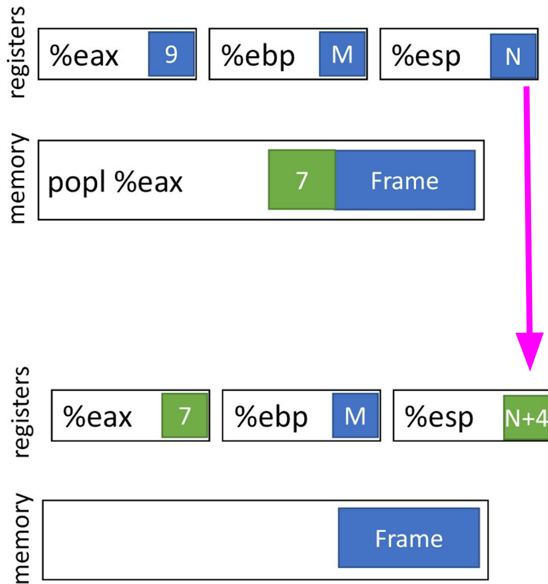






- Take a value from the stack
  - Pull from stack pointer
  - Value goes from %esp
  - Add to %esp

Frame Instructions: **pop**

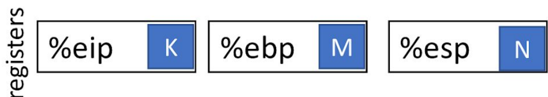


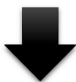
- Take a value from the stack
  - Pull from stack pointer
  - Value goes from %esp
  - Add to %esp

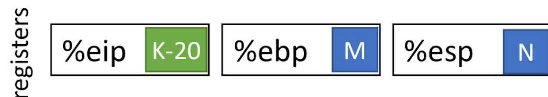
Why? - Think about where esp is located and how the stack grows

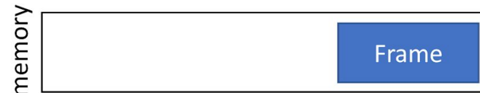
opcode dst

Control flow instructions: `jmp`



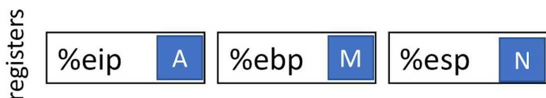


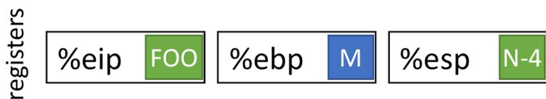




- `%eip` points to the currently executing instruction (in the text section)
- Has unconditional and conditional forms
- Uses relative addressing

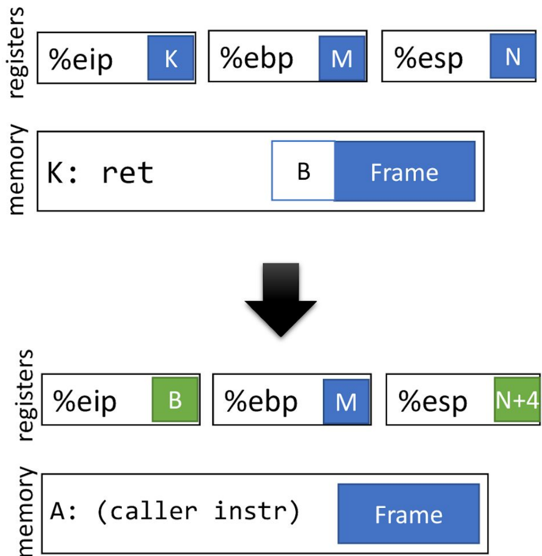
Control flow instructions: `call`





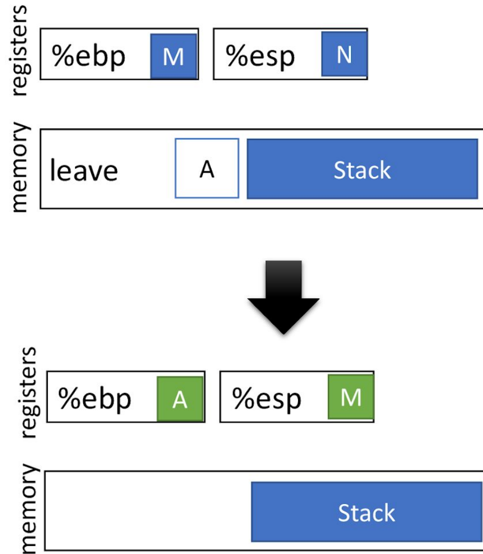
- Saves the instruction pointer to the stack
- Jumps to the argument value

Control flow instructions: `ret`



- Pops the stack into the instruction pointer

Stack instructions: **`leave`** (and **`enter`**)



- Equivalent to

```
movl %ebp, %esp
popl %ebp
```

- copy EBP to ESP and then restore the old EBP from the stack

registers:

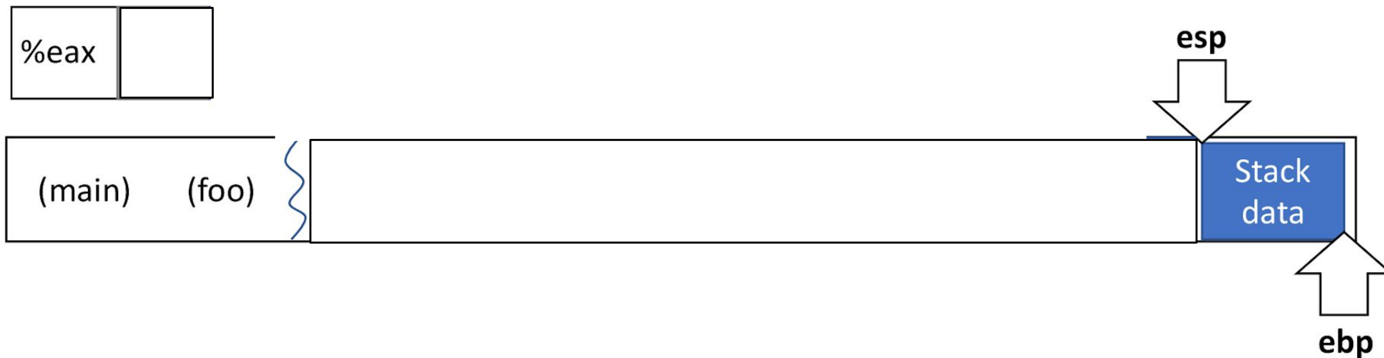
Address	Value	Register	Value
0x100	0xFF	%eax	0x100
0x104	0xAB	%ecx	0x1
0x108	0x13	%edx	0x3
0x10C	0x11		

Fill in the following table showing the values for the indicated operands:

This exercise gives you practice with the different operand forms.

Operand	Value	Comment
%eax	0x100	Register
0x104	0xAB	Absolute address
$0x108	0x108	Immediate
(%eax)	0xFF	Address 0x100
4(%eax)	0xAB	Address 0x104
9 (%eax, %ecx)	0x11	Address 0x10C

Implementing a function call

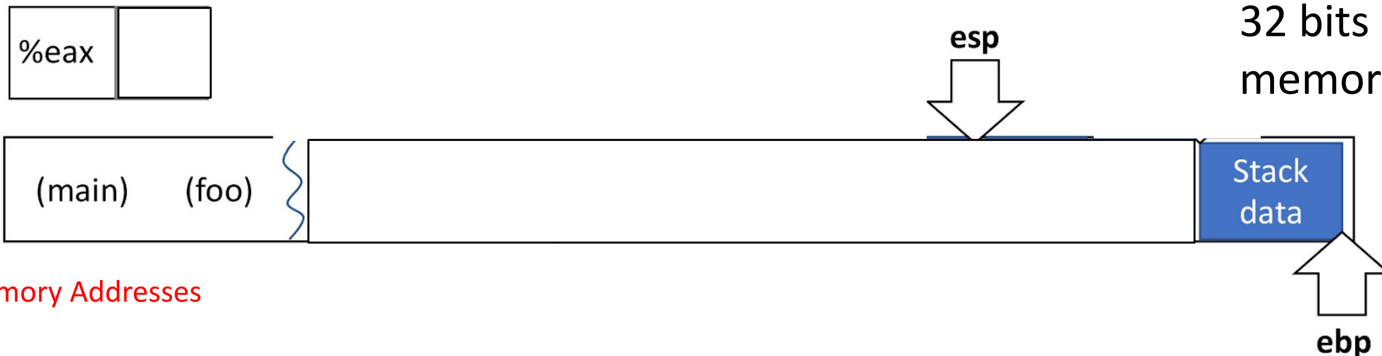



```
main:
    …
    subl    $8, %esp
    movl    $2, 4(%esp)
    movl    $1, (%esp)
    call    foo
    addl    $8, %esp
    …
```

```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    movl    $3, -4(%ebp)
    movl    8(%ebp), %eax
    addl    $9, %eax
    leave
    ret
```

Implementing a function call

1 byte = 8 bits
32 bits = size of memory address



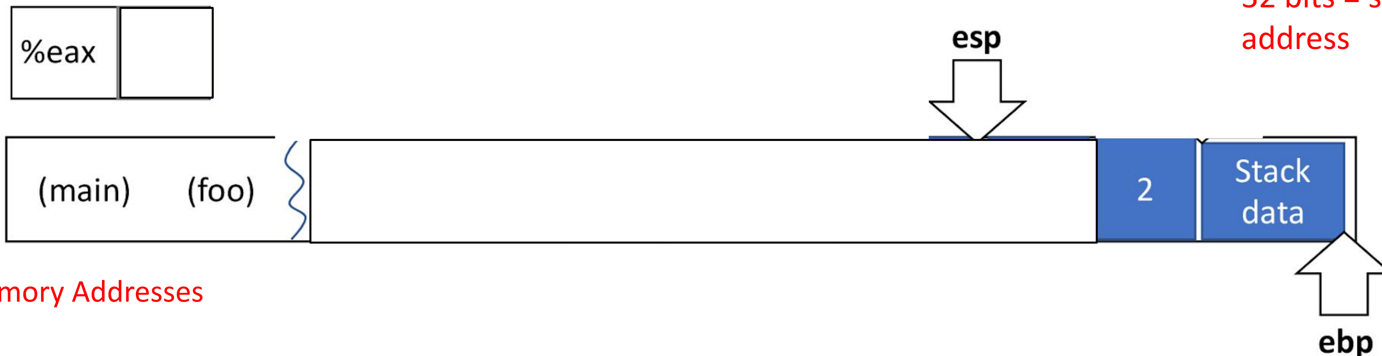
Allocate space on the stack

```
main:
    …
    subl    $8, %esp
    movl    $2, 4(%esp)
    movl    $1, (%esp)
    call    foo
    addl    $8, %esp
    …
```

```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    movl    $3, -4(%ebp)
    movl    8(%ebp), %eax
    addl    $9, %eax
    leave
    ret
```

Implementing a function call

1 byte = 8 bits
32 bits = size of memory address



Low Memory Addresses

High Memory Addresses

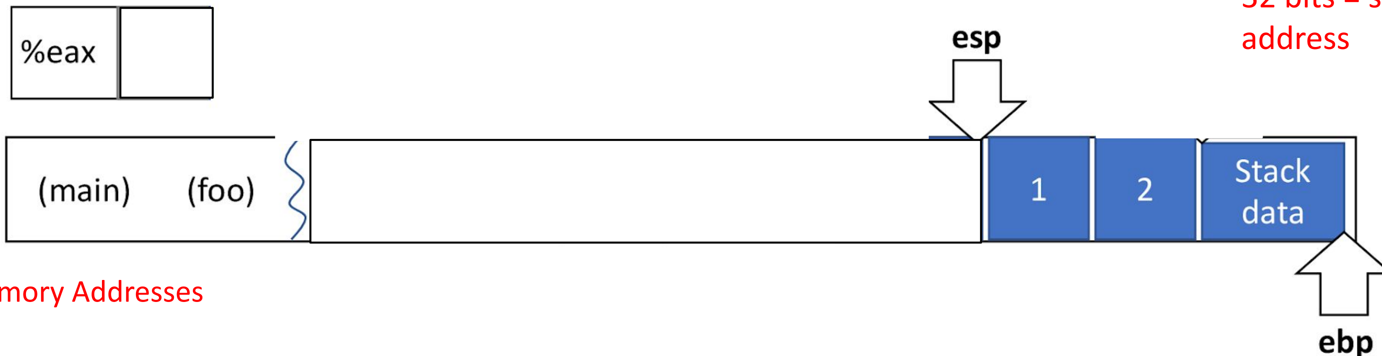
Dereferences memory 4 bytes above %esp

```
main:
    …
    subl    $8, %esp
    movl    $2, 4(%esp)
    movl    $1, (%esp)
    call    foo
    addl    $8, %esp
    …
```

```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    movl    $3, -4(%ebp)
    movl    8(%ebp), %eax
    addl    $9, %eax
    leave
    ret
```

Implementing a function call

1 byte = 8 bits
32 bits = size of memory address



Low Memory Addresses

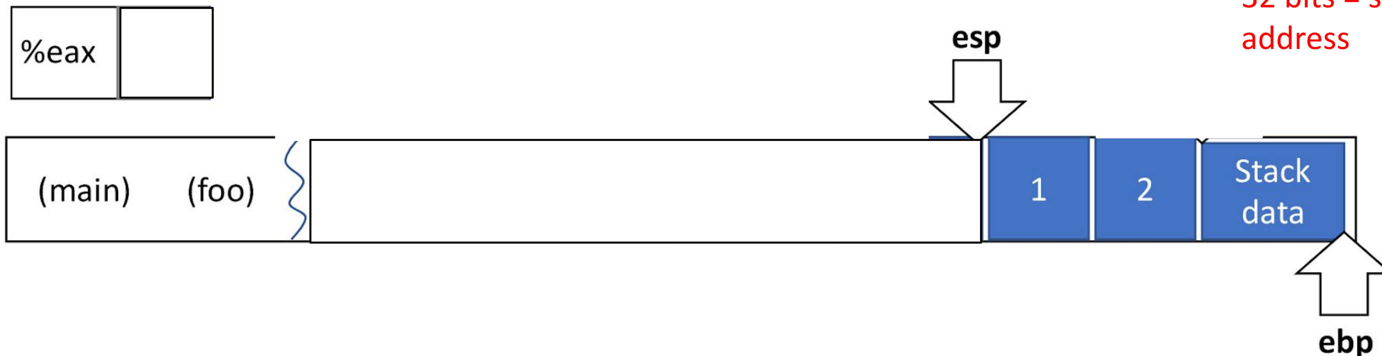
High Memory Addresses

```
main:
    …
    subl    $8, %esp
    movl    $2, 4(%esp)
eip → movl  $1, (%esp)
    call    foo
    addl    $8, %esp
    …
```

```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    movl    $3, -4(%ebp)
    movl    8(%ebp), %eax
    addl    $9, %eax
    leave
    ret
```

Implementing a function call

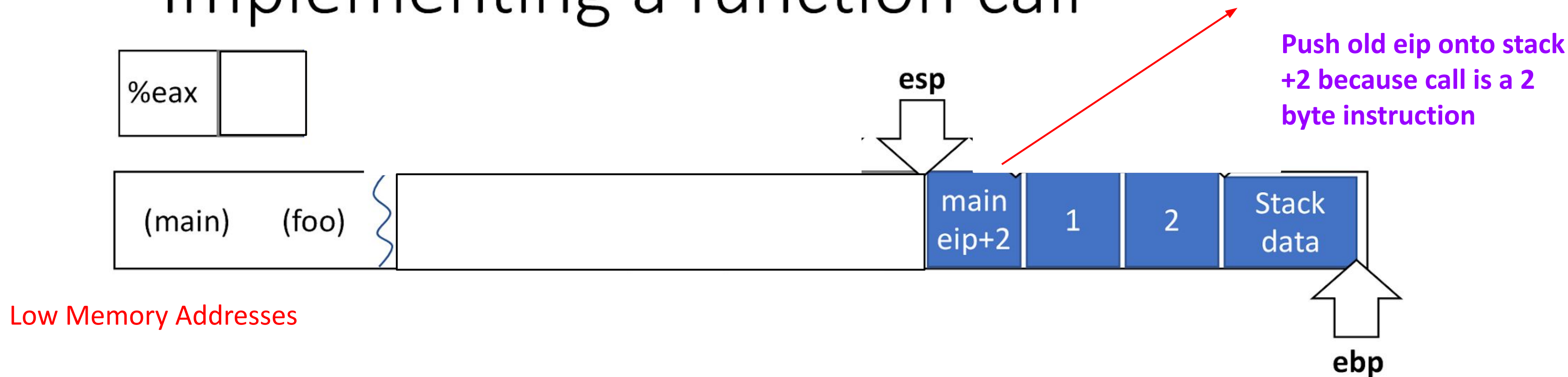
1 byte = 8 bits
32 bits = size of memory address




```
main:
    …
    subl    $8, %esp
    movl    $2, 4(%esp)
    movl    $1, (%esp)
eip → call    foo
    addl    $8, %esp
    …
```

```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    movl    $3, -4(%ebp)
    movl    8(%ebp), %eax
    addl    $9, %eax
    leave
    ret
```

Implementing a function call

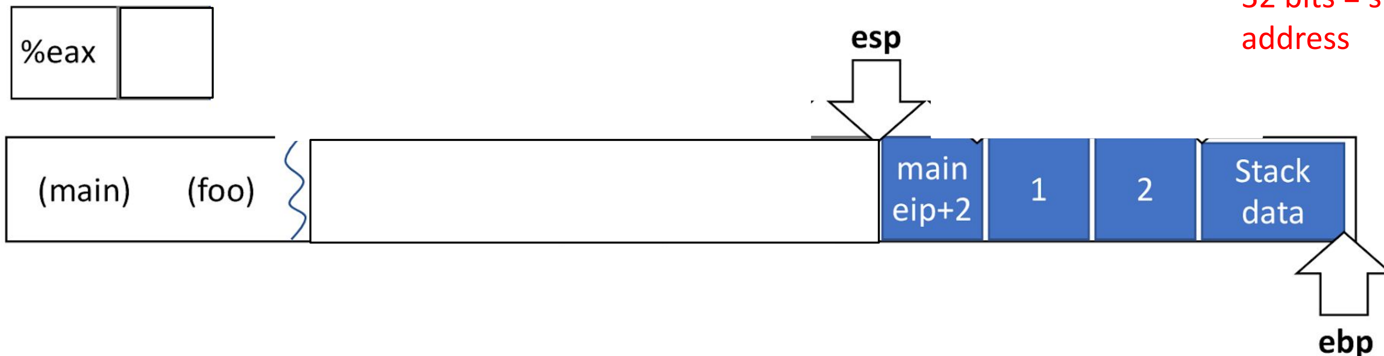



```
main:
    …
    subl    $8, %esp
    movl    $2, 4(%esp)
    movl    $1, (%esp)
eip → call    foo
    addl    $8, %esp
    …
```

```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    movl    $3, -4(%ebp)
    movl    8(%ebp), %eax
    addl    $9, %eax
    leave
    ret
```

Implementing a function call

1 byte = 8 bits
32 bits = size of memory address



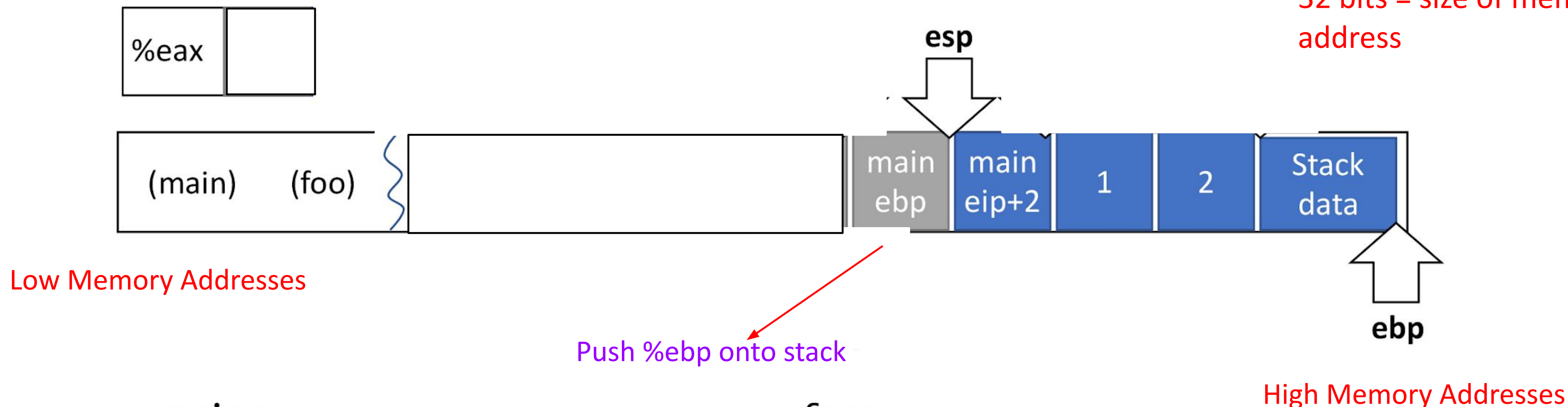

```
main:
    …
    subl    $8, %esp
    movl    $2, 4(%esp)
    movl    $1, (%esp)
    call    foo
    addl    $8, %esp
    …
```

eip →

```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    movl    $3, -4(%ebp)
    movl    8(%ebp), %eax
    addl    $9, %eax
    leave
    ret
```

Implementing a function call

1 byte = 8 bits
32 bits = size of memory address



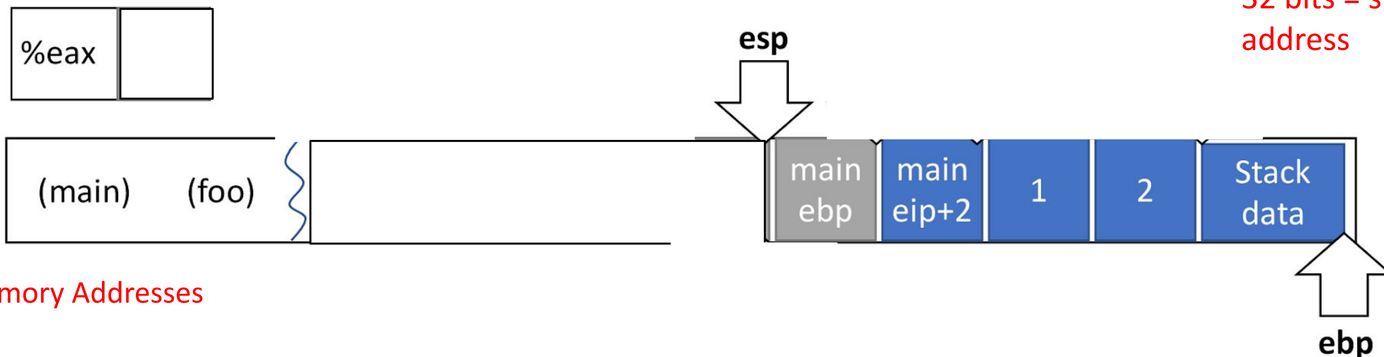

```
main:
    …
    subl    $8, %esp
    movl    $2, 4(%esp)
    movl    $1, (%esp)
    call    foo
    addl    $8, %esp
    …
```

eip →

```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    movl    $3, -4(%ebp)
    movl    8(%ebp), %eax
    addl    $9, %eax
    leave
    ret
```

Implementing a function call

1 byte = 8 bits
32 bits = size of memory address



Low Memory Addresses

High Memory Addresses

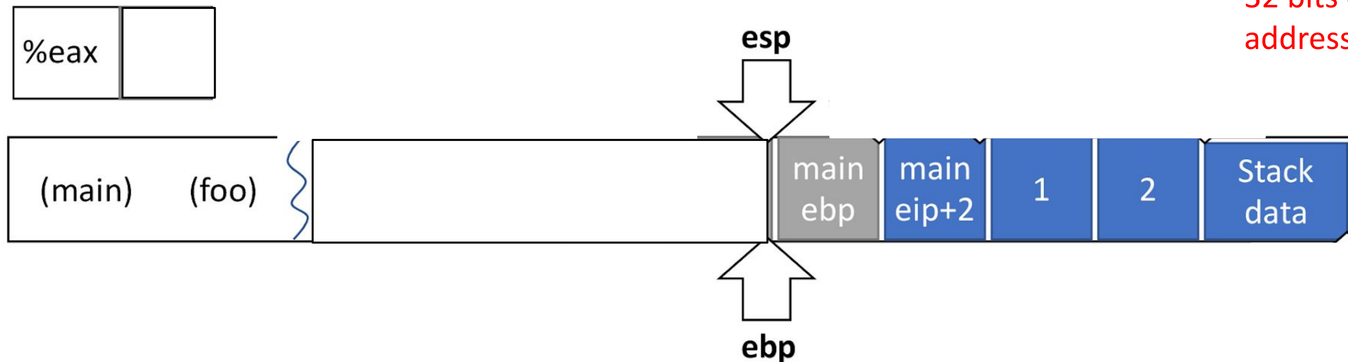
```
main:
    …
    subl    $8, %esp
    movl    $2, 4(%esp)
    movl    $1, (%esp)
    call    foo
    addl    $8, %esp
    …
```

eip →

```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    movl    $3, -4(%ebp)
    movl    8(%ebp), %eax
    addl    $9, %eax
    leave
    ret
```

Implementing a function call

1 byte = 8 bits
32 bits = size of memory address

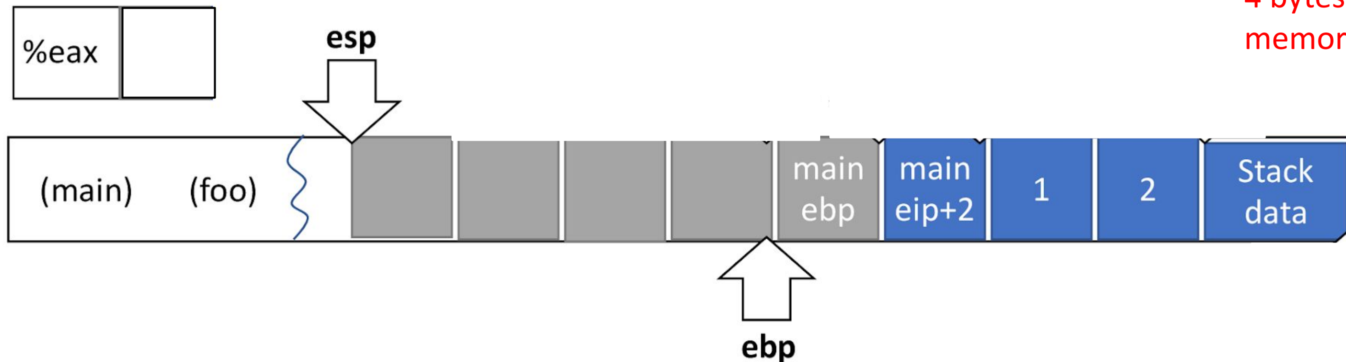



```
main:
    …
    subl    $8, %esp
    movl    $2, 4(%esp)
    movl    $1, (%esp)
    call    foo
    addl    $8, %esp
    …
```

```
foo:
    pushl   %ebp
    movl    %esp, %ebp      <- eip
    subl    $16, %esp
    movl    $3, -4(%ebp)
    movl    8(%ebp), %eax
    addl    $9, %eax
    leave
    ret
```

Implementing a function call

1 byte = 8 bits
4 bytes = 32 bits = size of memory address

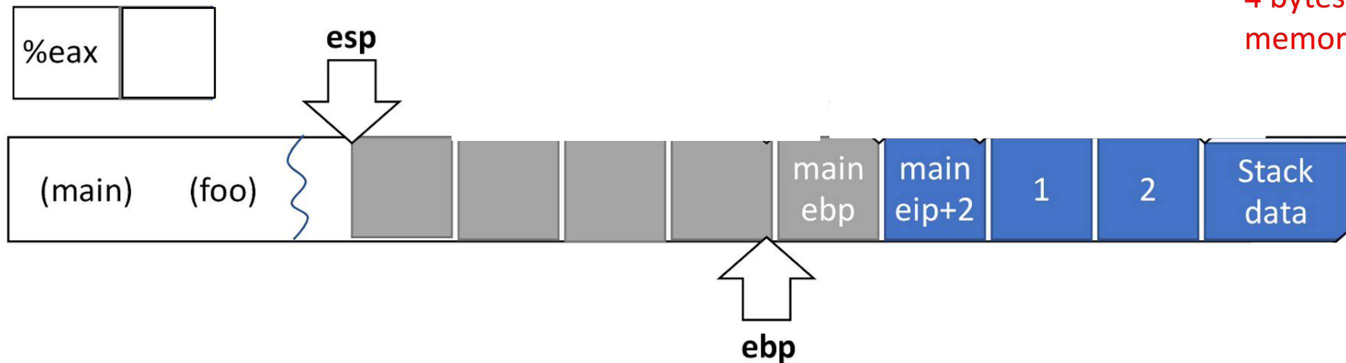



```
main:
      …
      subl    $8, %esp
      movl    $2, 4(%esp)
      movl    $1, (%esp)
      call    foo
      addl    $8, %esp
      …
```

```
foo:
      pushl   %ebp
      movl    %esp, %ebp
eip → subl    $16, %esp
      movl    $3, -4(%ebp)
      movl    8(%ebp), %eax
      addl    $9, %eax
      leave
      ret
```

Implementing a function call

1 byte = 8 bits
4 bytes = 32 bits = size of memory address

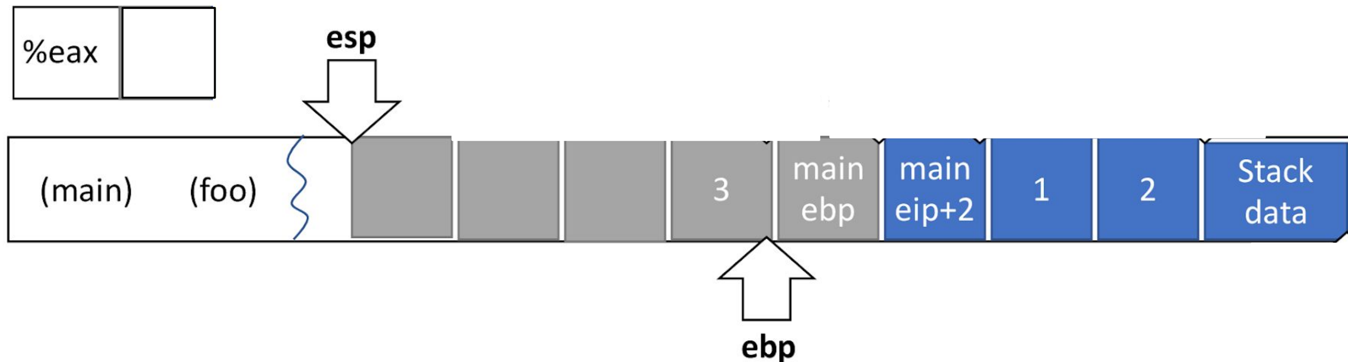



```
main:
    …
    subl    $8, %esp
    movl    $2, 4(%esp)
    movl    $1, (%esp)
    call    foo
    addl    $8, %esp
    …
```

Allocate 16 bytes on the stack

```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    movl    $3, -4(%ebp)
    movl    8(%ebp), %eax
    addl    $9, %eax
    leave
    ret
```

Implementing a function call



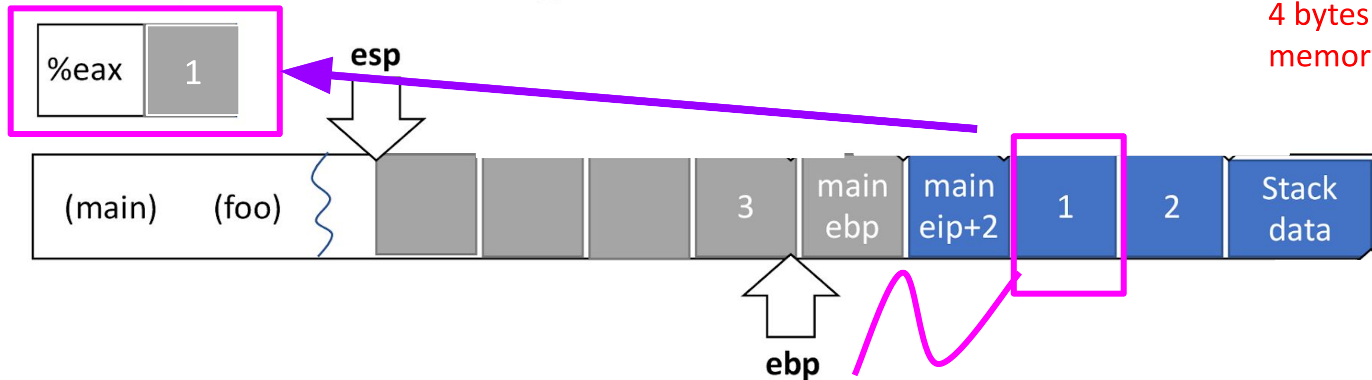
Move 3 in address -4 bytes from %ebp

```
main:
    …
    subl    $8, %esp
    movl    $2, 4(%esp)
    movl    $1, (%esp)
    call    foo
    addl    $8, %esp
    …
```

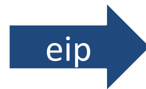
```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    movl    $3, -4(%ebp)
    movl    8(%ebp), %eax
    addl    $9, %eax
    leave
    ret
```

Implementing a function call

1 byte = 8 bits
4 bytes = 32 bits = size of memory address

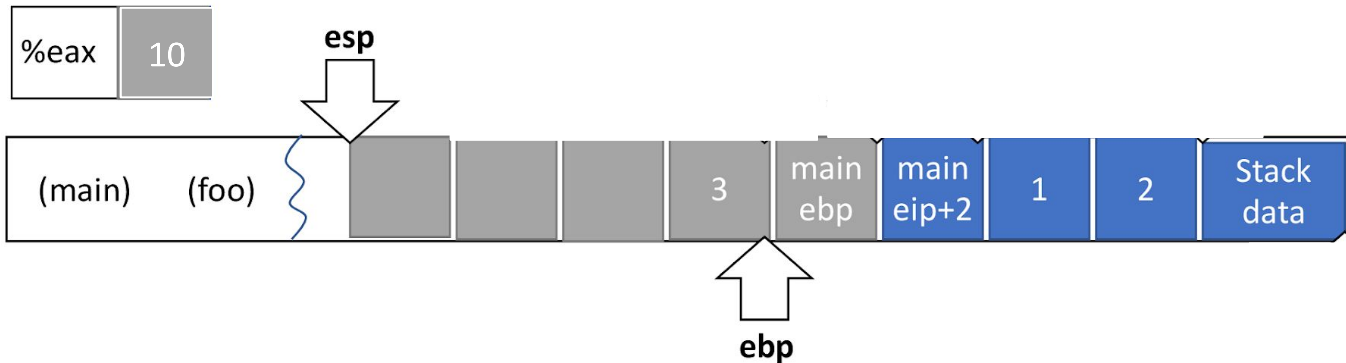



```
main:
    …
    subl    $8, %esp
    movl    $2, 4(%esp)
    movl    $1, (%esp)
    call    foo
    addl    $8, %esp
    …
```




```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    movl    $3, -4(%ebp)
    movl    8(%ebp), %eax
    addl    $9, %eax
    leave
    ret
```

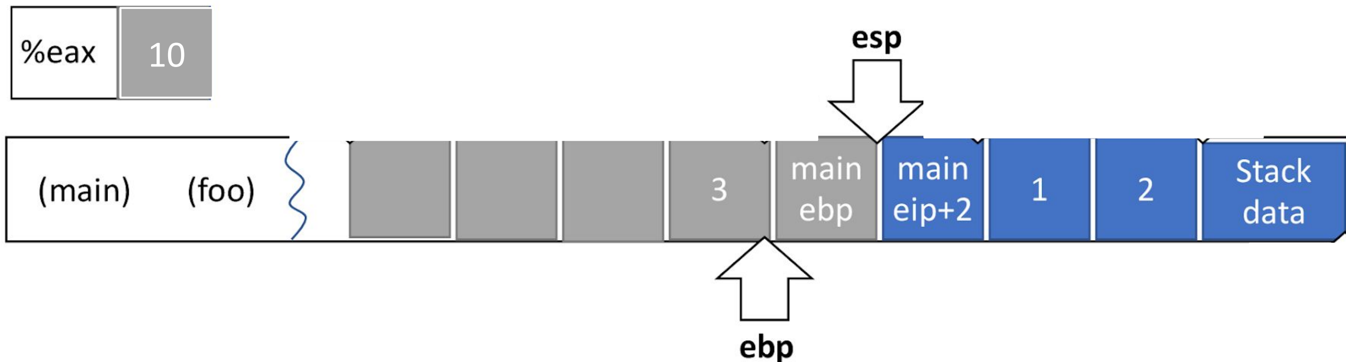
Implementing a function call




```
main:
    …
    subl    $8, %esp
    movl    $2, 4(%esp)
    movl    $1, (%esp)
    call    foo
    addl    $8, %esp
    …
```

```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    movl    $3, -4(%ebp)
    movl    8(%ebp), %eax
    addl    $9, %eax
    leave
    ret
```

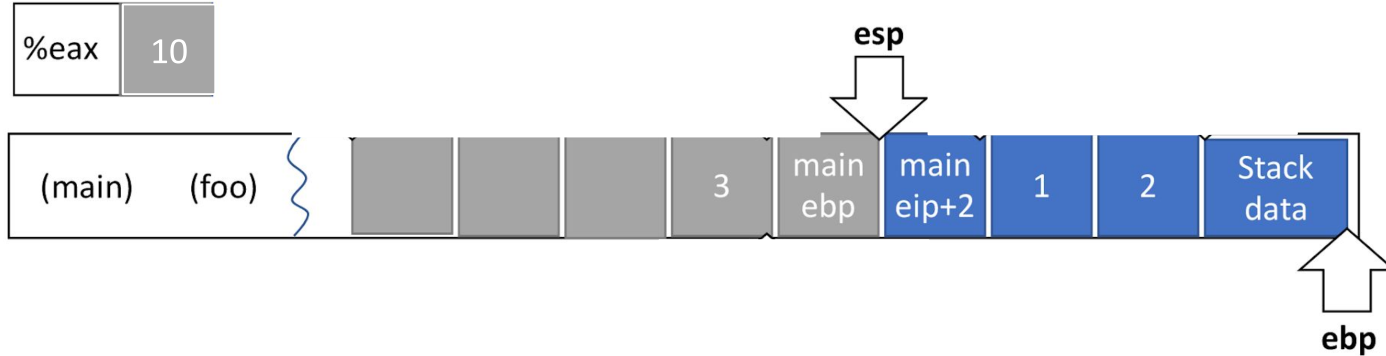
Implementing a function call




```
main:
    …
    subl    $8, %esp
    movl    $2, 4(%esp)
    movl    $1, (%esp)
    call    foo
    addl    $8, %esp
    …
```

```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    movl    $3, -4(%ebp)
    movl    8(%ebp), %eax
    addl    $9, %eax
    leave
    ret
```

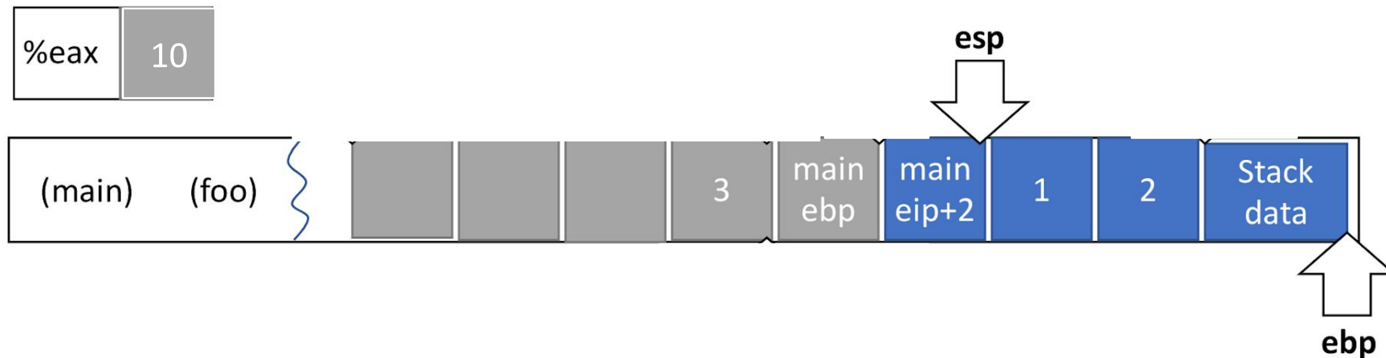
Implementing a function call




```
main:
    …
    subl    $8, %esp
    movl    $2, 4(%esp)
    movl    $1, (%esp)
    call    foo
    addl    $8, %esp
    …
```

```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    movl    $3, -4(%ebp)
    movl    8(%ebp), %eax
    addl    $9, %eax
    leave
    ret
```

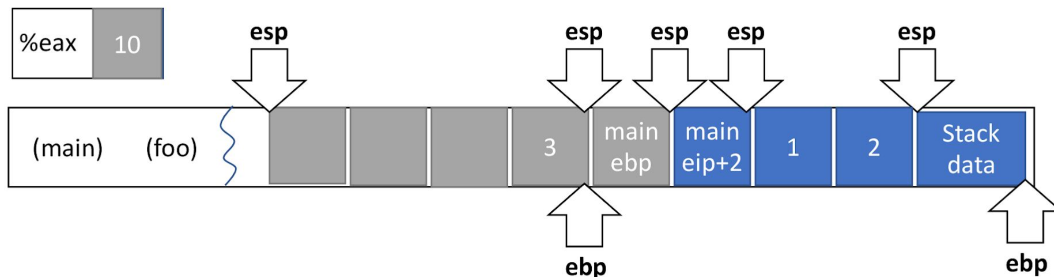
Implementing a function call




```
main:
    …
    subl    $8, %esp
    movl    $2, 4(%esp)
    movl    $1, (%esp)
    call    foo
    addl    $8, %esp
    …
```

```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    movl    $3, -4(%ebp)
    movl    8(%ebp), %eax
    addl    $9, %eax
    leave
    ret
```

Implementing a function call




```
main:
    …
    subl    $8, %esp
    movl    $2, 4(%esp)
    movl    $1, (%esp)
    call    foo
    addl    $8, %esp
    …
```

```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    movl    $3, -4(%ebp)
    movl    8(%ebp), %eax
    addl    $9, %eax
    leave
    ret
```

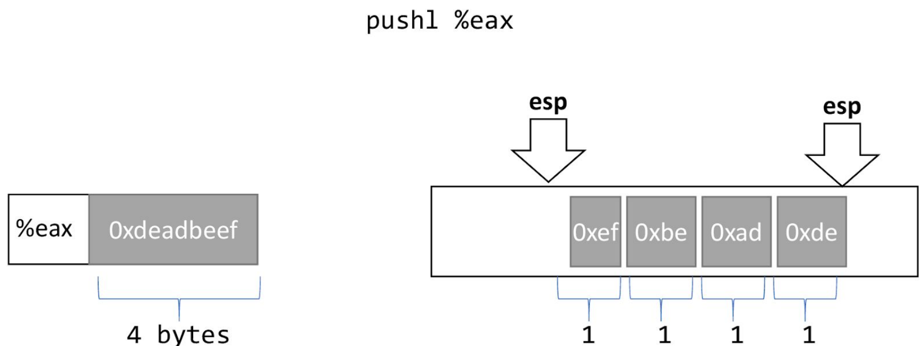
Function Calls: High level points

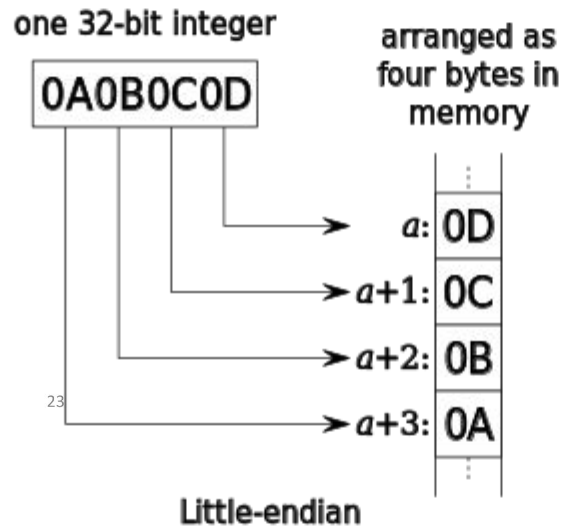
- Locals are organized into stack frames
  - Callees exist at lower address than the caller
- On call:
  - Save `%eip` so you can restore control
  - Save `%ebp` so you can restore data
- Implementation details are largely by convention
  - Somewhat codified by hardware

Data types / Endianness

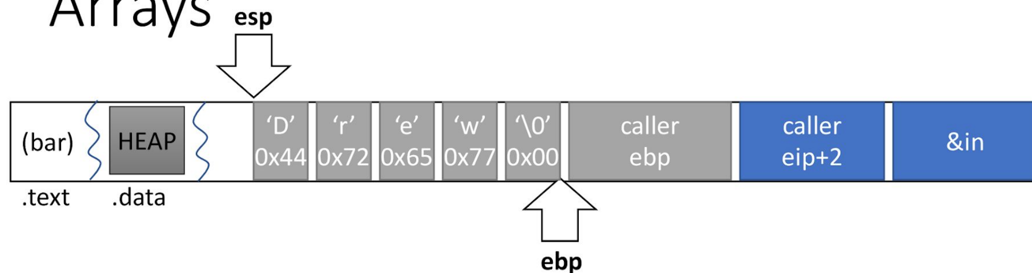
- x86 is a little-endian architecture

Data is stored in memory with the least significant byte in the lowest-numbered address. That is, the “littlest” byte (in the sense of significance) comes first in memory.





Arrays




```
void bar(char * in){
  char name[5];
  strcpy(name, in);
}
```

```
bar:
  pushl   %ebp
  movl    %esp, %ebp
  subl    $5, %esp
  movl    8(%ebp), %eax
  movl    %eax, 4(%esp)
  leal    -5(%ebp), %eax
  movl    %eax, (%esp)
  call    strcpy
  leave
  ret
```

Next time on CSE 127…

Buffer overflow explanation and review