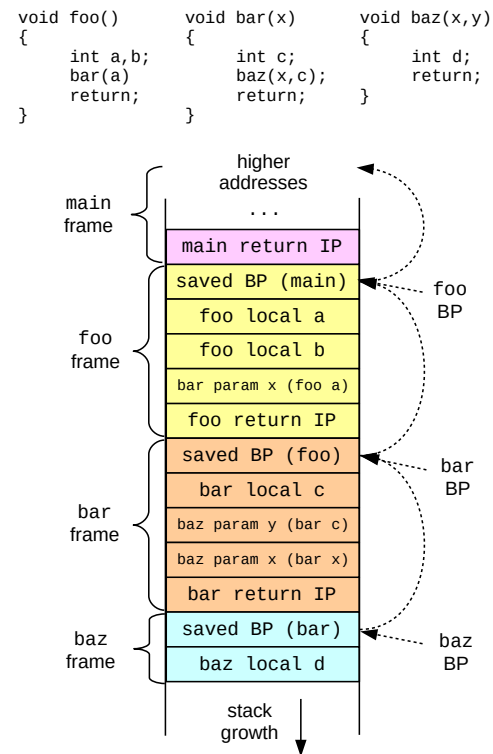


# CS 261 Fall 2021

Mike Lam, Professor



## x86-64 Procedures

# Topics

- ABIs, the runtime stack, and control transfer
- Data transfer and local storage
- Security issues

# Procedure calls

- A **procedure** is a portion of code packaged for re-use
  - Key abstraction in software development
  - Provide **modularity** and **encapsulation**
  - Many alternative names: functions, methods, subroutines
- Well-designed procedures have:
  - Well-documented, typed arguments and return value(s)
  - Clear impact on program state (or no impact)
    - Also known as “side effects”

# Problem

- Impossible to implement procedures in assembly with branches or jumps alone
  - Once you've jumped, how do you return?
  - Can hard-code for one call site, but not for 2+
- Need a mechanism for “remembering” where we came from
  - And any machine state important for getting back
  - Don't want to use registers because there are so few
  - Solution: use memory! **(but how and where?)**

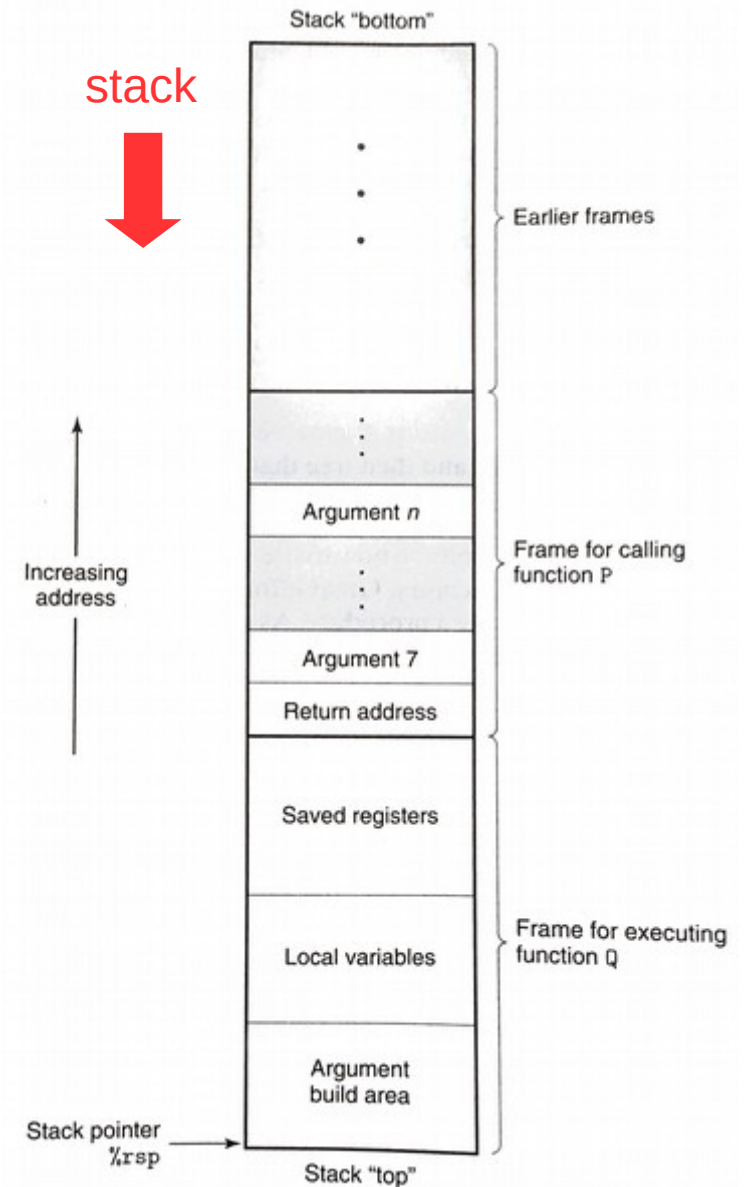
# ABI

- **Application Binary Interface (ABI)**
  - Interface between program & system at the binary level
  - Includes rules about how procedure calls are implemented
  - These rules are referred to as **calling conventions**
  - We will study the standard x86-64 calling conventions
- Calling conventions specify:
  - Control transfer
  - Data transfer
  - Local storage

# Runtime stack

- Basic idea: maintain a system **stack frame** for each procedure call
  - All active procedure have a frame
  - Each frame stores information about a single active call
    - Arguments, local variables, return address
  - GDB's "**backtrace**" command follows the chain up
  - Recursion just works!

Here function P has called function Q

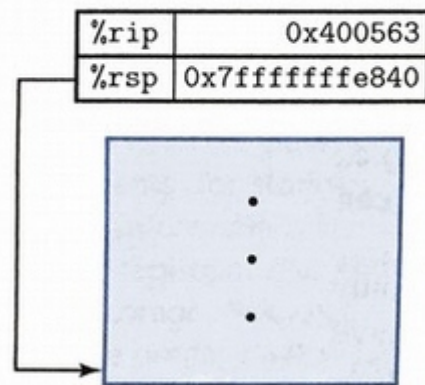


# Control transfer

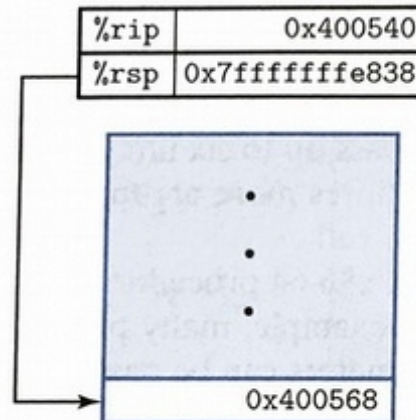
- Use stack to store return addresses
  - **Return address**: the instruction AFTER the `call`
  - `call` / `callq` pushes 64-bit return address onto stack
  - `ret` / `retq` pops the return address and sets `%rip`

```
400550 <main>:  
...  
400563  callq 400540 <foo>  
400568  movq 0x8(%rsp), %rdx  
...
```

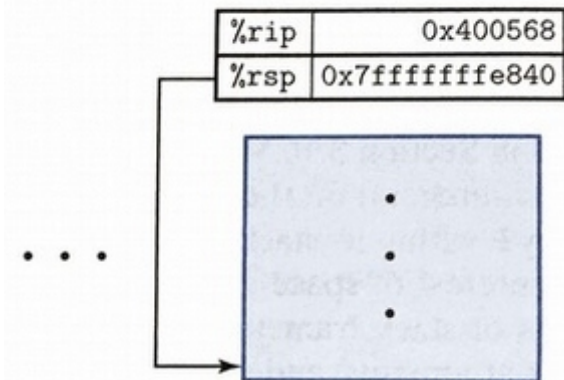
```
400540 <foo>:  
400540  xorq %rax, %rax  
...  
40054d  retq
```



(a) Executing `call`



(b) After `call`



(c) After `ret`

# Data transfer

- In x86-64, up to six **integral** (integer or pointer) **arguments** are passed to a procedure via registers:
  - %rdi, %rsi, %rdx, %rcx, %r8, %r9
  - Other arguments are passed on the stack (and pushed in reverse order)
- A single **return value** is passed back via %rax
  - Large structs often “returned” using a pointer

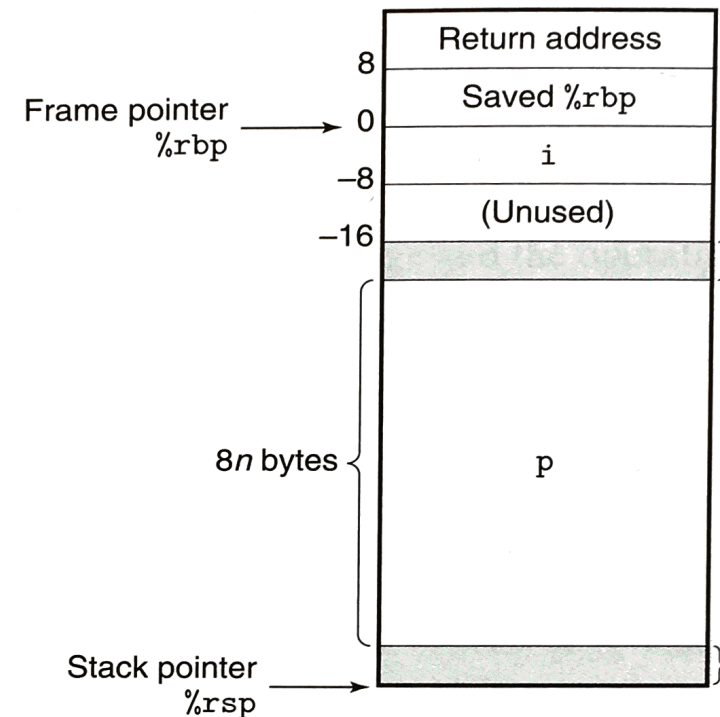


# Local storage (registers)

- Some registers are designated **callee-saved**
  - In x86-64: %rbx, %rbp, %r12, %r13, %r14, %r15
  - A procedure must save/restore these registers (often using push/pop) if they are used during the procedure
  - When possible, avoid using these registers inside procedures (lower overhead)
- Other registers (except %rsp) are **caller-saved**
  - Caller must save them if they need to be preserved
  - The stack pointer is a special case (used for communication)

# Local storage (memory)

- Procedures can allocate space on the stack for **local variables**
  - Subtract # of bytes needed from `%rsp`
  - Deallocate by restoring old `%rsp` value
- Variable-sized allocations require special handling
  - Use **base / frame pointer** (`%rbp`) to track “anchor” for current frame
  - Save previous base pointer on stack at beginning of function
  - Section 3.10.5 in textbook

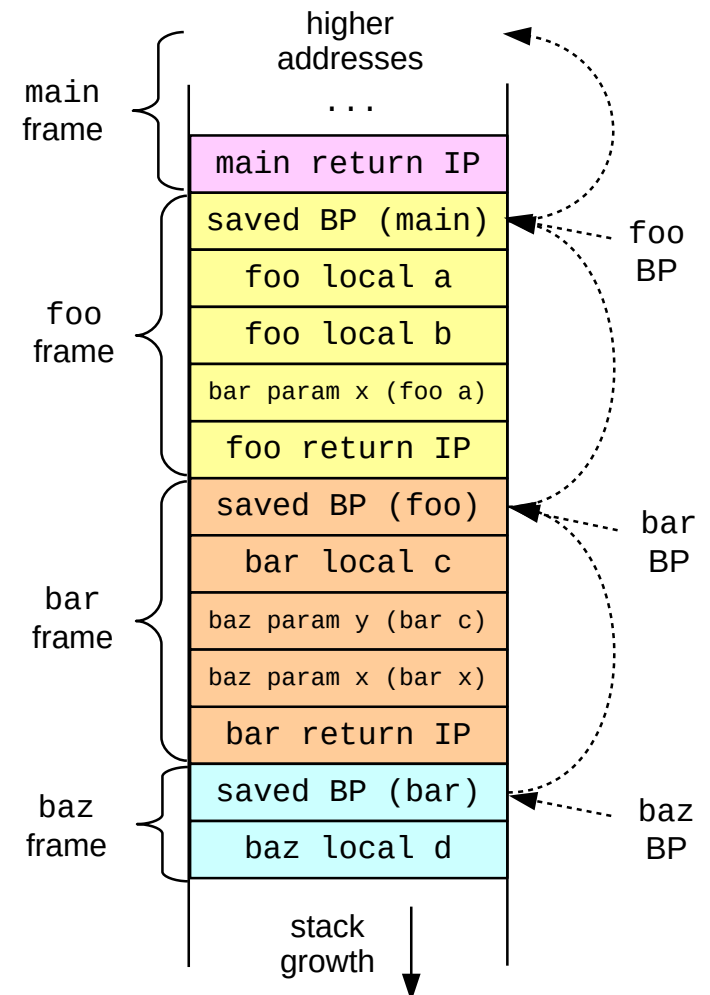
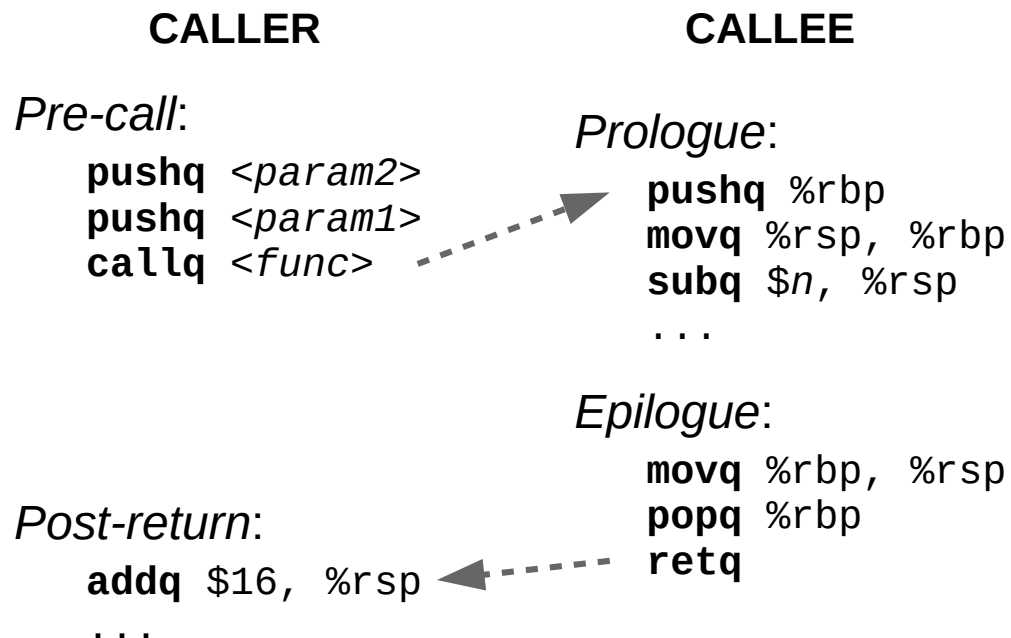


# Base pointers

## CS 430/432 preview

- Use **base pointer** (%rbp) to track the beginning of current frame
  - Parameters at positive offsets
  - Local values at negative offsets
  - Chain of base pointers up the stack
  - Push/pop BP like return address

```
void foo()  
{  
    int a,b;  
    bar(a);  
    return;  
}  
  
void bar(x)  
{  
    int c;  
    baz(x,c);  
    return;  
}  
  
void baz(x,y)  
{  
    int d;  
    return;  
}
```



# Question

- What is the security problem with the following C function?

```
void echo () {  
    char buf[8];  
    gets(buf);  
    printf(buf);  
}
```

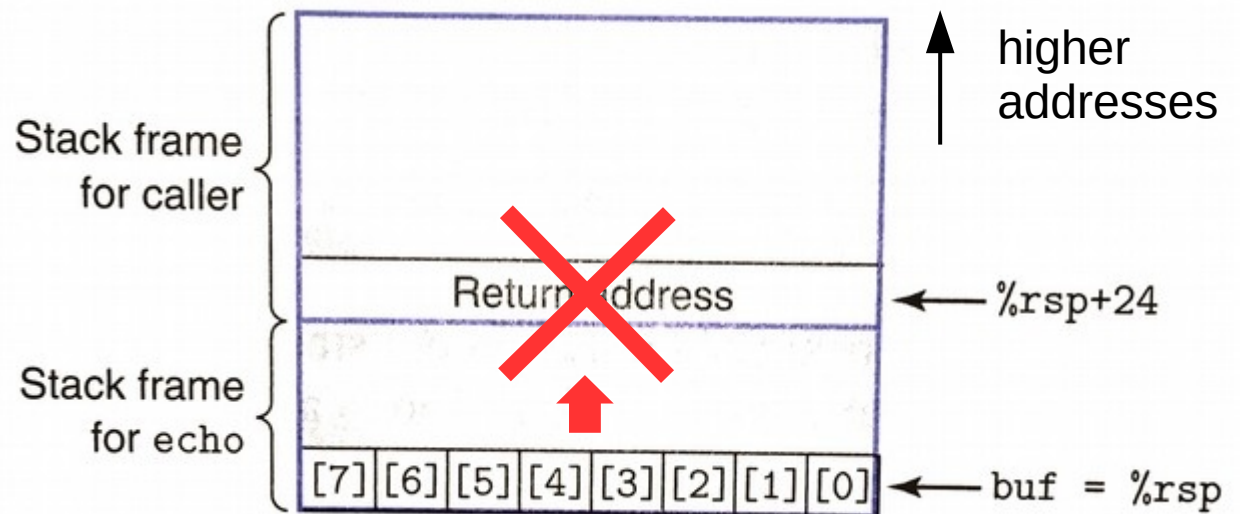
- A) It reads from an unspecified file stream
- B) It writes to standard output
- C) It can write to memory in echo's stack frame
- D) It can write to memory in the caller's stack frame
- E) It stores a character array on the stack

# Buffer overflows

- Major x86-64 security issue
  - C and assembly do not check for out-of-bounds array accesses
  - x86-64 stores return addresses and data on the same stack
  - Out-of-bound writes to local variables may overwrite other stack frames
  - Allows attackers to change control flow just by providing the right "data"
  - Many historical exploits (including Morris worm)

```
void echo ()  
{  
    // other code  
    // omitted  
  
    char buf[8];  
    gets(buf);  
    printf(buf);  
}
```

**DO NOT WRITE  
CODE LIKE THIS!**

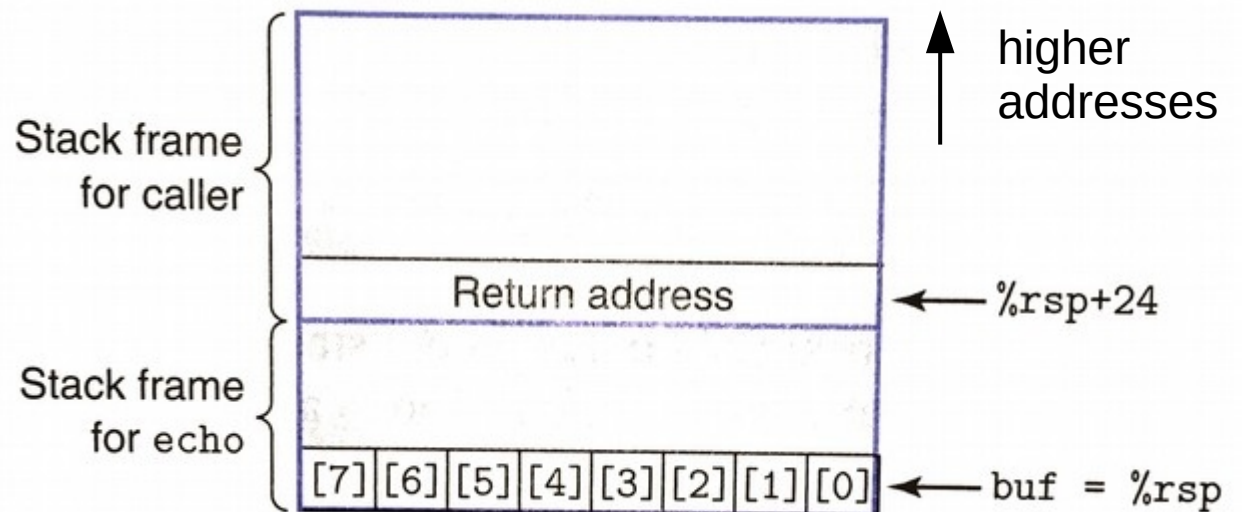


# Buffer overflows

- Shellcode (exploit code)
  - Pre-compiled snippets of code that exploit a buffer overflow

```
char shellcode[] =  
    "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"  
    "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"  
    "\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

Complication: Must pad the shellcode with address of the buffer (guess and/or use a NOP-sled)



# Mitigating buffer overflows

- Stack randomization
  - Randomize starting location of stack
  - Makes it more difficult to guess buffer address
  - In Linux: **address-space layout randomization**
- Corruption detection
  - Insert a **canary** (guard value) on stack after each array
  - Check canary before returning from function
- Read-only code regions
  - Mark stack memory as "no-execute"
  - Hinders just-in-time compilation and instrumentation

# Exercise

- Trace the following code--what is the value of %rax at the end?
  - Initial values: %rsp = 0x7fffffffefe488, %rip = 0x4004e8

4004d6 <leaf>:

4004d6: 48 8d 7f 0f  
4004da: c3

**leaq** 0xf(%rdi),%rdi  
**retq**

4004db <top>:

4004db: 48 83 ef 05  
4004df: e8 f2 ff ff ff  
4004e4: 48 01 ff  
4004e7: c3

**subq** \$0x5,%rdi  
**callq** 4004d6  
**addq** %rdi,%rdi  
**retq**

4004e8 <main>:

4004e8: 48 c7 c7 64 00 00 00  
4004ef: e8 e7 ff ff ff  
4004f4: 48 89 f8  
4004f7: c3

**movq** \$100,%rdi  
**callq** 4004db  
**movq** %rdi,%rax  
**retq**



# Exercise

- Trace the following code--what is the value of %rax at the end?

		%rip	0x4004e8
4004d6 <leaf>:		%rsp	0x7fffffffefe488
4004d6: <b>leaq</b>	0xf(%rdi),%rdi	%rdi	???
4004da: <b>retq</b>		%rax	???
4004db <top>:			...
4004db: <b>subq</b>	\$0x5,%rdi		
4004df: <b>callq</b>	4004d6		
4004e4: <b>addq</b>	%rdi,%rdi		
4004e7: <b>retq</b>			
		0x7fffffffefe488	???
4004e8 <main>:		0x7fffffffefe480	???
4004e8: <b>movq</b>	\$100,%rdi	0x7fffffffefe478	???
4004ef: <b>callq</b>	4004db	0x7fffffffefe470	???
4004f4: <b>movq</b>	%rdi,%rax		
4004f7: <b>retq</b>			...

# Exercise

- Trace the following code--what is the value of %rax at the end?

```
4004d6 <leaf>:  
  4004d6: leaq    0xf(%rdi),%rdi  
  4004da: retq
```

```
4004db <top>:  
  4004db: subq    $0x5,%rdi  
  4004df: callq   4004d6  
  4004e4: addq    %rdi,%rdi  
  4004e7: retq
```

```
4004e8 <main>:  
  4004e8: movq    $100,%rdi  
  4004ef: callq   4004db  
  4004f4: movq    %rdi,%rax  
  4004f7: retq
```

%rip

0x4004ef

%rsp

0x7fffffffefe488

%rdi

100

%rax

???

...

0x7fffffffefe488

???

0x7fffffffefe480

???

0x7fffffffefe478

???

0x7fffffffefe470

???

...

# Exercise

- Trace the following code--what is the value of %rax at the end?

```
4004d6 <leaf>:
  4004d6: leaq    0xf(%rdi),%rdi
  4004da: retq
```

```
4004db <top>:
  4004db: subq    $0x5,%rdi
  4004df: callq   4004d6
  4004e4: addq    %rdi,%rdi
  4004e7: retq
```

```
4004e8 <main>:
  4004e8: movq    $100,%rdi
  4004ef: callq   4004db
  4004f4: movq    %rdi,%rax
  4004f7: retq
```

%rip

0x4004db

%rsp

0x7fffffffef480

%rdi

100

%rax

???

...

0x7fffffffef488

???

0x7fffffffef480

0x4004f4

0x7fffffffef478

???

0x7fffffffef470

???

...

# Exercise

- Trace the following code--what is the value of %rax at the end?

		%rip	0x4004df
4004d6 <leaf>:		%rsp	0x7fffffffefe480
4004d6: <b>leaq</b>	0xf(%rdi),%rdi	%rdi	95
4004da: <b>retq</b>		%rax	???
4004db <top>:			...
4004db: <b>subq</b>	\$0x5,%rdi		
4004df: <b>callq</b>	4004d6		
4004e4: <b>addq</b>	%rdi,%rdi		
4004e7: <b>retq</b>			
		0x7fffffffefe488	???
4004e8 <main>:		0x7fffffffefe480	0x4004f4
4004e8: <b>movq</b>	\$100,%rdi	0x7fffffffefe478	???
4004ef: <b>callq</b>	4004db	0x7fffffffefe470	???
4004f4: <b>movq</b>	%rdi,%rax		
4004f7: <b>retq</b>			...

# Exercise

- Trace the following code--what is the value of %rax at the end?

```
4004d6 <leaf>:
  4004d6: leaq    0xf(%rdi),%rdi
  4004da: retq
```

```
4004db <top>:
  4004db: subq    $0x5,%rdi
  4004df: callq   4004d6
  4004e4: addq    %rdi,%rdi
  4004e7: retq
```

```
4004e8 <main>:
  4004e8: movq    $100,%rdi
  4004ef: callq   4004db
  4004f4: movq    %rdi,%rax
  4004f7: retq
```

%rip

0x4004d6

%rsp

0x7fffffffef478

%rdi

95

%rax

???

...

0x7fffffffef488

???

0x7fffffffef480

0x4004f4

0x7fffffffef478

0x4004e4

0x7fffffffef470

???

...

# Exercise

- Trace the following code--what is the value of %rax at the end?

```
4004d6 <leaf>:  
  4004d6: leaq    0xf(%rdi),%rdi  
  4004da: retq
```

```
4004db <top>:  
  4004db: subq    $0x5,%rdi  
  4004df: callq   4004d6  
  4004e4: addq    %rdi,%rdi  
  4004e7: retq
```

```
4004e8 <main>:  
  4004e8: movq    $100,%rdi  
  4004ef: callq   4004db  
  4004f4: movq    %rdi,%rax  
  4004f7: retq
```

%rip

0x4004da

%rsp

0x7fffffffefe478

%rdi

110

%rax

???

...

0x7fffffffefe488

???

0x7fffffffefe480

0x4004f4

0x7fffffffefe478

0x4004e4

0x7fffffffefe470

???

...

# Exercise

- Trace the following code--what is the value of %rax at the end?

```
4004d6 <leaf>:  
  4004d6: leaq    0xf(%rdi),%rdi  
  4004da: retq
```

```
4004db <top>:  
  4004db: subq    $0x5,%rdi  
  4004df: callq   4004d6  
  4004e4: addq    %rdi,%rdi  
  4004e7: retq
```

```
4004e8 <main>:  
  4004e8: movq    $100,%rdi  
  4004ef: callq   4004db  
  4004f4: movq    %rdi,%rax  
  4004f7: retq
```

%rip

0x4004e4

%rsp

0x7fffffffefe480

%rdi

110

%rax

???

...

0x7fffffffefe488

???

0x7fffffffefe480

0x4004f4

0x7fffffffefe478

0x4004e4

0x7fffffffefe470

???

...

# Exercise

- Trace the following code--what is the value of %rax at the end?

```
4004d6 <leaf>:
  4004d6: leaq    0xf(%rdi),%rdi
  4004da: retq
```

```
4004db <top>:
  4004db: subq    $0x5,%rdi
  4004df: callq   4004d6
  4004e4: addq    %rdi,%rdi
  4004e7: retq
```

```
4004e8 <main>:
  4004e8: movq    $100,%rdi
  4004ef: callq   4004db
  4004f4: movq    %rdi,%rax
  4004f7: retq
```

%rip

0x4004e7

%rsp

0x7fffffffefe480

%rdi

220

%rax

???

...

0x7fffffffefe488

???

0x7fffffffefe480

0x4004f4

0x7fffffffefe478

0x4004e4

0x7fffffffefe470

???

...



# Exercise

- Trace the following code--what is the value of %rax at the end?

```
4004d6 <leaf>:
  4004d6: leaq    0xf(%rdi),%rdi
  4004da: retq
```

```
4004db <top>:
  4004db: subq    $0x5,%rdi
  4004df: callq   4004d6
  4004e4: addq    %rdi,%rdi
  4004e7: retq
```

```
4004e8 <main>:
  4004e8: movq    $100,%rdi
  4004ef: callq   4004db
  4004f4: movq    %rdi,%rax
  4004f7: retq
```

%rip

0x4004f4

%rsp

0x7fffffffffe488

%rdi

220

%rax

???

...

0x7fffffffffe488

???

0x7fffffffffe480

0x4004f4

0x7fffffffffe478

0x4004e4

0x7fffffffffe470

???

...

# Exercise

- Trace the following code--what is the value of %rax at the end?

		%rip	0x4004f7
4004d6 <leaf>:		%rsp	0x7fffffffefe488
4004d6: <b>leaq</b>	0xf(%rdi),%rdi	%rdi	220
4004da: <b>retq</b>		%rax	220
4004db <top>:			...
4004db: <b>subq</b>	\$0x5,%rdi	0x7fffffffefe488	???
4004df: <b>callq</b>	4004d6	0x7fffffffefe480	0x4004f4
4004e4: <b>addq</b>	%rdi,%rdi	0x7fffffffefe478	0x4004e4
4004e7: <b>retq</b>		0x7fffffffefe470	???
4004e8 <main>:			...
4004e8: <b>movq</b>	\$100,%rdi		
4004ef: <b>callq</b>	4004db		
4004f4: <b>movq</b>	%rdi,%rax		
4004f7: <b>retq</b>			

# Exercise

- Trace the following code--what is the value of %rax at the end?

```
4004d6 <leaf>:  
  4004d6: leaq    0xf(%rdi),%rdi  
  4004da: retq
```

```
4004db <top>:  
  4004db: subq    $0x5,%rdi  
  4004df: callq   4004d6  
  4004e4: addq    %rdi,%rdi  
  4004e7: retq
```

```
4004e8 <main>:  
  4004e8: movq    $100,%rdi  
  4004ef: callq   4004db  
  4004f4: movq    %rdi,%rax  
  4004f7: retq
```

%rip

???

%rsp

0x7fffffffef490

%rdi

220

%rax

220

...

0x7fffffffef488

???

0x7fffffffef480

0x4004f4

0x7fffffffef478

0x4004e4

0x7fffffffef470

???

...

# Exercise

- Trace the following code--what is the value of %rax at the end?

		%rip	???
4004d6 <leaf>:		%rsp	0x7fffffffefe490
4004d6:	leaq 0xf(%rdi),%rdi	%rdi	220
4004da:	retq	%rax	220
4004db <top>:		...	
4004db:	subq \$0x5,%rdi	0x7fffffffefe488	???
4004df:	callq 4004d6	0x7fffffffefe480	0x4004f4
4004e4:	addq %rdi,%rdi	0x7fffffffefe478	0x4004e4
4004e7:	retq	0x7fffffffefe470	???
4004e8 <main>:		...	
4004e8:	movq \$100,%rdi		
4004ef:	callq 4004db		
4004f4:	movq %rdi,%rax		
4004f7:	retq		