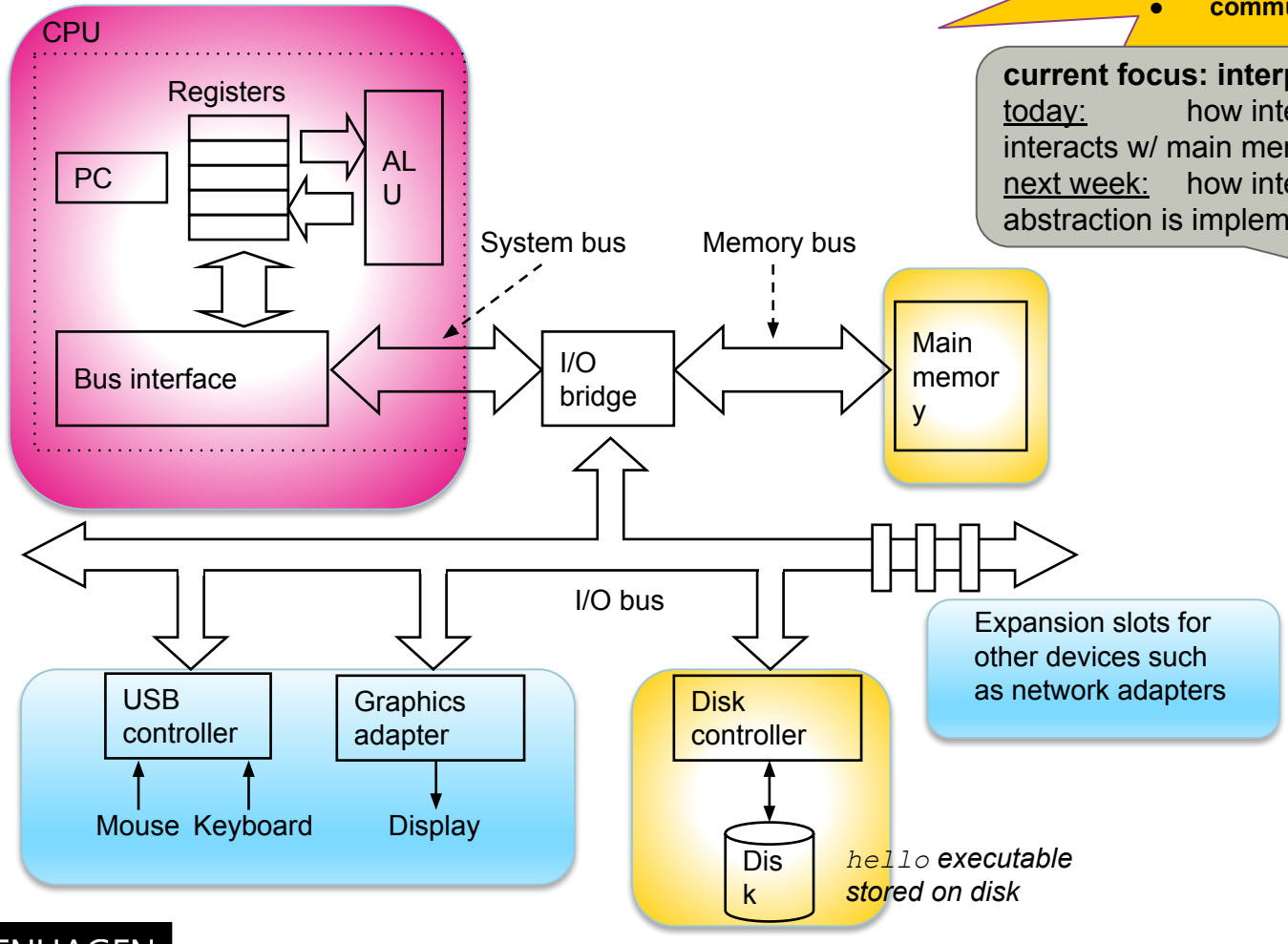
Operating Systems and C
Fall 2022
3. Representing Instructions

Computer Hardware

recall fundamental abstractions:
- **interpreter**,
- **memory**,
- **communication**

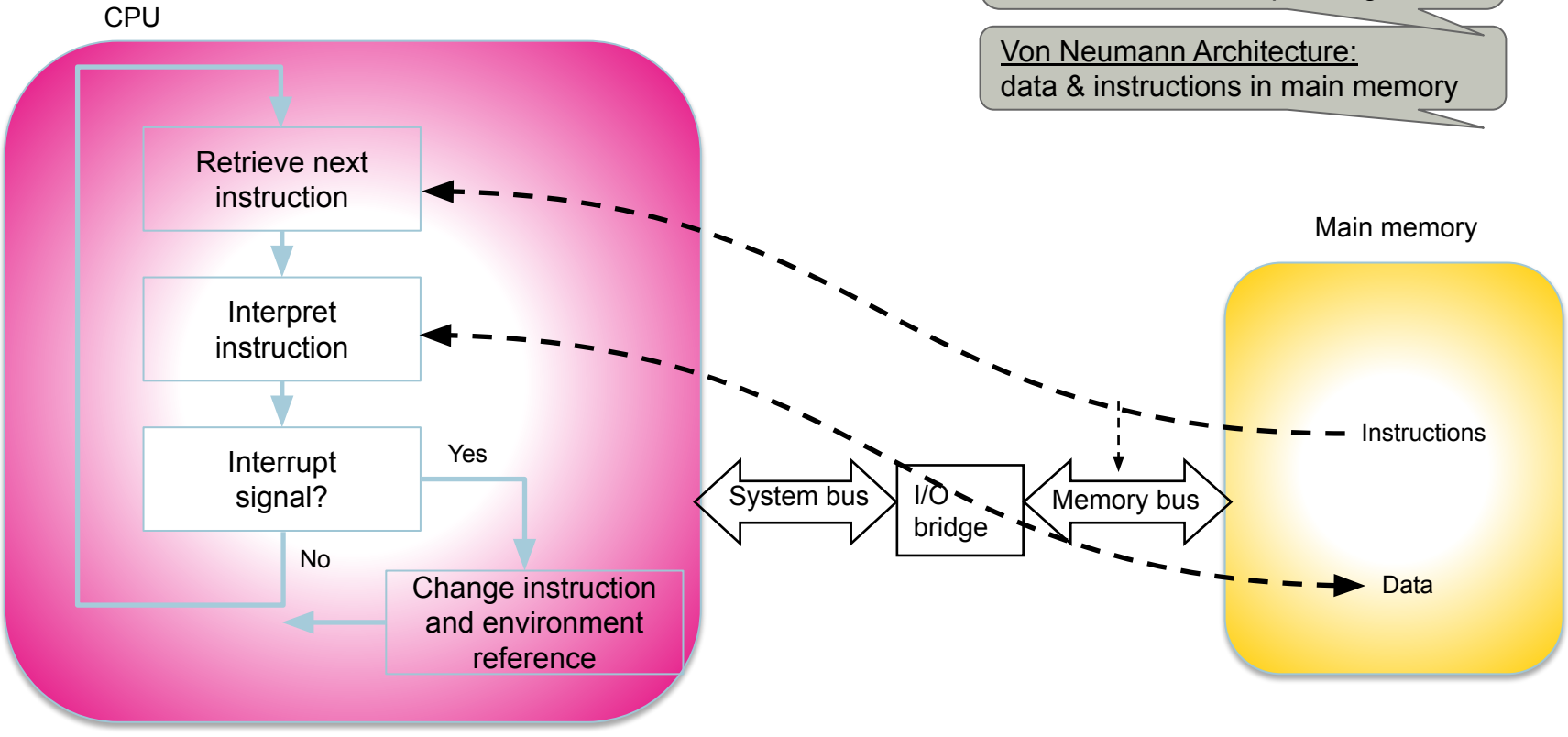
current focus: interpreter.
today: how interpreter interacts w/ main memory.
next week: how interpreter abstraction is implemented.



How does a processor work?

Sequential execution of instructions.
Simplification (caching, pipelining, multi-core, …)
Move data around, op on regs, …

Von Neumann Architecture:
data & instructions in main memory



Instruction repertoire: CISC / RISC

A single core CPU is an interpreter

Last Week: Data Representation

Sequence of bits (organized in bytes) **interpreted as**:

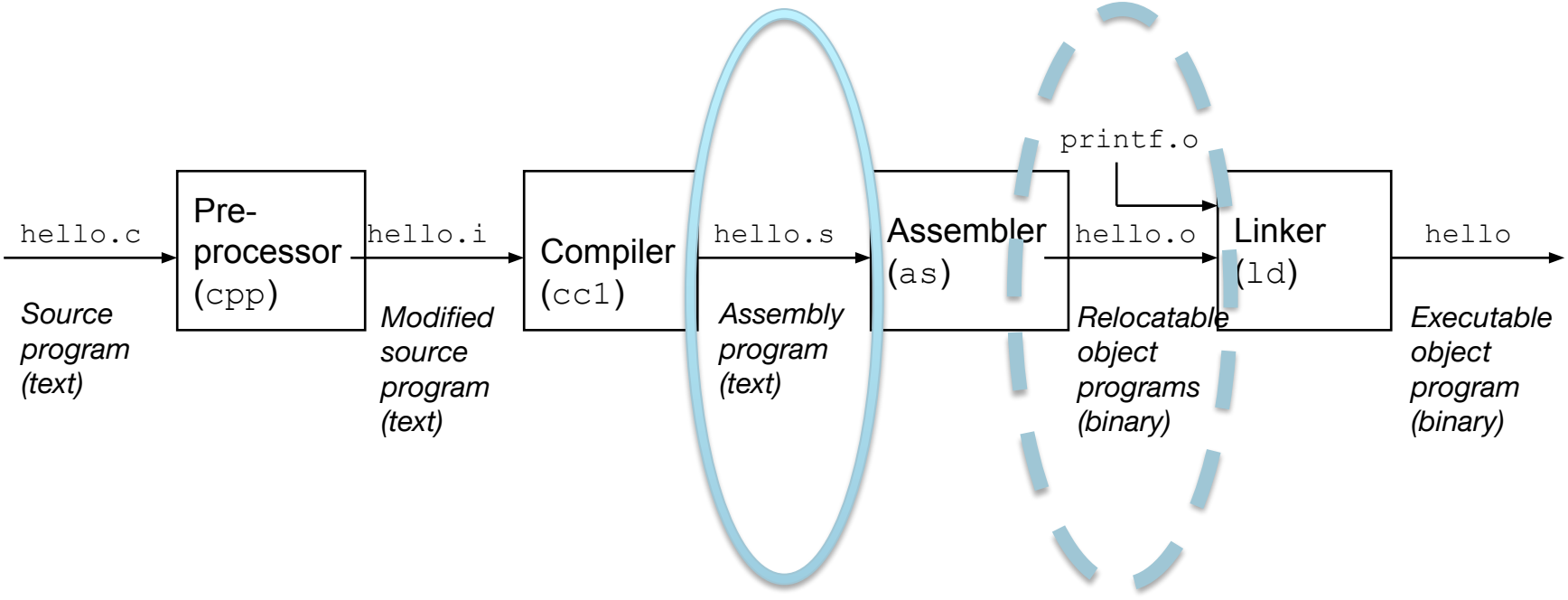
- Sequence of bits
- Boolean
- Integer (Two's complement)
- Float (IEEE 754) – Interested? Check out posit
- Characters – ASCII (1B), UTF-8 (1-4B)
- Strings – Array of characters terminated by '\0'

How to represent instruction?
(What is an instruction in the first place?)

Compilation phases

https://github.com/gcc-mirror/gcc

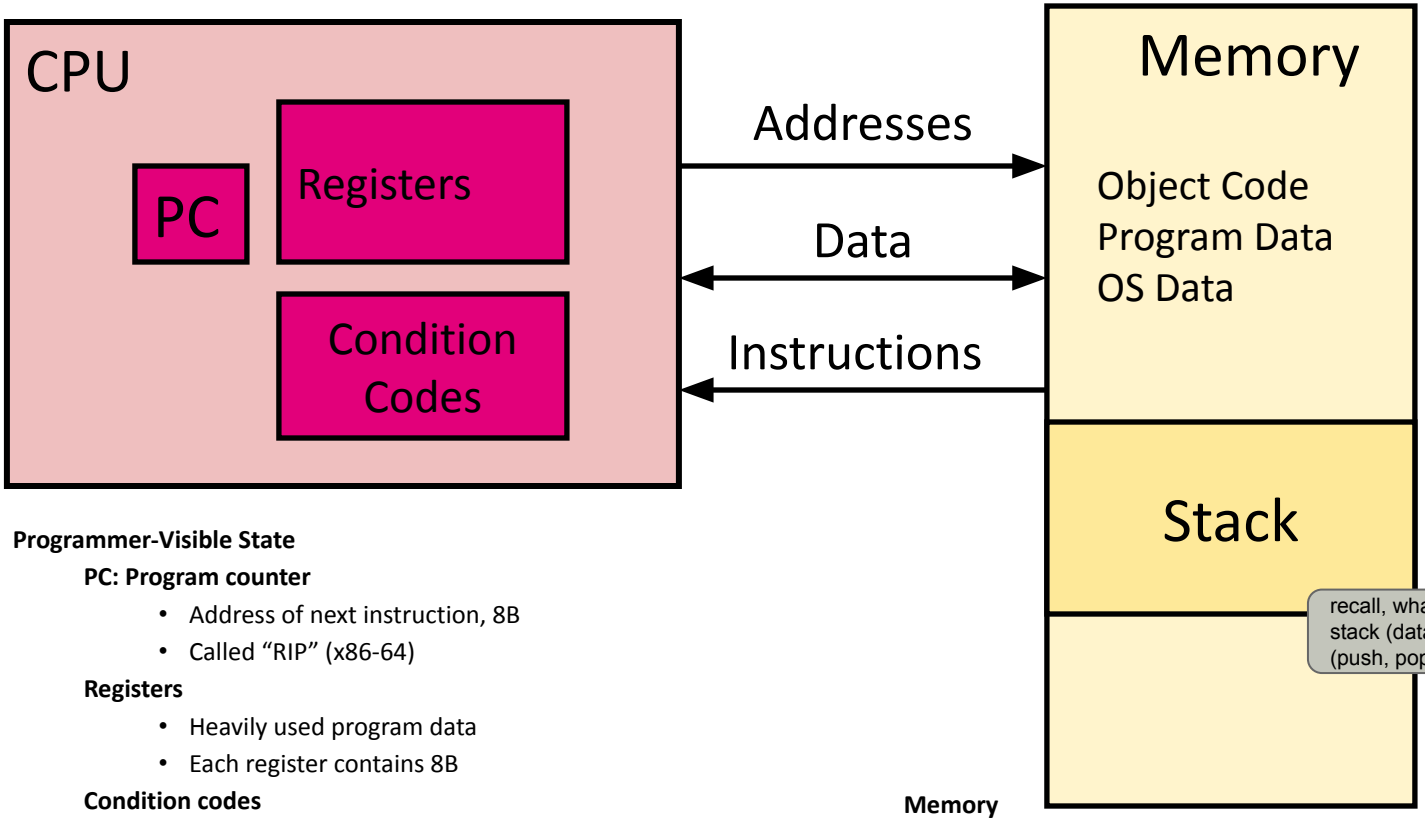
we're looking at these (mostly former)



$ gcc -save-temps hello.c

X86-64 Processor Abstraction

instruction either
- op on registers (state), or
- transfers data to/from mem



recall, what is a stack (data structure)? (push, pop)

Programmer-Visible State

PC: Program counter
- Address of next instruction, 8B
- Called "RIP" (x86-64)

Registers
- Heavily used program data
- Each register contains 8B

Condition codes
- Store status information about most recent arithmetic operation
- Used for conditional branching

Memory
- Byte addressable array
- Code, user data, (some) OS data
- Includes stack used to support procedures

Instructions (**Outline** for today)

Three classes of instructions:

1. Transfer between memory and register
   - Load/store data: register <-> memory
   - Push/pop: register <-> stack

How are registers organized?
How is memory addressed?

2. Arithmetic and comparison functions

3. Transfer control
   - Jumps to/from procedures
   - Conditional branches

How are procedure calls organized?

AT&T syntax

The GNU tools (gcc, gdb) use AT&T Syntax for assembly.
example: movq %rsp, %rbp

Syntax is of the form
OPERATOR source, destination

never more than 2 operands.
when there are 2, this is the form.

Register names are prefixed with **%**

The **alternative** is the **Intel syntax** (on windows): MOVQ EBP, ESP – no %
Look for % in the assembly code, if they are present you are dealing with AT&T syntax

Registers

How are registers organized?

The **%rip** register is the current **i**nstruction **p**ointer.
Contains address of next instruction to be executed.

most instructions implicitly increment it.
explicitly updated ⇒ change in control flow.

There are **16 general purpose registers** in x86-64.
Additional registers for floating point, SIMD, …
16 registers: r0, r1, …, r15

"register file"

General-Purpose Registers

For historical reasons, r0-r7 are called **original registers**. They have the following names:

- ax: register a
- bx: register b
- cx: register c
- dx: register d

- bp: register **b**ase **p**ointer (start of stack)
- sp: register **s**tack **p**ointer (current location in stack, grow downwards)

- si: register source index (source for data copies)
- di: register destination index (destination for data copies)

General-Purpose Registers

why: e.g.
C short is 2B

Register values can be accessed at different levels of granularity:

- **8B**:
  - original registers: **prefix r** — rax, rsp, rsi
  - other registers: no suffix — r8, r15
- **4B**:
  - original registers: **prefix e** — eax, esp, esi
  - other registers: **suffix d** — r8d, r15d
- **2B:**
  - original registers: **no prefix** — ax, sp, si
  - other registers: **suffix w** — r8w, r15w
- **1B** (high byte):
  - original registers (bits 8-15 from ax-dx) — ah, bh, ch, dh
- **1B** (low byte):
  - original registers (bits 0-7 from ax-dx) — al, bl, cl, dl
  - other registers: suffix b — r8b, r15b

Examples

focus on circles now.
size of registers?

ex3.s ex3.c

```c
#include <stdio.h>

int main()
{
        int age = 10;
        int height = 152;

        printf("I am %#x years old, and %+d cm high.\n", age, height);

        return 1;;
}
```

$ gcc –S -o ex3.s ex3.c
$ vi ex3.s

ex3.s

```
        .file   "ex3.c"
        .section        .rodata
        .align 8
.LC0:
        .string "I am %#x years old, and %+d cm high.\n"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    $10, -8(%rbp)
        movl    $152, -4(%rbp)
        movl    -4(%rbp), %edx
        movl    -8(%rbp), %eax
        movl    %eax, %esi
        movl    $.LC0, %edi
        movl    $0, %eax
        call    printf
        movl    $1, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.4) 5.4.0 20160609"
        .section        .note.GNU-stack,"",@progbits
```

mov

How is memory addressed?

need arithmetic on them
- increment for next instruction
- offset for *n*th array item

mov: copies data from one location to another

(pointer is not an assembly concept.
it's a C concept.)
in assembly, you have registers, and
what is the content of those.

mov

mov applied to arguments of 1B, 2B, 4B, 8B

- 1B – byte (b): movb
- 2B – word (w): movw
- 4B – double word (l): movl
- 8B – quadword (q): movq

mov

possible moves:
- register-to-memory
- register-to-register

if you want memory-to-memory, you do that in 2 steps via. register.

Access modes

- Direct: immediate values prefixed by $

  constant

  ```
  movq $0x2a, %rax      // put the immediate value 0x2a into rax
  ```

- Register: memory at (register)

  ```
  movq %r10, (%r11)     // store data from r10 to address pointed to by r11
  movq (%r10), %r11     // load data from address pointed to by r10 to r11
  ```

mov

- Register plus offset: memory at offset(register)

  // store data from r10 at the address pointed to by (r11) - 8B

  movq %r10, -8(%r11)

  // load data from address pointed to by r10 + 4B into r11

  movq 4(%r10), %r11

- Register * scale plus offset: (offset, register, scale)

  // store data from r10 at address (r9+r11*4)

  movq %r10, (%r9, %r11, 4)

Example

a lot of the instructions are just moves!

ex3.s ex3.c

```c
#include <stdio.h>

int main()
{
        int age = 10;
        int height = 152;

        printf("I am %#x years old, and %+d cm high.\n", age, height);

        return 1;;
}
```

$ gcc –S -o ex3.s ex3.c
$ vi ex3.s

ex3.s

```
        .file   "ex3.c"
        .section        .rodata
        .align 8
.LC0:
        .string "I am %#x years old, and %+d cm high.\n"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    $10, -8(%rbp)
        movl    $152, -4(%rbp)
        movl    -4(%rbp), %edx
        movl    -8(%rbp), %eax
        movl    %eax, %esi
        movl    $.LC0, %edi
        movl    $0, %eax
        call    printf
        movl    $1, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.4) 5.4.0 20160609"
        .section        .note.GNU-stack,"",@progbits
```

Conversions: movs, movz

- Sign extension: **movs[**two suffixes] SRC, DEST

  DEST = sign extension of SRC

  Two suffixes: bw (1B to 2B), bl, bq, wl (2B to 4B), wq, lq

- Zero extension: **movz[**two suffixes] SRC, DEST

  DEST = sign extension of SRC

C is a procedural language

How are procedure calls organized?

"C has been designed to make functions efficient and easy to use"
K&R

"Procedural programming is a programming paradigm, derived from structured programming, based on the concept of the procedure call. Procedures, also known as routines, subroutines, or functions, simply contain a series of computational steps to be carried out."
Wikipedia!

arith.c

```c
#include <stdio.h>
#include <stdlib.h>

int logical(int x, int y)
{
        int t1 = x^y;
        int t2 = t1 >> 17;
        int mask = (1<<13) - 7;
        int rval = t2 & mask;
        return rval;
}

int main(int argc, char* argv[])
{
        if (argc != 3) {
                printf("Usage: arith x y\n");
                return 1;
        }

        int x = atoi(argv[1]);
        int y = atoi(argv[2]);
        printf("Arguments x: %d, y: %d\n", x, y);
        printf("Logical returns: %d\n", logical(x,y));
        printf("\n");

        return 0;
}
```

GP Register Usage during function calls

First six arguments of a function stored in
    di, si, dx, cx, r8, r9
Remaining arguments are on the
    stack (more later)

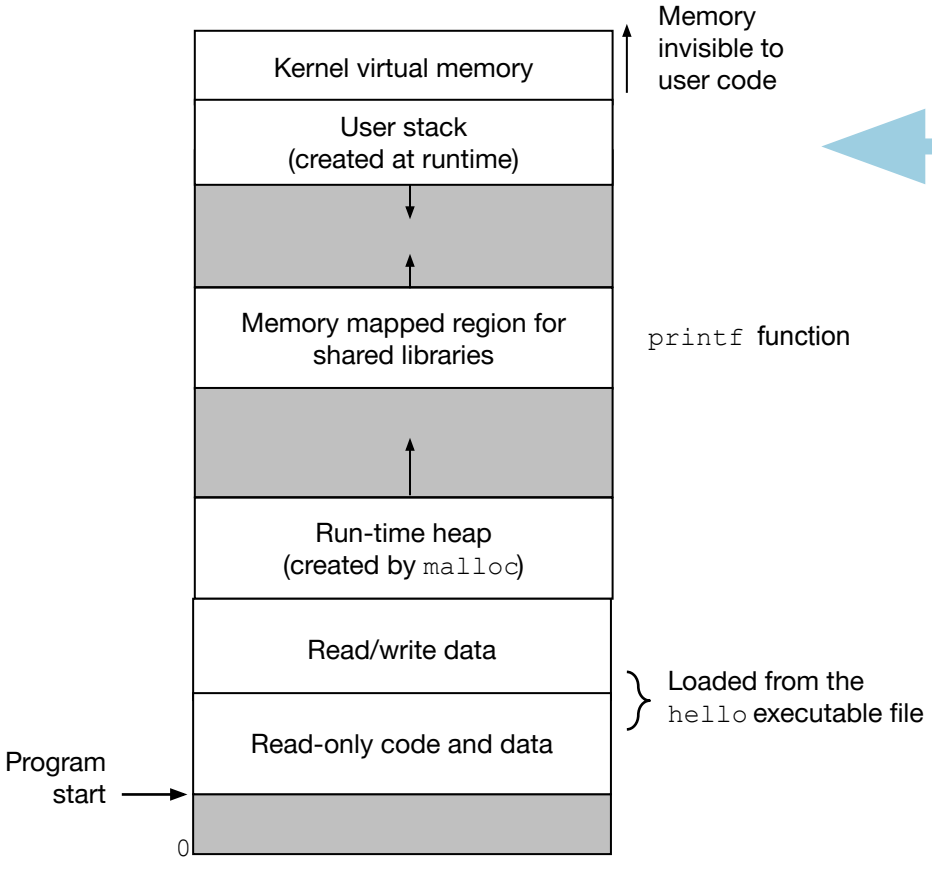
Return value is in rax

What is the stack?
How is the stack used?

Calling a function preserves rbp, rbx, r12-15. The other registers **might be overwritten**.

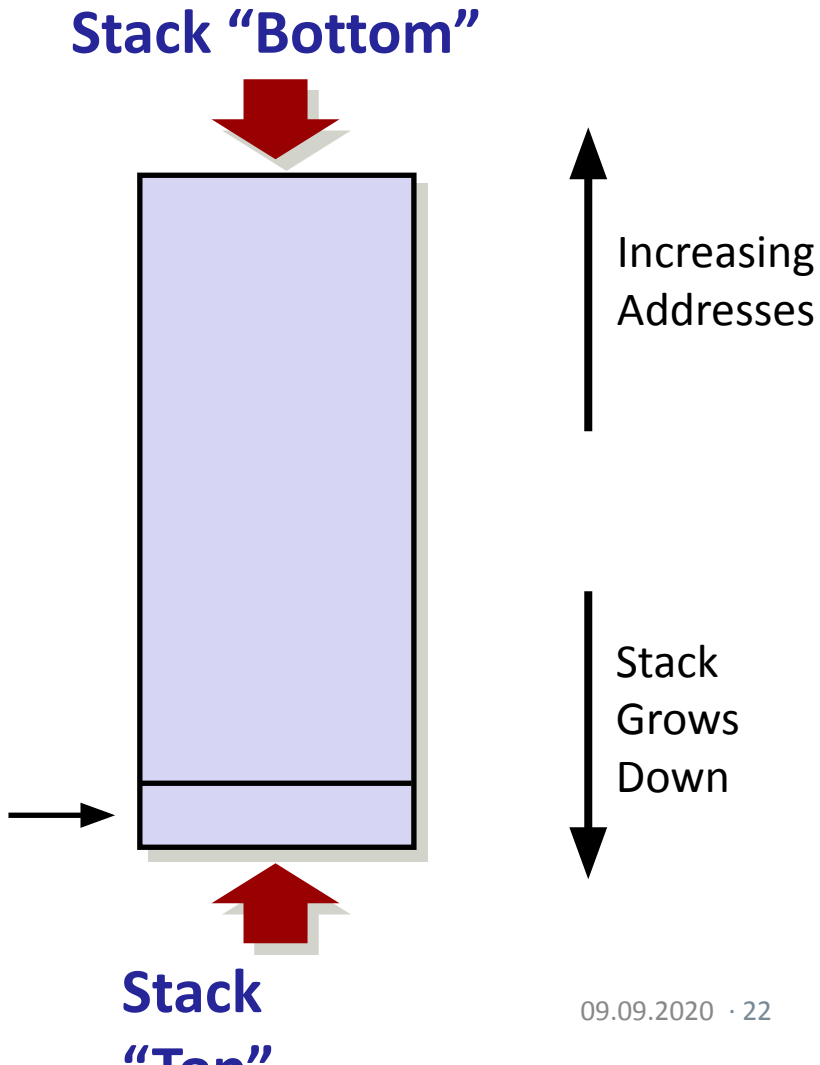
Virtual Memory

What is the stack?



The Stack

- Region of memory managed with stack discipline
- Grows towards lower addresses
- Regiser %rbp points to the bottom of the stack
- Register %rsp points to the top of the stack, it is the **stack pointer**

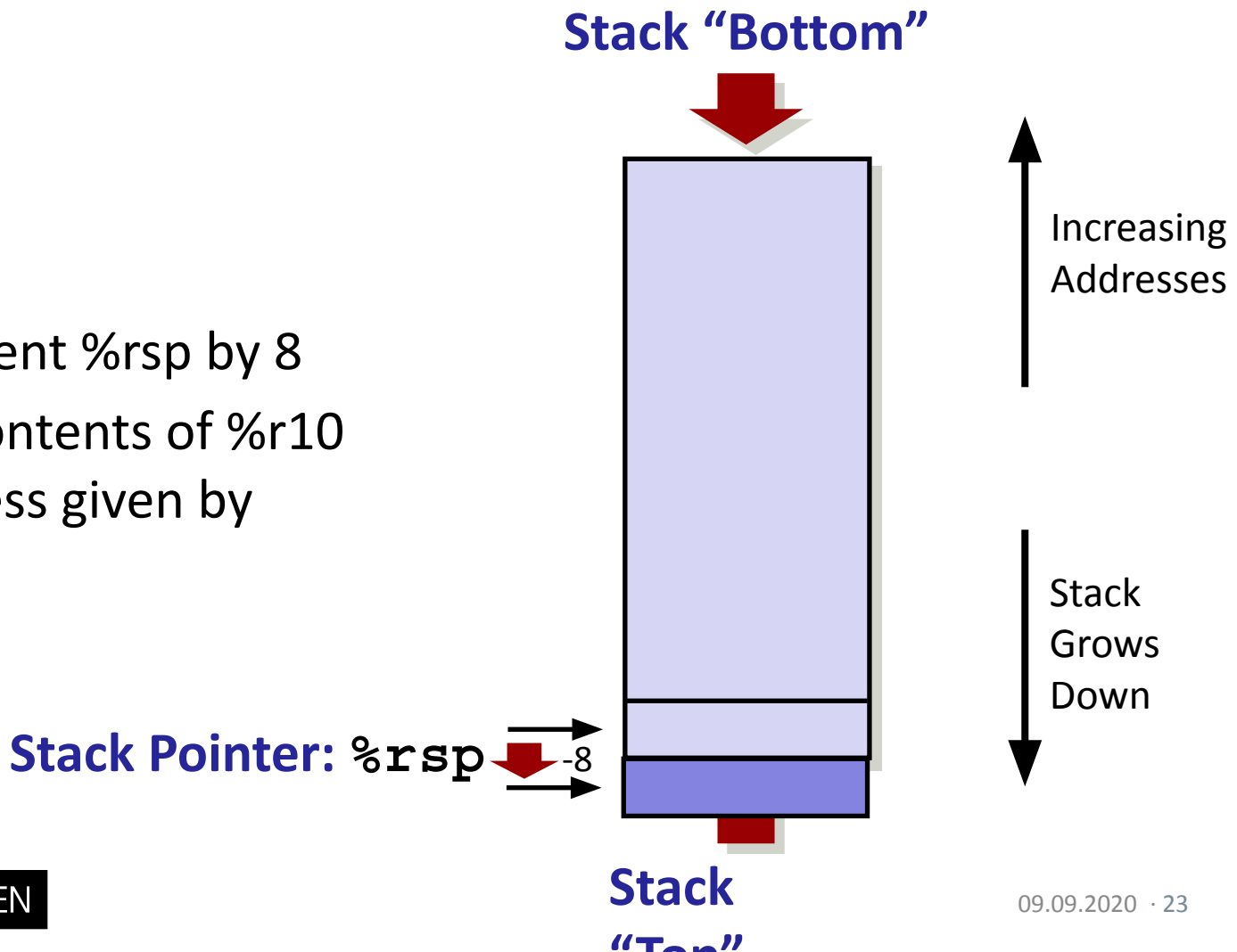


Stack - Push

Operator push

pushq %r10

1. Decrement %rsp by 8
2. Write contents of %r10 at address given by %rsp

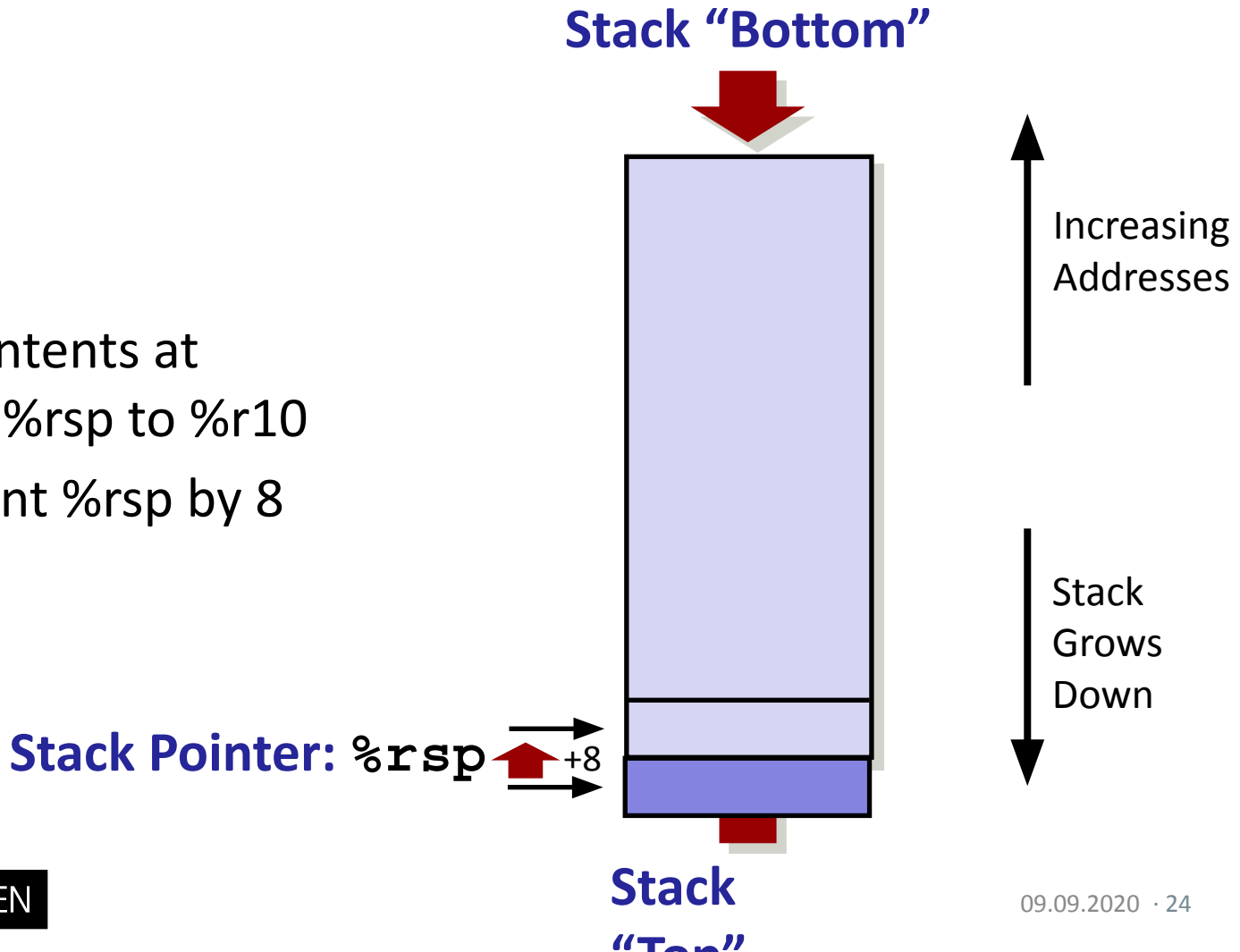


Stack - Pop

Operator pop

popq %r10

1. Copy contents at address %rsp to %r10
2. Increment %rsp by 8



Procedure Call

Why / How
is the stack used?

Stack is used to support control flow:

- Calling procedure (call instruction)
- Returning from procedure (ret instruction)

Example

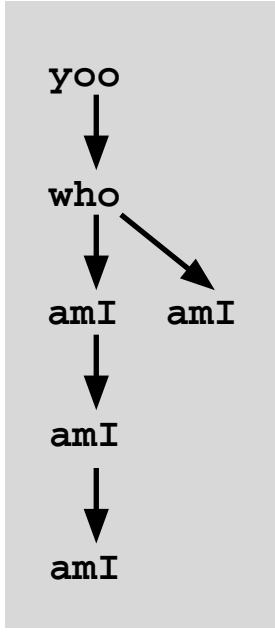
why/how is the stack relevant
when talking about procedure calls?

```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

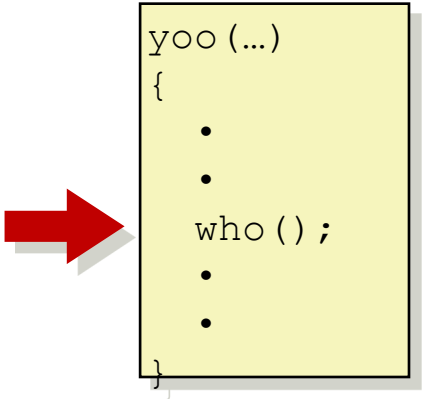
Example
Call Chain

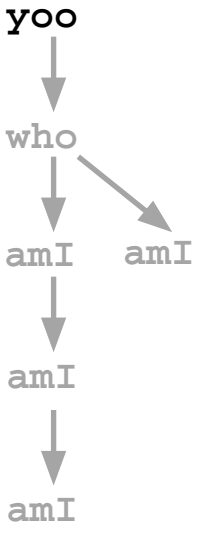


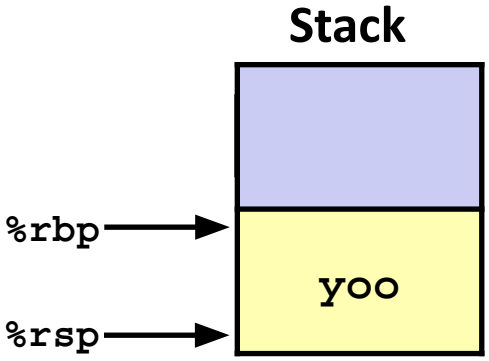
Procedure `amI()` is <u>recursive</u>

we use the stack to keep track of deep
(possibly recursive) procedure calls.

Example





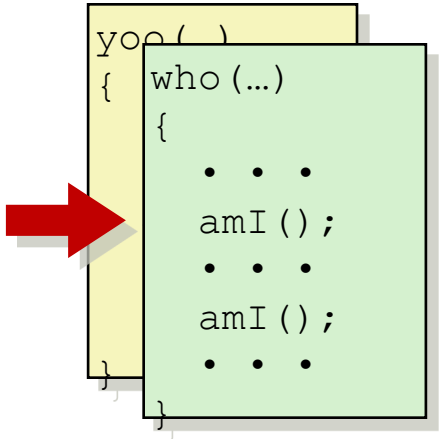


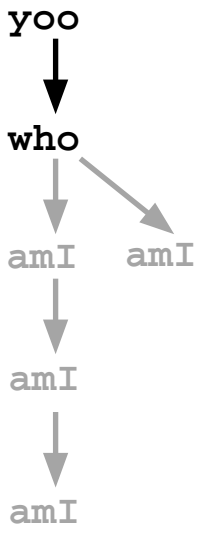
stack frame

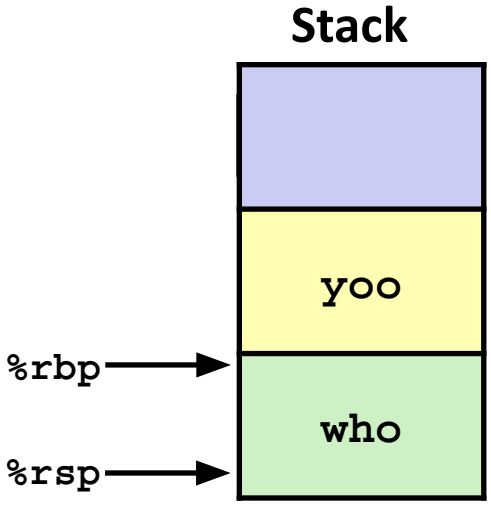
- caller's saved registers[1]
- local vars
- args
- return address[2]

[1] : to restore caller state on return (bottom frame has no caller).
[2] : pushes it when it calls (top frame does not need this)

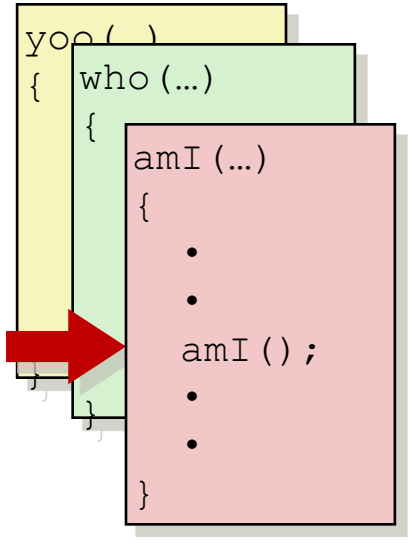
Example

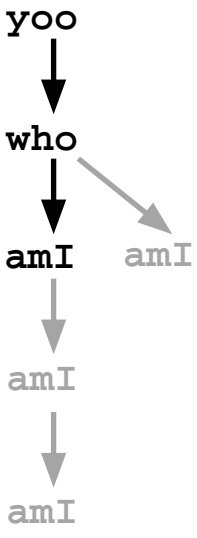


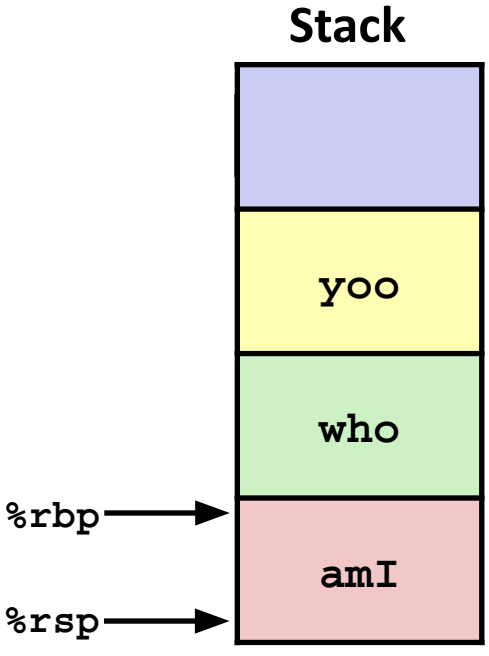




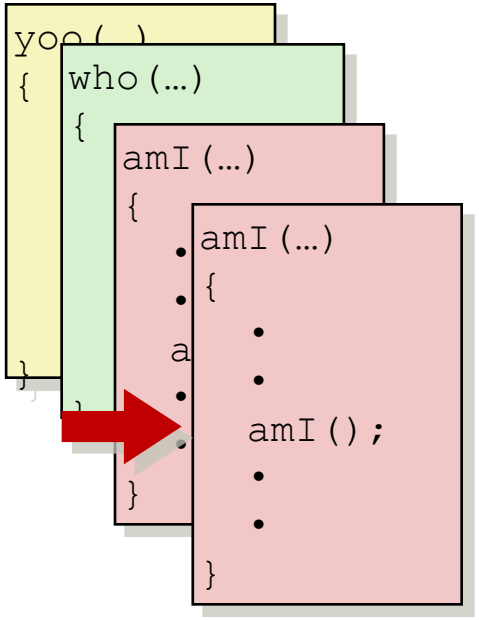
Example

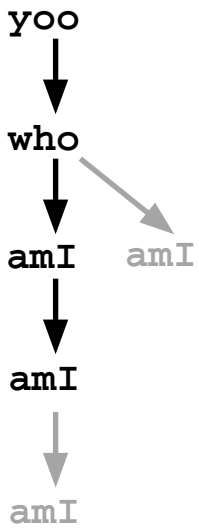

yoo( )
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}

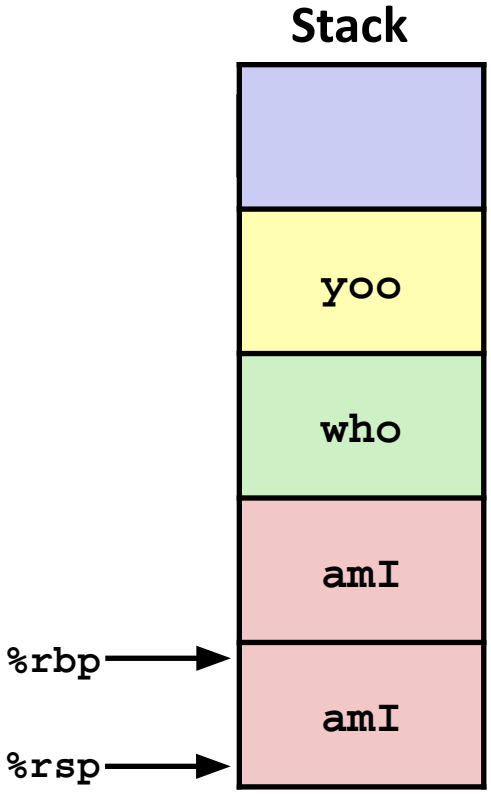

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

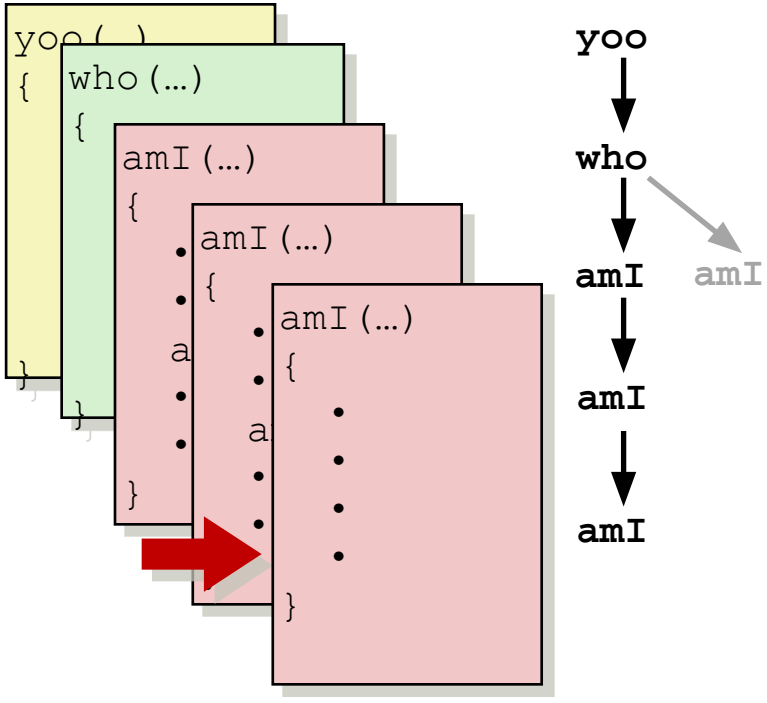
Example


who(…)
{
amI(…)
{
amI(…)
{
amI();
}
}

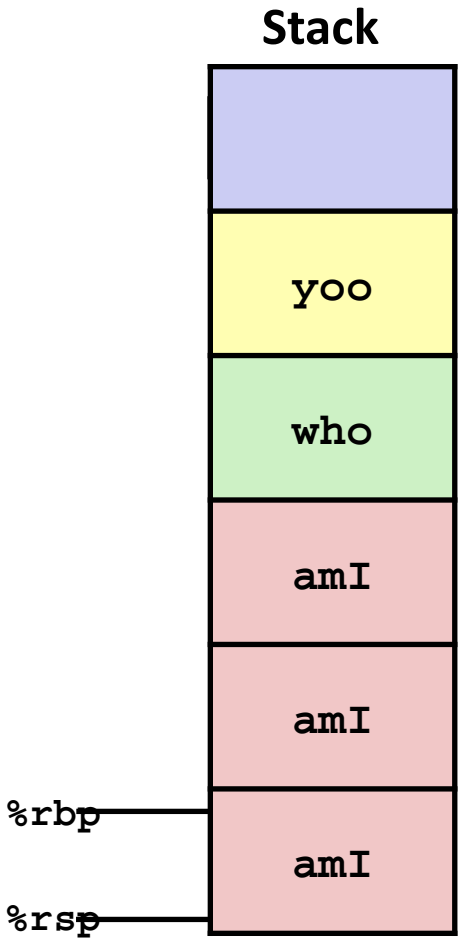

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

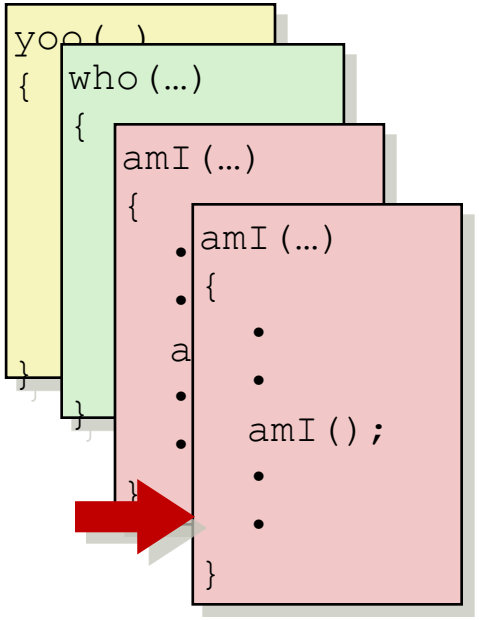
Example

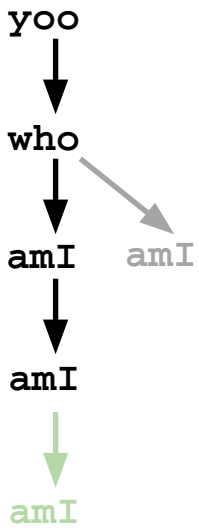

who (…)
amI (…)
amI (…)
amI (…)

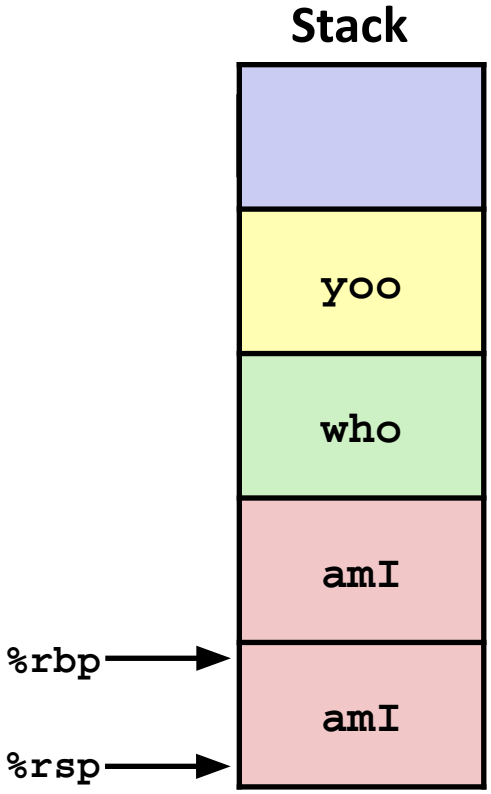
yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
amI
%rbp
%rsp

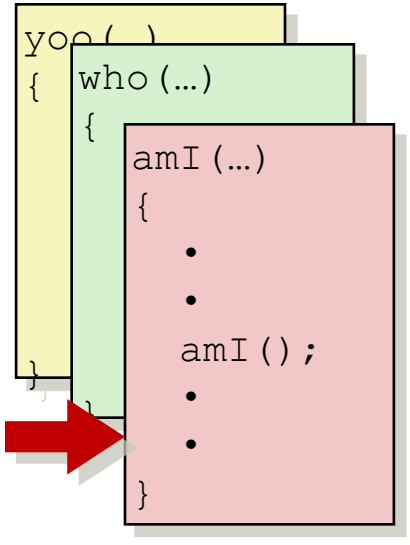
Example

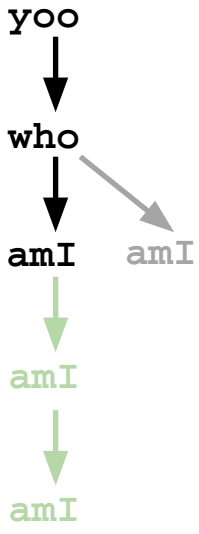

yoo( )
{
who(…)
{
amI(…)
{
amI(…)
{
amI();
}

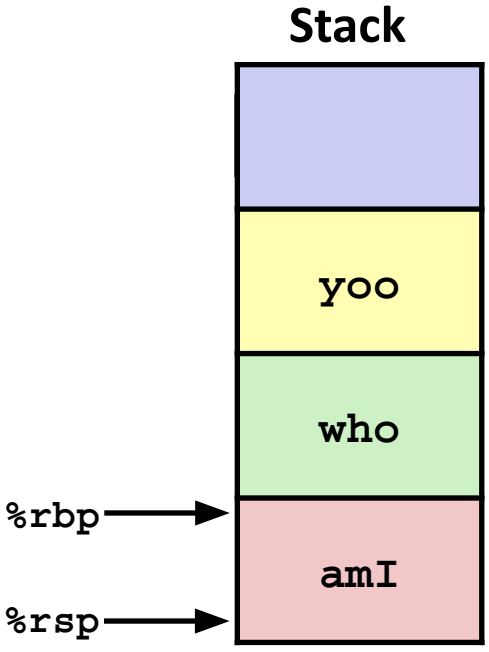

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

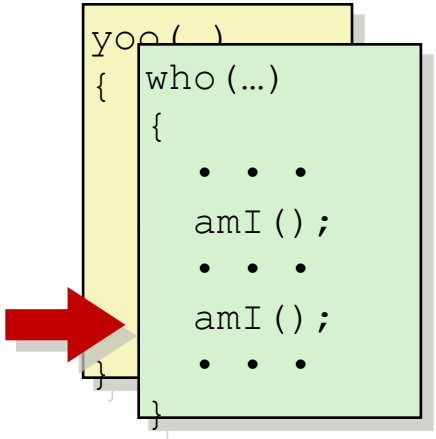
Example

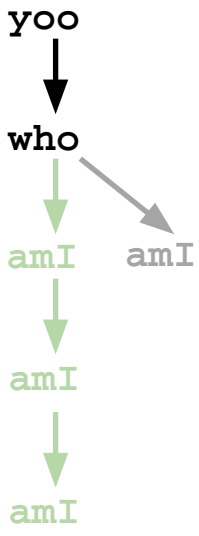

yoo( )
{
who(…)
{
amI(…)
{
amI();
}

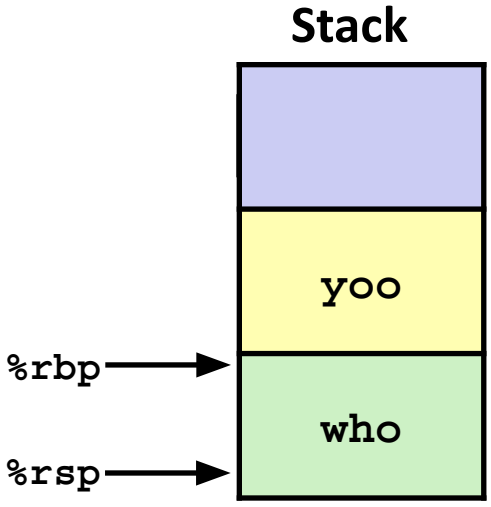

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

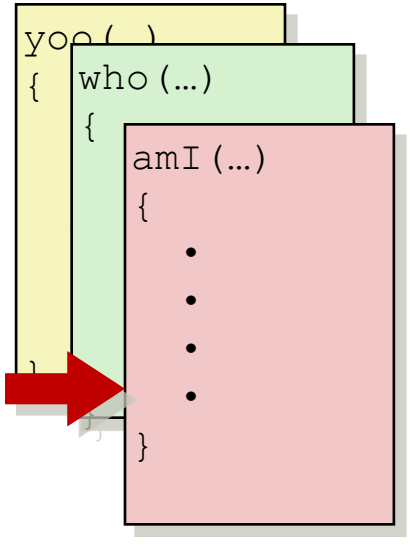
Example

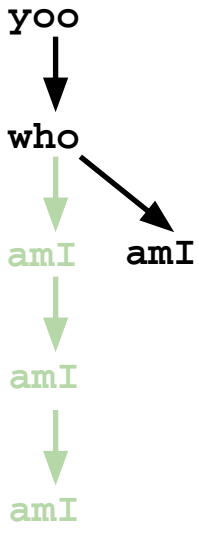


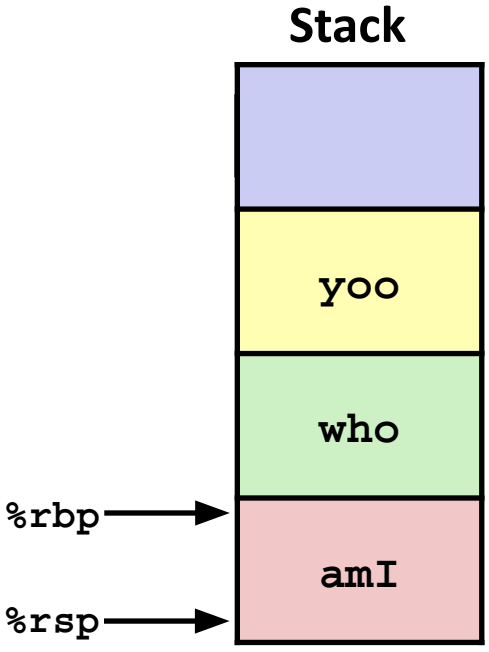




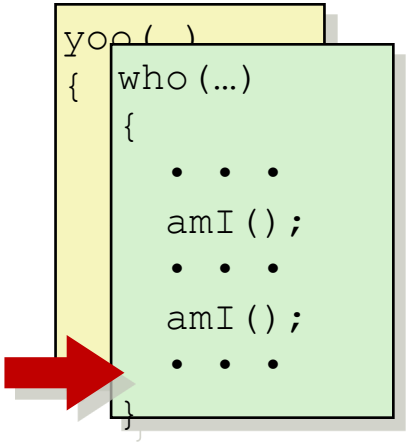
Example

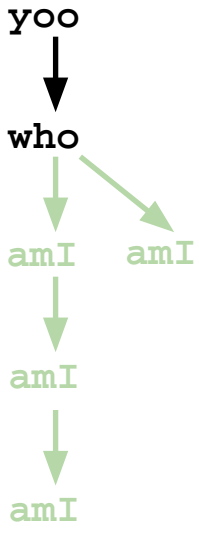


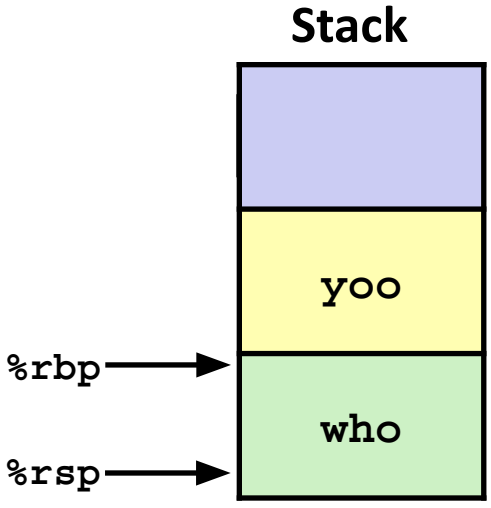




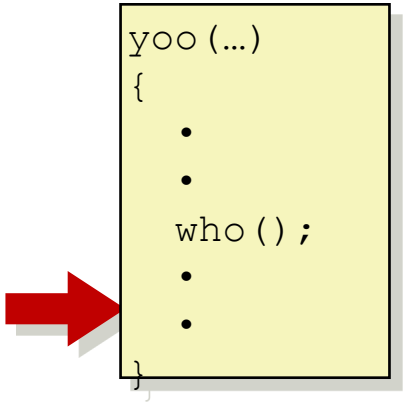
Example

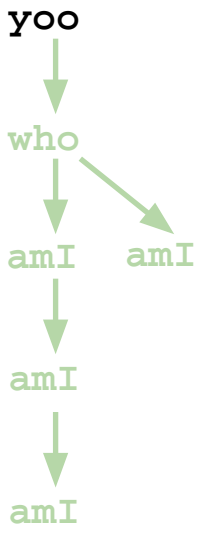


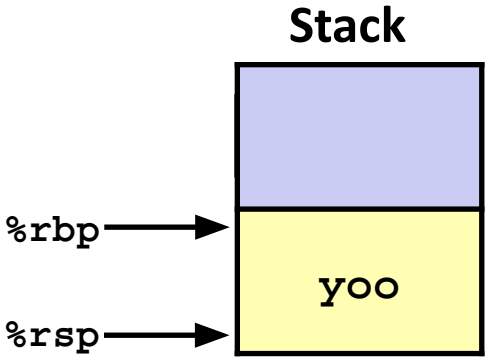




Example







Procedure Call

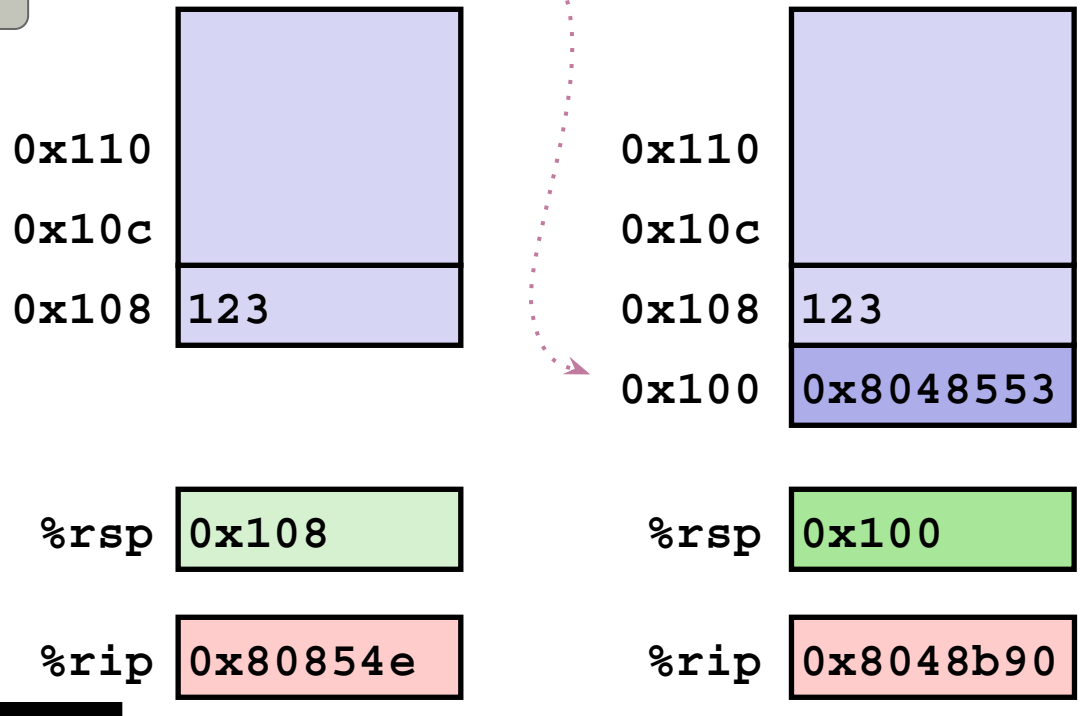
Instructions:

- **call**: push **return address** on stack; jump to label/address
  - Return address is address of instruction right after call instruction
- **ret**: pop address from stack; jump to address

Procedure Call Example

```
804854e:  e8 3d 06 00 00   call   8048b90 <main>
8048553:  50               pushl  %eax
```

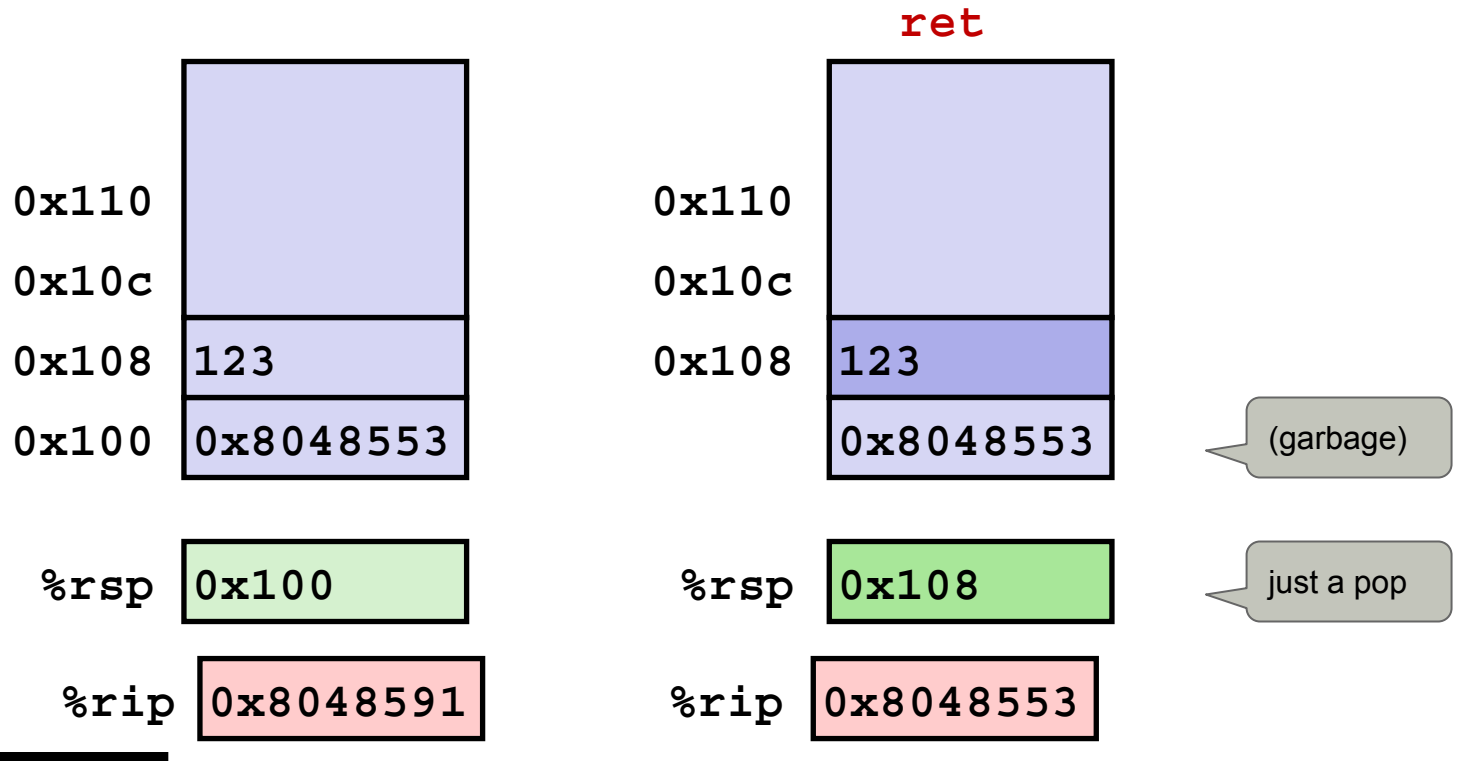
return address
= address of next instruction



instruction pointer
updated to callee

Procedure Call Example

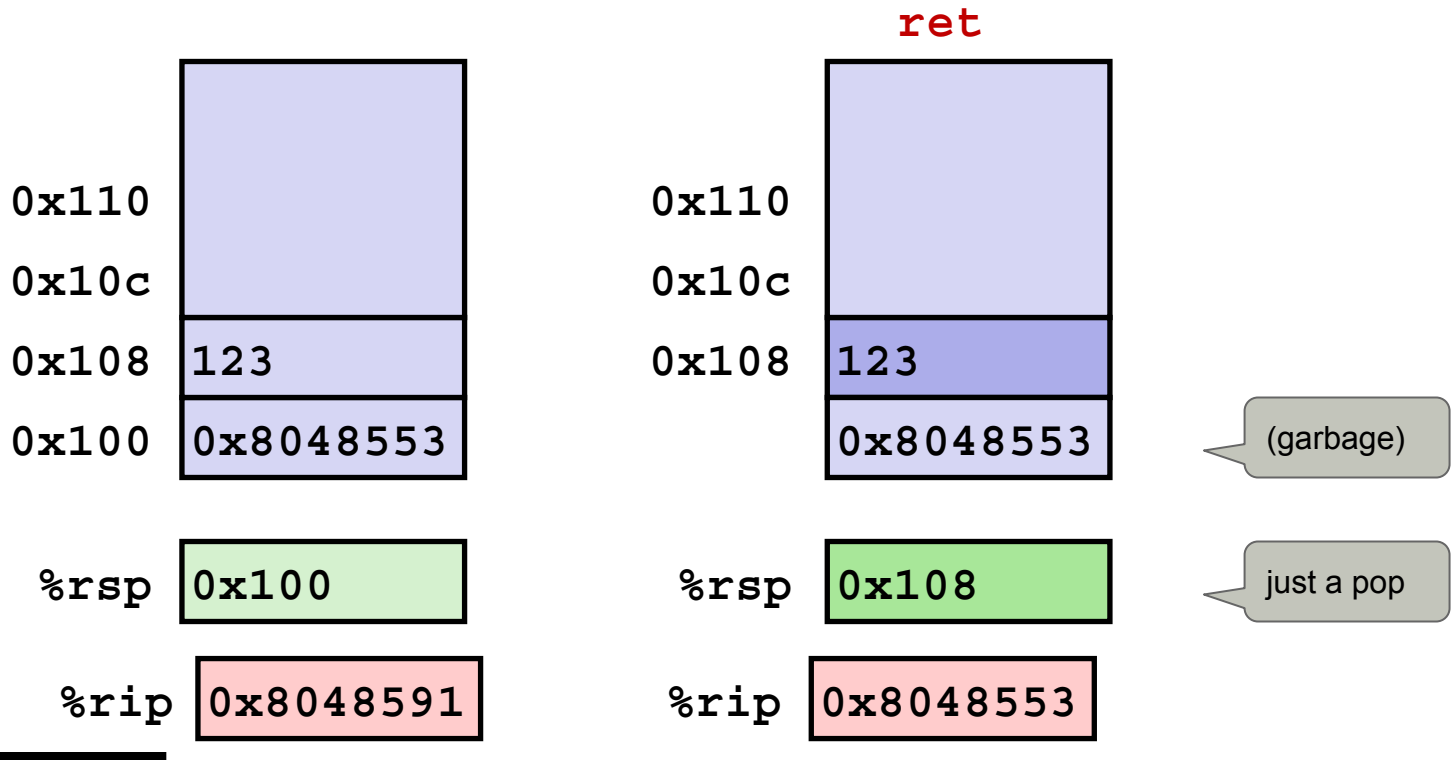
```
8048591:  c3                    ret
```



Q: what happens if user input overwrites stack pointer, or return address?

Procedure Call Example

```
8048591:  c3                    ret
```



Q: what happens if user input overwrites stack pointer, or return address?
A: user has full control over control flow. (exploit)

Calling a function (x86-64)

To call a function, a program:

1. **Places the first six integer or pointer parameters in %rdi, %rsi, %rdx, %rcx, %r8 and %r9**
2. Pushes onto the stack subsequent parameters and parameters larger than 8B (in order).
3. Executes the call instruction, which:
   - Pushes the return address onto the stack
   - Jumps to the start of the specified function

Executing a function

The C run-time system introduces instruction to set-up and clean-up the stack in each procedure.

Set-up consists in allocation and initialization of a stack-frame. Clean-up: deallocating a stack frame.

A stack-frame is the space needed on the stack by a procedure for storing:

- The return address
- (some) parameters
- Local variables

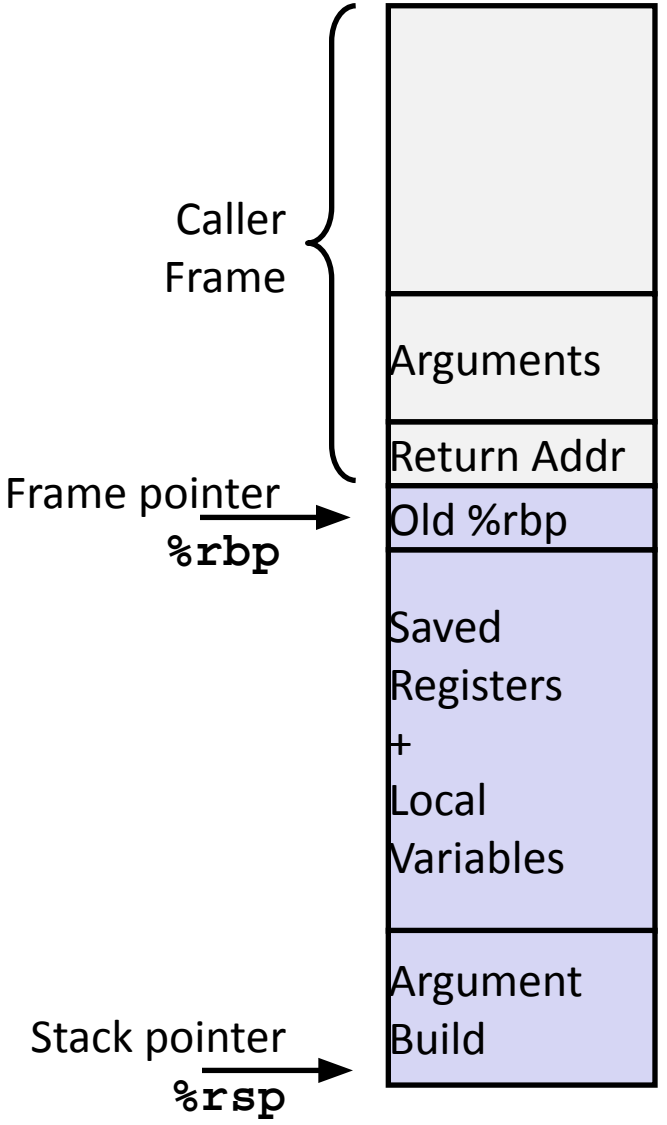
Stack Frame

Caller:

- Arguments
  - pushed by program (if needed)
- Return address
  - pushed by call

Callee:

- Previous frame pointer (%rbp)
- Other callee-save registers (%rbx, %r12-15)
- Space for local variables
- Arguments for next function (when about to call another function)



Example

arith.s | arith.c

```c
#include <stdio.h>
#include <stdlib.h>

int logical(int x, int y)
{
        int t1 = x^y;
        int t2 = t1 >> 17;
        int mask = (1<<13) - 7;
        int rval = t2 & mask;
        return rval;
}

int main(int argc, char* argv[])
{
        if (argc != 3) {
                printf("Usage: arith x y\n");
                return 1;
        }

        int x = atoi(argv[1]);
        int y = atoi(argv[2]);
        printf("Arguments x: %d, y: %d\n", x, y);
        printf("Logical returns: %d\n", logical(x,y));
        printf("\n");

        return 0;
}
```

Example

```
5  logical:
6  .LFB2:
7          .cfi_startproc
8          pushq   %rbp
9          .cfi_def_cfa_offset 16
10         .cfi_offset 6, -16
11         movq    %rsp, %rbp
12         .cfi_def_cfa_register 6
13         movl    %edi, -20(%rbp)
14         movl    %esi, -24(%rbp)
15         movl    -20(%rbp), %eax
16         xorl    -24(%rbp), %eax
17         movl    %eax, -16(%rbp)
18         movl    -16(%rbp), %eax
19         sarl    $17, %eax
20         movl    %eax, -12(%rbp)
21         movl    $8185, -8(%rbp)
22         movl    -12(%rbp), %eax
23         andl    -8(%rbp), %eax
24         movl    %eax, -4(%rbp)
25         movl    -4(%rbp), %eax
26         popq    %rbp
27         .cfi_def_cfa 7, 8
28         ret
29         .cfi_endproc
```

Set-up:

- Previous stack frame base %rbp pushed on stack
- %rbp is the only callee save register
- Frame pointer re-initialised

Function:

- 4 local variables at positions relative to stack frame base %rbp

  t1: -16(%rbp) t2: -12(%rbp)

  mask: -8(%rbp) rval: -4(%rbp)

  remember: stack grows down

- %eax holds intermediate results
- %eax holds return value at the end of the function

Clean-up:

- Previous stack frame base restored
- ret manipulates %rsp and %rip to return control to return address

Example

```
42 main:
43 .LFB3:
44         .cfi_startproc
45         pushq   %rbp
46         .cfi_def_cfa_offset 16
47         .cfi_offset 6, -16
48         movq    %rsp, %rbp
49         .cfi_def_cfa_register 6
50         subq    $32, %rsp
51         movl    %edi, -20(%rbp)
52         movq    %rsi, -32(%rbp)
53         cmpl    $3, -20(%rbp)
54         je      .L4
55         movl    $.LC0, %edi
56         call    puts
57         movl    $1, %eax
58         jmp     .L5
59 .L4:
60         movq    -32(%rbp), %rax
61         addq    $8, %rax
62         movq    (%rax), %rax
63         movq    %rax, %rdi
64         call    atoi
65         movl    %eax, -8(%rbp)
66         movq    -32(%rbp), %rax
67         addq    $16, %rax
68         movq    (%rax), %rax
69         movq    %rax, %rdi
70         call    atoi
71         movl    %eax, -4(%rbp)
72         movl    -4(%rbp), %edx
73         movl    -8(%rbp), %eax
74         movl    %eax, %esi
75         movl    $.LC1, %edi
76         movl    $0, %eax
77         call    printf
78         movl    -4(%rbp), %edx
79         movl    -8(%rbp), %eax
80         movl    %edx, %esi
81         movl    %eax, %edi
82         call    logical
83         movl    %eax, %esi
84         movl    $.LC2, %edi
85         movl    $0, %eax
86         call    printf
87         movl    $10, %edi
88         call    putchar
89         movl    $0, %eax
90 .L5:
91         leave
92         .cfi_def_cfa 7, 8
93         ret
94         .cfi_endproc
```

that's main (moving along)

Handled Without Special Consideration

- Stack frames mean that each function call has private storage
  - Saved registers & local variables
  - Saved return pointer
- Register saving conventions prevent one function call from corrupting another's data
- Stack discipline follows call / return pattern
  - If P calls Q, then Q returns before P
  - Last-In, First-Out

Take-aways

Reading assembly is useful (**performance**, **security**)

Stack used to orchestrate control flow between caller and callee

Stack frame contains caller args, return address, previous stack base pointer, other callee-save registers, and local variables.

Contents of stack frame accessed with register (%rbp) + offset addressing

Additional Slides

Instructions

Three classes of instructions:

1. Transfer between memory and register
   - Load/store data: register <-> memory
   - Push/pop: register <-> stack

2. **Arithmetic and comparison functions**

3. Transfer control
   - Jumps to/from procedures
   - Conditional branches

Arithmetic Functions

Same suffixes as mov:
b, w, l, q

- Unary: inc, dec, neg, not

  Example: incl %r10
- Binary: add, sub, imul, xor, or, and

  Form: OP SRC, DEST => DEST = DEST OP SRC

  => DEST OP= SRC

  Example: addq -8, %rsp

Arithmetic Functions

- Shift operations: sal/shl, sar/shr

  s – shift; a – arithmetic; h – logical; r – right; l - left

  Form: OP k, DEST => shift DEST by k bits
- Special arithmetic:
  - imulq SRC – signed multiply of %rax by SRC
    result stored in %rdx:%rax
  - mulq SRC – unsigned multiply of %rax by SRC
    result stored in %rdx:%rax
  - idivq SRC – signed divide %rdx:%rax by SRC
    result stored in %rdx
  - divq SRC – unsigned divide of %rdx:%rax by SRC
    result stored in %rdx

Arithmetic Functions

- Load effective address: leaq
  - Form: leverages addressing modes to compute arithmetic functions
  - Example:

```
leal (%eax,%eax,2), %eax  ;x <- x+x*2
```

Instructions

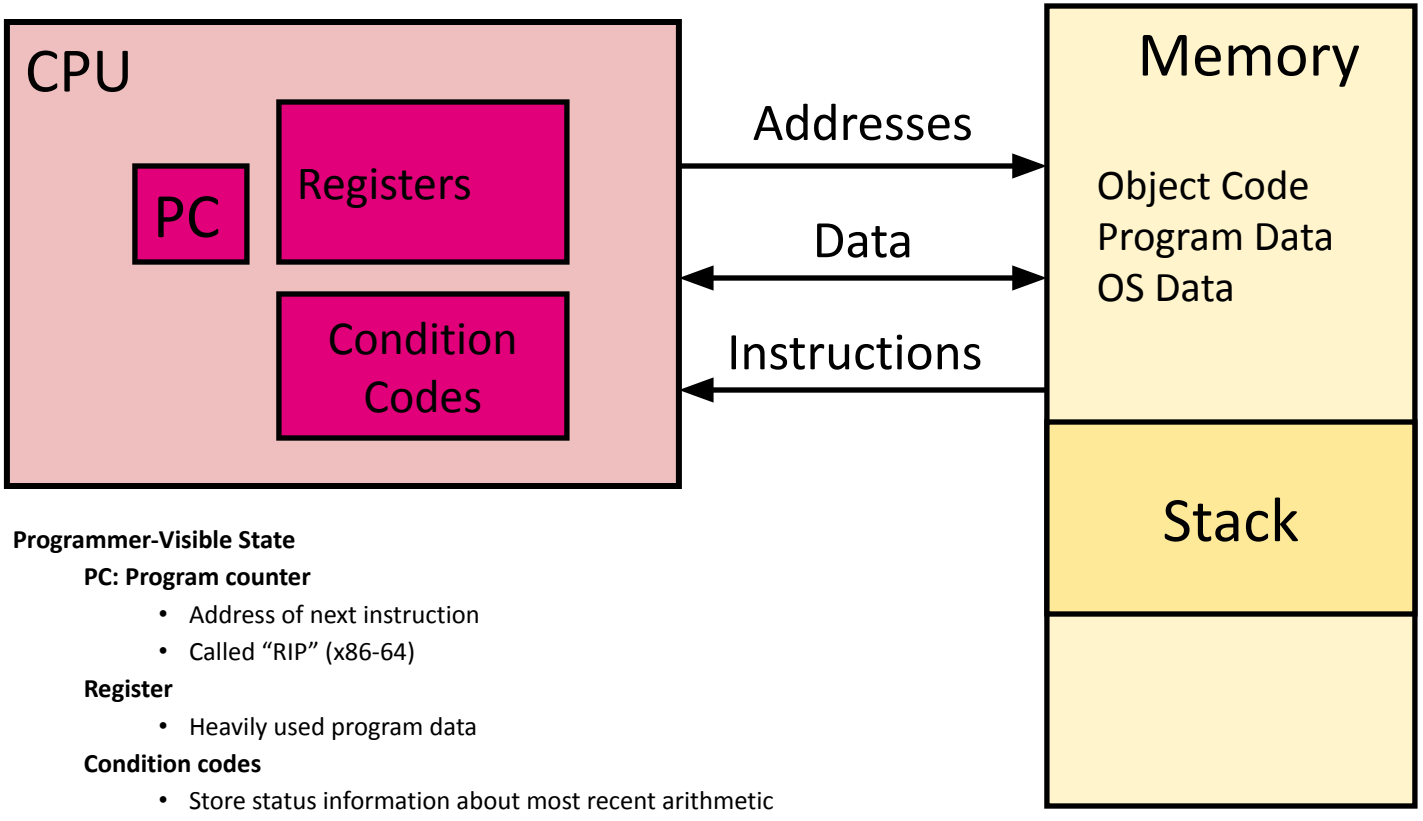
Three classes of instructions:

1. Transfer between memory and register
   - Load/store data: register <-> memory
   - Push/po: register <-> stack

2. Arithmetic and comparison functions

3. Transfer control
   - Jumps to/from procedures
   - **Conditional branches**

X86-64 Assembly



Programmer-Visible State

PC: Program counter

- Address of next instruction
- Called “RIP” (x86-64)

Register

- Heavily used program data

Condition codes

- Store status information about most recent arithmetic operation
- Used for conditional branching

Memory

- Byte addressable array
- Code, user data, (some) OS data
- Includes stack used to support procedures

Condition Codes

1. ZF – result was zero
2. CF – result caused Carry out of most significant bit
3. SF - result was negative (sign bit was set)
4. OF – result caused overflow

Setting Condition Codes

- Comparison: cmp S2, S1

  Set condition codes based on S1-S2

- Test: test S2, S1

  Set condition codes based on S1 & S2

Accessing Condition Codes

https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf

Bringing up **low byte** value 0x0 or 0x1 in register D (e.g., %al or %r10b)

Instruction		Description	Condition Code
sete / **setz**	*D*	Set if equal/zero	**ZF**
setne / **setnz**	*D*	Set if not equal/nonzero	**~ZF**
sets	*D*	Set if negative	**SF**
setns	*D*	Set if nonnegative	**~SF**
setg / **setnle**	*D*	Set if greater (signed)	**~(SF^0F)&~ZF**
setge / **setnl**	*D*	Set if greater or equal (signed)	**~(SF^0F)**
setl / **setnge**	*D*	Set if less (signed)	**SF^0F**
setle / **setng**	*D*	Set if less or equal	**(SF^OF)|ZF**
seta / setnbe	*D*	Set if above (unsigned)	**~CF&~ZF**
setae / **setnb**	*D*	Set if above or equal (unsigned)	**~CF**
setb / **setnae**	*D*	Set if below (unsigned)	**CF**
setbe / **setna**	*D*	Set if below or equal (unsigned)	**CF|ZF**

Example

```
cmpl %eax, %edx
sete %al
movsbq %al, %rax
```

Jump instructions

Instruction		Description	Condition Code
jmp	*Label*	Jump to label	
jmp	**Operand*	Jump to specified location	
je / jz	*Label*	Jump if equal/zero	**ZF**
jne / jnz	*Label*	Jump if not equal/nonzero	**~ZF**
js	*Label*	Jump if negative	**SF**
jns	*Label*	Jump if nonnegative	**~SF**
jg / jnle	*Label*	Jump if greater (signed)	**~(SF^0F)&~ZF**
jge / jnl	*Label*	Jump if greater or equal (signed)	**~(SF^0F)**
jl / jnge	*Label*	Jump if less (signed)	**SF^0F**
jle / jng	*Label*	Jump if less or equal	**(SF^OF)|ZF**
ja / jnbe	*Label*	Jump if above (unsigned)	**~CF&~ZF**
jae / jnb	*Label*	Jump if above or equal (unsigned)	**~CF**
jb / jnae	*Label*	Jump if below (unsigned)	**CF**
jbe / jna	*Label*	Jump if below or equal (unsigned)	**CF|ZF**