

```
29 #include "balloon.h"
30
31 int struct {
32     ScePspFVector3 mode;
33     ScePspFVector3 pos;
34     int sbuf[3];
35     float scnt;
36     t;
37 } BALLOONDAT;
38
39 static BALLOONDAT balloon;
40 static ScePspFVector3 sphere[28];
41 static ScePspFVector3 pole[28];
42
43 extern void DrawSphere(ScePspFVector3 *array, float r);
44 extern void DrawPole(ScePspFVector3 *array, float r);
45
46 void init_balloon(void)
47 {
48     int i;
49
50     balloon.mode=MODE;
51     balloon.pos.x= 0;
52     balloon.pos.y=-8;
53     balloon.pos.z= 0;
54     balloon.t=0.0f;
55     balloon.scnt=2;
56
57     for (i=0; i<3; i++)
58     {
59         balloon.sbuf[i]=
60         balloon.sbuf[i]
61         balloon.sbuf[i]
62     }
63
64 void draw_balloon(void)
65 {
66     ScePspFVector3 vec;
67     cable(SCEGU_TEXTURE);
68     (); balloon.pos);
69 }
```

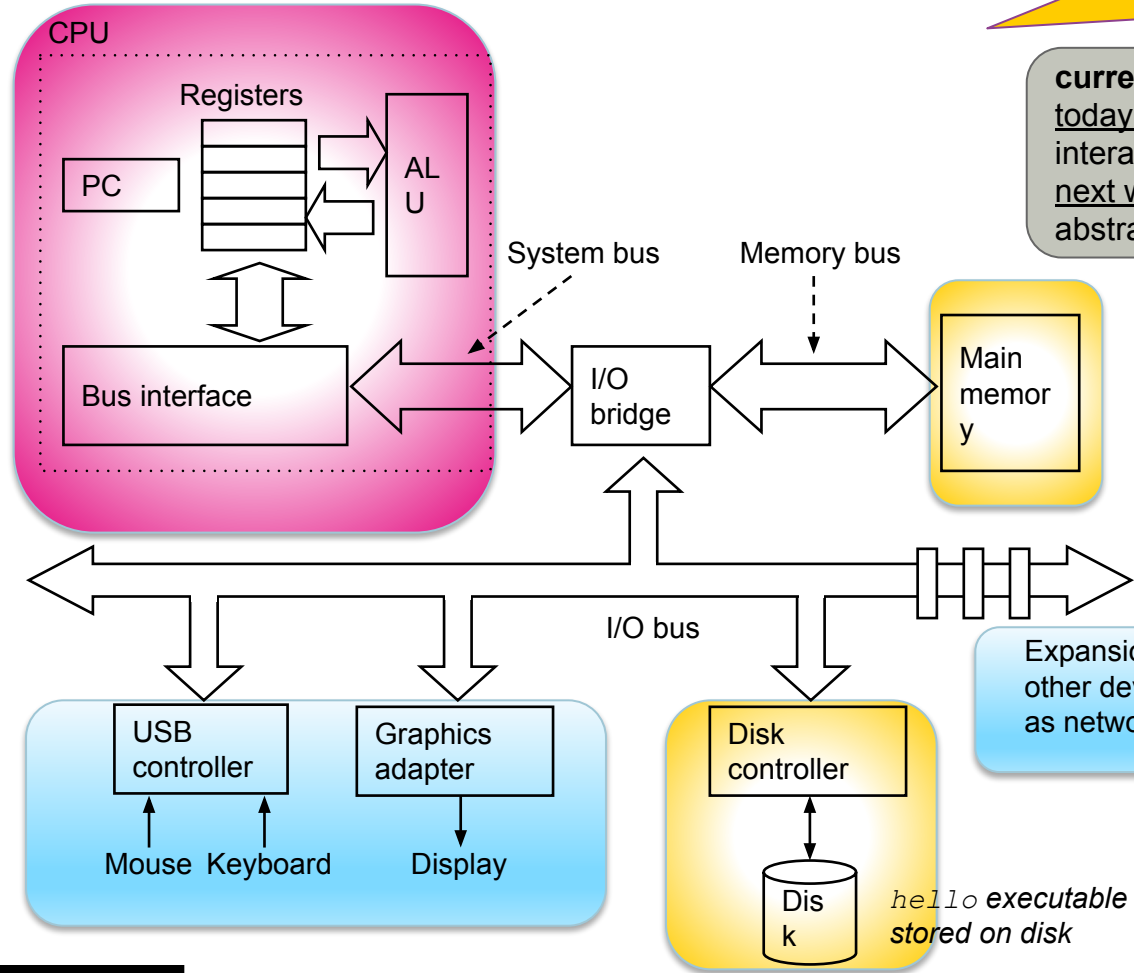
Operating Systems and C Fall 2022 3. Representing Instructions

Computer Hardware

recall fundamental abstractions:

- interpreter,
- memory,
- communication

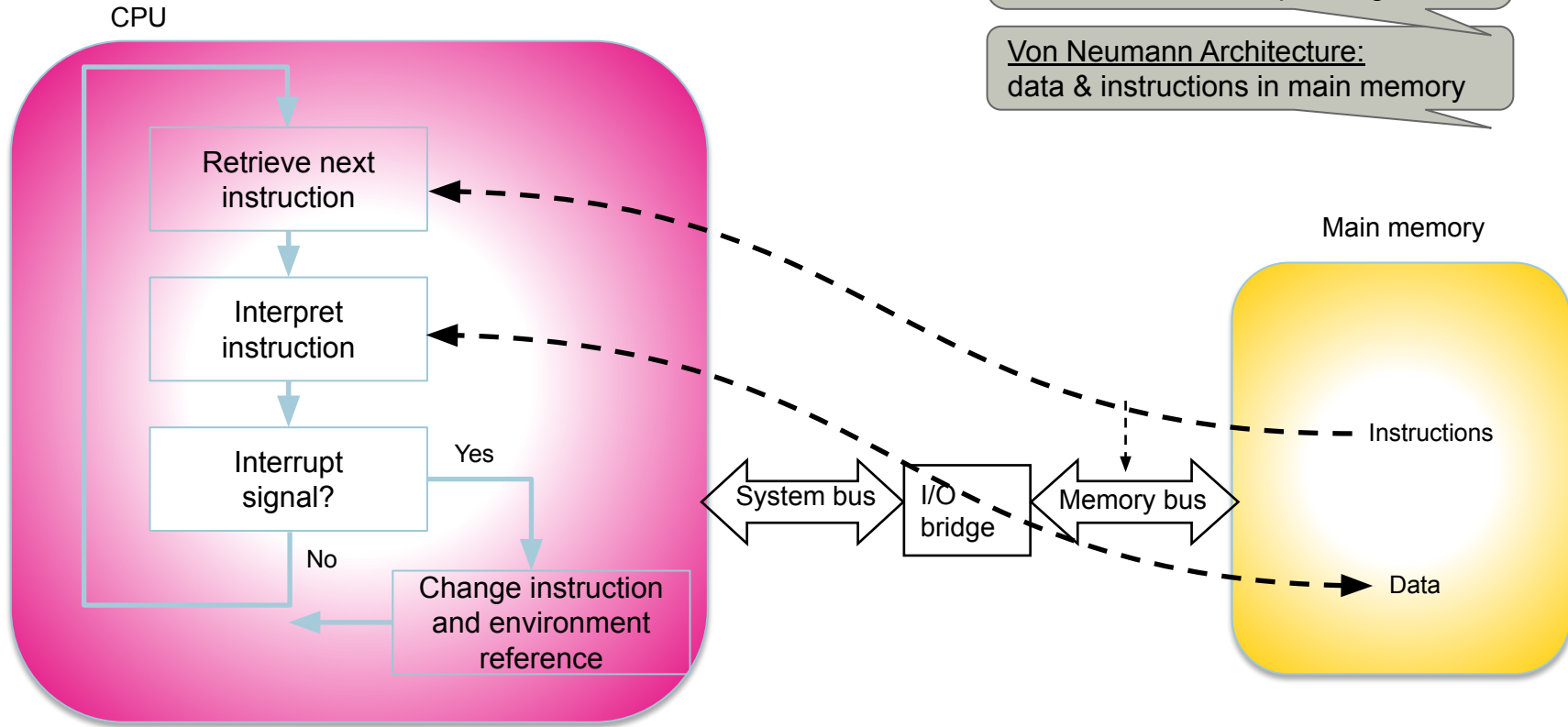
current focus: interpreter.
today: how interpreter interacts w/ main memory.
next week: how interpreter abstraction is implemented.



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:**

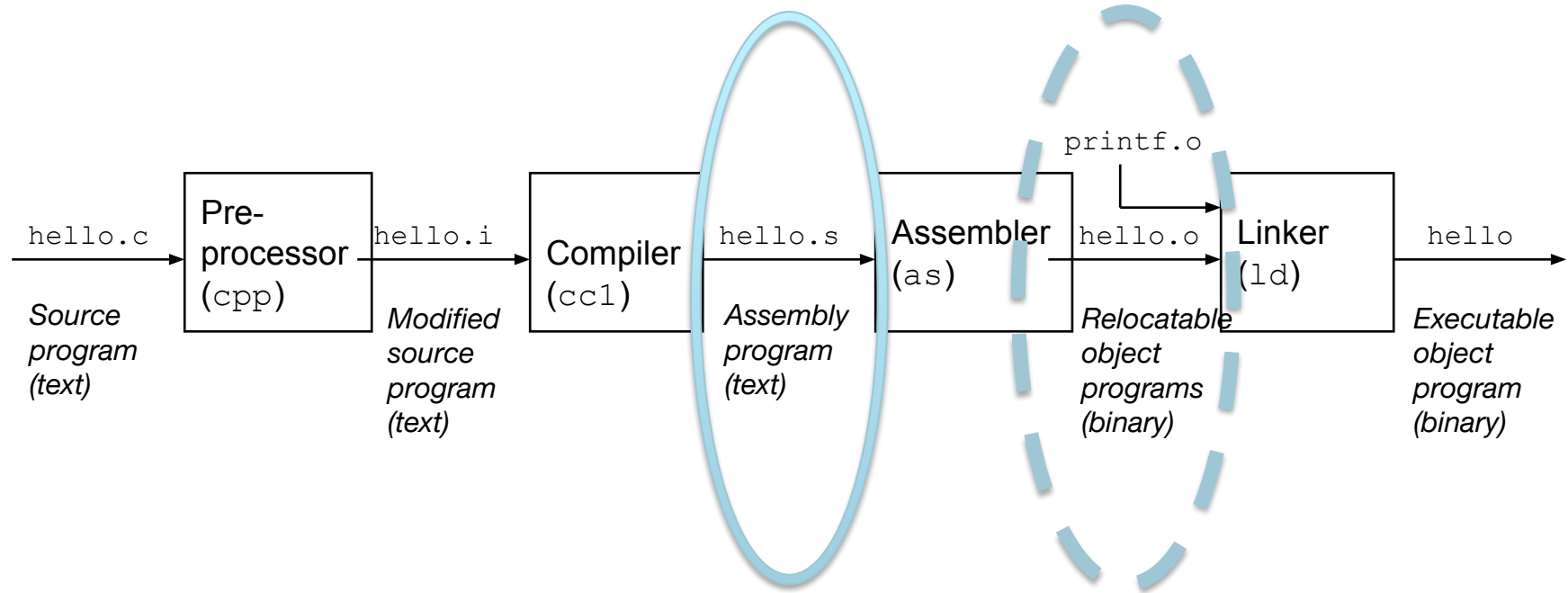
- Sequence of bits
- Boolean
- Integer (Two's complement)
- Float ([IEEE 754](#)) – Interested? Check out [posit](#)
- Characters – ASCII (1B), UTF-8 (1-4B)
- Strings – Array of characters terminated by '\0'

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

Compilation phases

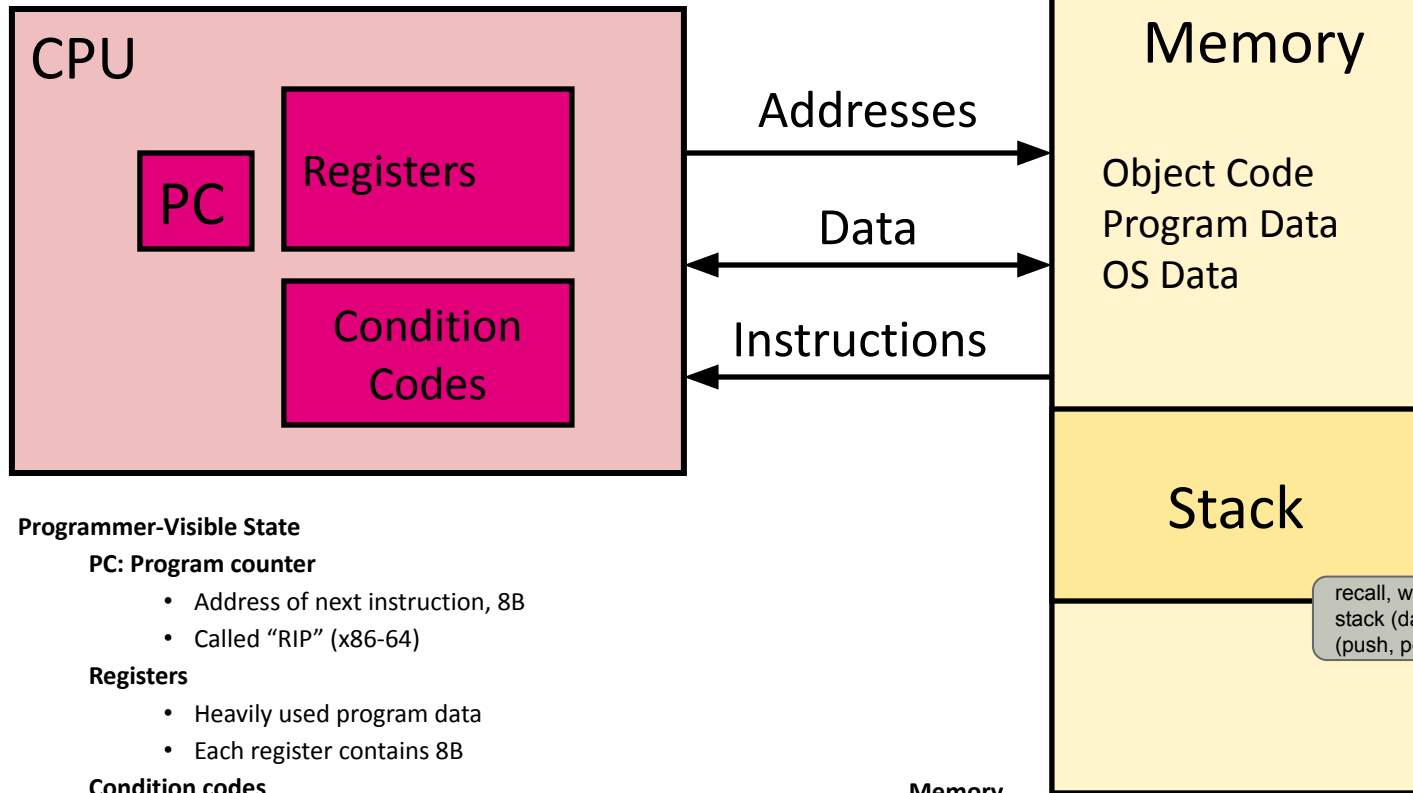
<https://github.com/gcc-mirror/gcc>

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

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

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

Instructions (**Outline** for today)

Three classes of instructions:

1. Transfer between memory and register

- Load/store data: register \leftrightarrow memory
- Push/pop: register \leftrightarrow 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

How are registers organized?

The **%rip** register is the current **instruction pointer**.
Contains address of next instruction to be executed.

most instructions implicitly increment it.
explicitly updated \Rightarrow 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
#include <stdio.h>
2
3 int main()
4 {
5     int age = 10;
6     int height = 152;
7
8     printf("I am %d years old, and %d cm high.\n", age, height);
9
10    return 1;;
11 }
```

```
$ gcc -S -o ex3.s ex3.c
$ vi ex3.s
```

```
ex3.s
1  .file "ex3.c"
2  .section .rodata
3  .align 8
4  .LC0:
5  .string "I am %d years old, and %d cm high.\n"
6  .text
7  .globl main
8  .type main, @function
9  main:
10 .LFB0:
11 .cfi_startproc
12 pushq %rbp
13 .cfi_def_cfa_offset 16
14 .cfi_offset 6, -16
15 movq %rsp, %rbp
16 .cfi_def_cfa_register 6
17 subq $16, %rsp
18 movl $10, -8(%rbp)
19 movl $152, -4(%rbp)
20 movl -4(%rbp), %edx
21 movl -8(%rbp), %eax
22 movl %eax, %esi
23 movl $.LC0, %edi
24 movl $0, %eax
25 call printf
26 movl $1, %eax
27 leave
28 .cfi_def_cfa 7, 8
29 ret
30 .cfi_endproc
31 .LFE0:
32 .size main, .-main
33 .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 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

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
```

- 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
#include <stdio.h>
2
3 int main()
4 {
5     int age = 10;
6     int height = 152;
7
8     printf("I am %x years old, and %d cm high.\n", age, height);
9
10    return 1;;
11 }
```

```
$ gcc -S -o ex3.s ex3.c
$ vi ex3.s
```

```
ex3.s
1 .file "ex3.c"
2 .section .rodata
3 .align 8
4 .LC0:
5 .string "I am %x years old, and %d cm high.\n"
6 .text
7 .globl main
8 .type main, @function
9 main:
10 .LFB0:
11 .cfi_startproc
12 pushq %rbp
13 .cfi_def_cfa_offset 16
14 .cfi_offset 6, -16
15 movq %rsp, %rbp
16 .cfi_def_cfa_register 6
17 subq $16, %rsp
18 movl $10, -8(%rbp)
19 movl $152, -4(%rbp)
20 movl -4(%rbp), %edx
21 movl -8(%rbp), %eax
22 movl %eax, %esi
23 movl $.LC0, %edi
24 movl $0, %eax
25 call printf
26 movl $1, %eax
27 leave
28 .cfi_def_cfa 7, 8
29 ret
30 .cfi_endproc
31 .LFE0:
32 .size main,.-main
33 .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

“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!

How are procedure calls organized?

```
arith.c
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int logical(int x, int y)
5 {
6     int t1 = x^y;
7     int t2 = t1 >> 17;
8     int mask = (1<<13) - 7;
9     int rval = t2 & mask;
10    return rval;
11 }
12
13 int main(int argc, char* argv[])
14 {
15     if (argc != 3) {
16         printf("Usage: arith x y\n");
17         return 1;
18     }
19
20     int x = atoi(argv[1]);
21     int y = atoi(argv[2]);
22     printf("Arguments x: %d, y: %d\n", x, y);
23     printf("Logical returns: %d\n", logical(x,y));
24     printf("\n");
25
26     return 0;
27 }
```

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)

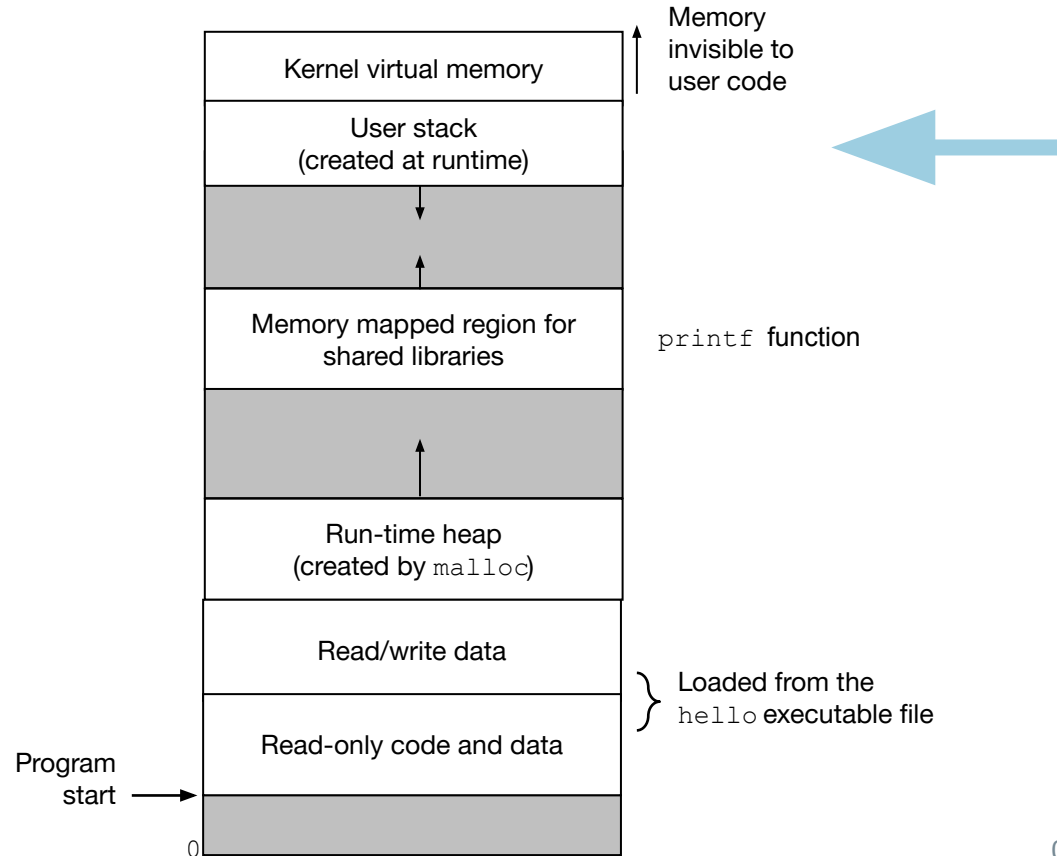
Return value is in rax

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

What is the stack?
How is the stack used?

Virtual Memory

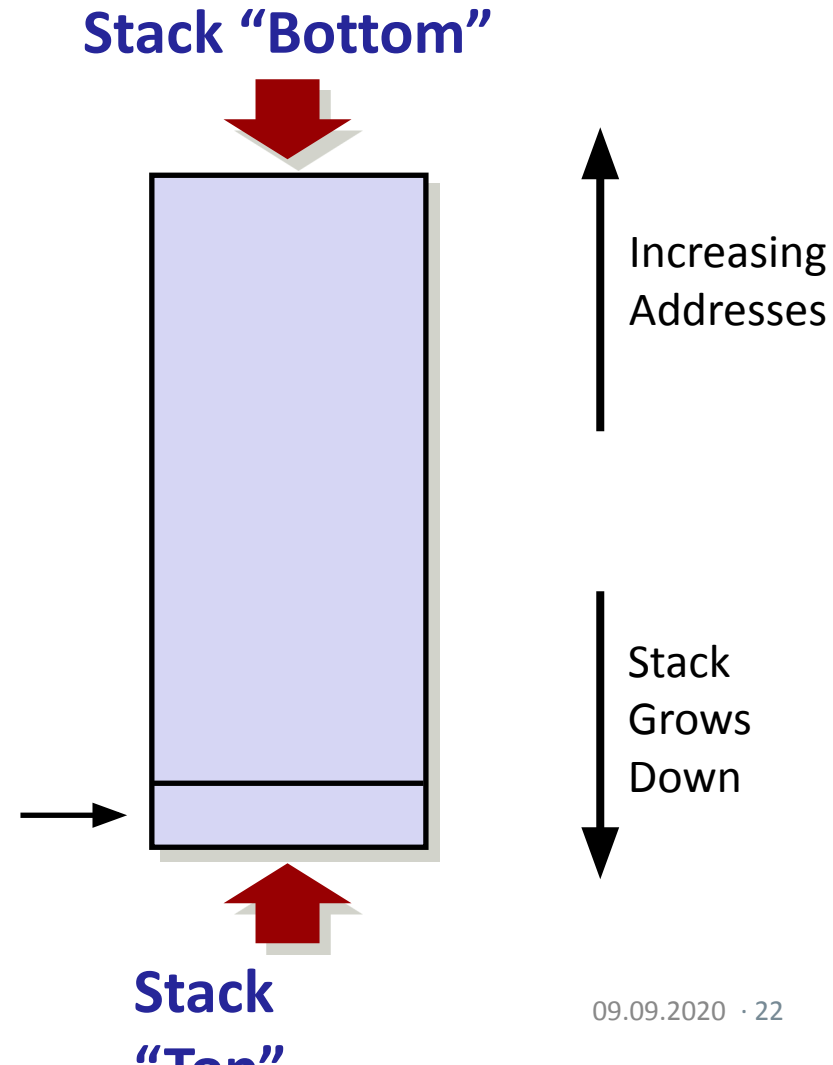
What is the stack?



The Stack

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

Stack Pointer: `%rsp`

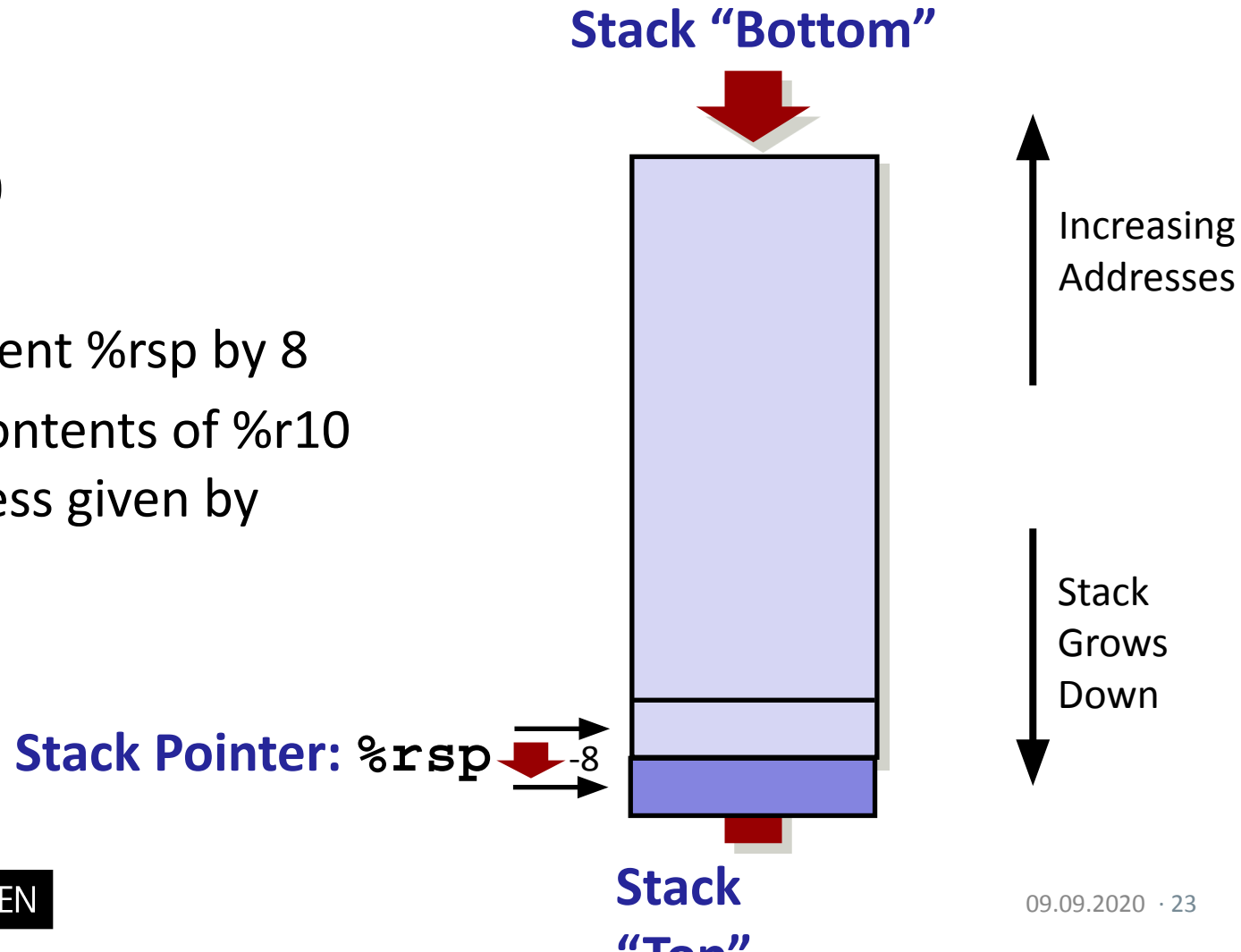


Stack - Push

Operator push

```
pushq %r10
```

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

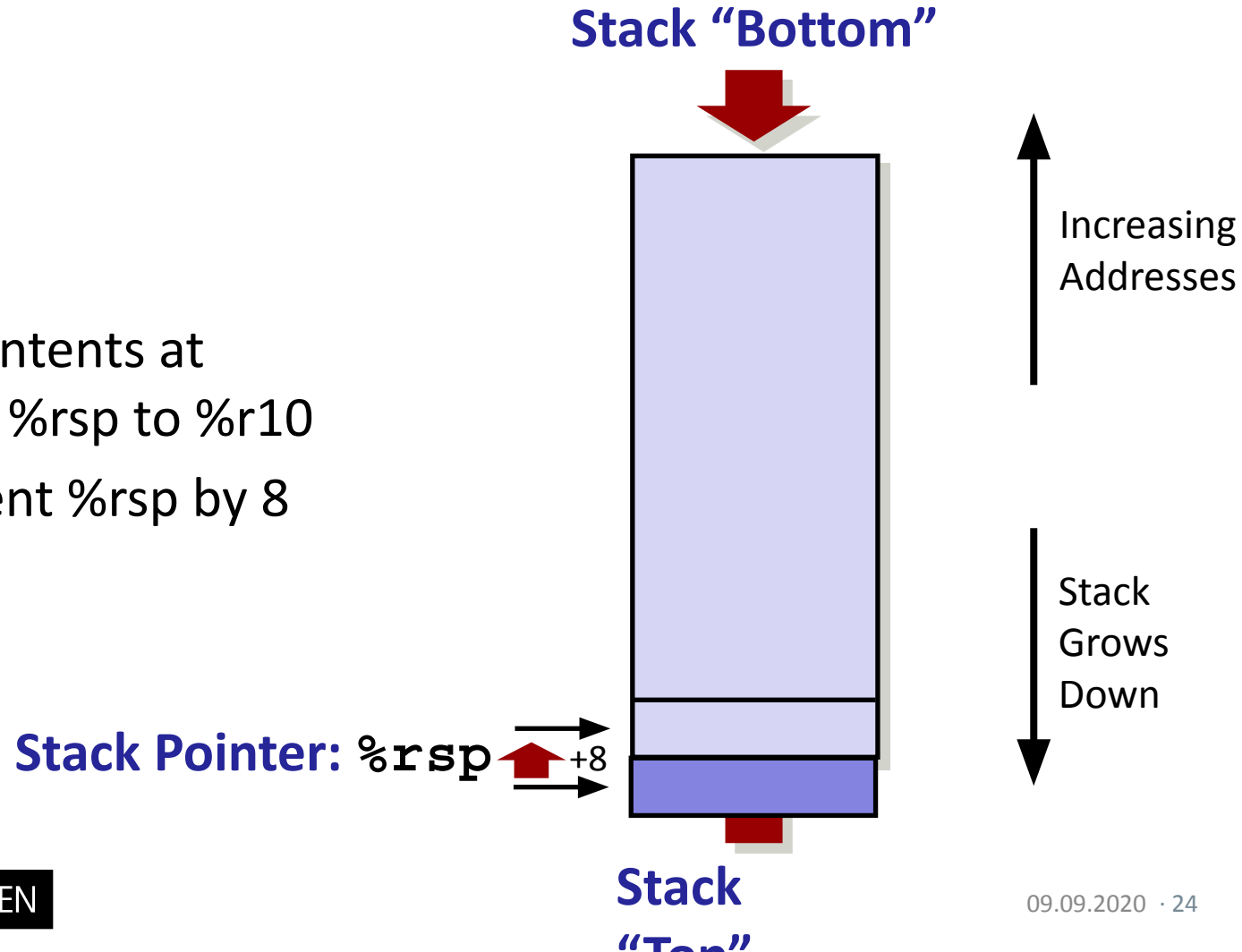


Stack - Pop

Operator pop

```
popq %r10
```

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



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?

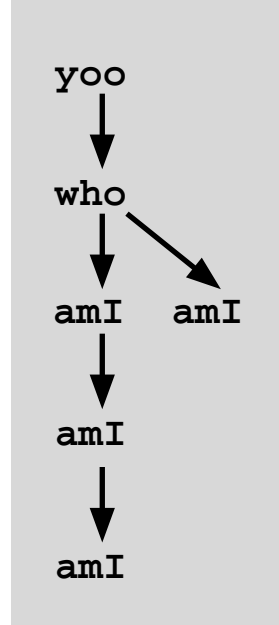
```
yoo (...)  
{  
  .  
  .  
  who ();  
  .  
  .  
}
```

```
who (...)  
{  
  . . .  
  amI ();  
  . . .  
  amI ();  
  . . .  
}
```

```
amI (...)  
{  
  .  
  .  
  amI ();  
  .  
  .  
}
```

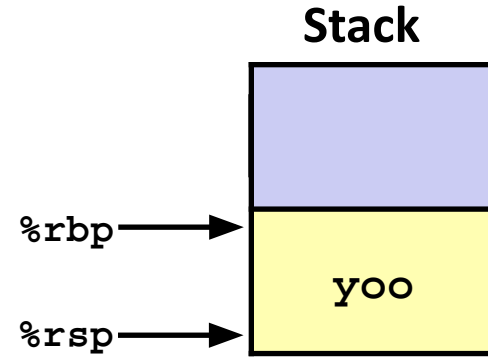
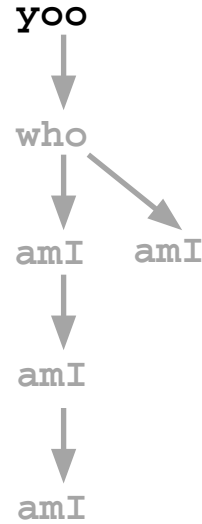
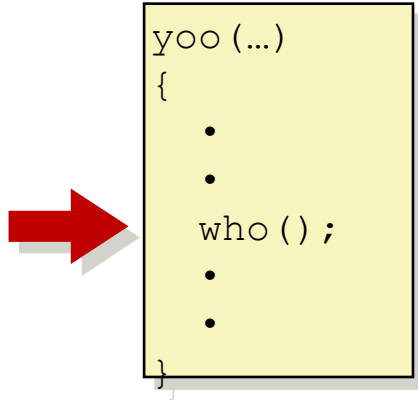
Procedure amI () is recursive

Example Call Chain



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

Example



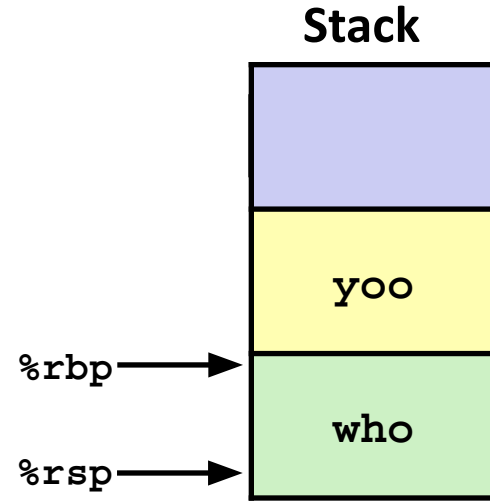
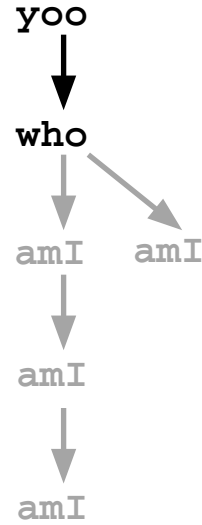
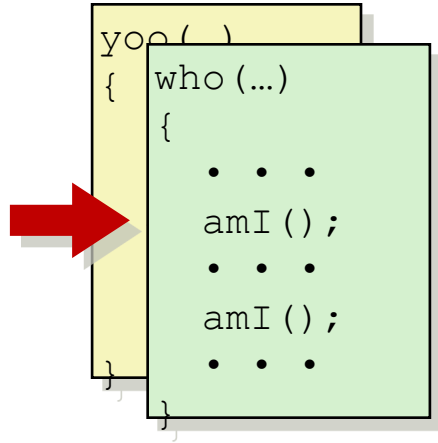
stack frame

- caller's saved registers⁽¹⁾
- local vars
- args
- return address⁽²⁾

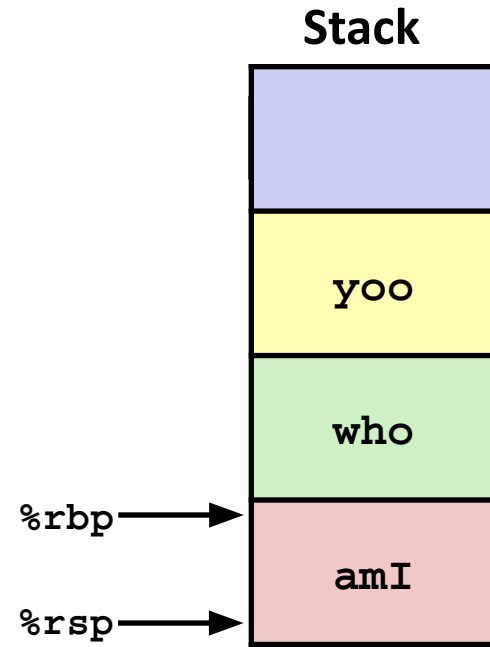
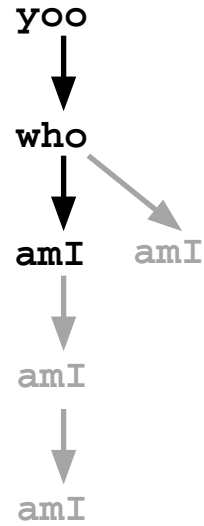
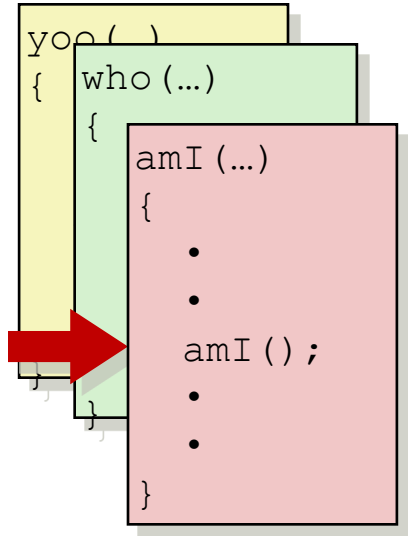
⁽¹⁾: to restore caller state on return
(bottom frame has no caller).

⁽²⁾: pushes it when it calls
(top frame does not need this)

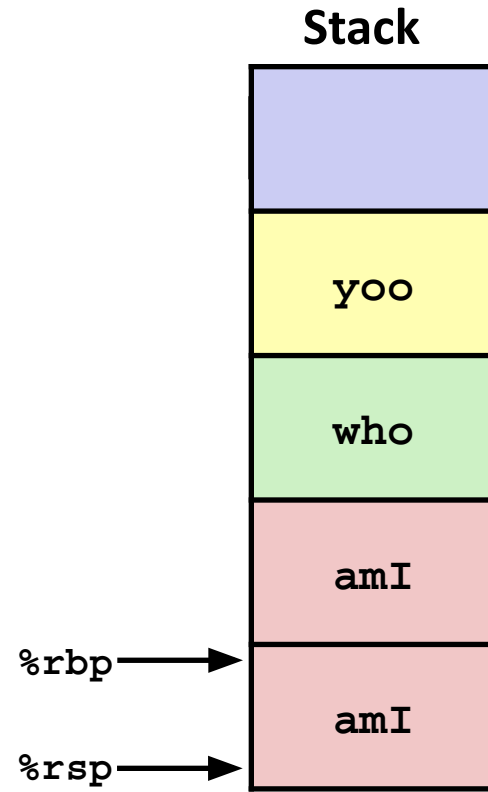
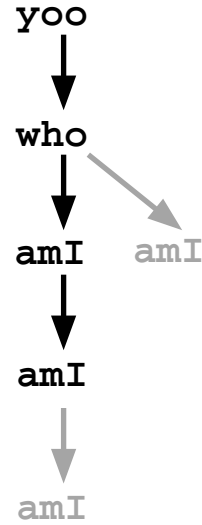
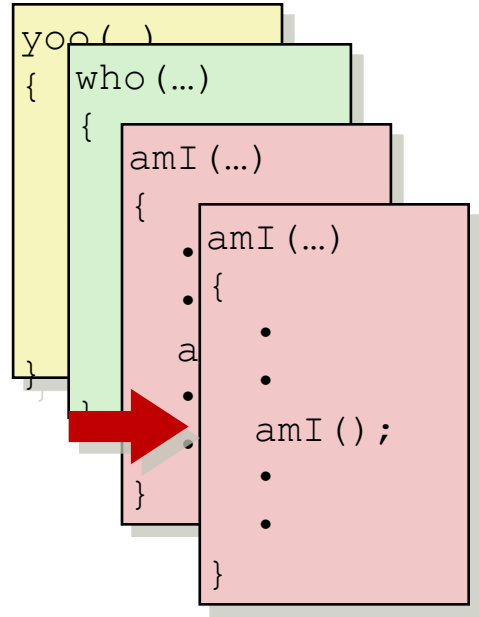
Example



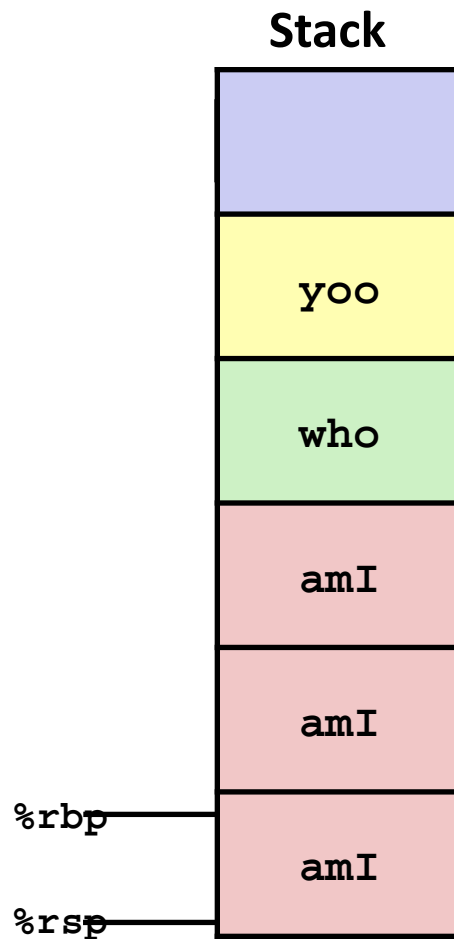
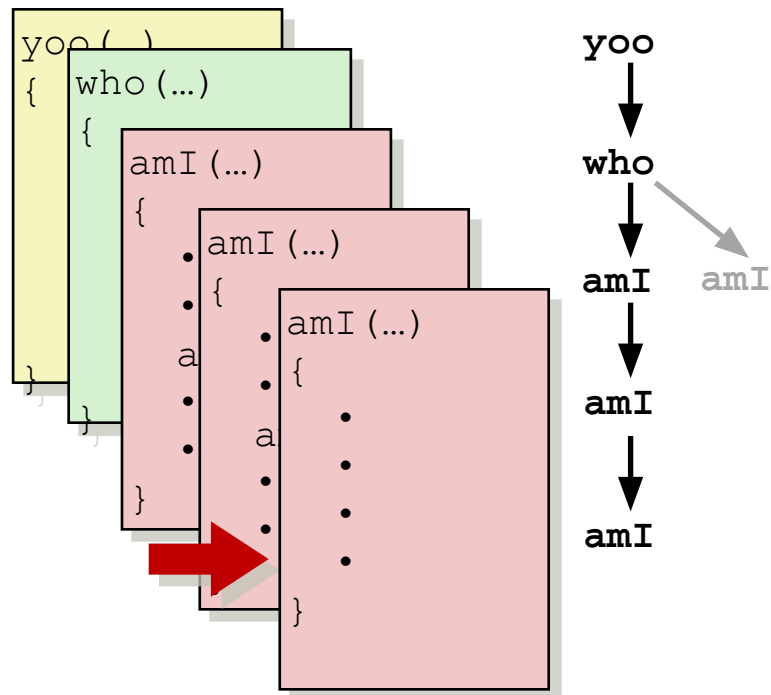
Example



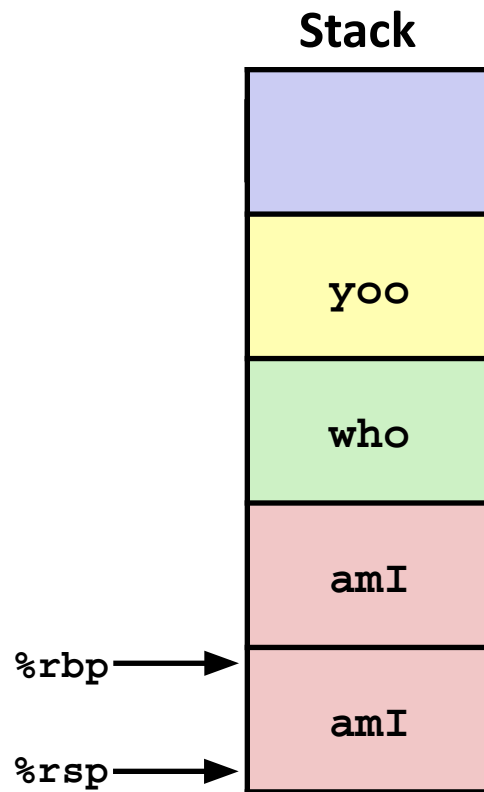
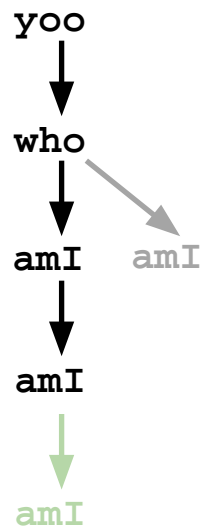
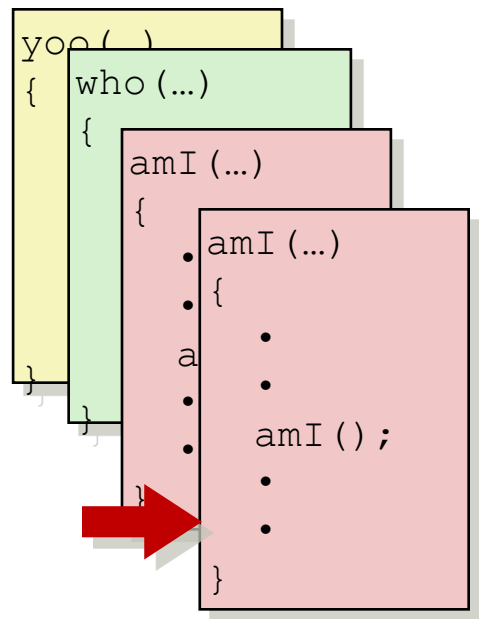
Example



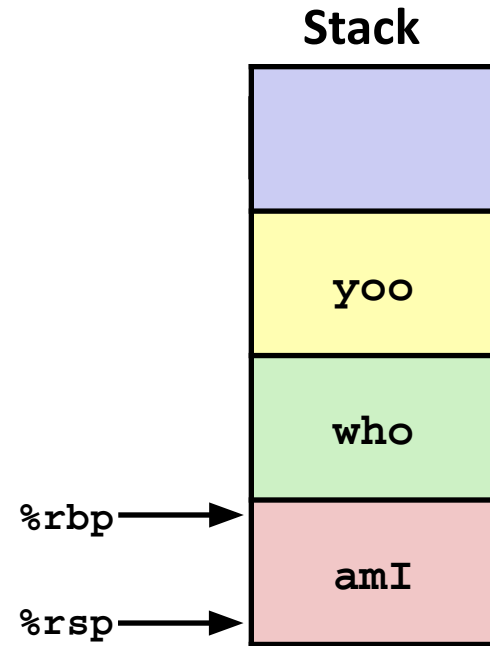
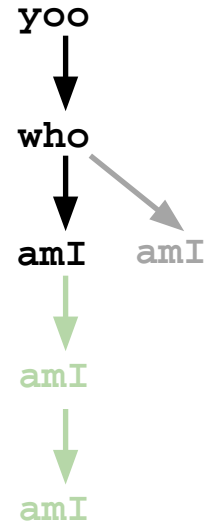
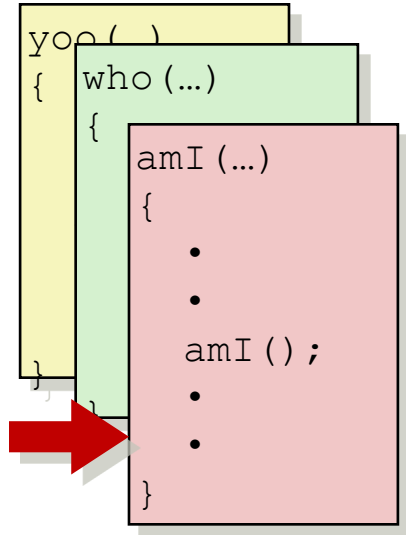
Example



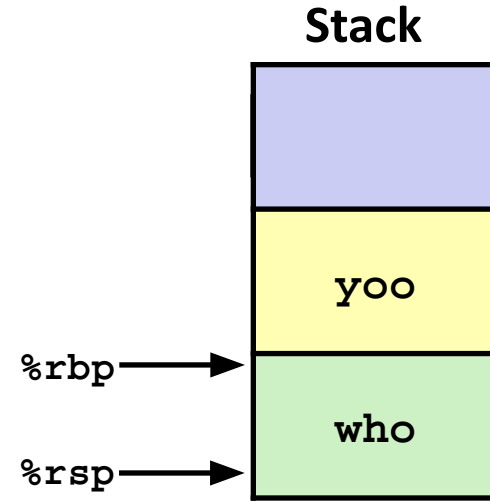
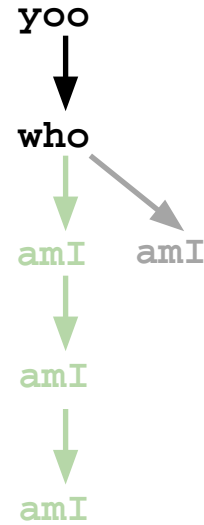
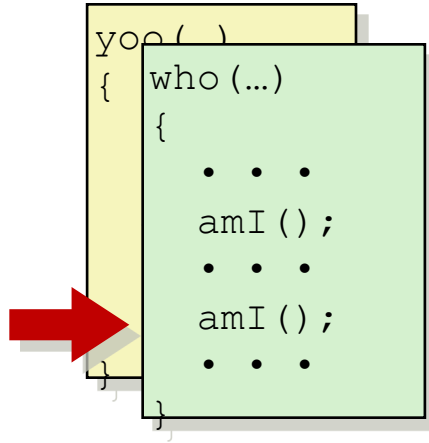
Example



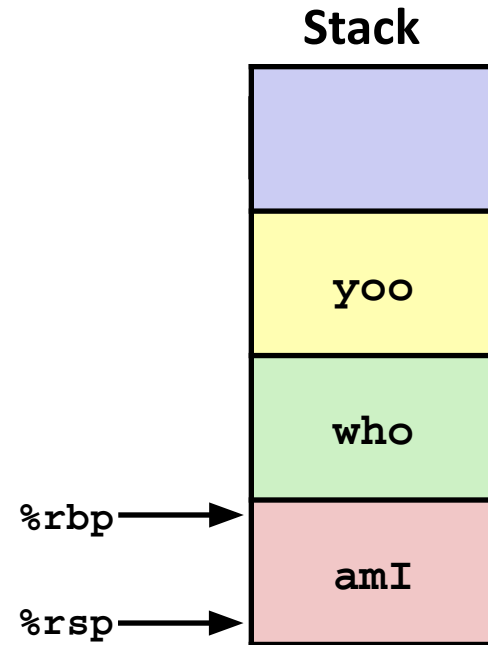
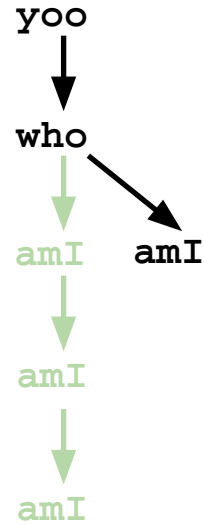
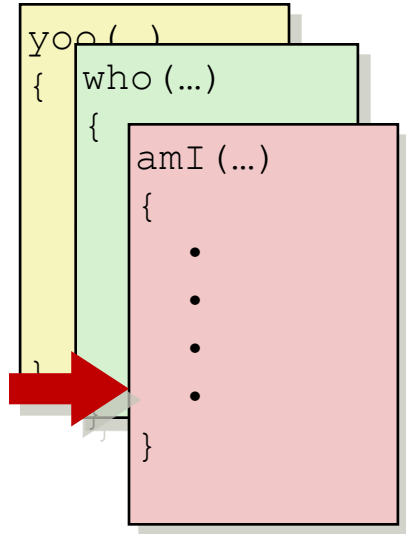
Example



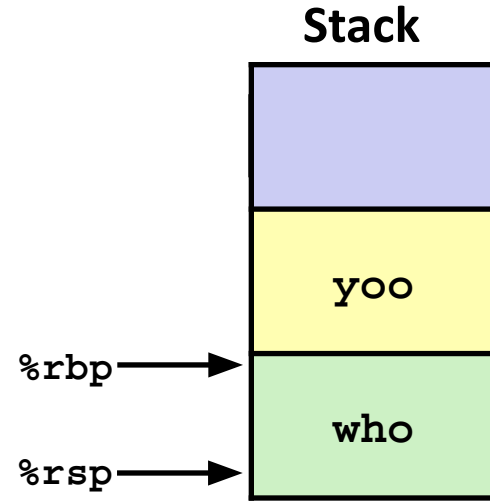
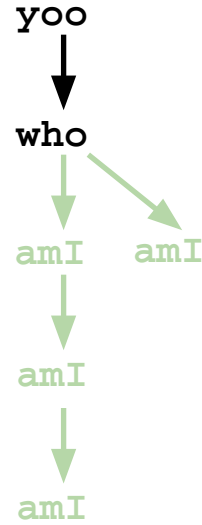
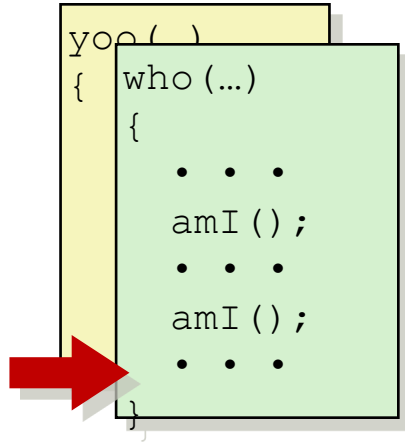
Example



Example

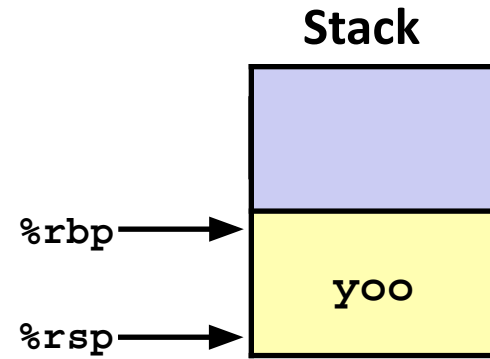
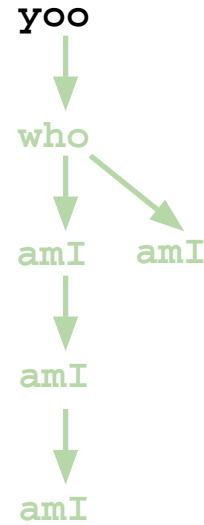



Example



Example

```
yoo (...)  
{  
  .  
  .  
  who ();  
  .  
  .  
}
```



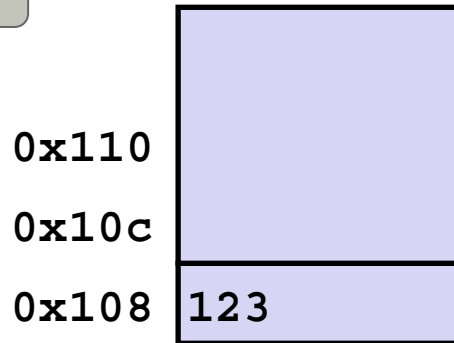
Instructions:

- **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
```

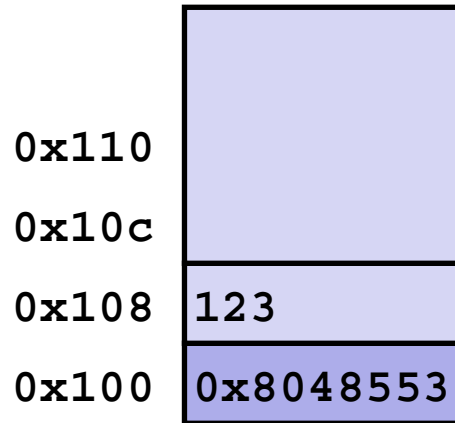
return address
= address of next instruction



%rsp 0x108

%rip 0x80854e

call 8048b90



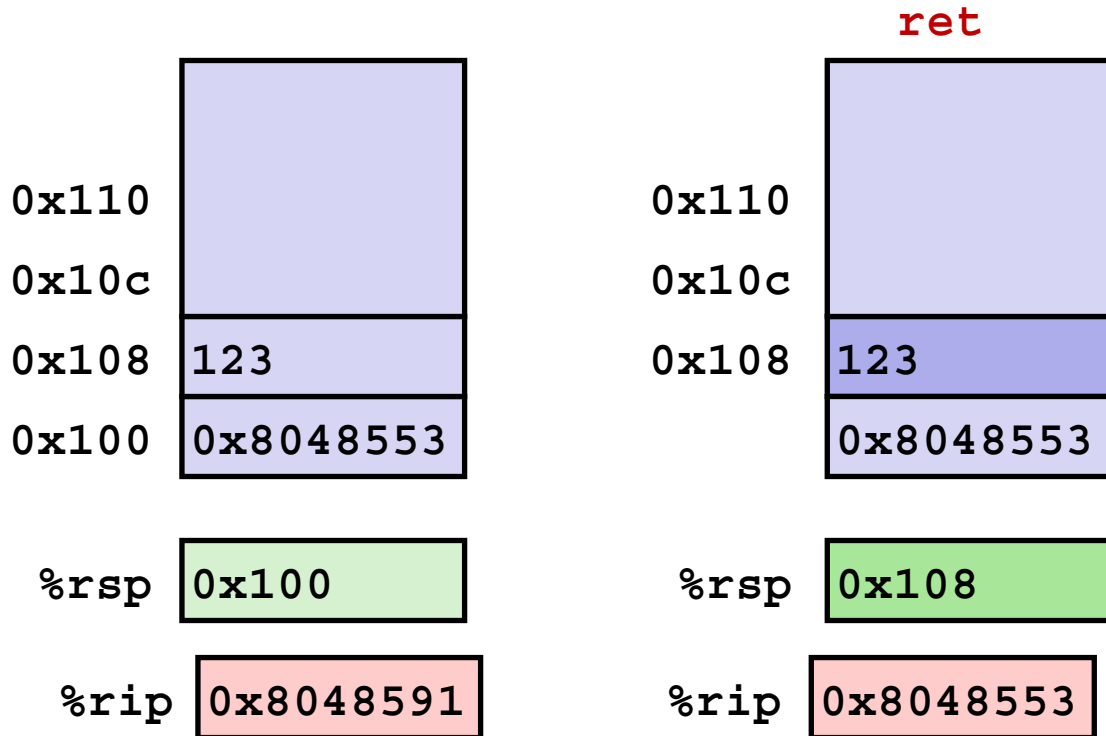
%rsp 0x100

%rip 0x8048b90

instruction pointer
updated to callee

Procedure Call Example

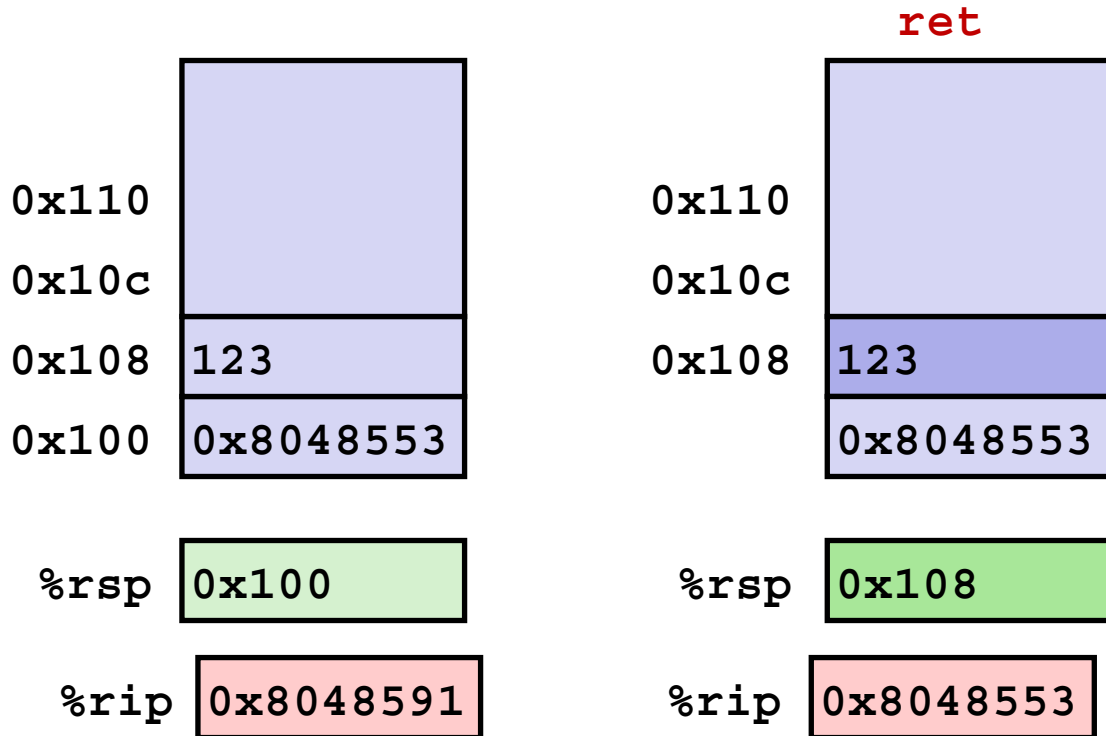
```
8048591: c3                ret
```



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)

(garbage)

just a pop

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

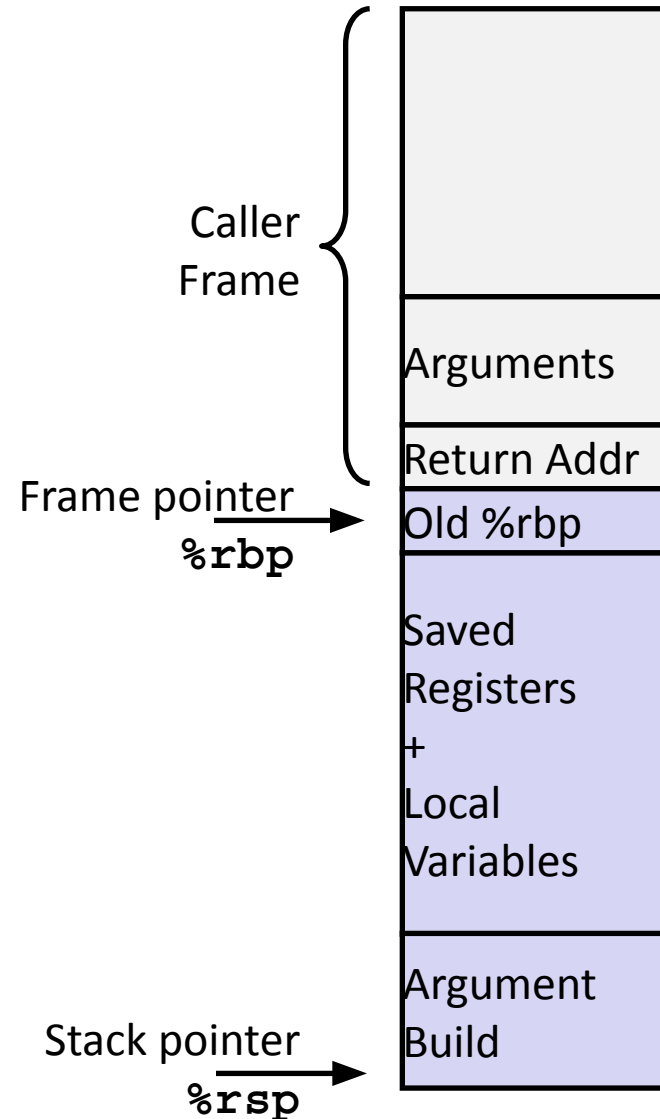
Stack Frame

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
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int logical(int x, int y)
5 {
6     int t1 = x^y;
7     int t2 = t1 >> 17;
8     int mask = (1<<13) - 7;
9     int rval = t2 & mask;
10    return rval;
11 }
12
13 int main(int argc, char* argv[])
14 {
15     if (argc != 3) {
16         printf("Usage: arith x y\n");
17         return 1;
18     }
19
20     int x = atoi(argv[1]);
21     int y = atoi(argv[2]);
22     printf("Arguments x: %d, y: %d\n", x, y);
23     printf("Logical returns: %d\n", logical(x,y));
24     printf("\n");
25
26     return 0;
27 }
```

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

remember:
stack
grows
down

Clean-up:

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

Example

```
41 main:                                     .type main, @function
42 .LFB3:
43 .cfi_startproc
44 pushq %rbp
45 .cfi_def_cfa_offset 16
46 .cfi_offset 6, -16
47 movq %rsp, %rbp
48 .cfi_def_cfa_register 6
49 subq $32, %rsp
50 movl %edi, -20(%rbp)
51 movq %rsi, -32(%rbp)
52 cmpl $3, -20(%rbp)
53 je .L4
54 movl $.LC0, %edi
55 call puts
56 movl $1, %eax
57 jmp .L5
58
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 \leftrightarrow memory
 - Push/pop: register \leftrightarrow 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:

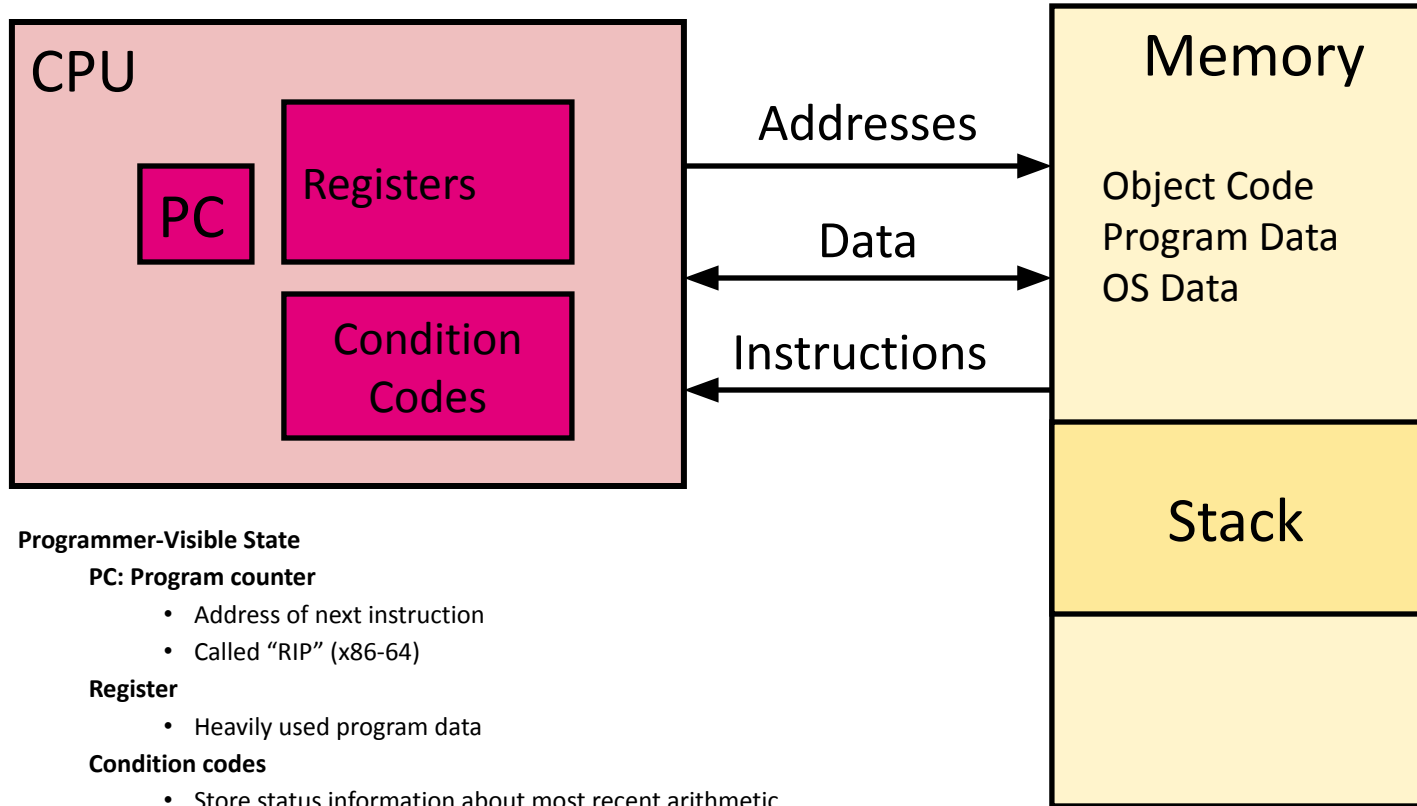
```
leal (%eax,%eax,2), %eax ;x <- x+x*2
```

Instructions

Three classes of instructions:

1. Transfer between memory and register
 - Load/store data: register \leftrightarrow memory
 - Push/pop: register \leftrightarrow 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

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

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

Example

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

Jump instructions

	Instruction	Description	Condition Code
	jmp <i>Label</i>	Jump to label	
	jmp <i>*Operand</i>	Jump to specified location	
	je / jz <i>Label</i>	Jump if equal/zero	ZF
	jne / jnz <i>Label</i>	Jump if not equal/nonzero	~ZF
	js <i>Label</i>	Jump if negative	SF
	jns <i>Label</i>	Jump if nonnegative	~SF
	jg / jnle <i>Label</i>	Jump if greater (signed)	~(SF^0F)&~ZF
	jge / jnl <i>Label</i>	Jump if greater or equal (signed)	~(SF^0F)
	j1 / jnge <i>Label</i>	Jump if less (signed)	SF^0F
	jle / jng <i>Label</i>	Jump if less or equal	(SF^0F) ZF
	ja / jnbe <i>Label</i>	Jump if above (unsigned)	~CF&~ZF
	jae / jnb <i>Label</i>	Jump if above or equal (unsigned)	~CF
	jb / jnae <i>Label</i>	Jump if below (unsigned)	CF
	jbe / jna <i>Label</i>	Jump if below or equal (unsigned)	CF ZF