

Jin-Soo Kim  
(jinsoo.kim@snu.ac.kr)

Systems Software &  
Architecture Lab.

Seoul National University

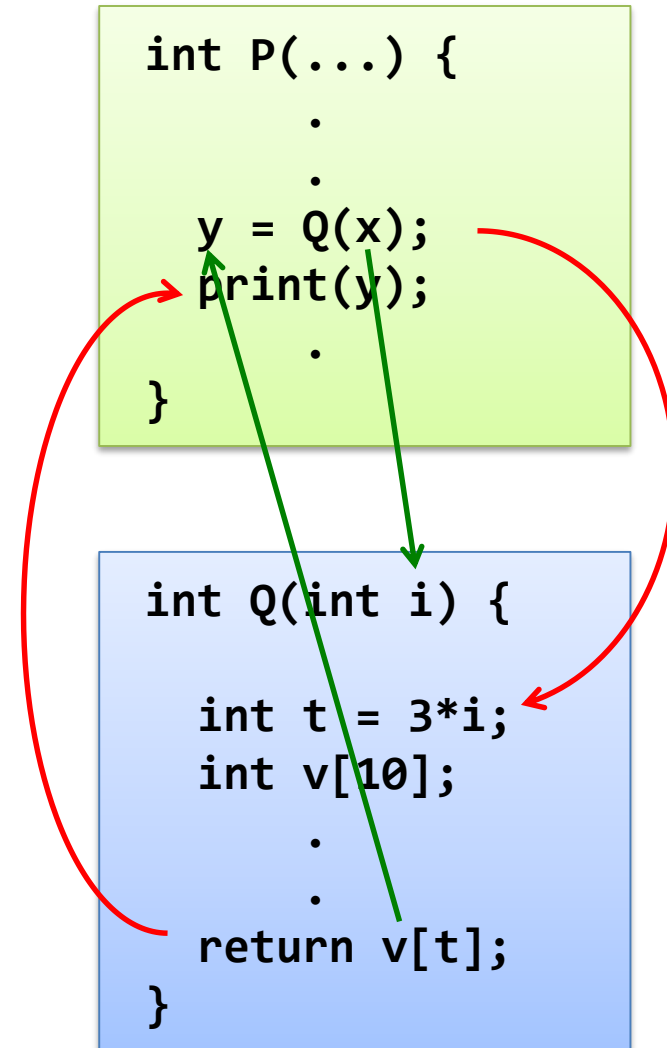
Fall 2018

# Assembly III: Procedures



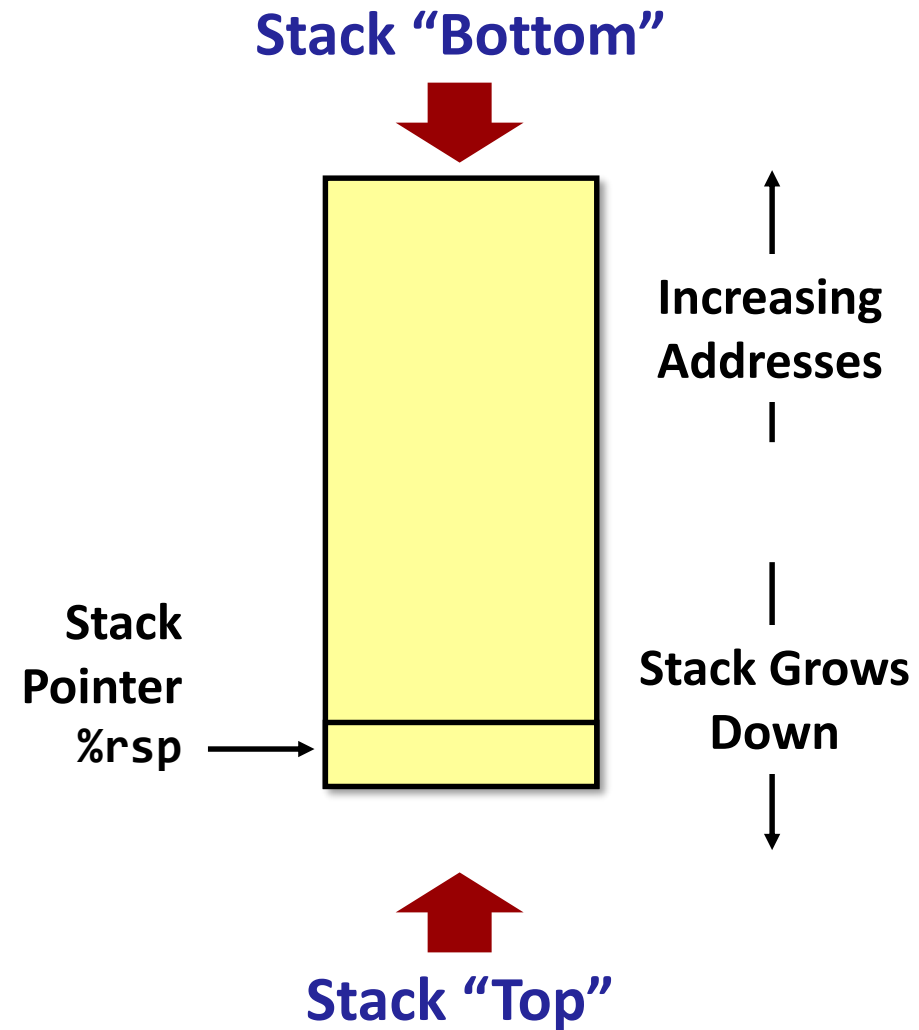
# Mechanisms in Procedures

- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value
- Memory management
  - Allocate during procedure execution
  - Deallocate upon return
- All implemented with machine instructions



# x86-64 Stack

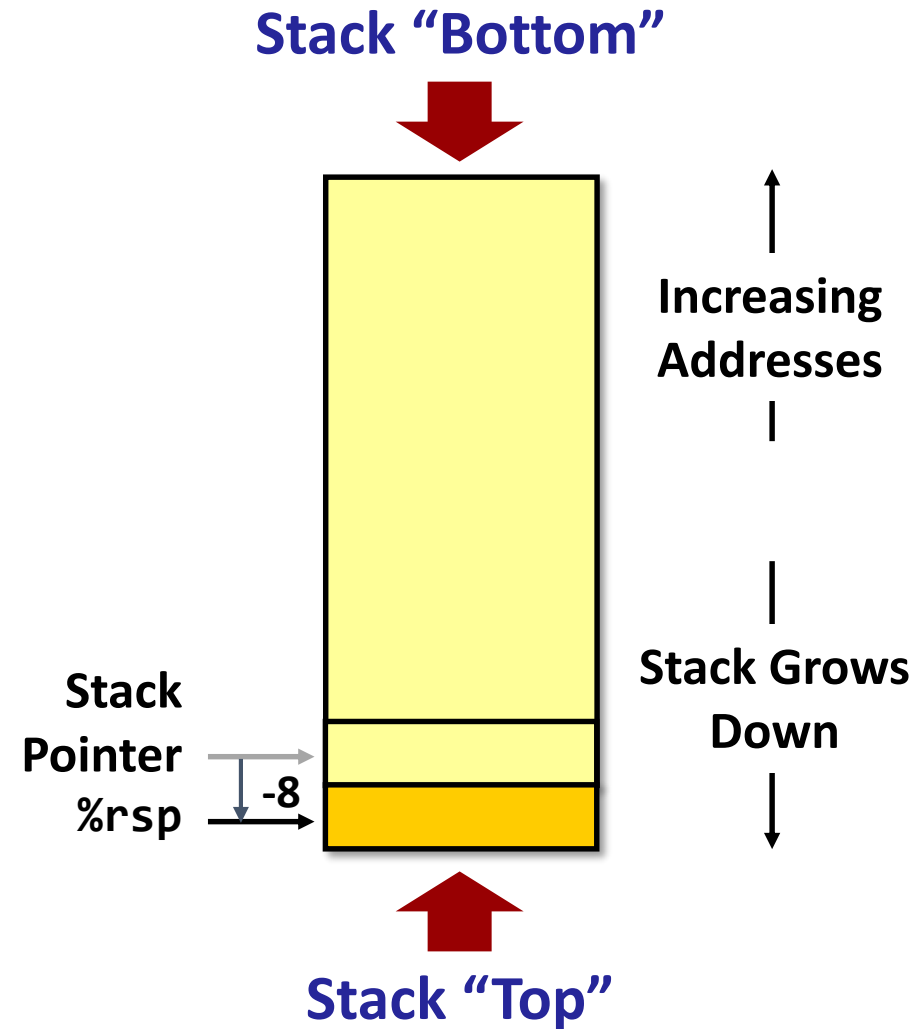
- Region of memory managed with stack discipline
  - Last-In, First-Out (LIFO)
  - Push & Pop
- Grows toward lower addresses
- Register `%rsp` contains lowest stack address
  - Address of “top” element



# x86-64 Stack: Push

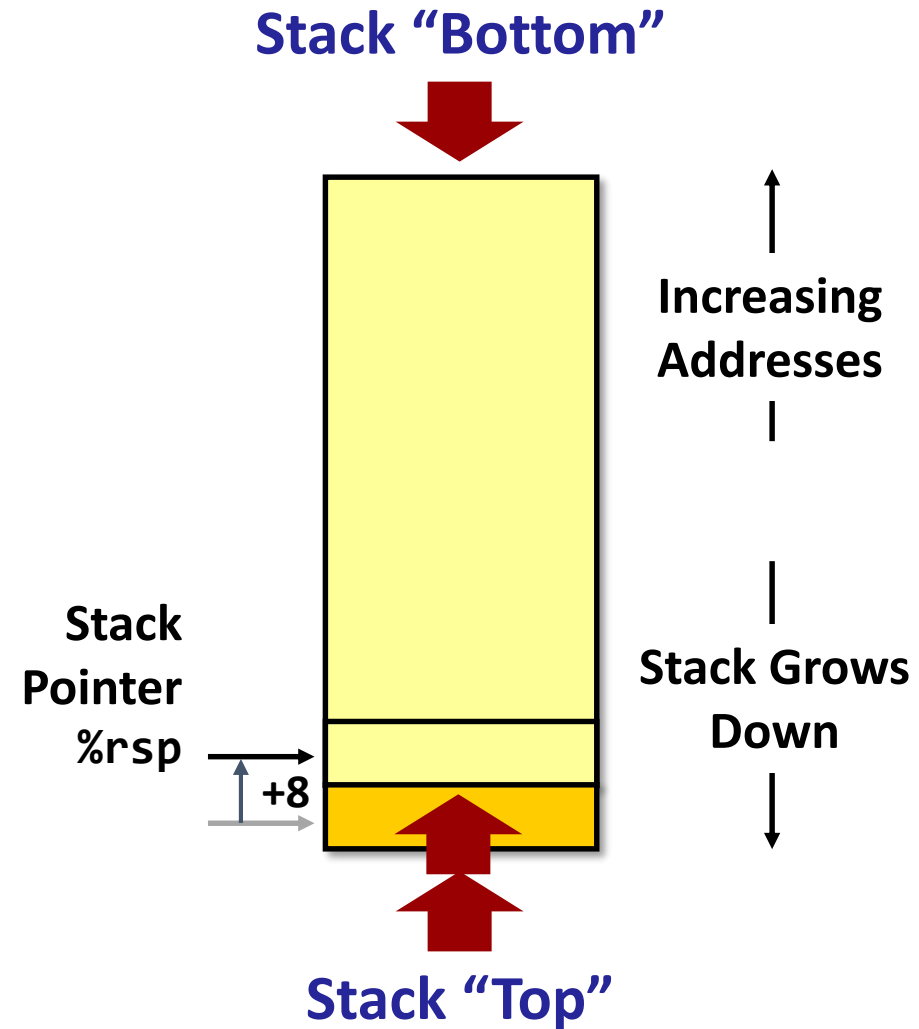
## ▪ `pushq Src`

- Fetch operand at *Src*
- Decrement `%rsp` by 8
- Write operand at address given by `%rsp`



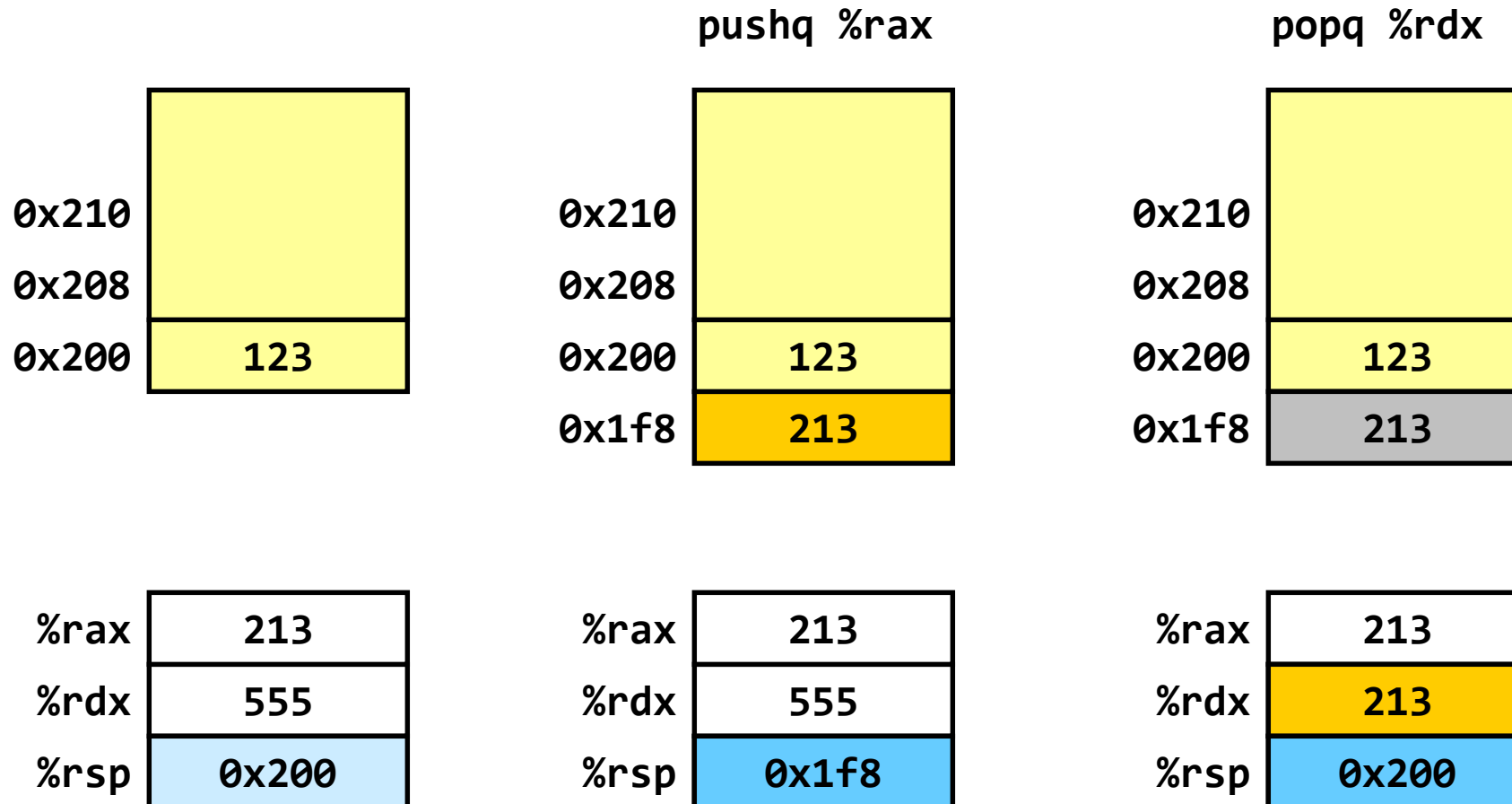
# x86-64 Stack: Pop

- **popq *Dest***
  - Read value at address given by `%rsp`
  - Increment `%rsp` by 8
  - Store value at *Dest* (must be a register)



# x86-64 Stack: Example

- Stack operation examples



# Procedure Control Flow

- Use stack to support procedure call and return
- **Procedure call: `call Label`**
  - Push return address on stack
    - Address of the next instruction right after call
  - Jump to *Label*
- **Procedure return: `ret`**
  - Pop address from stack
  - Jump to address

# Example

```
0000000000400546 <main>:
 400546:    48 8d 64 24 f8          lea    -0x8(%rsp),%rsp
 40054b:    bf 0a 00 00 00        mov    $0xa,%edi
 400550:    e8 13 00 00 00        callq  400568 <fact>
 400555:    48 89 c7              mov    %rax,%rdi
 400558:    e8 21 00 00 00        callq  40057e <print>
 40055d:    b8 00 00 00 00        mov    $0x0,%eax
 400562:    48 8d 64 24 08        lea    0x8(%rsp),%rsp
 400567:    c3                    retq

0000000000400568 <fact>:
 400568:    b8 01 00 00 00        mov    $0x1,%eax
 40056d:    eb 08                jmp    400577 <fact+0xf>
 40056f:    48 0f af c7          imul  %rdi,%rax
 400573:    48 83 ef 01          sub   $0x1,%rdi
 400577:    48 83 ff 01          cmp   $0x1,%rdi
 40057b:    7f f2                jg    40056f <fact+0x7>
 40057d:    c3                    retq
```

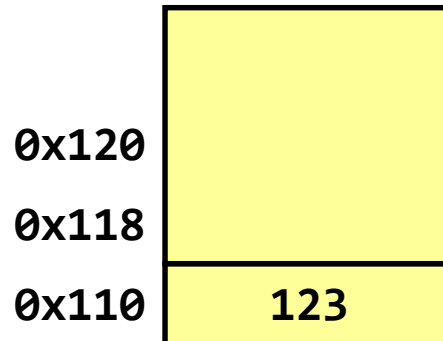


# Procedure Call Example

```

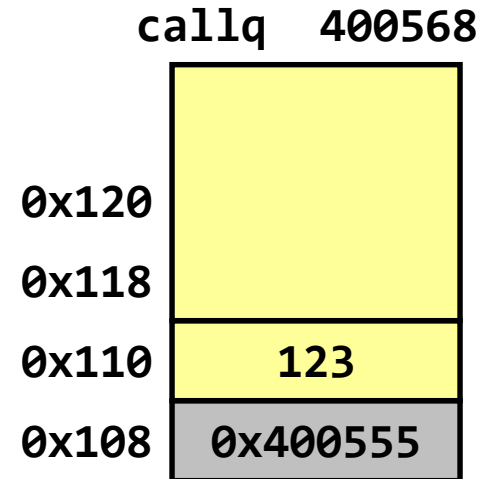
400550:  e8 13 00 00 00    callq 400568 <fact>
400555:  48 89 c7          mov  %rax,%rdi
    
```

$0x00400555$   
 $+0x00000013$   
 $=0x00400568$



`%rsp` 0x110

`%rip` 0x400550



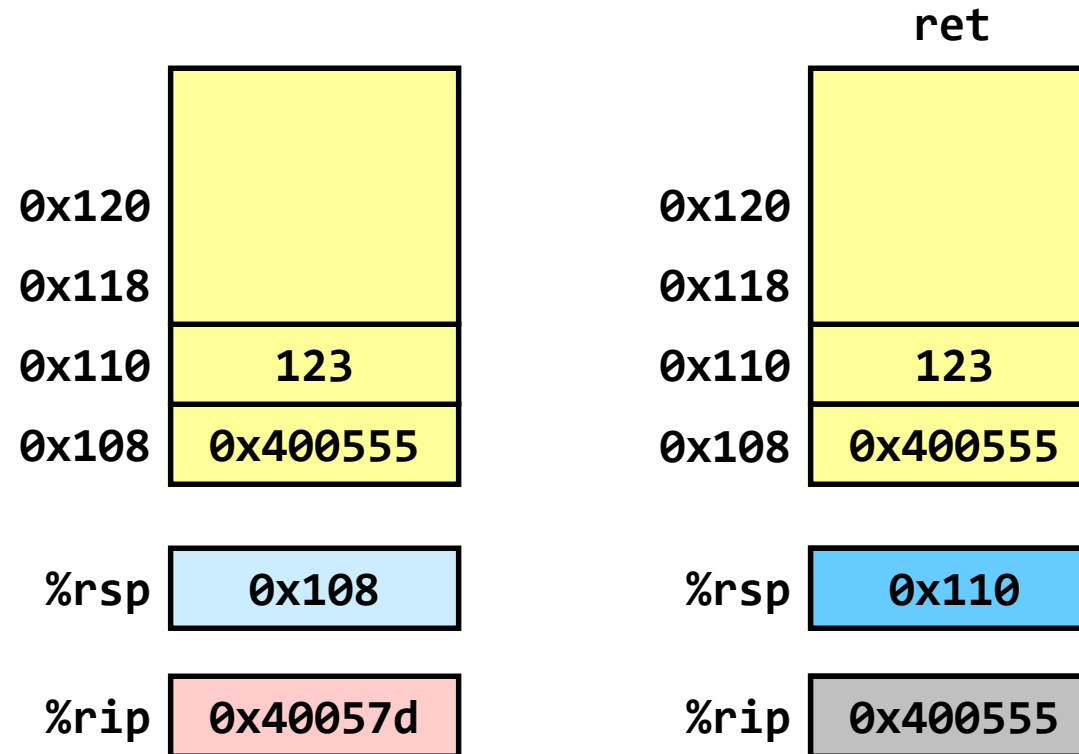
`%rsp` 0x108

`%rip` 0x400568

`%rip` is program counter

# Procedure Return Example

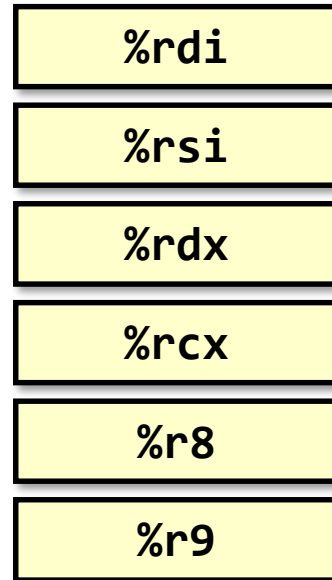
```
40057d:    c3          retq
```



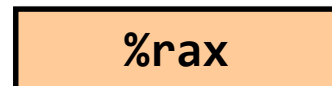
**%rip is program counter**

# Passing Arguments

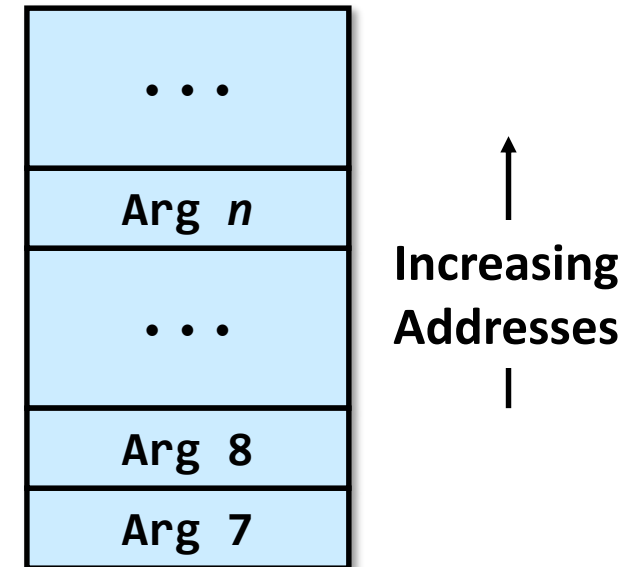
- First 6 arguments:
  - “Diane’s silk dress costs \$89”



- Return value



- Remaining arguments:
  - Push the rest on the stack in reverse order
  - Only allocate stack space when needed

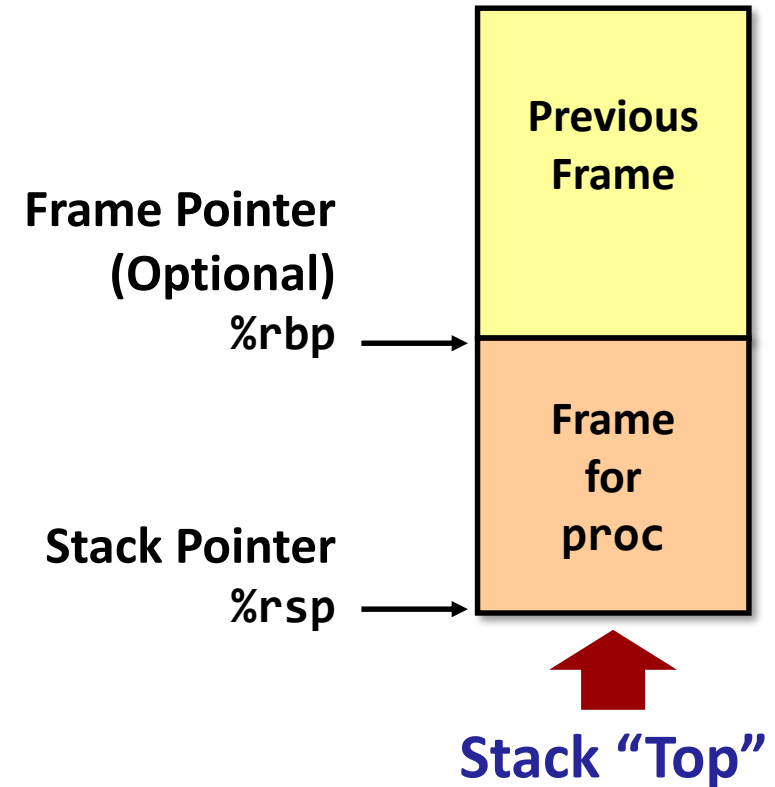


# Stack-based Languages

- Languages that support recursion (e.g. C, C++, Pascal, Java)
  - Code must be “Reentrant”
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store state of each instantiation
    - Arguments, local variables, return address
- Stack discipline
  - State for given procedure needed for limited time
    - From when called to when return
  - Callee returns before caller does
- Stack allocated in *frames*
  - State for single procedure instantiation

# Stack Frame

- **Contents**
  - Return information
  - Arguments
  - Local variables & temp space
- **Management**
  - “**Set-up**” code: space allocated when enter procedure
  - “**Finish**” code: deallocate when return
  - Stack pointer `%rsp` indicates stack top
  - Optional frame pointer `%rbp` indicates start of current frame



# Stack Frames: Example (I)

## Code Structure

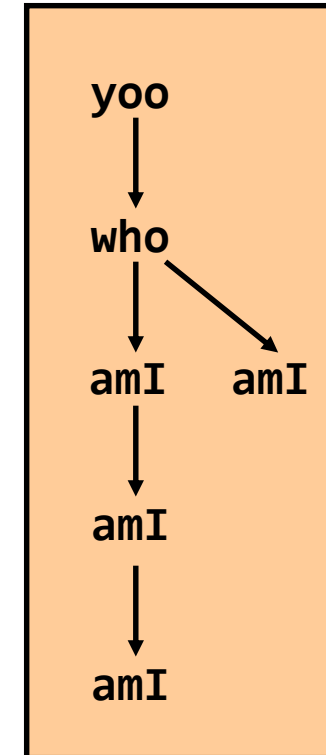
```
yoo(...)  
{  
  •  
  •  
  who();  
  •  
  •  
}
```

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

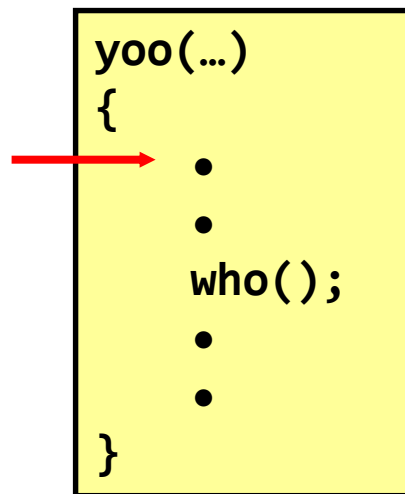
```
amI(...)  
{  
  •  
  •  
  amI();  
  •  
  •  
}
```

- Procedure **amI** recursive

## Call Chain

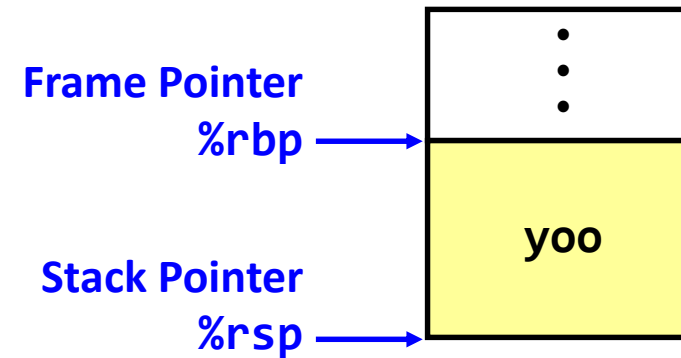


# Stack Frames: Example (2)

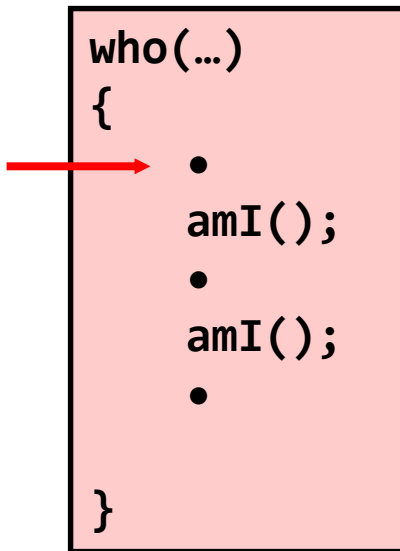


Call Chain

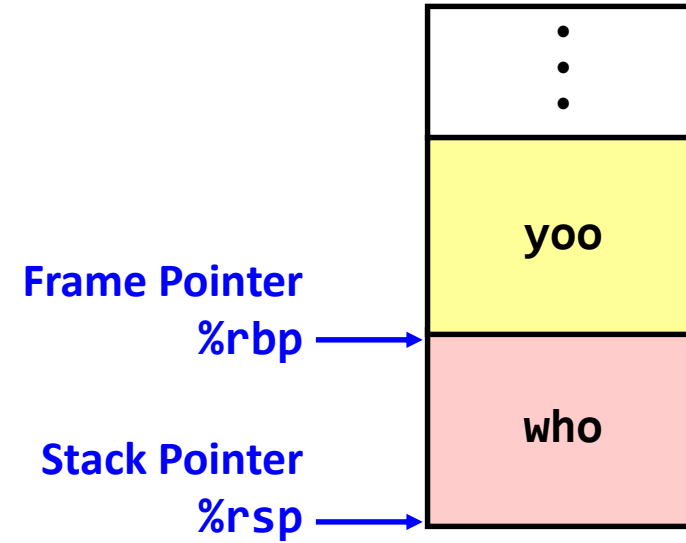
yoo



# Stack Frames: Example (3)

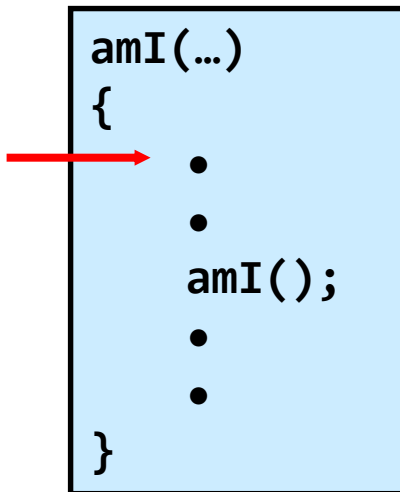


## Call Chain

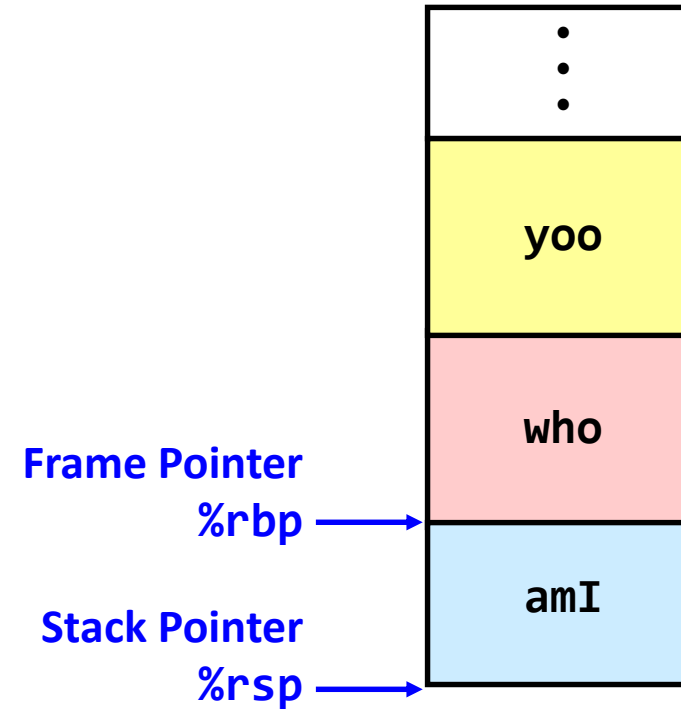
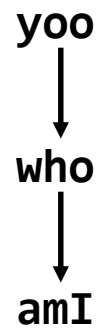




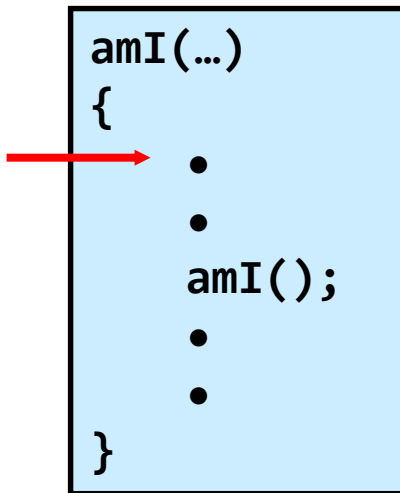
# Stack Frames: Example (4)



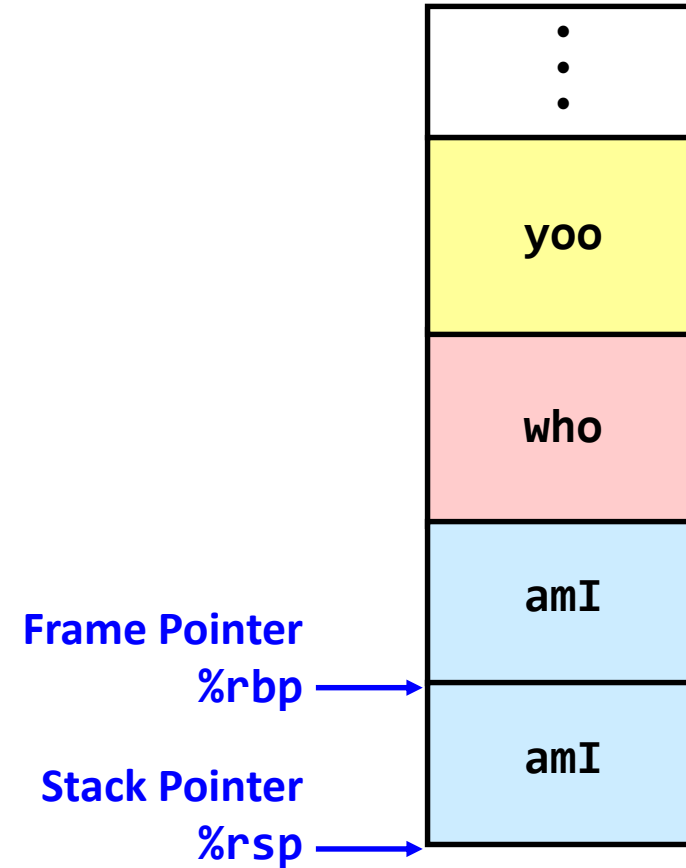
## Call Chain



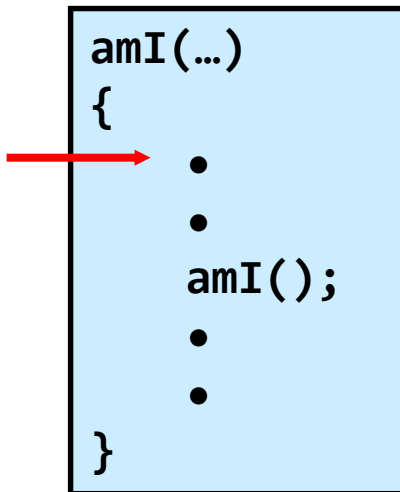
# Stack Frames: Example (5)



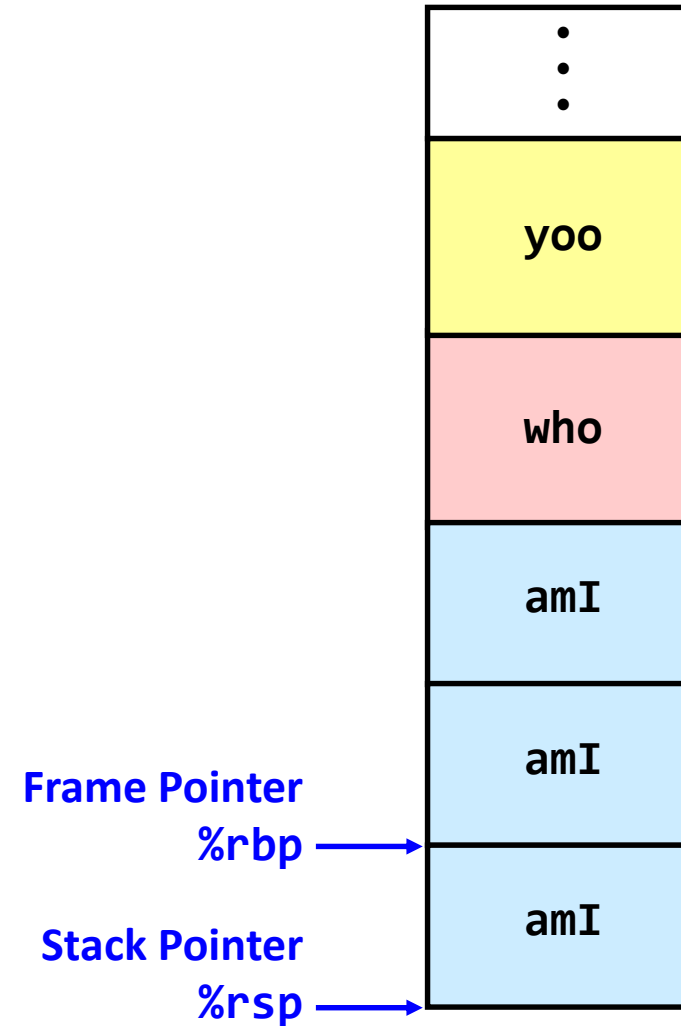
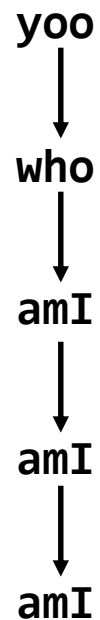
## Call Chain



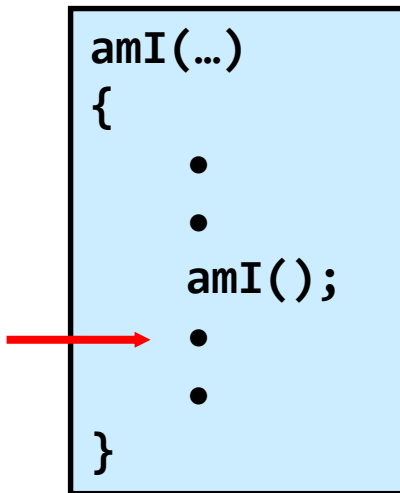
# Stack Frames: Example (6)



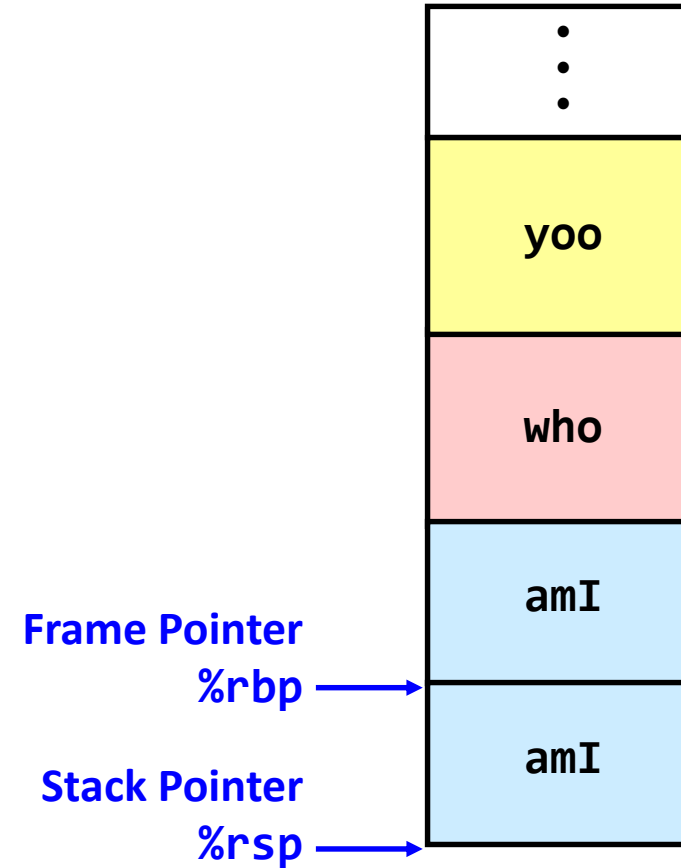
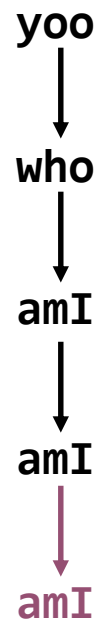
## Call Chain



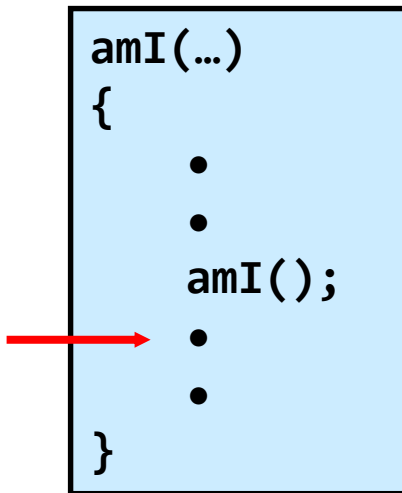
# Stack Frames: Example (7)



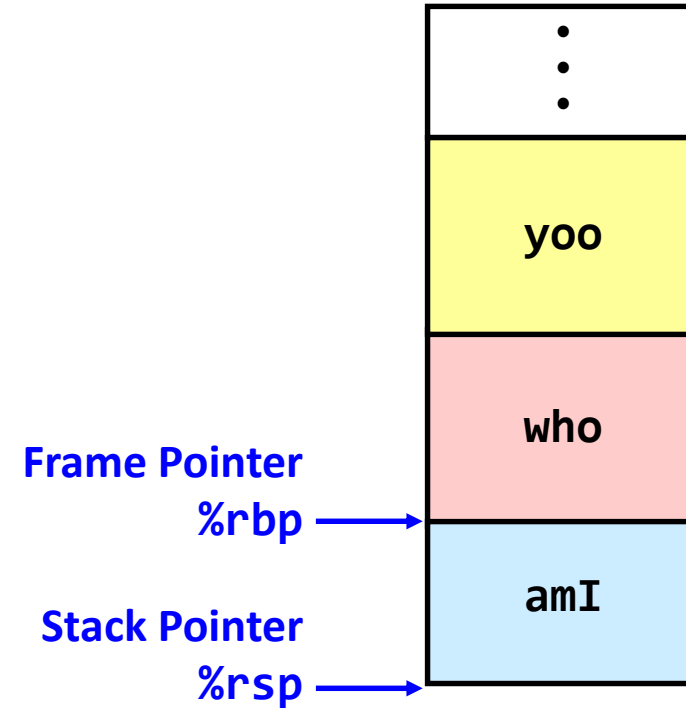
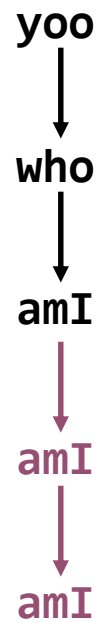
## Call Chain



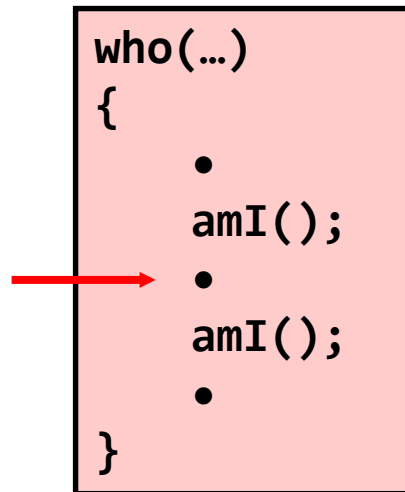
# Stack Frames: Example (8)



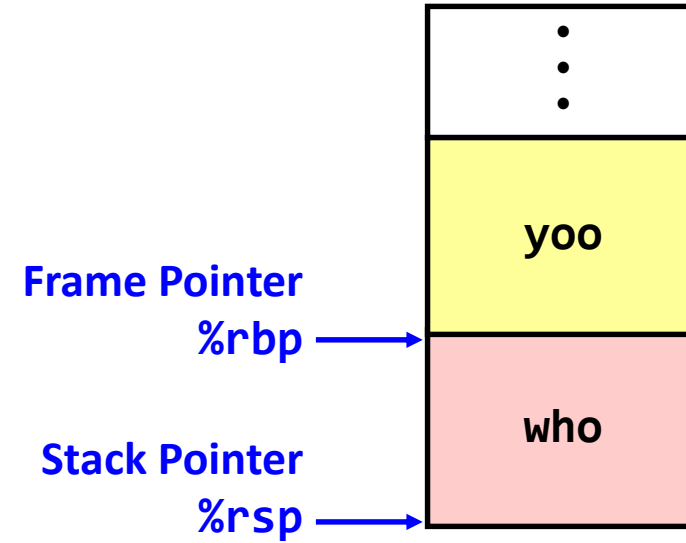
## Call Chain



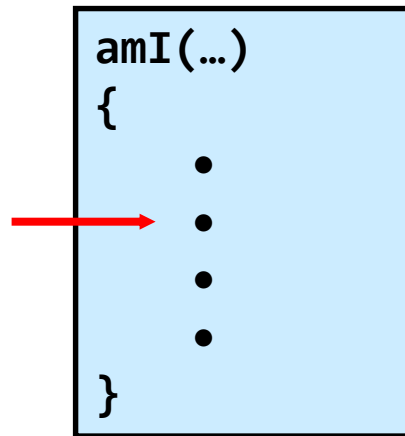
# Stack Frames: Example (9)



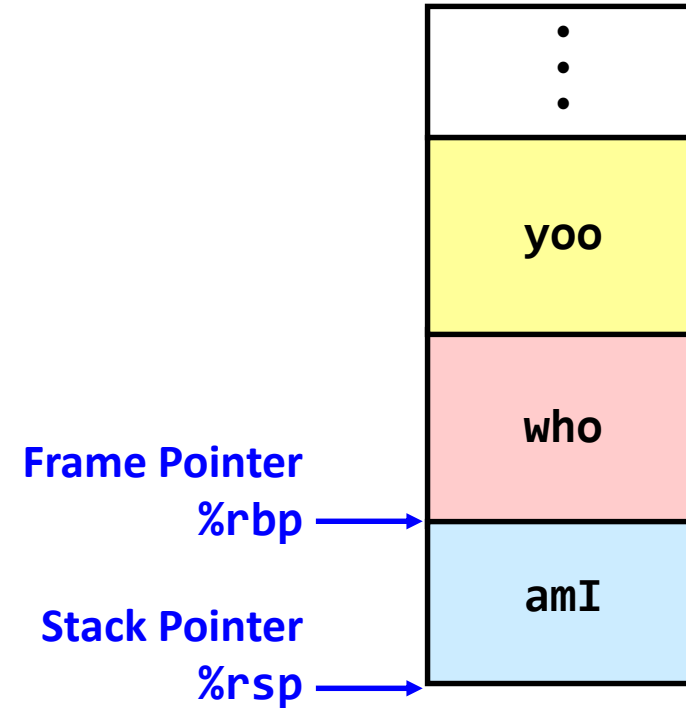
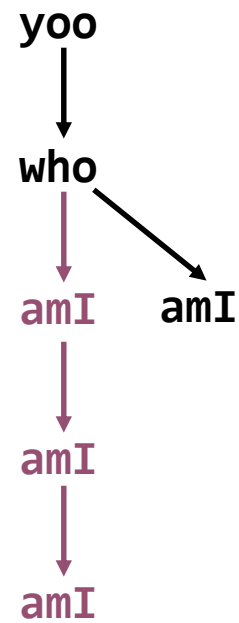
## Call Chain



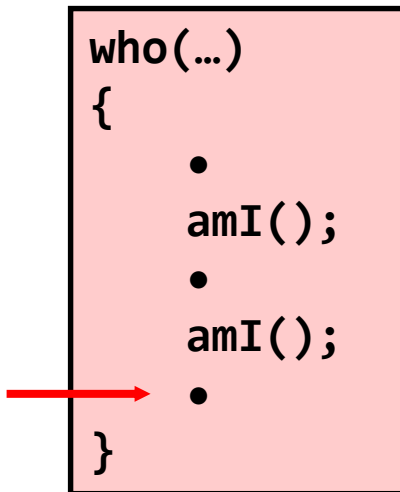
# Stack Frames: Example (10)



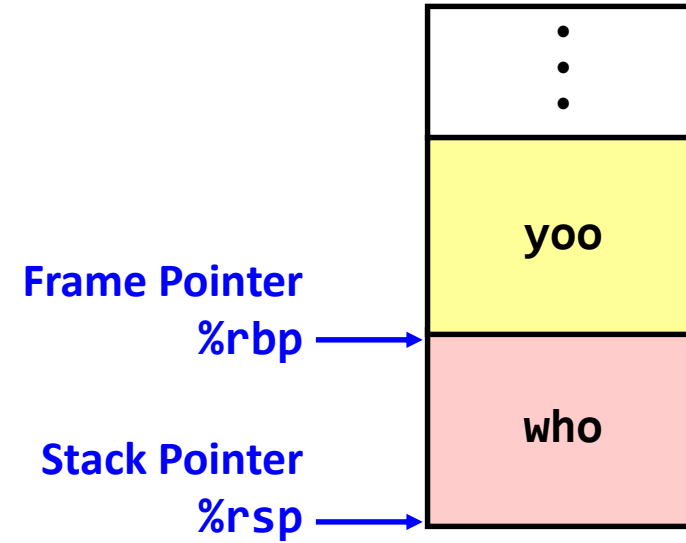
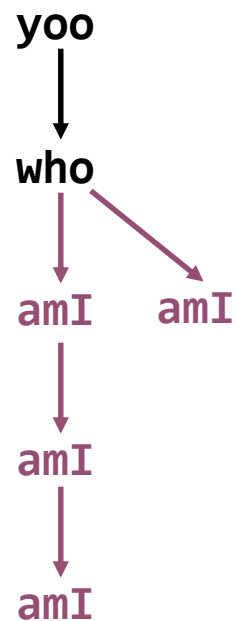
## Call Chain



# Stack Frames: Example (I I)

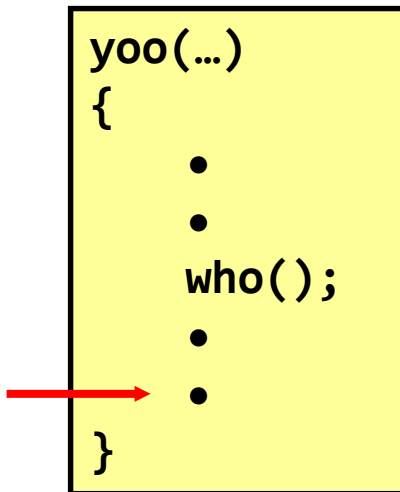


## Call Chain

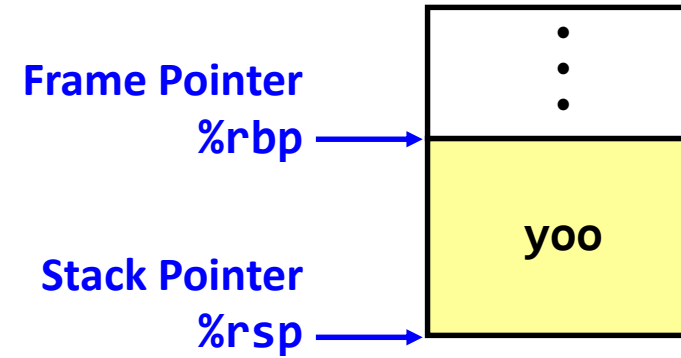
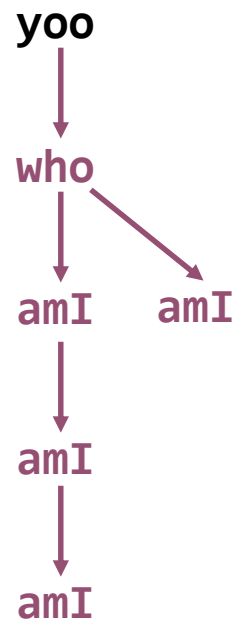




# Stack Frames: Example (12)



## Call Chain



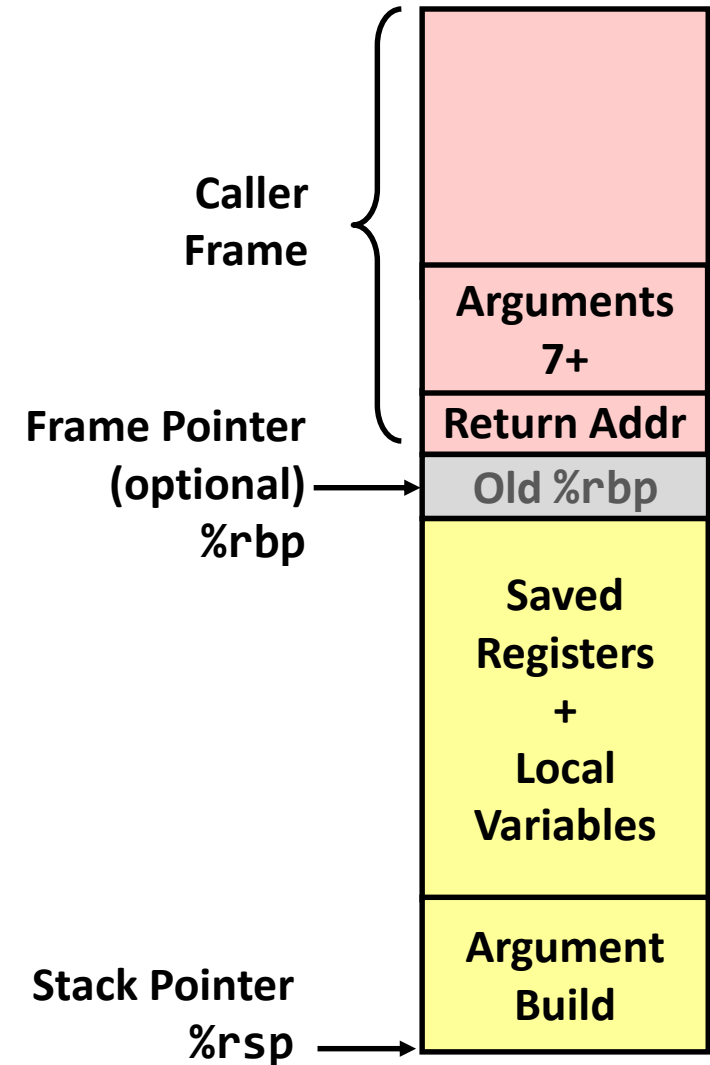
# x86-64/Linux Stack Frame

## ■ Current stack frame (“Top” to Bottom)

- “Argument build:” Parameters for function about to call
- Local variables
  - if can’t keep in registers
- Saved register context
- Old frame pointer (optional)

## ■ Caller stack frame

- Return address
  - Pushed by `call` instruction
- Arguments for this call



# Revisiting Swap

```
long v1 = 1111;
long v2 = 2222;

void swap (long *xp, long *yp)
{
    long t = *xp;
    *xp = *yp;
    *yp = t;
}

int main (void)
{
    swap (&v1, &v2);
    ...
}
```

```
v2:
    .quad    2222
    ...
v1:
    .quad    1111
    ...
swap:
    movq    (%rdi), %rax
    movq    (%rsi), %rdx
    movq    %rdx, (%rdi)
    movq    %rax, (%rsi)
    ret

main:
    ...
    movq    $v2, %rsi
    movq    $v1, %rdi
    call   swap
    ...
    ret
```

# Register Saving Problem

- When procedure `yoo()` calls `who()`:
  - `yoo()` is the caller, `who()` is the callee
- Can register be used for temporary storage?

```
yoo:  
  . . .  
  movq $15213, %rdx  
  call who  
  addq %rdx, %rax  
  . . .  
  ret
```

```
who:  
  . . .  
  subq $91125, %rdx  
  . . .  
  ret
```

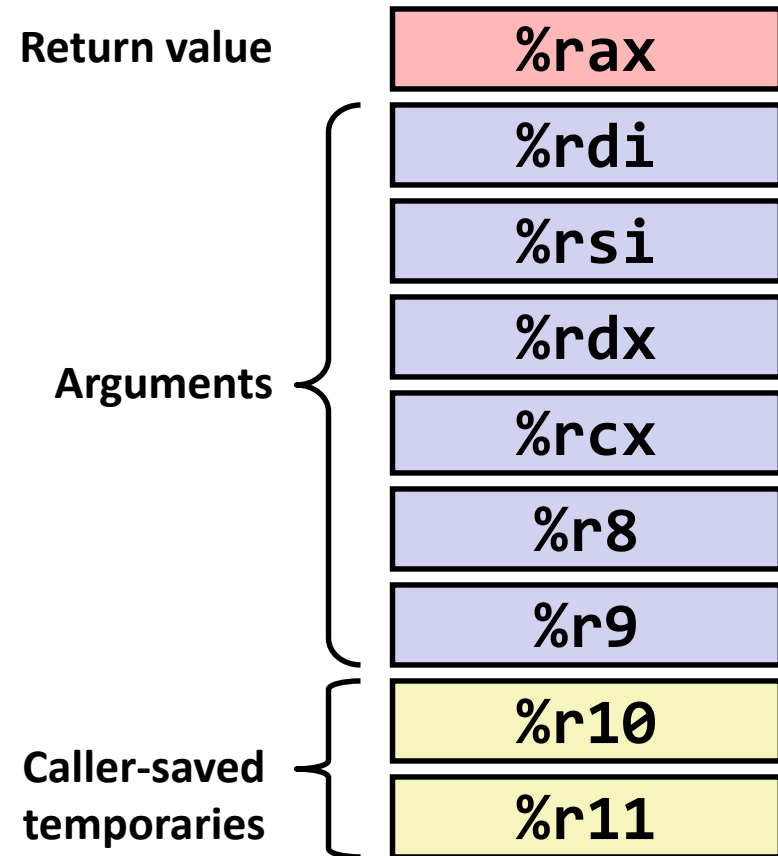
- Contents of register `%rdx` overwritten by `who()`

# Register Saving Conventions

- “**Caller saved**” registers
  - Caller saves temporary values in its frame before the call
  - Contents of these registers can be modified as a result of procedure call
  - x86-64: **%rax**, %rdi, %rsi, %rdx, %rcx, %r8, %r9, %r10, %r11
- “**Callee saved**” registers
  - Callee saves temporary values in its frame before using
  - Callee restores them before returning to caller
  - The contents of these registers are preserved across a procedure call
  - x86-64: %rbx, %r12, %r13, %r14, %r15, %rbp

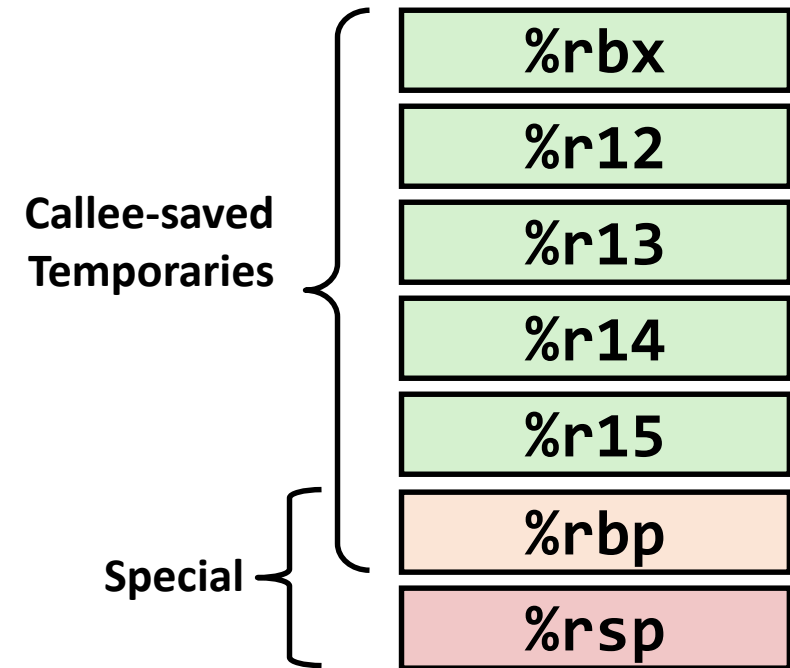
# x86-64/Linux Register Usage (I)

- **%rax**
  - Return value
  - Also caller-saved
  - Can be modified by procedure
- **%rdi, ..., %r9**
  - Arguments
  - Also caller-saved
  - Can be modified by procedure
- **%r10, %r11**
  - Caller-saved
  - Can be modified by procedure



# x86-64/Linux Register Usage (2)

- **%rbx, %r12, %r13, %r14, %r15**
  - Callee-saved
  - Callee must save & restore
- **%rbp**
  - Callee-saved
  - Callee must save & restore
  - May be used as frame pointer
- **%rsp**
  - Special from of callee save
  - Restored to original value upon exit from procedure



# Recursive Factorial: rfact

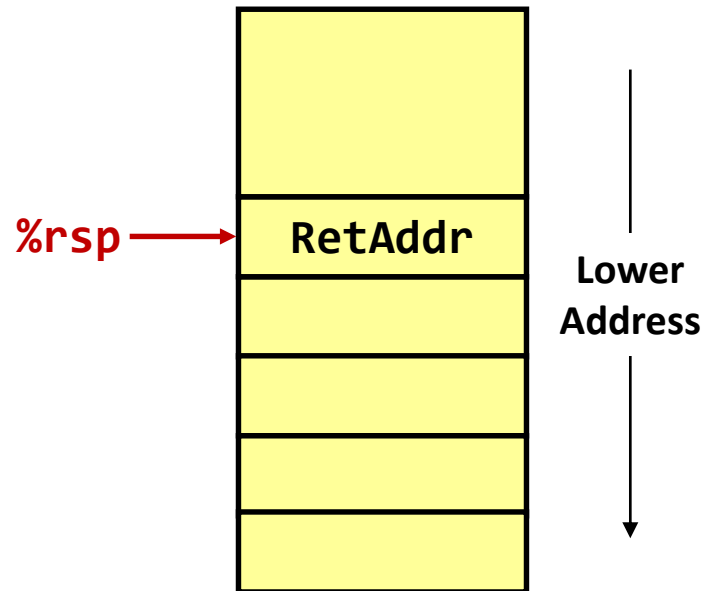
- Registers
  - %rax used without first saving
  - %rbx used, but save at beginning & restore at end

```
long rfact(long x)
{
    long rval;
    if (x <= 1)
        return 1;
    rval = rfact(x-1);
    return rval * x;
}
```

```
rfact:
    cmpq    $1, %rdi
    jle    .L3
    pushq  %rbx
    movq   %rdi, %rbx
    leaq   -1(%rdi), %rdi
    call   rfact
    imulq  %rbx, %rax
    jmp    .L2
.L3:
    movl   $1, %eax
    ret
.L2:
    popq   %rbx
    ret
```



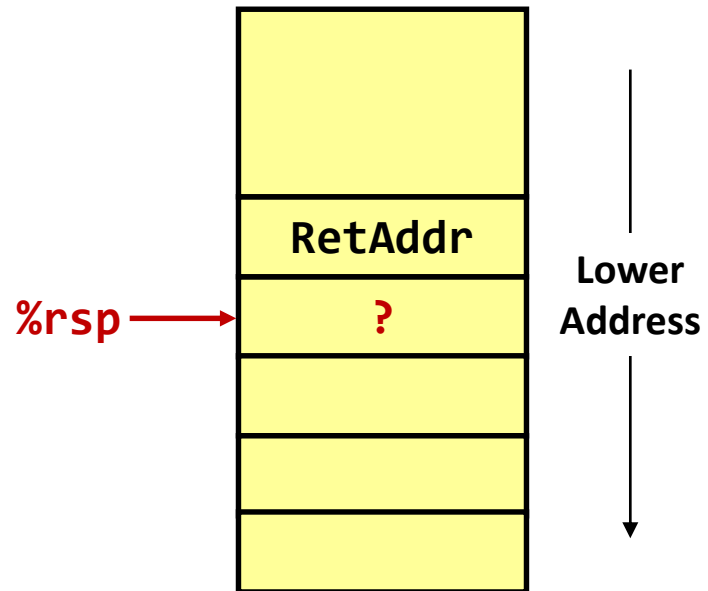
# Example: rfact(3)



Registers	
%rdi	3
%rax	?
%rbx	?

```
rfact:
%rip → cmpq   $1, %rdi
      jle   .L3
      pushq %rbx
      movq  %rdi, %rbx
      leaq  -1(%rdi), %rdi
      call  rfact
A: imulq  %rbx, %rax
      jmp  .L2
.L3:
      movl  $1, %eax
      ret
.L2:
      popq  %rbx
      ret
```

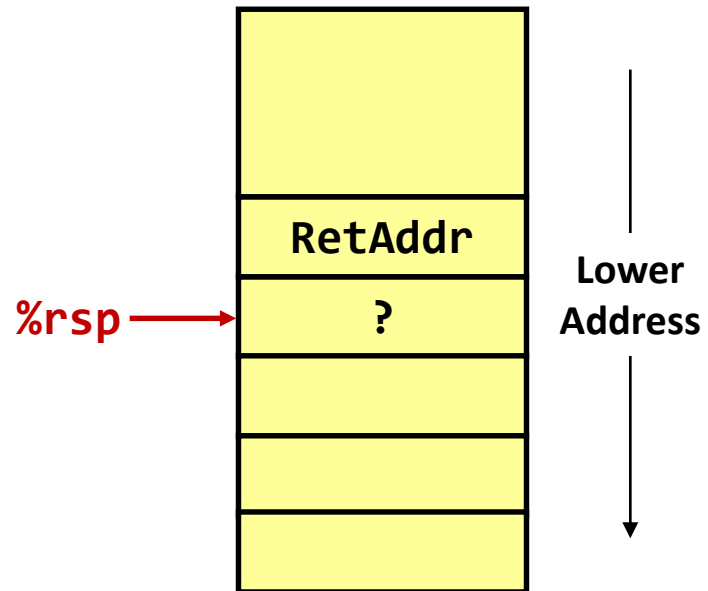
# Example: rfact(3)



Registers	
%rdi	3
%rax	?
%rbx	?

```
rfact:
    cmpq    $1, %rdi
    jle     .L3
    pushq   %rbx
    movq    %rdi, %rbx
    leaq   -1(%rdi), %rdi
    call    rfact
A: imulq   %rbx, %rax
    jmp     .L2
.L3:
    movl    $1, %eax
    ret
.L2:
    popq    %rbx
    ret
```

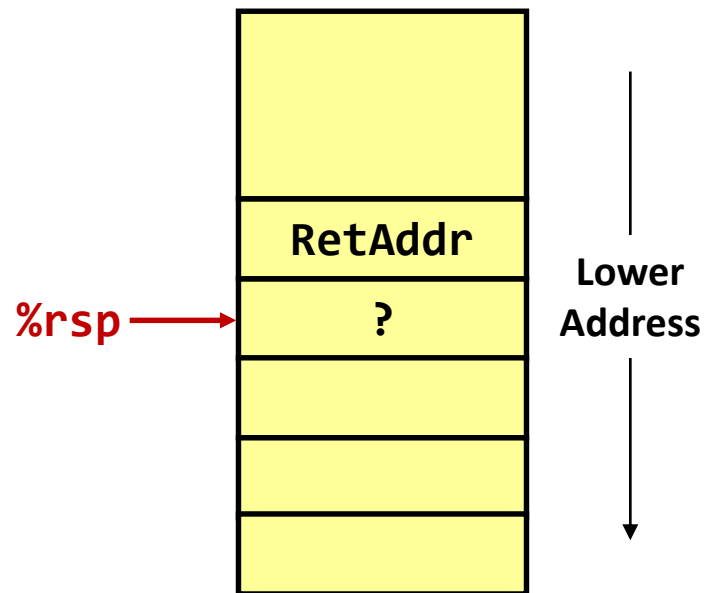
# Example: rfact(3)



Registers	
%rdi	3
%rax	?
%rbx	3

```
rfact:
    cmpq    $1, %rdi
    jle     .L3
    pushq   %rbx
    movq    %rdi, %rbx
    leaq   -1(%rdi), %rdi
    call   rfact
A: imulq   %rbx, %rax
    jmp     .L2
.L3:
    movl   $1, %eax
    ret
.L2:
    popq   %rbx
    ret
```

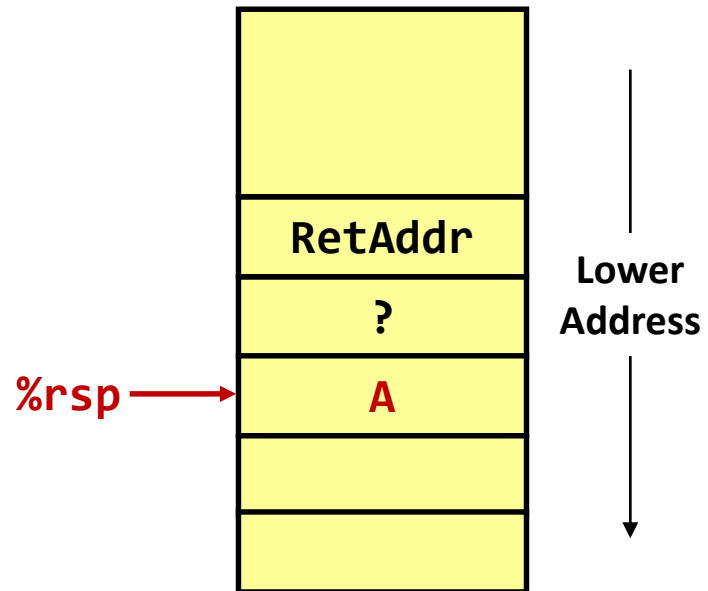
# Example: rfact(3)



Registers	
%rdi	2
%rax	?
%rbx	3

```
rfact:
    cmpq    $1, %rdi
    jle    .L3
    pushq  %rbx
    movq   %rdi, %rbx
    leaq  -1(%rdi), %rdi
    call   rfact
A: imulq  %rbx, %rax
    jmp   .L2
.L3:
    movl   $1, %eax
    ret
.L2:
    popq   %rbx
    ret
```

# Example: rfact(3)

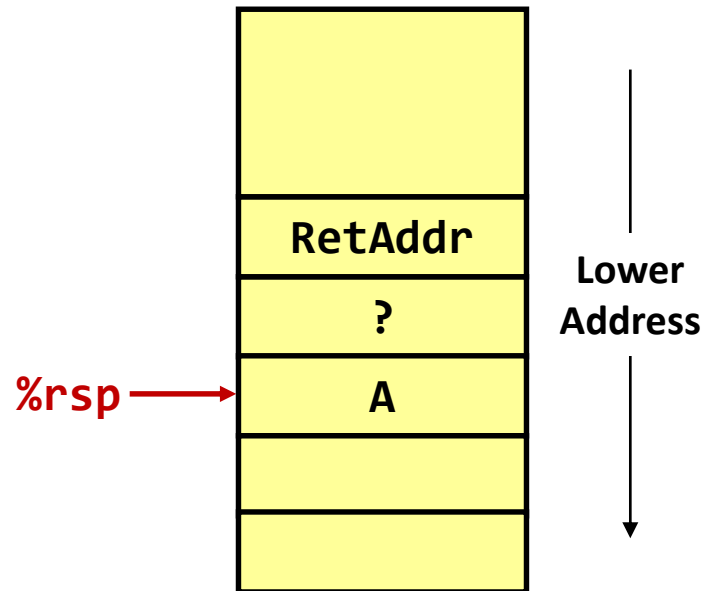


Registers	
<code>%rdi</code>	2
<code>%rax</code>	?
<code>%rbx</code>	3

```
rfact:
    cmpq    $1, %rdi
    jle    .L3
    pushq  %rbx
    movq   %rdi, %rbx
    leaq  -1(%rdi), %rdi
    call   rfact
A: imulq  %rbx, %rax
    jmp   .L2
.L3:
    movl  $1, %eax
    ret
.L2:
    popq  %rbx
    ret
```

`%rip` points to the `call rfact` instruction.

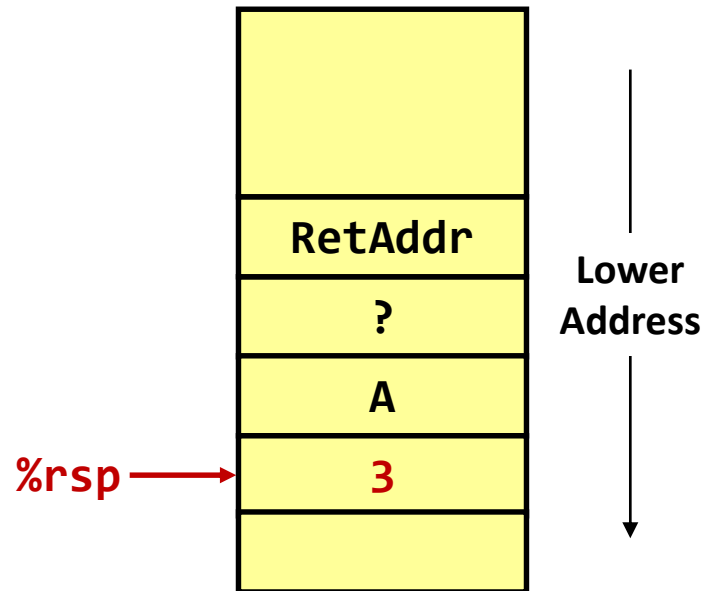
# Example: rfact(3)



Registers	
%rdi	2
%rax	?
%rbx	3

```
rfact:
%rip → cmpq   $1, %rdi
      jle   .L3
      pushq %rbx
      movq  %rdi, %rbx
      leaq  -1(%rdi), %rdi
      call  rfact
A:    imulq %rbx, %rax
      jmp  .L2
.L3:  movl   $1, %eax
      ret
.L2:  popq   %rbx
      ret
```

# Example: rfact(3)

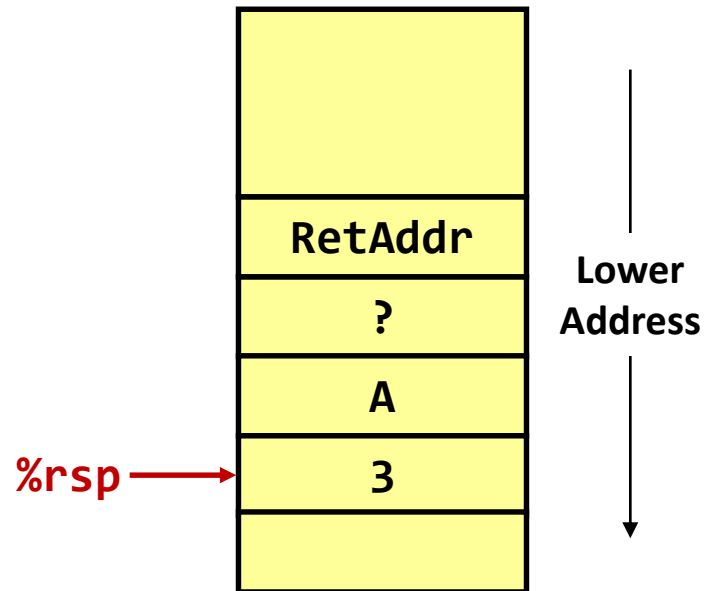


Registers	
%rdi	2
%rax	?
%rbx	3

```
rfact:
    cmpq    $1, %rdi
    jle    .L3
    pushq  %rbx
    movq   %rdi, %rbx
    leaq  -1(%rdi), %rdi
    call  rfact
A:    imulq %rbx, %rax
    jmp   .L2
.L3:
    movl  $1, %eax
    ret
.L2:
    popq  %rbx
    ret
```

%rip →

# Example: rfact(3)



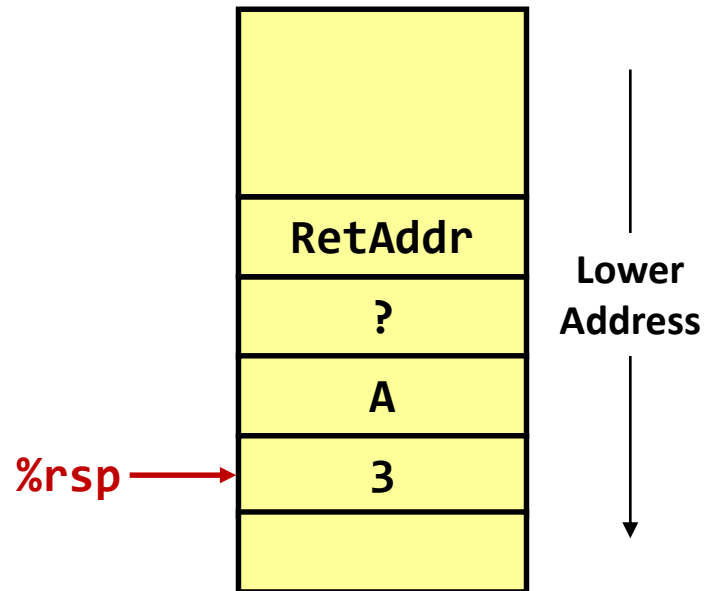
Registers	
<code>%rdi</code>	2
<code>%rax</code>	?
<code>%rbx</code>	2

```
rfact:
    cmpq    $1, %rdi
    jle    .L3
    pushq  %rbx
    movq   %rdi, %rbx
    leaq   -1(%rdi), %rdi
    call   rfact
A:    imulq %rbx, %rax
    jmp   .L2
.L3:
    movl   $1, %eax
    ret
.L2:
    popq   %rbx
    ret
```

`%rip` points to the `movq` instruction.



# Example: rfact(3)

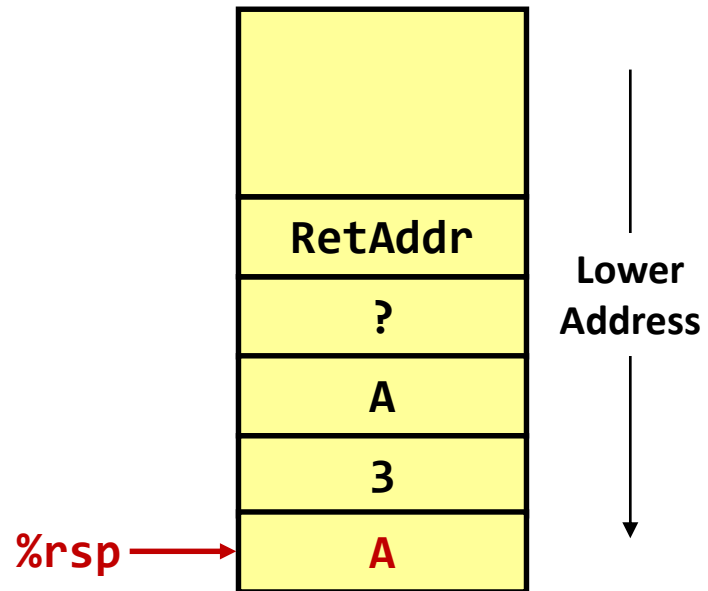


Registers	
%rdi	1
%rax	?
%rbx	2

```
rfact:
    cmpq    $1, %rdi
    jle     .L3
    pushq   %rbx
    movq    %rdi, %rbx
    leaq    -1(%rdi), %rdi
    call    rfact
A:   imulq  %rbx, %rax
    jmp     .L2
.L3:
    movl   $1, %eax
    ret
.L2:
    popq   %rbx
    ret
```

A red arrow labeled '%rip' points to the 'leaq' instruction.

# Example: rfact(3)

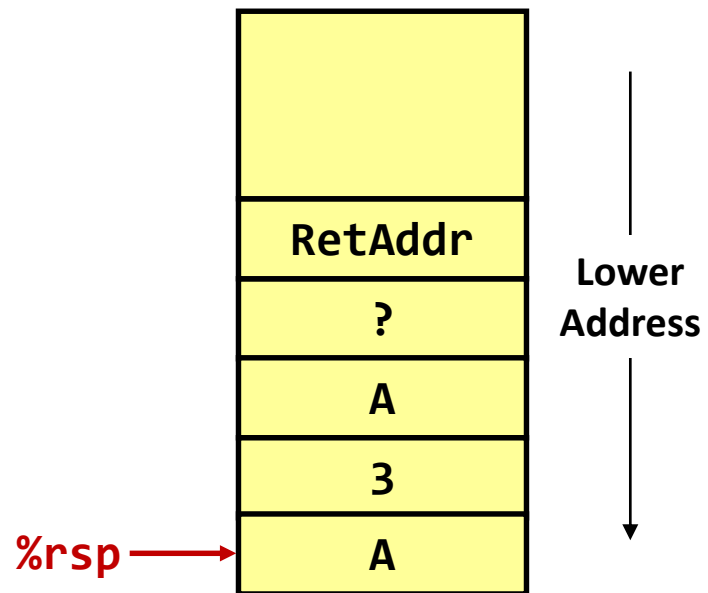


Registers	
%rdi	1
%rax	?
%rbx	2

```
rfact:
    cmpq    $1, %rdi
    jle    .L3
    pushq  %rbx
    movq   %rdi, %rbx
    leaq  -1(%rdi), %rdi
    call   rfact
A:    imulq %rbx, %rax
    jmp   .L2
.L3:
    movl  $1, %eax
    ret
.L2:
    popq  %rbx
    ret
```

%rip →

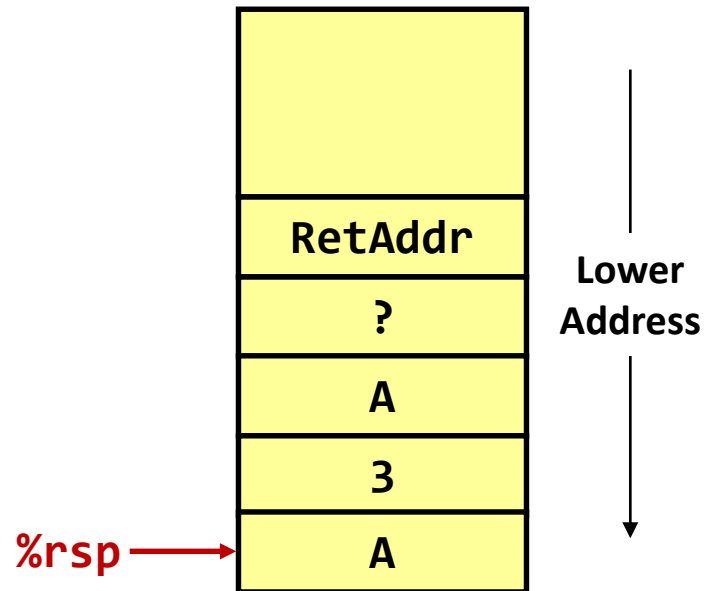
# Example: rfact(3)



Registers	
<code>%rdi</code>	1
<code>%rax</code>	?
<code>%rbx</code>	2

```
rfact:
%rip → cmpq   $1, %rdi
      jle   .L3
      pushq %rbx
      movq  %rdi, %rbx
      leaq  -1(%rdi), %rdi
      call  rfact
A:    imulq %rbx, %rax
      jmp  .L2
.L3:  movl   $1, %eax
      ret
.L2:  popq   %rbx
      ret
```

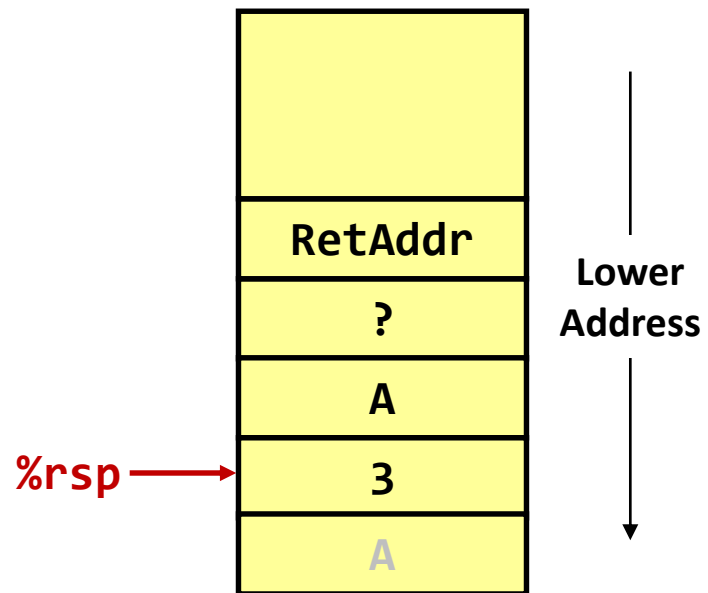
# Example: rfact(3)



Registers	
%rdi	1
%rax	<b>1</b>
%rbx	2

```
rfact:
    cmpq    $1, %rdi
    jle     .L3
    pushq   %rbx
    movq    %rdi, %rbx
    leaq   -1(%rdi), %rdi
    call    rfact
A:    imulq %rbx, %rax
        jmp     .L2
.L3:
%rip → movl    $1, %eax
        ret
.L2:
        popq   %rbx
        ret
```

# Example: rfact(3)

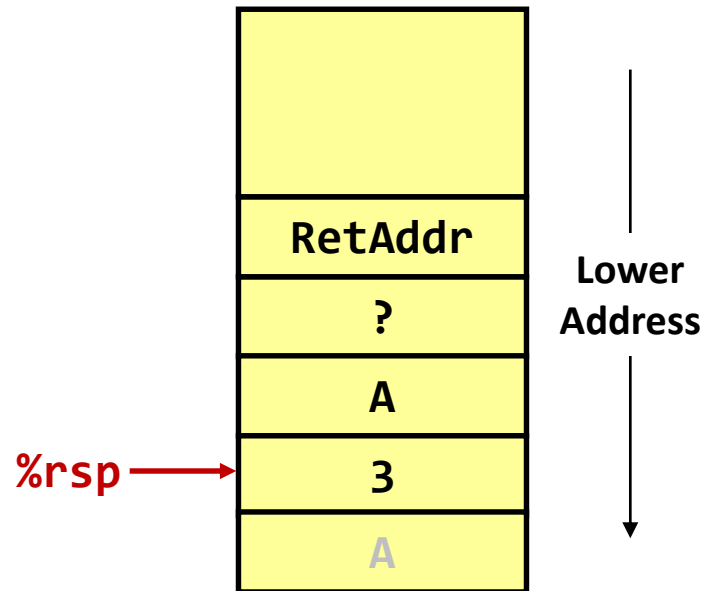


Registers	
<code>%rdi</code>	1
<code>%rax</code>	1
<code>%rbx</code>	2

```
rfact:
    cmpq    $1, %rdi
    jle    .L3
    pushq  %rbx
    movq   %rdi, %rbx
    leaq  -1(%rdi), %rdi
    call  rfact
    imulq %rbx, %rax
    jmp   .L2
.L3:
    movl  $1, %eax
    ret
.L2:
    popq  %rbx
    ret
```

`%rip` points to the instruction `imulq %rbx, %rax`.

# Example: rfact(3)

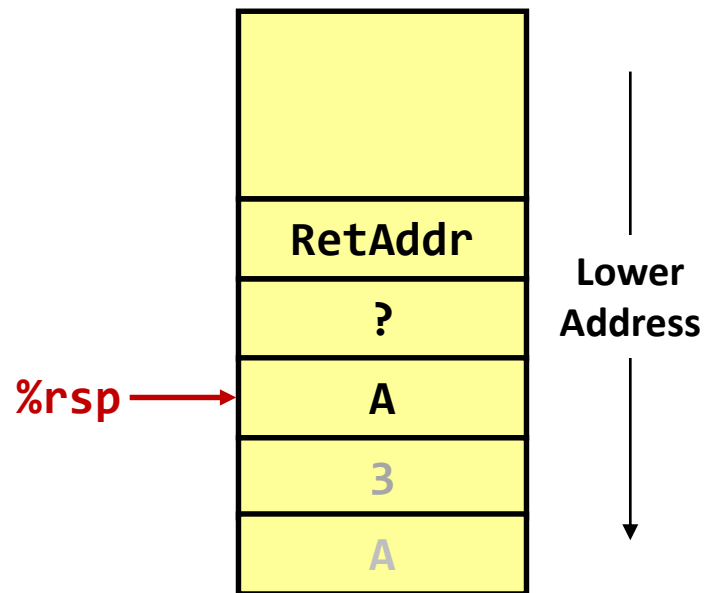


Registers	
<code>%rdi</code>	1
<code>%rax</code>	2
<code>%rbx</code>	2

```
rfact:
    cmpq    $1, %rdi
    jle    .L3
    pushq  %rbx
    movq   %rdi, %rbx
    leaq  -1(%rdi), %rdi
    call   rfact
    imulq %rbx, %rax
    jmp   .L2
.L3:
    movl  $1, %eax
    ret
.L2:
    popq  %rbx
    ret
```

`%rip` points to the instruction `imulq %rbx, %rax`.

# Example: rfact(3)

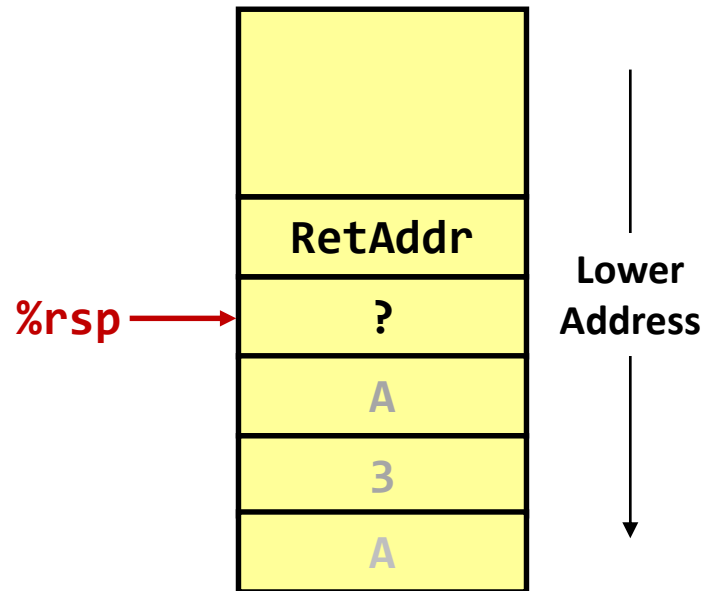


Registers	
<code>%rdi</code>	1
<code>%rax</code>	2
<code>%rbx</code>	3

```
rfact:
    cmpq    $1, %rdi
    jle    .L3
    pushq  %rbx
    movq   %rdi, %rbx
    leaq  -1(%rdi), %rdi
    call   rfact
A:    imulq %rbx, %rax
    jmp   .L2
.L3:
    movl  $1, %eax
    ret
.L2:
    popq  %rbx
    ret
```

`%rip` points to the `popq %rbx` instruction.

# Example: rfact(3)



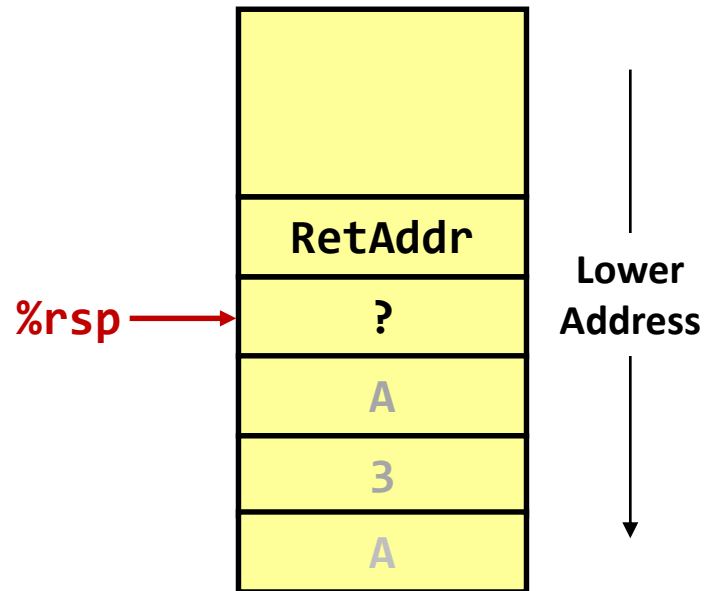
Registers	
%rdi	1
%rax	2
%rbx	3

```
rfact:
    cmpq    $1, %rdi
    jle     .L3
    pushq   %rbx
    movq    %rdi, %rbx
    leaq   -1(%rdi), %rdi
    call    rfact
    imulq   %rbx, %rax
    jmp     .L2
.L3:
    movl    $1, %eax
    ret
.L2:
    popq    %rbx
    ret
```

A red arrow labeled `%rip` points to the instruction `imulq %rbx, %rax`, which is also marked with a blue arrow labeled `A`.



# Example: rfact(3)

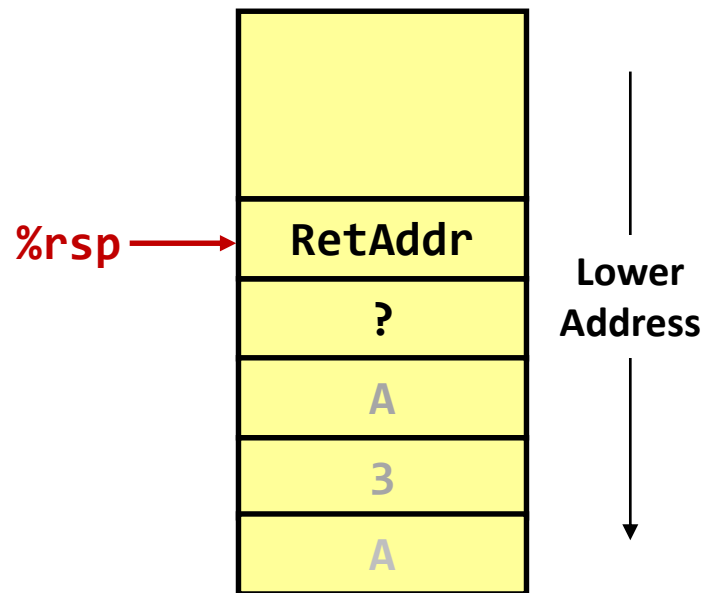


Registers	
%rdi	1
%rax	6
%rbx	3

```
rfact:
    cmpq    $1, %rdi
    jle     .L3
    pushq   %rbx
    movq    %rdi, %rbx
    leaq   -1(%rdi), %rdi
    call    rfact
    imulq   %rbx, %rax
    jmp     .L2
.L3:
    movl    $1, %eax
    ret
.L2:
    popq    %rbx
    ret
```

A red arrow labeled `%rip` points to the `imulq` instruction in the assembly code.

# Example: rfact(3)

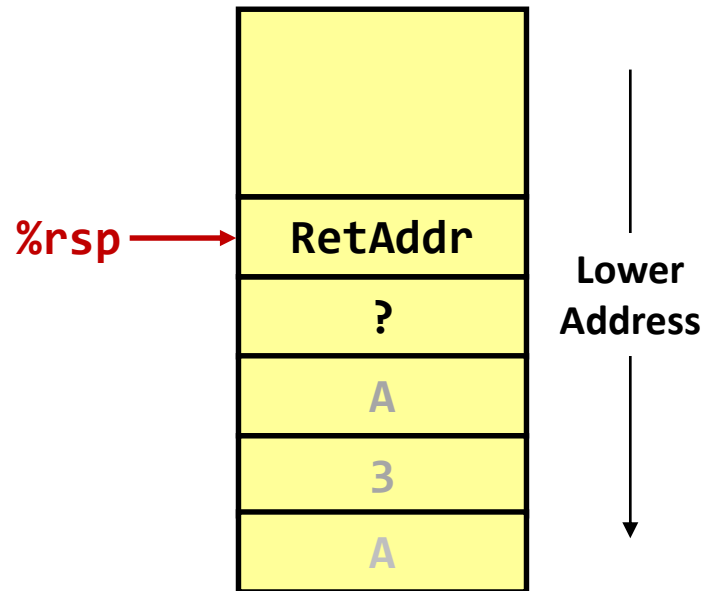


Registers	
<code>%rdi</code>	1
<code>%rax</code>	6
<code>%rbx</code>	?

```
rfact:
    cmpq    $1, %rdi
    jle    .L3
    pushq  %rbx
    movq   %rdi, %rbx
    leaq  -1(%rdi), %rdi
    call  rfact
A:    imulq %rbx, %rax
    jmp   .L2
.L3:
    movl  $1, %eax
    ret
.L2:
    popq  %rbx
    ret
```

`%rip` points to the `popq %rbx` instruction.

# Example: rfact(3)



Registers	
%rdi	1
%rax	6
%rbx	?

```
rfact:
    cmpq    $1, %rdi
    jle     .L3
    pushq   %rbx
    movq    %rdi, %rbx
    leaq   -1(%rdi), %rdi
    call    rfact
A:   imulq  %rbx, %rax
    jmp     .L2
.L3:
    movl   $1, %eax
    ret
.L2:
    popq   %rbx
    ret
```

%rip →

# Observations about Recursion

- **Handled without special consideration**
  - Stack frames mean each function call has private storage
    - Saved registers + local variables
    - Saved return address
  - Register saving conventions prevent one function call from corrupting another's data
    - Unless the C code explicitly does so (e.g. buffer overflow)
  - Stack discipline follows call / return pattern
    - If P calls Q, then Q returns before P
    - Last-In, First-Out
- **Also works for mutual recursion**
  - P calls Q; Q calls P

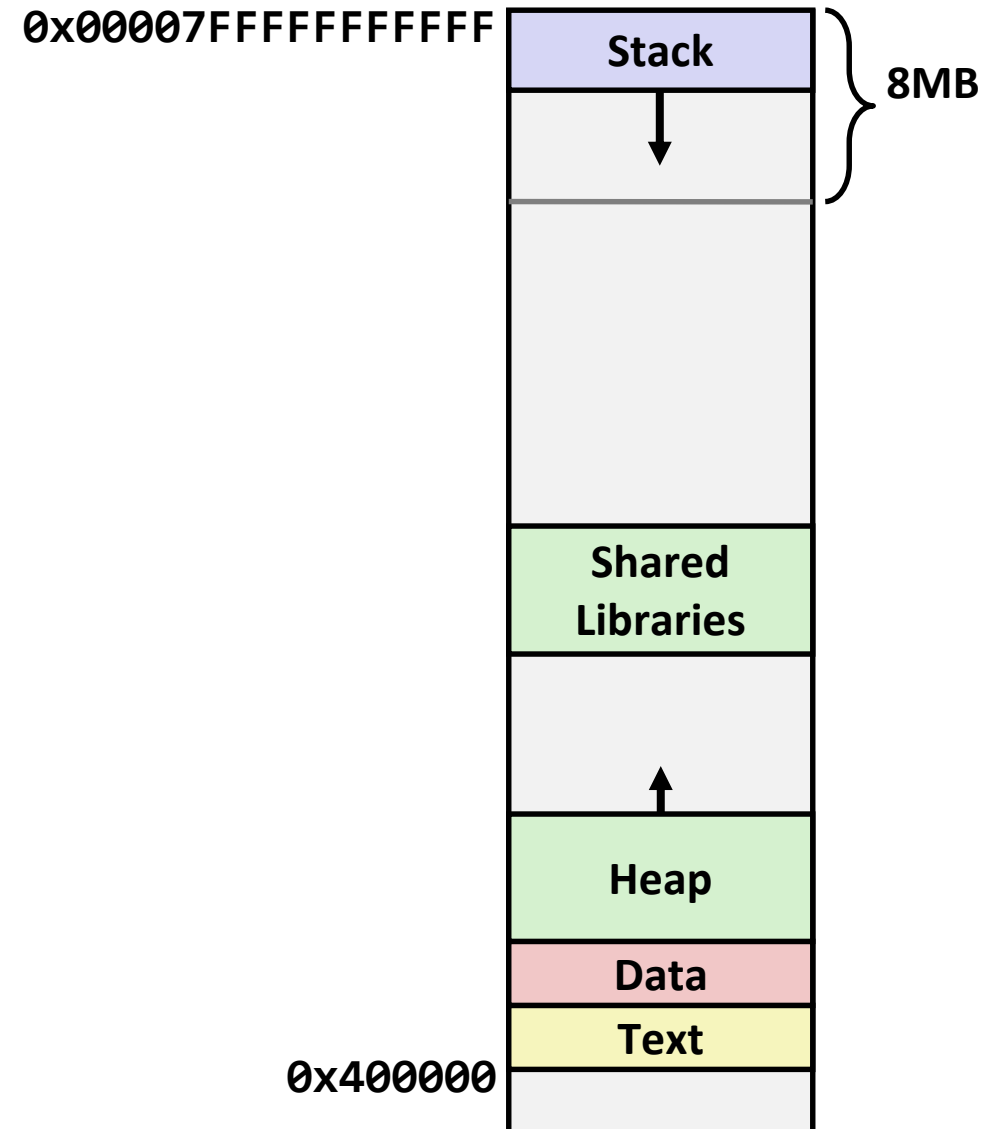
# Summary

- **Stack is the right data structure for procedure call / return**
  - Private storage for each instance of procedure call
  - Recursion handled by normal calling conventions
- **Mechanisms**
  - `call`, `ret`, `push`, `pop`, etc. instructions
  - Registers for passing arguments and return value
  - Stack memory
- **Policies**
  - Register usage (caller / callee save, `%rbp` & `%rsp`)
  - Stack frame organization

# Buffer Overflow

# x86-64/Linux Memory Layout

- **Stack**
  - Runtime stack (8MB limit)
- **Heap**
  - Dynamically allocated as needed
  - When call `malloc()`, `calloc()`, `new()`
- **Data**
  - Statically allocated data
  - e.g. global vars, static vars, string constants
- **Text / Shared libraries**
  - Executable machine instructions
  - Read-only

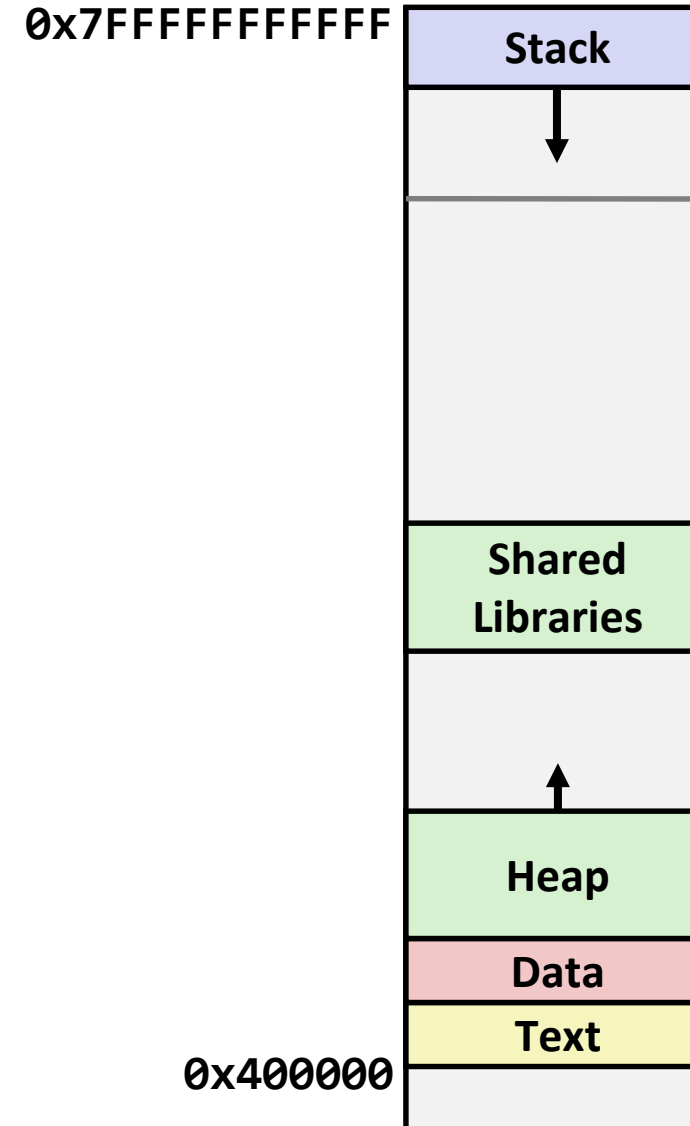


# x86-64 Addresses Example

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

int g = 1;
int main(void) {
    char *p = malloc(100);
    printf("main() = %p\n", main);
    printf("&g = %p\n", &g);
    printf("&p = %p\n", &p);
    printf("p = %p\n", p);
}
```

```
$ gcc -Og -g mem.c
$ ./a.out
main() = 0x4005f6
&g = 0x601048
&p = 0x7fff07b94b70
p = 0x1ece010
```





# Vulnerable Buffer Code

```
/* Echo Line */
void echo()
{
    // Way too small!
    char buf[4];
    gets(buf);
    puts(buf);
}

int main()
{
    printf("Type: ");
    echo();
    return 0;
}
```

```
$ ./bufdemo
Type:012
012

$ ./bufdemo
Type: 01234567890123456789012
01234567890123456789012

$ ./bufdemo
Type: 012345678901234567890123
Segmentation fault (core dumped)
```

# String Library Code

- Implementation of Unix function `gets()`
  - No way to specify limit on number of characters to read

```
char *gets(char *dest) { // Get string from stdin
    int c = getc();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getc();
    }
    *p = '\0';
    return dest;
}
```

- Similar problems with other Unix functions
  - `strcpy`: copies string of arbitrary length
  - `scanf` / `fscanf` / `sscanf`, given `%s` conversion specification

# Buffer Overflow Disassembly

- **echo():**

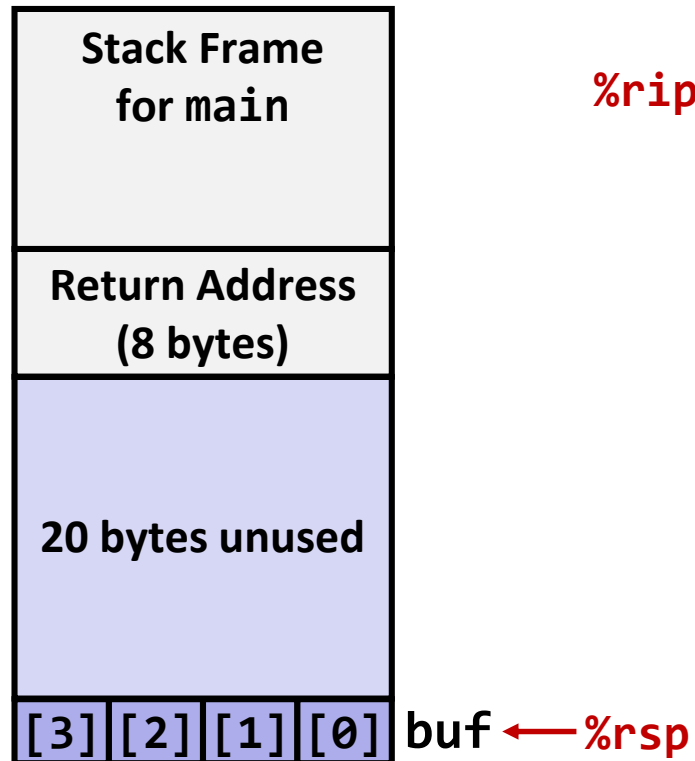
```
00000000004006cf <echo>:  
4006cf: 48 83 ec 18      sub    $0x18,%rsp  
4006d3: 48 89 e7         mov    %rsp,%rdi  
4006d6: e8 a5 ff ff ff  callq 400680 <gets>  
4006db: 48 89 e7         mov    %rsp,%rdi  
4006de: e8 3d fe ff ff  callq 400520 <puts@plt>  
4006e3: 48 83 c4 18     add    $0x18,%rsp  
4006e7: c3             retq
```

- **main():**

```
...  
4006ec: b8 00 00 00 00  mov    $0x0,%eax  
4006f1: e8 d9 ff ff ff  callq 4006cf <echo>  
4006f6: 48 83 c4 08     add    $0x8,%rsp  
...
```

# Buffer Overflow (I)

- Before call to `gets()`

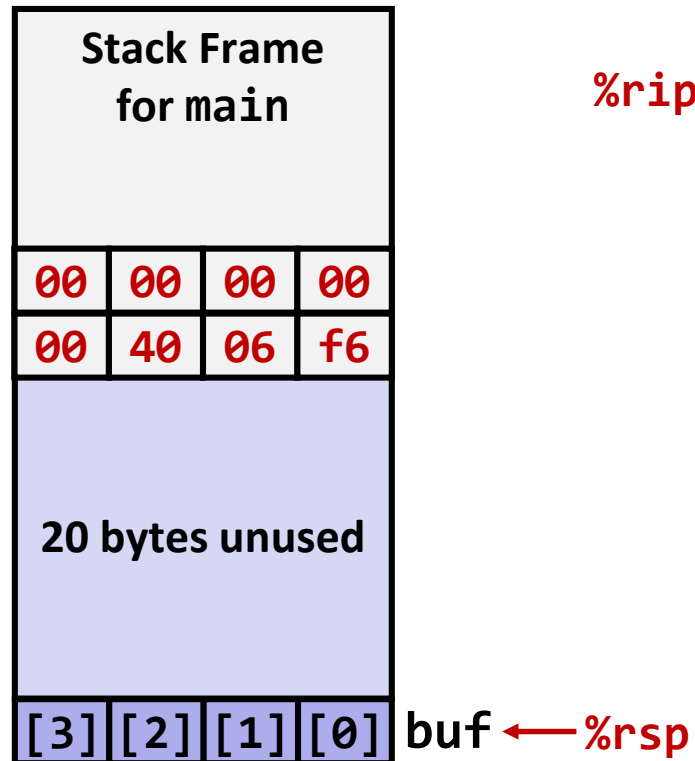


```
echo:
4006cf: sub    $0x18,%rsp
4006d3: mov    %rsp,%rdi
4006d6: callq 400680 <gets>
4006db: mov    %rsp,%rdi
4006de: callq 400520 <puts@plt>
4006e3: add   $0x18,%rsp
4006e7: retq

main:
...
4006ec: mov    $0x0,%eax
4006f1: callq 4006cf <echo>
4006f6: add   $0x8,%rsp
...
```

# Buffer Overflow (2)

- Before call to `gets()`



```
echo:
4006cf:  sub    $0x18,%rsp
4006d3:  mov    %rsp,%rdi
4006d6:  callq 400680 <gets>
4006db:  mov    %rsp,%rdi
4006de:  callq 400520 <puts@plt>
4006e3:  add    $0x18,%rsp
4006e7:  retq

main:
...
4006ec:  mov    $0x0,%eax
4006f1:  callq 4006cf <echo>
4006f6:  add    $0x8,%rsp
...
```

# Buffer Overflow (3)

- Overflowed buffer, but did not corrupt state

Stack Frame for main			
00	00	00	00
00	40	06	f6
00	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ← %rsp

echo:

```
4006cf: sub    $0x18,%rsp
4006d3: mov    %rsp,%rdi
4006d6: callq 400680 <gets>
4006db: mov    %rsp,%rdi
4006de: callq 400520 <puts@plt>
4006e3: add    $0x18,%rsp
4006e7: retq
```

```
$ ./bufdemo
```

```
Type: 01234567890123456789012
01234567890123456789012
```

# Buffer Overflow (4)

- Overflowed buffer, and corrupted return pointer

Stack Frame for main			
00	00	00	00
00	40	06	00
00	32	31	30
39	38	37	36
35	34	33	32
31	30	29	28
27	26	25	24
23	22	21	20

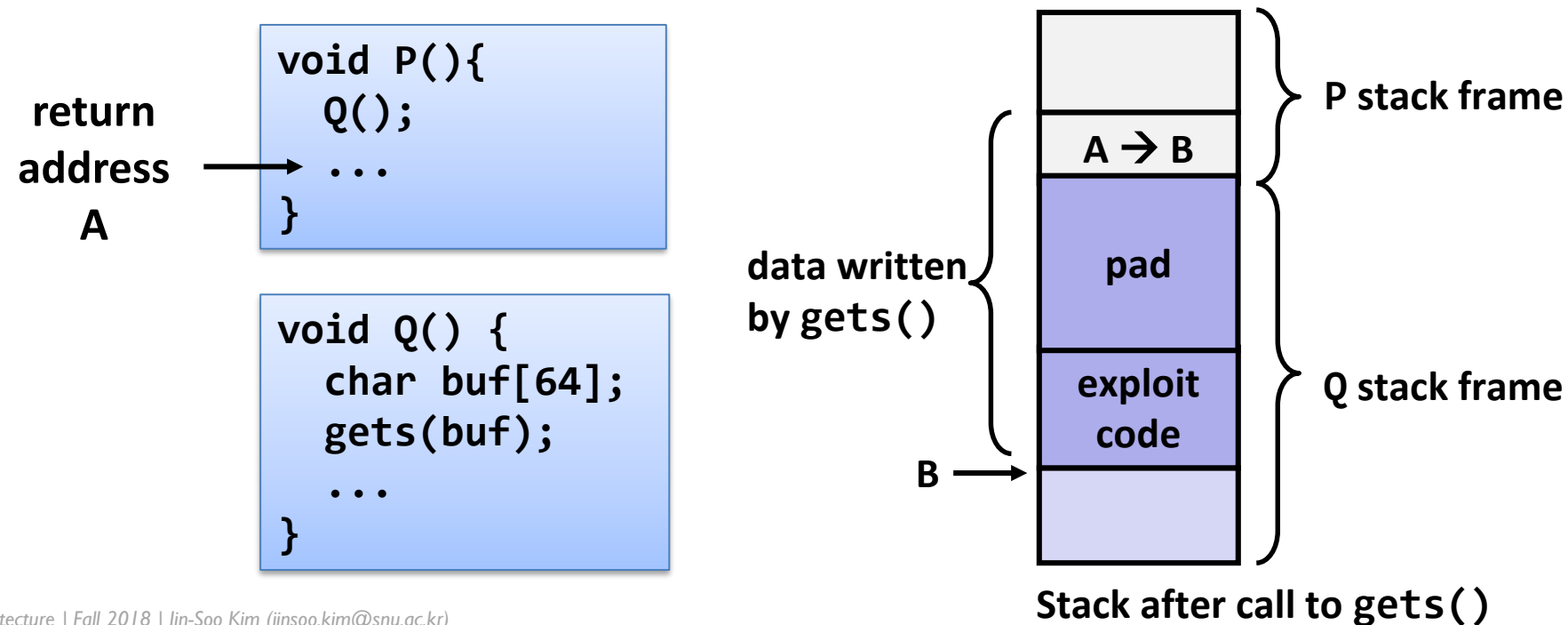
buf ← %rsp

```
echo:
4006cf: sub    $0x18,%rsp
4006d3: mov    %rsp,%rdi
4006d6: callq 400680 <gets>
4006db: mov    %rsp,%rdi
4006de: callq 400520 <puts@plt>
4006e3: add    $0x18,%rsp
4006e7: retq
```

```
$ ./bufdemo
Type: 012345678901234567890123
Segmentation fault (core dumped)
```

# Buffer Overflow Attack

- Malicious use of buffer overflow
  - Input string contains byte representation of executable code
  - Overwrite return address A with address of buffer B
  - When P() executes `ret`, will jump to exploit code





# Exploits Using Buffer Overflows

- Buffer overflow bugs can allow remote machines to execute arbitrary code on victim machines
- Distressingly common in real programs
  - Programmers keep making the same mistakes ☹️
  - Recent measures make these attacks much more difficult

## Google Security Blog

The latest news and insights from Google on security and safety on the Internet

CVE-2015-7547: glibc getaddrinfo stack-based buffer overflow

February 16, 2016

# Avoiding Buffer Overflow

- Use library routines that limit string lengths
  - `fgets()` instead of `gets()`
  - `strncpy()` instead of `strcpy()`
  - Don't use `scanf()` with `%s` conversion specification
    - Use `fgets()` to read the string
    - Or use `%ns` where `n` is a suitable integer

```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```

# System-Level Protections

- **Randomized stack offsets**
  - At start of program, allocate random amount of space on stack
  - Makes it difficult for hacker to predict beginning of inserted code
  
- **Executable space protection**
  - Mark certain areas of memory as non-executable (e.g. stack)
  - Requires hardware assistance:  
x86-64 added explicit “execute” permission

# Stack Canaries

## ■ Idea

- Place special value (“canary”) on stack just beyond buffer
- Check for corruption before exiting function

## ■ GCC implementation

- `-fstack-protector` (now the default)

```
$ ./bufdemo                                bufdemo with -fstack-protector
Type:01234567
01234567
$ ./bufdemo
Type: 012345678
012345678
*** stack smashing detected ***: ./bufdemo terminated
Aborted (core dumped)
```

# Summary

- **Memory layout**
  - OS/machine dependent (including kernel version)
  - Basic partitioning:  
stack, data, text, heap, shared libraries found in most machines
- **Avoiding buffer overflow vulnerability**
  - Important to use library routines that limit string lengths
- **Working with strange code**
  - Important to analyze nonstandard cases
    - e.g. What happens when stack corrupted due to buffer overflow?
  - Helps to step through with GDB