

The Stack & Procedures

CSE 351 Summer 2018

Instructor:

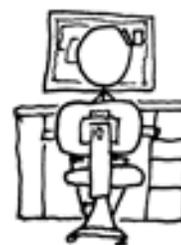
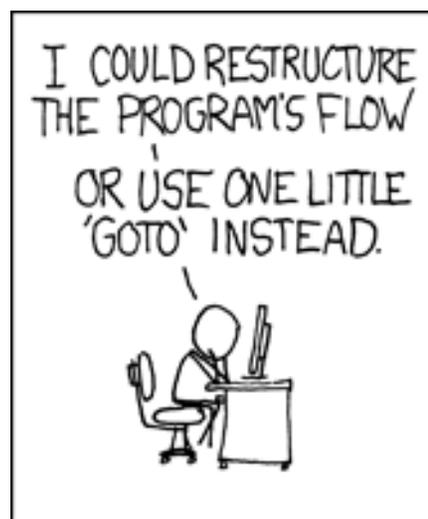
Justin Hsia

Teaching Assistants:

Josie Lee

Natalie Andreeva

Teagan Horkan



<http://xkcd.com/292/>

Administrivia

- ❖ Homework 2 due tonight
- ❖ Lab 2 due Monday (7/16)
- ❖ Homework 3 released
 - On midterm material, but due after the midterm
- ❖ **Midterm** (7/18, in lecture)
 - Reference sheet + 1 *handwritten* cheat sheet
 - Find a study group! Look at past exams!
 - Average is typically around 70%
 - **Review session** (7/16) in EEB 045 from 5:00-6:30 pm

Procedure Control Flow

- ❖ Use stack to support procedure call and return
- ❖ Procedure call: **call** *label*
 - 1) Push return address on stack (*why? which address?*)
 - 2) Jump to *label*

Procedure Control Flow

- ❖ Use stack to support procedure call and return
- ❖ Procedure call: **call** *label*
 - 1) Push return address on stack (*why? which address?*)
 - 2) Jump to *label*
- ❖ Return address:
 - Address of instruction immediately after **call** instruction
 - Example from disassembly:

```
400544: call    400550 <mult2>
400549: movq   %rax, (%rbx)
```

Return address = **0x400549**

- ❖ Procedure return: **ret**
 - 1) Pop return address from stack
 - 2) Jump to address
- next instruction happens to be a move, but could be anything

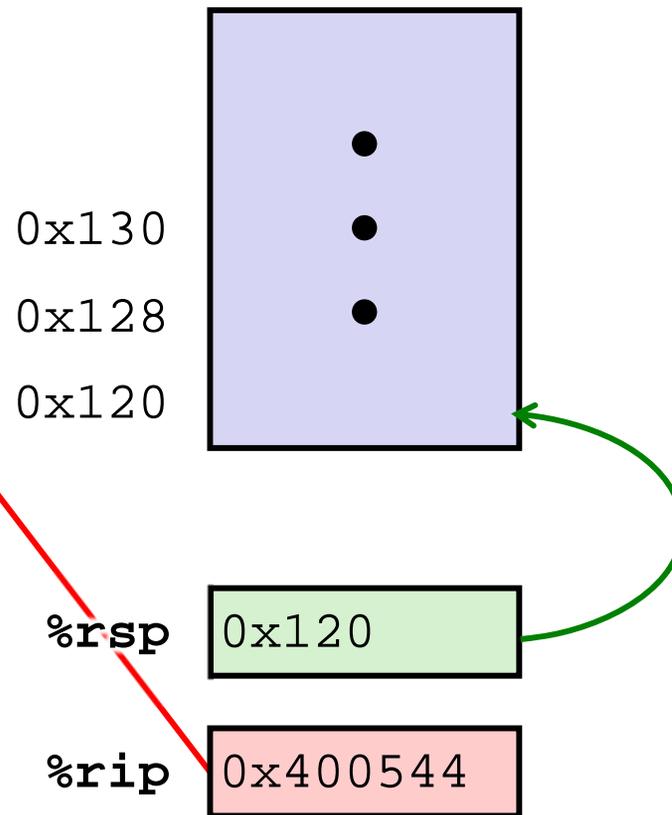
Procedure Call Example (step 1)

```

00000000000400540 <multstore>:
.
.
400544: call    400550 <mult2>
400549: movq   %rax, (%rbx)
.
.
    
```

```

00000000000400550 <mult2>:
400550: movq   %rdi,%rax
.
.
400557: ret
    
```



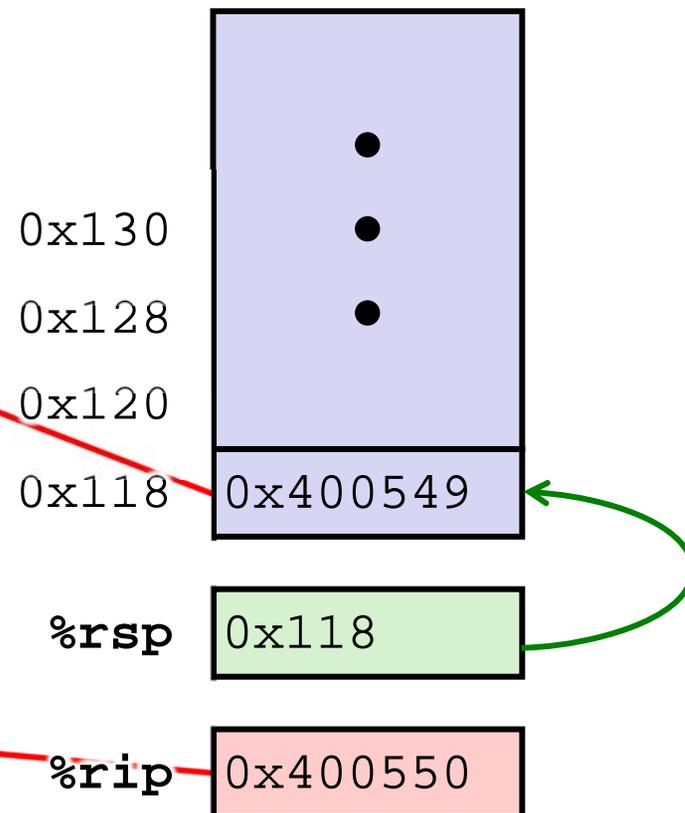
Procedure Call Example (step 2)

```

00000000000400540 <multstore>:
.
.
400544: call    400550 <mult2>
400549: movq   %rax, (%rbx)
.
.
    
```

```

00000000000400550 <mult2>:
400550: movq   %rdi, %rax
.
.
400557: ret
    
```



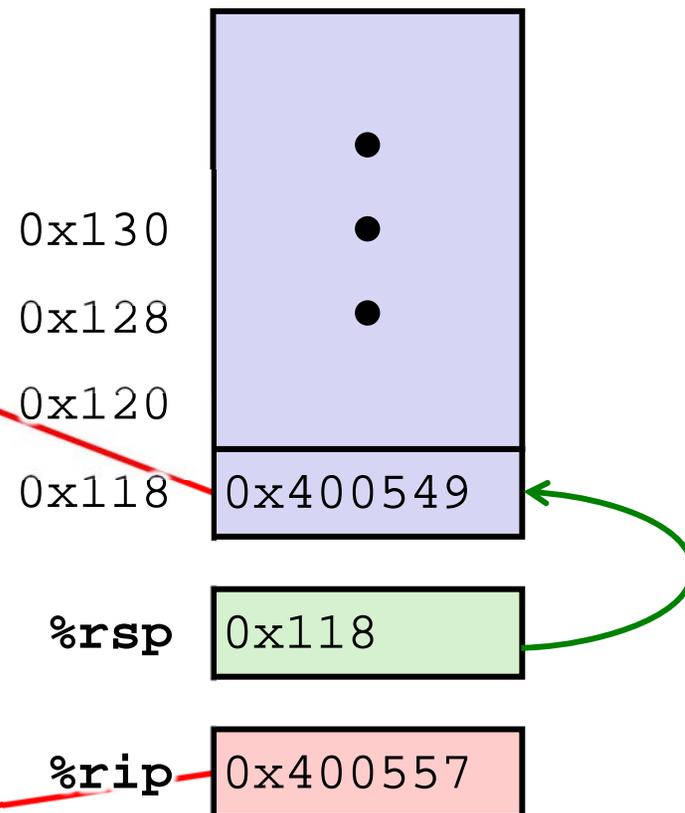
Procedure Return Example (step 1)

```

00000000000400540 <multstore>:
.
.
400544: call    400550 <mult2>
400549: movq   %rax, (%rbx)
.
.
    
```

```

00000000000400550 <mult2>:
400550: movq   %rdi,%rax
.
.
400557: ret
    
```



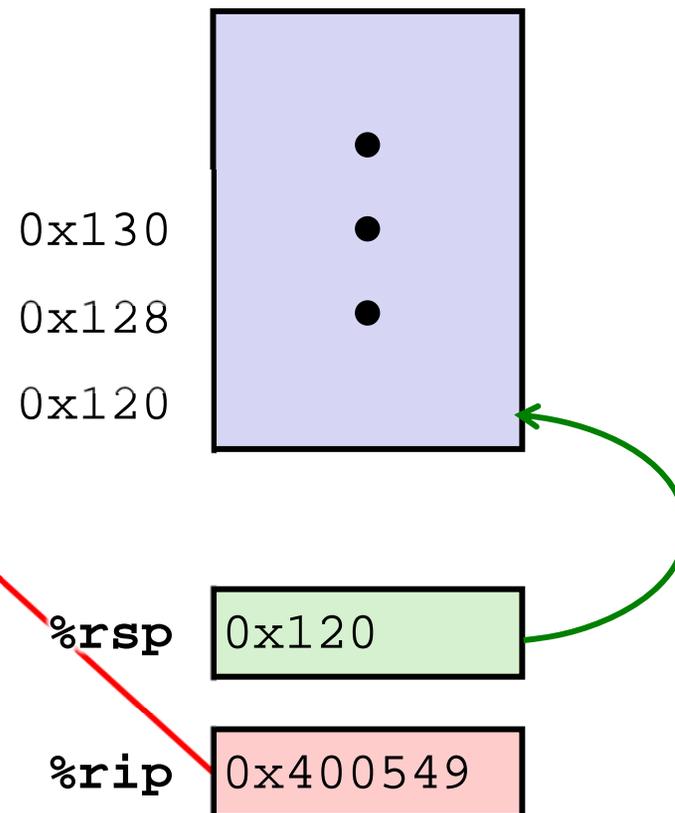
Procedure Return Example (step 2)

```

00000000000400540 <multstore>:
.
.
400544: call    400550 <mult2>
400549: movq   %rax, (%rbx)
.
.
    
```

```

00000000000400550 <mult2>:
400550: movq   %rdi,%rax
.
.
400557: ret
    
```



Procedures

- ❖ Stack Structure
- ❖ **Calling Conventions**
 - Passing control
 - **Passing data**
 - Managing local data
- ❖ Register Saving Conventions
- ❖ Illustration of Recursion

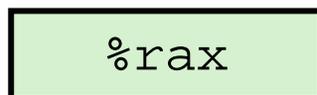
Procedure Data Flow

Registers (**NOT in Memory**)

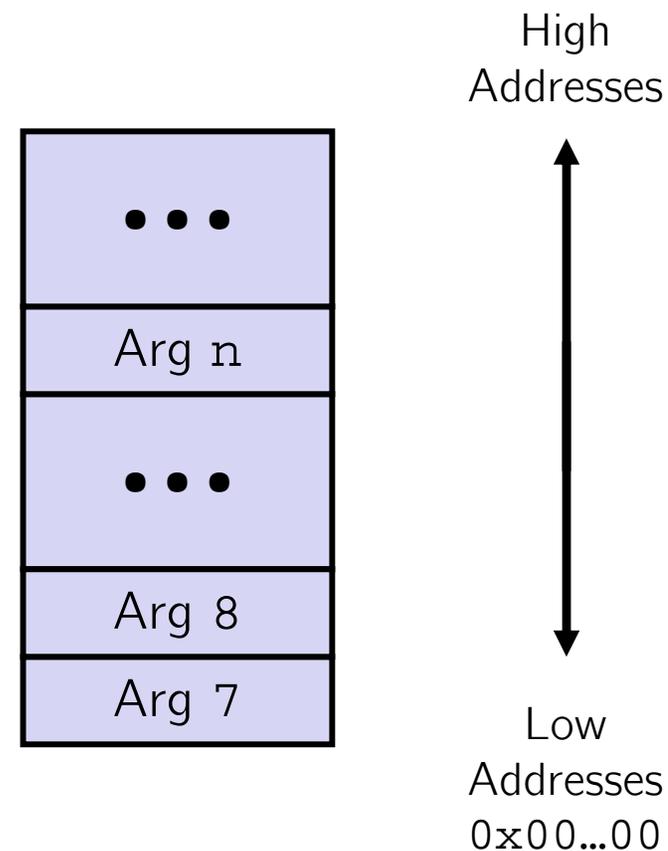
❖ First 6 arguments



❖ Return value



Stack (**Memory**)



- Only allocate stack space when needed

x86-64 Return Values

- ❖ By convention, values returned by procedures are placed in `%rax`
 - Choice of `%rax` is arbitrary
- 1) **Caller** must make sure to save the contents of `%rax` before calling a **callee** that returns a value
 - Part of register-saving convention
- 2) **Callee** places return value into `%rax`
 - Any type that can fit in 8 bytes – integer, float, pointer, etc.
 - For return values greater than 8 bytes, best to return a *pointer* to them
- 3) Upon return, **caller** finds the return value in `%rax`

Data Flow Examples

```
void multstore
(long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
    # x in %rdi, y in %rsi, dest in %rdx
    ...
400541: movq    %rdx,%rbx    # Save dest
400544: call   400550 <mult2> # mult2(x,y)
    # t in %rax
400549: movq    %rax,(%rbx)  # Save at dest
    ...
```

```
long mult2
(long a, long b)
{
    long s = a * b;
    return s;
}
```

```
0000000000400550 <mult2>:
    # a in %rdi, b in %rsi
400550: movq    %rdi,%rax    # a
400553: imulq   %rsi,%rax    # a * b
    # s in %rax
400557: ret                    # Return
```

Procedures

- ❖ Stack Structure
- ❖ **Calling Conventions**
 - Passing control
 - Passing data
 - **Managing local data**
- ❖ Register Saving Conventions
- ❖ Illustration of Recursion

Stack-Based Languages

- ❖ Languages that support recursion
 - e.g. C, Java, most modern languages
 - Code must be re-entrant
 - Multiple simultaneous instantiations of single procedure
 - Need some place to store *state* of each instantiation
 - Arguments, local variables, return pointer
- ❖ Stack allocated in frames
 - State for a single procedure instantiation
- ❖ Stack discipline
 - State for a given procedure needed for a limited time
 - Starting from when it is called to when it returns
 - Callee always returns before caller does

Call Chain Example

```

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

```

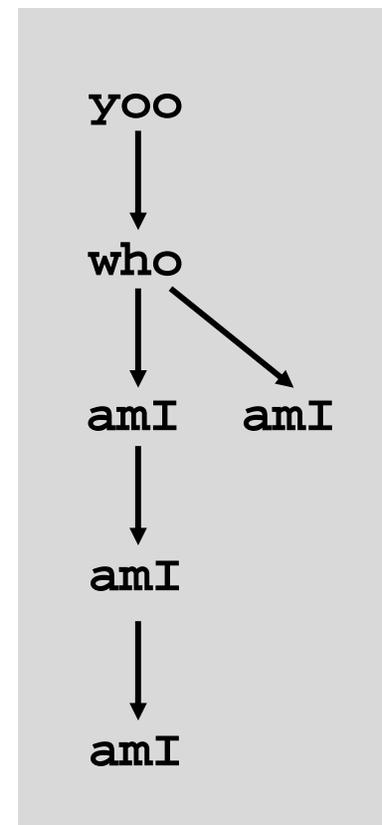
who(...)
{
    •
    amI();
    •
    amI();
    •
}
    
```

```

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

Procedure amI is recursive
(calls itself)

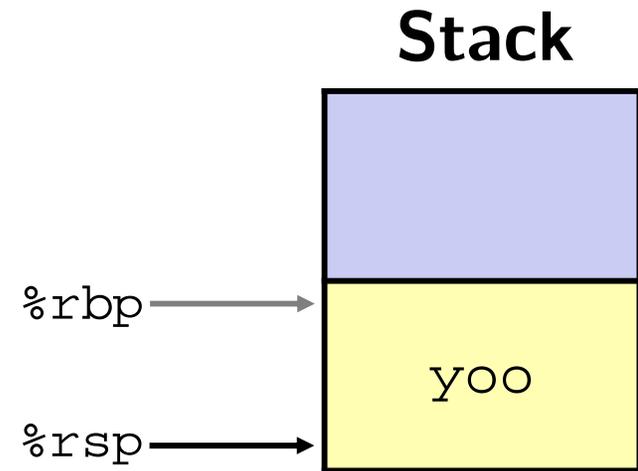
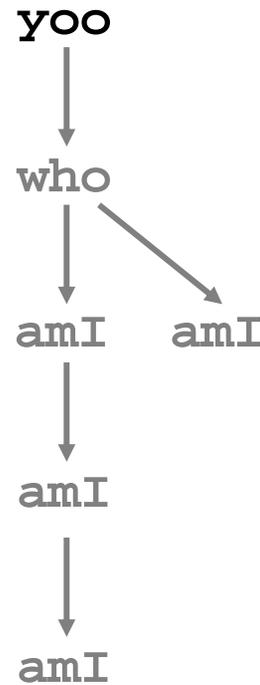
Example
Call Chain



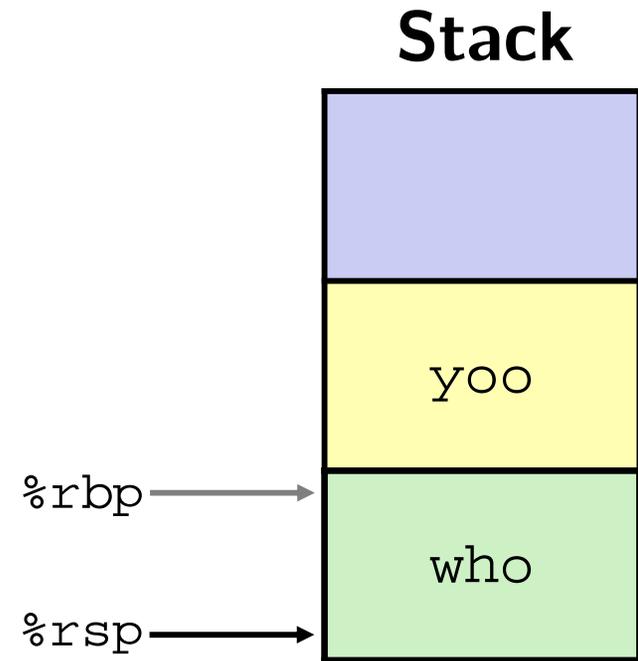
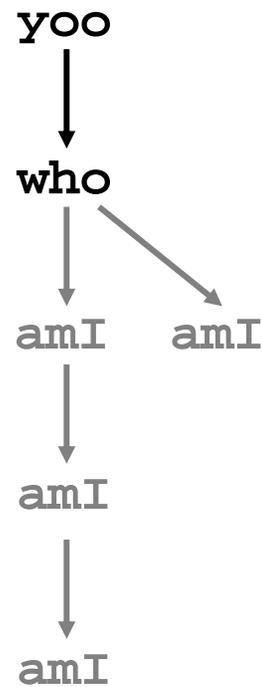
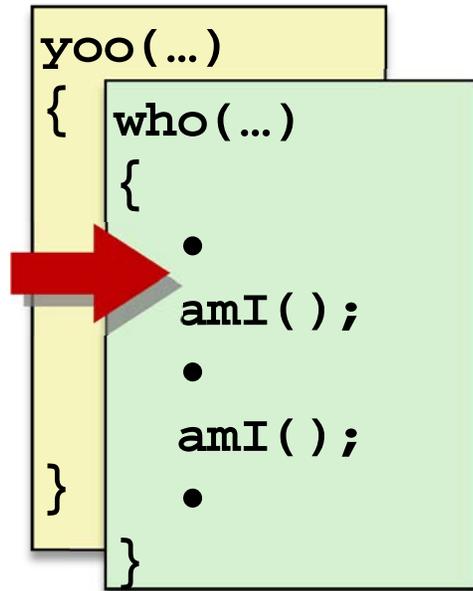
1) Call to yoo

```

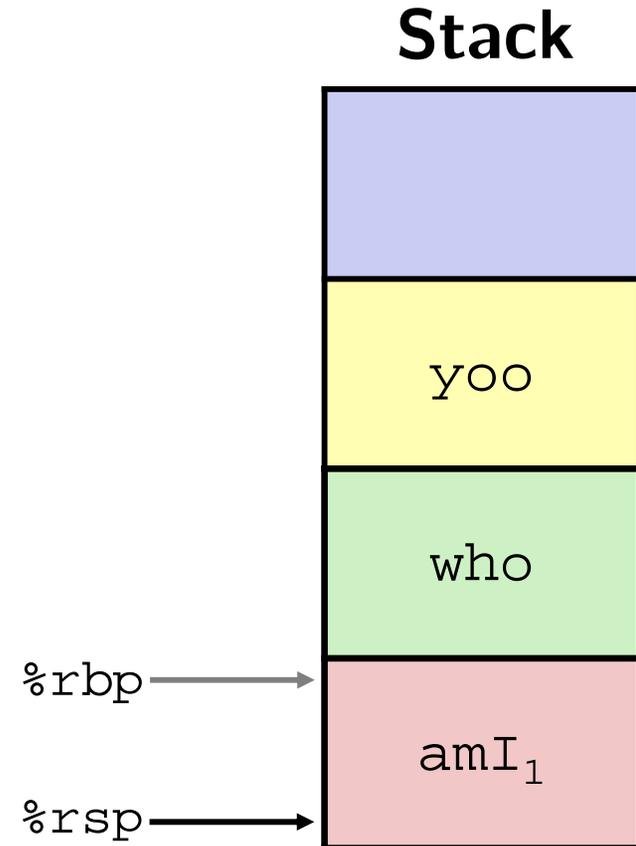
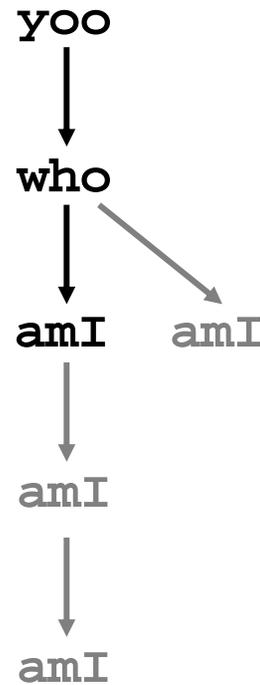
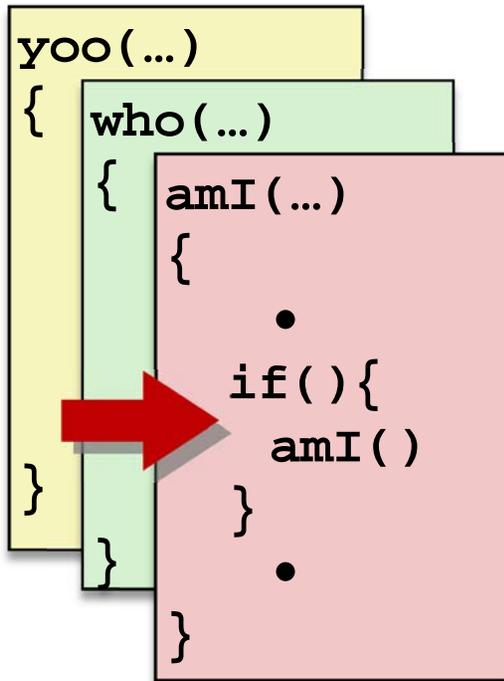
yoo(...)
{
  •
  •
  who();
  •
  •
}
    
```

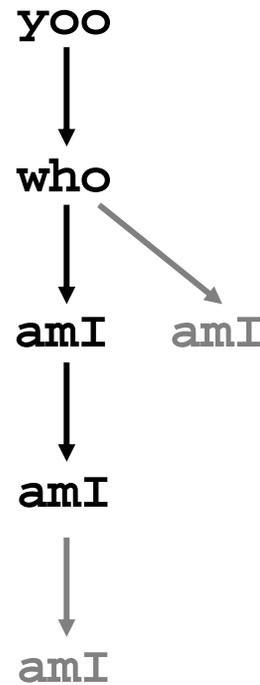
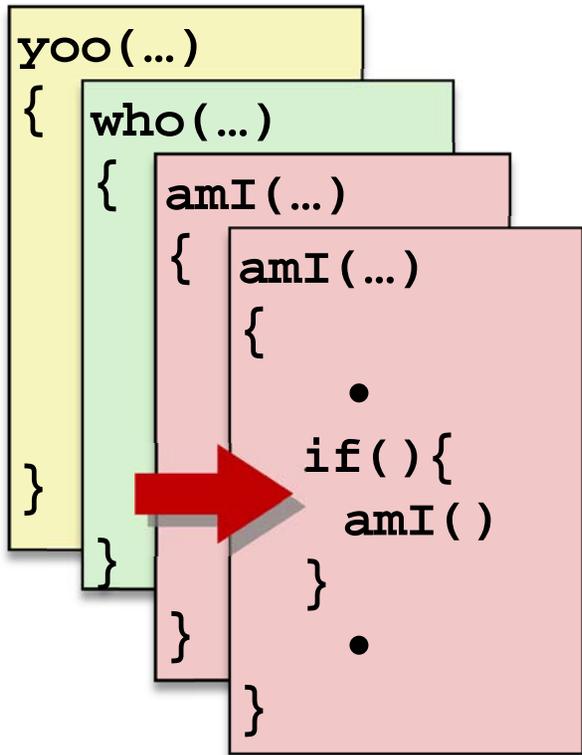
2) Call to who



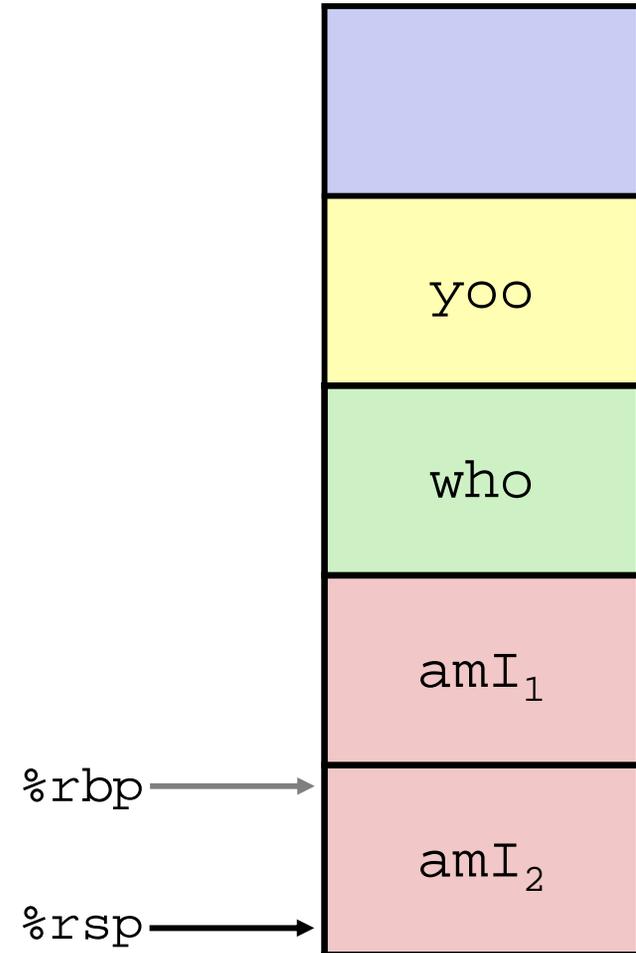
3) Call to amI (1)



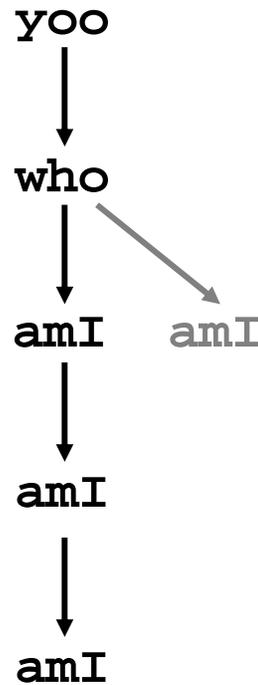
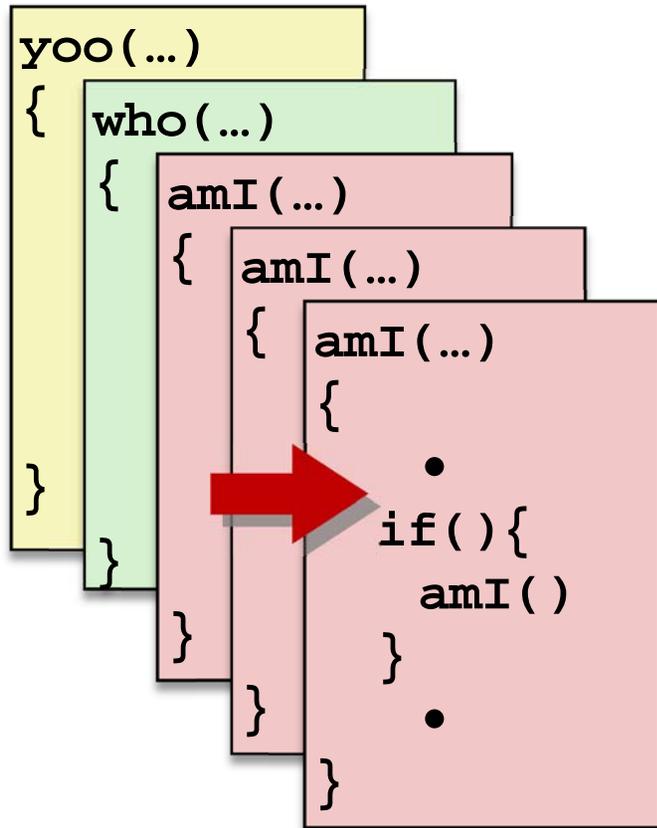
4) Recursive call to amI (2)



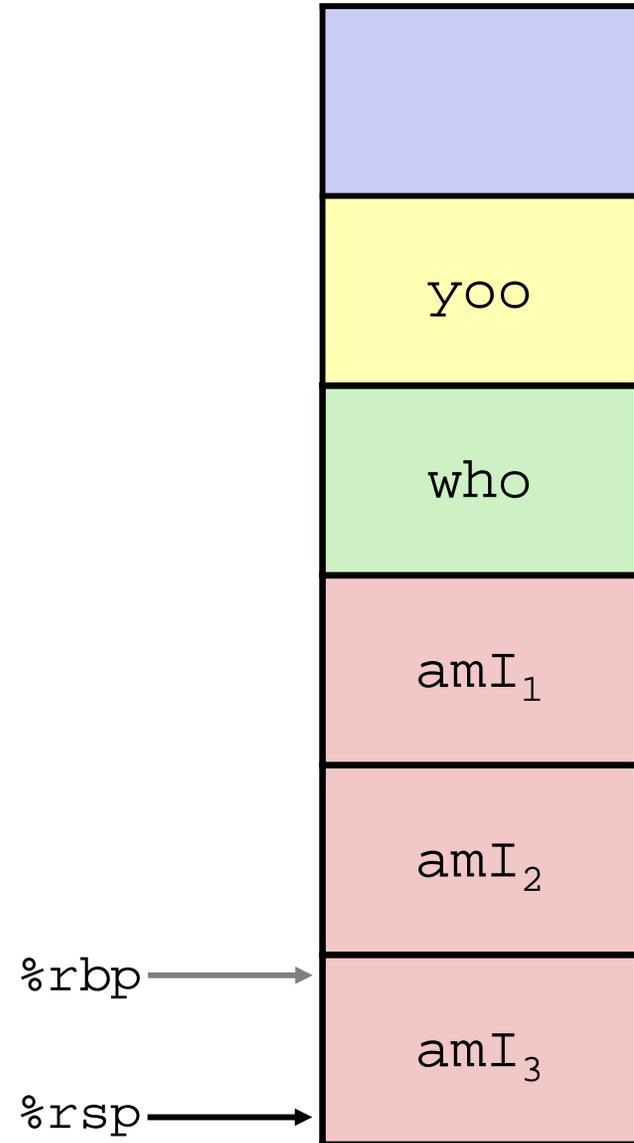
Stack



5) (another) Recursive call to amI (3)

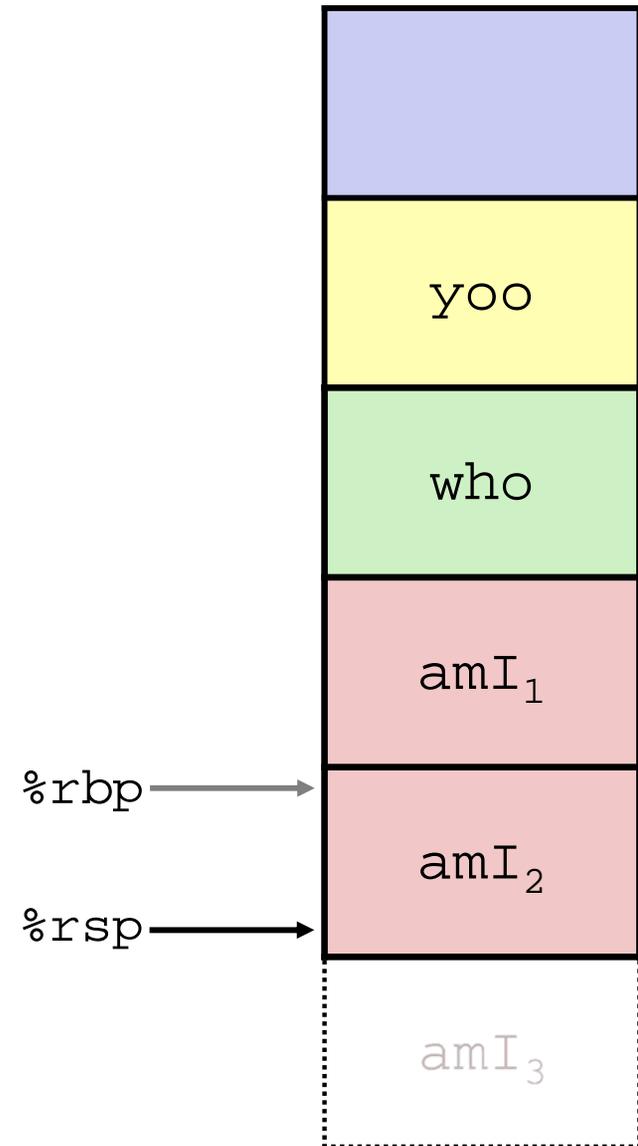
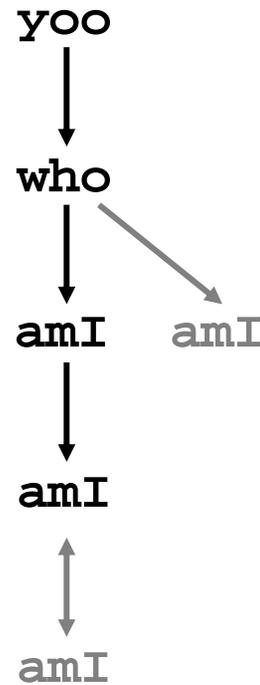
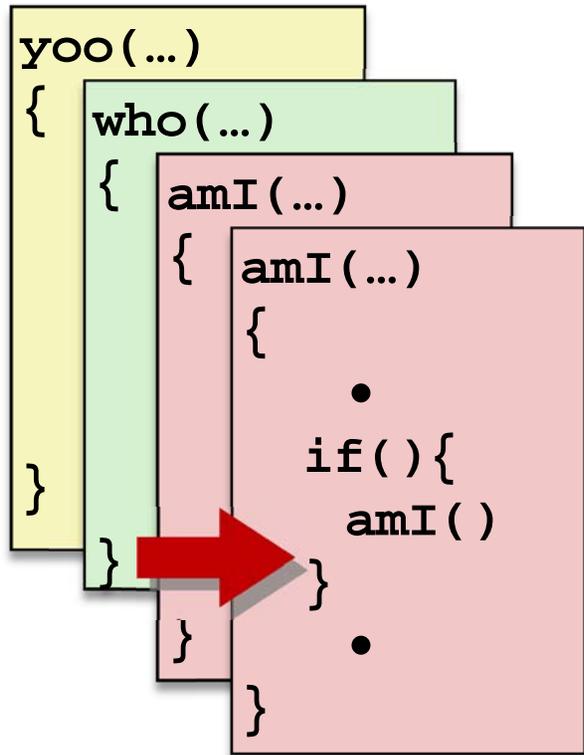


Stack



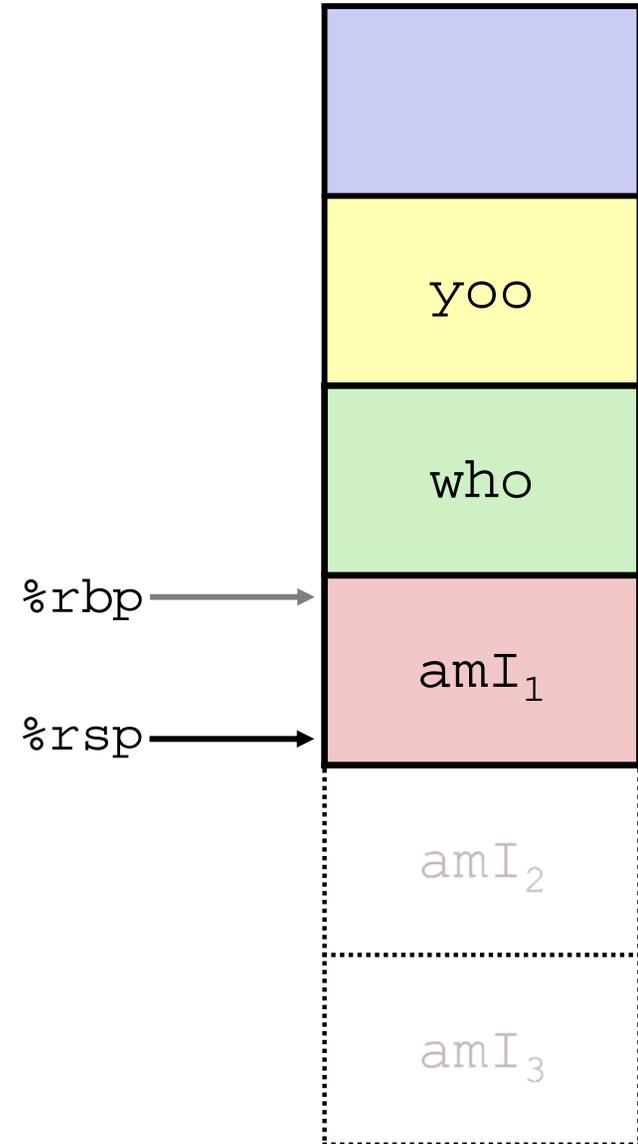
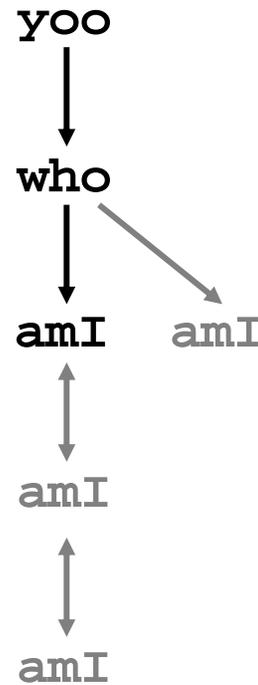
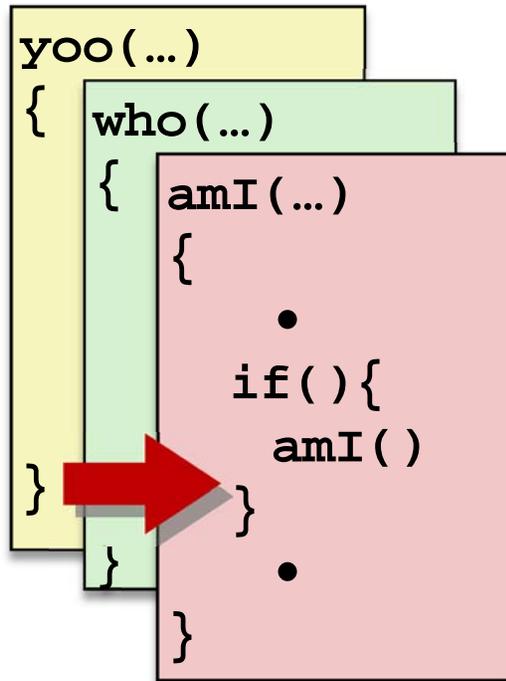
6) Return from (another) recursive call to amI

Stack

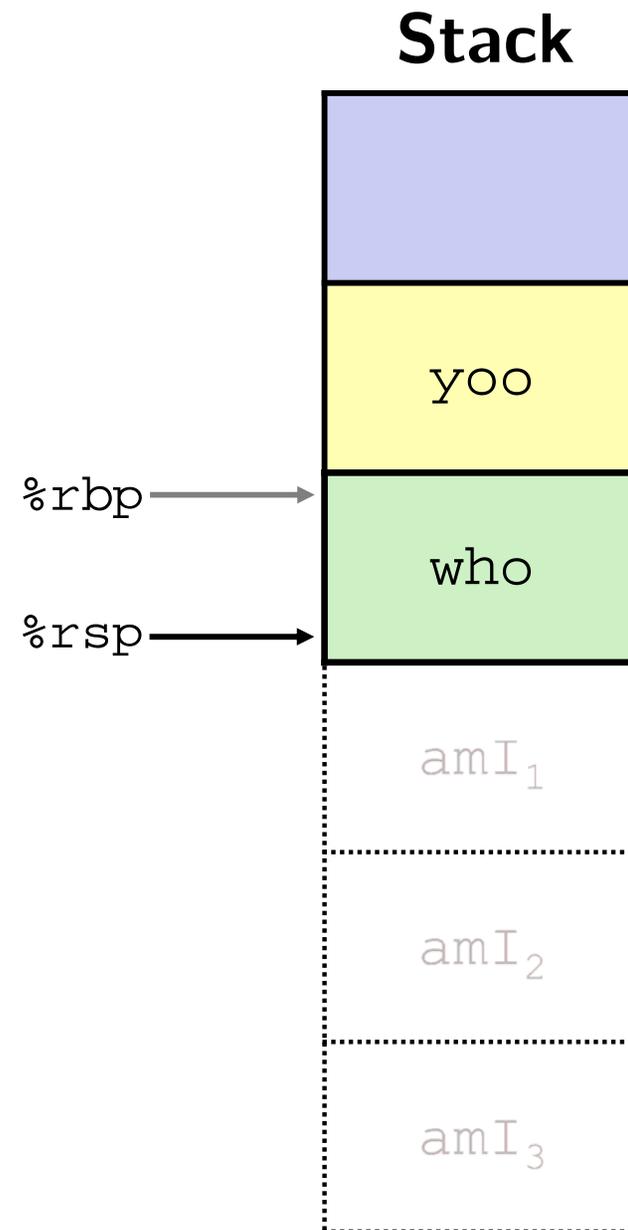
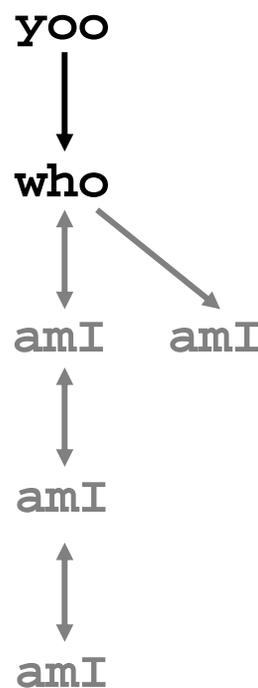
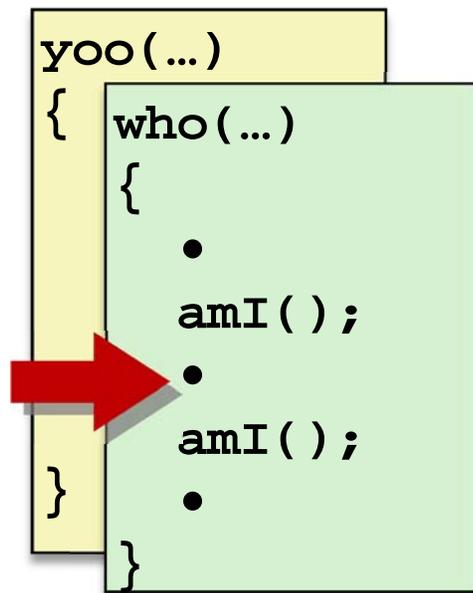


7) Return from recursive call to amI

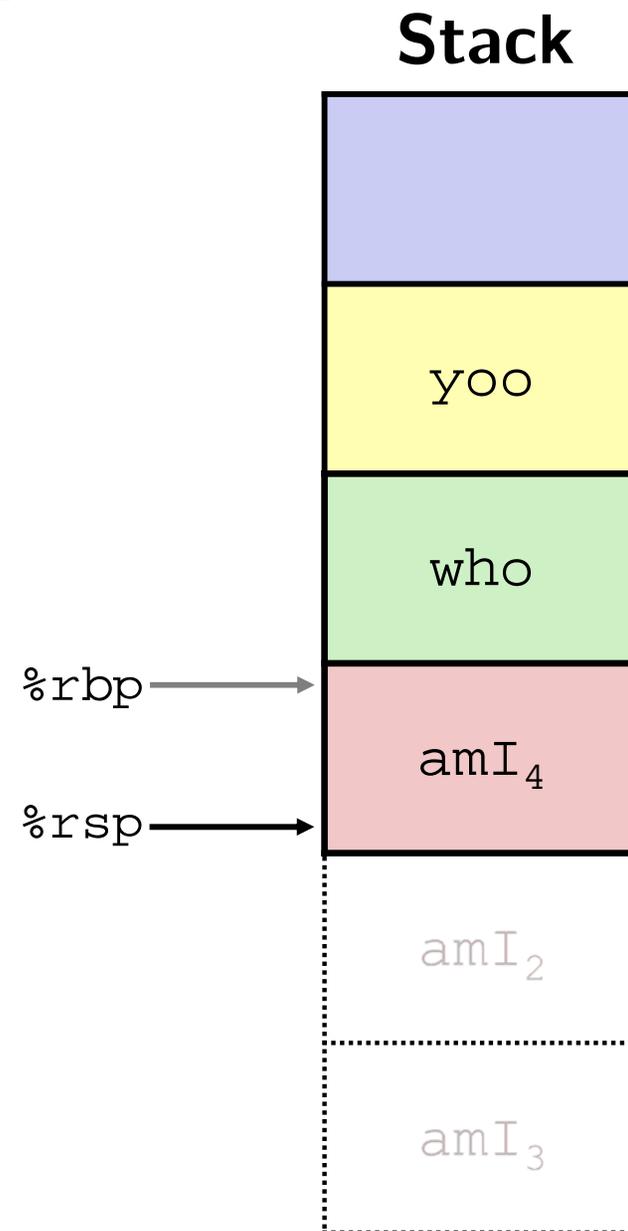
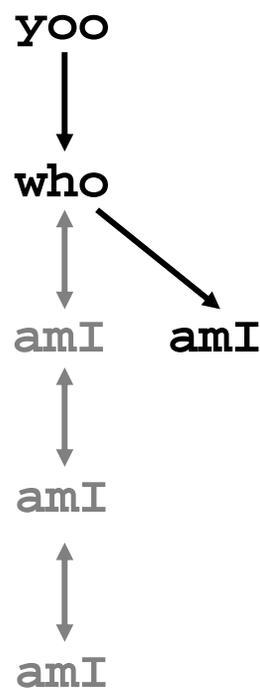
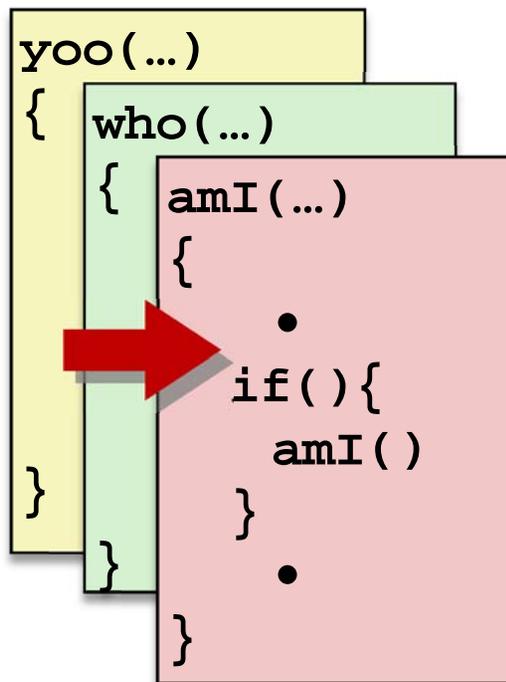
Stack



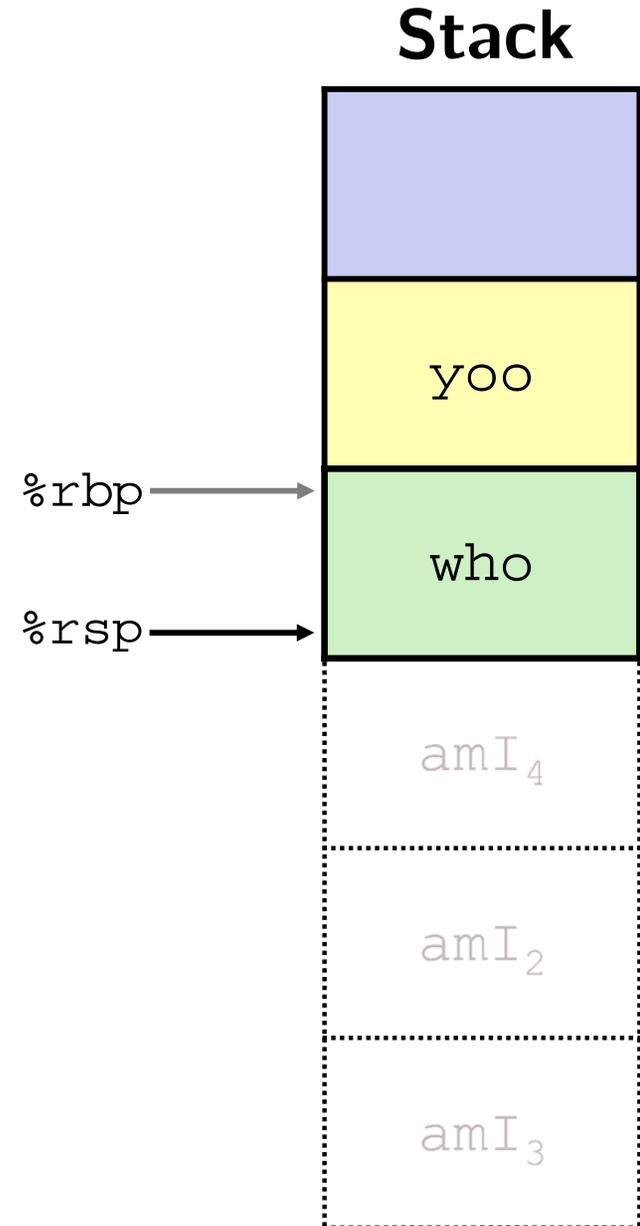
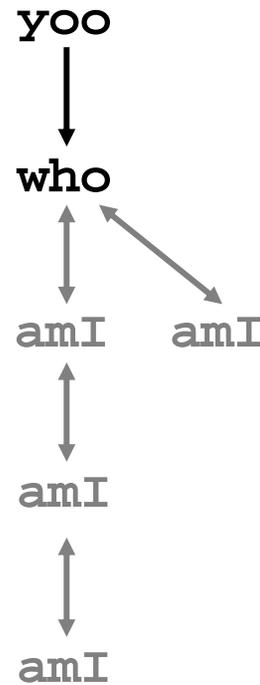
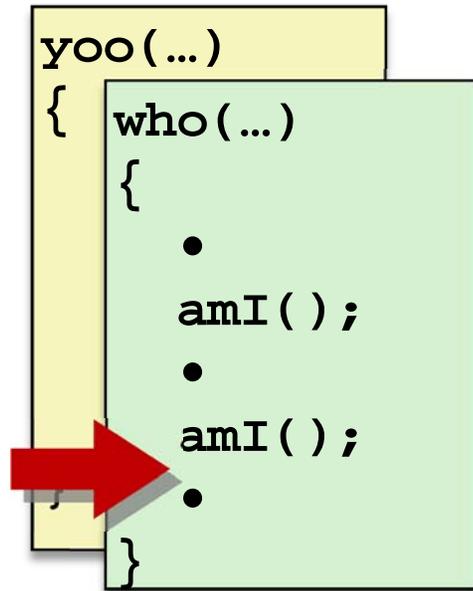
8) Return from call to amI



9) (second) Call to amI (4)



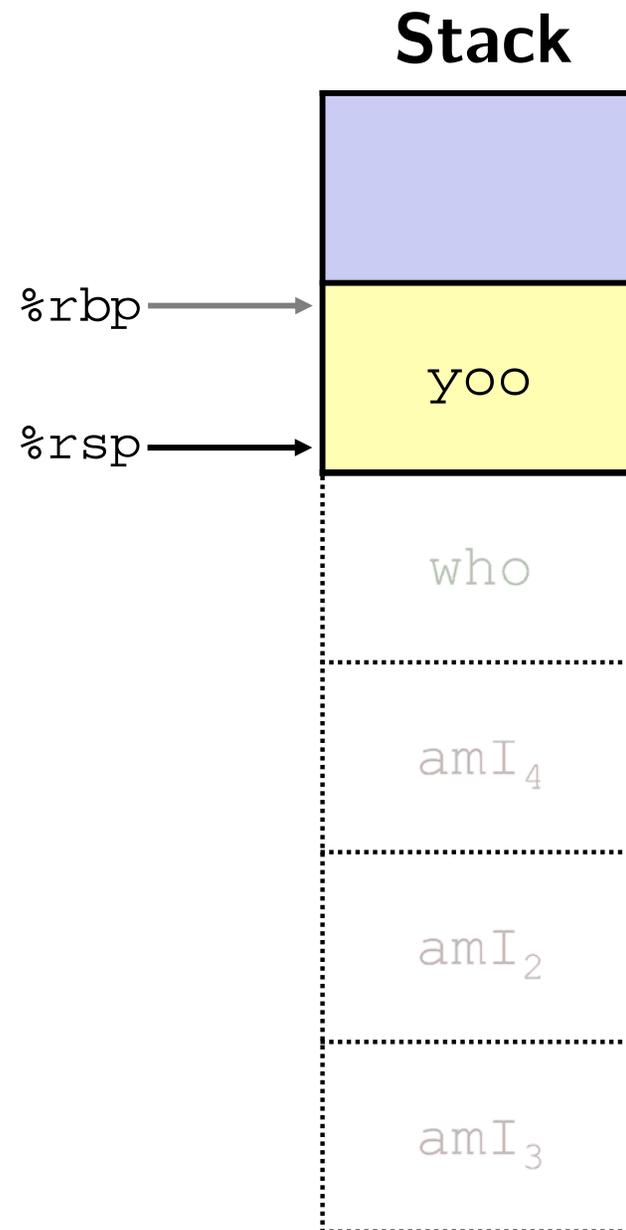
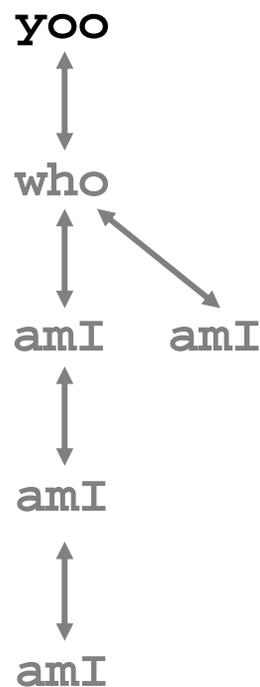
10) Return from (second) call to amI



11) Return from call to who

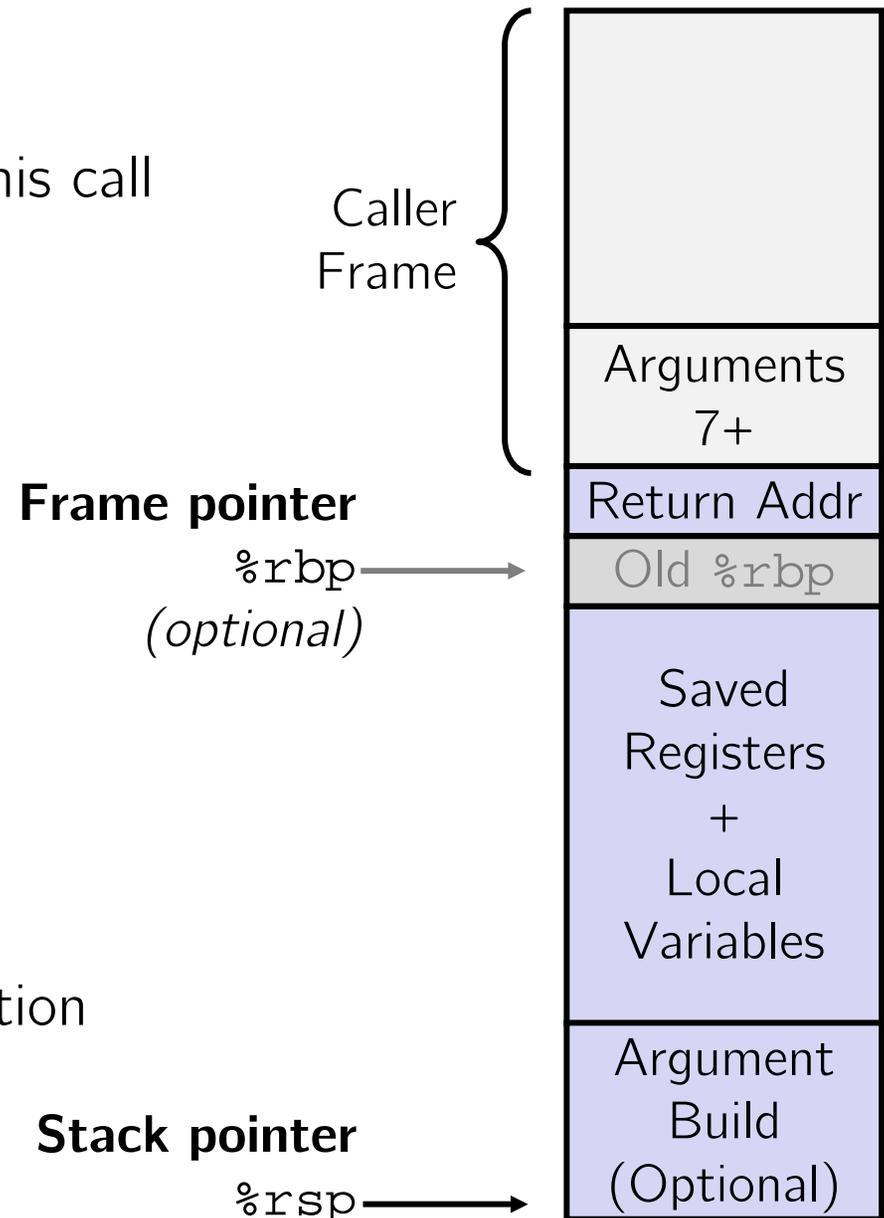
```

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

x86-64/Linux Stack Frame

- ❖ Caller's Stack Frame
 - Extra arguments (if > 6 args) for this call
- ❖ Current/**Callee** Stack Frame
 - Return address
 - Pushed by **call** instruction
 - Old frame pointer (optional)
 - Saved register context (when reusing registers)
 - Local variables (if can't be kept in registers)
 - "Argument build" area (if callee needs to call another function
 - parameters for function about to be called, if needed)



Peer Instruction Question

- ❖ Answer the following questions about when `main()` is run (assume `x` and `y` stored on the Stack):

```
int main() {
    int i, x = 0;
    for(i=0;i<3;i++)
        x = randSum(x);
    printf("x = %d\n",x);
    return 0;
}
```

```
int randSum(int n) {
    int y = rand()%20;
    return n+y;
}
```

- *Higher/larger address:* `x` or `y`?
- How many total stack frames are *created*?
- What is the maximum *depth* (# of frames) of the Stack?

A. 1 **B. 2** **C. 3** **D. 4**

Vote only on 3rd question at <http://PollEv.com/justinh>

Example: increment

```
long increment(long *p, long val) {  
    long x = *p;  
    long y = x + val;  
    *p = y;  
    return x;  
}
```

increment:

```
movq    (%rdi), %rax  
addq    %rax, %rsi  
movq    %rsi, (%rdi)  
ret
```

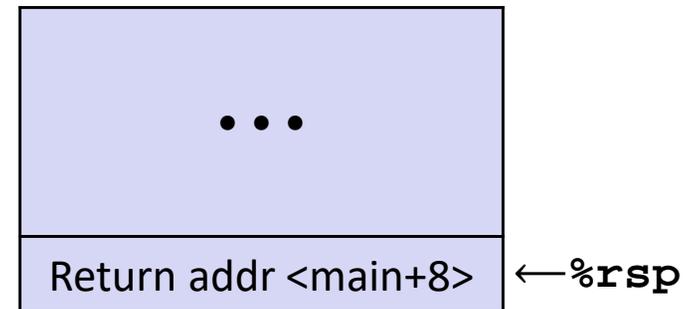
Register	Use(s)
%rdi	1 st arg (p)
%rsi	2 nd arg (val), y
%rax	x, return value

Procedure Call Example (initial state)

```
long call_incr() {  
    long v1 = 351;  
    long v2 = increment(&v1, 100);  
    return v1+v2;  
}
```

```
call_incr:  
    subq    $16, %rsp  
    movq    $351, 8(%rsp)  
    movl    $100, %esi  
    leaq    8(%rsp), %rdi  
    call    increment  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

Initial Stack Structure



- ❖ Return address on stack is the address of instruction immediately *following* the call to “call_incr”
 - Shown here as main, but could be anything)
 - Pushed onto stack by call call_incr

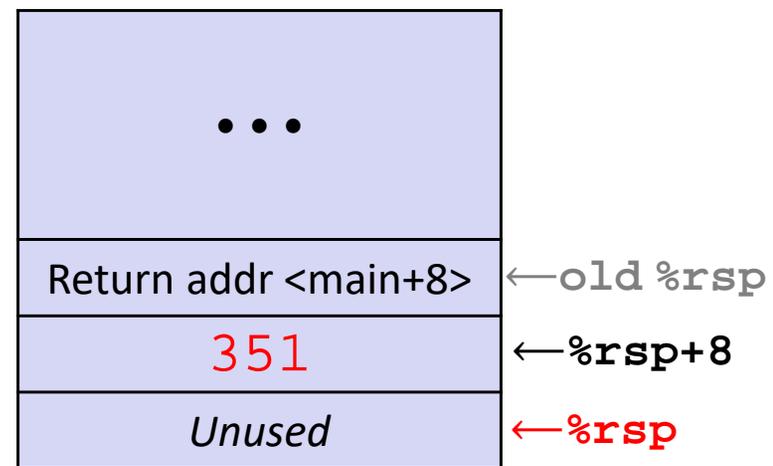
Procedure Call Example (step 1)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
    subq    $16, %rsp
    movq    $351, 8(%rsp)
    movl    $100, %esi
    leaq    8(%rsp), %rdi
    call    increment
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

} Allocate space for local vars

Stack Structure



- ❖ Setup space for local variables
 - Only `v1` needs space on the stack
- ❖ Compiler allocated extra space
 - Often does this for a variety of reasons, including alignment

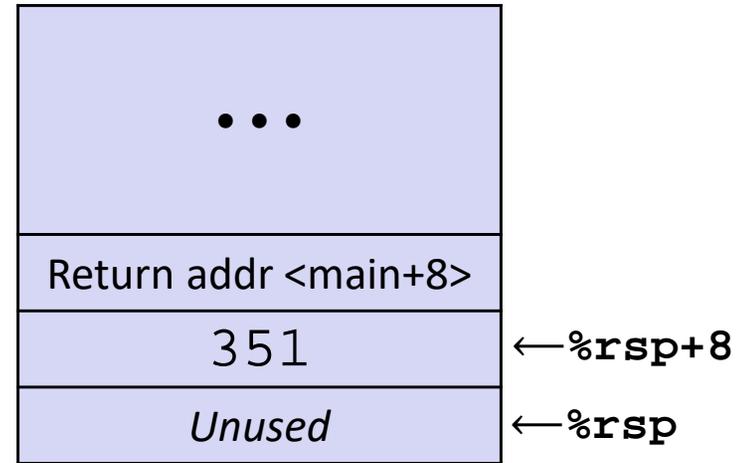
Procedure Call Example (step 2)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
    subq    $16, %rsp
    movq    $351, 8(%rsp)
    movl    $100, %esi
    leaq    8(%rsp), %rdi
    call    increment
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

} Set up parameters for call to increment

Stack Structure



Aside: movl is used because 100 is a small positive value that fits in 32 bits. High order bits of rsi get set to zero automatically. It takes *one less byte* to encode a movl than a movq.

Register	Use(s)
%rdi	&v1
%rsi	100

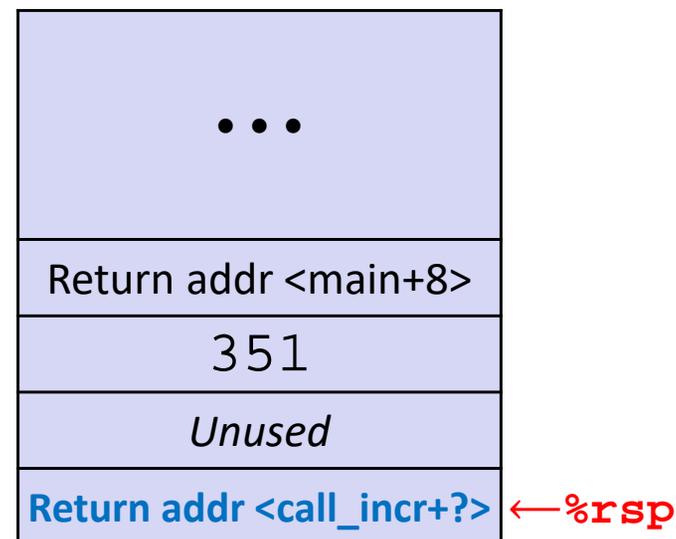
Procedure Call Example (step 3)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
    subq    $16, %rsp
    movq    $351, 8(%rsp)
    movl    $100, %esi
    leaq    8(%rsp), %rdi
    call   increment
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

```
increment:
    movq    (%rdi), %rax
    addq    %rax, %rsi
    movq    %rsi, (%rdi)
    ret
```

Stack Structure



- ❖ State while inside increment
 - **Return address** on top of stack is address of the addq instruction immediately following call to increment

Register	Use(s)
%rdi	&v1
%rsi	100
%rax	

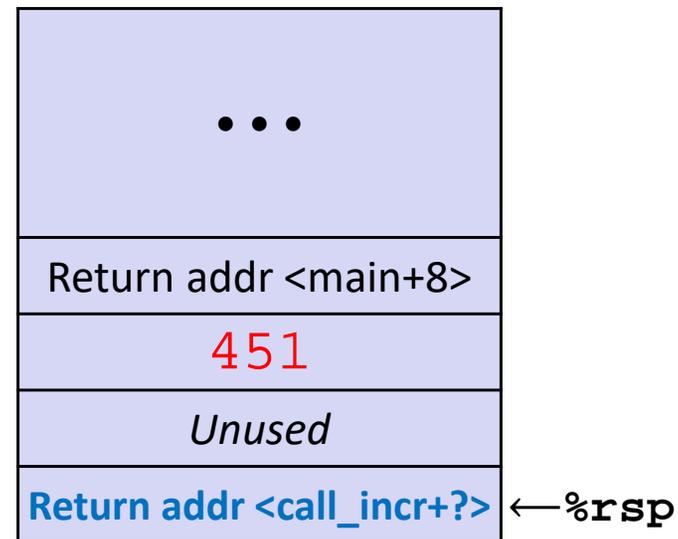
Procedure Call Example (step 4)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
    subq    $16, %rsp
    movq    $351, 8(%rsp)
    movl    $100, %esi
    leaq    8(%rsp), %rdi
    call   increment
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

```
increment:
    movq    (%rdi), %rax # x = *p
    addq    %rax, %rsi   # y = x+100
    movq    %rsi, (%rdi) # *p = y
    ret
```

Stack Structure



- ❖ State while inside increment
 - After code in body has been executed

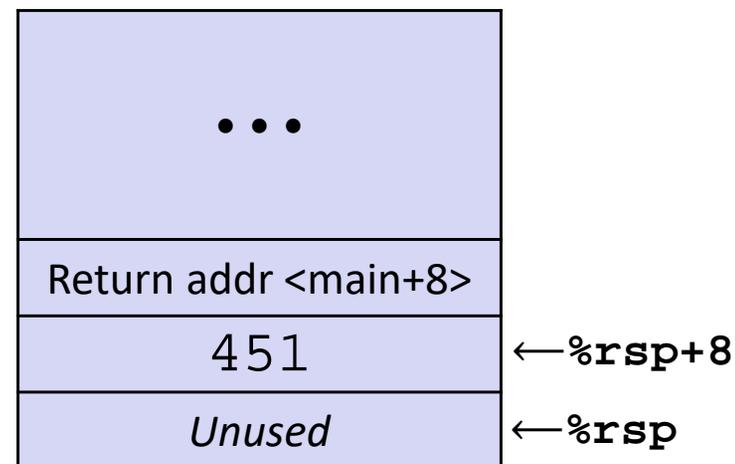
Register	Use(s)
%rdi	&v1
%rsi	451
%rax	351

Procedure Call Example (step 5)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
    subq    $16, %rsp
    movq    $351, 8(%rsp)
    movl    $100, %esi
    leaq   8(%rsp), %rdi
    call   increment
    addq   8(%rsp), %rax
    addq   $16, %rsp
    ret
```

Stack Structure



- ❖ After returning from call to increment
 - Registers and memory have been modified and return address has been popped off stack

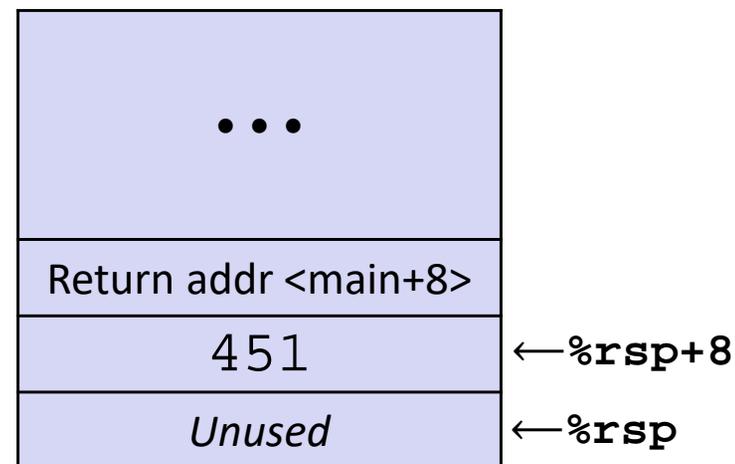
Register	Use(s)
%rdi	&v1
%rsi	451
%rax	351

Procedure Call Example (step 6)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
    subq    $16, %rsp
    movq    $351, 8(%rsp)
    movl    $100, %esi
    leaq    8(%rsp), %rdi
    call    increment
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

Stack Structure



← Update **%rax** to contain v1+v2

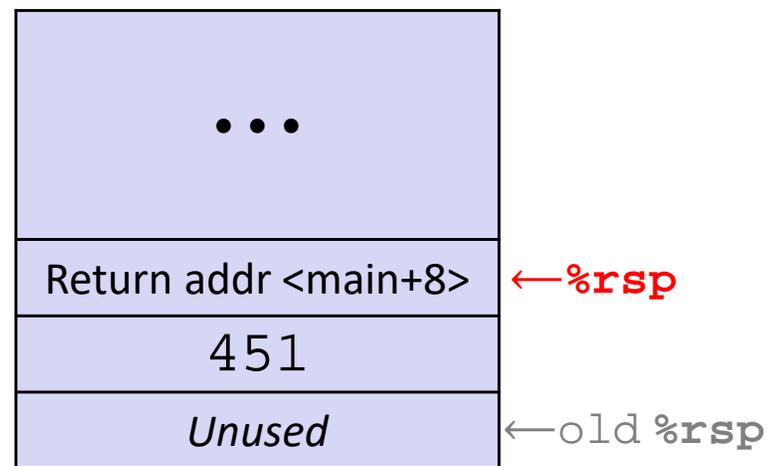
Register	Use(s)
%rdi	&v1
%rsi	451
%rax	451+351

Procedure Call Example (step 7)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
    subq    $16, %rsp
    movq    $351, 8(%rsp)
    movl    $100, %esi
    leaq    8(%rsp), %rdi
    call    increment
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

Stack Structure



← De-allocate space for local vars

Register	Use(s)
%rdi	&v1
%rsi	451
%rax	802

Procedure Call Example (step 8)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
    subq    $16, %rsp
    movq    $351, 8(%rsp)
    movl    $100, %esi
    leaq    8(%rsp), %rdi
    call    increment
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

Stack Structure



❖ State *just before* returning from call to call_incr

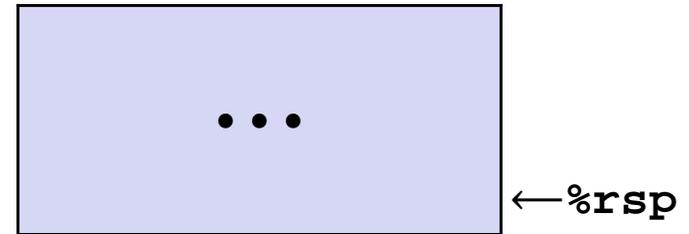
Register	Use(s)
%rdi	&v1
%rsi	451
%rax	802

Procedure Call Example (step 9)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
    subq    $16, %rsp
    movq    $351, 8(%rsp)
    movl    $100, %esi
    leaq    8(%rsp), %rdi
    call    increment
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

Final Stack Structure



- ❖ State immediately *after* returning from call to `call_incr`
 - Return addr has been popped off stack
 - Control has returned to the instruction immediately following the call to `call_incr` (not shown here)

Register	Use(s)
%rdi	&v1
%rsi	451
%rax	802

Lab 2 Demo

❖ Let's look at that binary bomb!

```
objdump -d bomb > bomb_disas //store disassembly of bomb in file  
// called bomb_disas
```

In GDB:

stepi <#>

// execute the next <#> asm instr (stepping into function calls)

nexti <#>

// execute the next <#> asm instr (stepping over function calls)

print /<format> <expr>

// print value of <expr> in <format>

x /<format> <addr>

// dereference <addr> and print in <format>

Notes:

Annoyingly, register names in <expr> and <addr> in GDB are preceded by '\$'

↳ so \$rsp instead of %rsp

Common format characters are

- 'b' for binary
- 'x' for hex
- 's' for string

Procedures

- ❖ Stack Structure
- ❖ Calling Conventions
 - Passing control
 - Passing data
 - Managing local data
- ❖ **Register Saving Conventions**
- ❖ Illustration of Recursion

Register Saving Conventions

- ❖ When procedure `yoo` calls `who`:
 - `yoo` is the *caller*
 - `who` is the *callee*
- ❖ Can registers be used for temporary storage?

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

```
who:  
  . . .  
  subq $18213, %rdx  
  . . .  
  ret
```

- No! Contents of register `%rdx` overwritten by `who`!
- This could be trouble – something should be done. Either:
 - *Caller* should save `%rdx` before the call (and restore it after the call)
 - *Callee* should save `%rdx` before using it (and restore it before returning)

Register Saving Conventions

❖ “*Caller-saved*” registers

- It is the **caller**'s responsibility to save any important data in these registers before calling another procedure (*i.e.* the **callee** can freely change data in these registers)
- **Caller** saves values in its stack frame before calling **Callee**, then restores values after the call

❖ “*Callee-saved*” registers

- It is the callee's responsibility to save any data in these registers before using the registers (*i.e.* the **caller** assumes the data will be the same across the **callee** procedure call)
- **Callee** saves values in its stack frame before using, then restores them before returning to **caller**

Silly Register Convention Analogy

- 1) Parents (*caller*) leave for the weekend and give the keys to the house to their child (*callee*)
 - Being suspicious, they put away/hid the valuables (*caller-saved*) before leaving
 - Warn child to leave the bedrooms untouched: “These rooms better look the same when we return!”
- 2) Child decides to throw a wild party (*computation*), spanning the entire house
 - To avoid being disowned, child moves all of the stuff from the bedrooms to the backyard shed (*callee-saved*) before the guests trash the house
 - Child cleans up house after the party and moves stuff back to bedrooms
- 3) Parents return home and are satisfied with the state of the house
 - Move valuables back and continue with their lives

x86-64 Linux Register Usage, part 1

❖ **%rax**

- Return value
- Also **caller**-saved & restored
- Can be modified by procedure

❖ **%rdi, ..., %r9**

- Arguments
- Also **caller**-saved & restored
- Can be modified by procedure

❖ **%r10, %r11**

- **Caller**-saved & restored
- Can be modified by procedure

Return value

%rax

Arguments

%rdi

%rsi

%rdx

%rcx

%r8

%r9

Caller-saved
temporaries

%r10

%r11

x86-64 Linux Register Usage, part 2

❖ `%rbx, %r12, %r13, %r14`

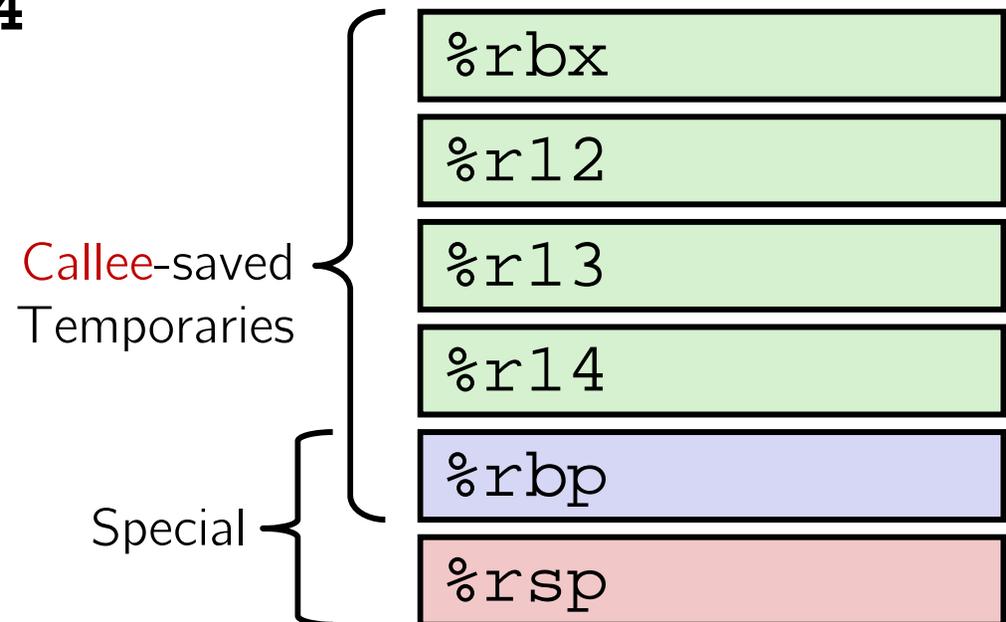
- **Callee**-saved
- **Callee** must save & restore

❖ `%rbp`

- **Callee**-saved
- **Callee** must save & restore
- May be used as frame pointer
- Can mix & match

❖ `%rsp`

- Special form of **callee** save
- Restored to original value upon exit from procedure



x86-64 64-bit Registers: Usage Conventions

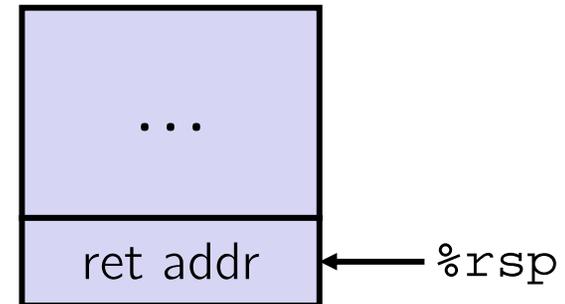
%rax	Return value - Caller saved	%r8	Argument #5 - Caller saved
%rbx	Callee saved	%r9	Argument #6 - Caller saved
%rcx	Argument #4 - Caller saved	%r10	Caller saved
%rdx	Argument #3 - Caller saved	%r11	Caller Saved
%rsi	Argument #2 - Caller saved	%r12	Callee saved
%rdi	Argument #1 - Caller saved	%r13	Callee saved
%rsp	Stack pointer	%r14	Callee saved
%rbp	Callee saved	%r15	Callee saved

Callee-Saved Example (step 1)

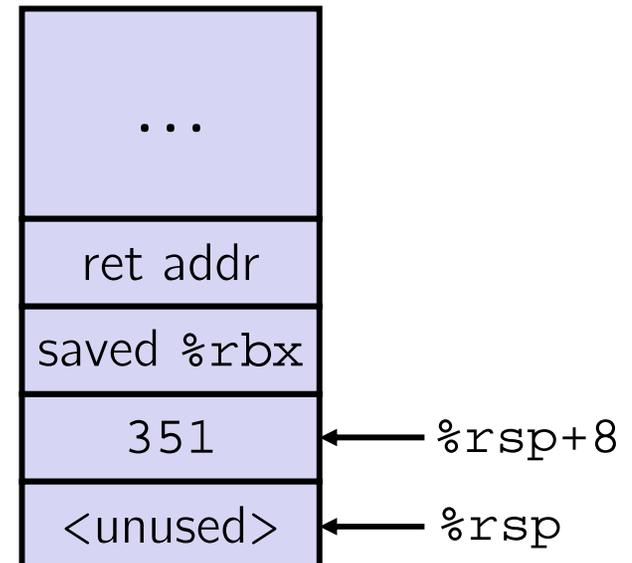
```
long call_incr2(long x) {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return x+v2;
}
```

```
call_incr2:
    pushq    %rbx
    subq    $16, %rsp
    movq    %rdi, %rbx
    movq    $351, 8(%rsp)
    movl    $100, %esi
    leaq    8(%rsp), %rdi
    call    increment
    addq    %rbx, %rax
    addq    $16, %rsp
    popq    %rbx
    ret
```

Initial Stack Structure



Resulting Stack Structure

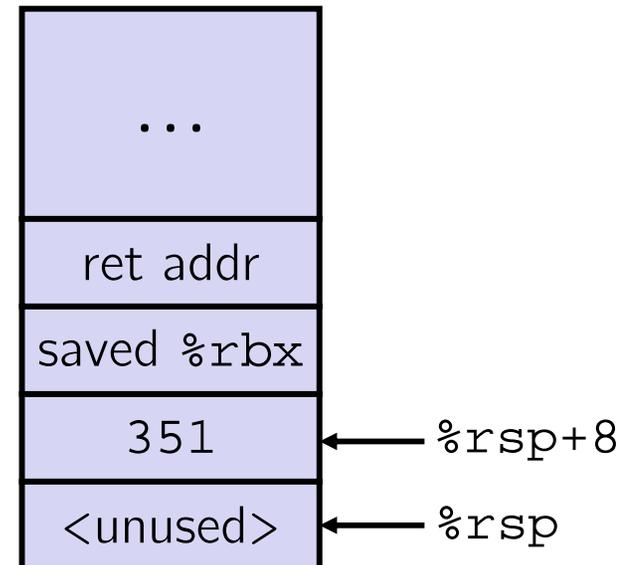


Callee-Saved Example (step 2)

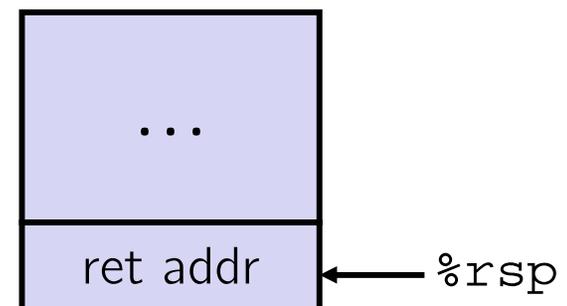
```
long call_incr2(long x) {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return x+v2;
}
```

```
call_incr2:
    pushq    %rbx
    subq    $16, %rsp
    movq    %rdi, %rbx
    movq    $351, 8(%rsp)
    movl    $100, %esi
    leaq    8(%rsp), %rdi
    call    increment
    addq    %rbx, %rax
    addq    $16, %rsp
    popq    %rbx
    ret
```

Stack Structure



Pre-return Stack Structure



Why Caller *and* Callee Saved?

- ❖ We want *one* calling convention to simply separate implementation details between caller and callee
- ❖ In general, neither caller-save nor callee-save is “best”:
 - If caller isn’t using a register, caller-save is better
 - If callee doesn’t need a register, callee-save is better
 - If “do need to save”, callee-save generally makes smaller programs
 - Functions are called from multiple places
- ❖ So... “some of each” and compiler tries to “pick registers” that minimize amount of saving/restoring

Register Conventions Summary

- ❖ **Caller**-saved register values need to be pushed onto the stack before making a procedure call *only if the Caller needs that value later*
 - **Callee** may change those register values
- ❖ **Callee**-saved register values need to be pushed onto the stack *only if the Callee intends to use those registers*
 - **Caller** expects unchanged values in those registers
- ❖ Don't forget to restore/pop the values later!