(Non-Tail) Function Calls

October 4, 2023

Previously on EECS 483...

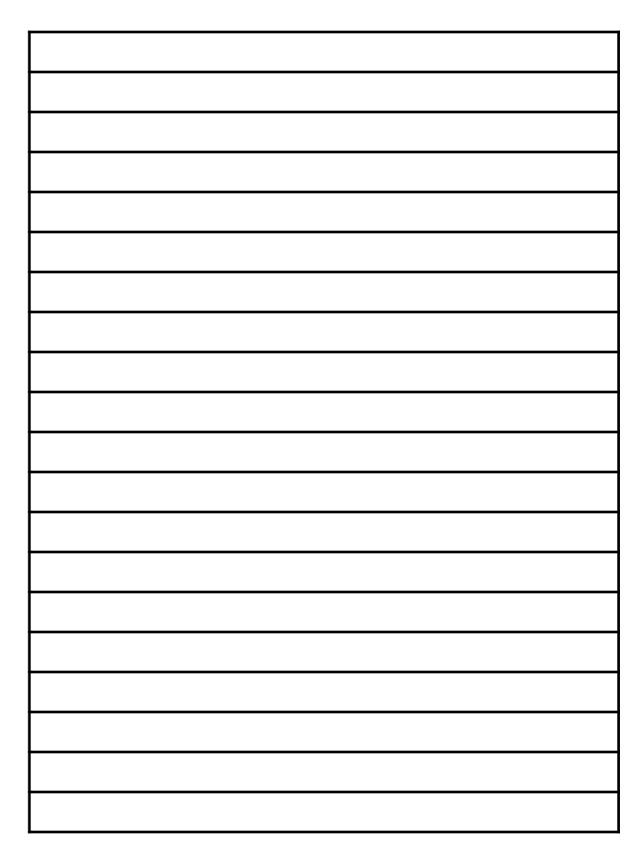
- Calling into Rust: System V AMD64 calling convention
- Tail Calls: compile directly to jumps

```
def f(y):
  let z = 2 in
 y * z
end
let a = 7 in
let b = 13 in
let x = f(a) in
x + b
```

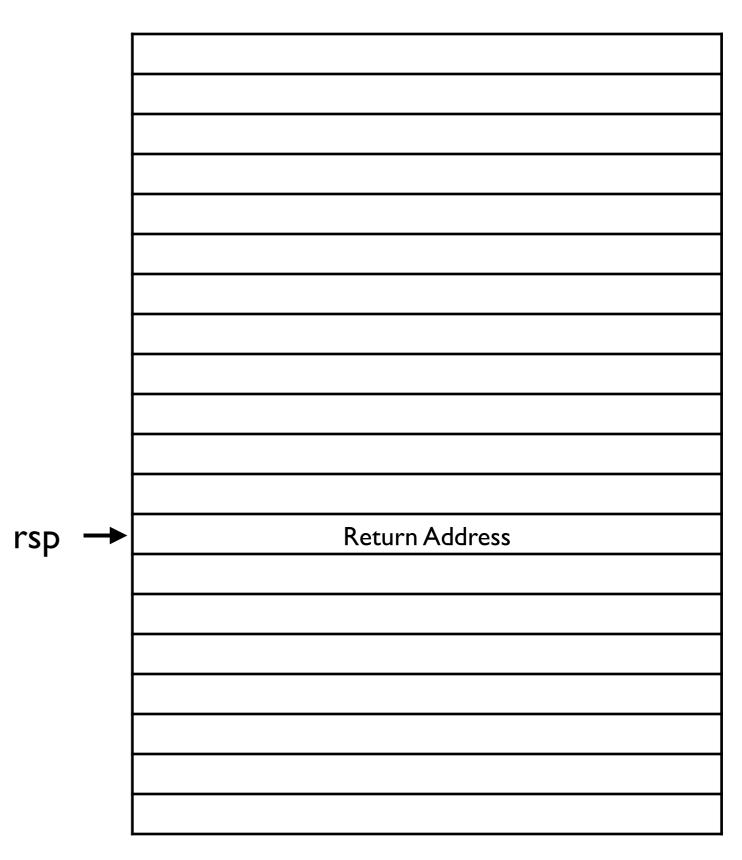
```
def f(y):
   let z = 2 in
   y * z
 end
\rightarrowlet a = 7 in
 let b = 13 in
 let x = f(a) in
 x + b
```

```
def f(y):
    let z = 2 in
    y * z
    end
→let a = 7 in
    let b = 13 in
```

let
$$x = f(a)$$
 in $x + b$

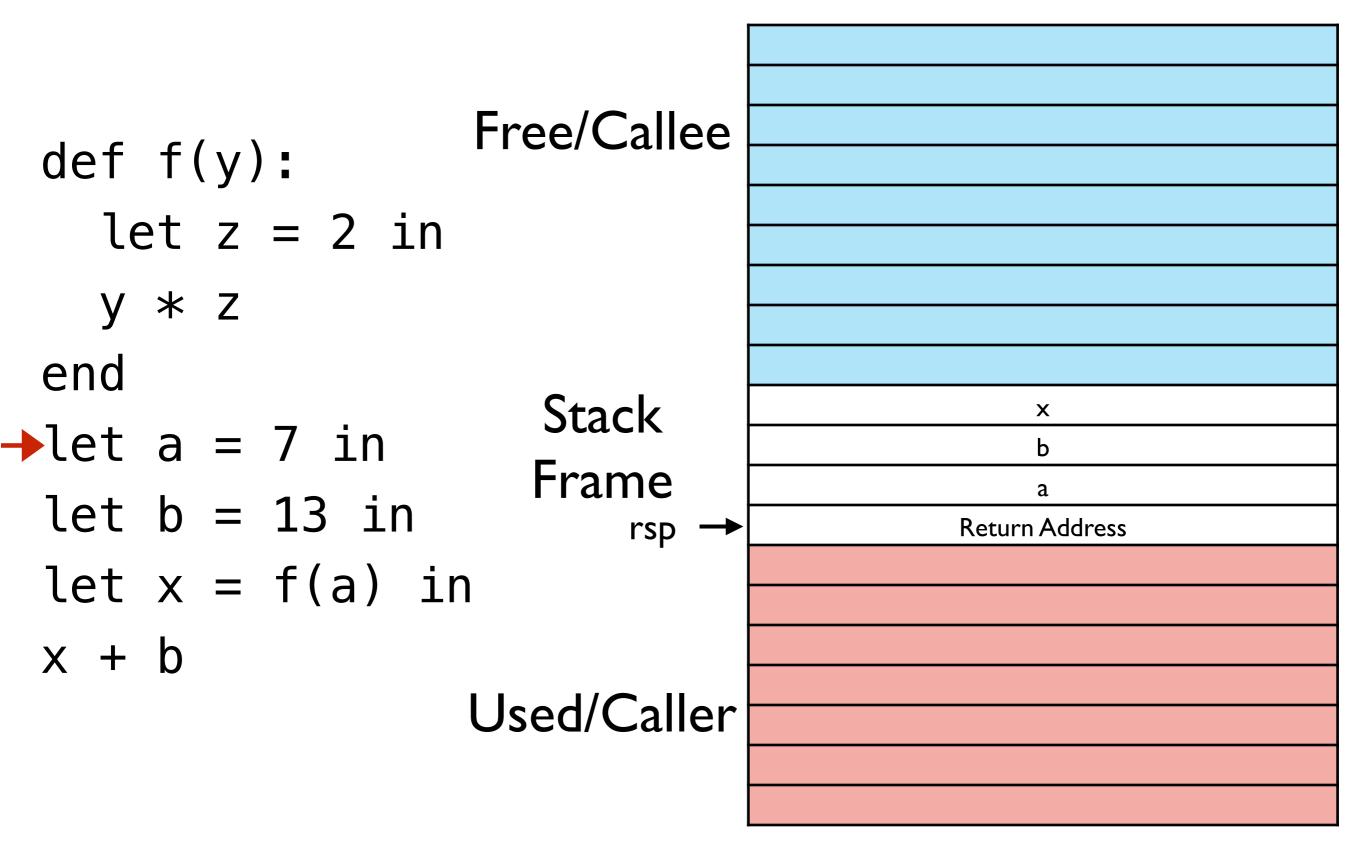


```
def f(y):
   let z = 2 in
   y * Z
 end
\rightarrowlet a = 7 in
 let b = 13 in
 let x = f(a) in
 x + b
```

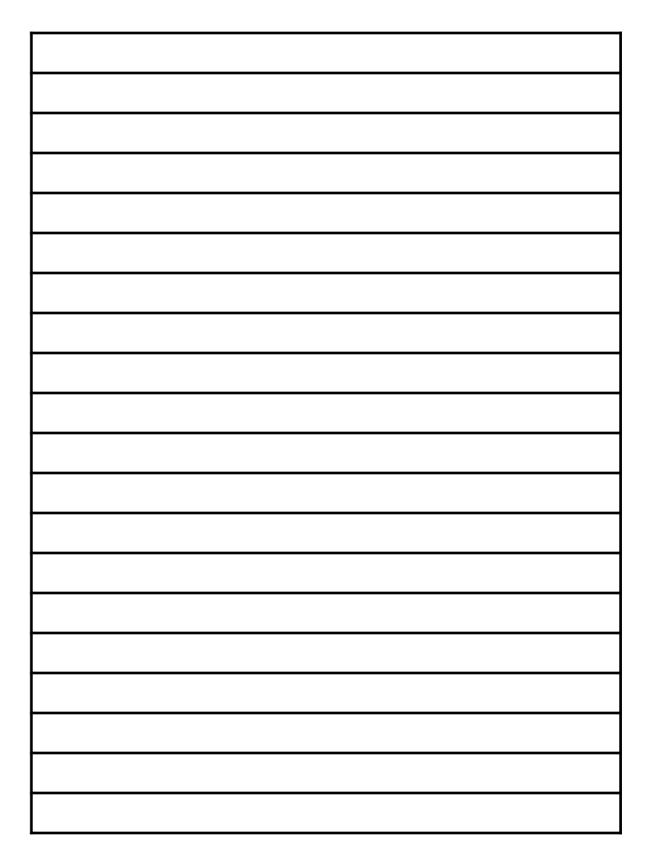


```
def f(y):
    let z = 2 in
    y * Z
 end
\rightarrowlet a = 7 in
 let b = 13 in
                                          Return Address
                            rsp →
 let x = f(a) in
 x + b
                    Used/Caller
```

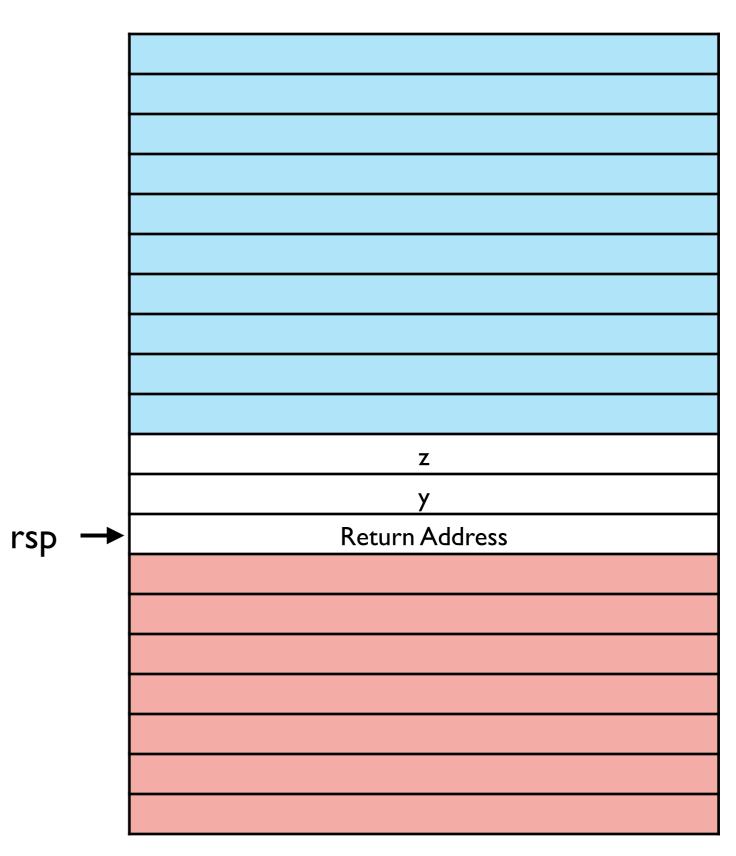
```
Free/Callee
 def f(y):
    let z = 2 in
    y * Z
 end
\rightarrowlet a = 7 in
 let b = 13 in
                                           Return Address
                            rsp →
 let x = f(a) in
 x + b
                    Used/Caller
```



```
def f(y):
\rightarrow let z = 2 in
   y * z
 end
 let a = 7 in
 let b = 13 in
 let x = f(a) in
 x + b
```



```
def f(y):
\rightarrow let z = 2 in
   y * Z
 end
 let a = 7 in
 let b = 13 in
 let x = f(a) in
 x + b
```



ı		
1		
ł		
İ		
l		
H		
ı		
	X	
	b	Z
	a	y
-	Return Address	Return Address
	Return Address	Return Address
1		
İ		
ł		
İ		
l		

	X		
	b		Z
	a		V
			/
\rightarrow	Return Address 0	\rightarrow	Return Address I

Address of a local is *relative* to the base of the stack frame



```
def f(y):
   let z = 2 in
   y * Z
 end
\rightarrow let a = 7 in
 let b = 13 in
 let x = f(a) in
 x + b
```



```
def f(y):
   let z = 2 in
   y * z
 end
 let a = 7 in
 let b = 13 in
\rightarrow let x = f(a) in
 x + b
```



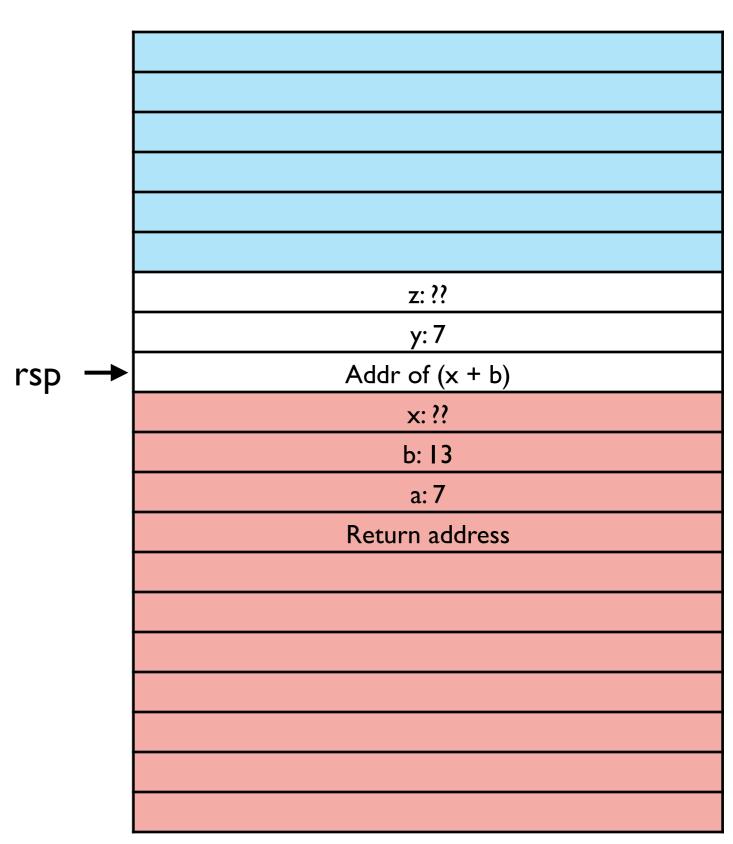
```
def f(y):
   let z = 2 in
   y * z
 end
 let a = 7 in
 let b = 13 in
\rightarrow let x = f(a) in
 x + b
```



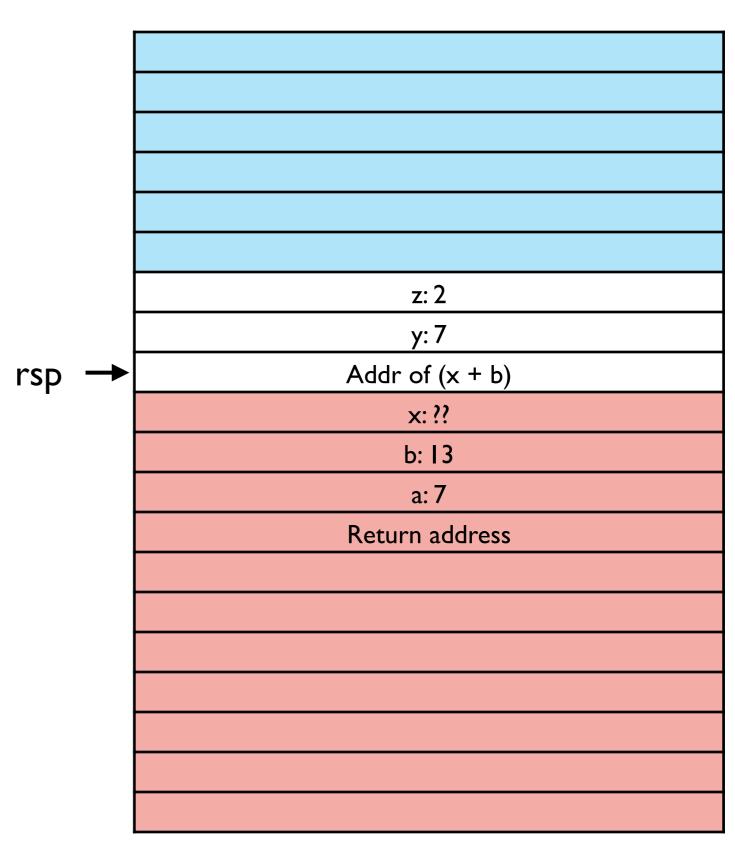
```
def f(y):
   let z = 2 in
   y * Z
 end
 let a = 7 in
 let b = 13 in
\rightarrow let x = f(a) in
 x + b
```



```
def f(y):
\rightarrow let z = 2 in
   y * Z
 end
 let a = 7 in
 let b = 13 in
 let x = f(a) in
 x + b
```



```
def f(y):
   let z = 2 in
→ y * z
 end
 let a = 7 in
 let b = 13 in
 let x = f(a) in
 x + b
```



```
def f(y):
   let z = 2 in
   y * Z
 end
 let a = 7 in
 let b = 13 in
let x = f(a) in
```



Snake Calling Convention (v0)

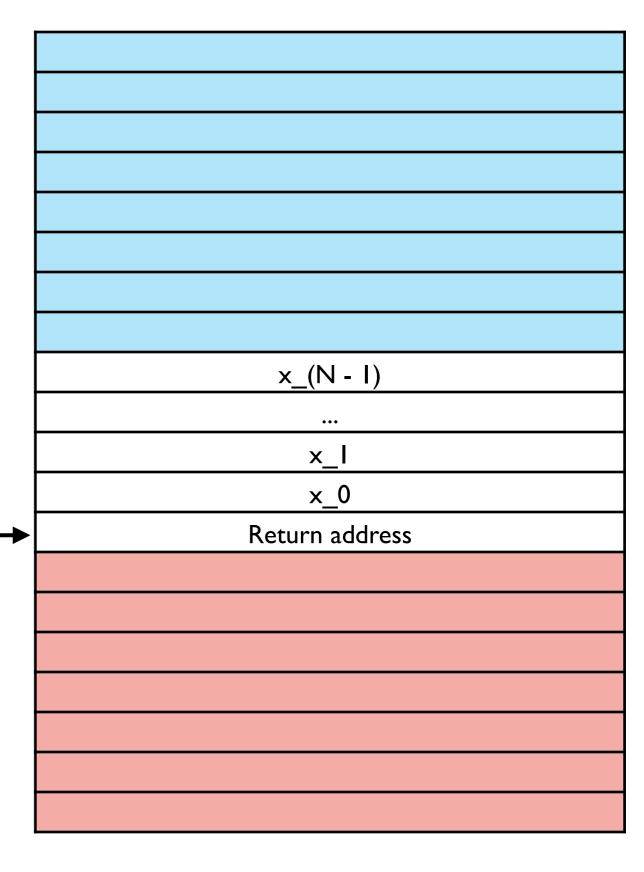
Snake Calling Convention (v0)

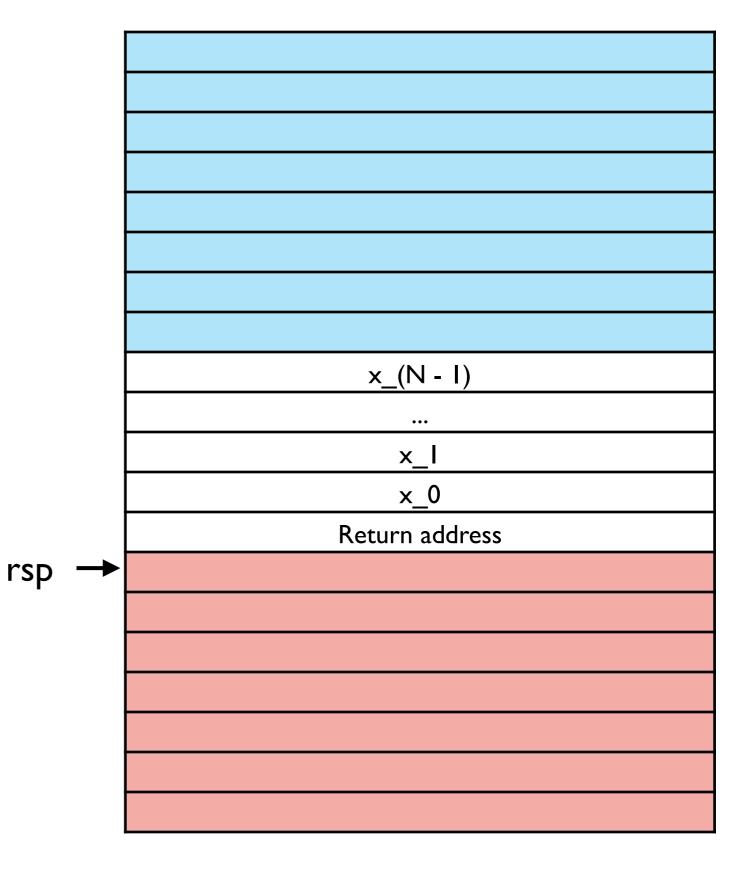
- Where are the arguments stored (relative to rsp)?
- Where is the return address stored (rel to rsp)?
- Where should we put the return value?
- Where should rsp point when we return?
- We don't use registers yet so don't worry about them

- Rsp points at the return address
- Args go at lower addresses, with first arg closest to rsp
- Caller's stack frames are at higher addresses

rsp

- Return value in RAX
- Upon return, rsp points to top of caller's stack frame





Benefits of Snake CC

- Easy to implement tail calls
- Arguments are treated uniformly with locals
- Return overlaps with SysV CC when there are no stack args

WARNING

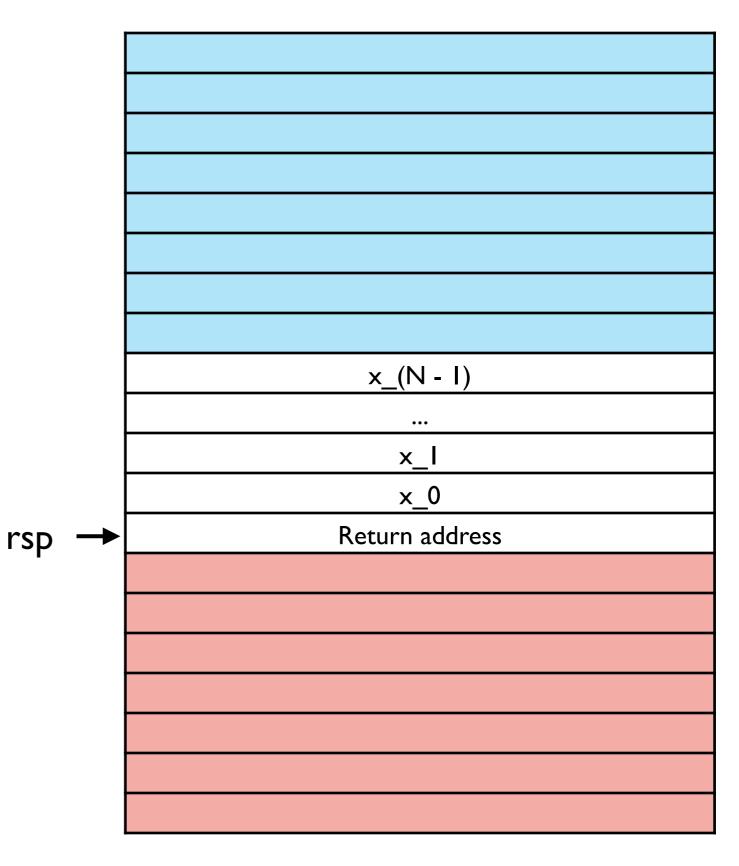
WARNING

RBP

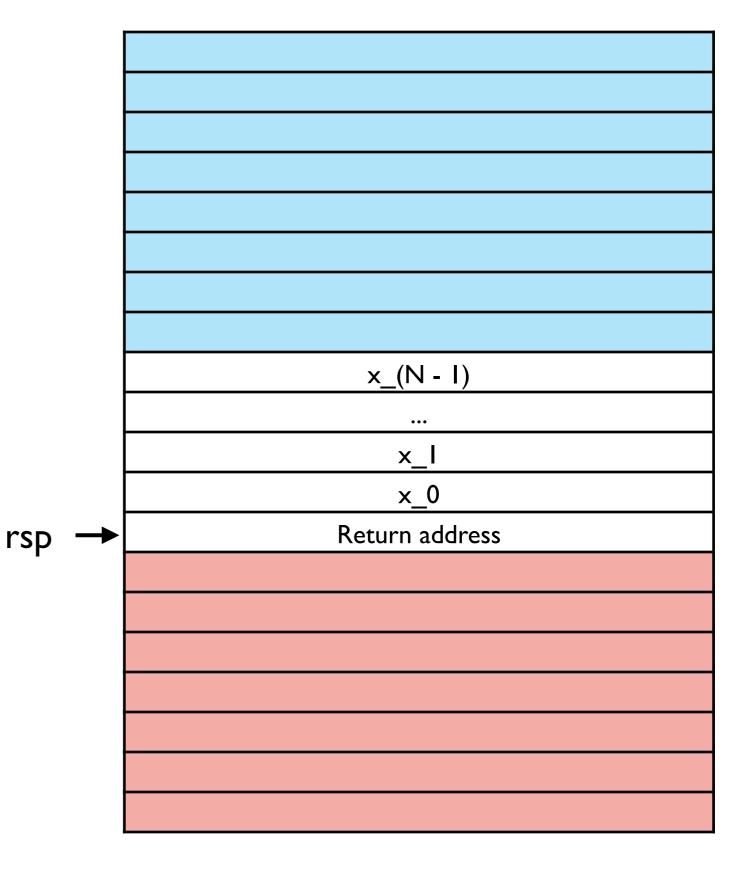
WARNING

- RBP
- Different from SysV ABI:
 - No register args (for now)
 - Placement of Args
 - "Caller-cleanup" vs "Callee-cleanup"

SYSVABI

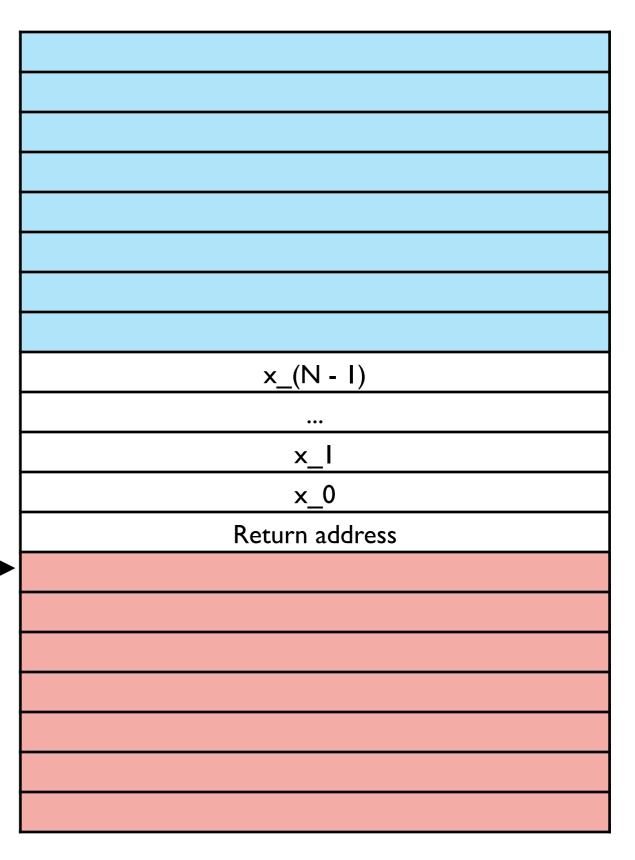


Snake after call

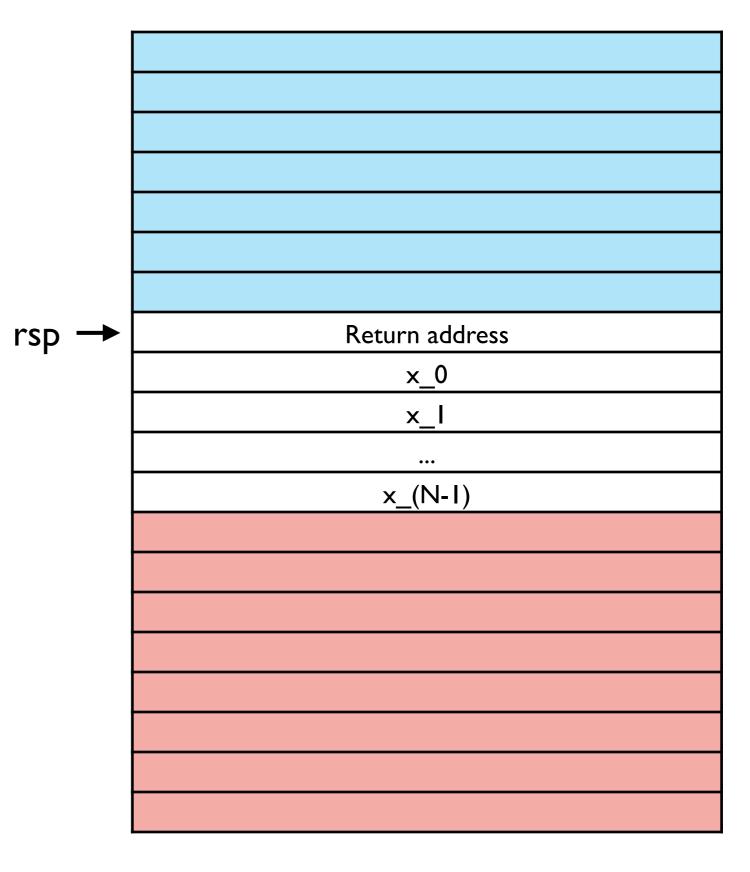


Snake after return

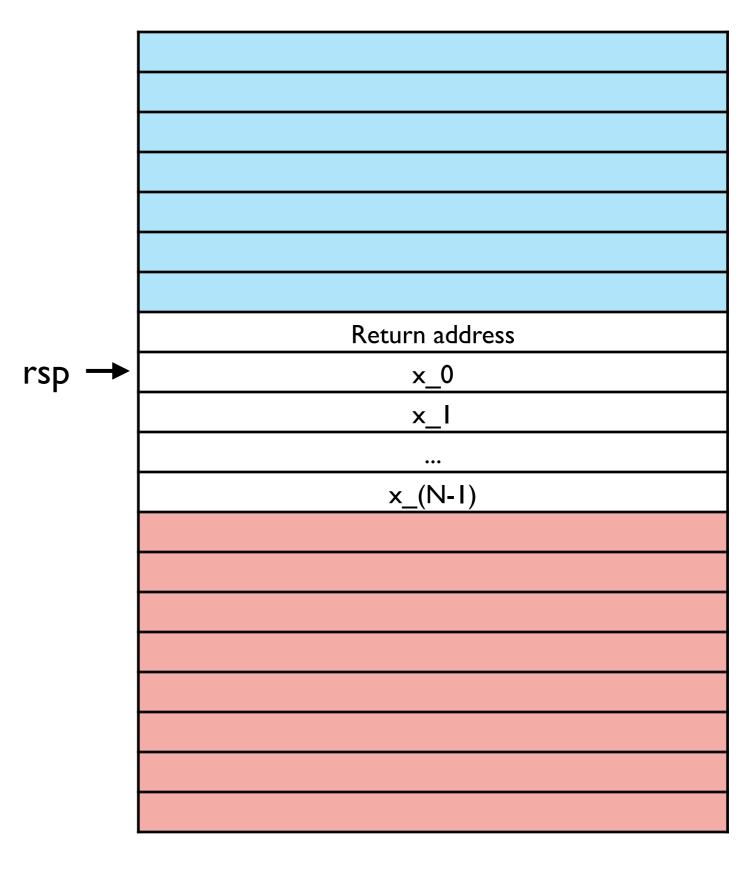
rsp



SysV after call



SysV after return



Implementing the Snake CC

- How to return?
- How to call?
- How to tail call?

To return x

	•••
	local 2
	local I
	local 0
	x_(N - I)
	•••
	x_I
	x_I x_0
rsp →	Return address
•	
l	

To return x

Stack

mov rax, [loc of x]
pop rbx
jmp rbx

rsp

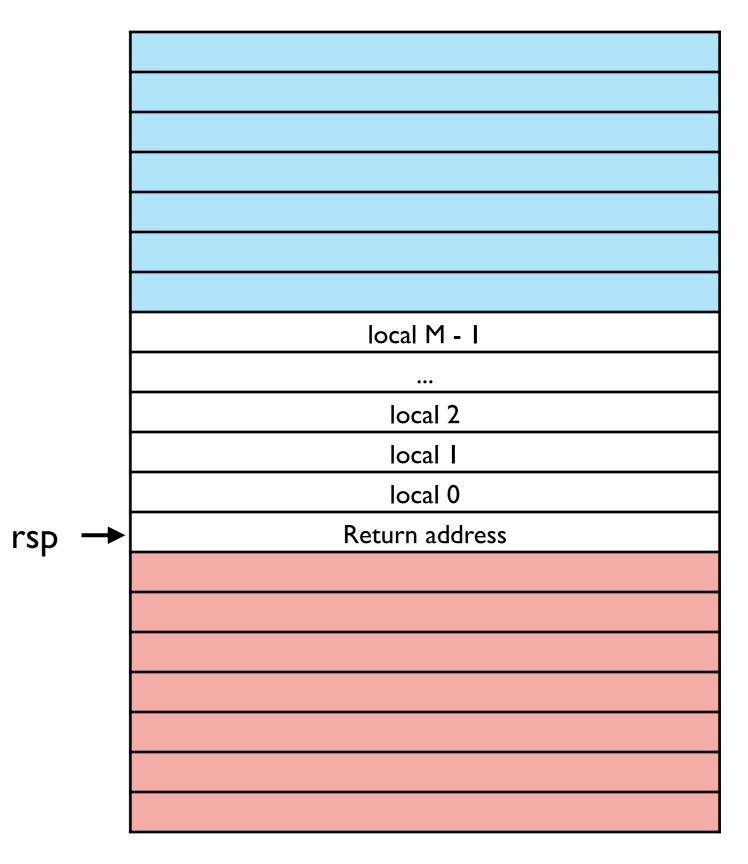
•••
local 2
local I
local 0
x_(N - I)
x_I
x_0
Return address

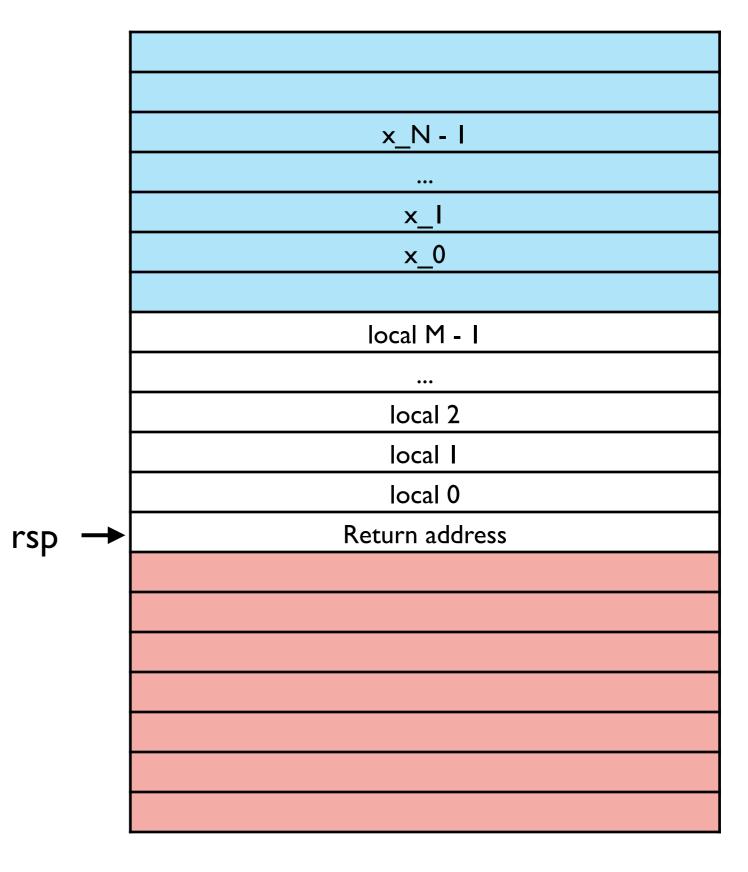
To return x

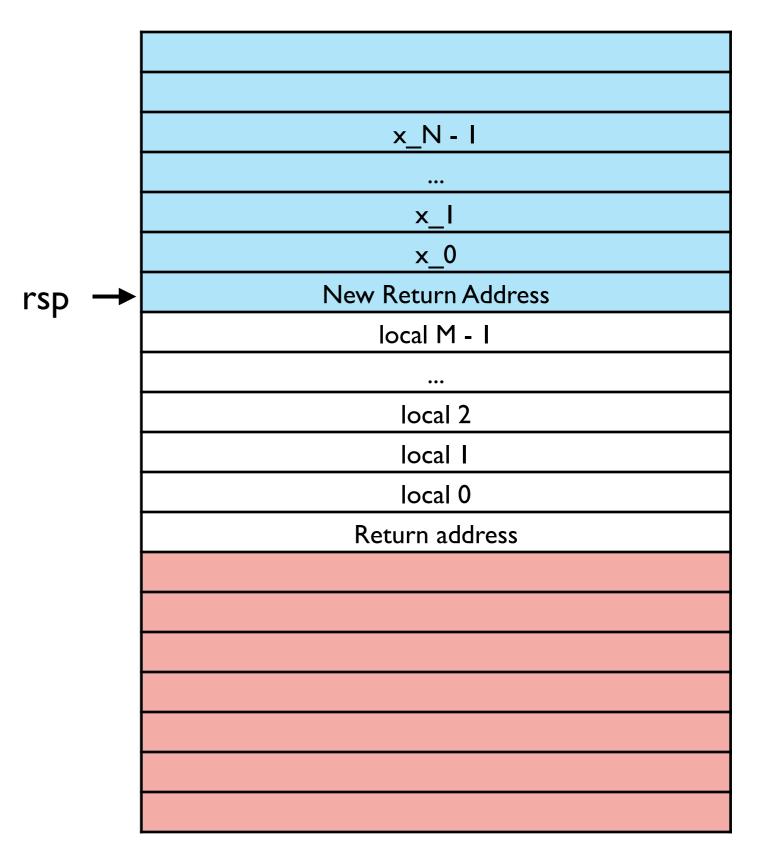
Stack

mov rax, [loc of x]
ret

	•••
	local 2
	local I
	local 0
	x_(N - I)
	•••
	x_I
	x_0
	Return address
rsp →	



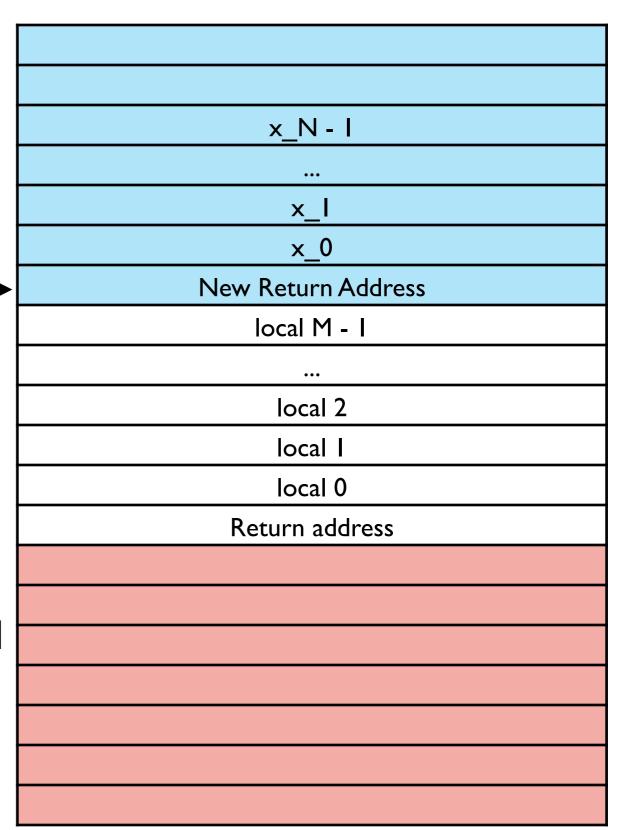




mov $[rsp - 8 * (M + 2)], [x_0]$

call

Stack



rsp

x_N - I
•••
x_I
x_0
New Return Address
local M - I
•••
local 2
local I
local 0
Return address

	x_N - I
	•••
	x_I
	x_0
	New Return Address
rsp →	local M - I
	•••
	local 2
	local I
	local 0
	Return address
•	

	x_N - I
	•••
	x_I
	x_0
	New Return Address
	local M - I
	•••
	local 2
	local I
	local 0
rsp →	Return address
•	
•	

Stack

_

```
mov [rsp - 8 * (M + 2)], [x_0]

mov [rsp - 8 * (M + 3)], [x_1] rsp \rightarrow

mov [rsp - 8 * (M + 2 + N)], [x_(N-1)]

sub rsp, 8 * M

call
```

add rsp, 8 * M

Stack

_

```
mov [rsp - 8 * (M + 2)], [x_0]

mov [rsp - 8 * (M + 3)], [x_1] rsp \rightarrow

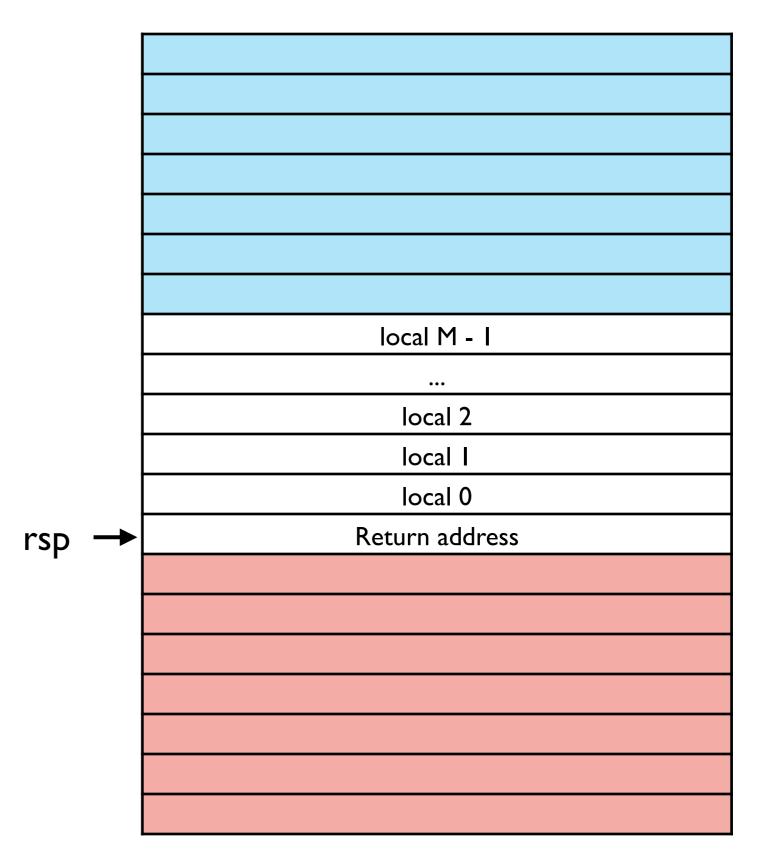
mov [rsp - 8 * (M + 2 + N)], [x_(N-1)]

sub rsp, 8 * M

call
```

add rsp, 8 * M

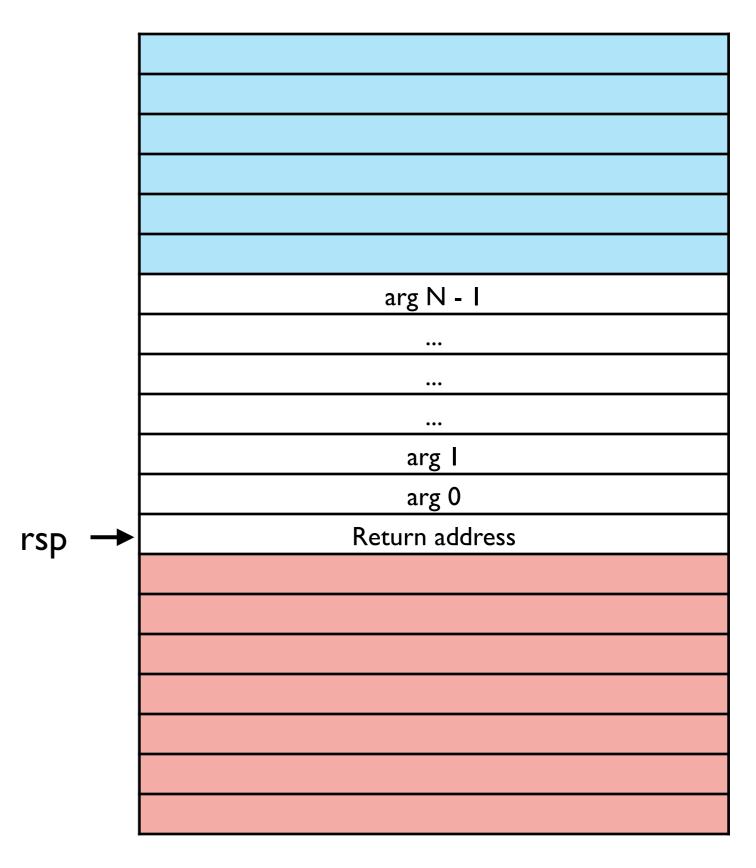




if N < M

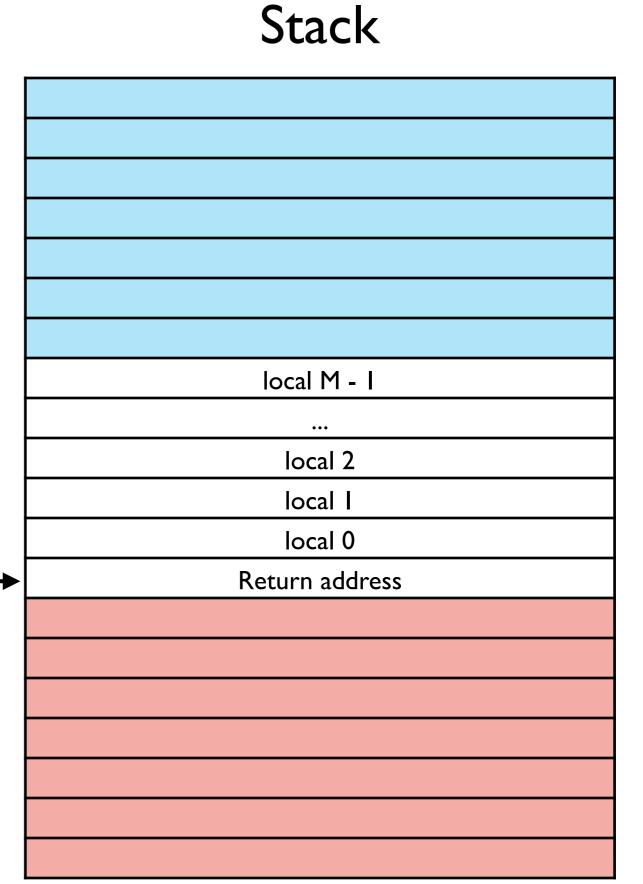


if N > M



Careful not to overwrite locals we are using!

rsp



	arg N - I
	•••
	arg 2
	arg I
	arg 0
	local M - I
	•••
	local 2
	local I
	local 0
rsp →	Return address
•	

	arg N - I
	•••
	arg 2
	arg I
	arg 0
	local M - I
	•••
	local 2
	local I
	local 0
rsp →	Return address
•	

	_	
		arg N - I
		•••
		arg 2
		arg I
		arg 0
		local M - I
		•••
		local 2
		local I
		arg 0
^sp -	→	Return address
•		

	arg N - I
	•••
	arg 2
	arg I
	arg 0
	local M - I
	•••
	local 2
	arg I
	arg 0
rsp →	Return address
•	

	arg N - I
	•••
	arg 2
	arg I
	arg 0
	local M - I
	•••
	arg 2
	arg I
	arg 0
rsp →	Return address
•	

	arg N - I
	•••
	arg N - I
	•••
	•••
	•••
	arg 2
	arg I
	arg 0
rsp →	Return address
· - F	

To TAIL call f with

N arguments

$$f(x_0, ... x_{N-1})$$

	arg N - I
	•••
	arg N - I
	•••
	•••
	•••
	•••
	arg 2
	arg I
	arg 0
rsp →	Return address
•	

To TAIL call f with

N arguments

$$f(x_0, ... x_{N-1})$$

let
$$y_0 = x_0, ...$$

in
 $f(y_0, ..., y_(N-1))$

arg N - I
•••
arg N - I
arg 2
arg l
arg 0
Return address

N arguments

$$f(x_0, ... x_{N-1})$$

let
$$y_0 = x_0, ...$$

in
 $f(y_0, ..., y_(N-1))$

generate these "unnecessary" temporaries in sequentialize

	arg N - I
	••
	arg N - I
	•••
	•••
	•••
	•••
	arg 2
	arg I
	arg 0
>	Return address

Alignment(!)

- We want to be able to call into Rust with the Sys V CC at any time
- But *that* calling convention has an alignment restriction.
- So to make it easy to implement that alignment, we should require a similar alignment so that we don't have to check alignment *dynamically*.
 - tradeoff: we use potentially? more space to avoid branches at runtime (v slow)

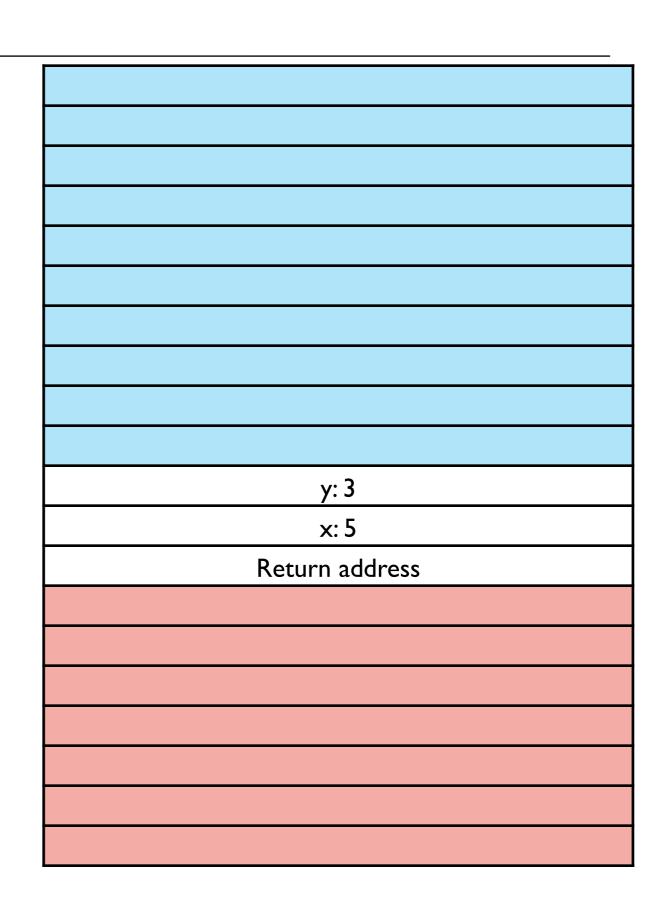
Alignment(!)

- But Sys V has the following alignment requirement:
 - Upon entry into a function, rsp + 8 % 16 == 0

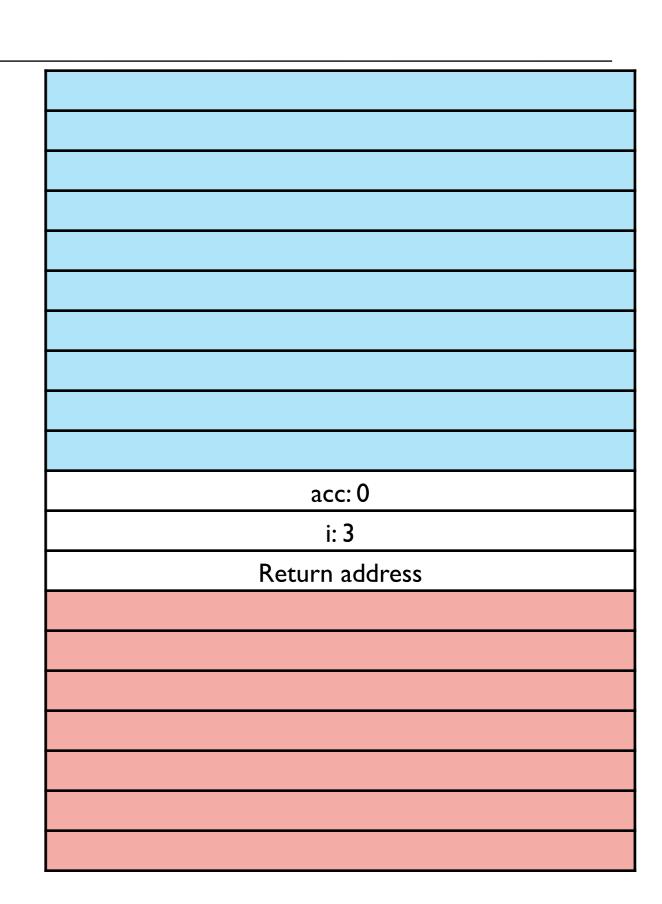
- To make this statically determined, we add a similar alignment requirement to Snake:
 - Upon entry into a function, rsp % 16 == 0
 - This way if we have no saved locals, we can just call
 - Sometimes need to include a dummy local

```
def multiply(x, y):
  def loop(i, acc):
    if i == 0:
      acc
    else:
      loop(i - 1,
            acc + x)
  end
  loop(y, 0)
end
```

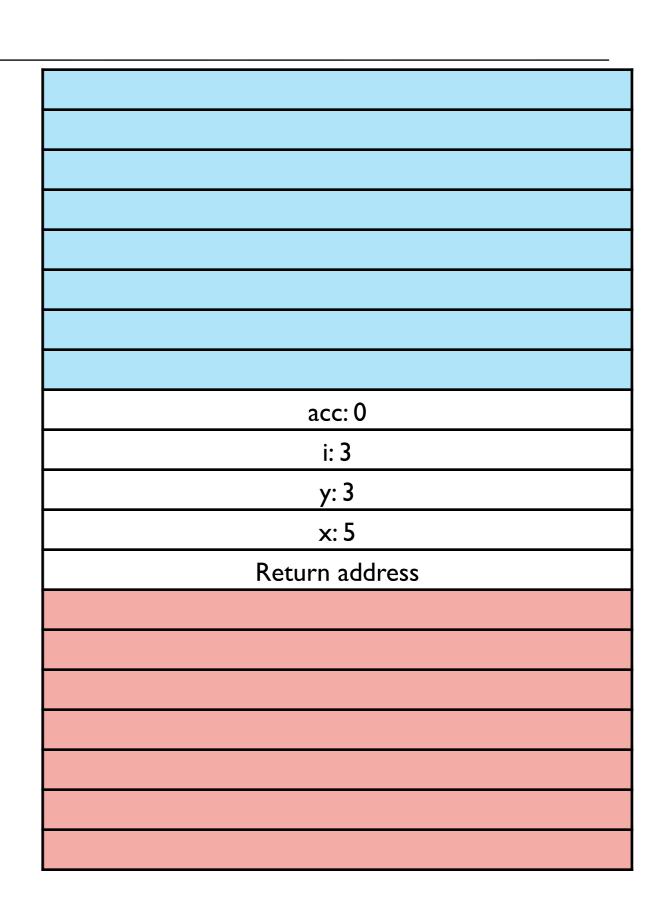
```
def multiply(x, y):
  def loop(i, acc):
     if i == 0:
       acc
    else:
       loop(i - 1,
             acc + x)
  end
\rightarrowloop(y, 0)
end
```



```
def multiply(x, y):
  def loop(i, acc):
   \rightarrow if i == 0:
       acc
    else:
       loop(i - 1,
             acc + x)
  end
  loop(y, 0)
end
```



```
def multiply(x, y):
  def loop(i, acc):
  →if i == 0:
      acc
    else:
      loop(i - 1,
            acc + x)
  end
  loop(y, 0)
end
```



A Problem?

```
def multiply(x, y):
  def loop(y):
    if y == 0:
      0
    else:
      x + loop(y - 1)
  end
end
multiply(5, 3)
```

```
def multiply(x, y):
  def loop(y):
     if y == 0:
     else:
                                             y: 3
       x + loop(y - 1)
                                          Return address 0
                           rsp
  end
\rightarrow loop(y) + 0
end
multiply(5, 3)
```

```
def multiply(x, y):
  def loop(y):
  \rightarrow if y == 0:
                                                 y: 3
                                              Return address I
     else:
                              rsp
                                                 y: 3
        x + loop(y - 1)
                                                 x: 5
                                              Return address 0
  end
  loop(y) + 0
end
multiply(5, 3)
```

```
def multiply(x, y):
   def loop(y):
   \rightarrow if y == 0:
                                                    y: 2
                                                Return address 2
                               rsp
                                                    y: 3
                                                Return address I
      else:
                                                    y: 3
         x + loop(y - 1)
                                                    x: 5
                                                Return address 0
   end
   loop(y) + 0
end
multiply(5, 3)
```

```
def multiply(x, y):
  def loop(y):
  \rightarrow if y == 0:
    else:
      x + loop(y - 1)
  end
  loop(y) + 0
end
multiply(5, 3)
```

y: I
Return address 2
y: 2
Return address 2
y: 3
Return address I
y: 3
x: 5
Return address 0

<pre>def multiply(x, y):</pre>	y: 0
rsp 	Return address 2
<pre>def loop(y):</pre>	y: I
	Return address 2
<pre>→ if y == 0:</pre>	y: 2
0	Return address 2
	y: 3
else:	Return address I
	y: 3
x + loop(y - 1)	x: 5
	Return address 0
end	
$1000(y) \pm 0$	
loop(y) + 0	
end	
multiply(5, 3)	

"Lambda Lifting"

```
def multiply(x, y):
                           def multiply(x, y):
                            loop(x, y) + 0
  def loop(y):
    if y == 0:
                           and
                           def loop(x, y):
                             if y == 0:
    else:
      x + loop(y - 1)
                               0
                             else:
  end
  loop(y) + 0
                               x + loop(x, y - 1)
                           end
end
                           multiply(5, 3)
multiply(5, 3)
```

Summary

- When calling Rust code, use the Sys V
- When calling Snake functions, use the Snake calling convention
 - Tail Call: overwrite our stack frame with arguments
 - Non Tail Call: push return address/args above our stack frame
- Returning:
 - works the same way for both: same "returning convention"
- Alignment
- Lambda Lifting