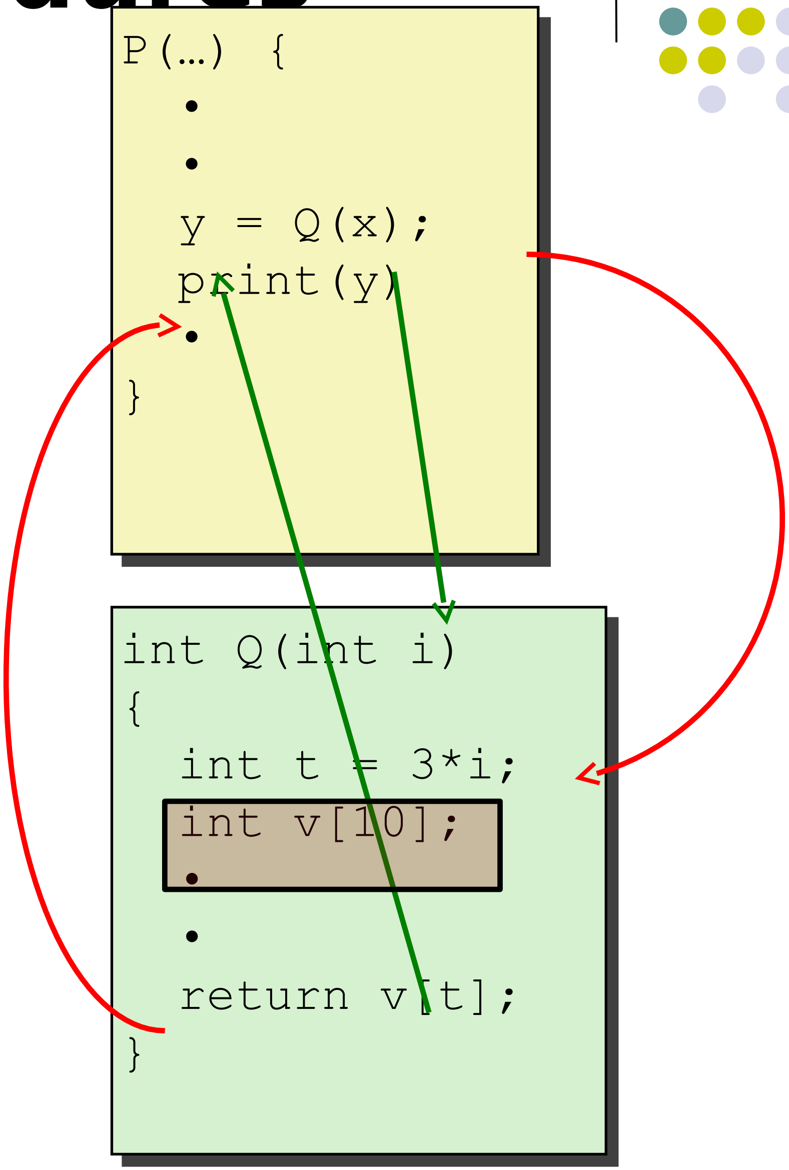
Computer Organization: A Programmer's Perspective

Machine-Level Programming (3: 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
- **Mechanisms all implemented with machine instructions**

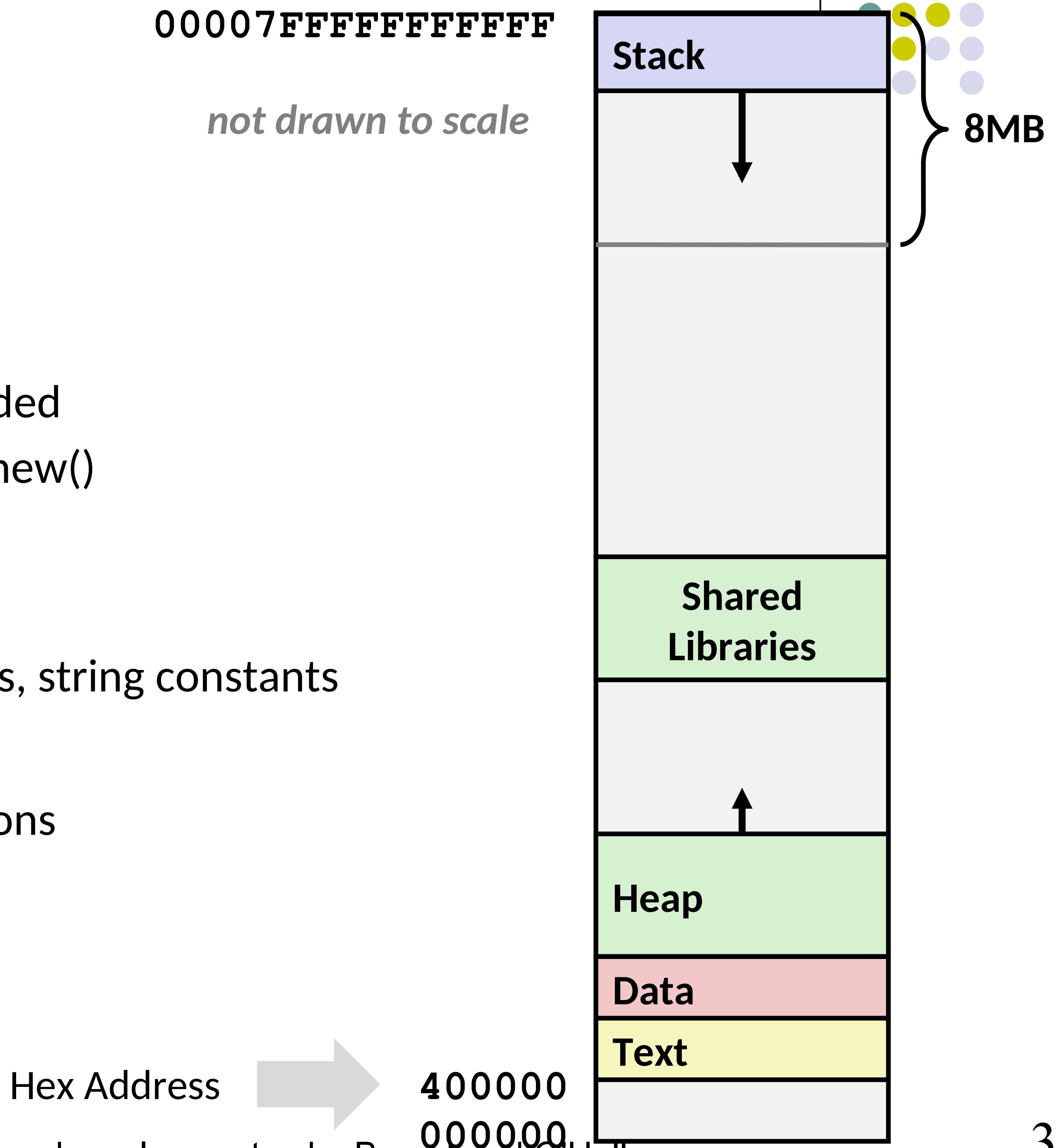


```
P(…) {
  •
  •
  y = Q(x);
  print(y)
  •
}
```

```
int Q(int i)
{
  int t = 3*i;
  int v[10];
  •
  •
  return v[t];
}
```

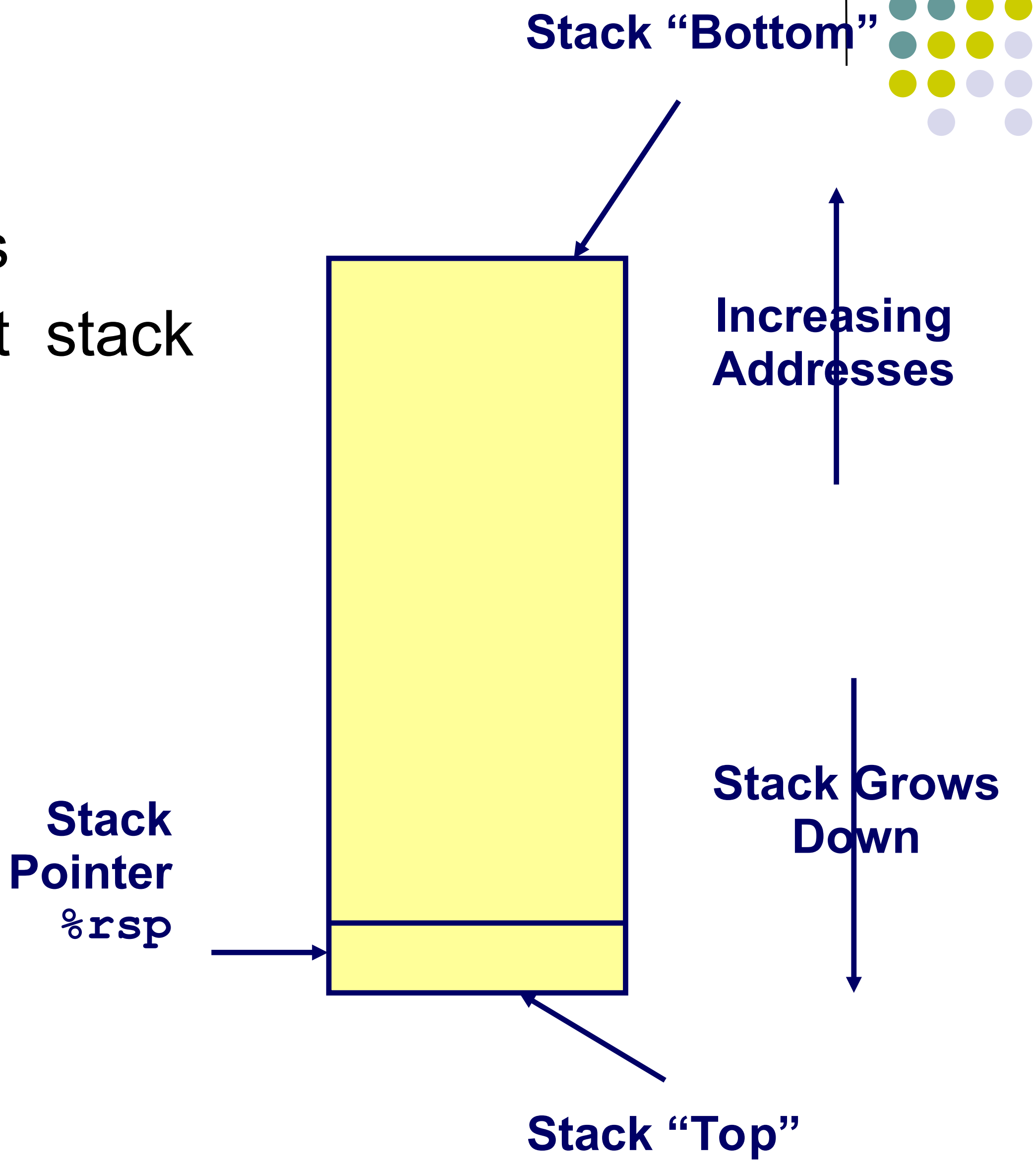
x86-64 Linux Memory Layout

- **Stack**
  - Runtime stack (8MB limit)
  - E. g., local variables
- **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



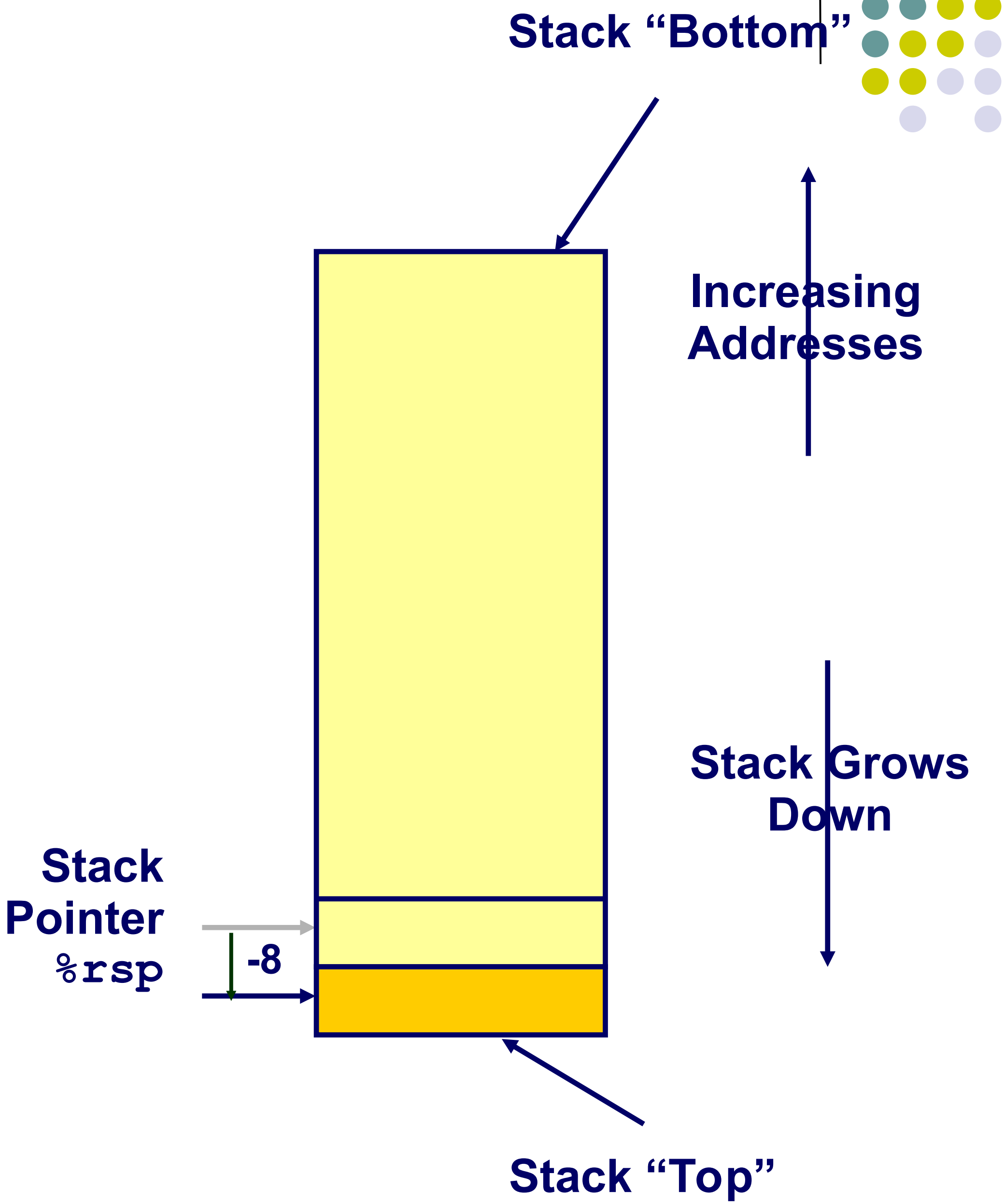
Stack

- Region of memory
- Managed with stack discipline
- Grows toward lower addresses
- Register `%rsp` indicates lowest stack address
  - address of top element



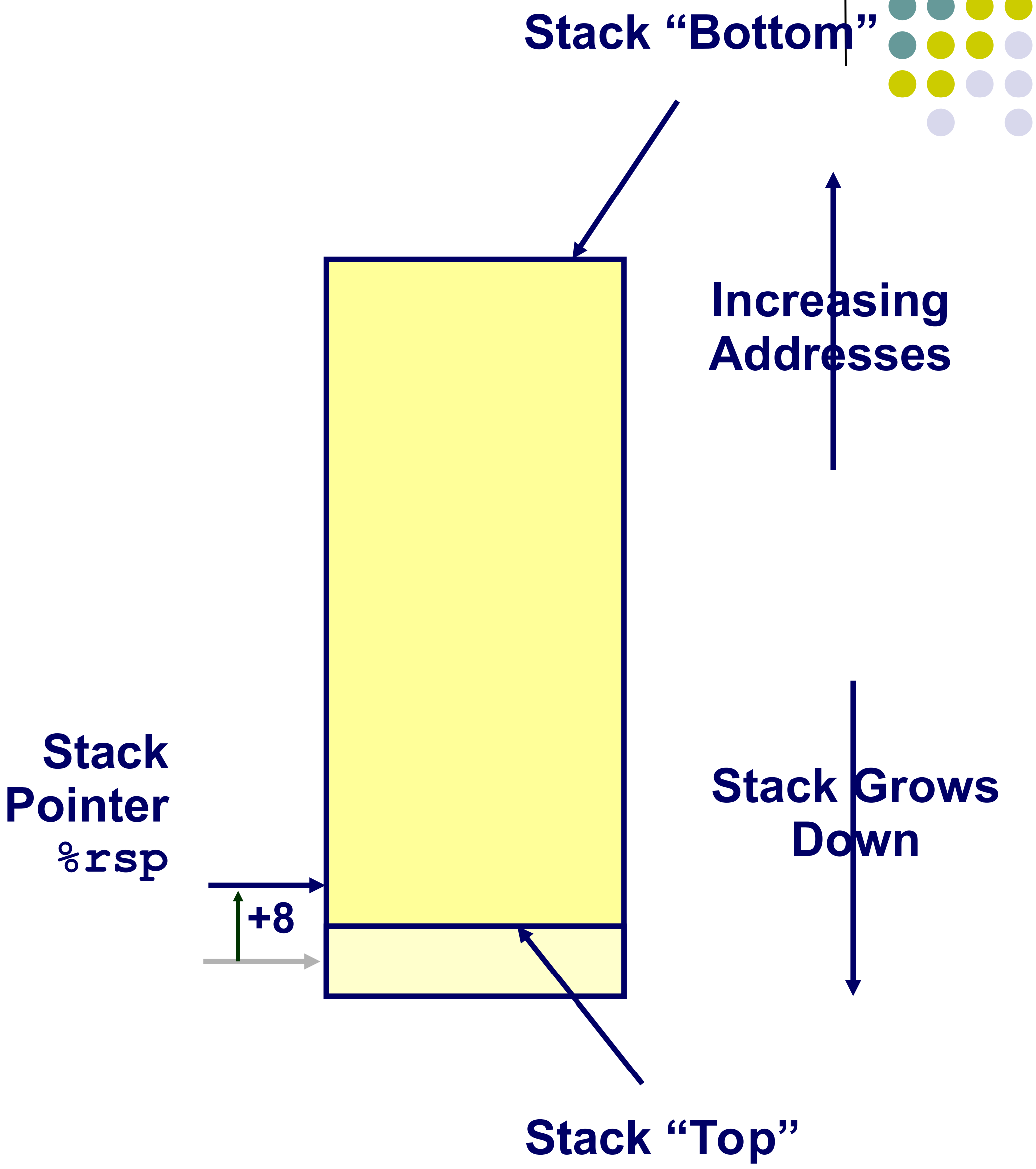
Stack Pushing

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

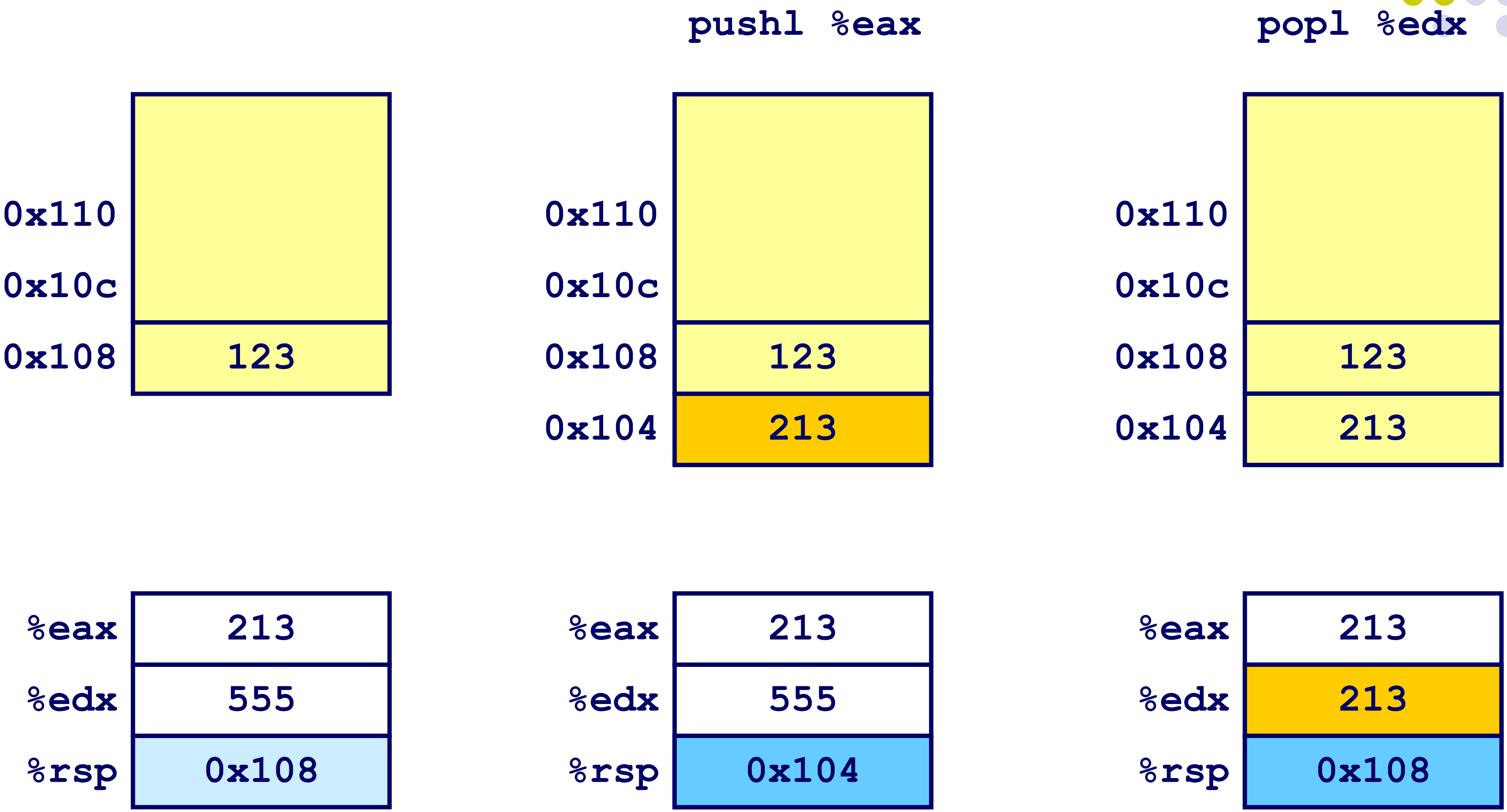


Stack Popping

- `popq` *Dest*
- Read operand at address given by `%rsp`
- Increment `%rsp` by 8
- Write to *Dest (register!)*



Stack Operation Examples
(32 bits: pushl, popl)



0x110	
0x10c	
0x108	123

%eax	213
%edx	555
%rsp	0x108

`pushl %eax`

0x110	
0x10c	
0x108	123
0x104	213

%eax	213
%edx	555
%rsp	0x104

`popl %edx`

0x110	
0x10c	
0x108	123
0x104	213

%eax	213
%edx	213
%rsp	0x108

Stack use in procedure calls

Procedure Control Flow

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

Code Example

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

```
0000000000400540 <multstore>:
  400540: push   %rbx              # Save %rbx
  400541: mov    %rdx,%rbx         # Save dest
  400544: callq  400550 <mult2>    # mult2(x,y)
  400549: mov    %rax,(%rbx)       # Save at dest
  40054c: pop    %rbx              # Restore %rbx
  40054d: retq                     # Return
```

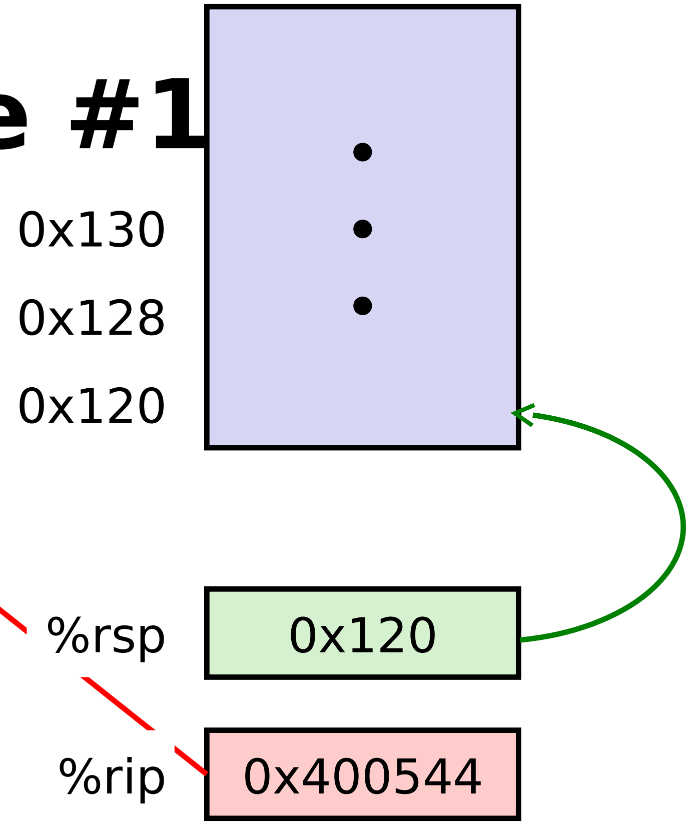
```
long mult2
  (long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax       # a
  400553:  imul   %rsi,%rax       # a * b
  400557:  retq                   # Return
```

Control Flow Example #1

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

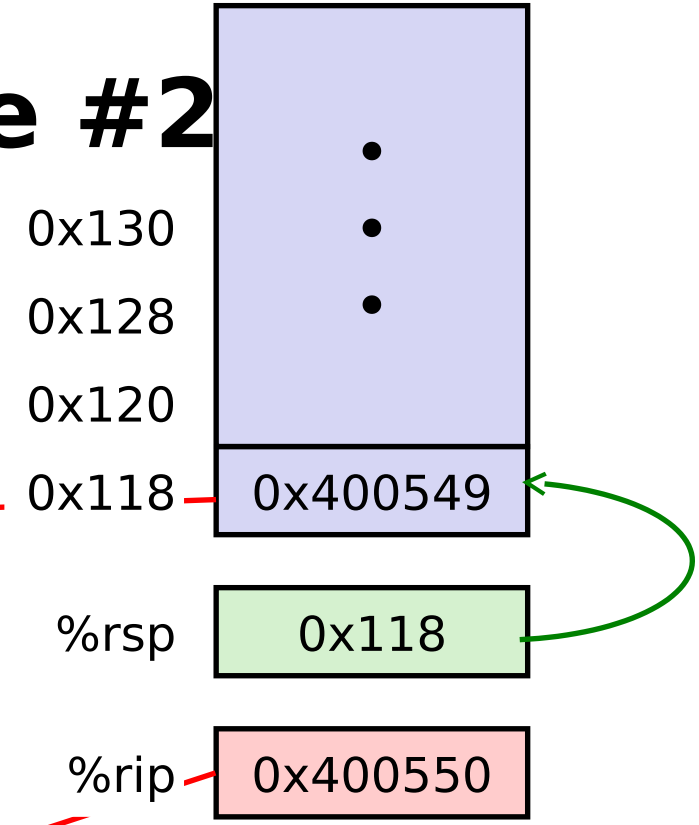
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Control Flow Example #2

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

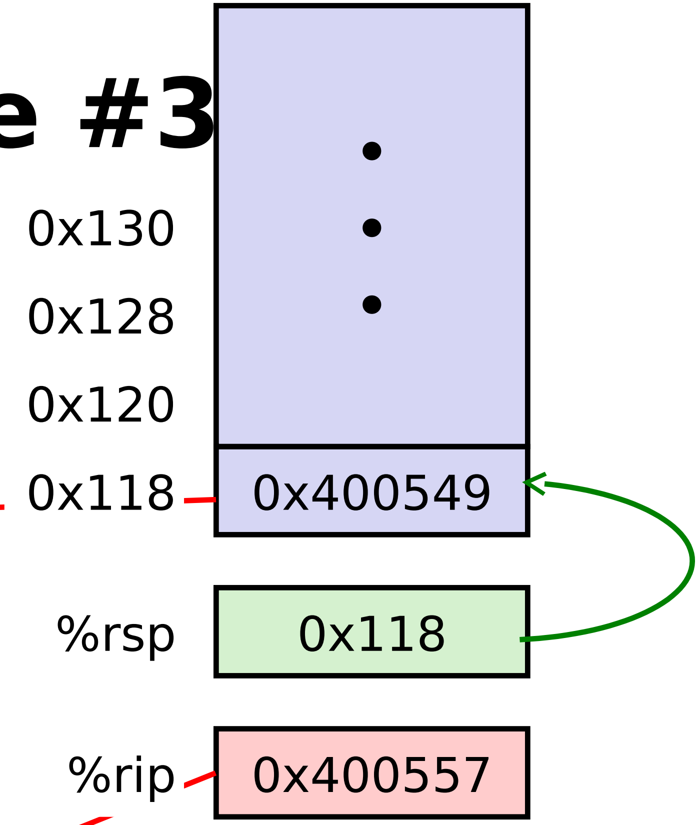
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Control Flow Example #3

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



• • •

0x130

0x128

0x120

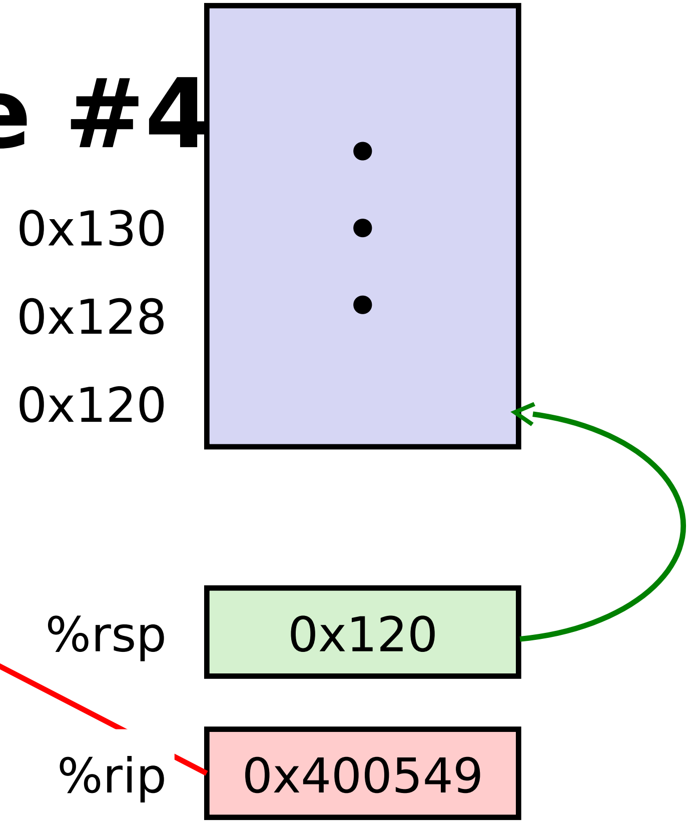
0x118 0x400549

%rsp 0x118

%rip 0x400557

Control Flow Example #4

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```



```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

… and in 32bit ISA:

Procedure Control Flow

Use stack to support procedure call and return

Procedure call:

`call` *`label`*   Push return address on stack; Jump to *`label`*

Return address value

Address of instruction beyond `call`

Example from disassembly

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

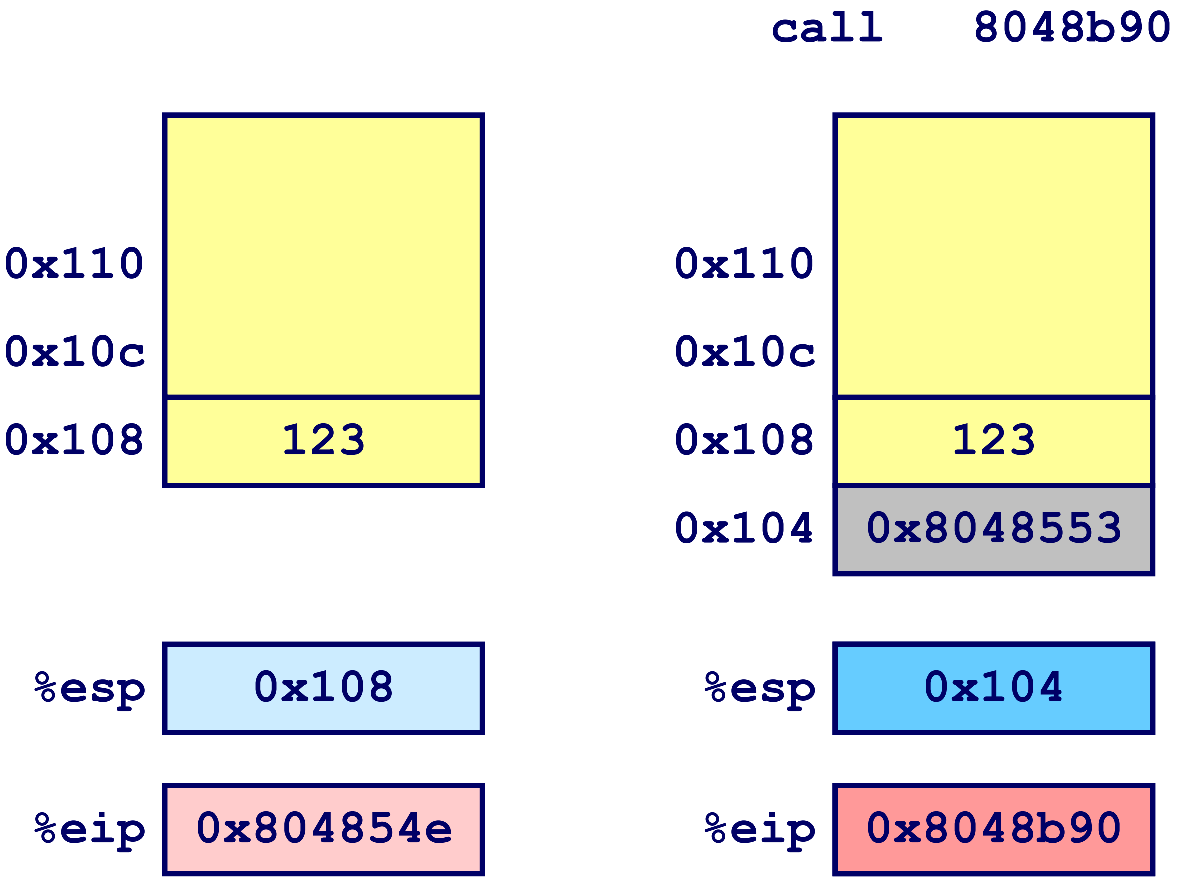
Return address = `0x8048553`

Procedure return:

`ret`   Pop address from stack; Jump to address

Procedure Call Example

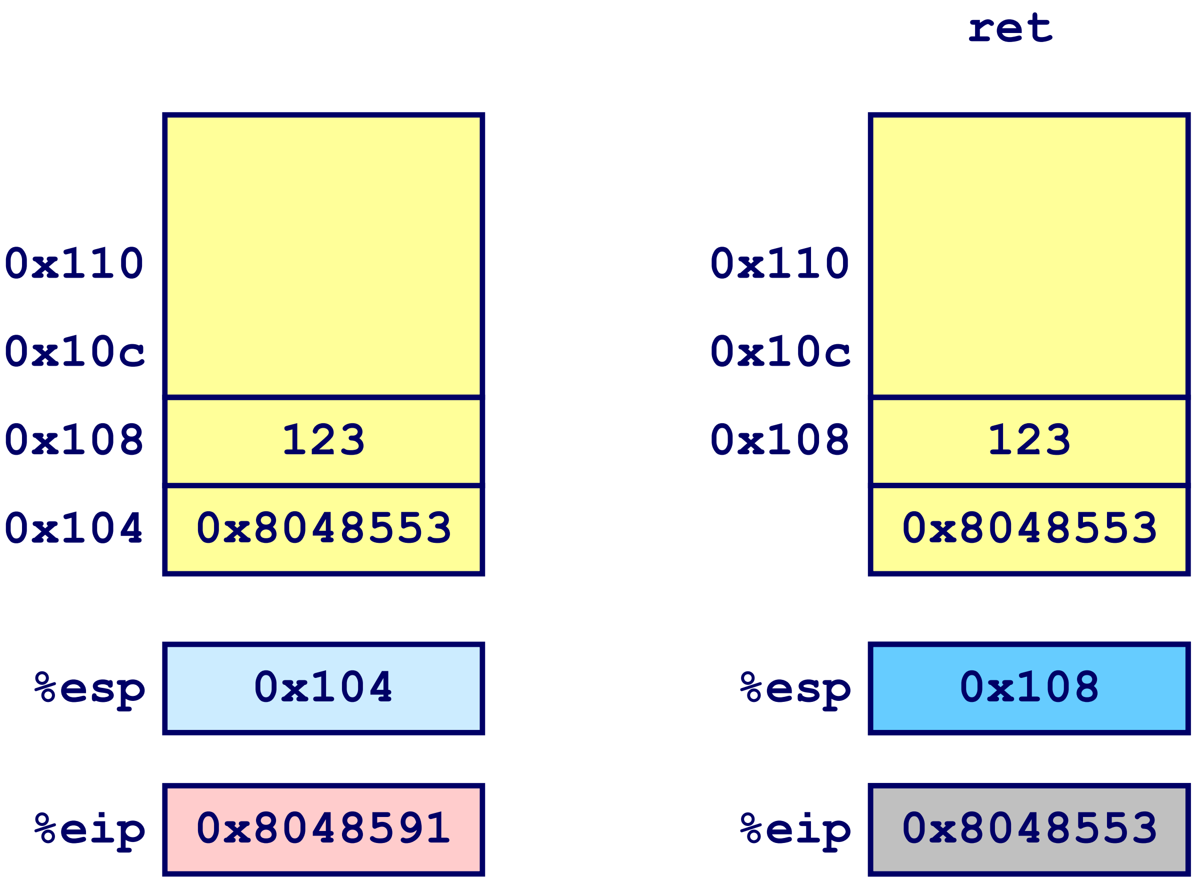
```
804854e:   e8 3d 06 00 00        call   8048b90 <main>
8048553:   50                    pushl  %eax
```



`%eip` **is program counter**

Procedure Return Example

```
8048591:   c3                    ret
```



ret

0x110
0x10c
0x108 123
0x104 0x8048553

%esp 0x104

%eip 0x8048591

0x110
0x10c
0x108 123
0x8048553

%esp 0x108

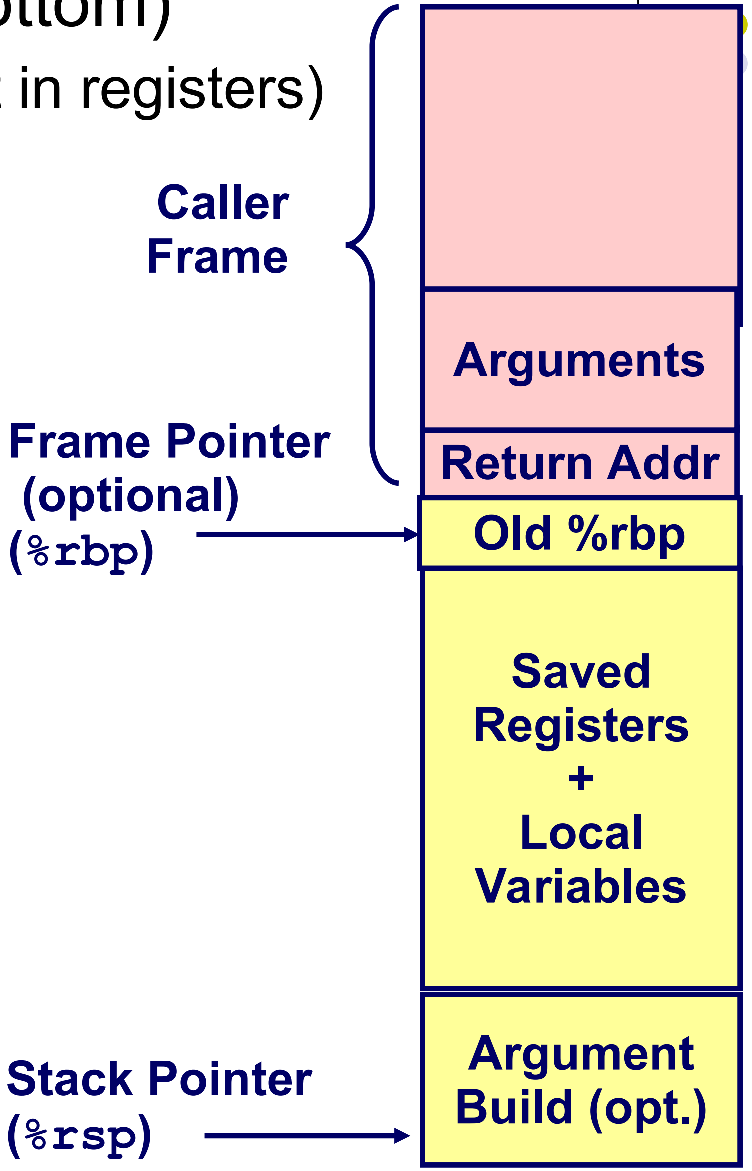
%eip 0x8048553

`%eip` **is program counter**

Linux Stack Frame

- Current Stack Frame ("Top" to Bottom)
  - Parameters for called function (if not in registers)
    - "Argument build"
  - Local variables
    - If can't keep in registers
  - Saved register context
  - Old frame pointer
- Caller Stack Frame
  - Return address
    - Pushed by `call` instruction
  - Arguments for this call

Changes between operating systems, compilers, linkers, etc. See REQUIRED reading on web page.



Example: incr (no recursion yet)

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

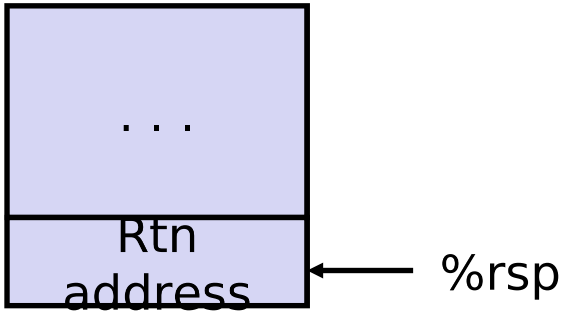
```
incr:
  movq    (%rdi), %rax
  addq    %rax, %rsi
  movq    %rsi, (%rdi)
  ret
```

Register	Use(s)
`%rdi`	Argument **`p`**
`%rsi`	Argument **`val`**, **`y`**
`%rax`	**`x`**, Return value

Example: Calling incr #1

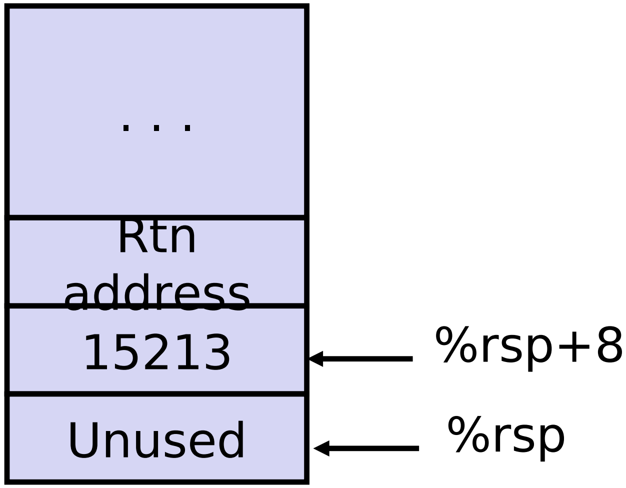
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

Initial Stack Structure



```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

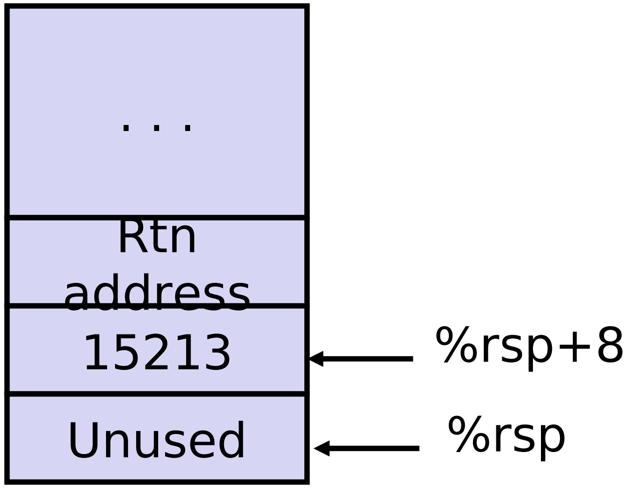
Resulting Stack Structure



Example: Calling incr #2

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

Stack Structure



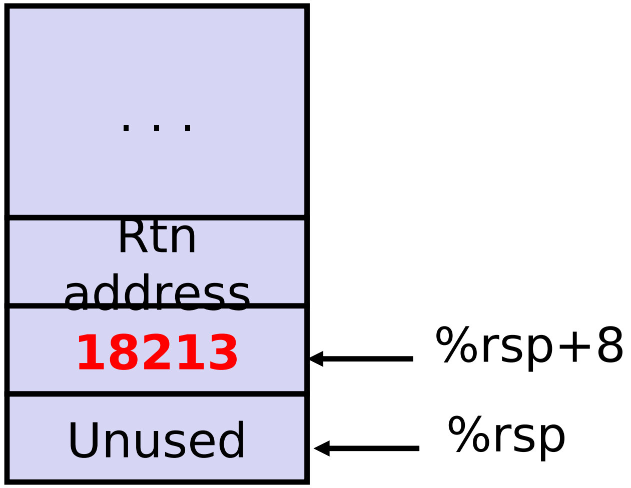
```
call_incr:
  subq     $16, %rsp
  movq     $15213, 8(%rsp)
  movl     $3000, %esi
  leaq     8(%rsp), %rdi
  call     incr
  addq     8(%rsp), %rax
  addq     $16, %rsp
  ret
```

Register	Use(s)
%rdi	&v1
%rsi	3000

Example: Calling incr #3

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

Stack Structure



```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

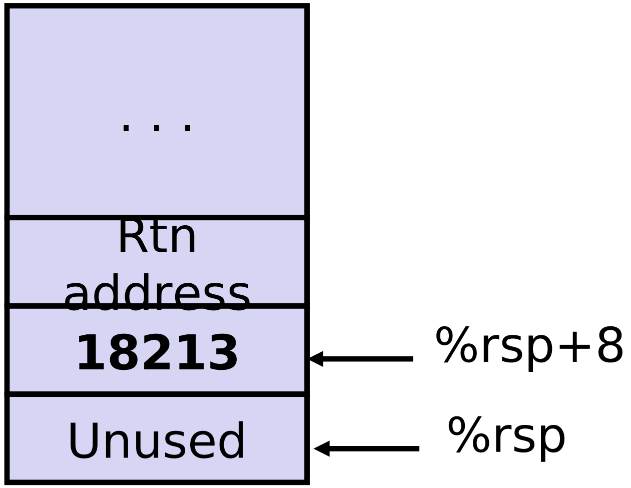
Register	Use(s)
%rdi	&v1
%rsi	3000

Example: Calling incr #4

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

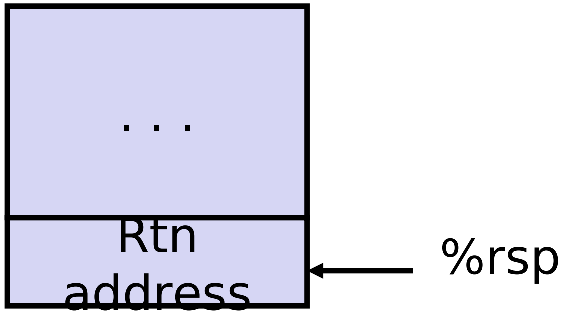
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure



Register	Use(s)
%rax	Return value

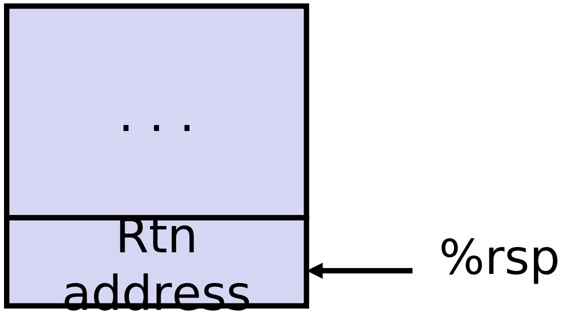
Updated Stack Structure



Example: Calling incr #5

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

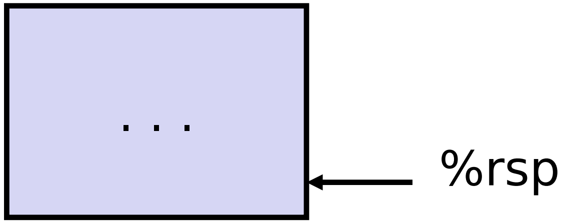
Updated Stack Structure



```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Register	Use(s)
`%rax`	Return value

Final Stack Structure



Register Saving Conventions

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

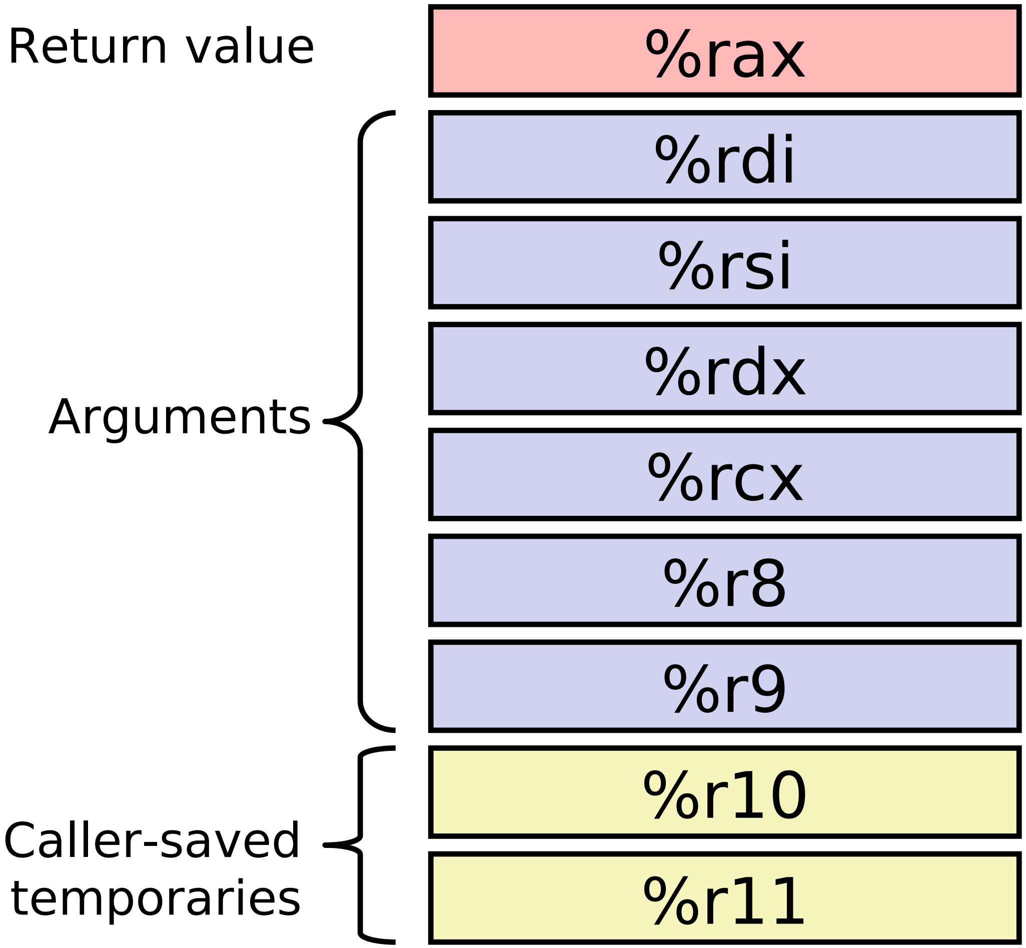
```
yoo:
    • • •
     movq $15213, %rdx
     call who
     addq %rdx, %rax
    • • •
     ret
```

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

  - Contents of register %rdx overwritten by who
  - This could be trouble → something should be done!
    - Need some coordination

x86-64 Linux Register Usage #1

- **%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



Return value

%rax

Arguments

%rdi

%rsi

%rdx

%rcx

%r8

%r9

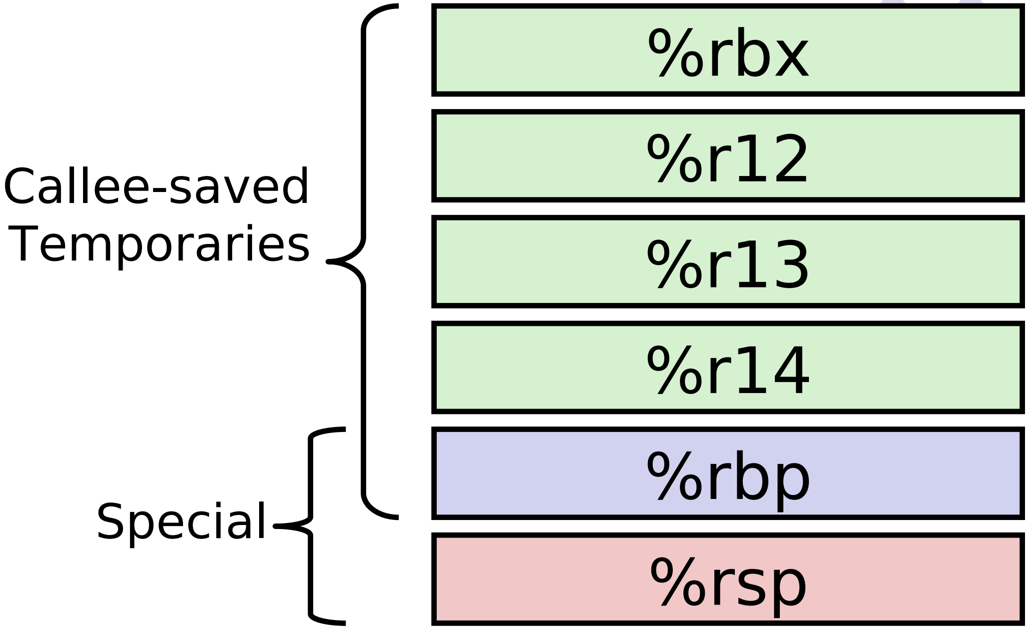
Caller-saved temporaries

%r10

%r11

x86-64 Linux Register Usage #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

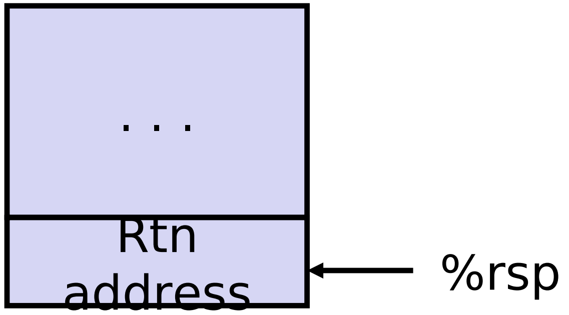


Callee-Saved Example #1

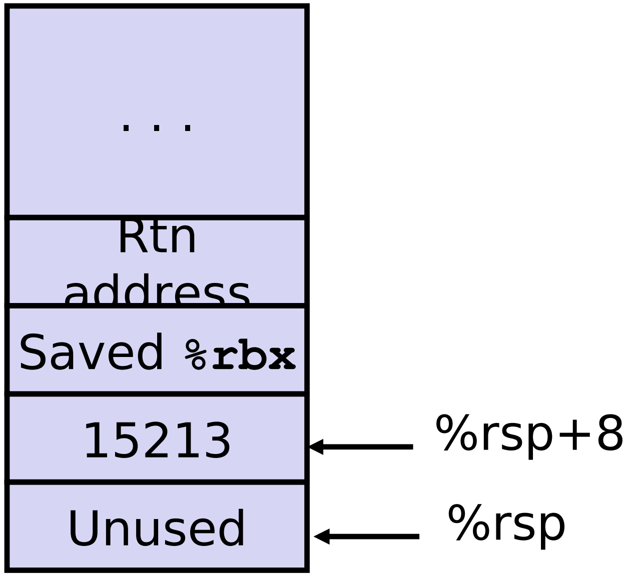
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq    %rbx
  subq     $16, %rsp
  movq     %rdi, %rbx
  movq     $15213, 8(%rsp)
  movl     $3000, %esi
  leaq     8(%rsp), %rdi
  call     incr
  addq     %rbx, %rax
  addq     $16, %rsp
  popq     %rbx
  ret
```

Initial Stack Structure



Resulting Stack Structure

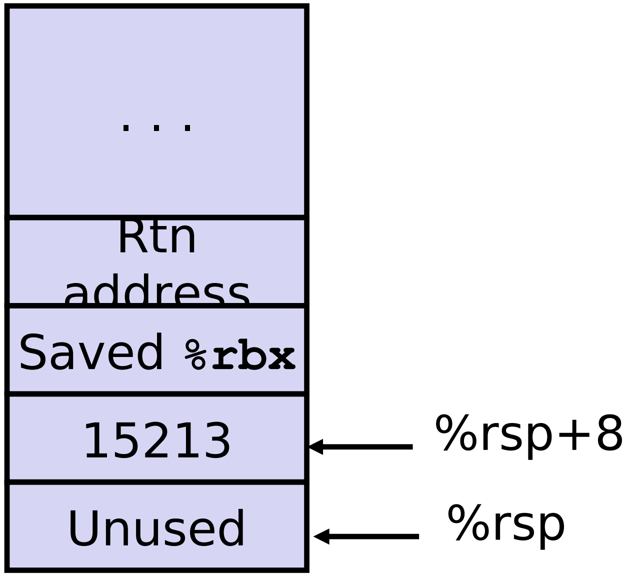


Callee-Saved Example #2

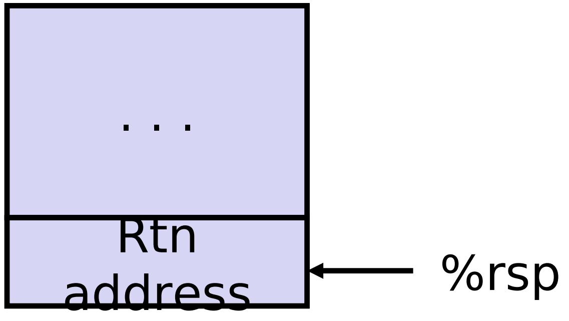
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Resulting Stack Structure



Pre-return Stack Structure



Calling Conventions in IA32

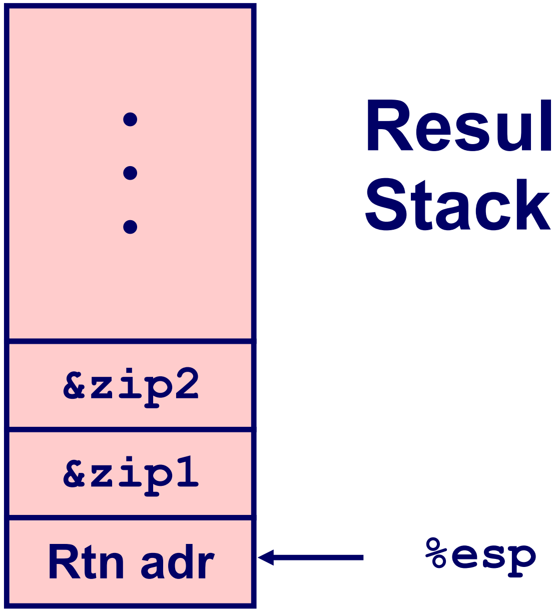
Revisiting swap

```
int zip1 = 15213;
int zip2 = 91125;

void call_swap()
{
  swap(&zip1, &zip2);
}
```

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

Calling swap from call_swap

```
call_swap:
    • • •
    pushl $zip2   # Global Var
    pushl $zip1   # Global Var
    call swap
    • • •
```



Resulting Stack

Revisiting swap

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

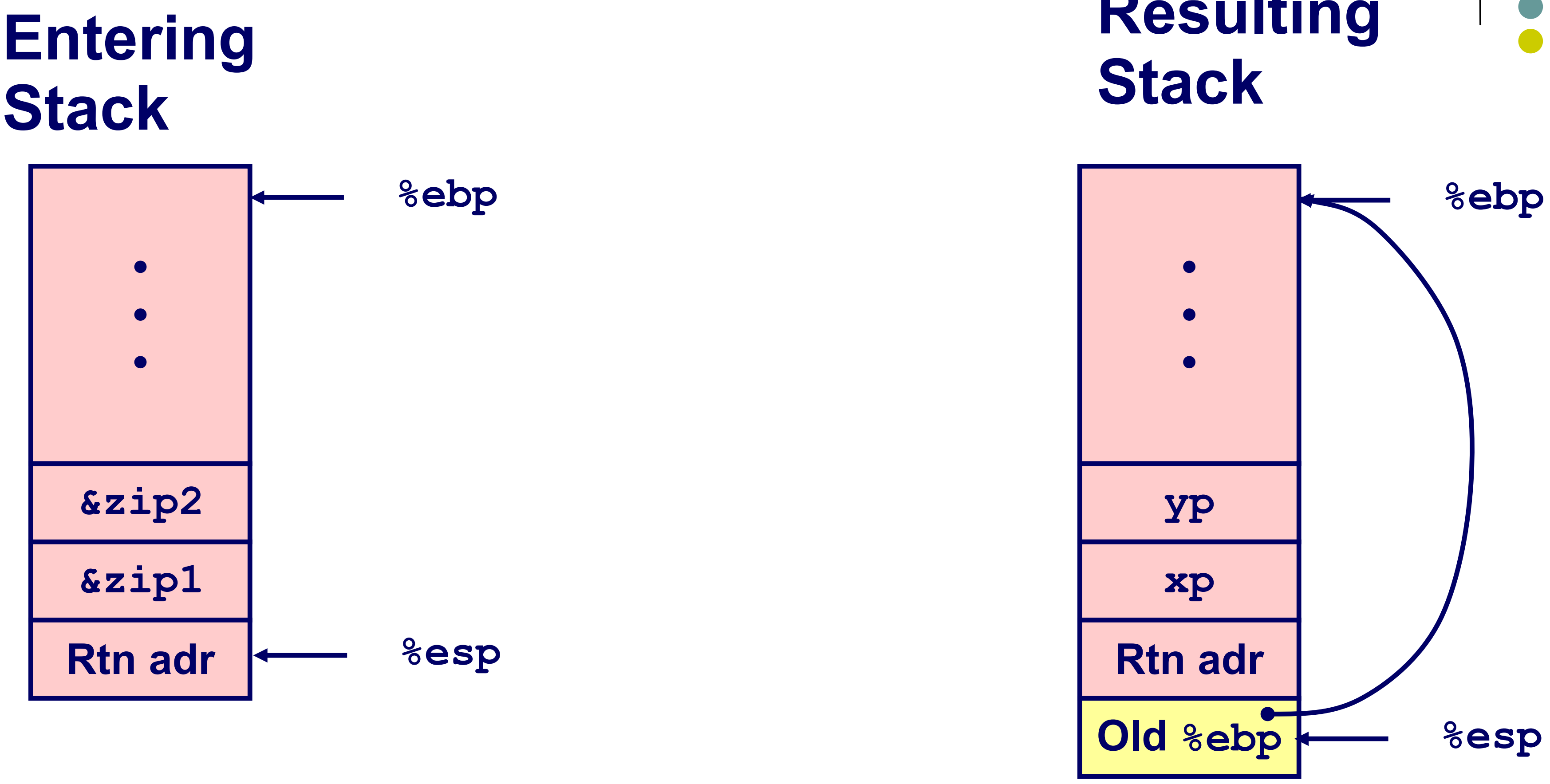
```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx

    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)

    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Set Up

Body

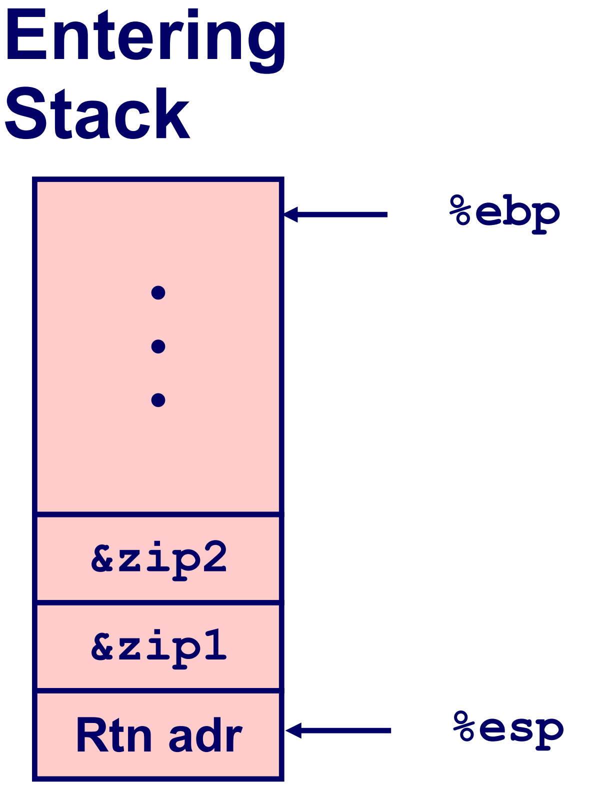
Finish

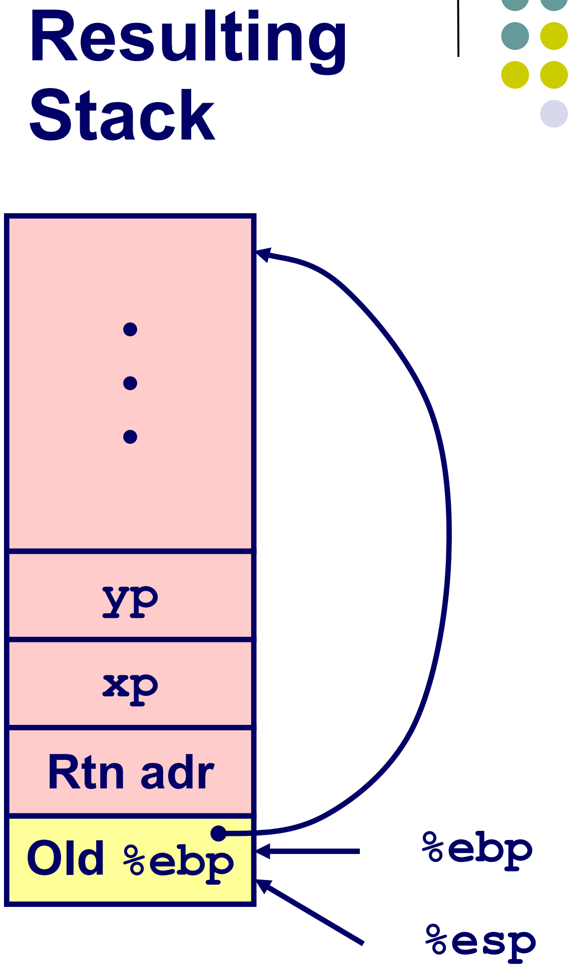
swap Setup #1



```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

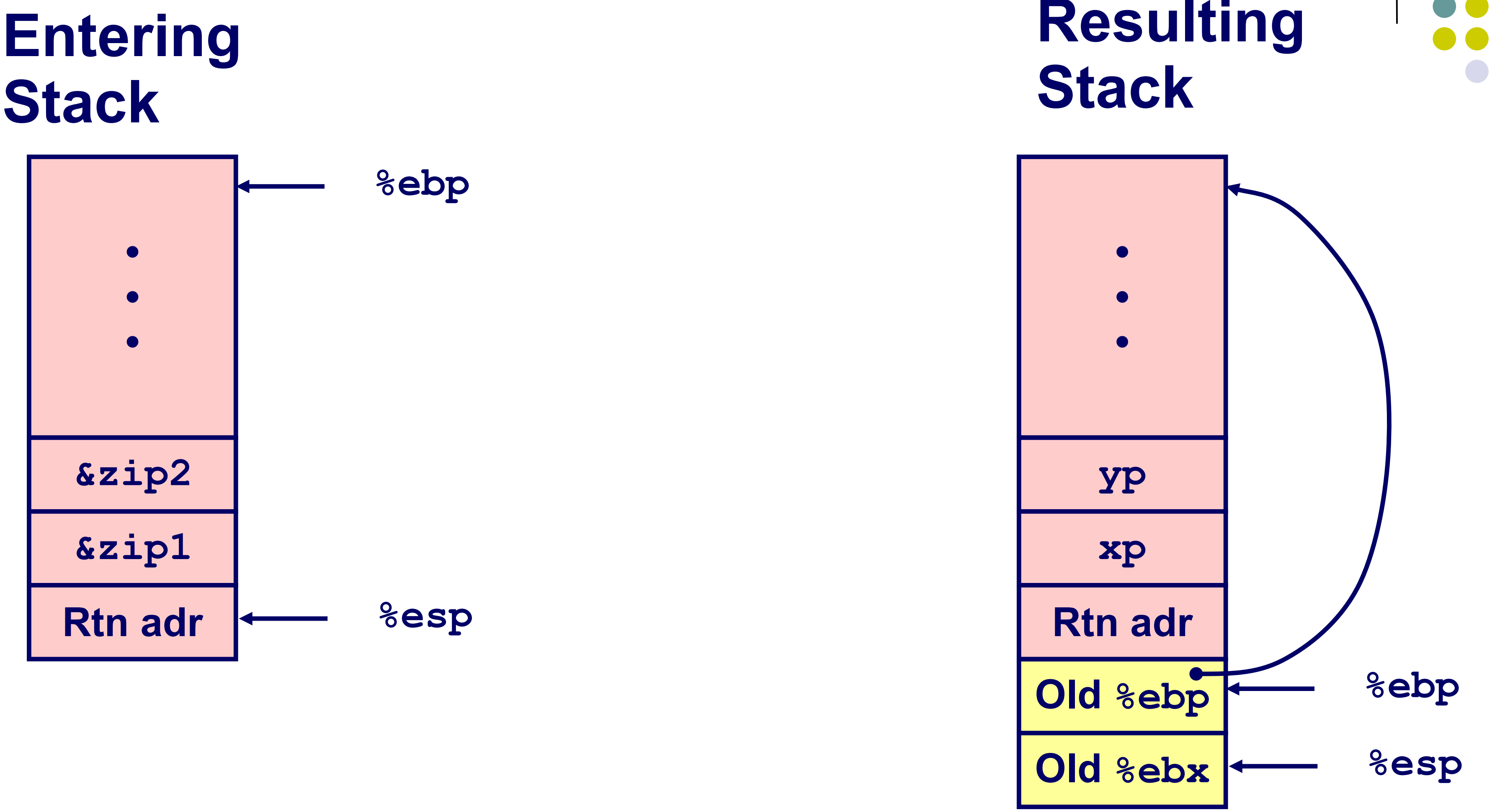
swap Setup #2





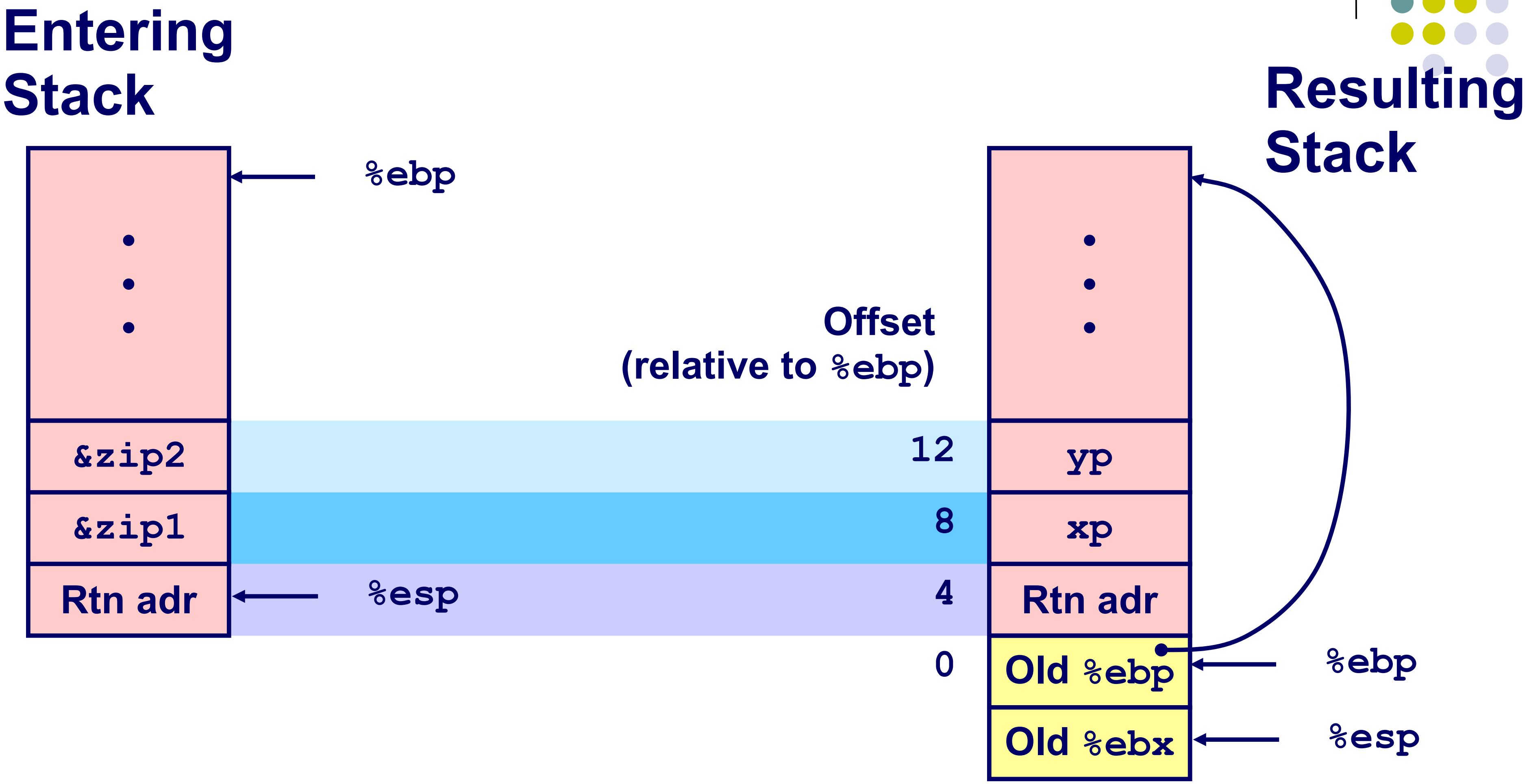
```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

swap Setup #3



```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

Effect of swap Setup

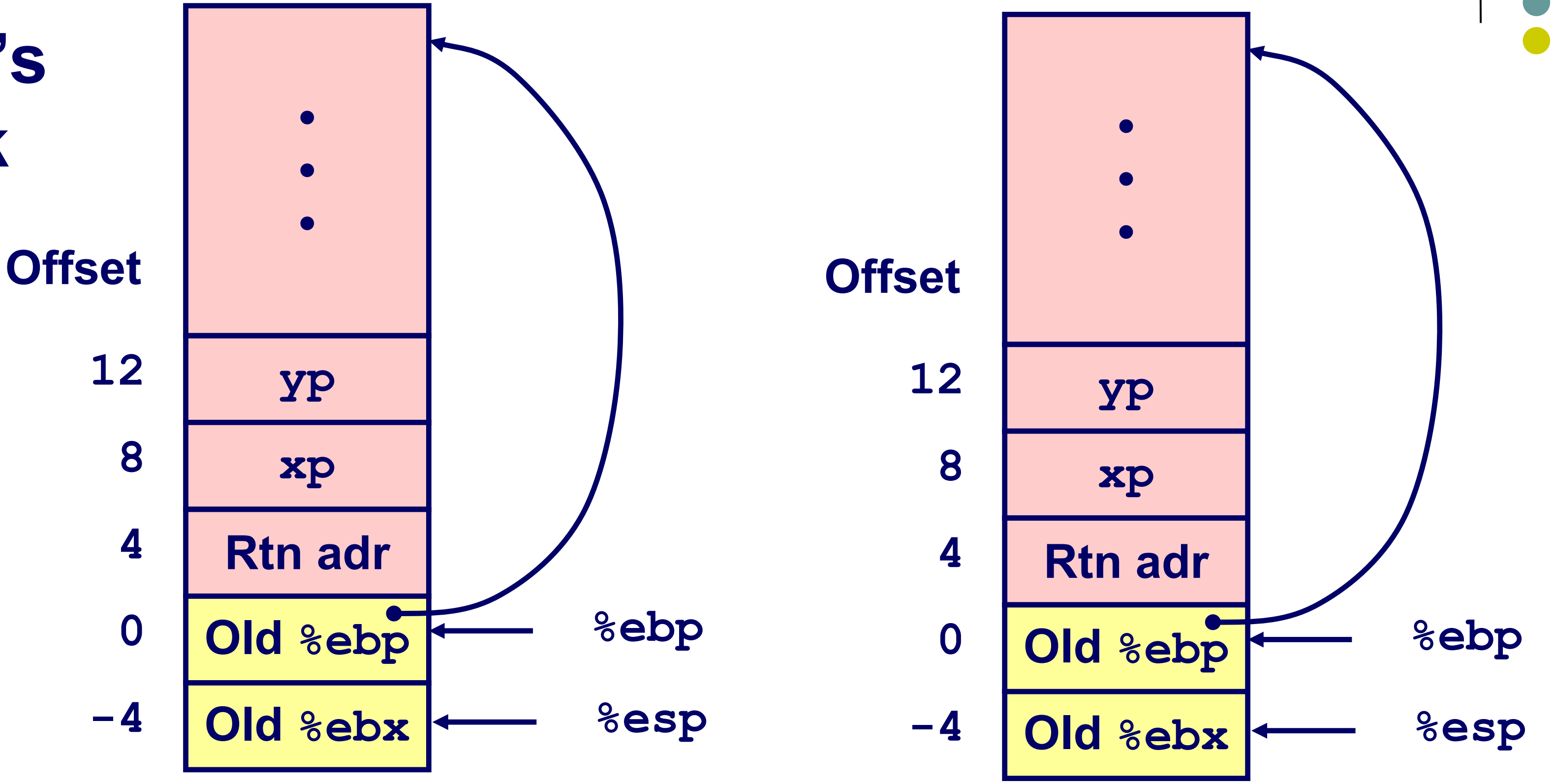


```
movl 12(%ebp),%ecx # get yp
movl 8(%ebp),%edx  # get xp
 . . .
```

Body

swap Finish #1

swap's Stack

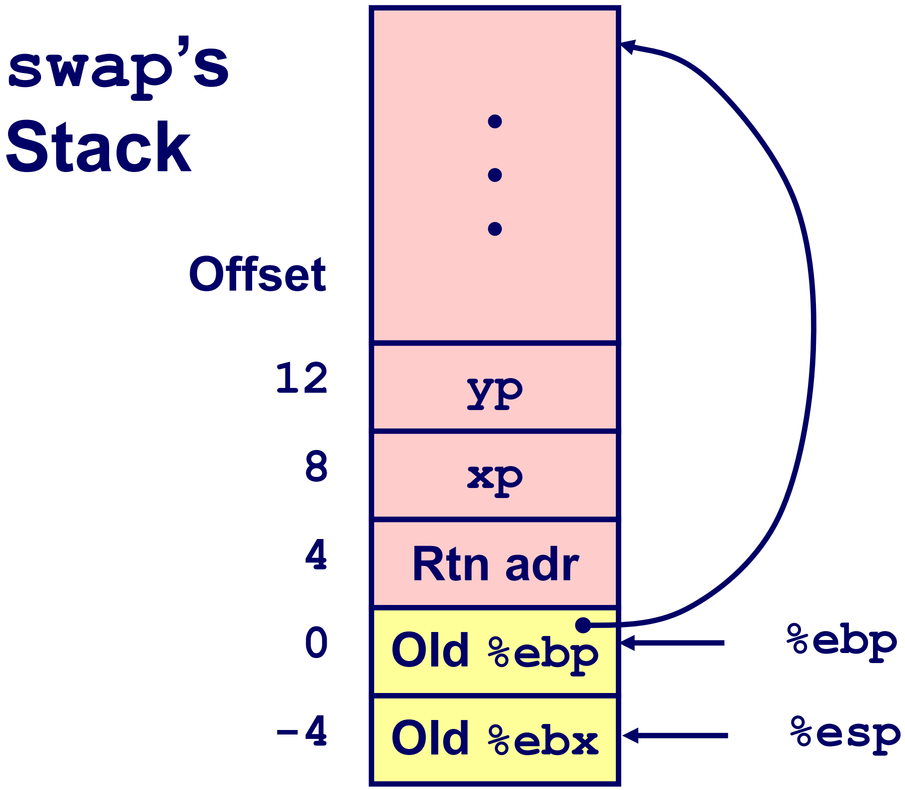


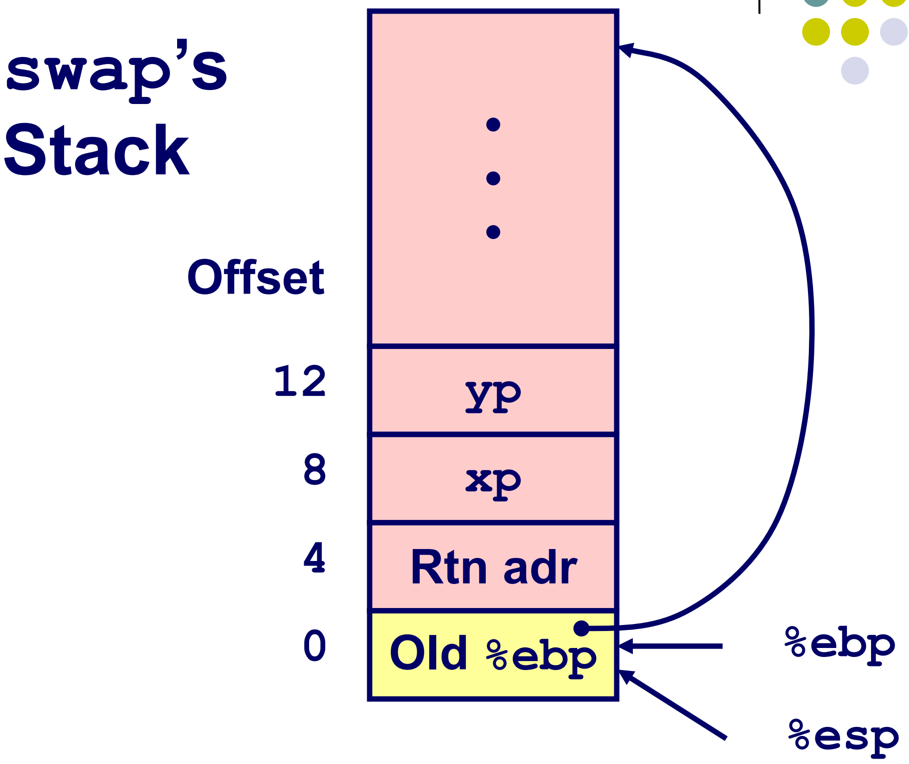
```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Observation

Saved & restored register `%ebx`

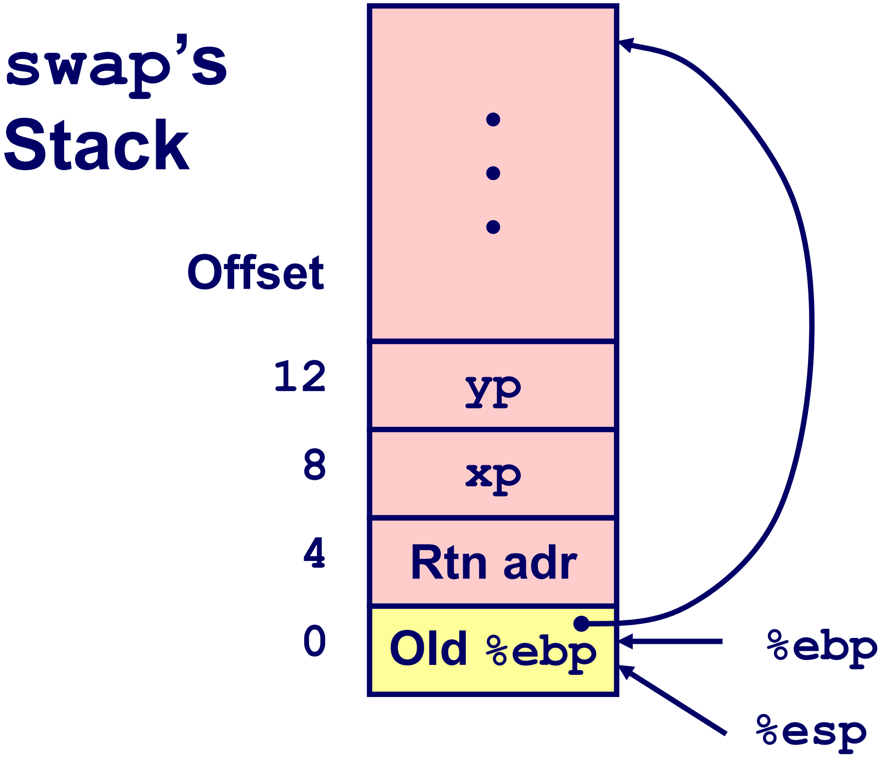
swap Finish #2

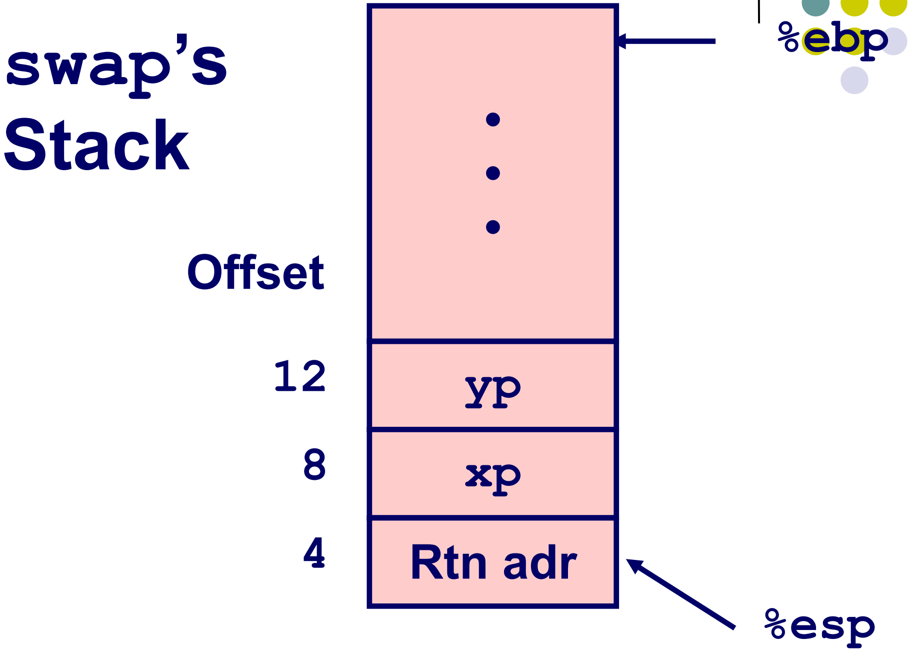




```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

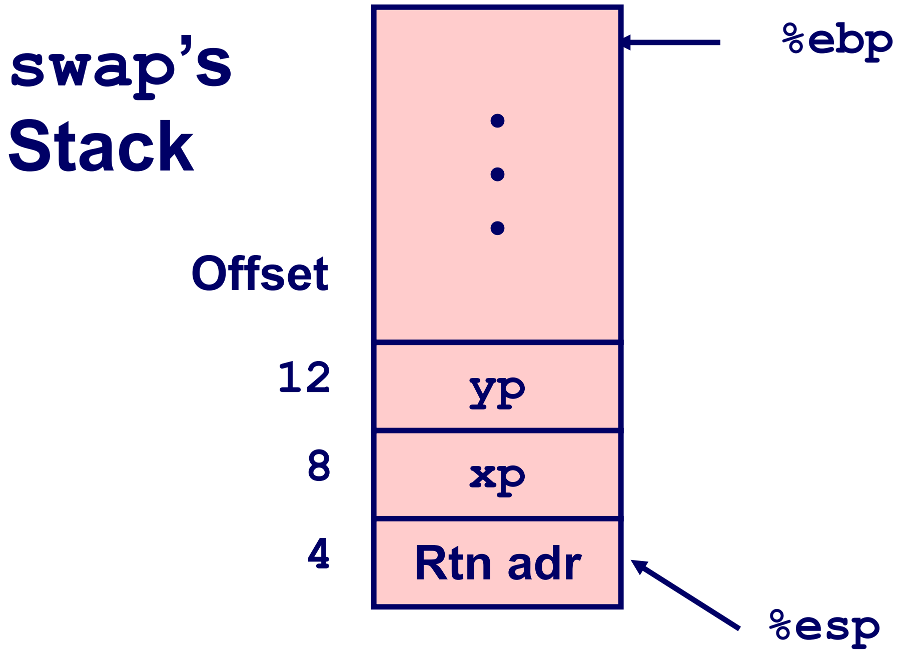
swap Finish #3

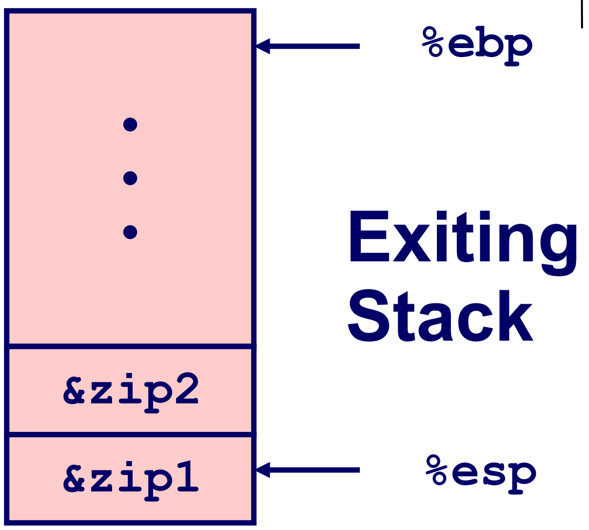




```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

swap Finish #4





```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Observation

- Saved & restored register `%ebx`
- Didn't do so for `%eax`, `%ecx`, or `%edx`

Register Saving Conventions

When procedure `yoo` calls `who`:

- `yoo` is the *caller*, `who` is the *callee*

Can Register be Used for Temporary Storage?

```
yoo:
    • • •
    movl $15213, %edx
    call who
    addl %edx, %eax
    • • •
    ret
```

```
who:
    • • •
    movl 8(%ebp), %edx
    addl $91125, %edx
    • • •
    ret
```

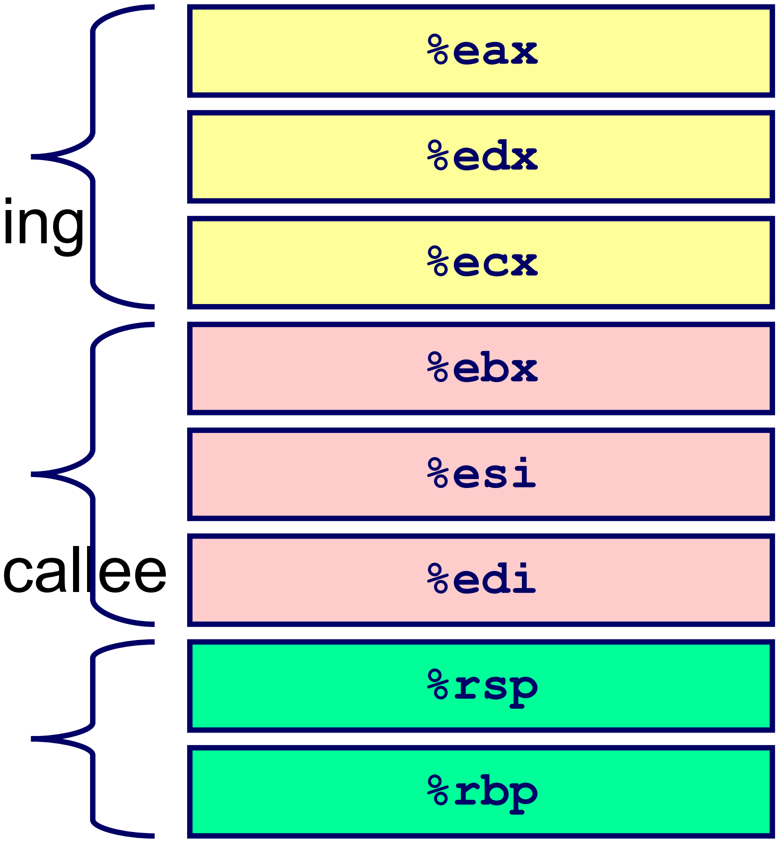
- Contents of register `%edx` overwritten by `who`

Register Saving Conventions

- When procedure `yoo` calls `who`:
  - `yoo` is the *caller*, `who` is the *callee*
- Can Register be Used for Temporary Storage?
- Conventions
  - "Caller Save"
    - Caller saves temporary in its frame before calling
  - "Callee Save"
    - Callee saves temporary in its frame before using

IA32/Linux Integer Register Usage

- Two have special uses: `%rbp`, `%rsp`
- Three managed as callee-save
  - `%ebx`, `%esi`, `%edi`
  - Old values saved on stack before using
- Three managed as caller-save
  - `%eax`, `%edx`, `%ecx`
  - Do what you please, but expect any callee to do so, as well



- Register `%eax` also stores returned value

Recursion

Recursion uses the stack!

- Code must be "*Reentrant*"
    - Multiple simultaneous instantiations of single procedure
    - Use stack to store state of each instantiation:
        - Arguments
        - Local variables, saved registers
        - Return pointers
- 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

Call Chain Example

Code Structure

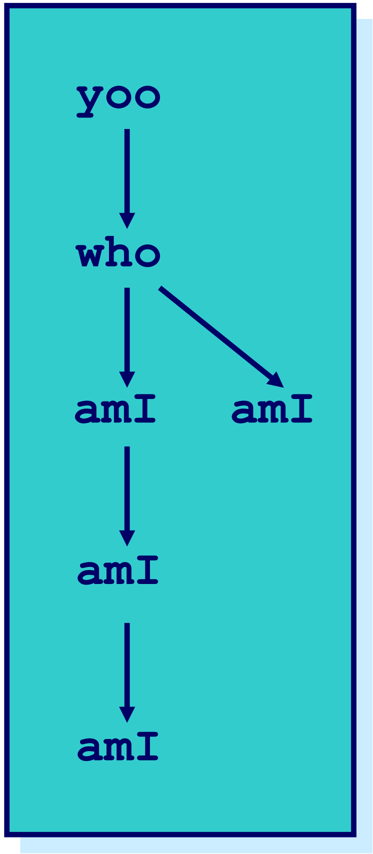
```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

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

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

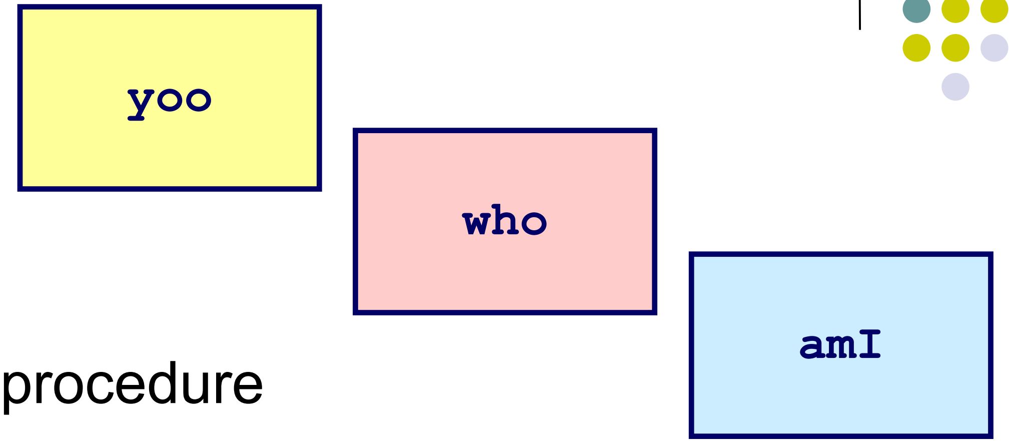
Procedure `amI` recursive

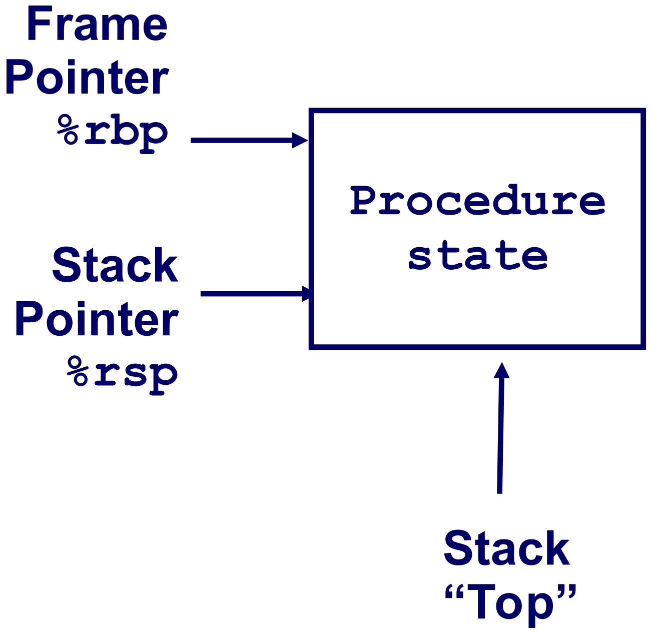
Call Chain



Stack Frames

- Contents
  - Local variables
  - Return information
  - Temporary space
- Management
  - Space allocated when enter procedure
    - “Set-up” code
  - Deallocated when return
    - “Finish” code
- Pointers
  - Stack pointer `%rsp` indicates stack top
  - Frame pointer `%rbp` indicates start of current frame



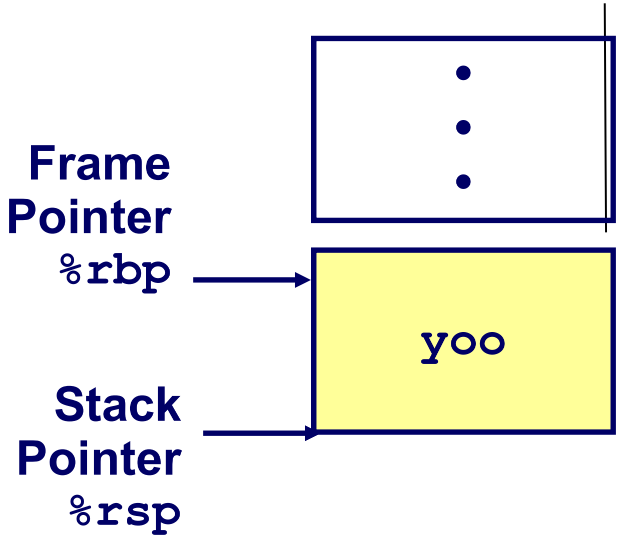


Stack Operation

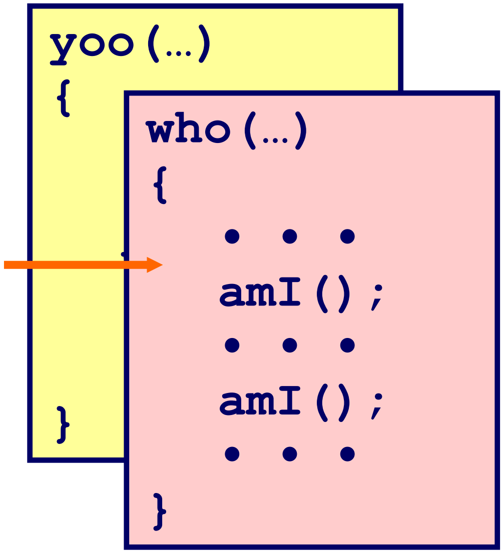
```
yoo(…)
{
    •
    •
    who();
    •
    •
}
```

Call Chain

yoo



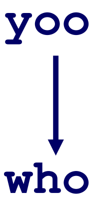
Stack Operation

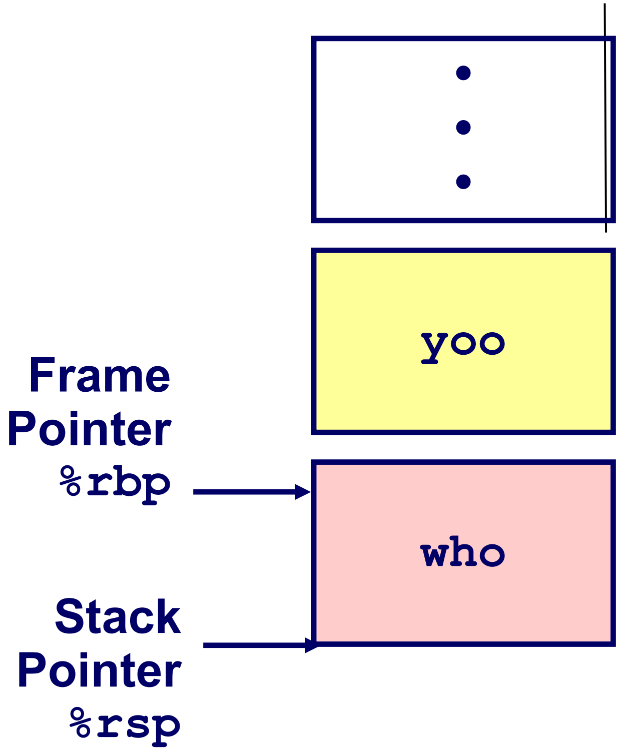


```
yoo(…)
{
  •
  •
}
```

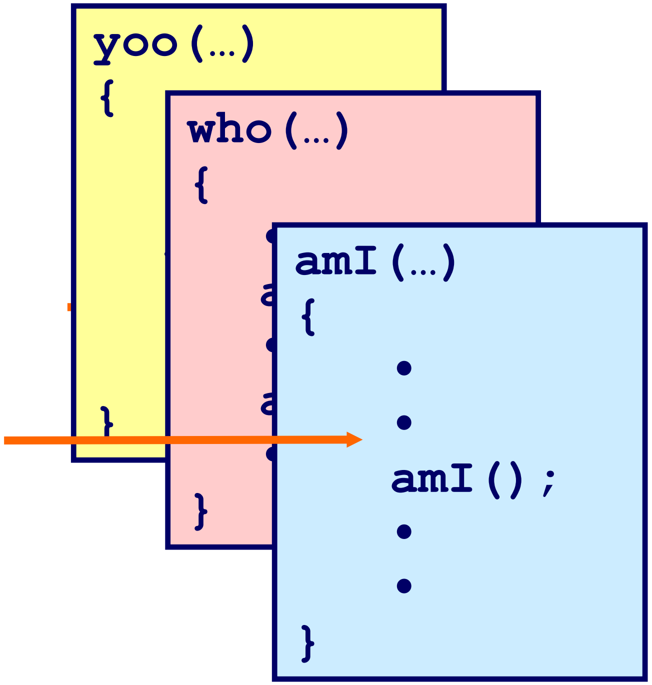
```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

Call Chain





Stack Operation

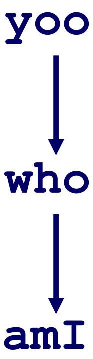


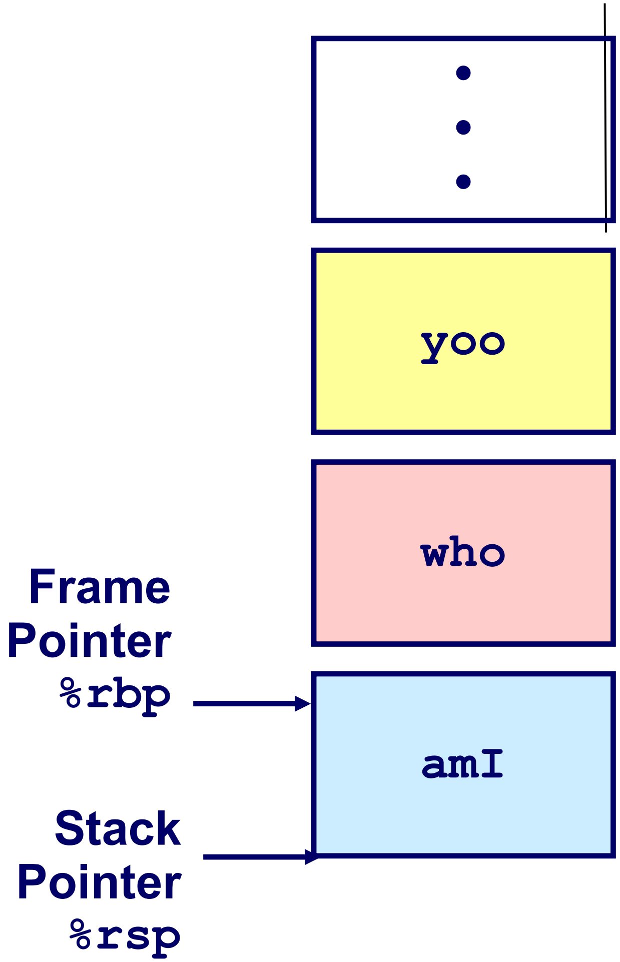
```
yoo(…)
{
}
```

```
who(…)
{
}
```

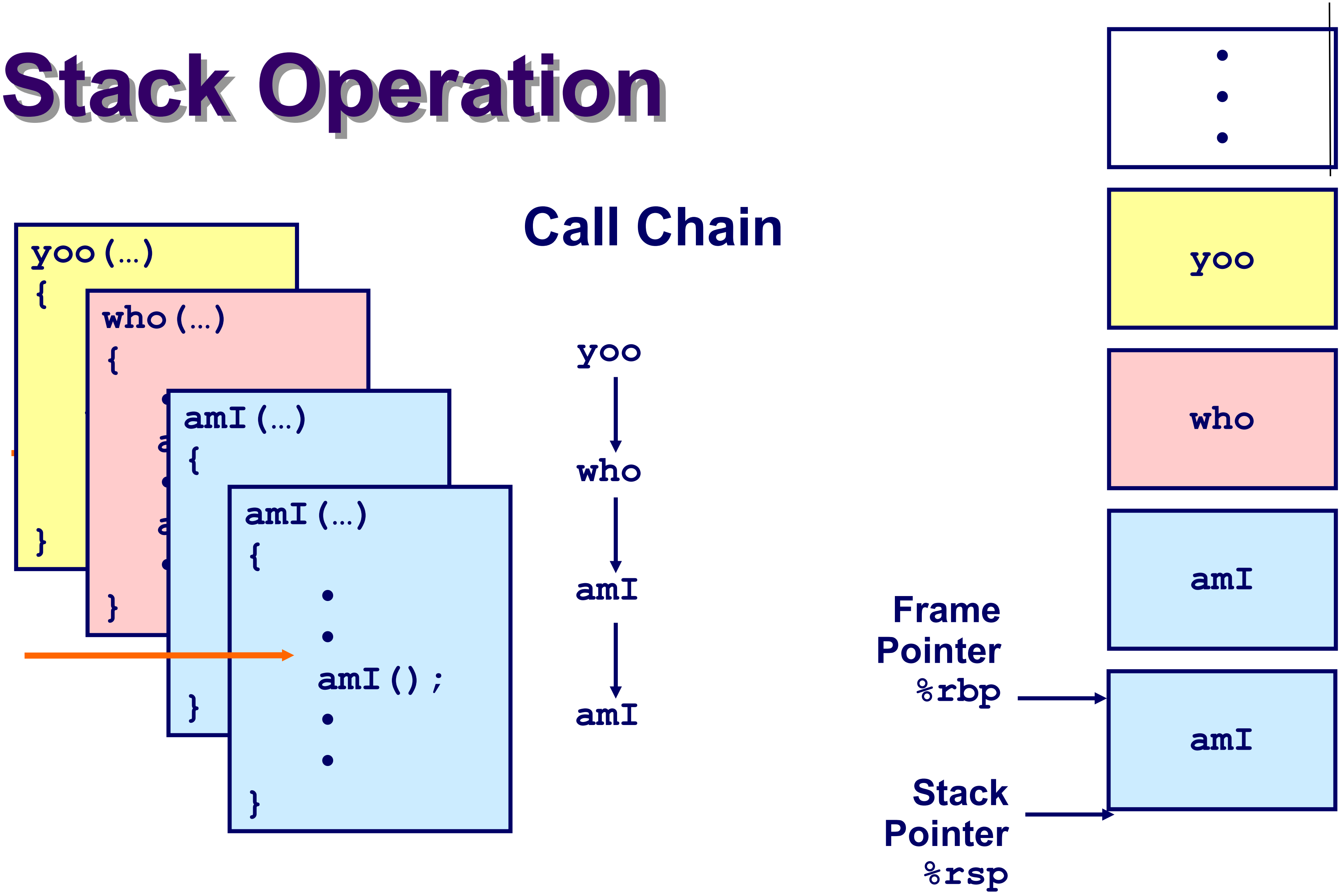
```
amI(…)
{
    •
    •
    amI();
    •
    •
}
```

Call Chain





Stack Operation

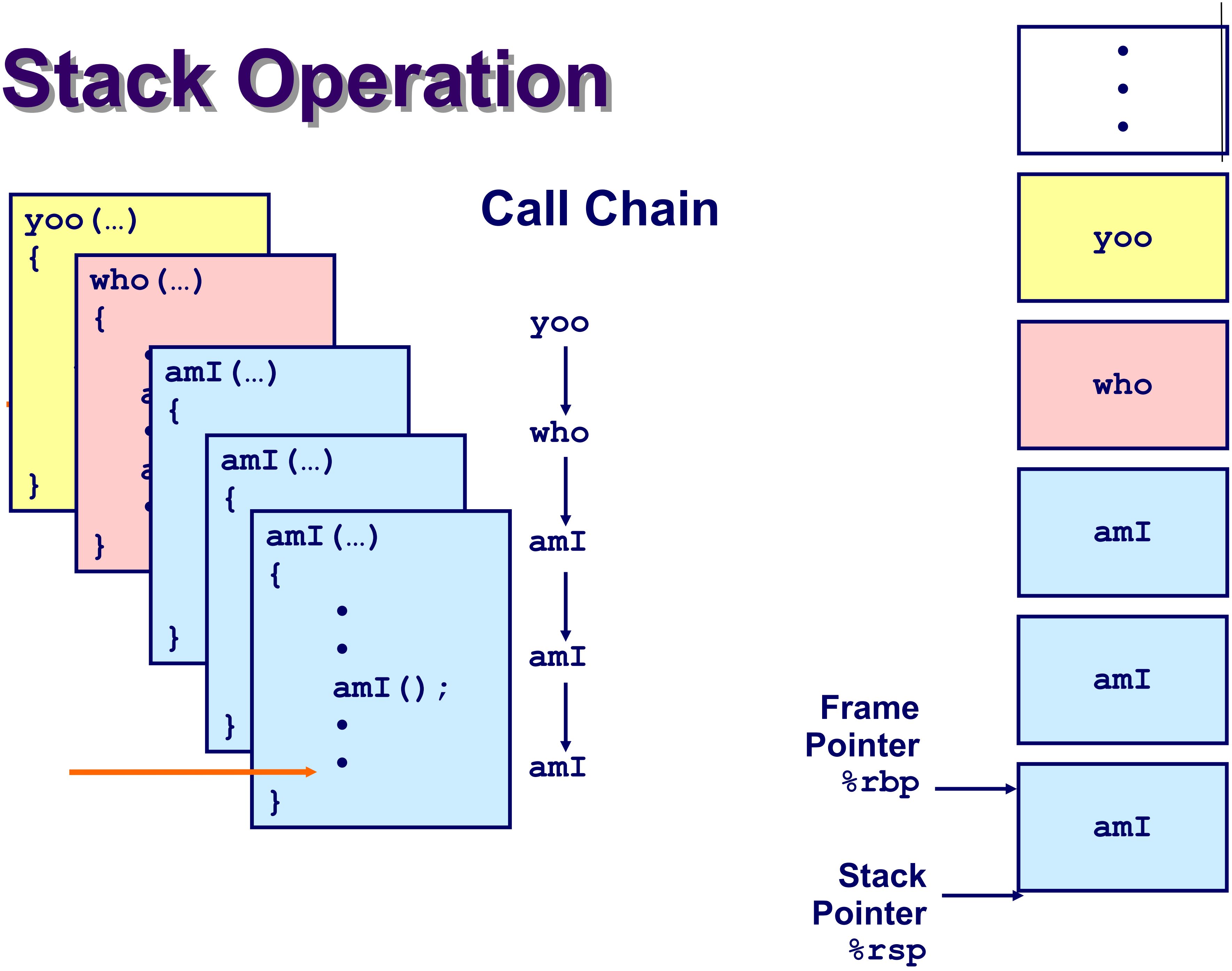


Call Chain

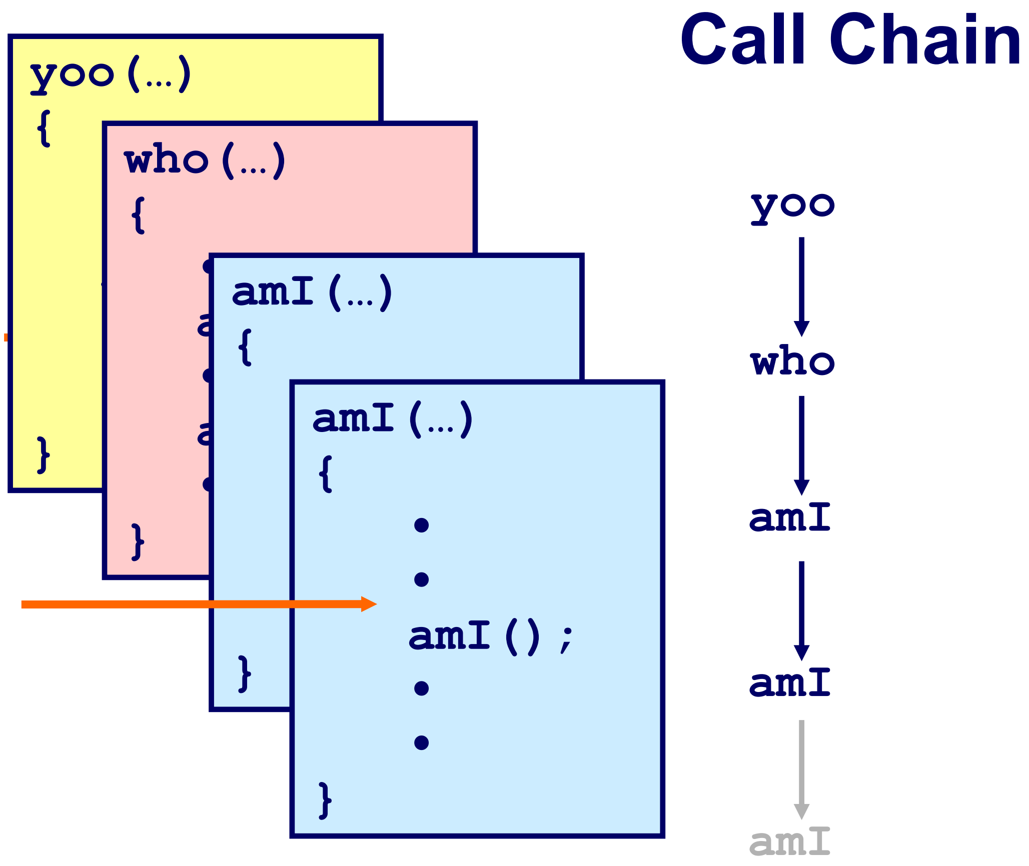
```
amI(…)
{
      •
      •
    amI();
      •
      •
}
```

Stack Operation

Call Chain



Stack Operation



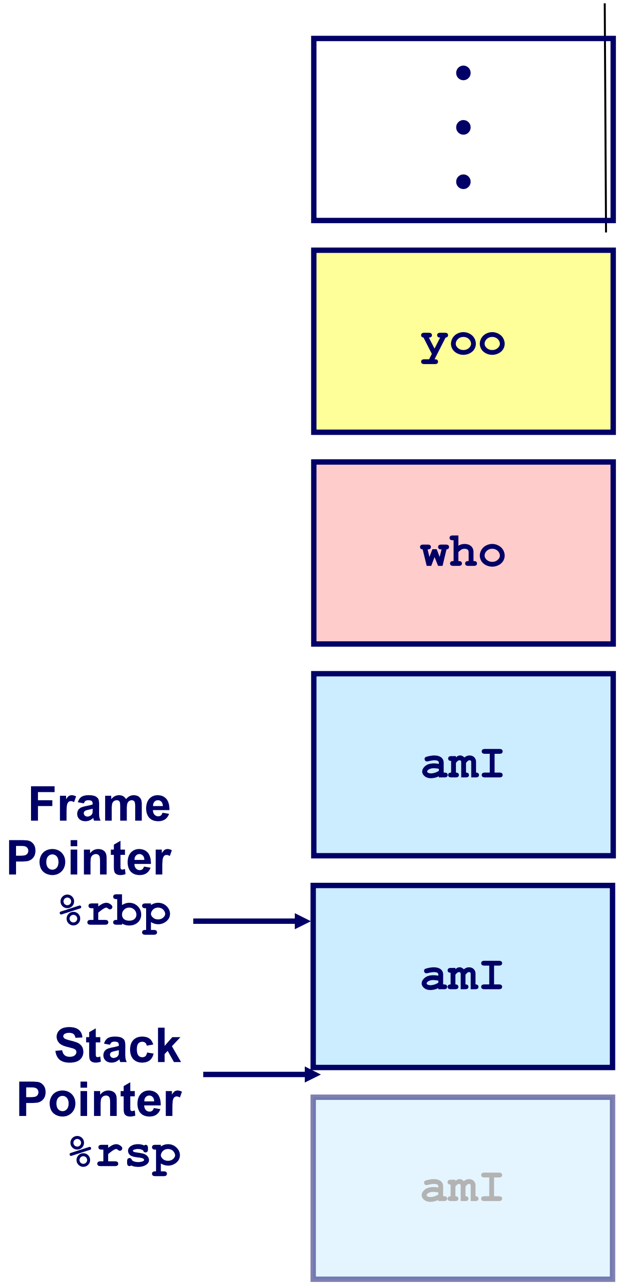
```
yoo(…)
{

}
```

```
who(…)
{

}
```

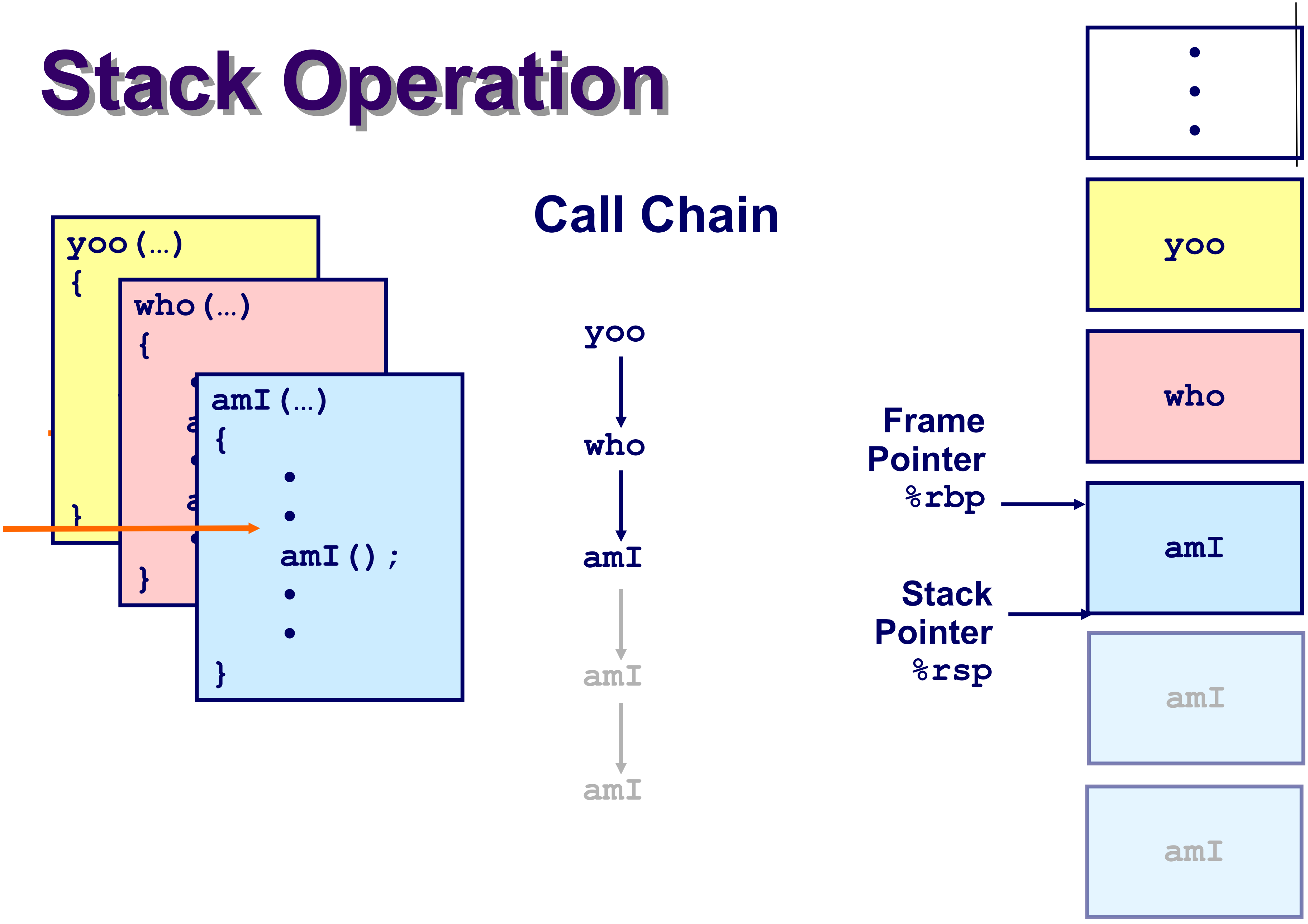
```
amI(…)
{

}
```

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

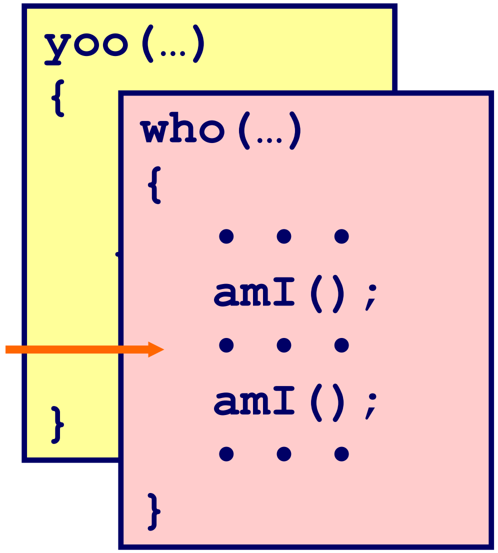
Call Chain



Stack Operation



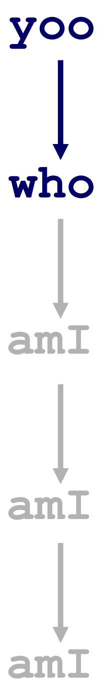
Stack Operation

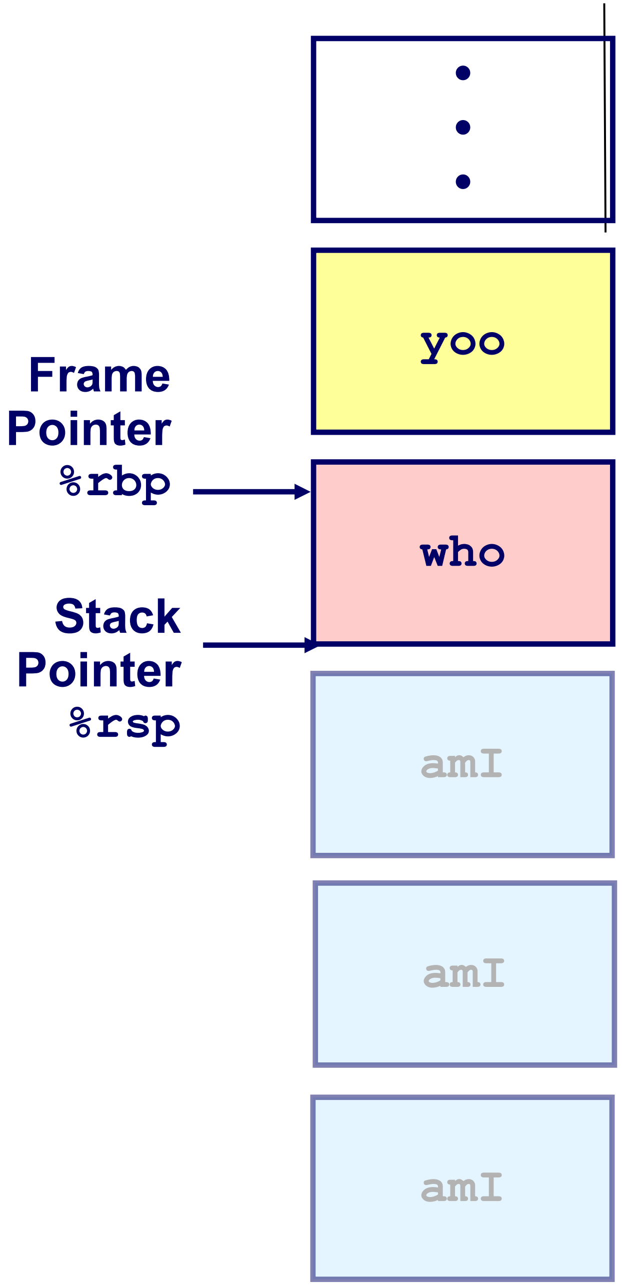


```
yoo(…)
{

}
```

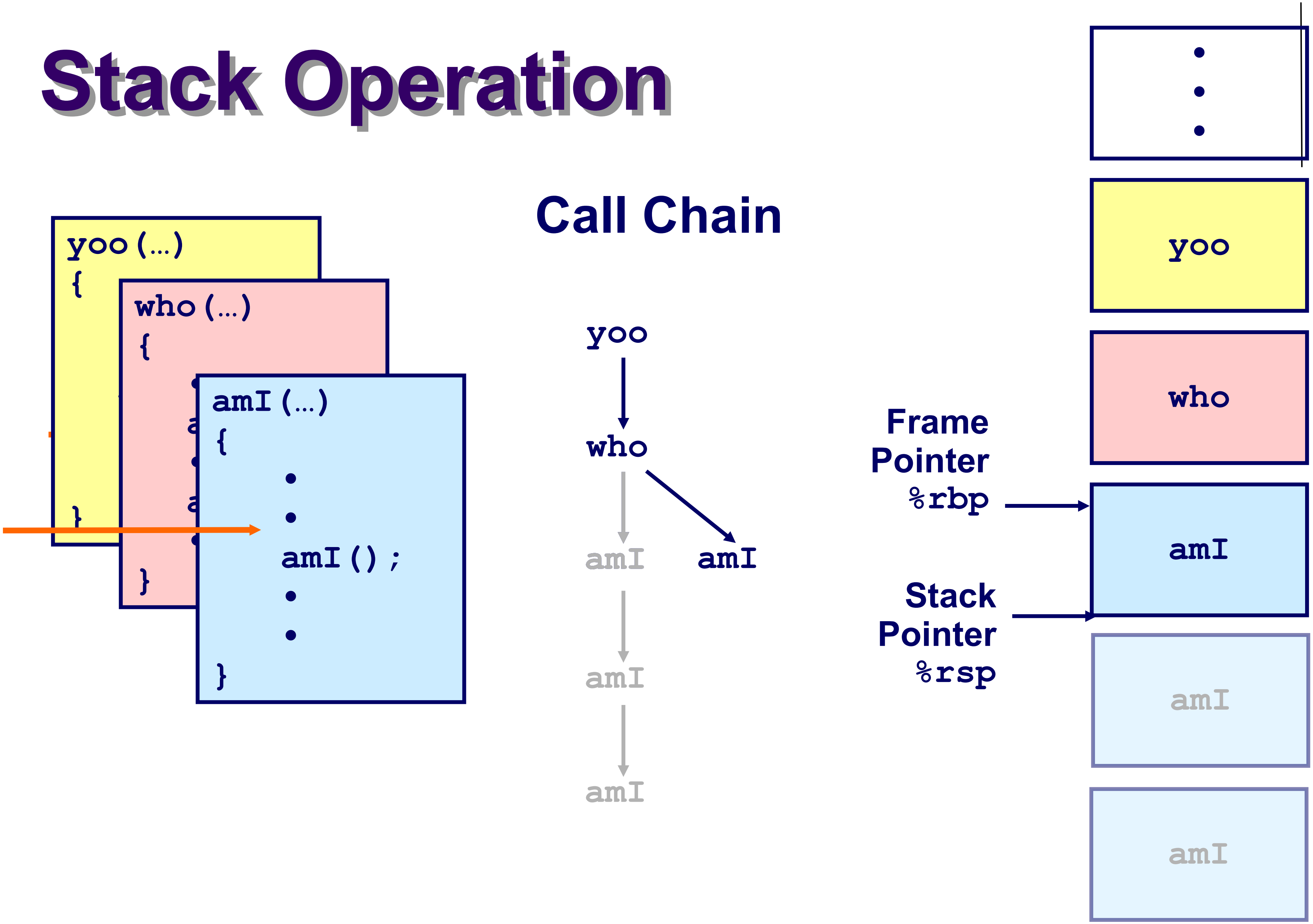
```
who(…)
{
  • • •
  amI();
  • • •
  amI();
  • • •
}
```

Call Chain

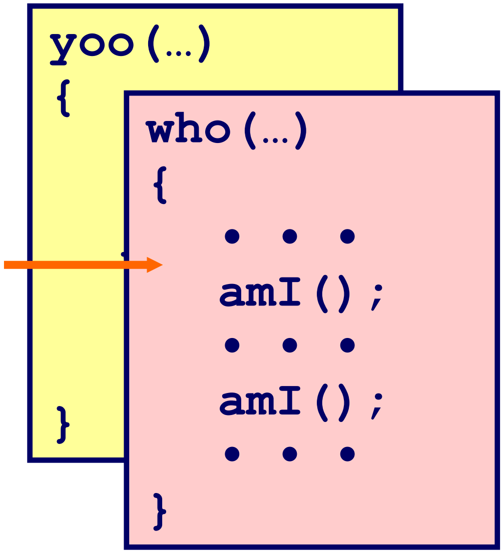




Stack Operation



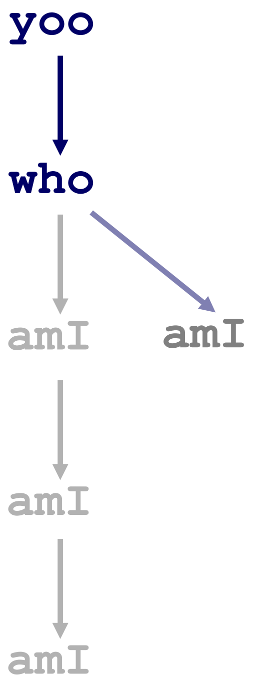
Stack Operation

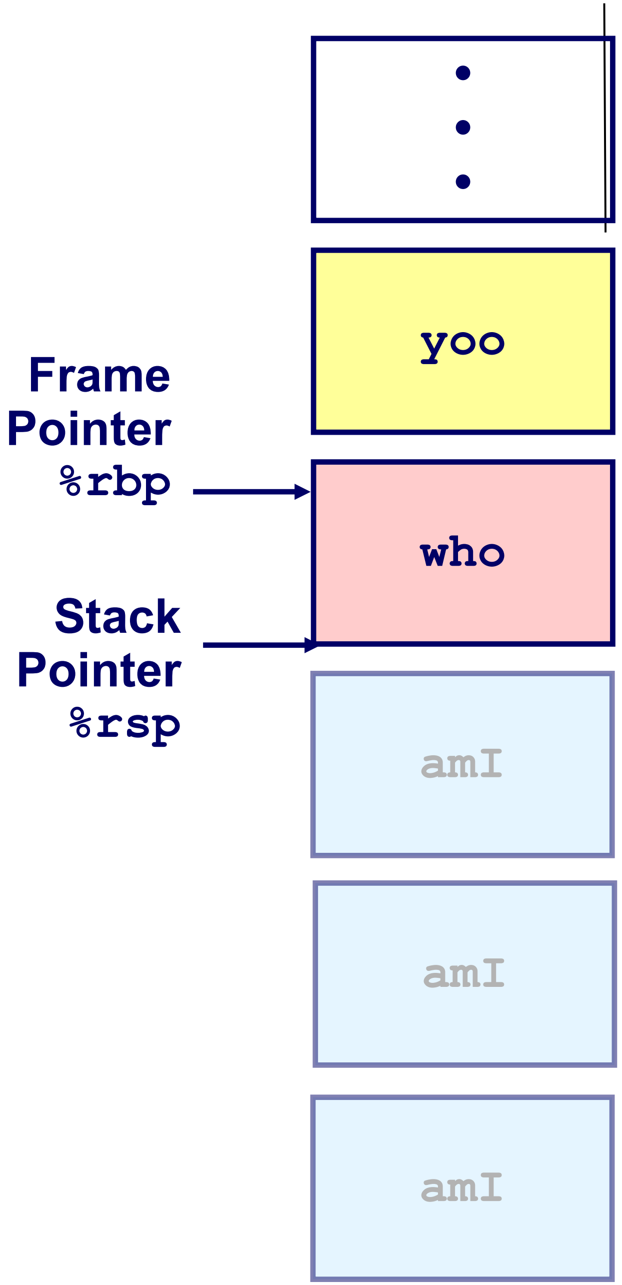


```
yoo(…)
{

}
```

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

Call Chain

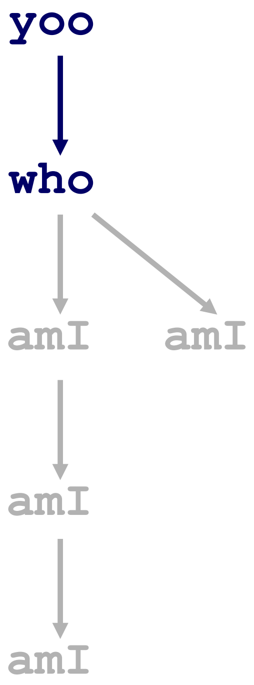


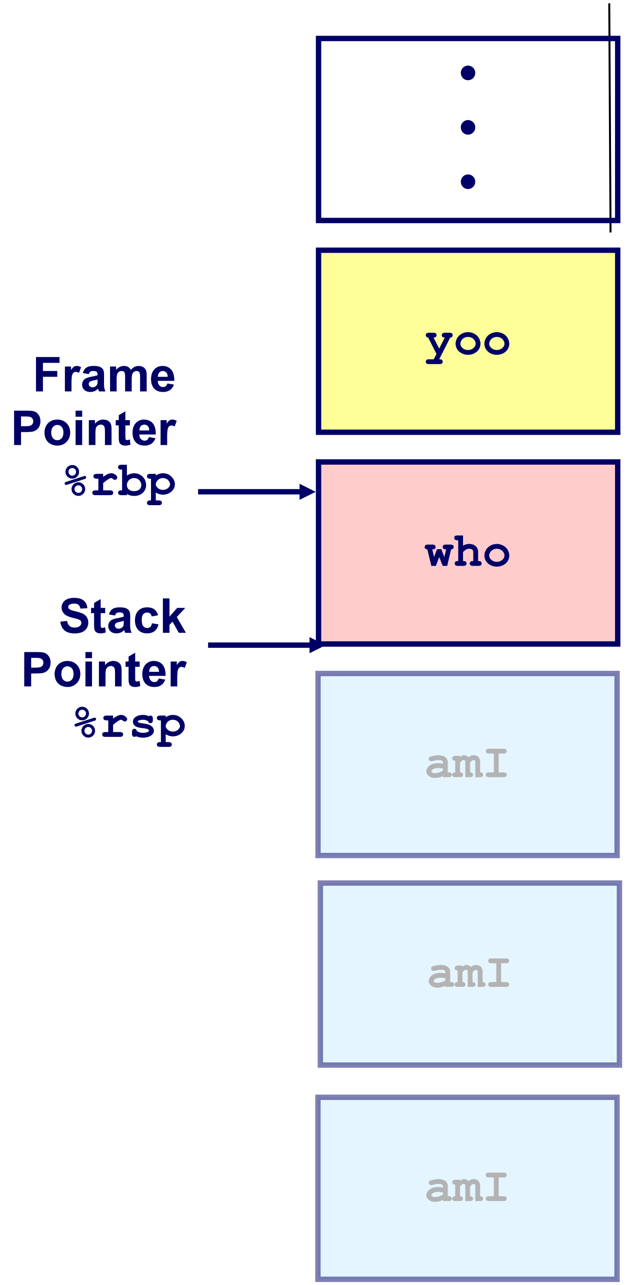


Stack Operation

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

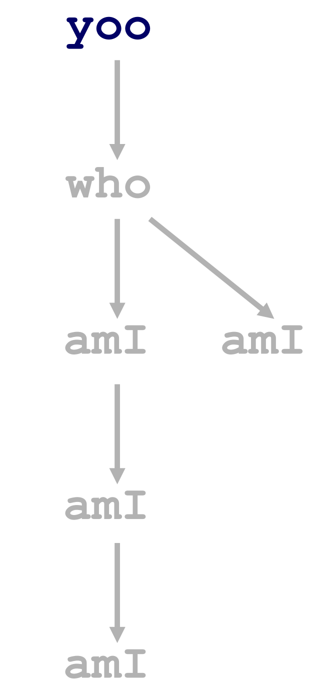
Call Chain



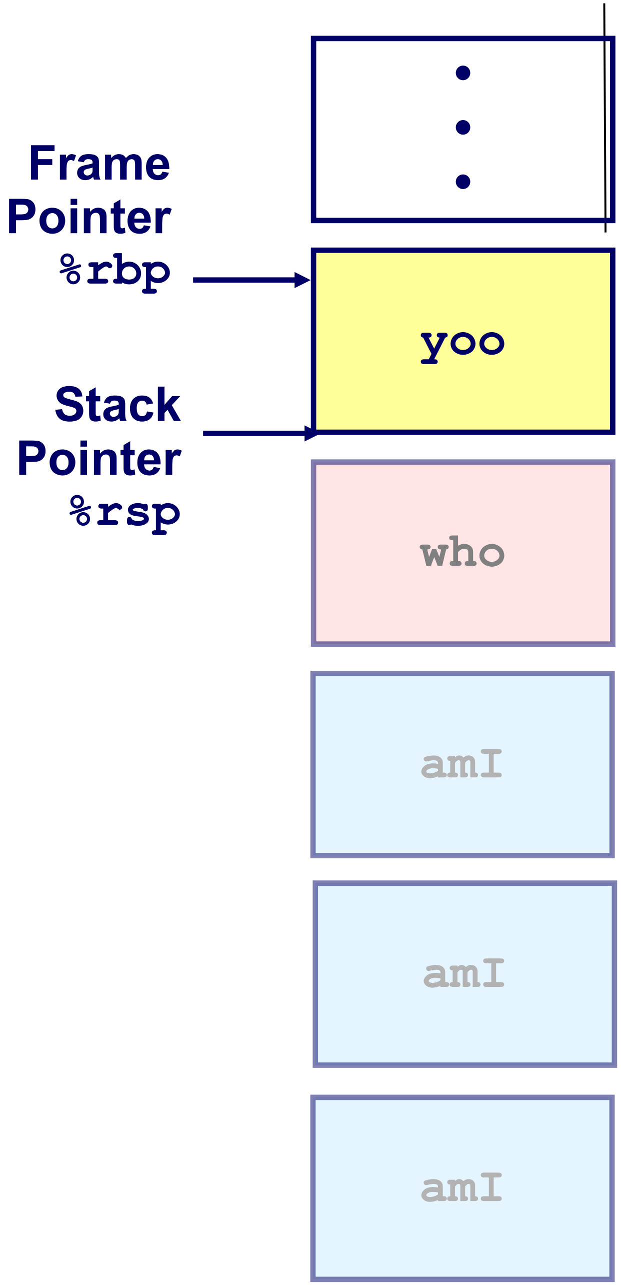


Stack Operation

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



Call Chain



Recursive Function

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Recursive Function Terminal Case

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

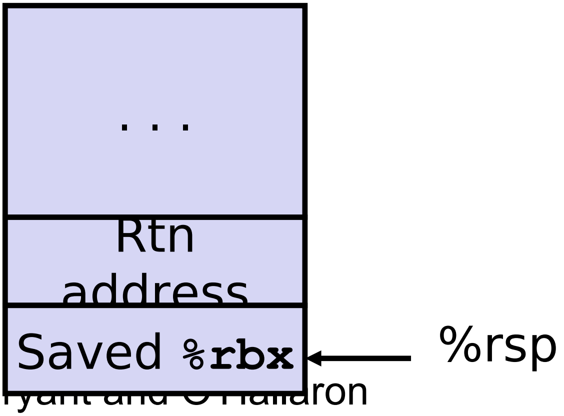
Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

Recursive Function Register Save

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
%rdi	x	Argument



Recursive Function Call Setup

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
%rdi	x >> 1	Rec. argument
%rbx	x & 1	Callee-saved

Recursive Function Call

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Recursive call return value	

Recursive Function Result

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

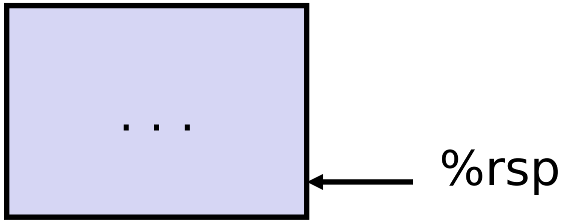
Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Return value	

Recursive Function Completion

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
`%rax`	Return value	Return value

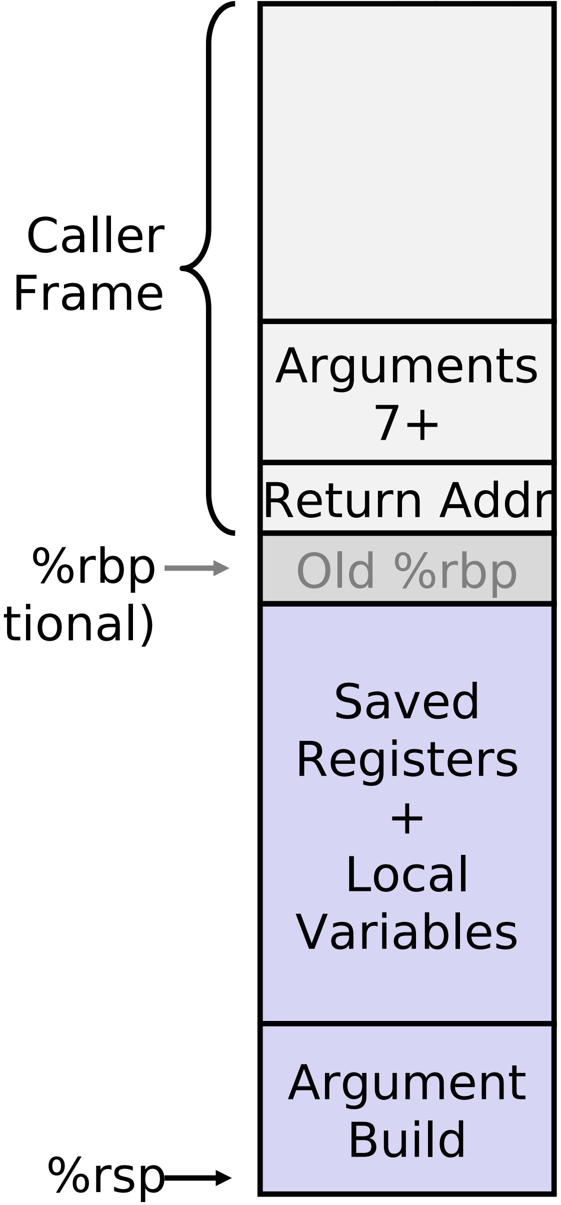


Observations About Recursion

- **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
    - Unless the C code explicitly does so (e.g., buffer overflow bug/attack)
  - 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

x86-64 Procedure Summary

- **Important Points**
  - Stack is the right data structure for procedure call / return
    - If P calls Q, then Q returns before P
- **Recursion (& mutual recursion) handled by normal calling conventions**
  - Can safely store values in local stack frame and in callee-saved registers
  - Put function arguments at top of stack
  - Result return in %rax
- Pointers are addresses of values
  - On stack or global



Optional Slides (for those interested in more examples)

Recursive Factorial

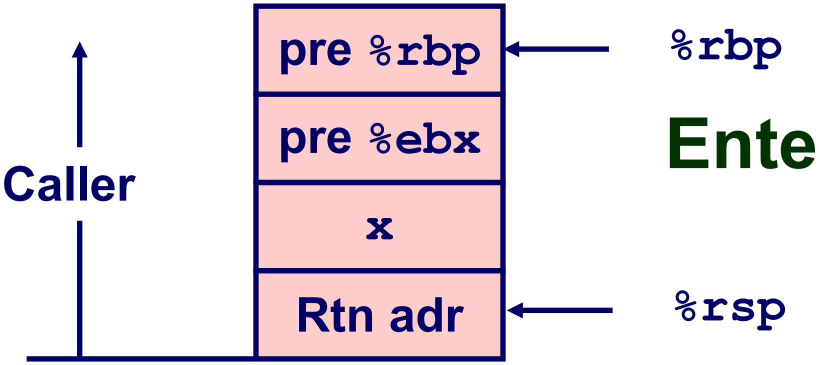
```
int rfact(int x)
{
  int rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1);
  return rval * x;
}
```

```
.globl rfact
    .type
rfact,@function
rfact:
    pushl %rbp
    movl %rsp,%rbp
    pushl %ebx
    movl 8(%rbp),%ebx
    cmpl $1,%ebx
    jle .L78
    leal -1(%ebx),%eax
    pushl %eax
    call rfact
    imull %ebx,%eax
    jmp .L79
    .align 4

.L78:
    movl $1,%eax
.L79:
    movl -4(%rbp),%ebx
    movl %rbp,%rsp
    popl %rbp
    ret
```

Registers

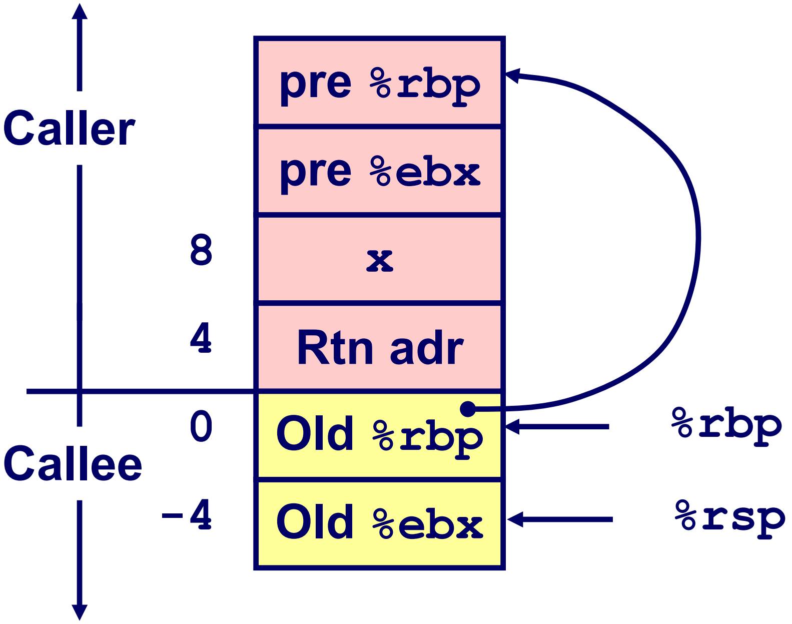
- `%eax` used without first saving
- `%ebx` used, but save at beginning & restore at end

Rfact Stack Setup



Entering Stack

```
rfact:
    pushl %rbp
    movl %rsp,%rbp
    pushl %ebx
```



Rfact Body

Recursion

```
  movl 8(%rbp),%ebx   # ebx = x
  cmpl $1,%ebx        # Compare x : 1
  jle .L78            # If <= goto Term
  leal -1(%ebx),%eax  # eax = x-1
  pushl %eax          # Push x-1
  call rfact          # rfact(x-1)
  imull %ebx,%eax     # rval * x
  jmp .L79            # Goto done
.L78:                # Term:
  movl $1,%eax        # return val = 1
.L79:                # Done:
```

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

Registers

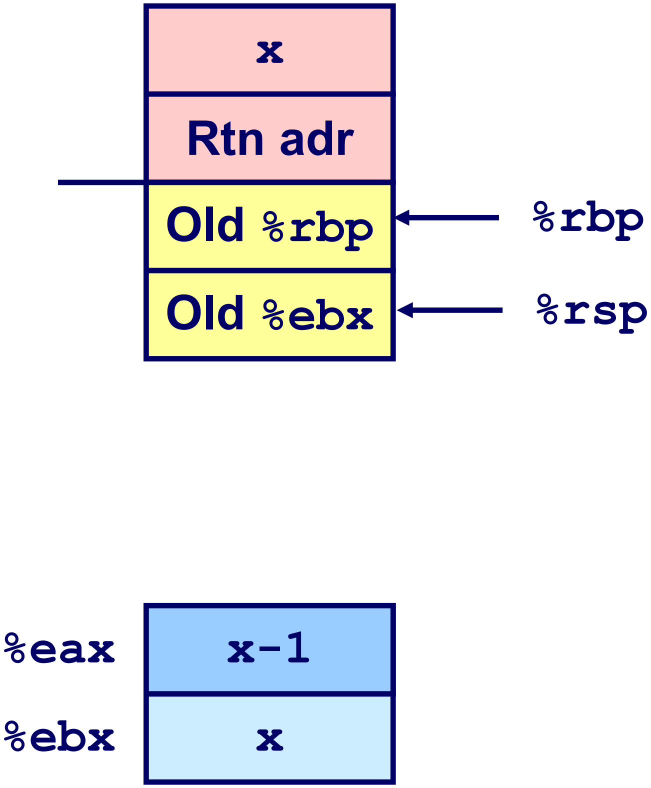
`%ebx` Stored value of x

`%eax`

- Temporary value of `x-1`
- Returned value from `rfact(x-1)`
- Returned value from this call

Rfact Recursion

```
leal -1(%ebx),%eax
```



x

Rtn adr

Old %rbp

Old %ebx

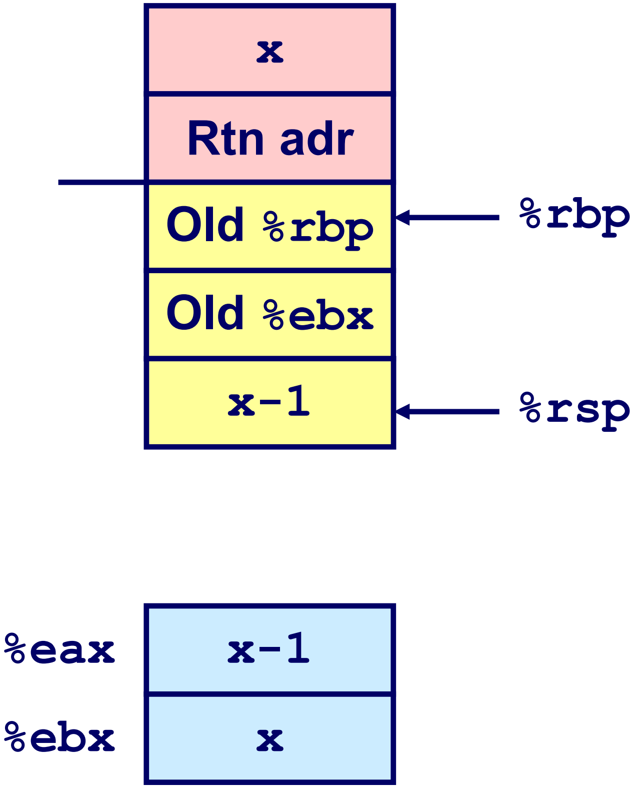
%rbp

%rsp

%eax x-1

%ebx x

```
pushl %eax
```



x

Rtn adr

Old %rbp

Old %ebx

x-1

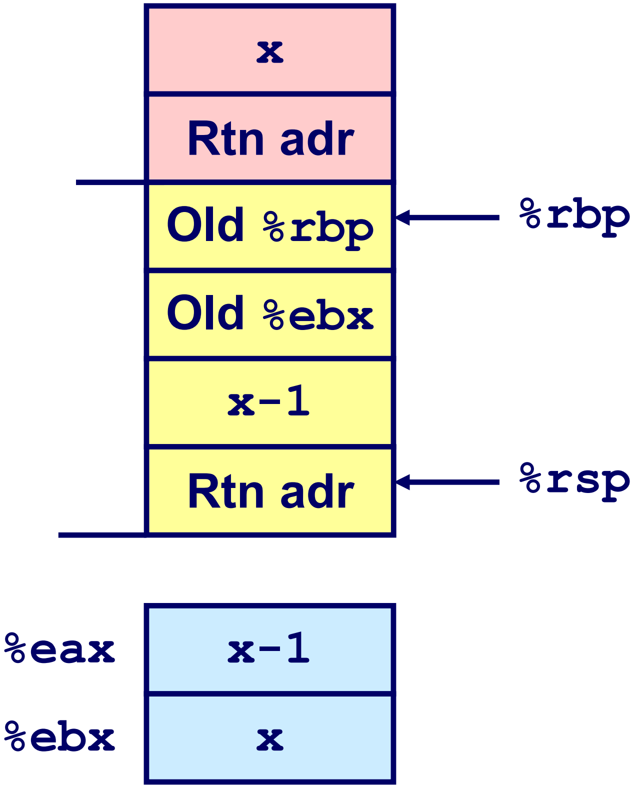
%rbp

%rsp

%eax x-1

%ebx x

```
call rfact
```



x

Rtn adr

Old %rbp

Old %ebx

x-1

Rtn adr

%rbp

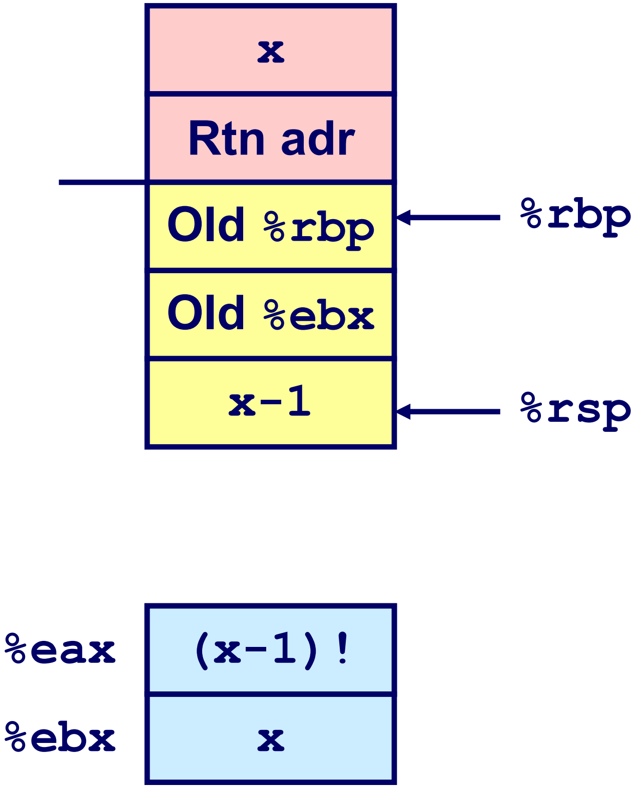
%rsp

%eax x-1

%ebx x

Rfact Result

Return from Call



x

Rtn adr

Old %rbp ← %rbp

Old %ebx

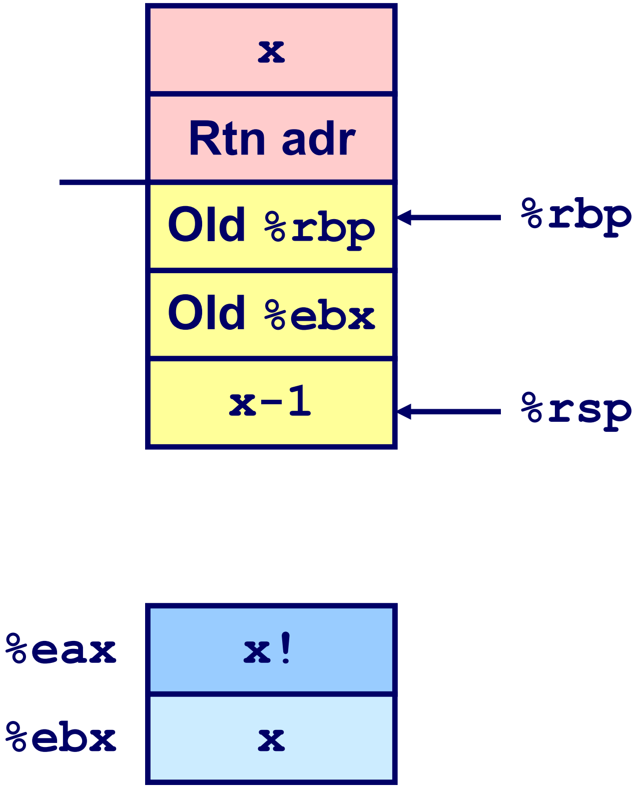
x-1 ← %rsp

%eax (x-1)!

%ebx x

Assume that rfact(x-1) returns (x-1)! in register %eax

```
imull %ebx,%eax
```



x

Rtn adr

Old %rbp ← %rbp

Old %ebx

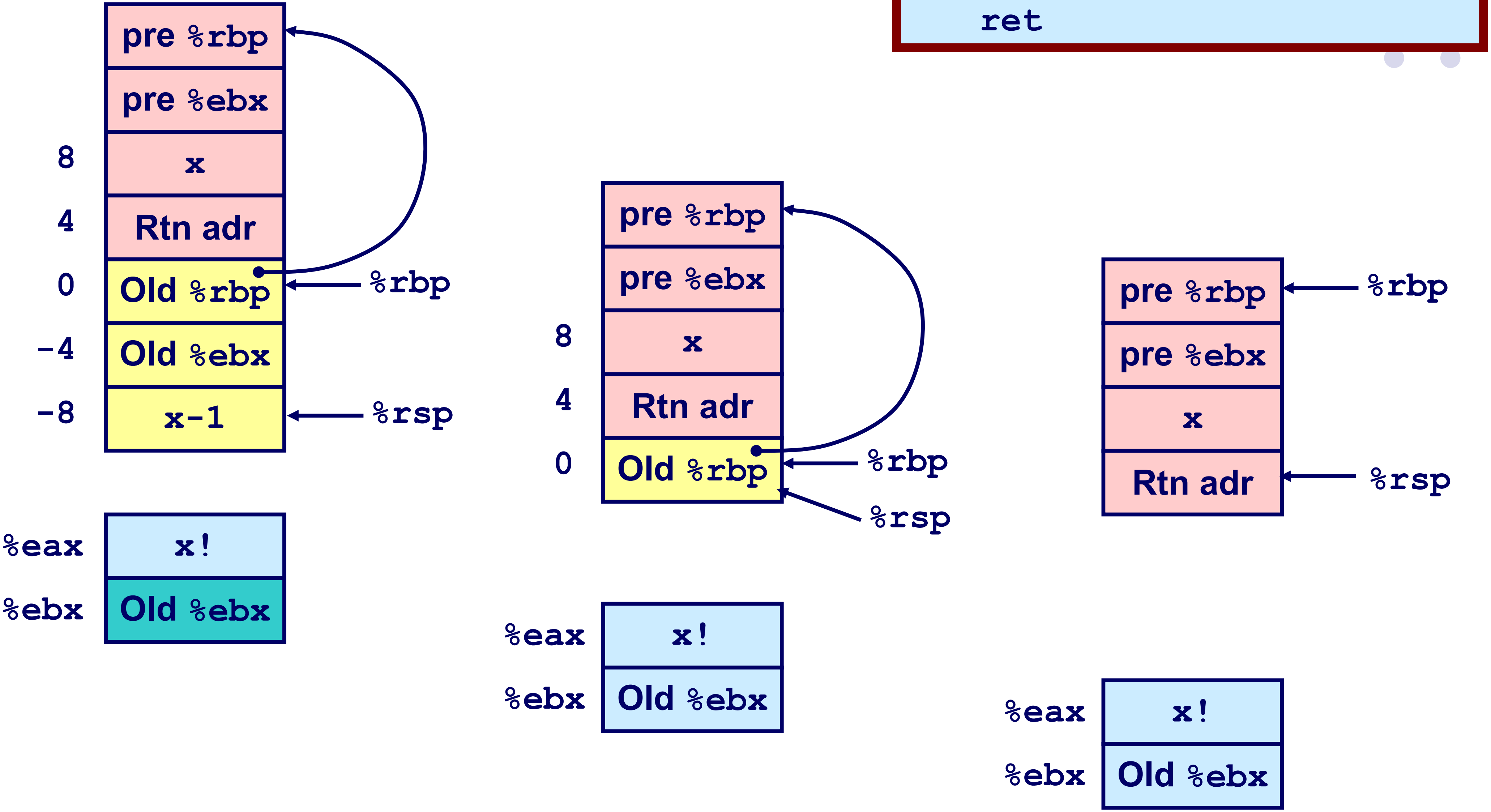
x-1 ← %rsp

%eax x!

%ebx x

Rfact Completion

```
    movl  -4(%rbp),%ebx
    movl  %rbp,%rsp
    popl  %rbp
    ret
```



Pointer Code

Recursive Procedure

```
void s_helper
  (int x, int *accum)
{
  if (x <= 1)
    return;
  else {
    int z = *accum * x;
    *accum = z;
    s_helper (x-1,accum);
  }
}
```

Top-Level Call

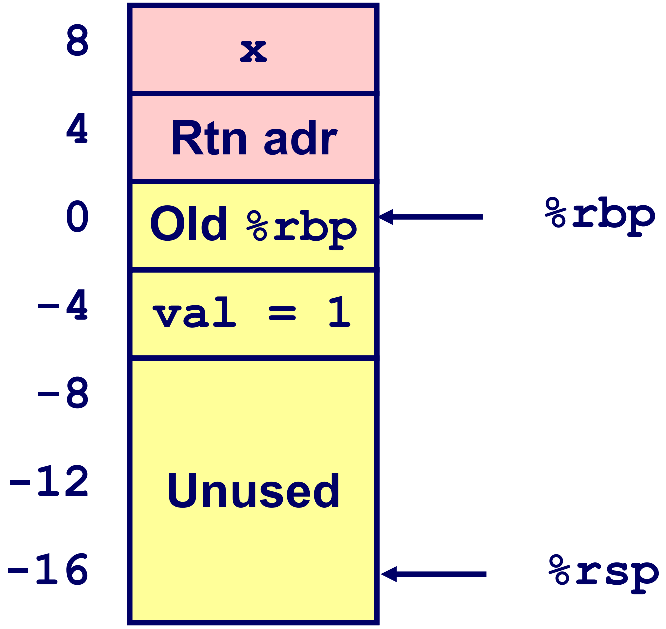
```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

Pass pointer to update location

Creating & Initializing Pointer

Initial part of `sfact`

```
_sfact:
   pushl %rbp          # Save %rbp
   movl %rsp,%rbp      # Set %rbp
   subl $16,%rsp       # Add 16 bytes
   movl 8(%rbp),%edx   # edx = x
   movl $1,-4(%rbp)    # val = 1
```



Using Stack for Local Variable

- Variable `val` must be stored on stack
  - Need to create pointer to it
- Compute pointer as `-4(%rbp)`
- Push on stack as second argument

```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

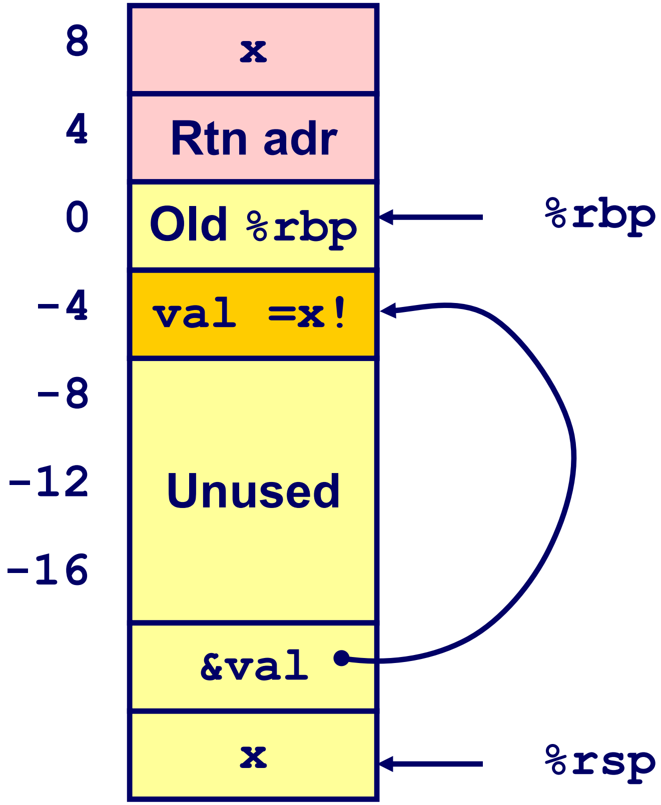
Passing Pointer

Calling s_helper from sfact

```
    leal -4(%rbp),%eax  # Compute &val
    pushl %eax          # Push on stack
    pushl %edx          # Push x
    call s_helper       # call
    movl -4(%rbp),%eax  # Return val
    • • •               # Finish
```

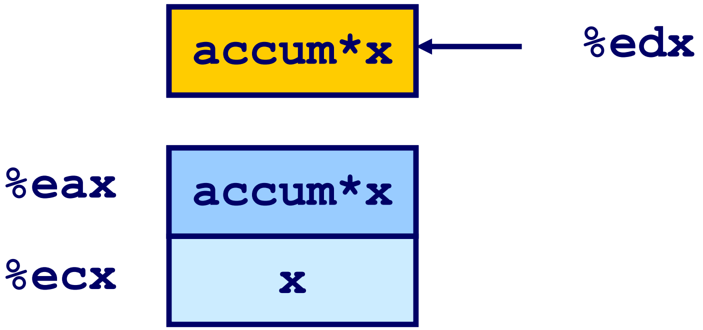
```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

Stack at time of call



Using Pointer

```
void s_helper
  (int x, int *accum)
{
  • • •
    int z = *accum * x;
    *accum = z;
  • • •
}
```



accum*x ← %edx

%eax accum*x

%ecx x

```
  • • •
  movl %ecx,%eax      # z = x
  imull (%edx),%eax # z *= *accum
  movl %eax,(%edx)   # *accum = z
  • • •
```

Register `%ecx` holds `x`

Register `%edx` holds pointer to `accum`

  Use access `(%edx)` to reference memory

Summary

The Stack Makes Recursion Work

- Private storage for each *instance* of procedure call
  - Instantiations don’t clobber each other
  - Addressing of locals+arguments relative to stack positions
- Can be managed by stack discipline
  - Procedures return in inverse order of calls

IA32 Procedures Combination of Instructions + Conventions

- Call / Ret instructions
- Register usage conventions
  - Caller / Callee save
  - `%rbp` and `%rsp`
- Stack frame organization conventions

שאלות?