Procedures and the **Call Stack**

Topics

- Procedures
- Call stack
- Procedure/stack instructions
- Calling conventions
- Register-saving conventions

Why Procedures?

Why functions? Why methods?

```
int contains_char(char* haystack, char needle) {
  while (*haystack != '\0') {
    if (*haystack == needle) return 1;
    haystack++;
  }
  return 0;
}
```

Procedural Abstraction

Implementing Procedures

How does a caller pass **arguments** to a procedure?

How does a caller get a **return value** from a procedure?

Where does a procedure store **local variables**?

How does a procedure know **where to return**
(what code to execute next when done)?

How do procedures **share limited registers** and **memory**?

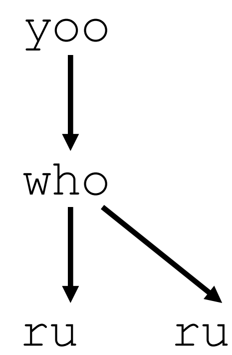
Call Chain

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

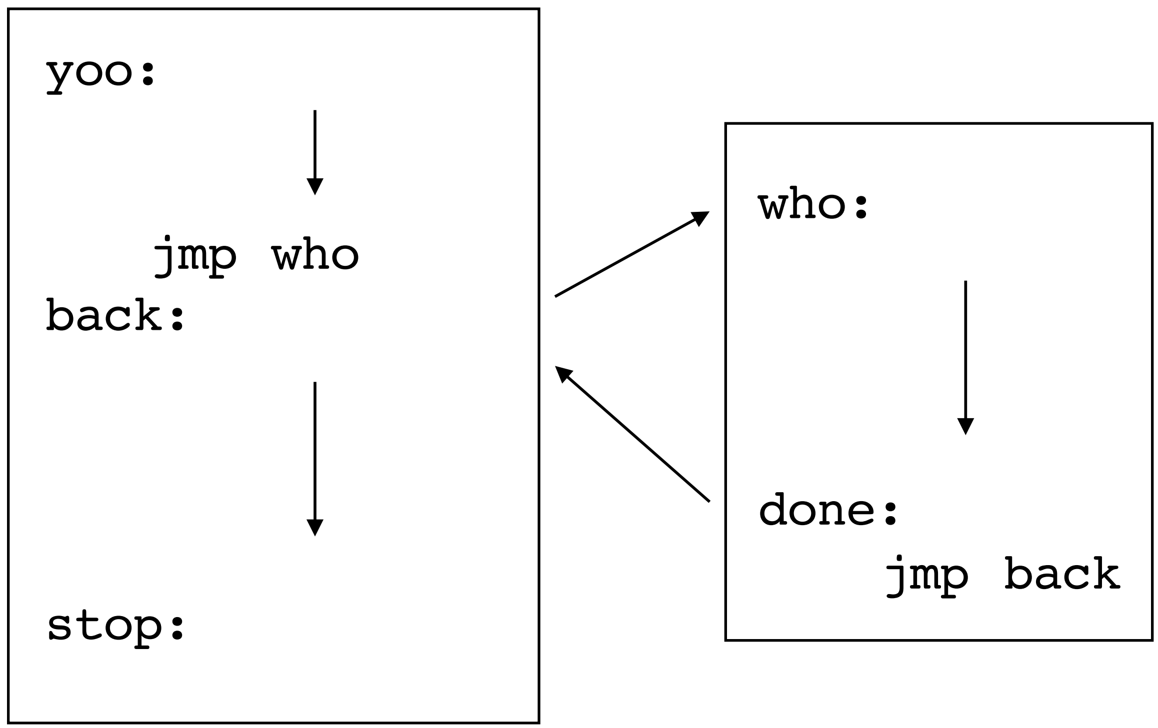
```
who(…)
{
   • • •
   ru();
   • • •
   ru();
   • • •
}
```

```
ru(…)
{
      •
      •
      •
}
```

Example Call Chain



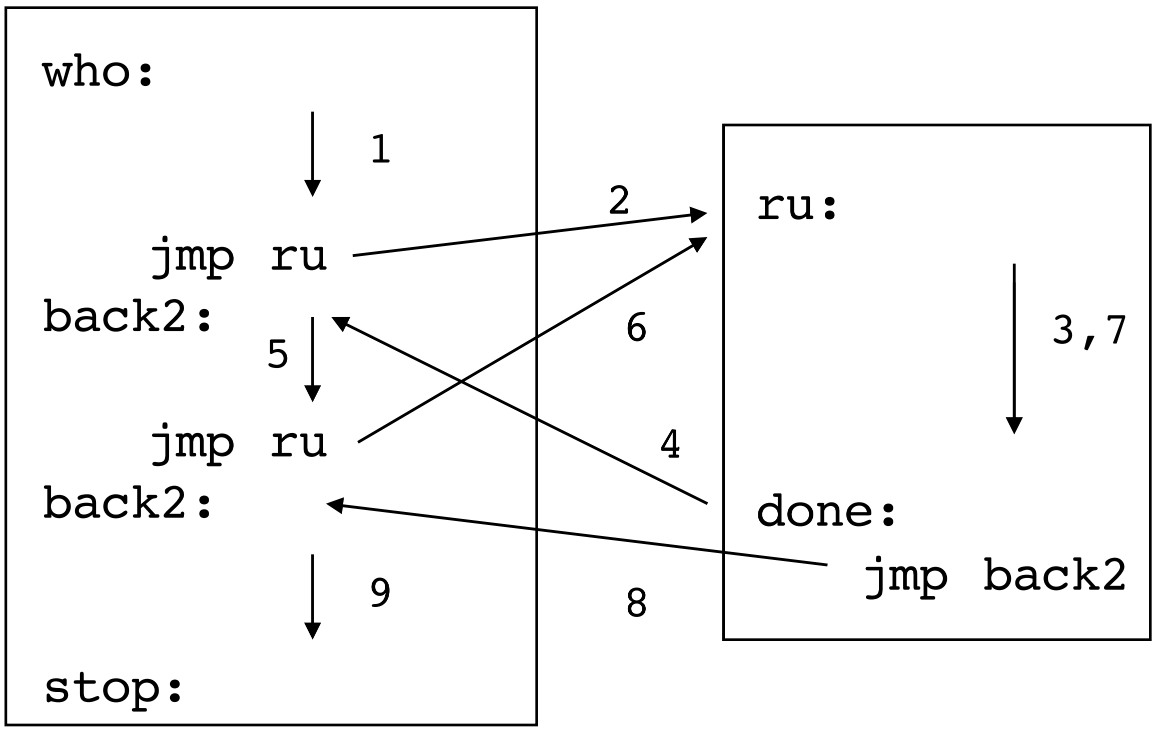
First Try (broken)



What if I want to call a function multiple times?

First Try (broken)

What if I want to call a function multiple times?



Implementing Procedures

How does a caller pass **arguments** to a procedure?

How does a caller get a **return value** from a procedure?

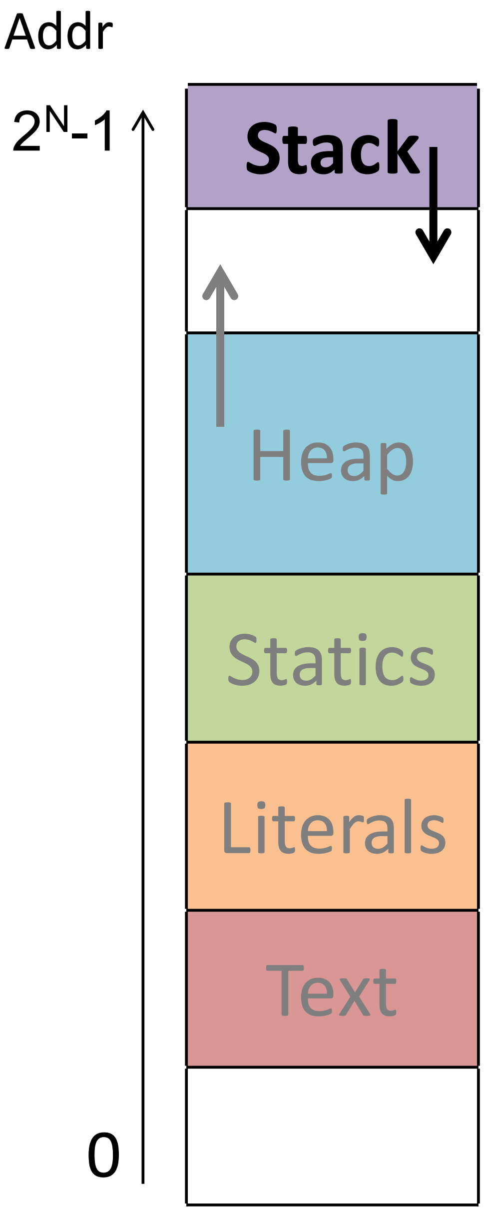
Where does a procedure store **local variables**?

How does a procedure know **where to return**
(what code to execute next when done)?

How do procedures **share limited registers** and **memory**?

All these need **separate storage *per call!***
(not just per procedure)

Memory Layout



Perm	Contents	Managed by	Initialized
RW	**Procedure context**	**Compiler**	**Run-time**
RW	Dynamic data structures	Programmer, malloc/free, new/GC	Run-time
RW	Global variables/ static data structures	Compiler/ Assembler/Linker	Startup
R	String literals	Compiler/ Assembler/Linker	Startup
X	Instructions	Compiler/ Assembler/Linker	Startup

Call Stack

We see x86 organization.

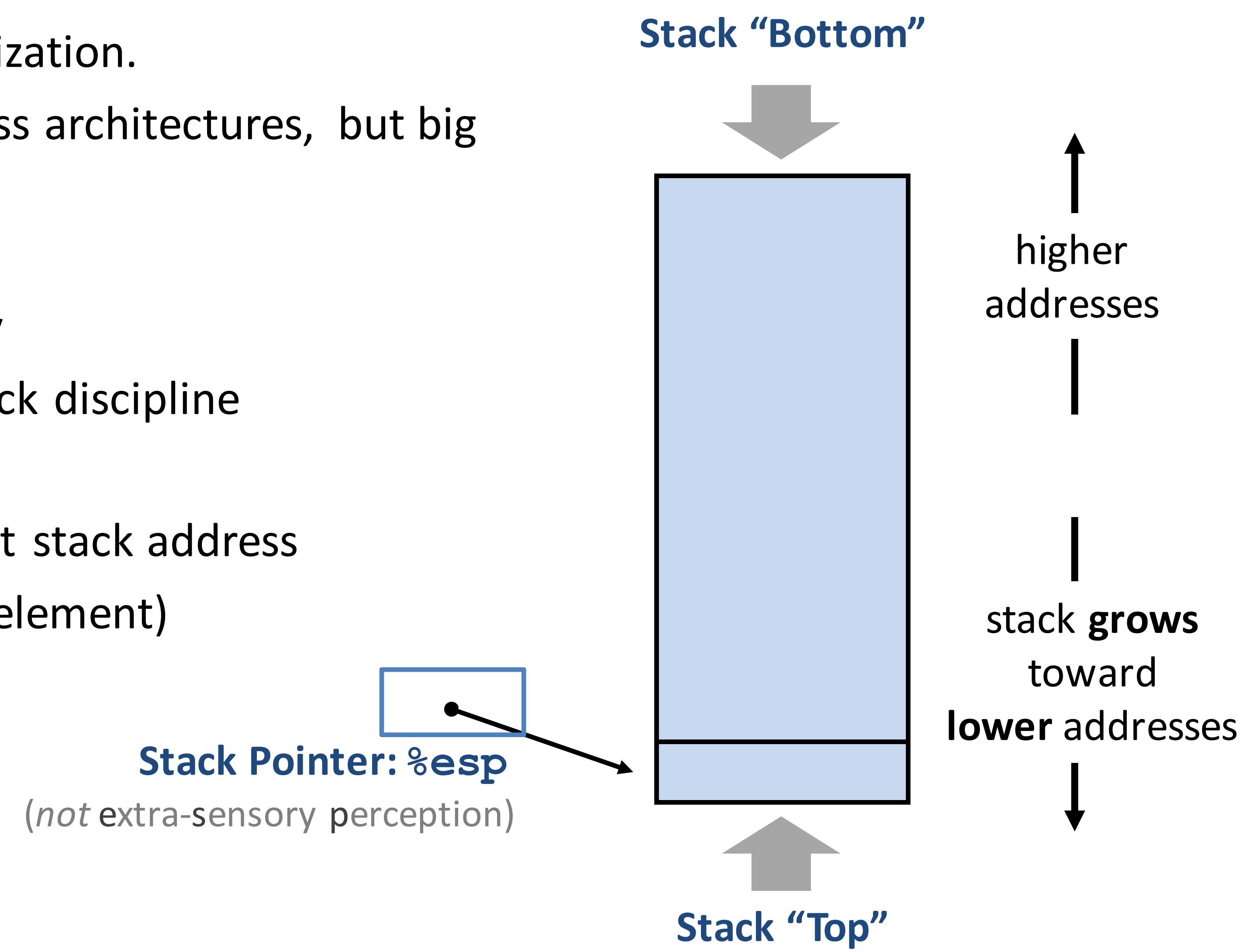
Details differ across architectures, but big ideas are shared.

Region of memory

Managed with stack discipline

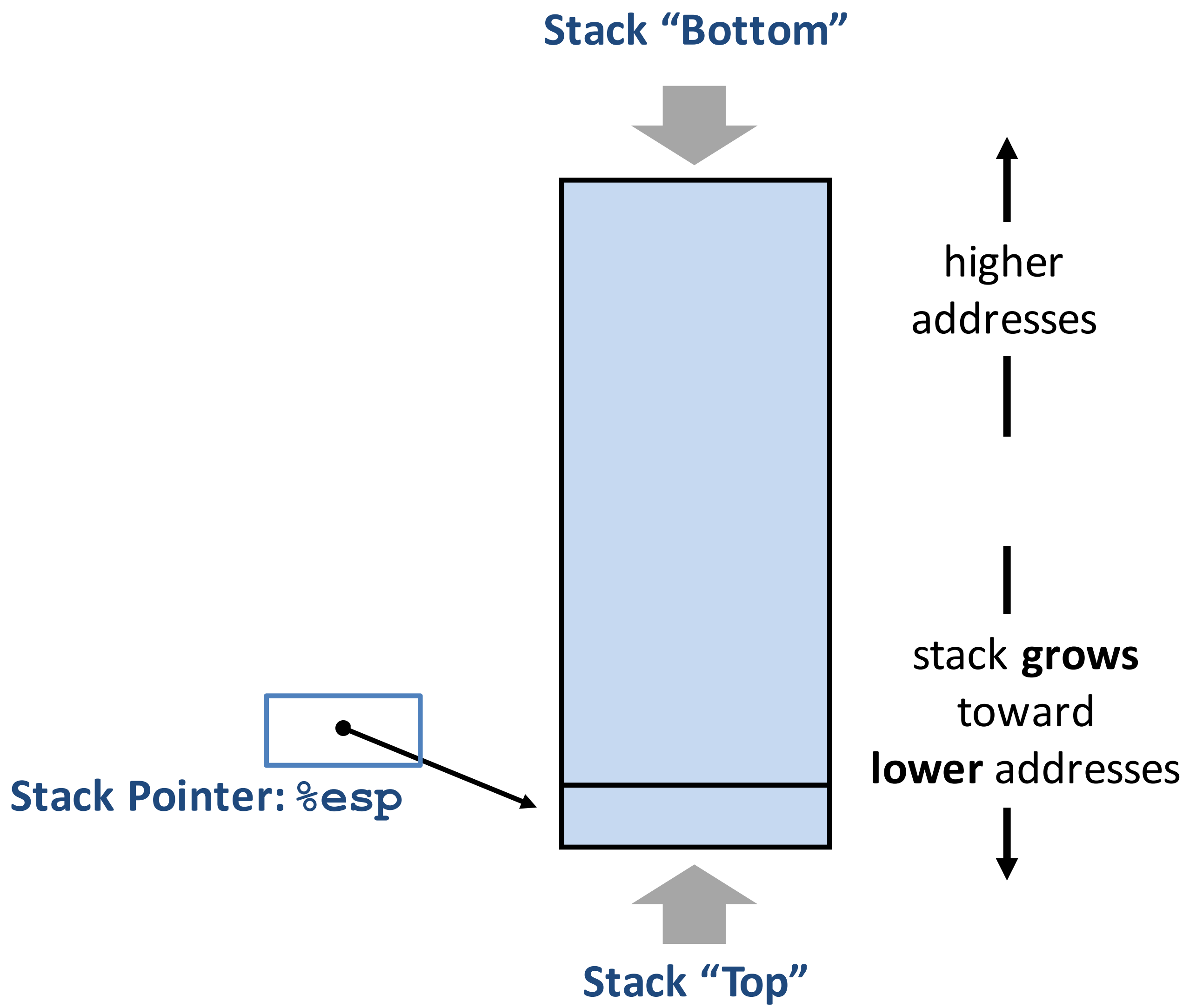
`%esp` holds lowest stack address

(address of "top" element)



IA32 Call Stack: **Push**

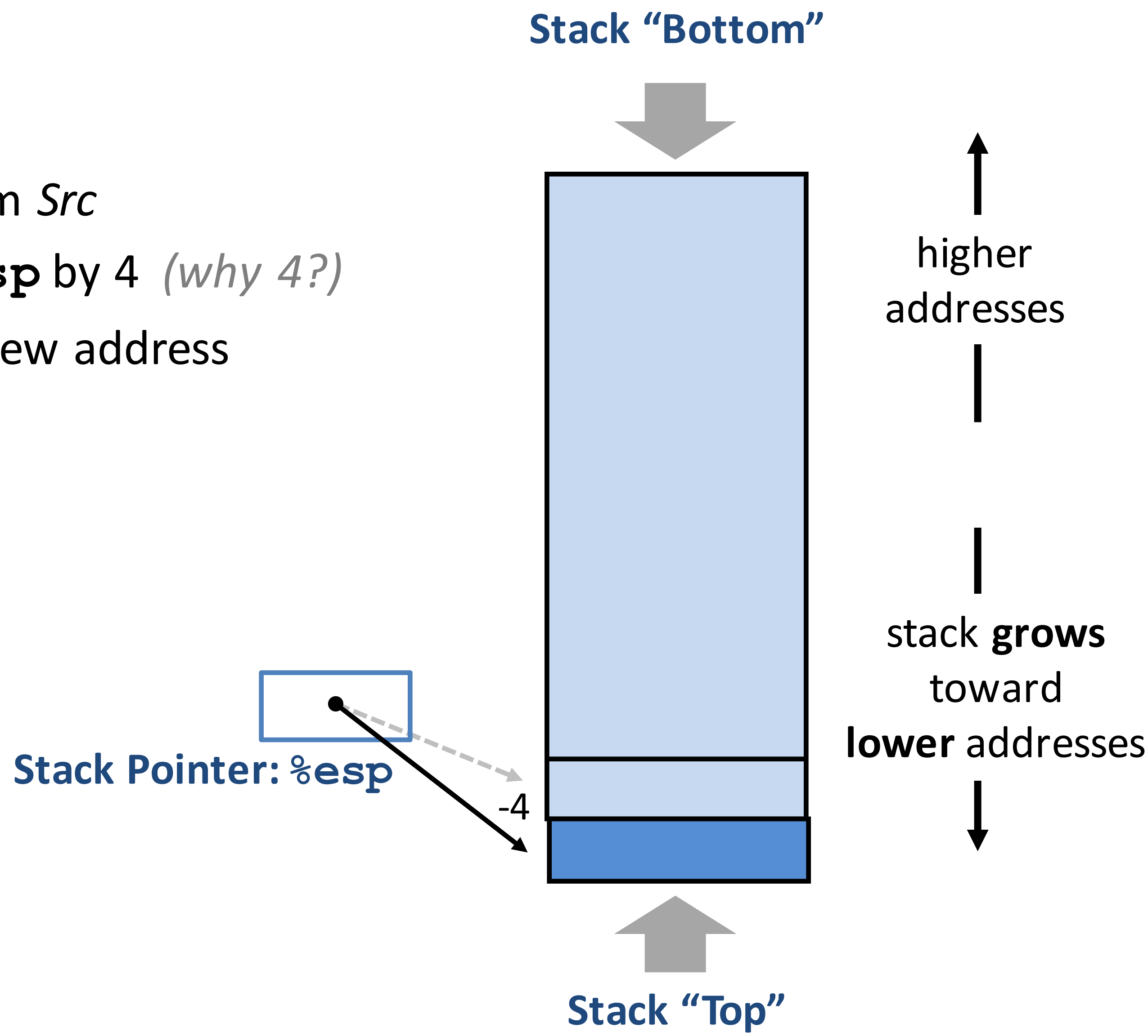
`pushl` *Src*



IA32 Call Stack: **Push**

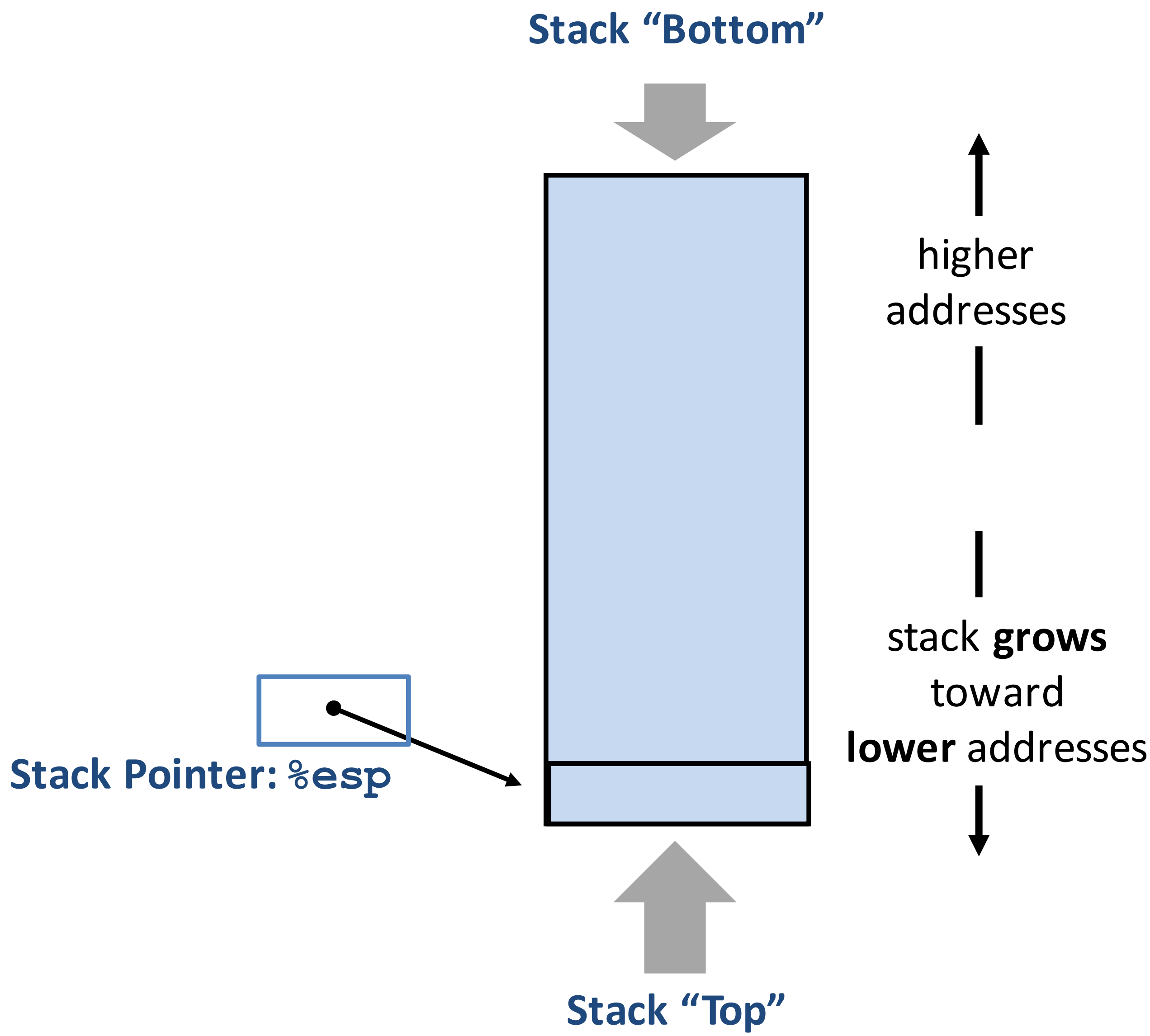
`pushl` ***Src***

1. Fetch value from *Src*
2. Decrement **`%esp`** by 4 *(why 4?)*
3. Store value at new address given by **`%esp`**



IA32 Call Stack: **Pop**

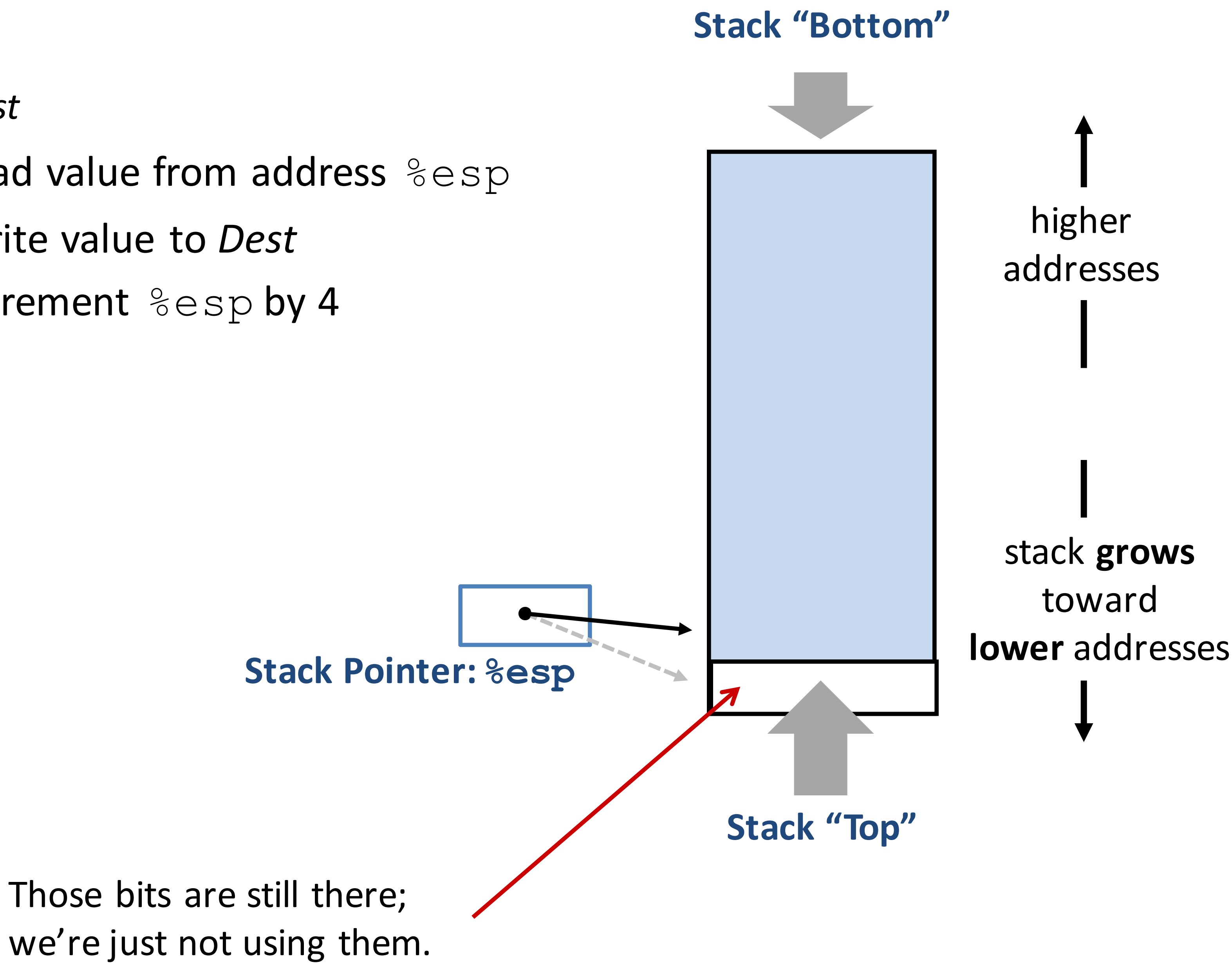
`popl` *Dest*



IA32 Call Stack: **Pop**

`popl` *Dest*

1. Load value from address `%esp`
2. Write value to *Dest*
3. Increment `%esp` by 4



Call Chain Example

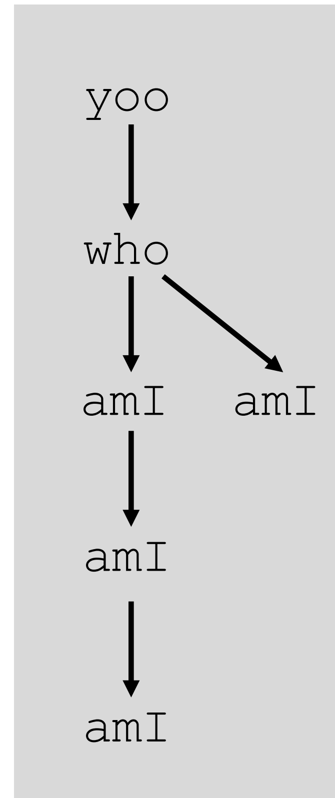
```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

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

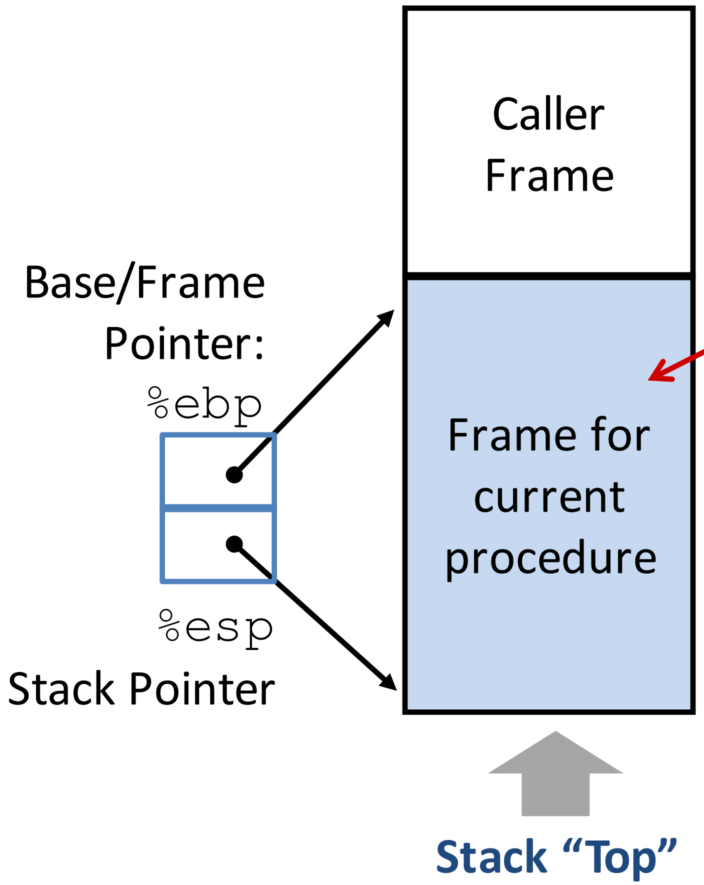
```
amI(…)
{
      •
      •
      amI();
      •
      •
}
```

Procedure amI is recursive (calls itself)

Example Call Chain



Stack frames support procedure calls.



Contents

- Local variables
- Function arguments
- Return information
- Temporary space

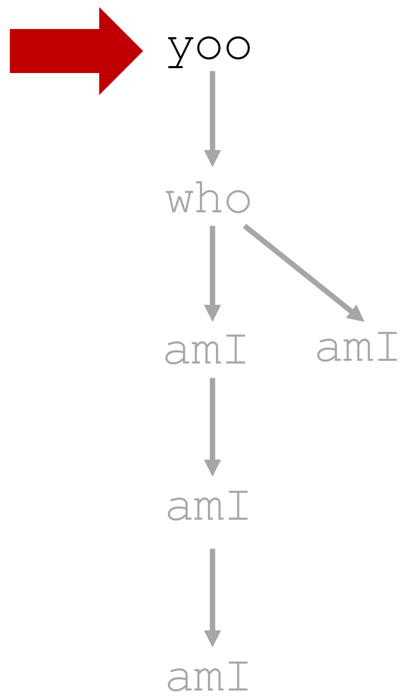
Management

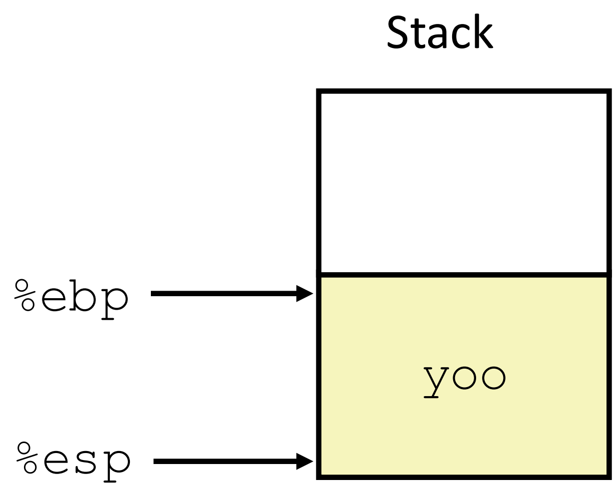
- Space allocated when procedure is entered
  - "Set-up" code
- Space deallocated upon return
  - "Finish" code

Why not just give every *procedure* a permanent chunk of memory to hold its local variables, etc?

Example

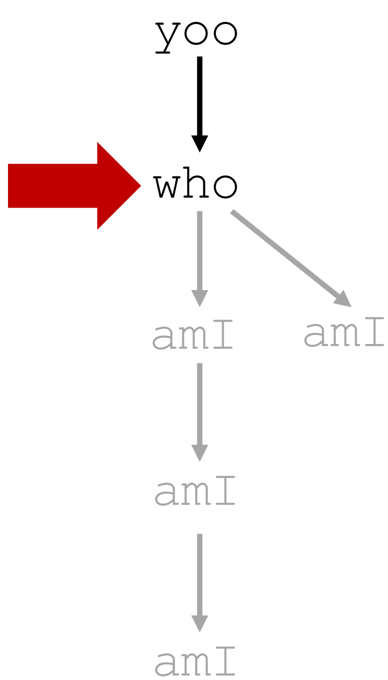
```
yoo(…)
{
    •
    •
    who();
    •
    •
}
```

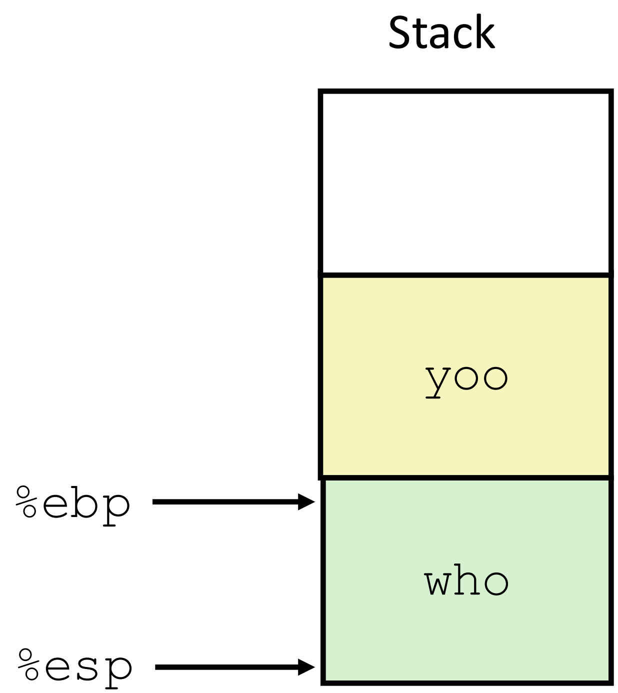




Example

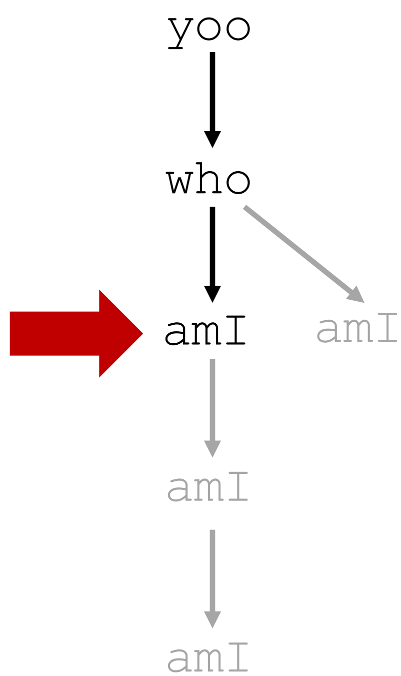
```
who(…)
{
    • • •
    amI();
    • • •
    amI();
    • • •
}
```

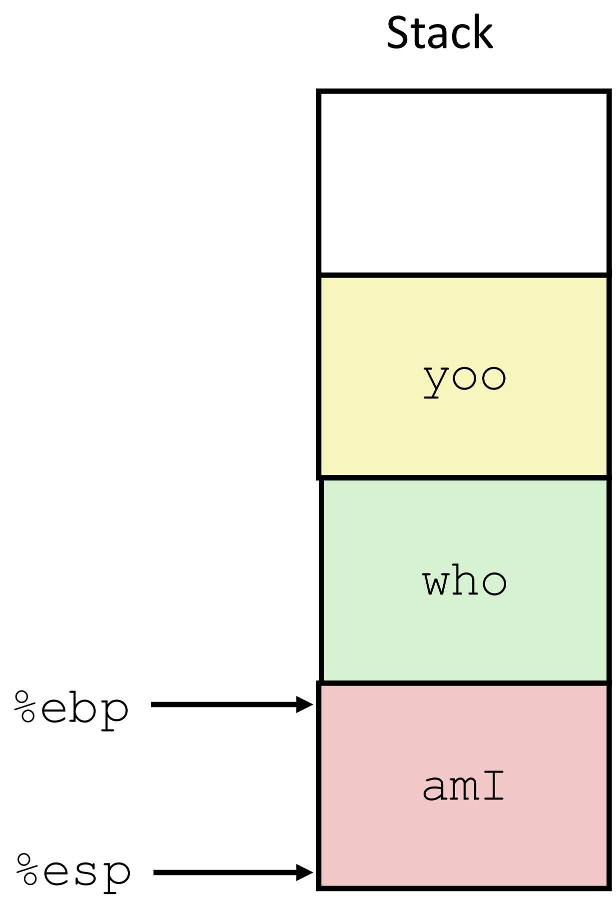




Example

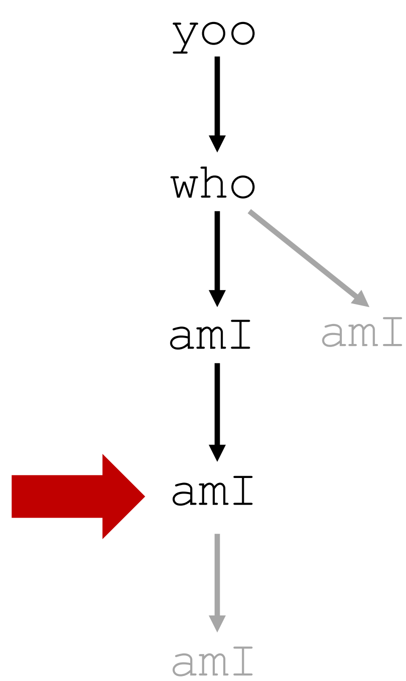
```
amI(…)
{
    •
    •
    amI();
    •
    •
}
```

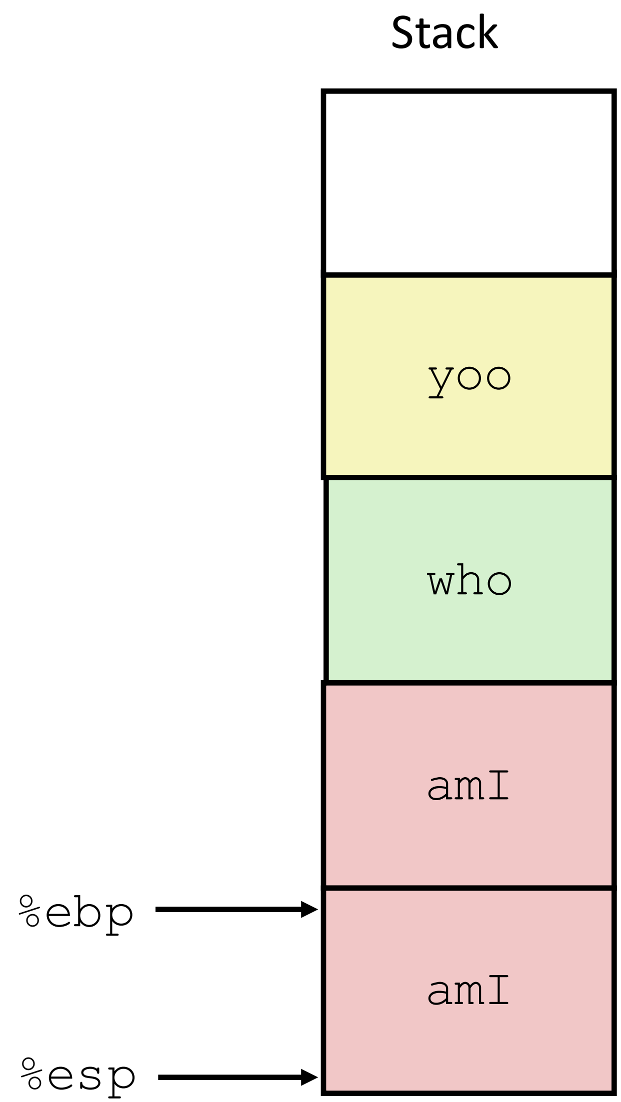




Example

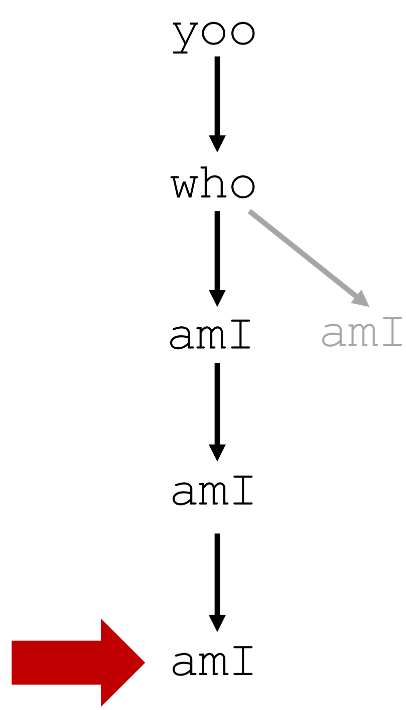
```
amI(…)
{
    •
    •
    amI();
    •
    •
}
```

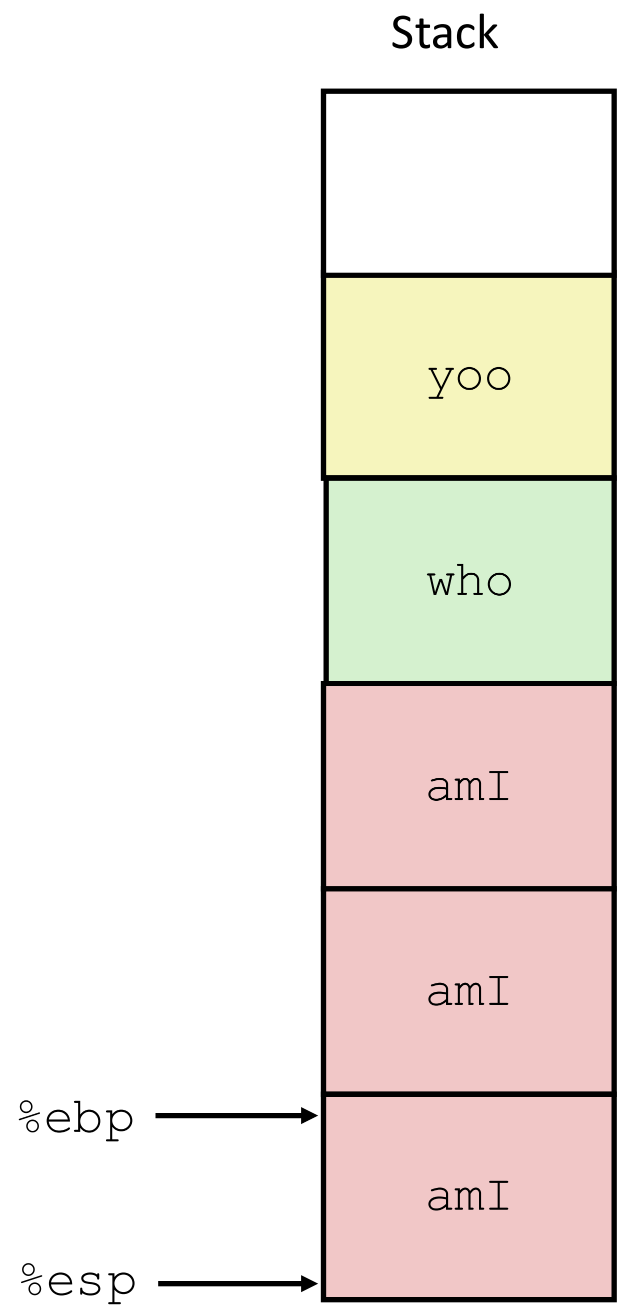




Example

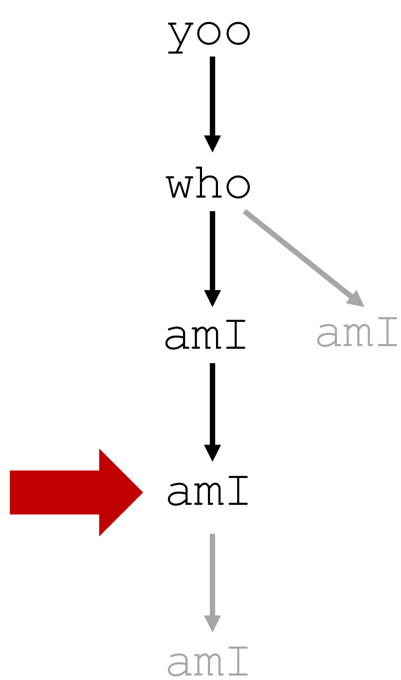
```
amI(…)
{
    •
    •
    amI();
    •
    •
}
```

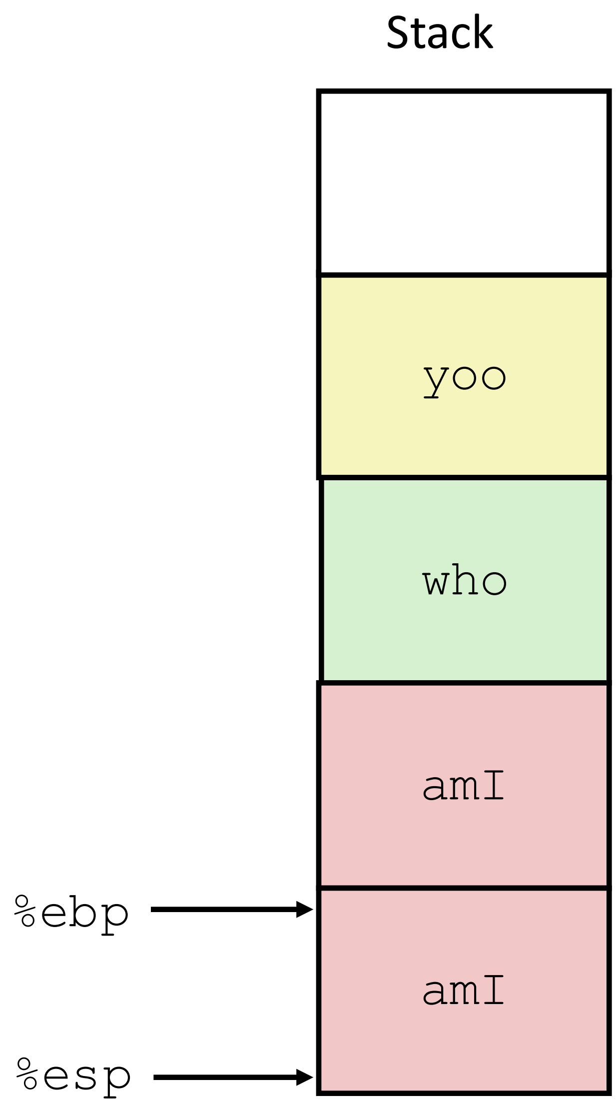




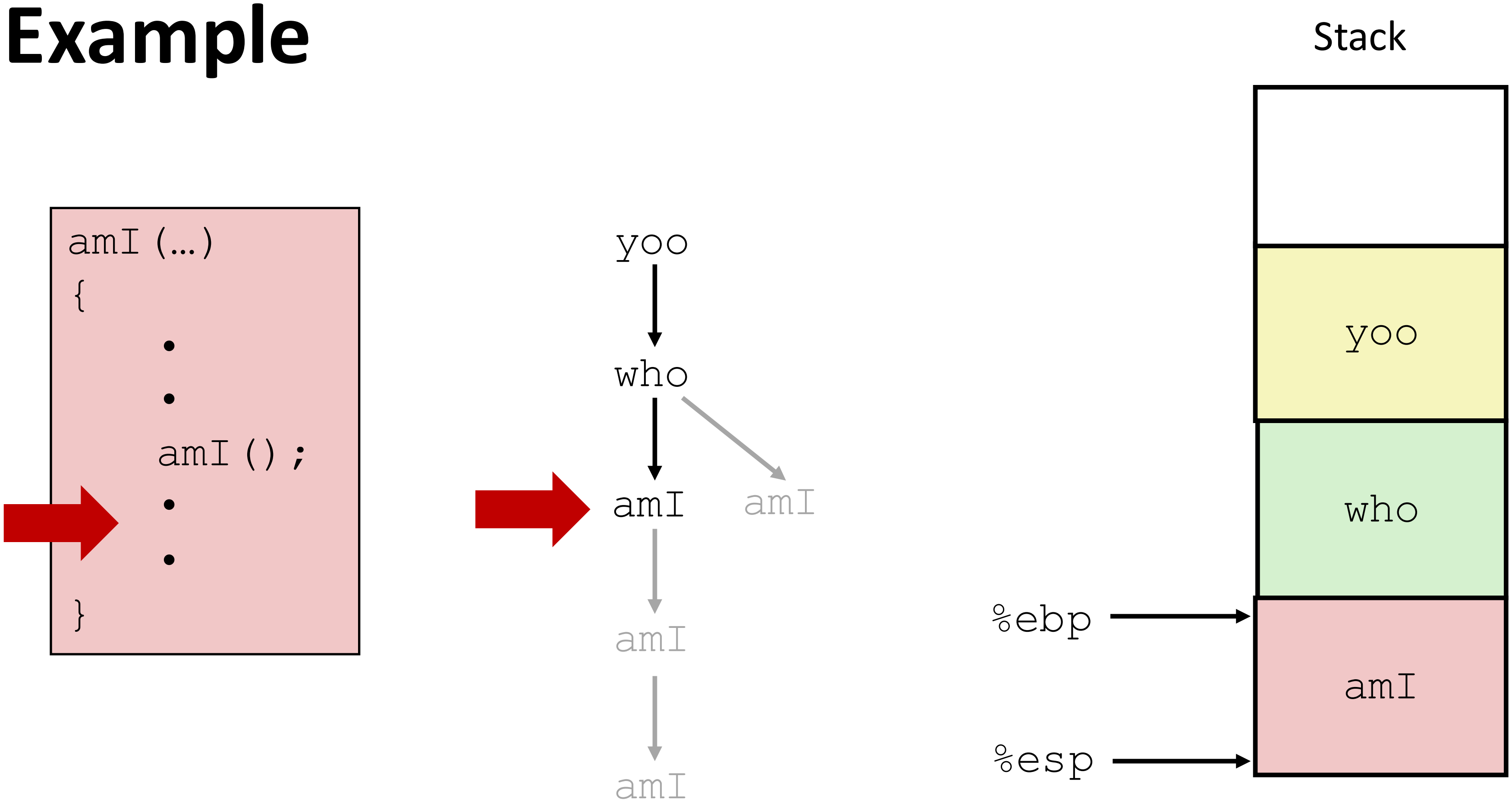
Example

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



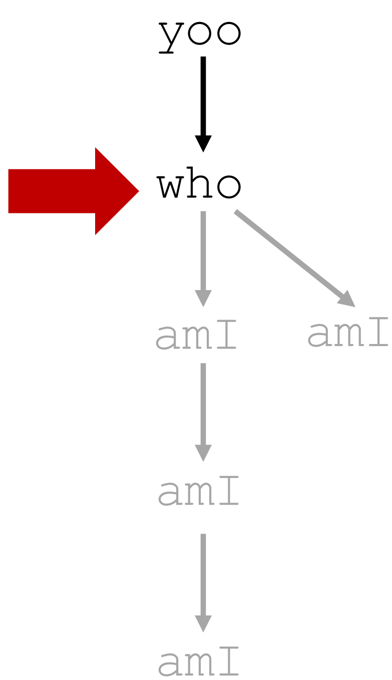


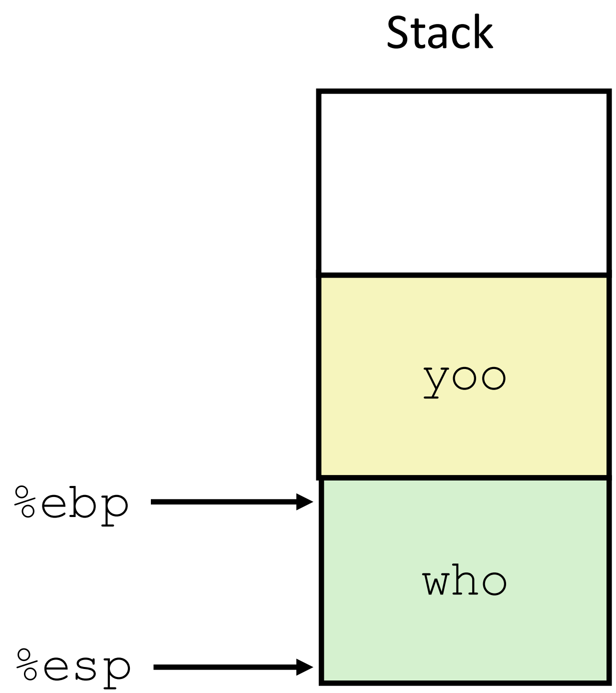
Example



Example

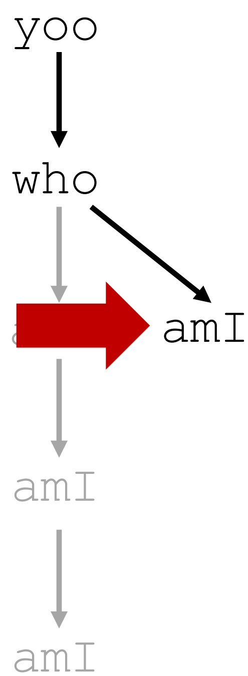
```
who(…)
{
    • • •
    amI();
    • • •
    amI();
    • • •
}
```

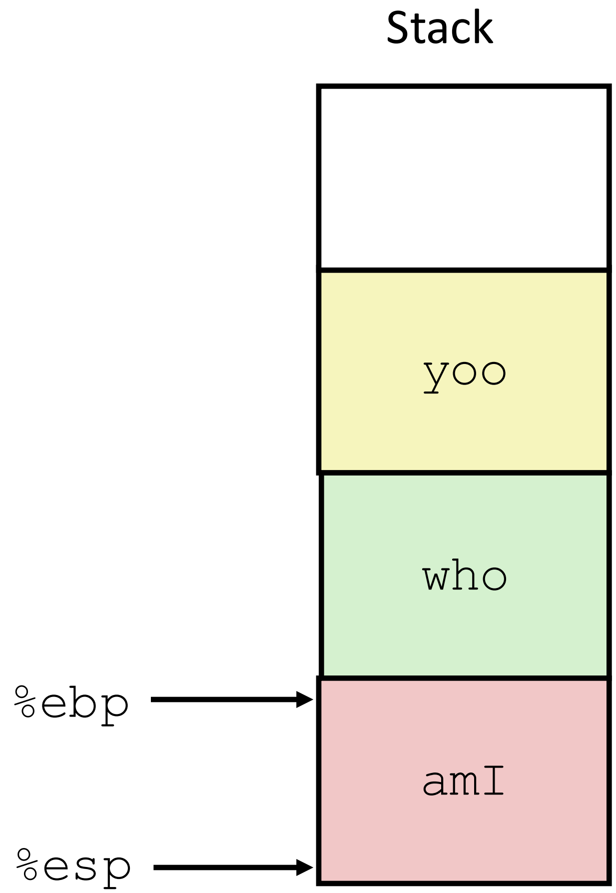




Example

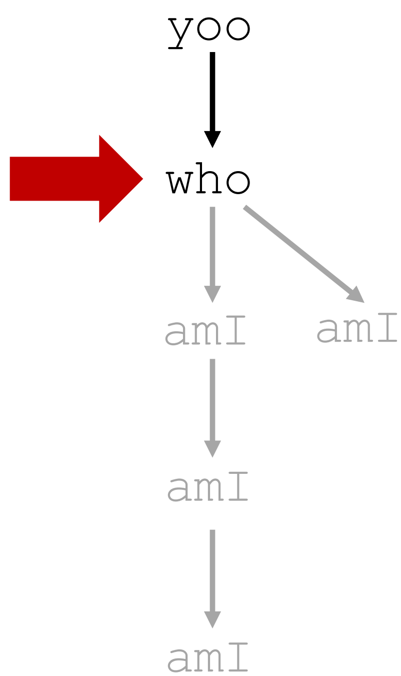
```
amI(…)
{
      •
      •
      •
      •
      •
}
```

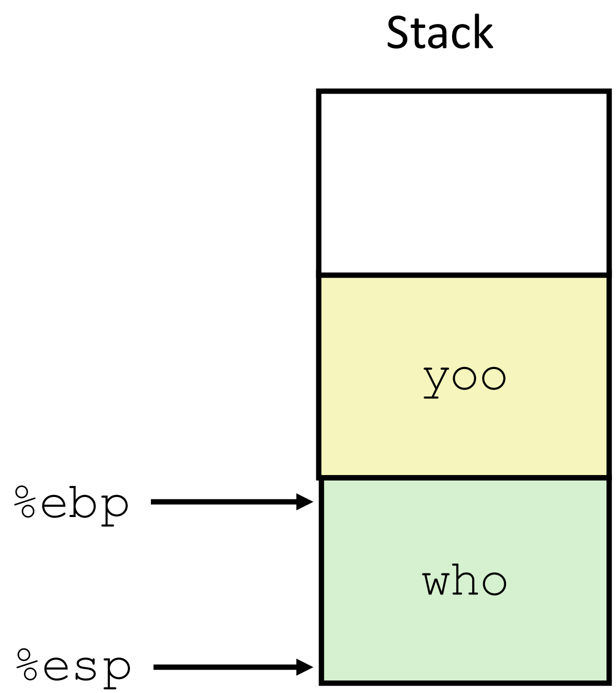




Example

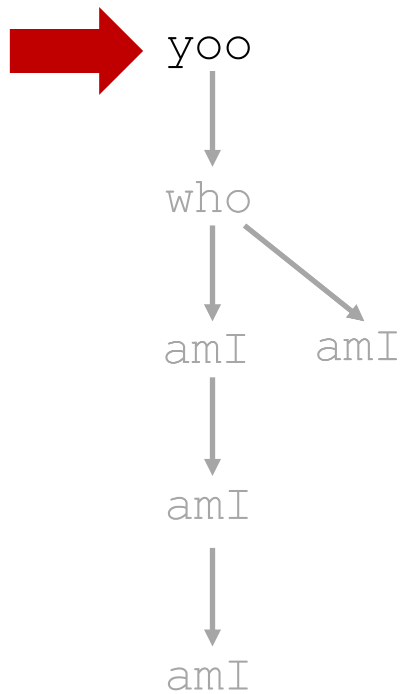
```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

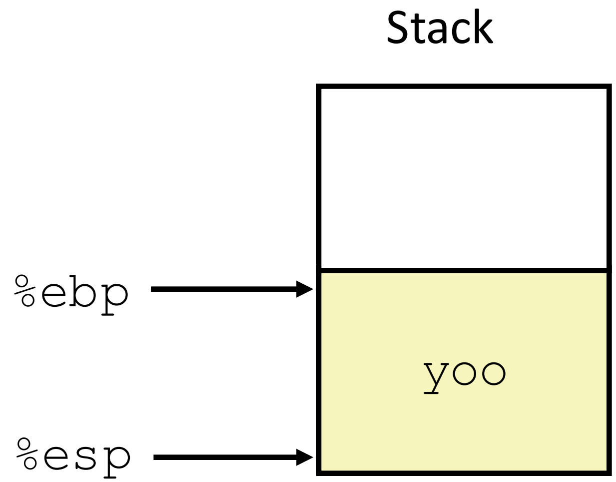




Example

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





How did we remember where to point %ebp when returning?

Procedure Control Flow Instructions

Procedure call: `call label`

1. Push return address on stack
2. Jump to *`label`*

Return address: Address of instruction after `call`. Example:

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

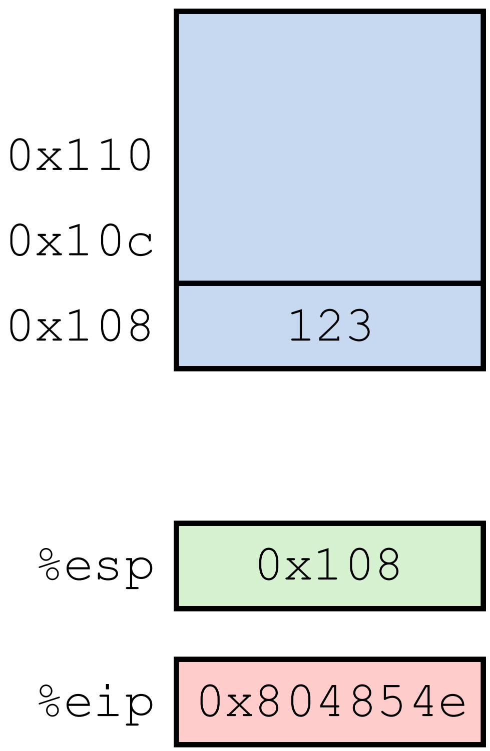
Procedure return: `ret`

1. Pop return address from stack
2. Jump to address

Procedure Call/Return: 1

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

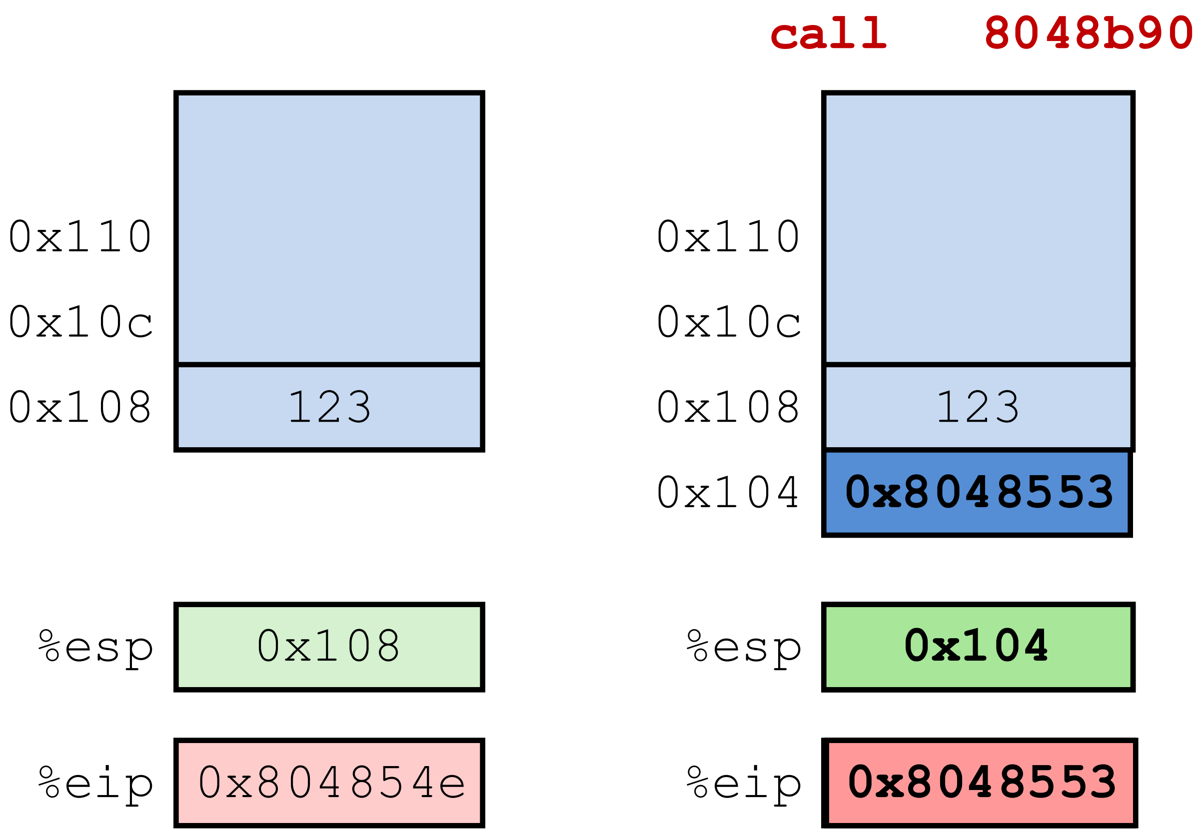
`call    8048b90`



`%eip` *= instruction pointer = program counter*

Procedure Call/Return: 2

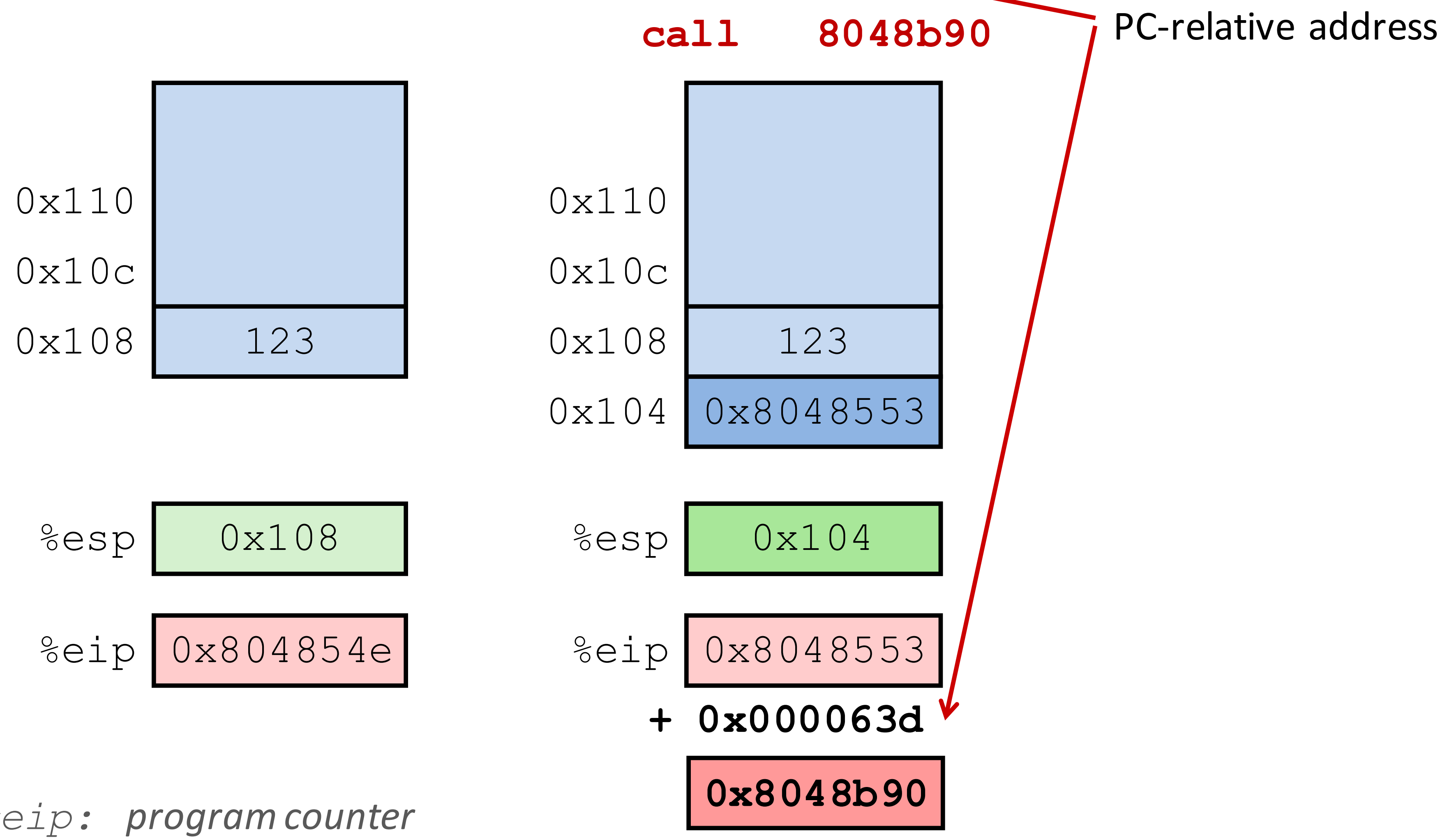
```
804854e:  e8 3d 06 00 00    call    8048b90 <main>
8048553:  50                pushl   %eax
```



`%eip` *= instruction pointer = program counter*

Procedure Call/Return: 3

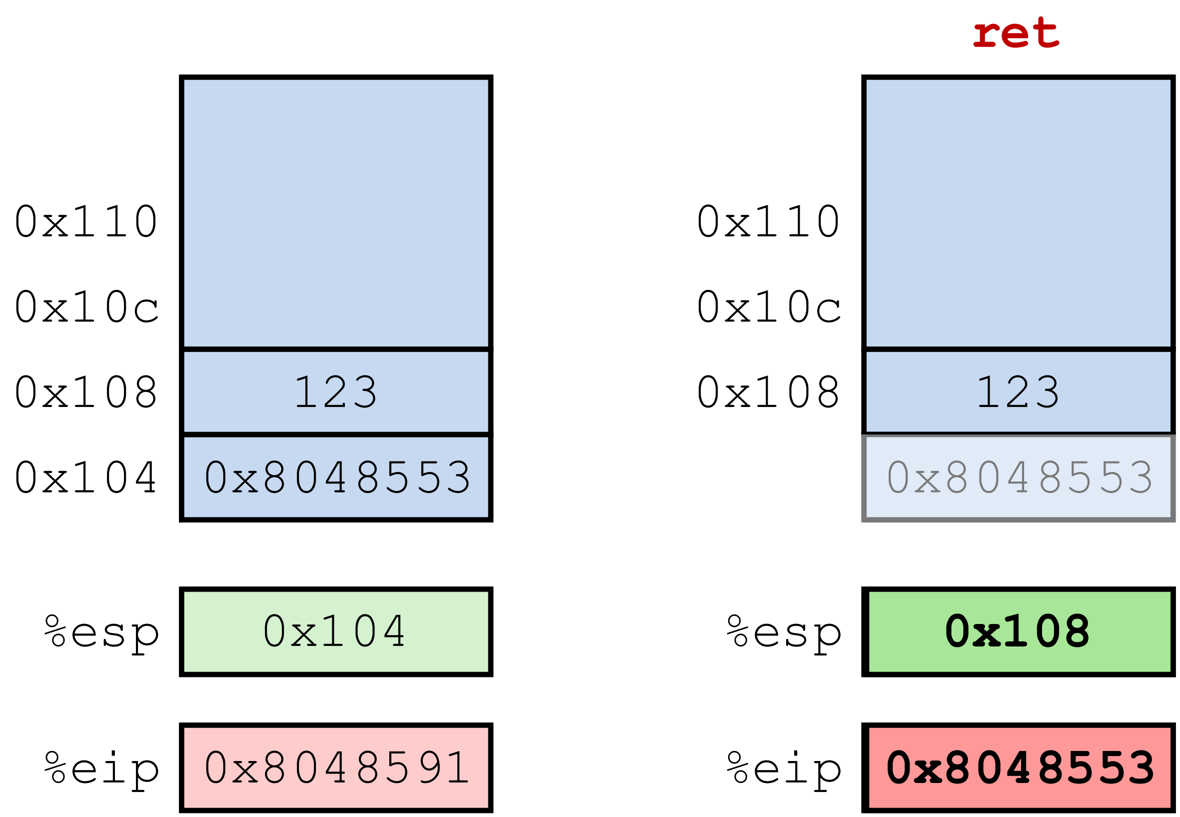
```
804854e:  e8 3d 06 00 00     call    8048b90 <main>
8048553:  50                 pushl   %eax
```



`%eip`: *program counter*

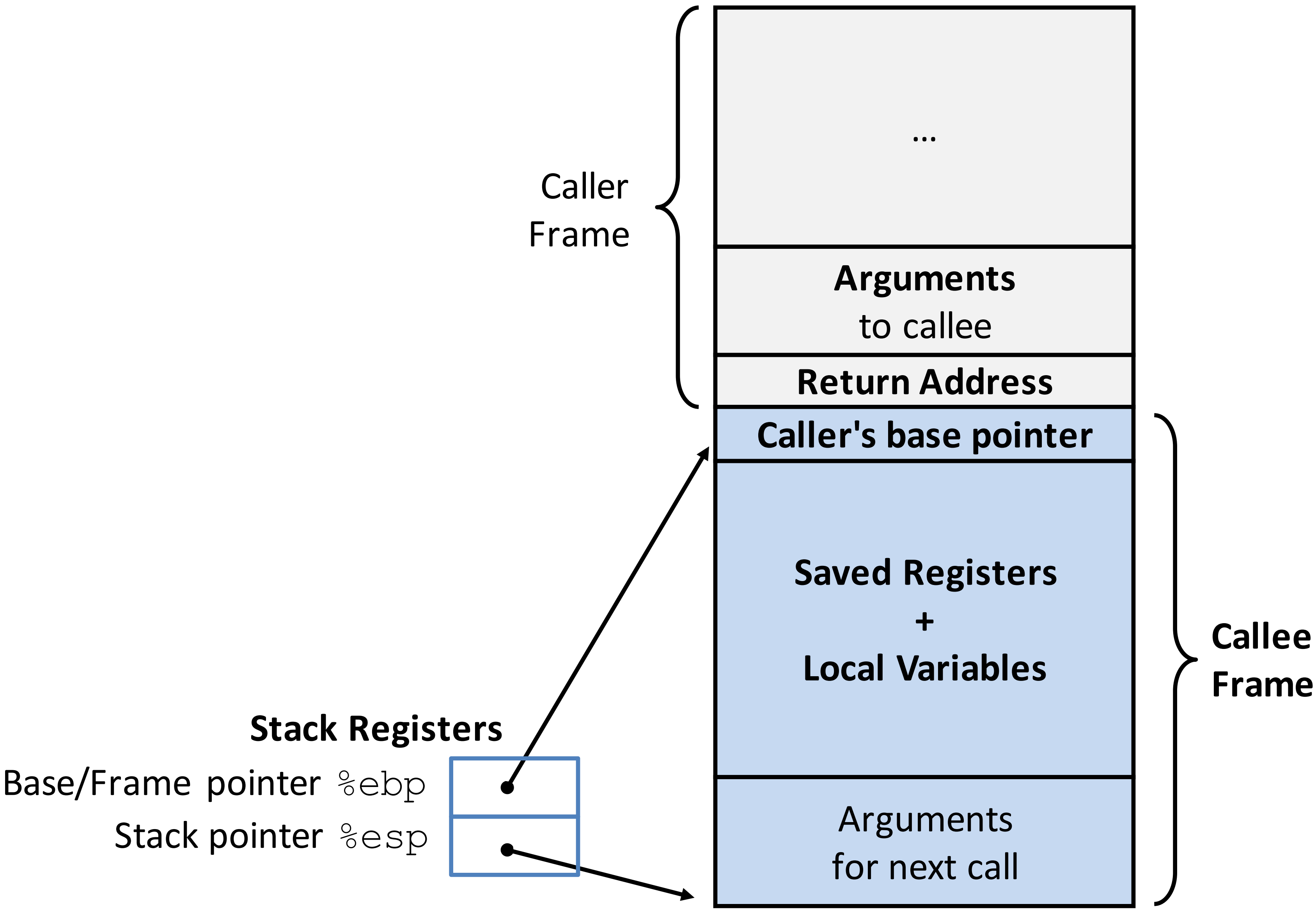
Procedure Call/Return: 4

```
8048591:  c3                         ret
```



`%eip:` *program counter*

IA32/Linux **Stack Frame**



Revisiting `swap`

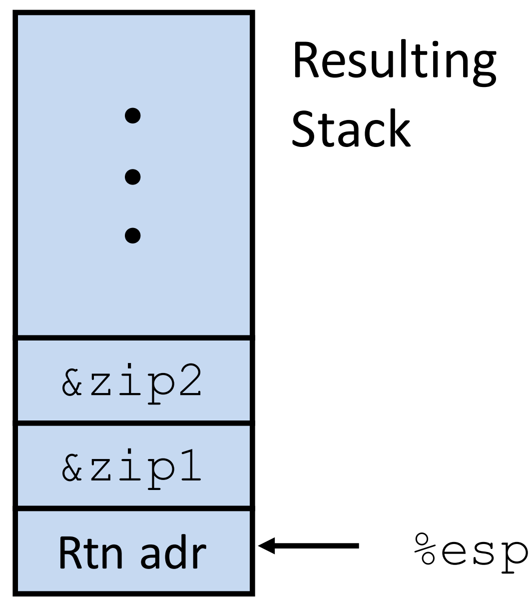
```
int zip1 = 02481;
int zip2 = 98195;

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
    • • •
```



Revisiting `swap`

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

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

Set Up

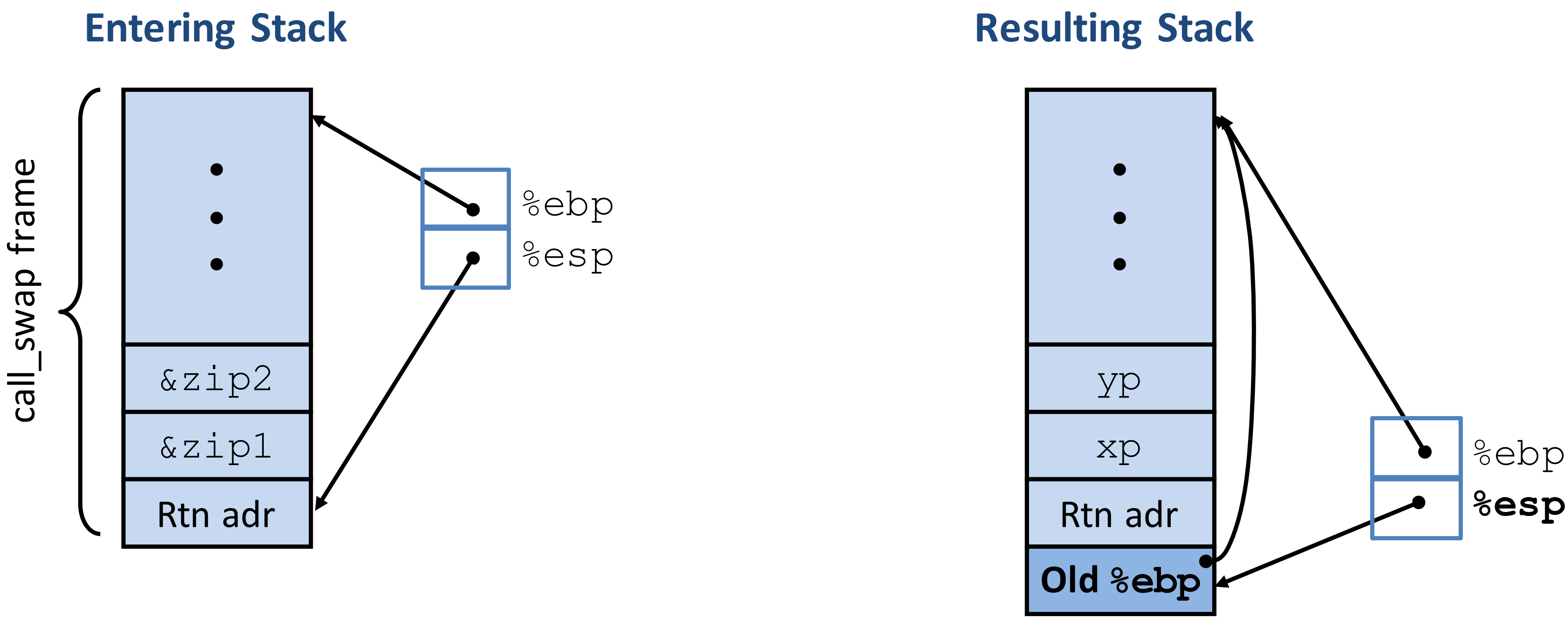
```
    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)
```

Body

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

Finish

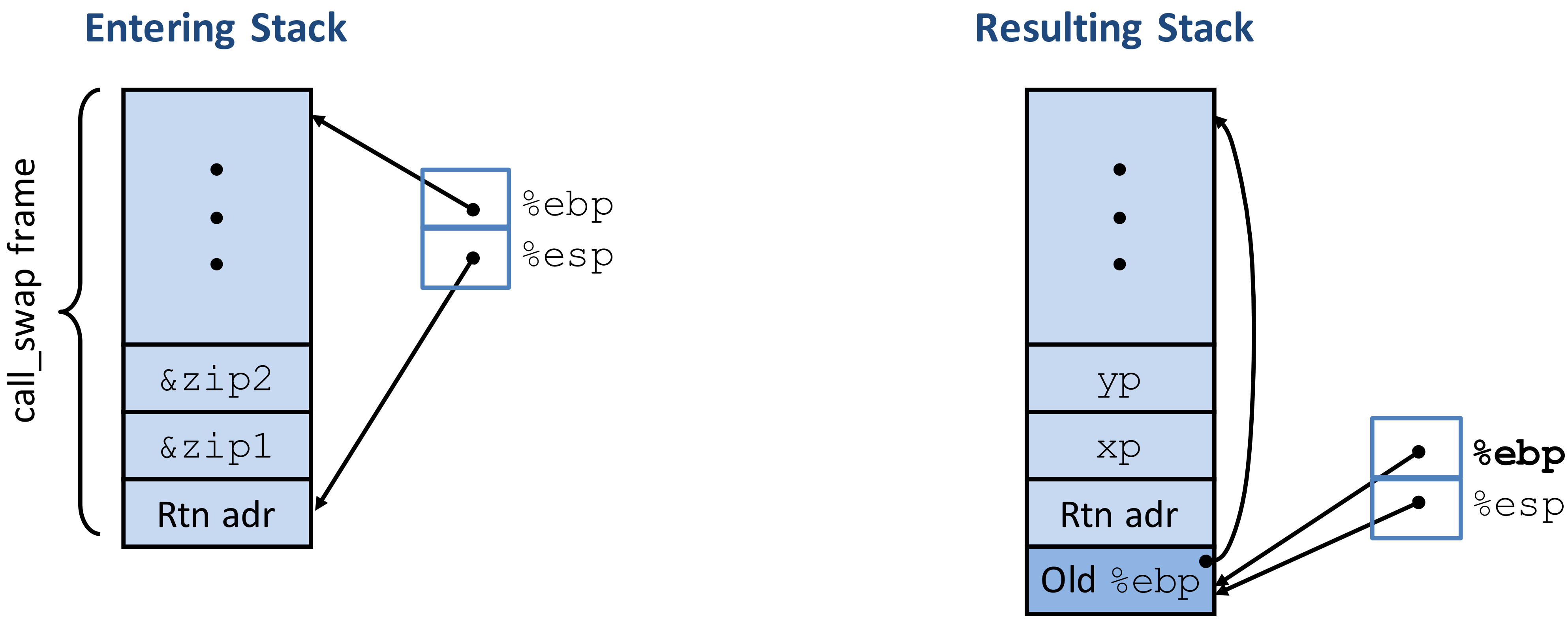
`swap` Setup #1




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

Set Up

swap Setup #2

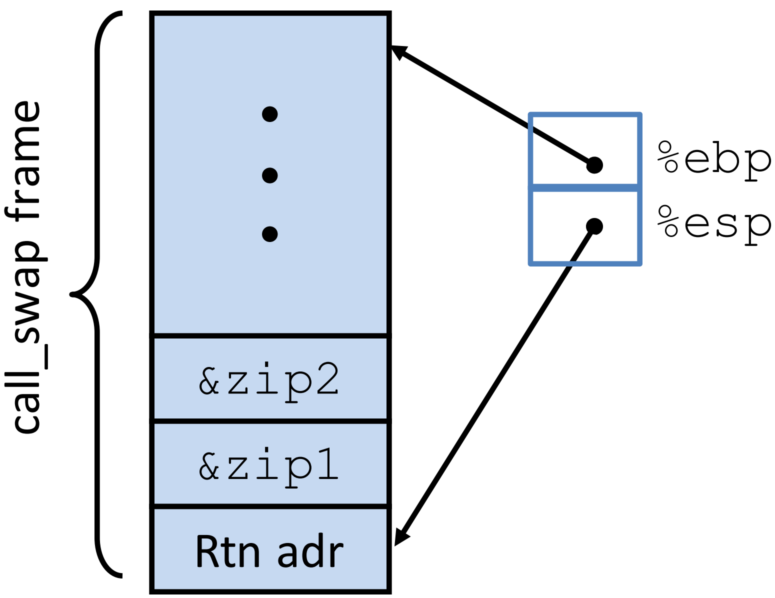



```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

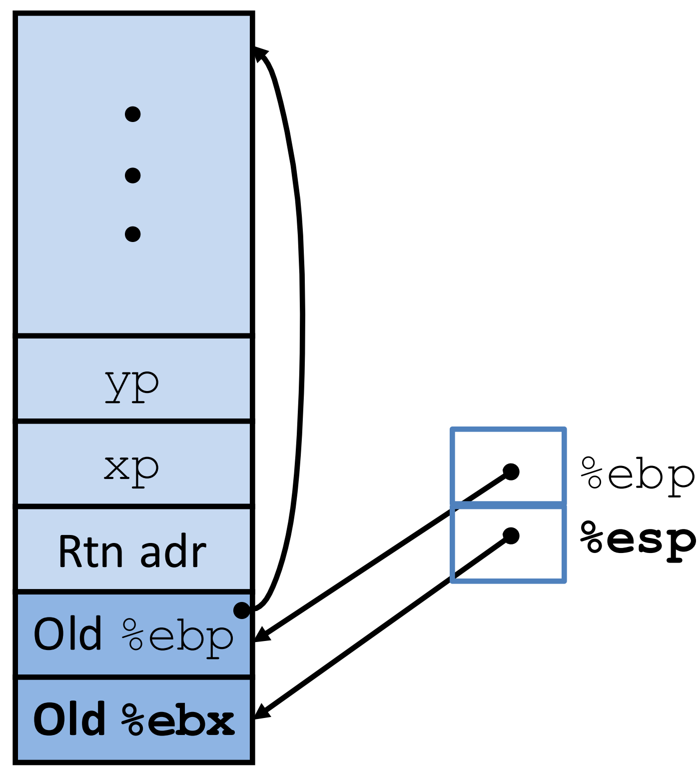
Set Up

swap Setup #3

Entering Stack



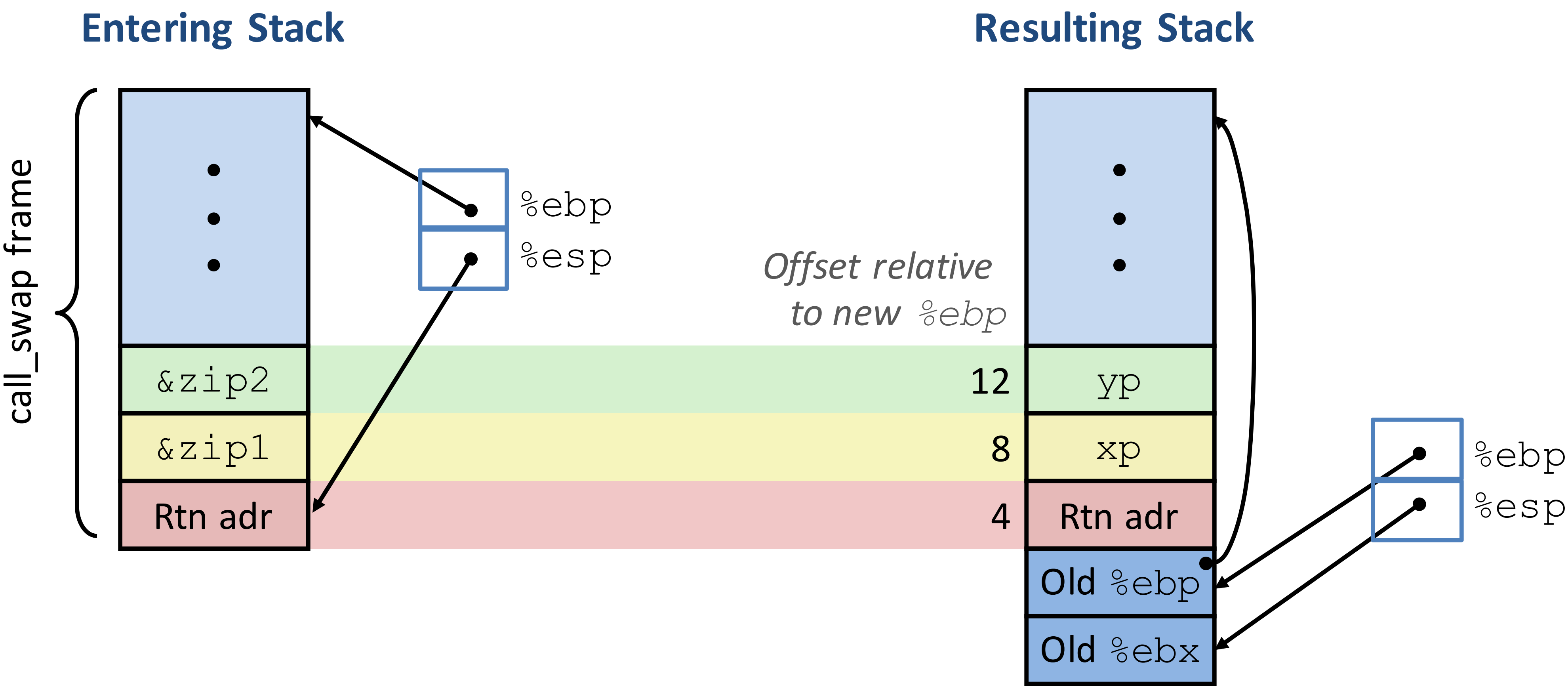
Resulting Stack




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

Set Up

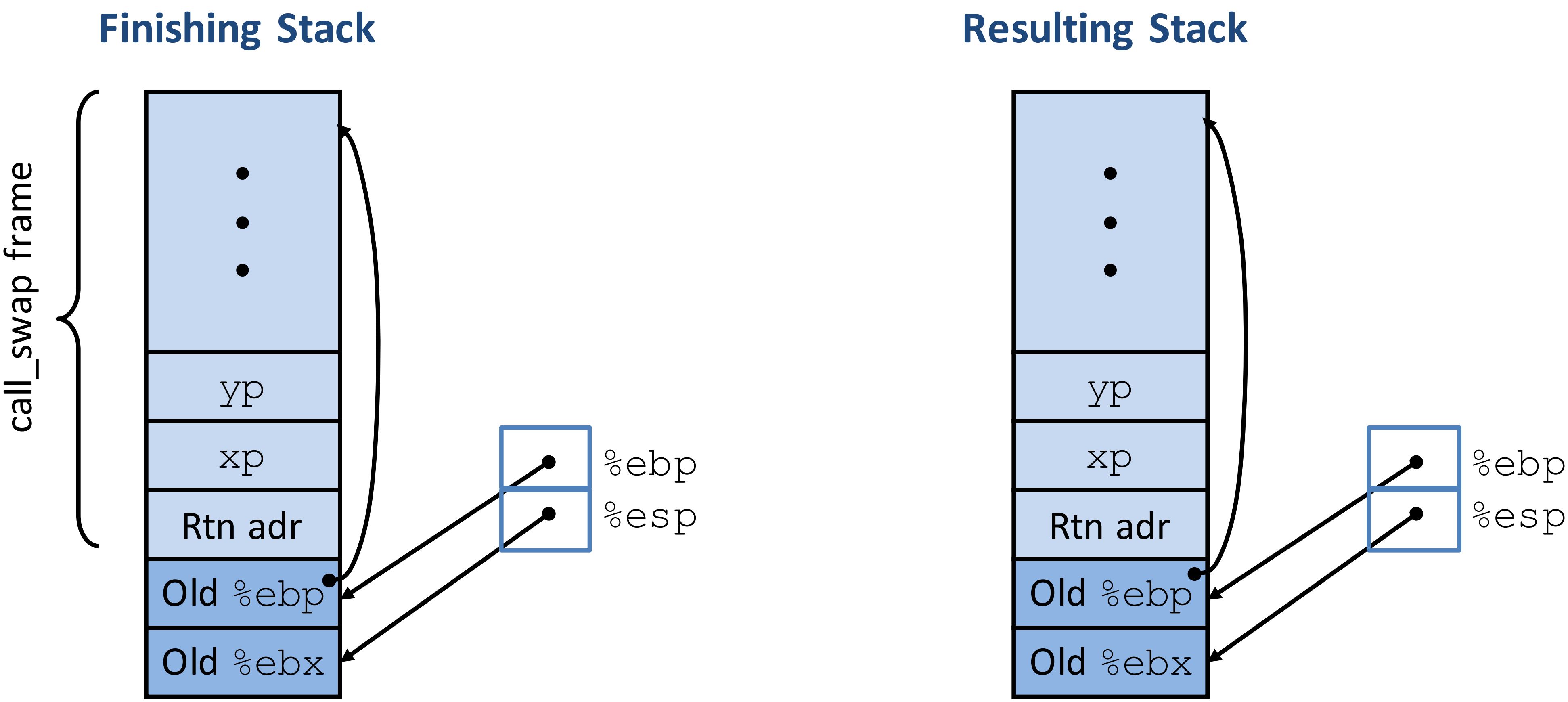
`swap` Body




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

Body

`swap` Finish #1

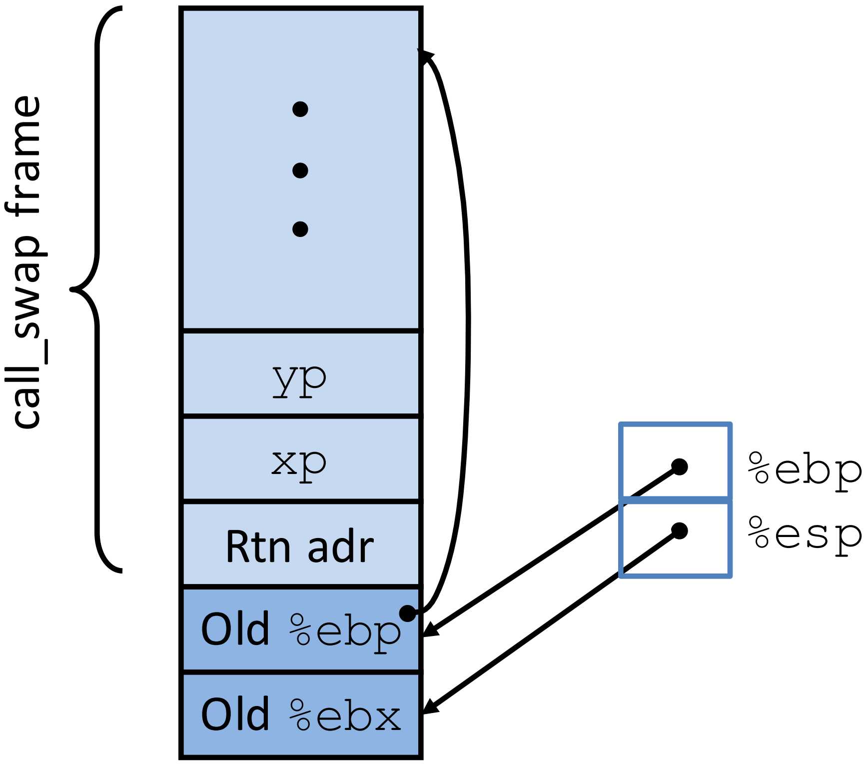



```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

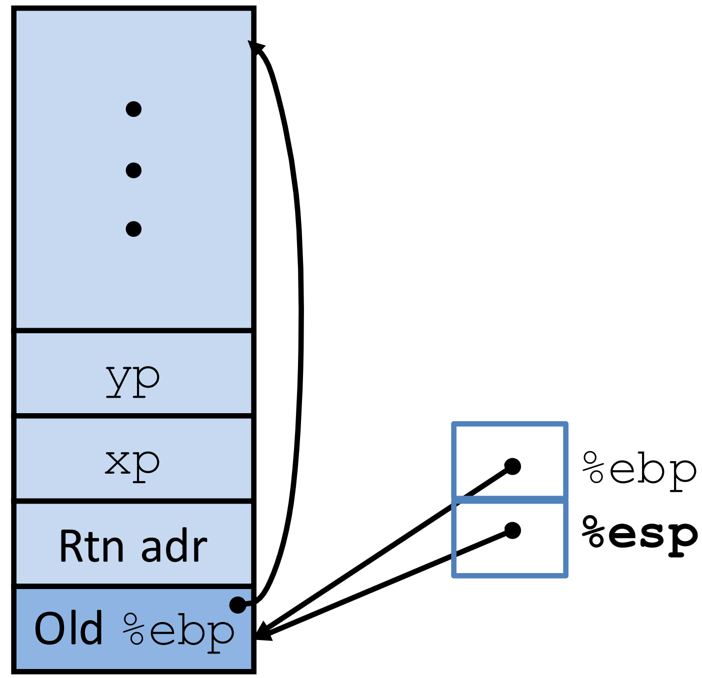
Finish

`swap` Finish #2

Finishing Stack



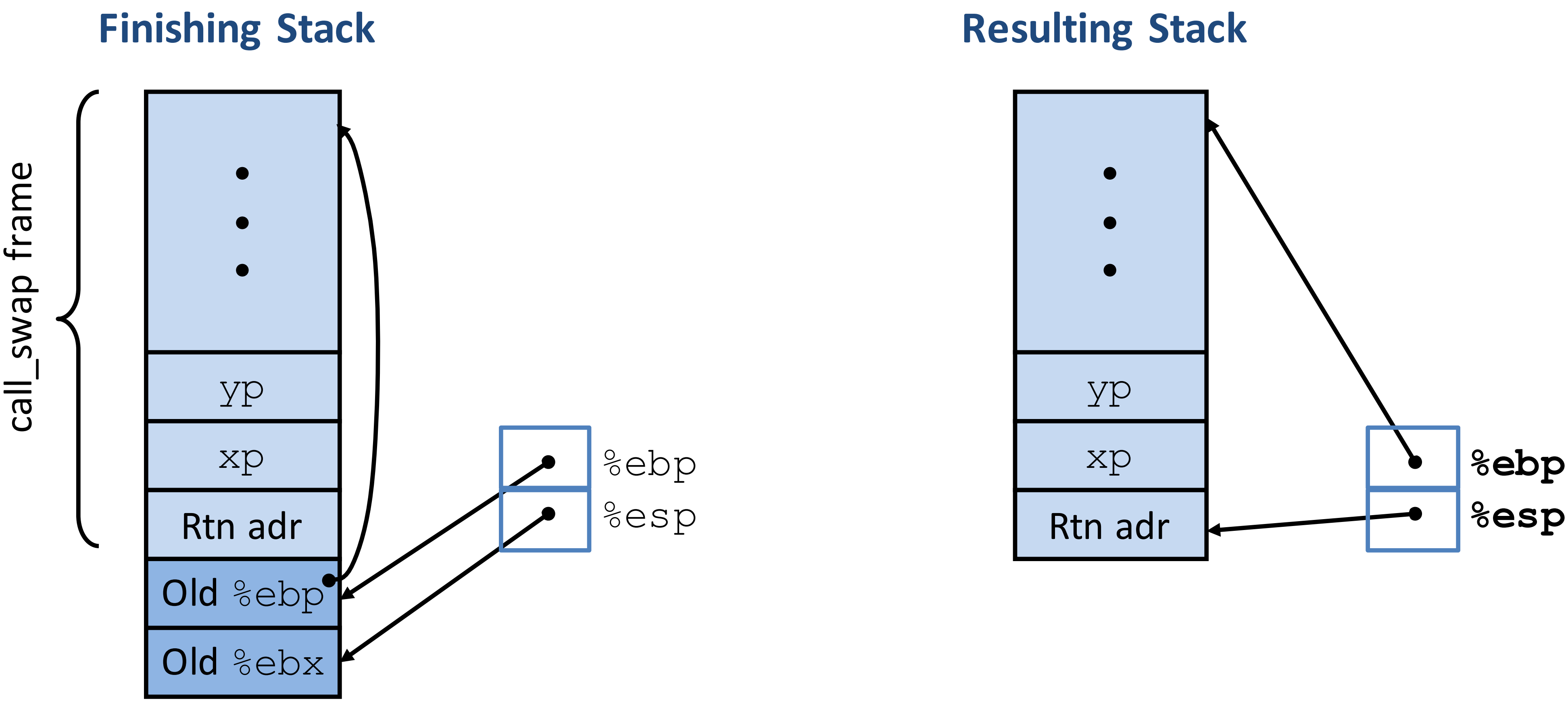
Resulting Stack




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

Finish

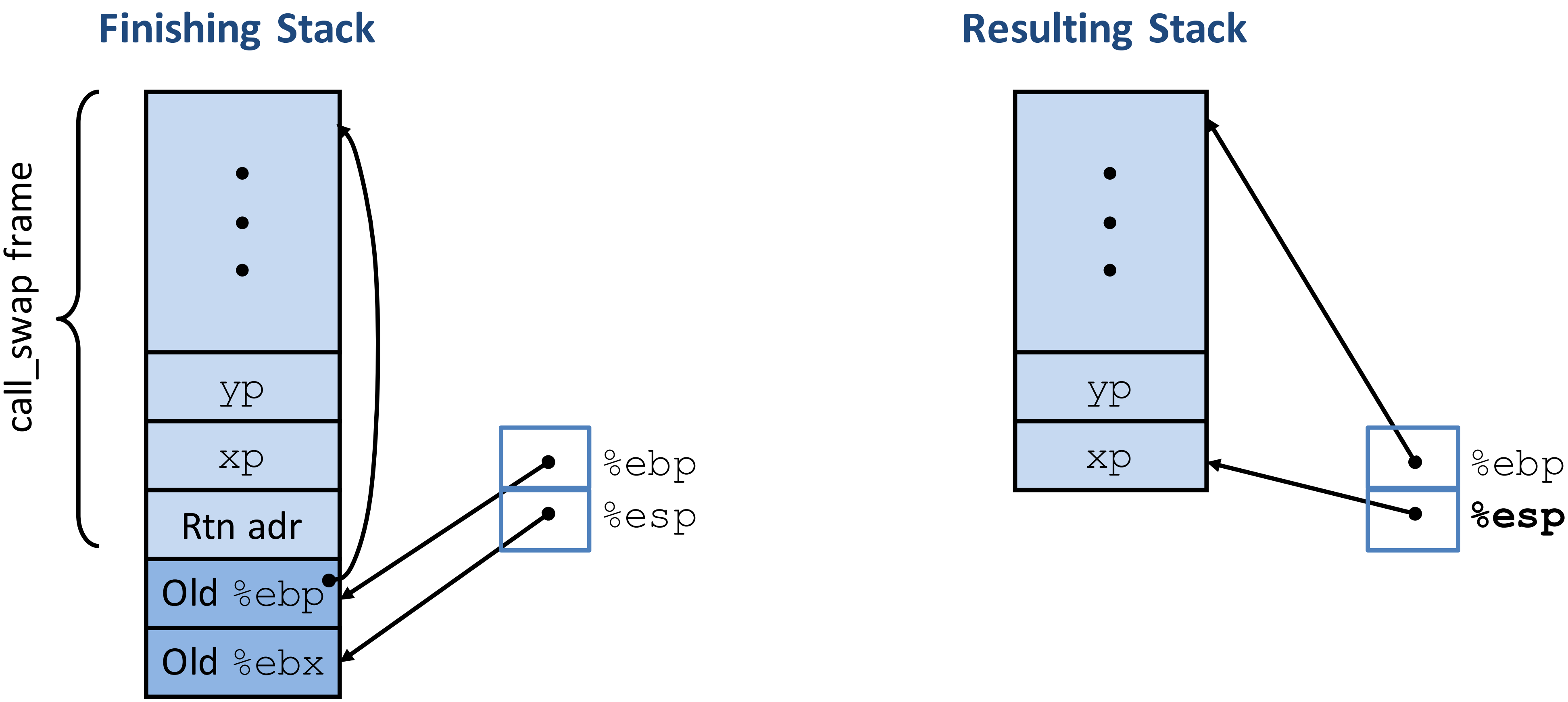
swap Finish #3




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

Finish

`swap` Finish #4




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

Finish

Disassembled `swap`

```
080483a4 <swap>:
 80483a4:    55                push   %ebp
 80483a5:    89 e5             mov    %esp,%ebp
 80483a7:    53                push   %ebx
 80483a8:    8b 55 08          mov    0x8(%ebp),%edx
 80483ab:    8b 4d 0c          mov    0xc(%ebp),%ecx
 80483ae:    8b 1a             mov    (%edx),%ebx
 80483b0:    8b 01             mov    (%ecx),%eax
 80483b2:    89 02             mov    %eax,(%edx)
 80483b4:    89 19             mov    %ebx,(%ecx)
 80483b6:    5b                pop    %ebx
 80483b7:    c9                leave
 80483b8:    c3                ret
```

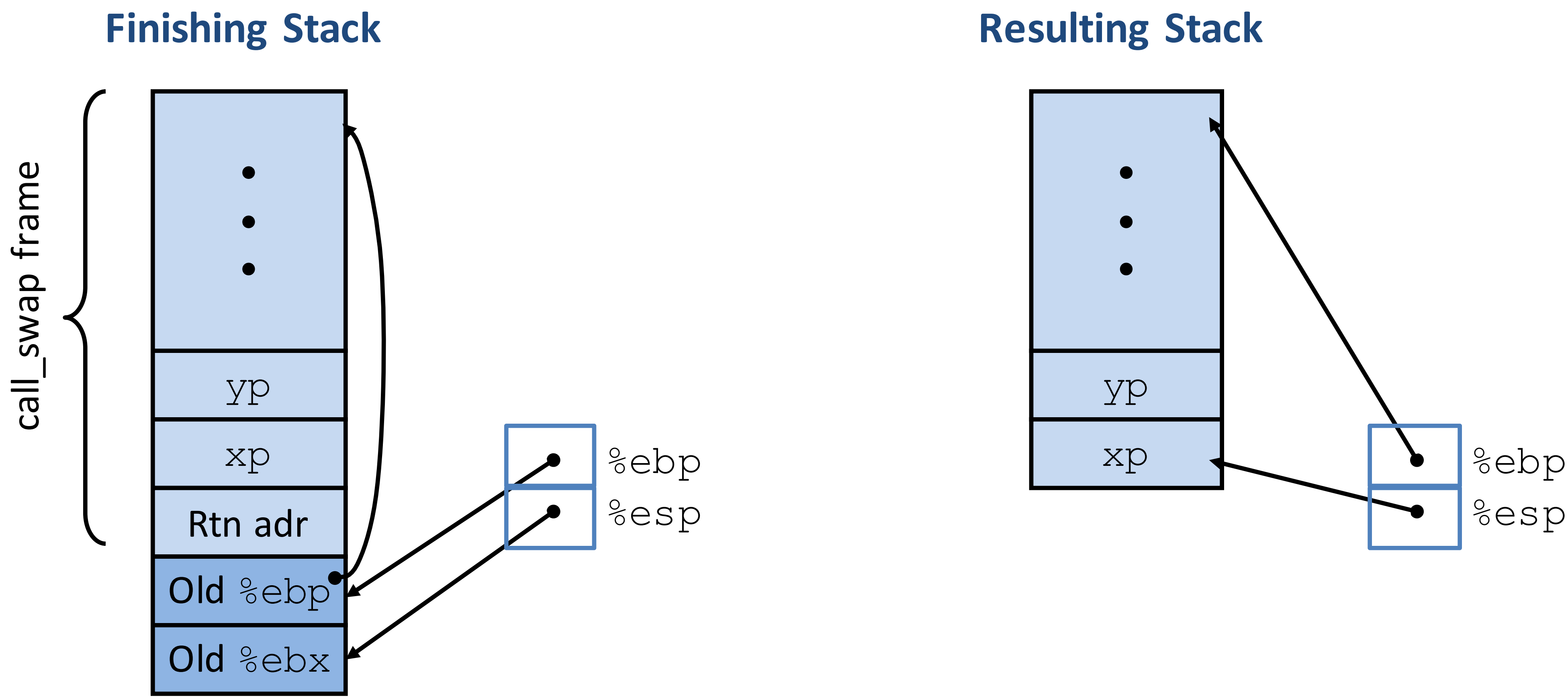
```
mov    %ebp,%esp
pop    %ebp
```

Calling Code

```
8048409:    e8 96 ff ff ff    call 80483a4 <swap>
804840e:    8b 45 f8          mov  0xfffffff8(%ebp),%eax
```

relative address (little endian)

`swap` Finish #4




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

Finish

Observation

Saved & restored register `%ebx`

but not `%eax`, `%ecx`, or `%edx`

Register Saving Conventions

When procedure `yoo` calls `who`:

`yoo` is the ***caller***

`who` is the ***callee***

Will register contents still be there after a procedure call?

```
yoo:
    • • •
    movl $12345, %edx
    call who
    addl %edx, %eax
    • • •
    ret
```

?

```
who:
    • • •
    movl 8(%ebp), %edx
    addl $98195, %edx
    • • •
    ret
```

Register Saving Conventions

When procedure `yoo` calls `who`:

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

Will register contents still be there after a procedure call?

Conventions

- ***Caller Save***
  - **Caller** saves temporary values in its frame **before calling**
- ***Callee Save***
  - **Callee** saves temporary values in its frame **before using**

IA32/Linux **Register Saving Conventions**

`%eax, %edx, %ecx`

Caller saves prior to call

if needs values after call

`%eax`

also used to **return** integer value

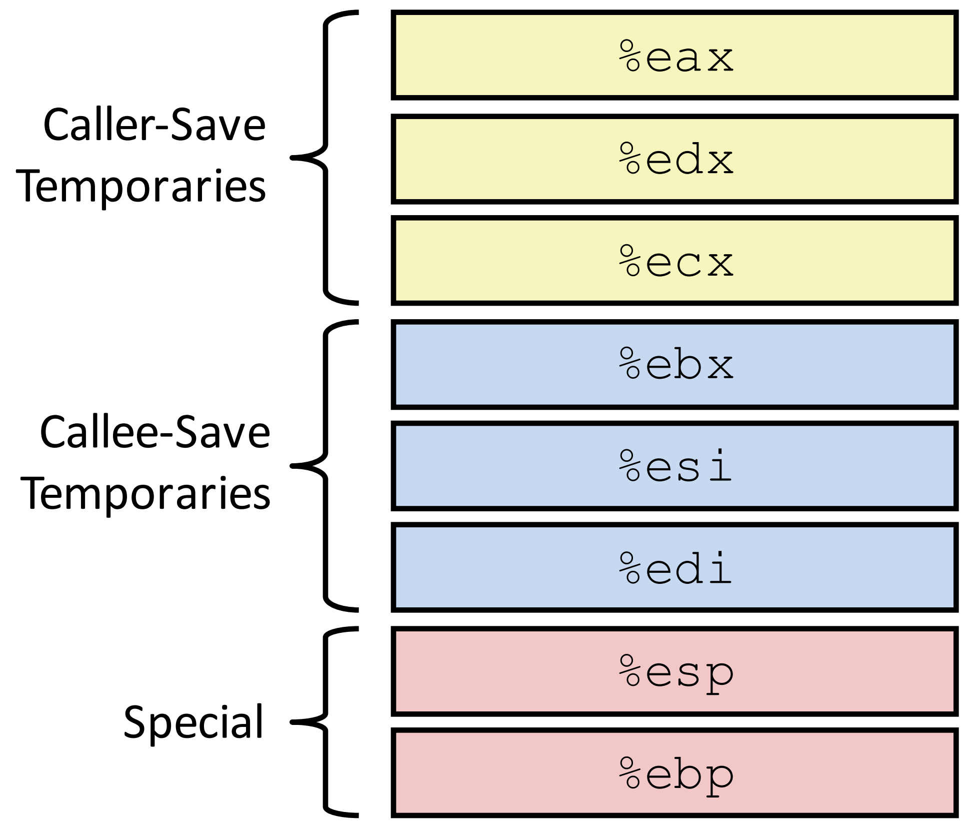
`%ebx, %esi, %edi`

Callee saves if will use

`%esp, %ebp`

special form of callee save

restored to original values before returning



A Puzzle

C function body:

```
*p = d;
return x - c;
```

assembly:

```
movsbl 12(%ebp),%edx
movl   16(%ebp),%eax
movl   %edx,(%eax)
movswl 8(%ebp),%eax
movl   20(%ebp),%edx
subl   %eax,%edx
movl   %edx,%eax
```

Write the C function header, types, and order of parameters.

movsbl = move sign-extending a byte to a long (4-byte)
movswl = move sign-extending a word (2-byte) to a long (4-byte)

Example: **Pointers to Local Variables**

Top-Level Call

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

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);
  }
}
```

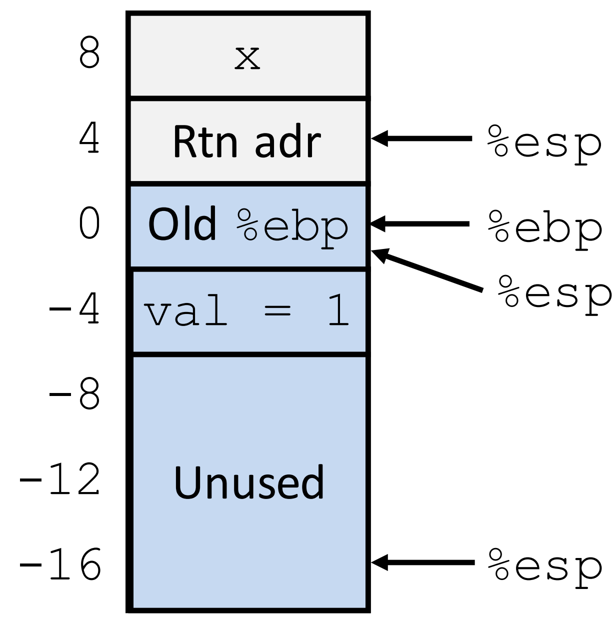
sfact(3)	val = 1
s_helper(3, &val)	val = 3
s_helper(2, &val)	val = 6
s_helper(1, &val)	val = 6.

Pass pointer to update location

Creating & Initializing Pointer

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

Must store **`val`** in memory (stack); registers do not have addresses.



Initial part of `sfact`

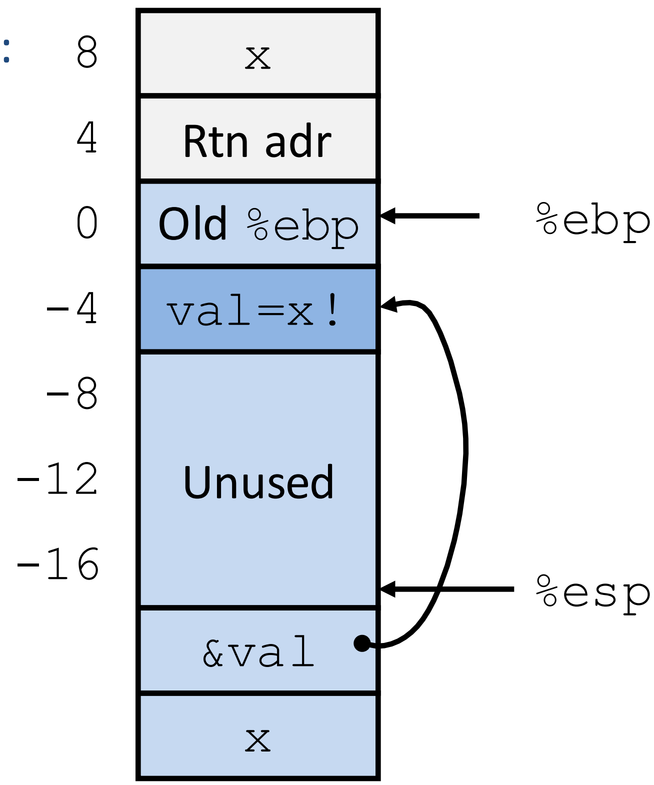
```
_sfact:
  pushl %ebp           # Save %ebp
  movl %esp,%ebp       # Set %ebp
  subl $16,%esp        # Add 16 bytes
  movl 8(%ebp),%edx    # edx = x
  movl $1,-4(%ebp)     # val = 1
```

Passing Pointer

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

Must store **val** in memory (stack); registers do not have addresses.

Stack at time of call:



Calling `s_helper` from `sfact`

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

IA32/Linux **Procedure Summary**

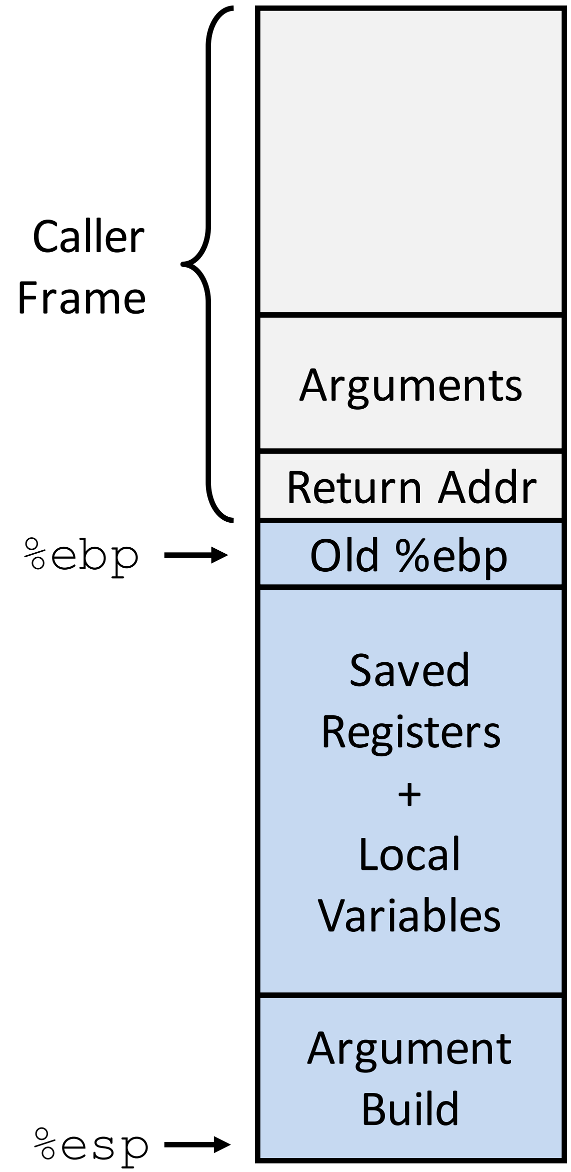
`call, ret, push, pop`

Stack discipline fits procedure call / return.*

- If P calls Q, Q returns before P, including recursion.

Conventions support arbitrary function calls.

- Safely store per-call values in local stack frame and callee-saved registers
- Push function arguments at top of stack before call
- Result returned in **`%eax`**



*Take 251 to learn about languages where it doesn't.