Machine-Level Programming III - Procedures

Today

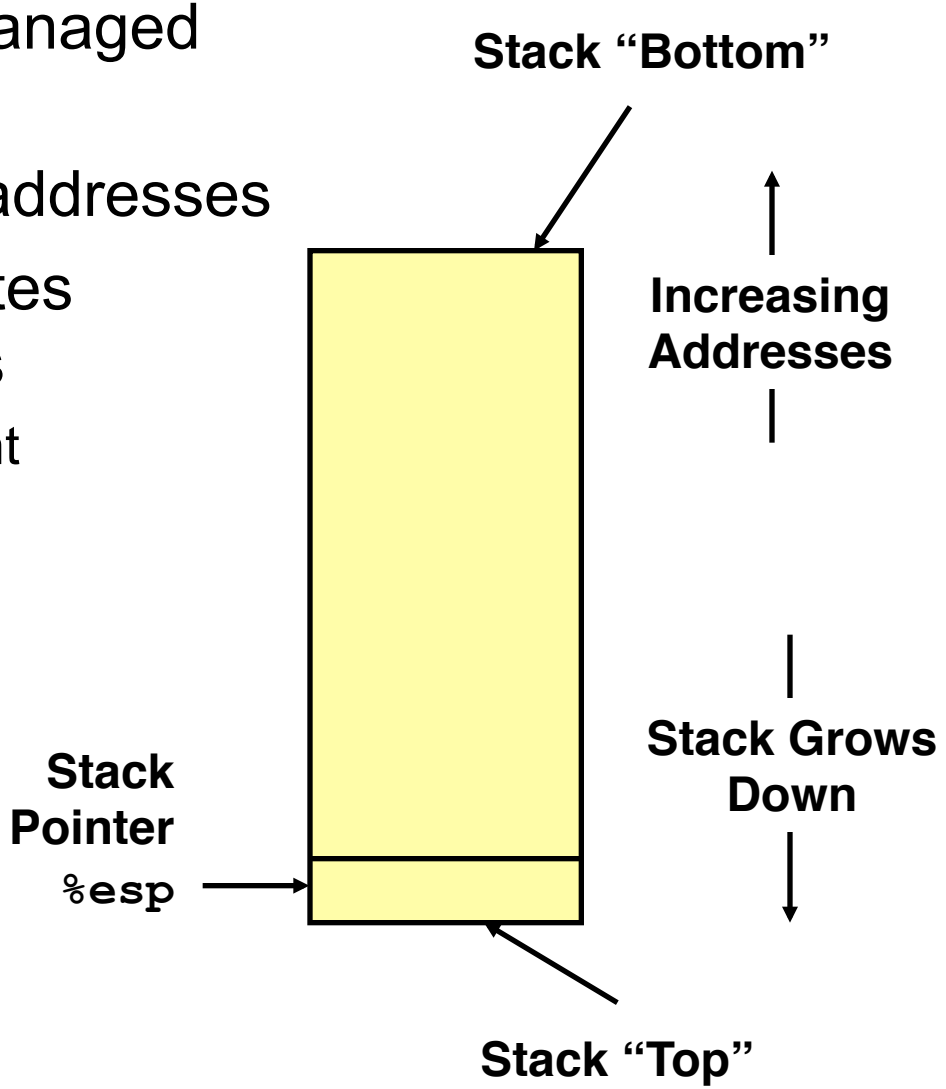
- IA32 stack discipline
- Register saving conventions
- Creating pointers to local variables

Next time

- Structured data

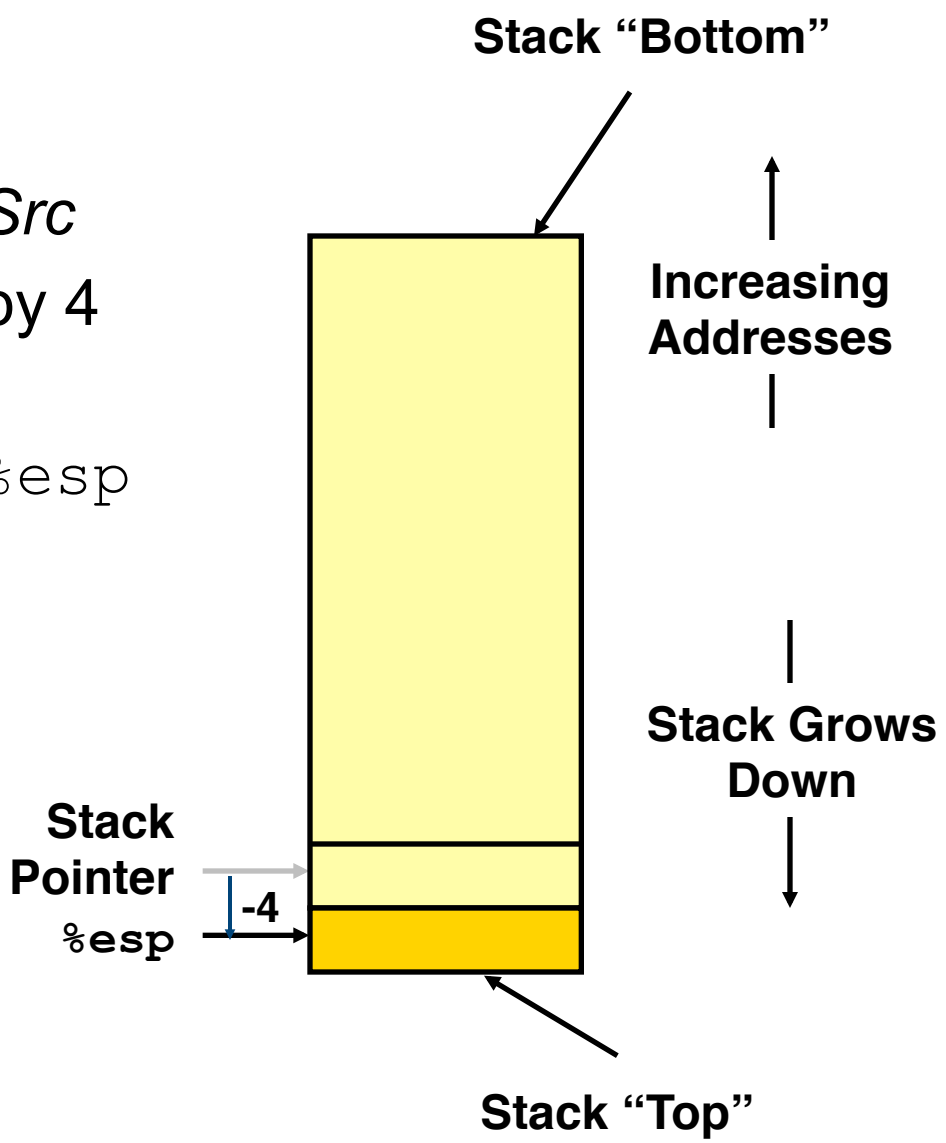
IA32 Stack

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



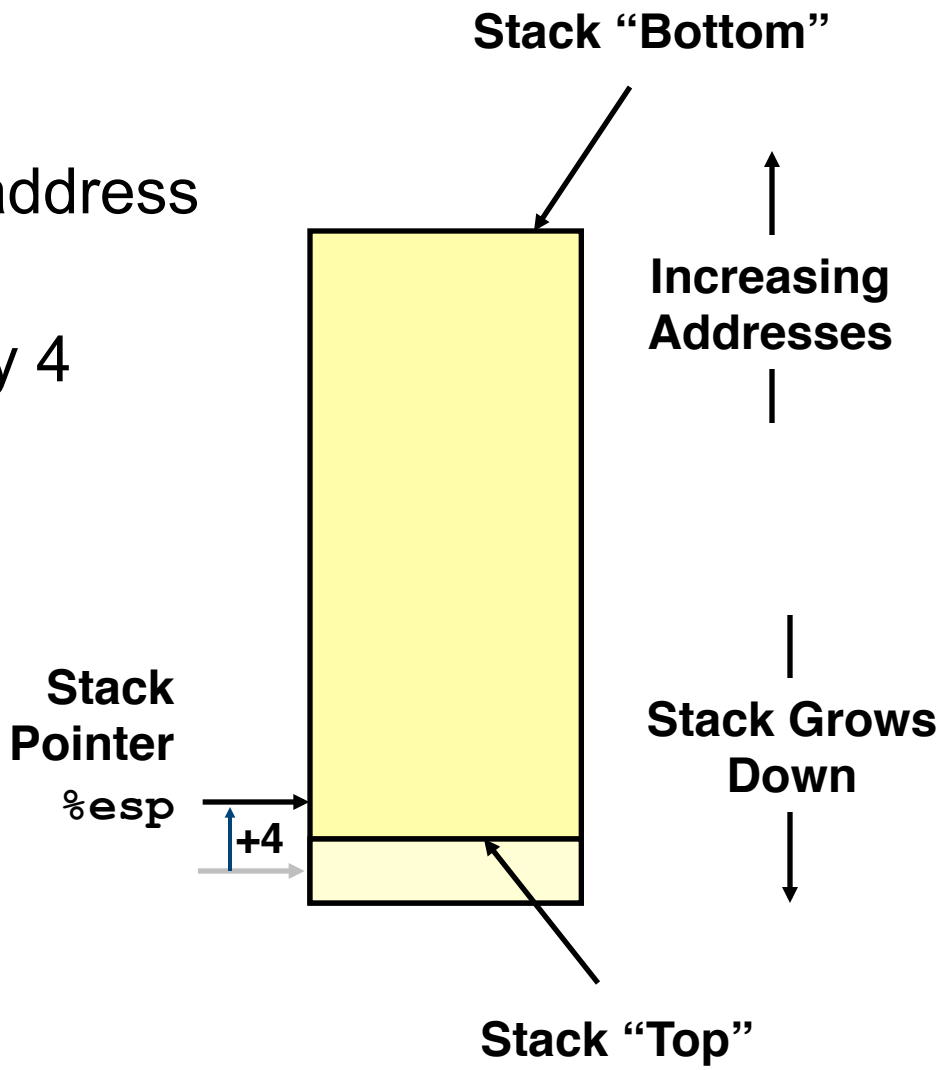
IA32 Stack pushing

- Pushing
  - `pushl` *Src*
  - Fetch operand at *Src*
  - Decrement `%esp` by 4
  - Write operand at address given by `%esp`

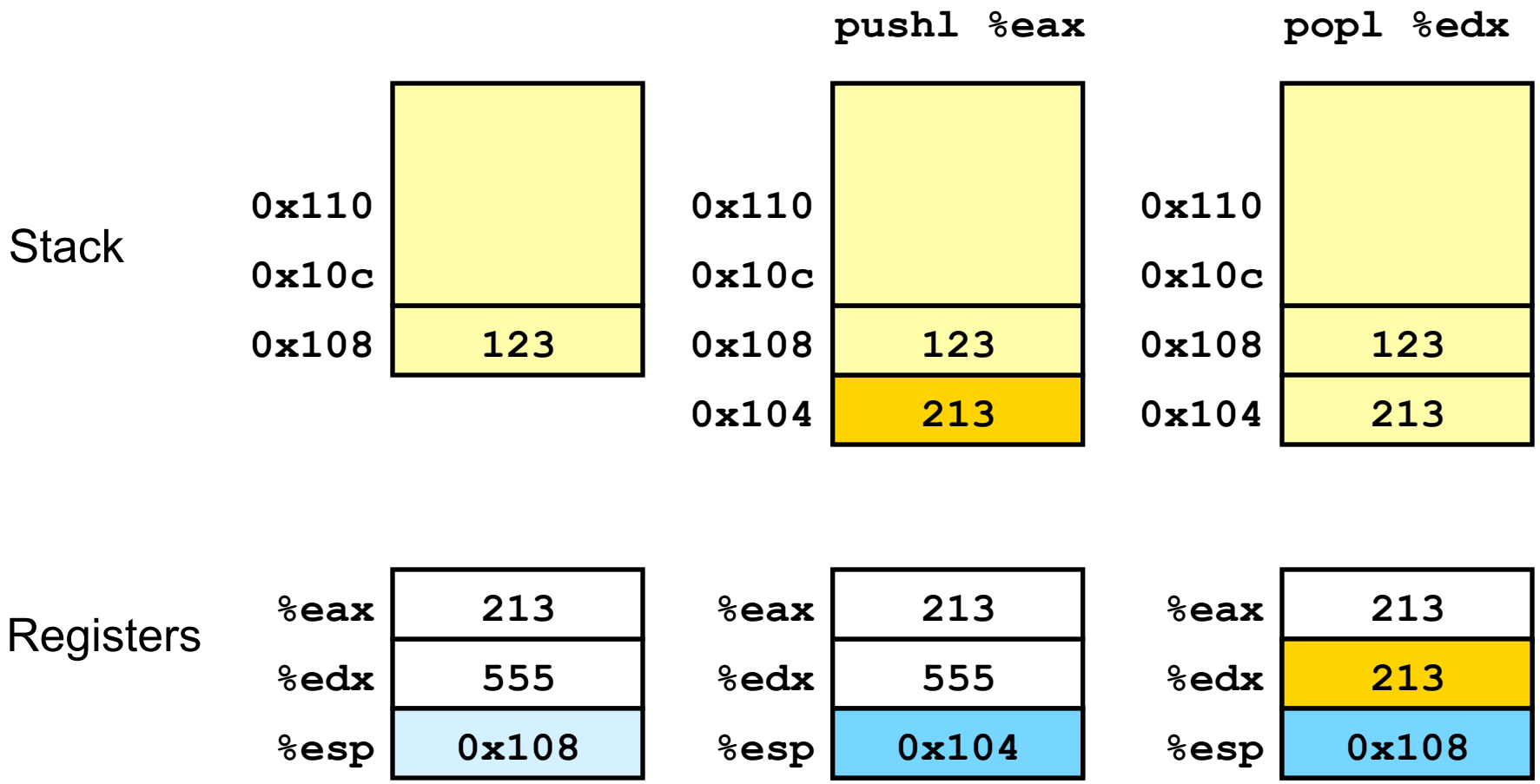


IA32 Stack popping

- Popping
  - `popl` *Dest*
  - Read operand at address given by `%esp`
  - Increment `%esp` by 4
  - Write to *Dest*



Stack operation examples



Procedure control flow

- Use stack to support procedure call and return
- Procedure call

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

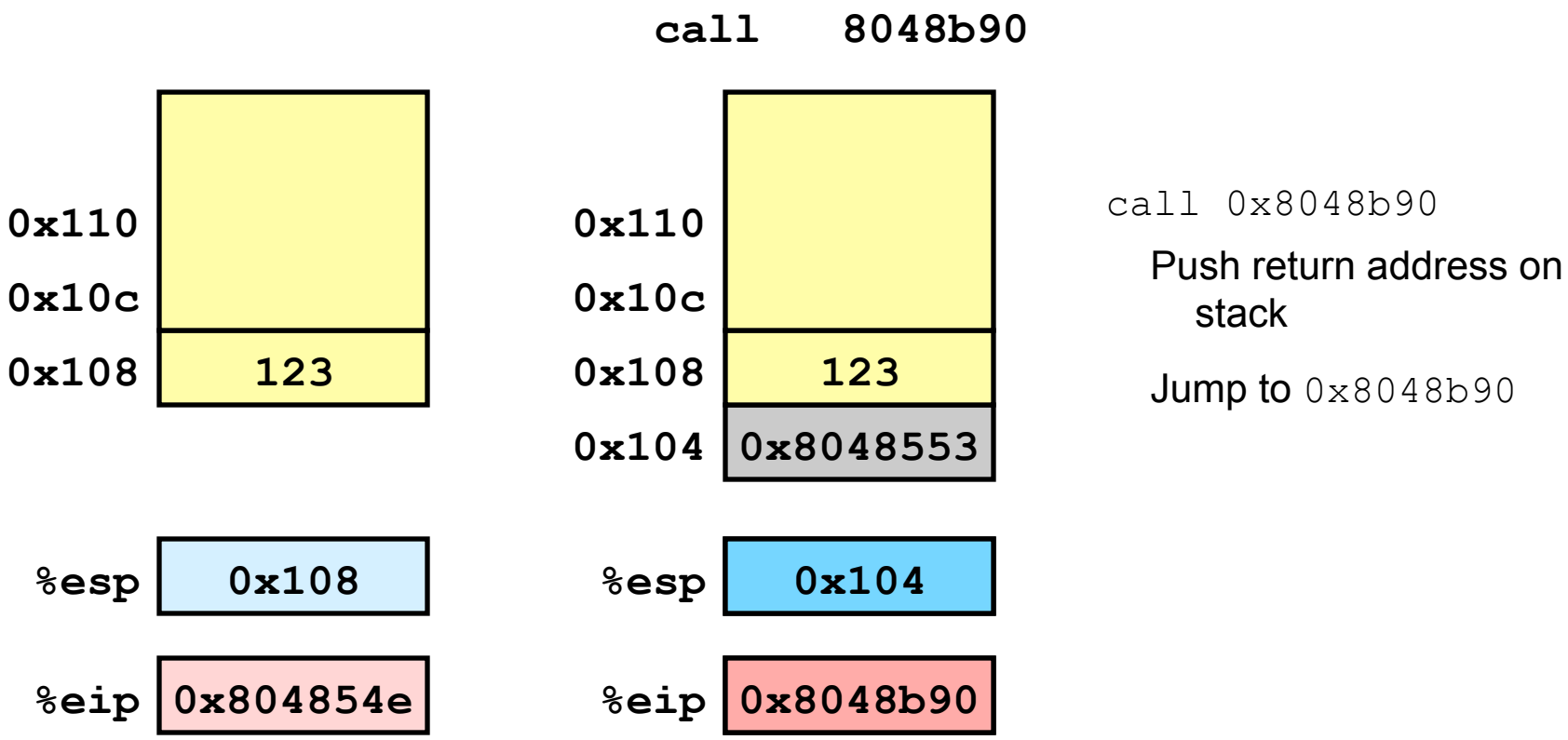
  `call` *`*Operand`* Similar, but indirect
- Procedure return
  - `leave` Prepare stack for return
  - `ret` Pop address from stack; jump there (stack should be ready)
- Return address value
  - Address of instruction immediately following `call`
  - Example from disassembly

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

Return address: `0x8048553`

Procedure call example

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



`call 0x8048b90`

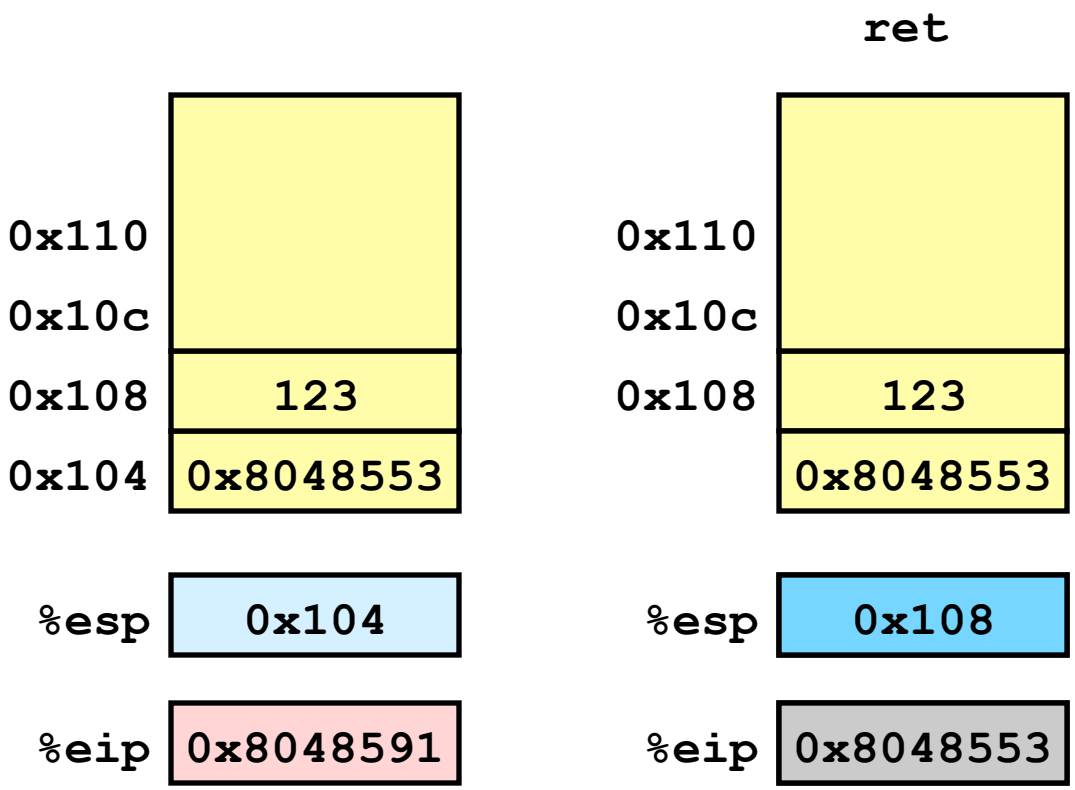
Push return address on stack

Jump to `0x8048b90`

`%eip` **is program counter**

Procedure return example

```
8048591:   c3                    ret
```



`ret`

Pop address from stack

Jump to address

`%eip` is program counter

Stack-based languages

- Languages that support recursion
  - e.g., C, Pascal, Java
  - Code must be "*reentrant*"
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store state of each instantiation
    - Arguments
    - Local variables
    - Return pointer
- 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 of a single procedure instantiation

Call chain example

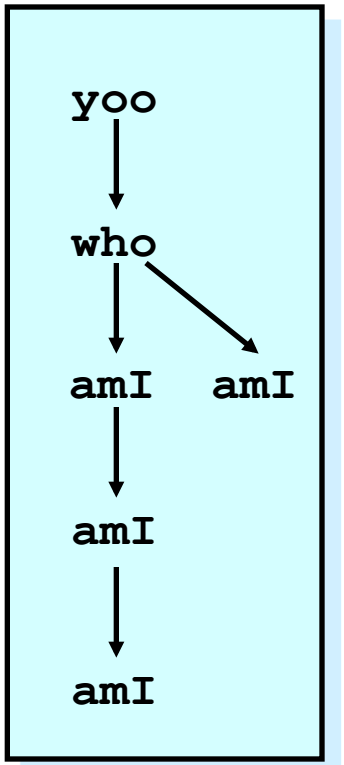
Code structure

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

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

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

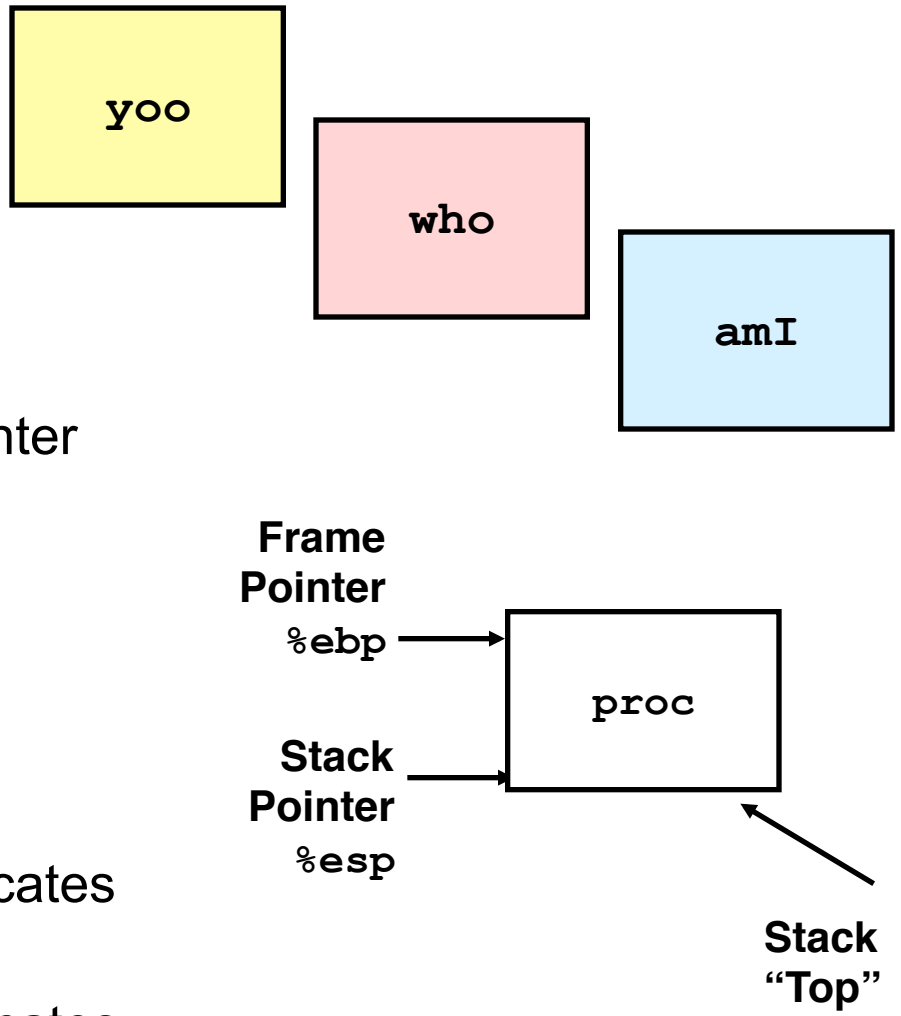
Call Chain



Procedure `amI` recursive

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 `%esp` indicates stack top
  - Frame pointer `%ebp` indicates start of current frame

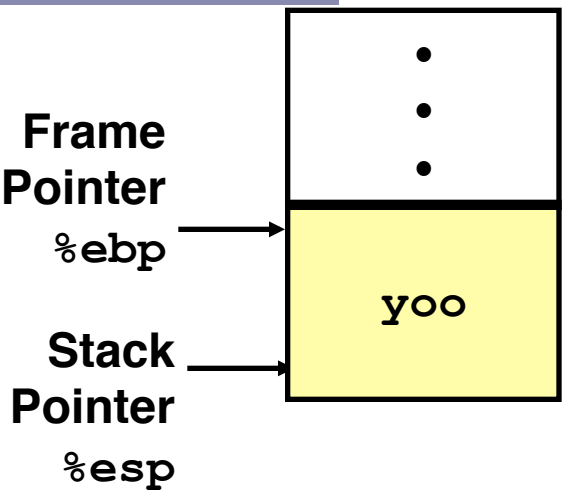


Stack operation

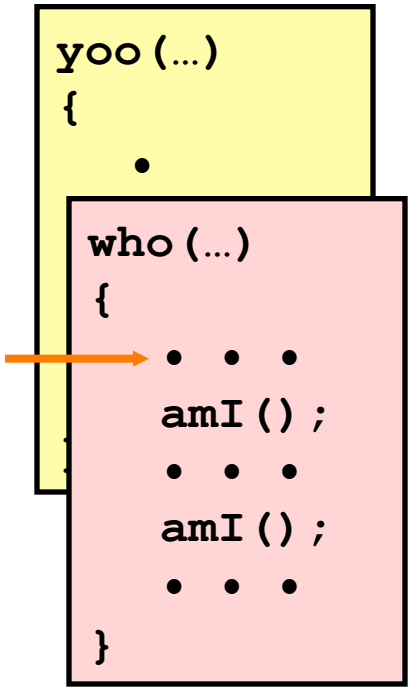
```
yoo(...)
{
    •
    •
    who();
    •
    •
}
```

Call Chain

yoo



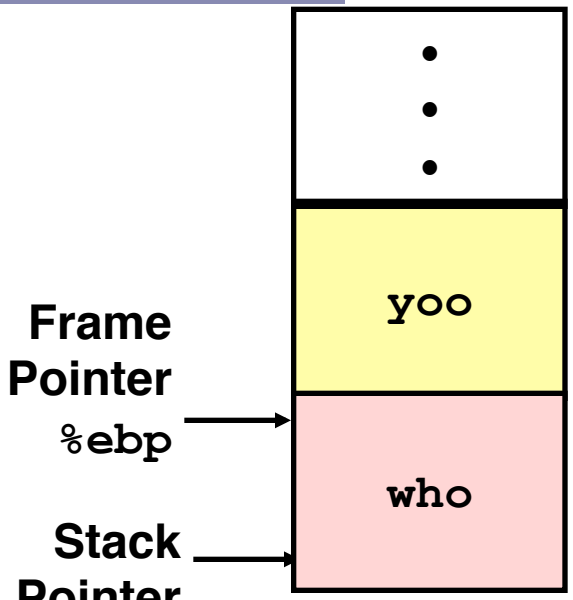
Stack operation



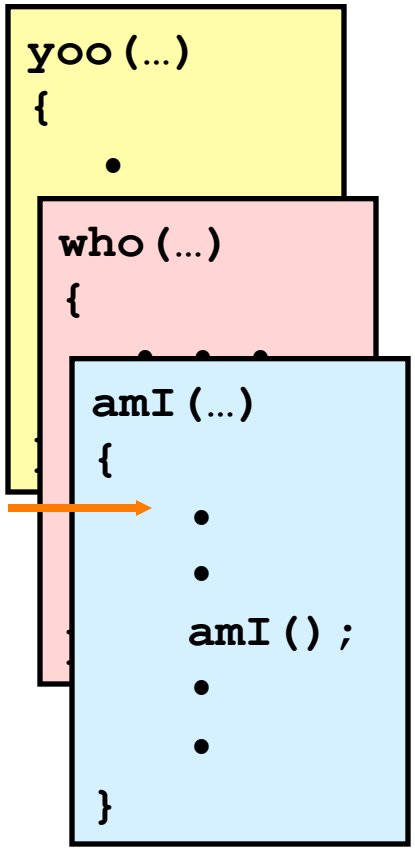
Call Chain

yoo

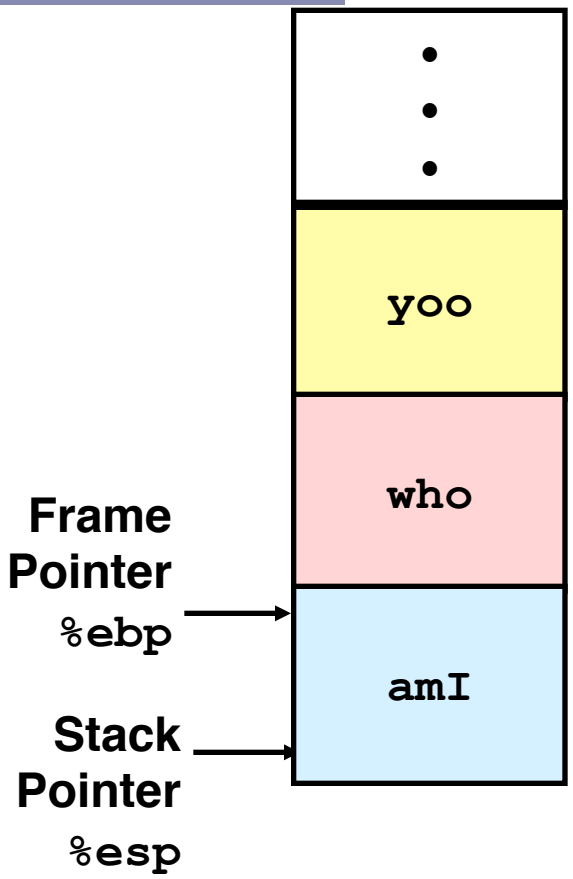
who



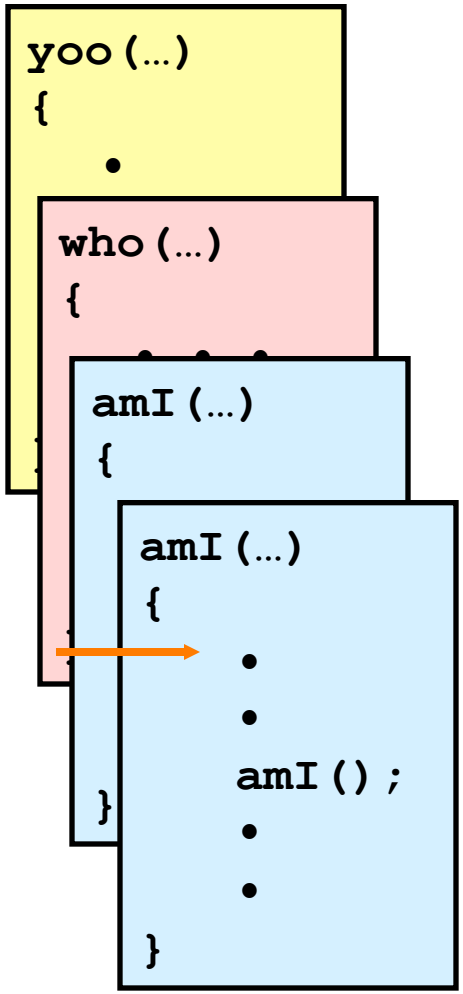
Stack operation



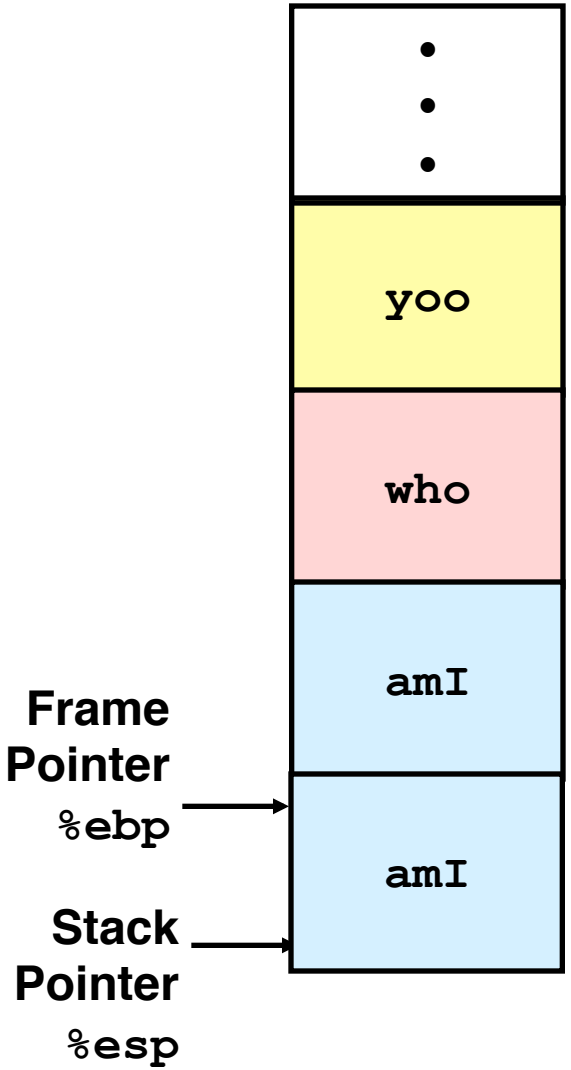
Call Chain



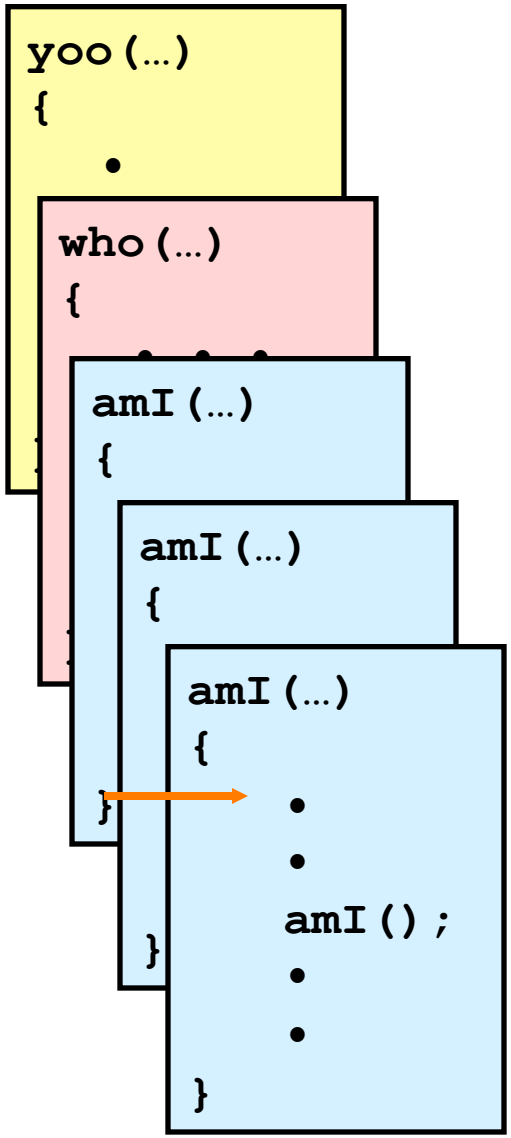
Stack operation



Call Chain

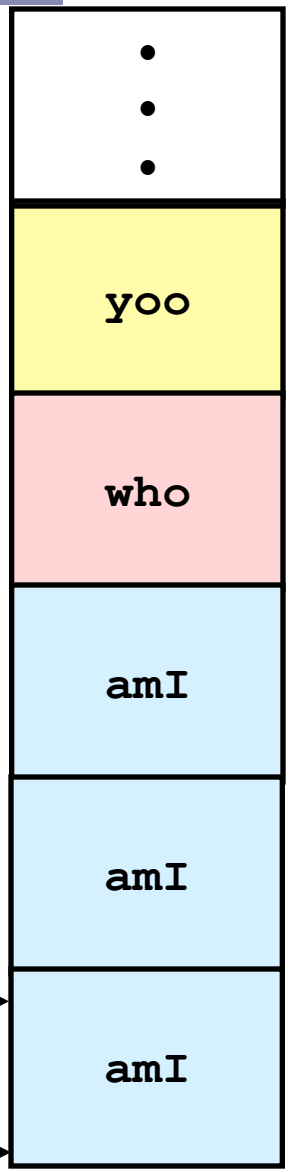


Stack operation



Call Chain

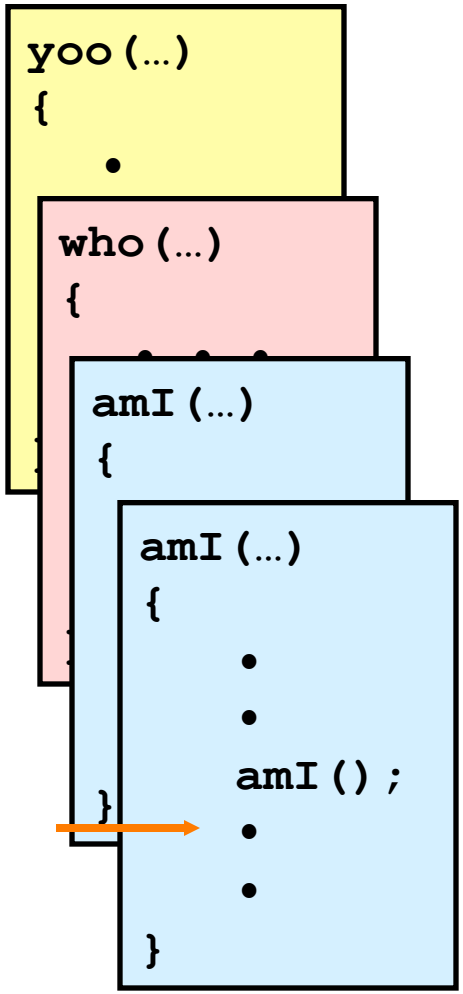
yoo
who
amI
amI
amI



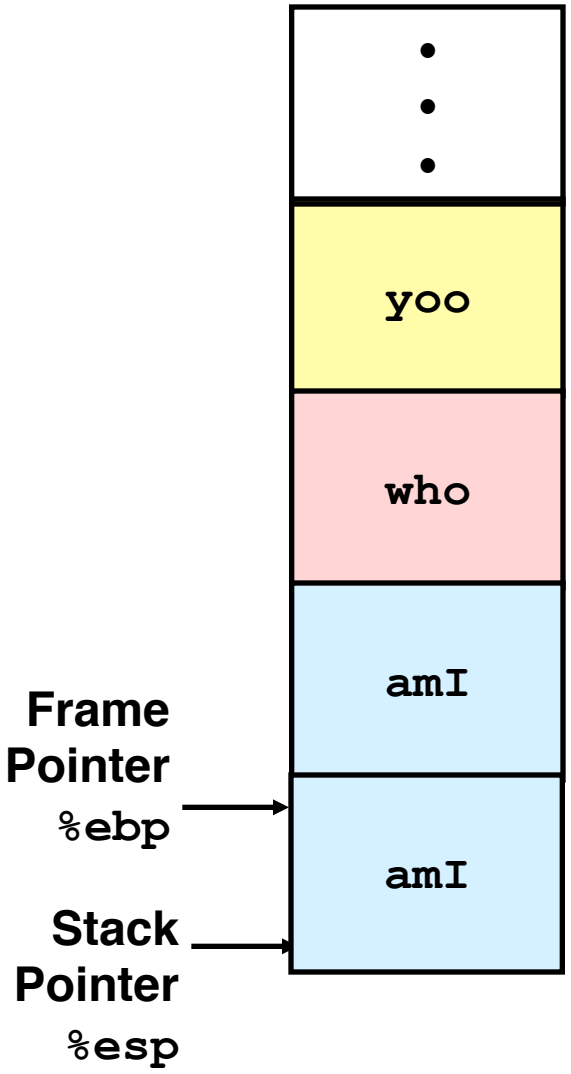
Frame Pointer `%ebp`

Stack Pointer `%esp`

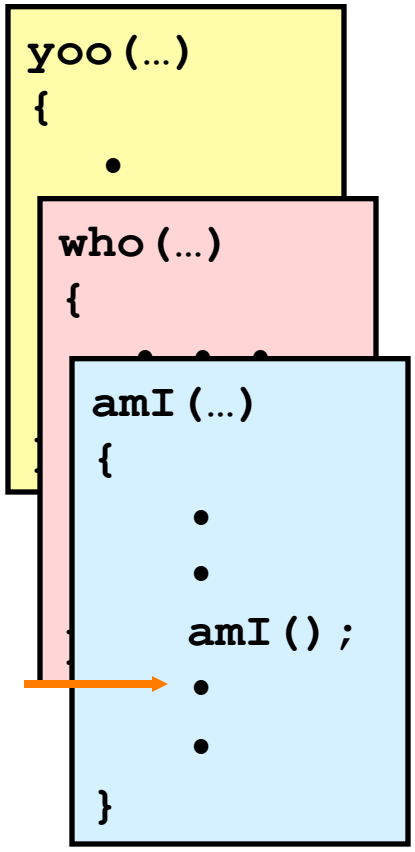
Stack operation



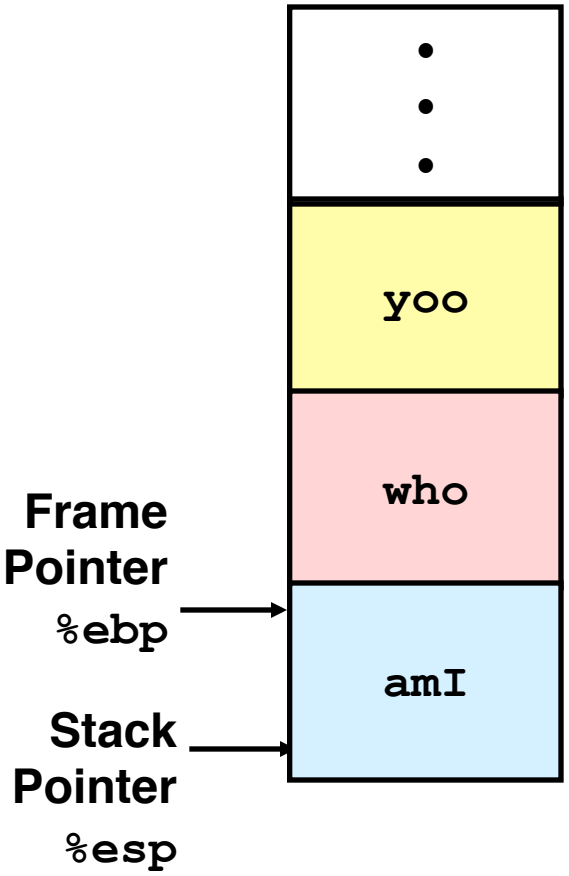
Call Chain



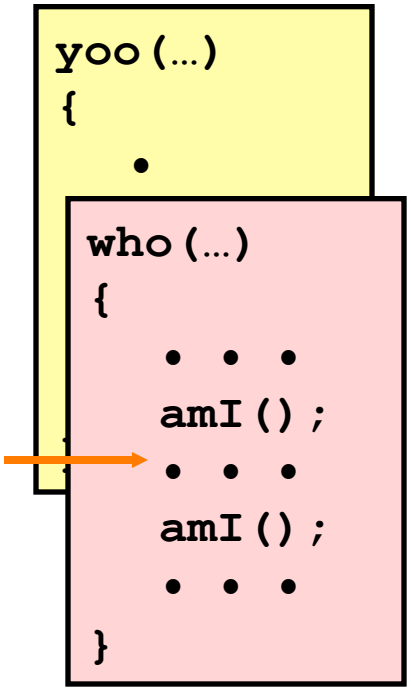
Stack operation



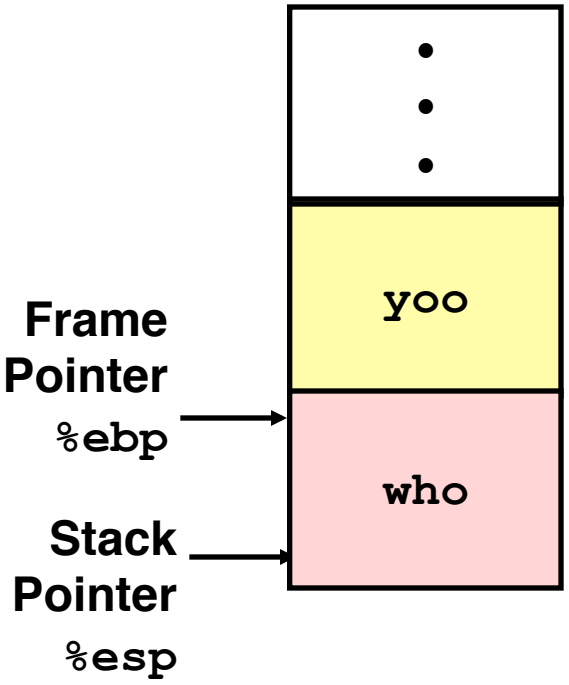
Call Chain



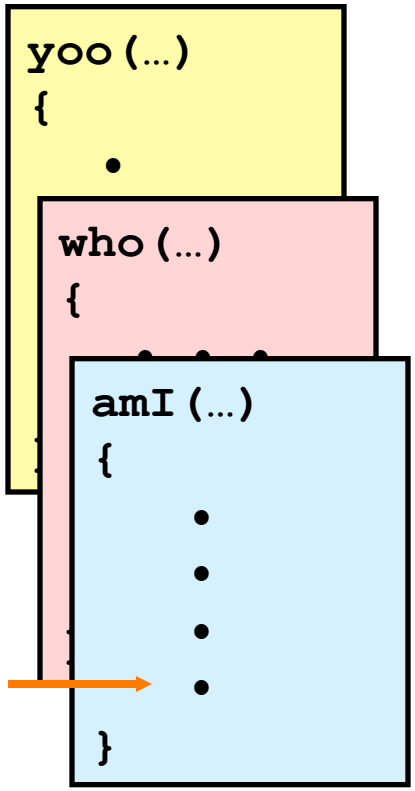
Stack operation



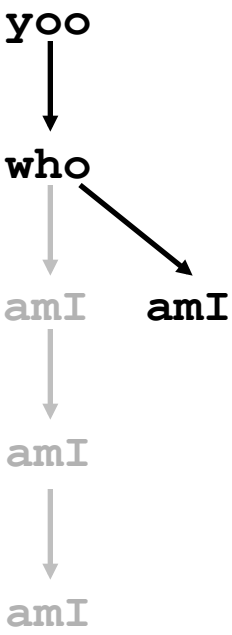
Call Chain

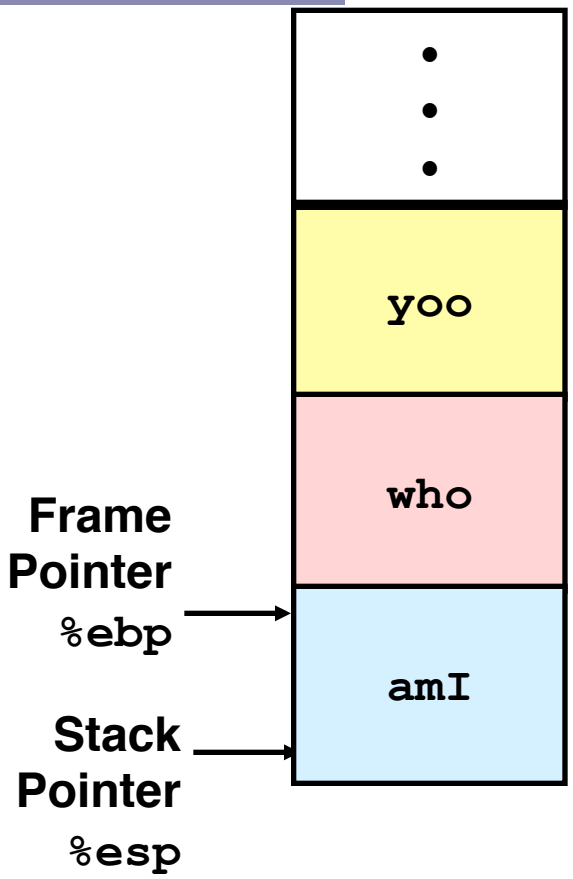


Stack operation

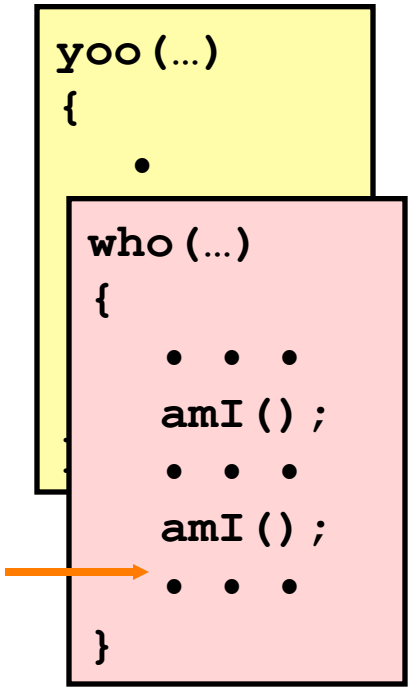


Call Chain

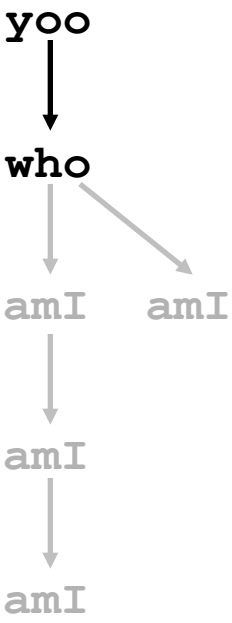


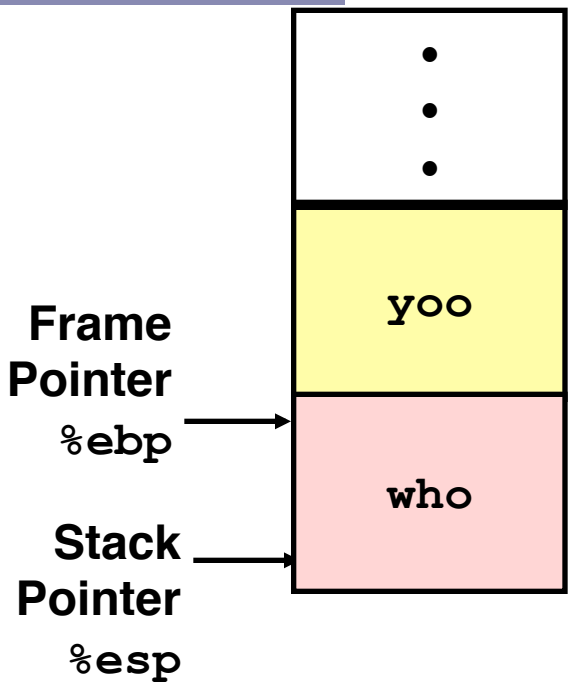


Stack operation



Call Chain

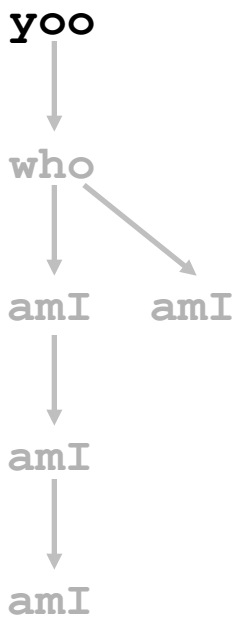


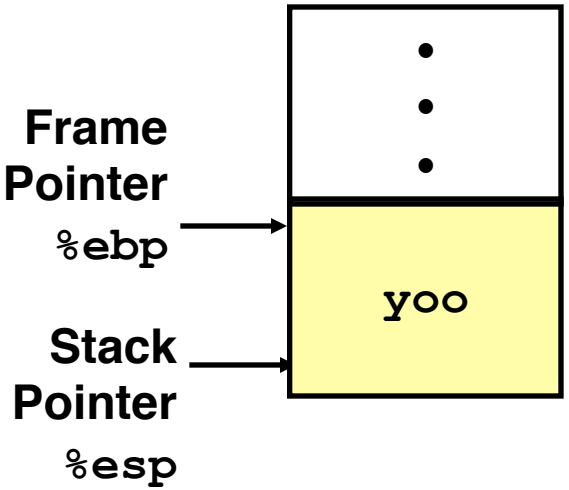


Stack operation

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

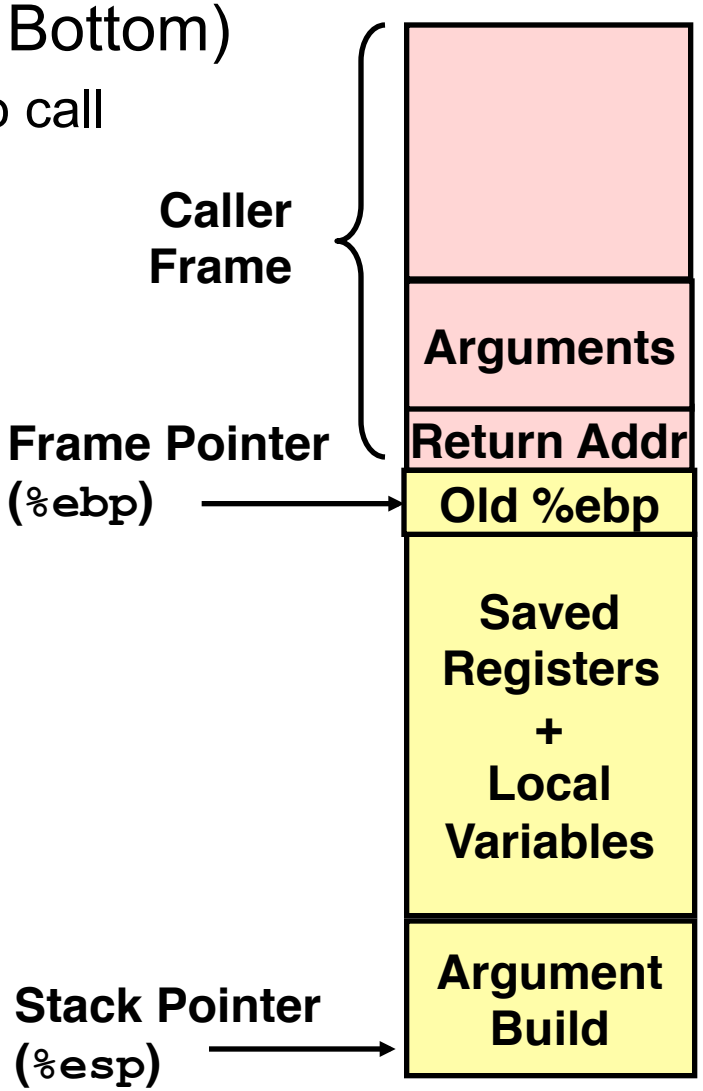
Call Chain





IA32/Linux stack frame

- Current stack frame ("Top" to Bottom)
  - Parameters for function about to call
    - "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



Revisiting swap

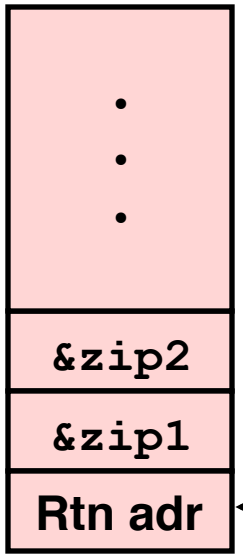
```
int zip1 = 15213;
int zip2 = 91125;

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

Calling `swap` from `call_swap`

```
call_swap:
    pushl   %ebp
    movl    %esp,
    subl    $8, %
    movl    $zip2, 4(%esp)
    movl    $zip1, (%esp)
    call    swap
    . . .
```

Global variable

```
void swap(int *xp, int *yp)
...
```



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

    popl %ebx
    popl %ebp
    ret
```

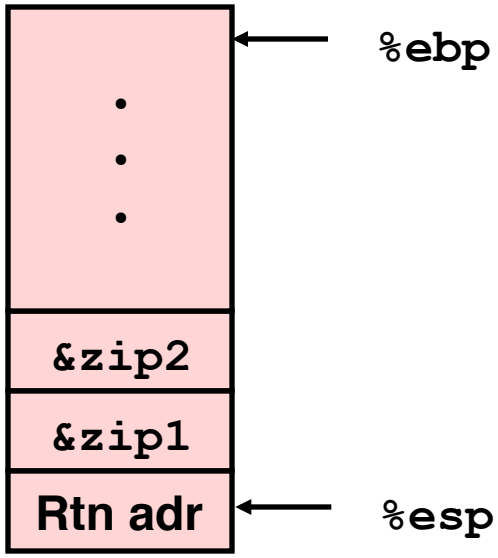
Set Up

Body

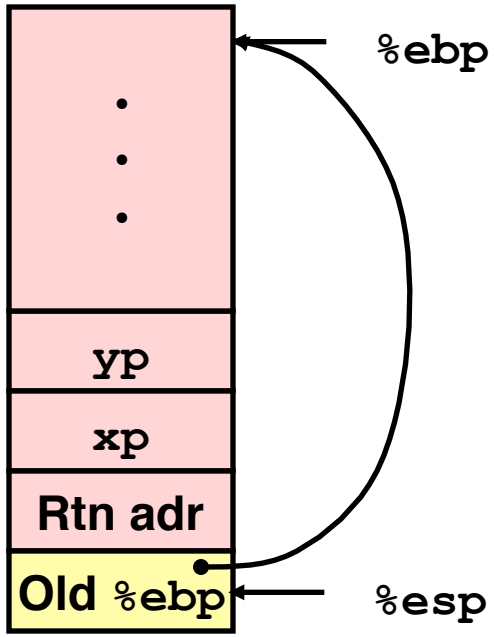
Finish

swap Setup #1

Entering Stack

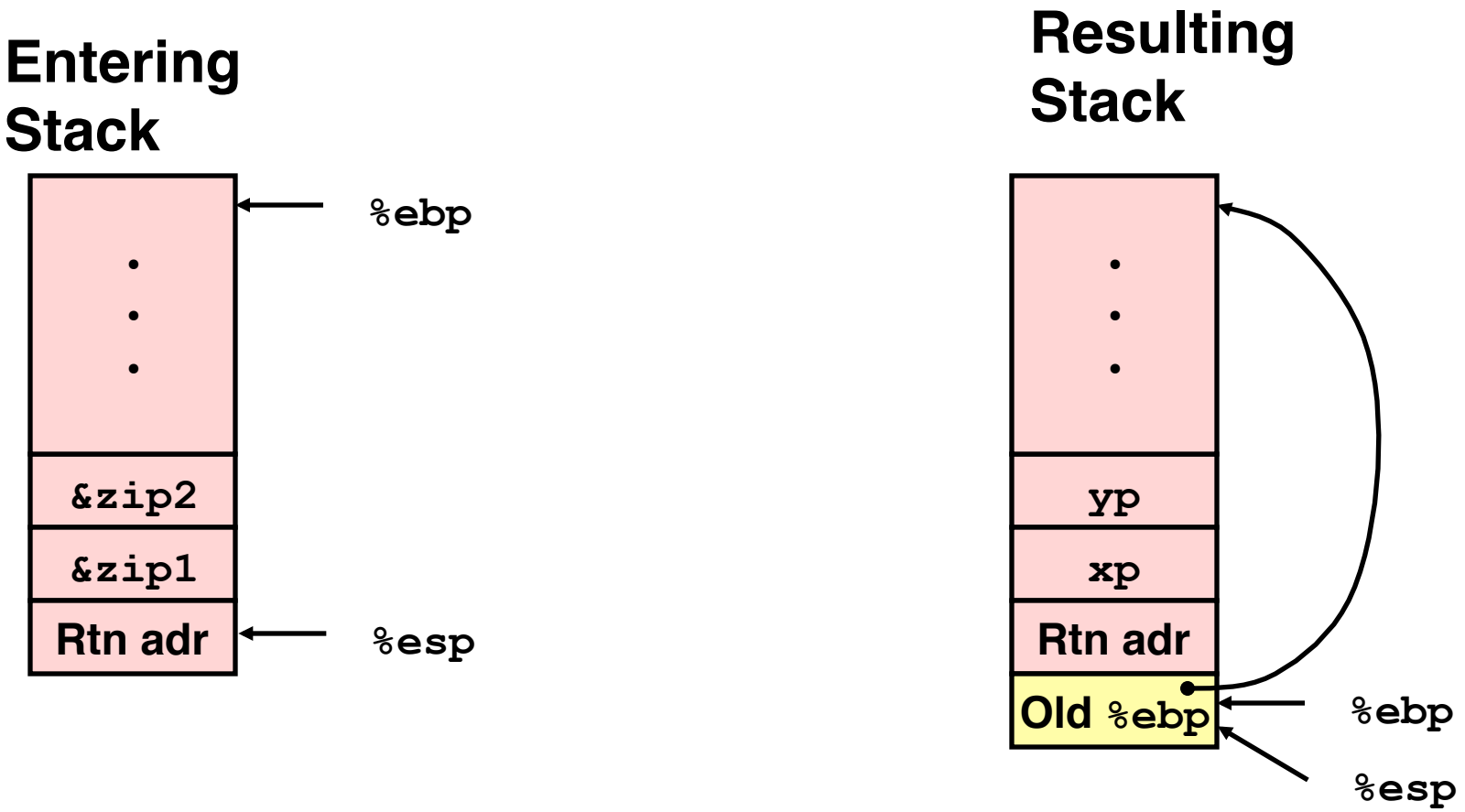


Resulting Stack



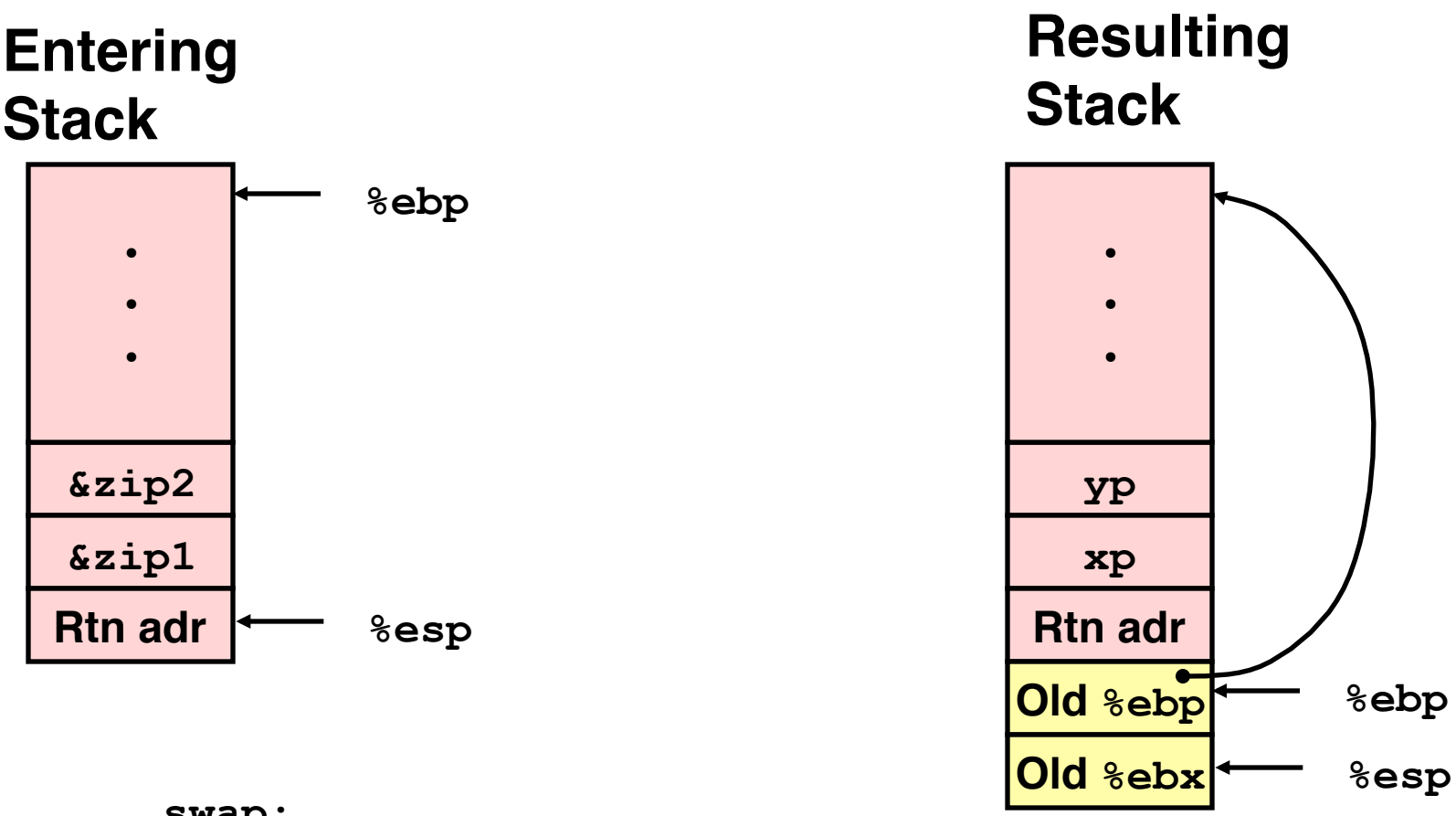

```
swap:
    pushl %ebp          Save old %ebp
    movl %esp,%ebp      Set %ebp as frame pointer
    pushl %ebx          Save %ebx
```

swap Setup #2



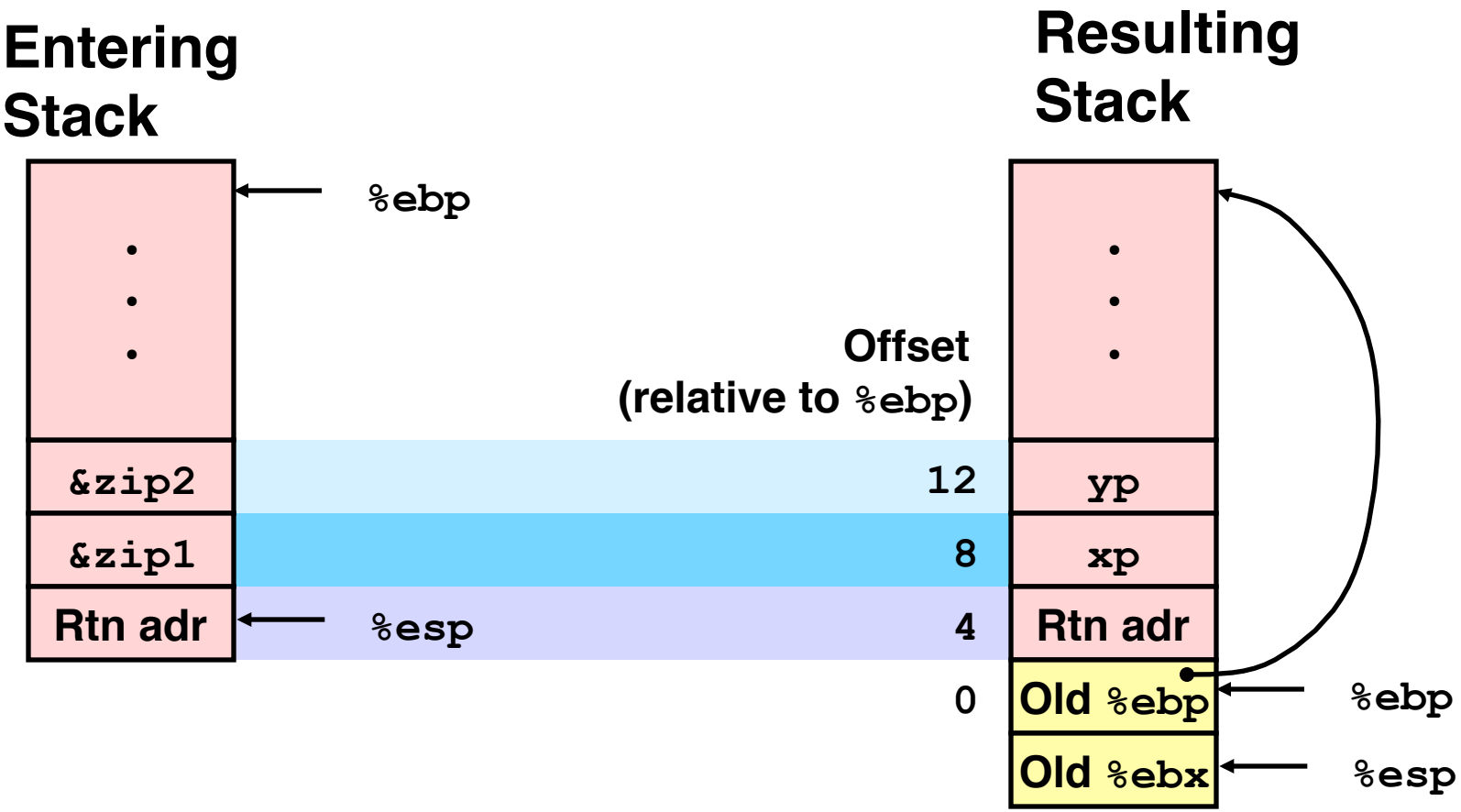

```
swap:
    pushl %ebp          Save old %ebp
    movl %esp,%ebp      Set %ebp as frame pointer
    pushl %ebx          Save %ebx
```

swap Setup #3




```
swap:
    pushl %ebp          Save old %ebp
    movl %esp,%ebp      Set %ebp as frame pointer
    pushl %ebx          Save %ebx
```

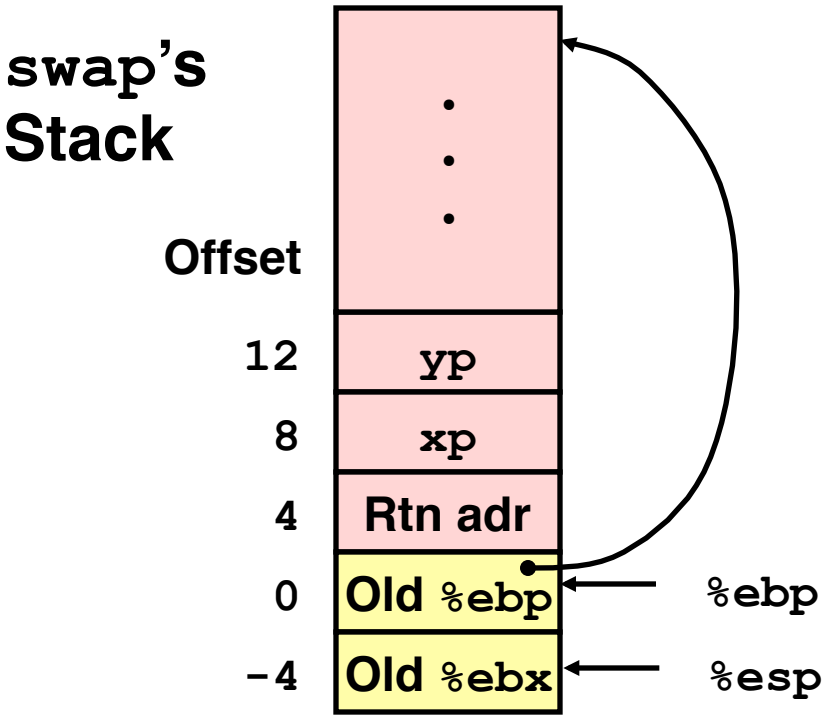
Effect of swap setup

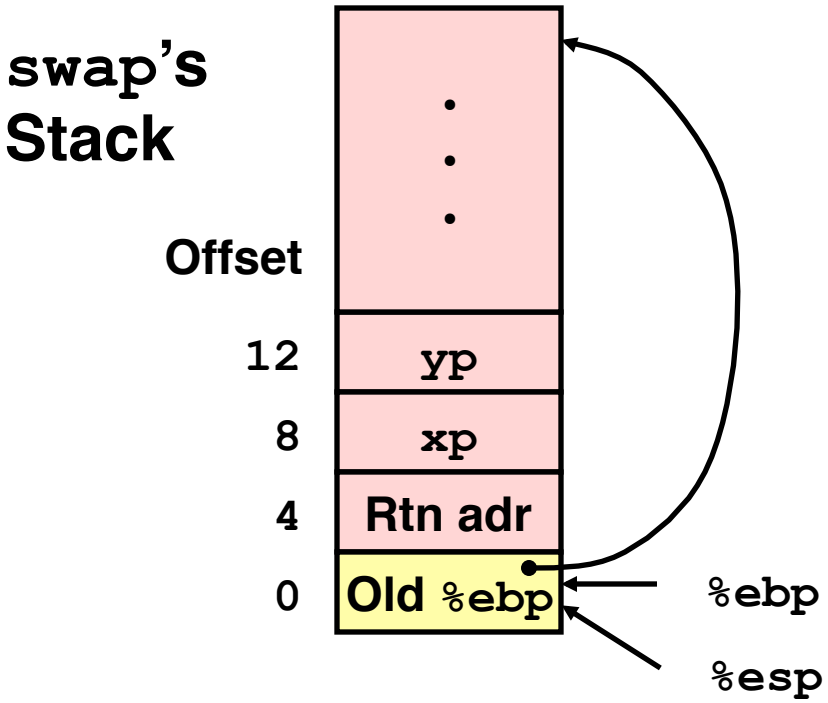



```
movl 12(%ebp),%ecx # get yp
movl 8(%ebp),%edx  # get xp
. . .
```

Body

swap Finish #1

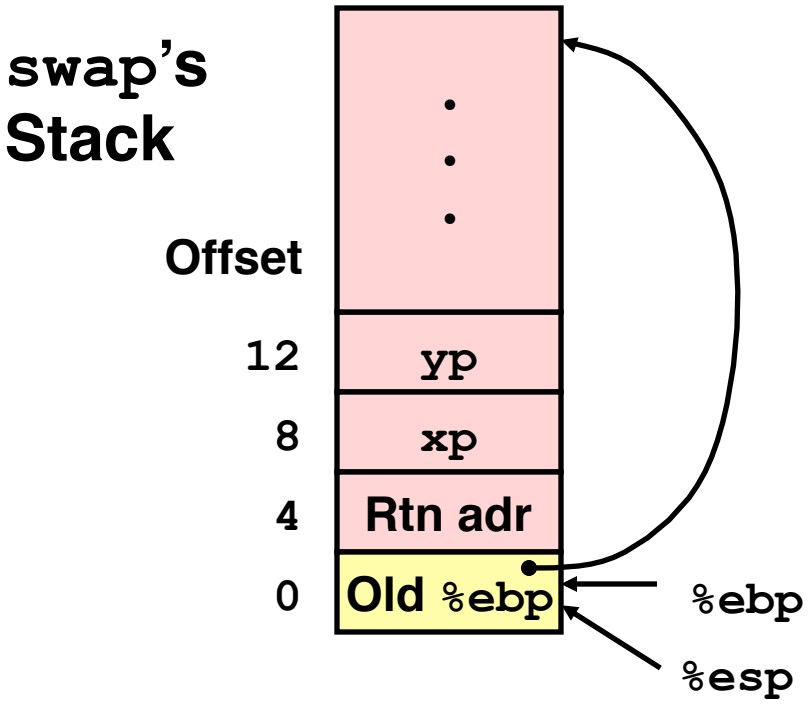


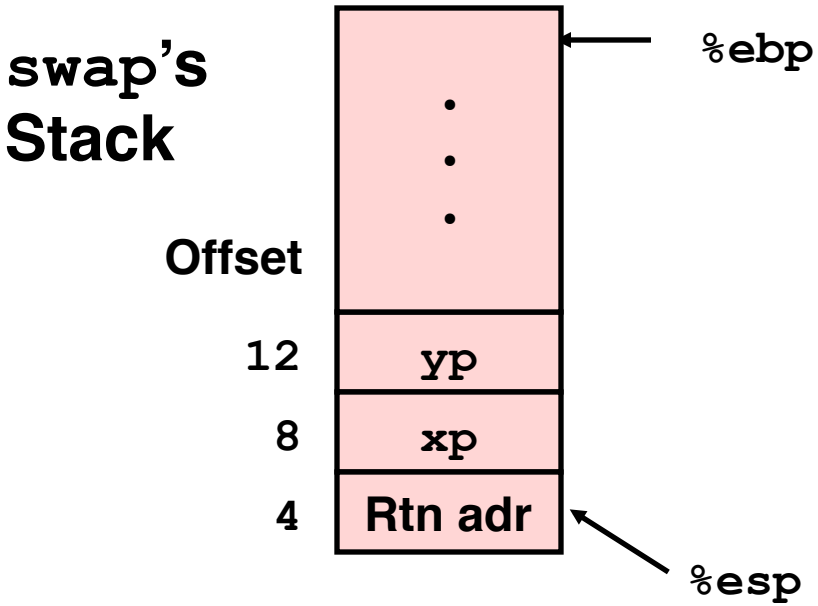



```
popl %ebx
popl %ebp
ret
```

- Observation
  - Saved & restored register `%ebx`

swap Finish #2



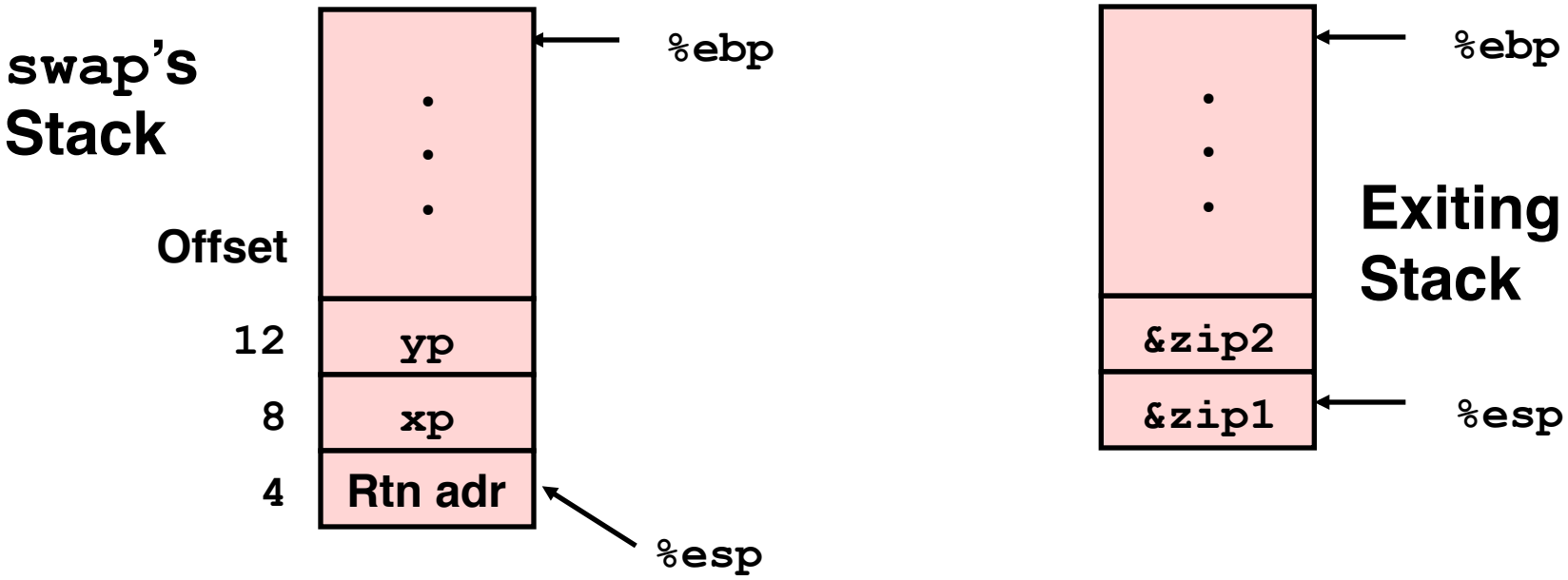


- Restore saved `%ebp` and … set stack ptr to end of caller's frame

```
popl %ebx
popl %ebp
ret
```

Pop address from stack & jump there

swap Finish #3

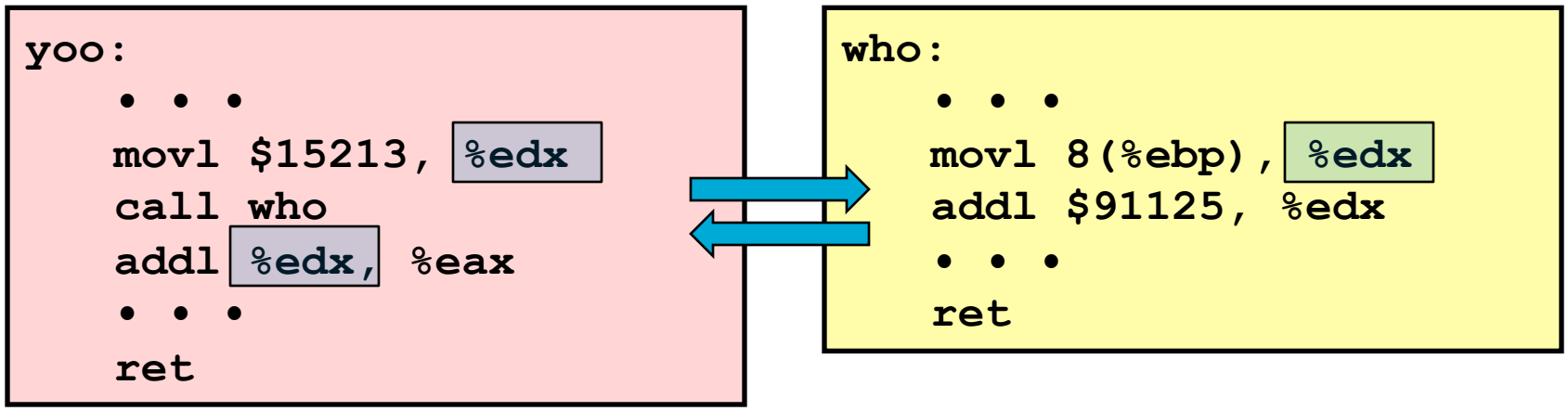


- Observation
  - Saved & restored register `%ebx`
  - Didn't do so for `%eax`, `%ecx`, or `%edx`

```
popl %ebx
popl %ebp
ret
```

Register saving conventions

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



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

Register saving conventions

- When procedure `yoo` calls `who`:
  - `yoo` is the *caller*, `who` is the *callee*
- Can a 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

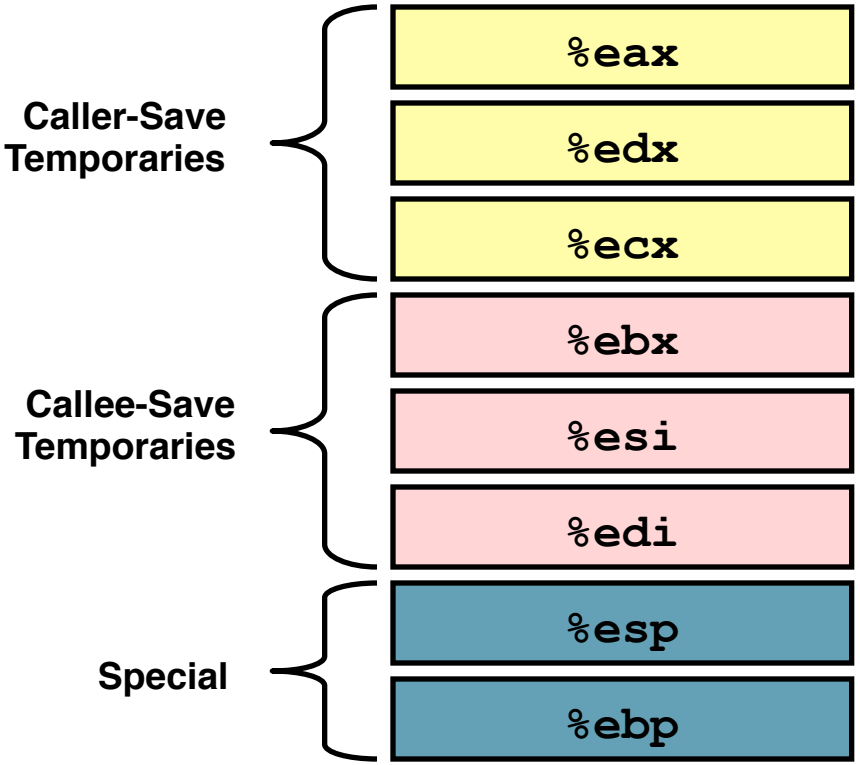
IA32/Linux register usage

- Integer registers
  - Two have special uses

    `%ebp`, `%esp`
  - Three managed as callee-save

    `%ebx`, `%esi`, `%edi`
    - Old values saved on stack prior to 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



Recursive factorial

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

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

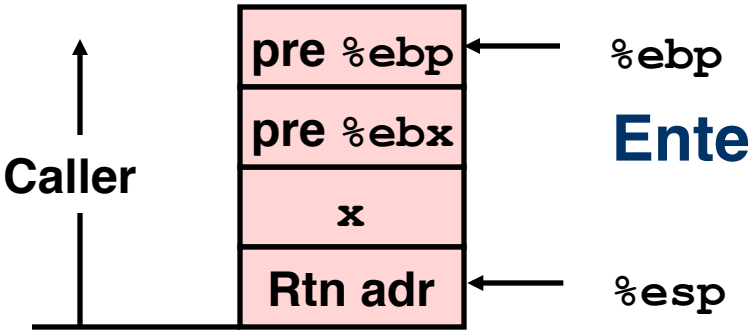
```
.globl rfact
        .type   rfact, @function
rfact:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        subl    $4, %esp
        movl    8(%ebp), %ebx
        movl    $1, %eax
        cmpl    $1, %ebx
        jle     .L3
        leal    -1(%ebx), %eax
        movl    %eax, (%esp)
        call    rfact
        imull   %ebx, %eax
.L3:
        addl    $4, %esp
        popl    %ebx
        popl    %ebp
        ret
```

Recursive factorial

```
.globl rfact
        .type   rfact, @function
rfact:
        pushl   %ebp            Save old %ebp
        movl    %esp, %ebp      Set %ebp as frame pointer
        pushl   %ebx            Save old %ebx
        subl    $4, %esp        Allocate 4B in stack
        movl    8(%ebp), %ebx   Get x
        movl    $1, %eax        result=1
        cmpl    $1, %ebx        Compare n:1
        jle     .L3             If <= goto done
        leal    -1(%ebx), %eax  Compute n - 1
        movl    %eax, (%esp)    Store it at top of stack
        call    rfact           Call rfact
        imull   %ebx, %eax      Compute result = return * n
.L3:                            done
        addl    $4, %esp        Deallocate 4B
        popl    %ebx            Restore %ebx
        popl    %ebp            Restore %ebp
        ret                     Return result
```

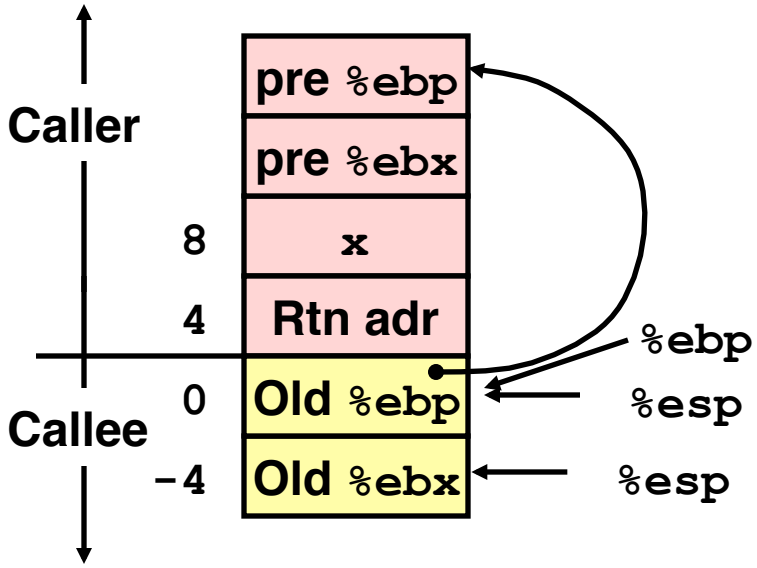
```
int rfact(int x)
{
  int result;
  if (x <= 1)
    result = 1;
  else
    result = n* rfact(x-1);
  return result;
}
```

Rfact stack setup



Entering Stack

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



Rfact body

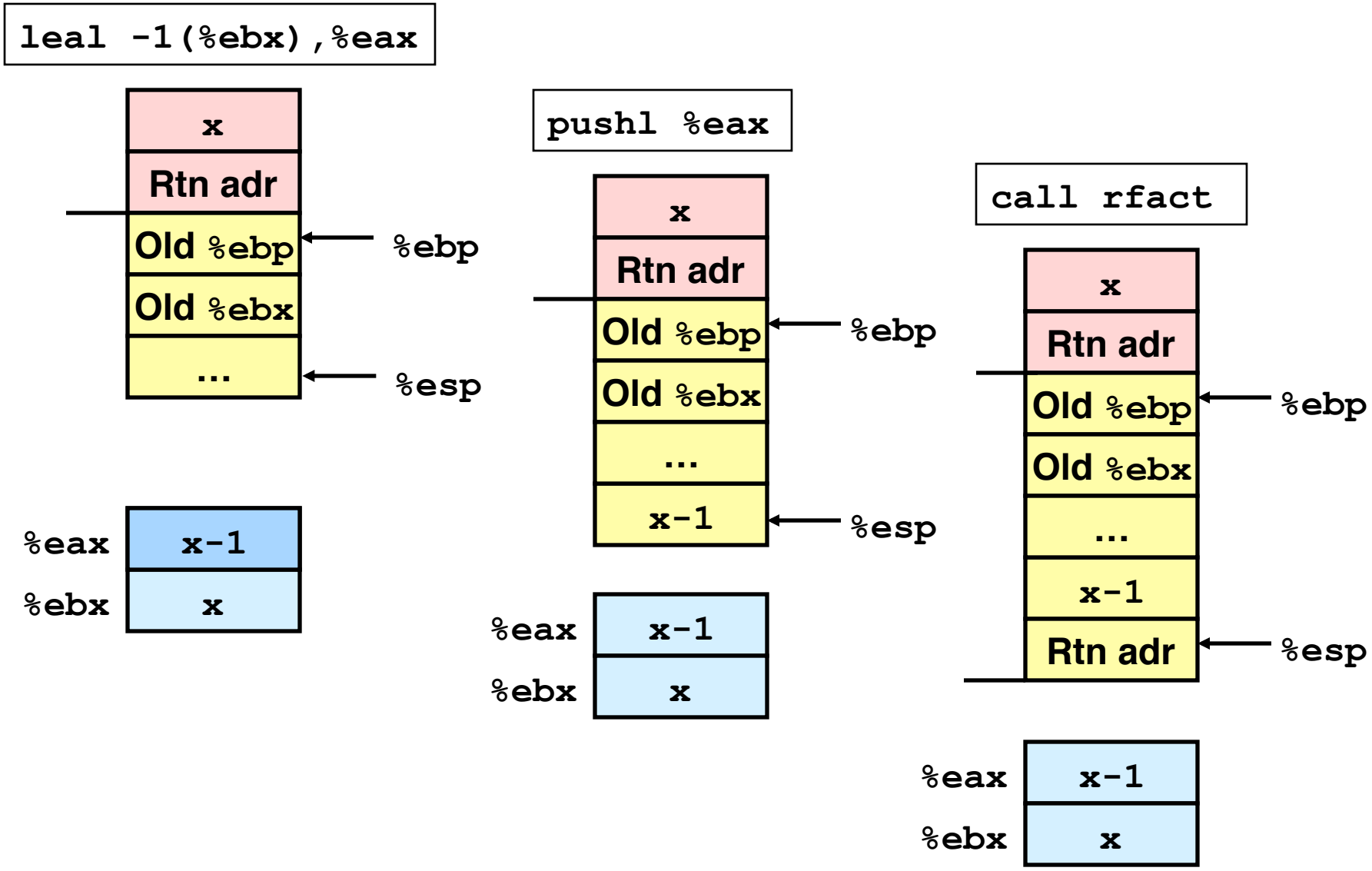
Recursion

```
subl    $4, %esp          Allocate 4B on stack
movl    8(%ebp), %ebx     Get x
movl    $1, %eax          result = 1
cmpl    $1, %ebx          Compare x: 1
jle     .L3               If <=, goto done
leal    -1(%ebx), %eax    Compute x-1
movl    %eax, (%esp)      Store at top of stack
call    rfact             Call rfact(x-1)
imull   %ebx, %eax        Compute result =
                                return value * x
```

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

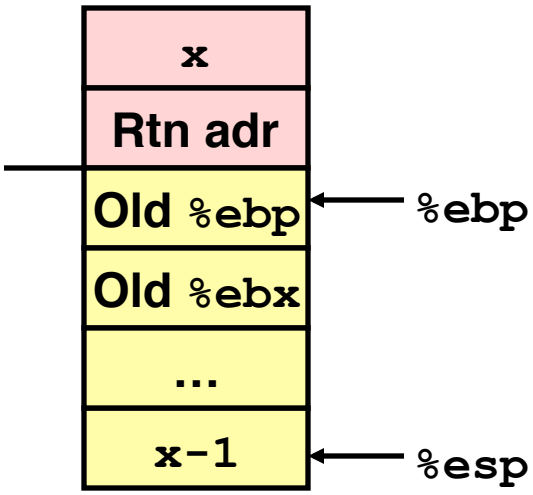
- Registers
  - `%ebx` Stored value of x
  - `%eax`
    - Temporary value of `x-1`
    - Returned value from `rfact(x-1)`
    - Returned value from this call

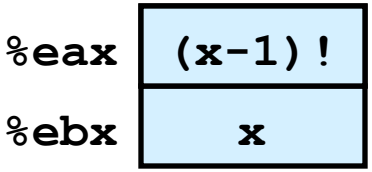
Rfact recursion



Rfact result

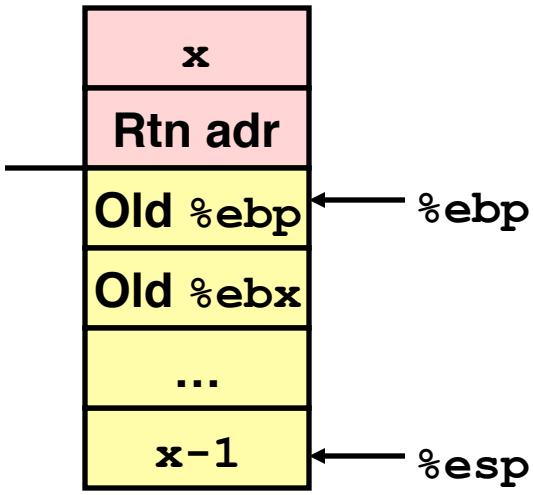
Return from Call

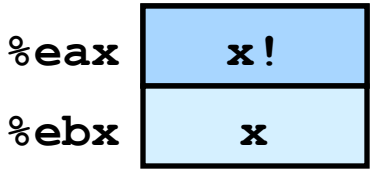




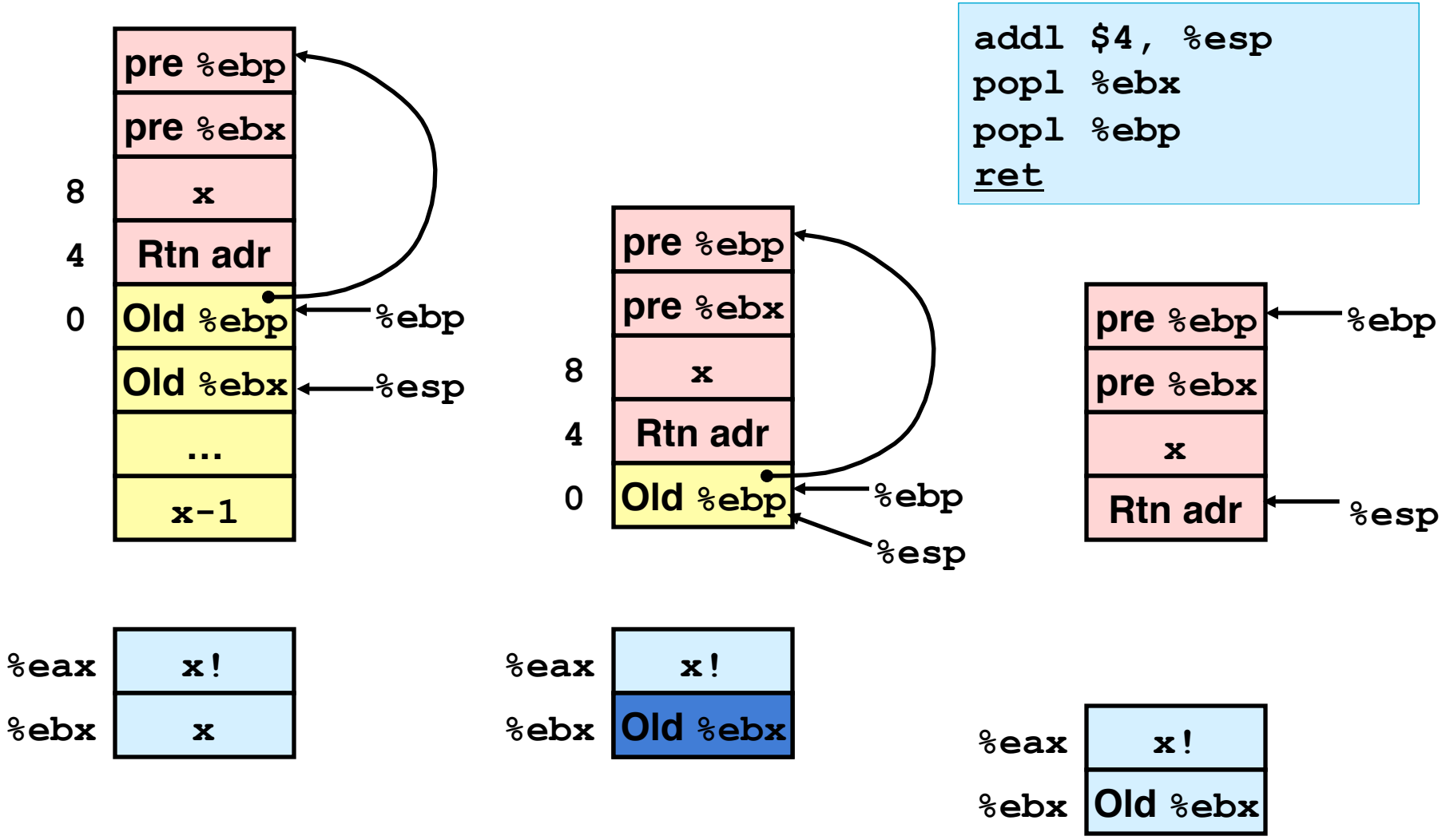
Assume that `rfact(x-1)` returns `(x-1)!` in register `%eax`

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





Rfact completion



Summary

- The stack makes recursion work
  - Private storage for each instance of procedure call
    - Instantiations don't clobber each other
    - Addressing of locals + arguments can be 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
    - %ebp and %esp
  - Stack frame organization conventions