# Lecture 9:
# Assembly Cont…

# Announcements

- Upcoming Midterm Exam
  - July 22nd , This Wednesday (in two days)
  - Format: Online 80 minute exam via Sakai Quizzes/Exams
  - Sample questions released to help study
  - We'll talk more about the details in the next slide
- Project 2
  - Due in one week (July 28th)
  - Piazza, Piazza, Piazza: keep discussing and helping each other out
- Project 1 Grades
  - Will be released by this Wednesday
- Lecture Today:
  - Finish up the rest of assembly
- Rest of lecture time / recitation:
  - Any questions on assembly
  - Any questions on any of the course material up until this point in preparation for Midterm

# Midterm Details

- This Wednesday, July 22nd
- Covers all material up until today, July 20th
- Via Sakai Quizzes/Exams
- Exam will be open between:
  - Open date: Wednesday, July 22nd 8:00am EST
  - Close date: Thursday, July 23rd 8:00am EST
  - 24-hour open time period to account for time zone differences
- 80 minutes to complete the exam
- No proctoring
- Approximately ~30-40 Questions
  - Most multiple choice/short open-ended (Should not take more than a minute per question)
  - ~4/5 longer open-ended questions
- No open book
- Open note/Cheat sheet allowed
  - Be mindful of time
- Can use scrap paper/calculator
- You must take the exam and submit it before the closing date
- Please schedule accordingly based on your time zone

# Today's Outline

- Finishing Up Stack Procedure Control

# Recap:

# Conditional Branching: Jumping

- jX Instructions
  - Jump to different part of code depending on condition codes

| jX | Condition | Description |
|---|---|---|
| `jmp` | `1` | Unconditional |
| `je` | `ZF` | Equal / Zero |
| `jne` | `~ZF` | Not Equal / Not Zero |
| `js` | `SF` | Negative |
| `jns` | `~SF` | Nonnegative |
| `jg` | `~(SF^OF)&~ZF` | Greater (Signed) |
| `jge` | `~(SF^OF)` | Greater or Equal (Signed) |
| `jl` | `(SF^OF)` | Less (Signed) |
| `jle` | `(SF^OF)\|ZF` | Less or Equal (Signed) |
| `ja` | `~CF&~ZF` | Above (unsigned) |
| `jb` | `CF` | Below (unsigned) |

# Conditionals Branching: Expressing with Goto Code

C allows `goto` statement

Jump to position designated by label

```
long absdiff
   (long x, long y)
{
     long result;
     if (x > y)
          result = x-y;
     else
          result = y-x;
     return result;
}
```

```
long absdiff_j
   (long x, long y)
{
     long result;
     int ntest = x <= y;
     if (ntest)
        goto Else;
     result = x-y;
     goto Done;
 Else:
     result = y-x;
 Done:
     return result;
}
```

# “Do-While” Loop Compilation

## Goto Version

```
long pcount_goto
  (unsigned long x) {
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

| Register | Use(s) |
|---|---|
| `%rdi` | Argument `x` |
| `%rax` | `result` |

```
        movl    $0, %eax      #  result = 0
    .L2:                      # loop:
        movq    %rdi, %rdx
        andl    $1, %rdx      #  t = x & 0x1
        addq    %rdx, %rax    #  result += t
        shrq    %rdi          #  x >>= 1
        jne     .L2           #  if (x) goto loop
        ret
```

# Loops: General "Do-While" Translation

**C Code**

```
do
  Body
  while (Test);
```

**Goto Version**

```
loop:
  Body
  if (Test)
    goto loop
```

Body:

```
{
  Statement1;
  Statement;
    …
  Statementn;
}
```

# Loops: General "While" Translation #1

- "Jump-to-middle" translation
- Used with `–Og`

**While version**

```
while (Test)
  Body
```

**Goto Version**

```
  goto test;
loop:
  Body
test:
  if (Test)
    goto loop;
done:
```

# Loops: General "While" Translation #2

**While version**

```
while (Test)
   Body
```

- "Do-while" conversion
- Used with –O1

**Do-While Version**

```
  if (!Test)
    goto done;
  do
    Body
    while(Test);
done:
```

**Goto Version**

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

# “For” Loop -> While Loop

**For Version**

```
for (Init; Test; Update)
        Body
```

**While Version**

```
Init;
while (Test) {
    Body
    Update;
}
```

# Switch Statement Example

```
long switch_eg
   (long x, long y, long z)
{
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

- Multiple case labels
  - Here: 5 & 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4

# Jump Table Structure

### Switch Form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

### Jump Table

```
jtab: Targ0
      Targ1
      Targ2
        •
        •
        •
      Targn-1
```

### Jump Targets

```
Targ0:   Code Block 0

Targ1:   Code Block 1

Targ2:   Code Block 2

  •
  •
  •

Targn-1: Code Block n–1
```

### Translation (Extended C)

```
goto *JTab[x];
```

# Assembly Setup Explanation

- Table Structure
  - Each target requires 8 bytes
  - Base address at **`.L4`**
- Jumping
  - **Direct: `jmp .L8`**
    - Jump target is denoted by label **`.L8`**
  - **Indirect: `jmp *.L4(,%rdi,8)`**
    - Start of jump table: **`.L4`**
    - Must scale by factor of 8 (addresses are 8 bytes)
    - Fetch target from effective address **`.L4 + x*8`**
    - Only for $0 \le$ **`x`** $\le 6$

**Jump table**

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8 # x = 0
  .quad     .L3 # x = 1
  .quad     .L5 # x = 2
  .quad     .L9 # x = 3
  .quad     .L8 # x = 4
  .quad     .L7 # x = 5
  .quad     .L7 # x = 6
```

# Switch Cases Overview

```
    case 1:        // .L3
        w = y*z;
        break;
    case 2:        // .L5
        w = y/z;
        /* Fall Through */
    case 3:        // .L9
        w += z;
        break;
    case 5:
    case 6:        // .L7
        w -= z;
        break;
    default:       // .L8
        w = 2;
```

```
.L3:
   movq    %rsi,   %rax   # y
   imulq   %rdx,   %rax   # y*z
   ret
.L5:                      # Case 2
   movq     %rsi, %rax
   cqto
   idivq    %rcx          #  y/z
   jmp      .L6        #  goto merge
.L9:                      # Case 3
   movl     $1, %eax      #  w = 1
.L6:                      # merge:
   addq     %rcx, %rax #  w += z
   ret
.L7:                   # Case 5,6
  movl  $1, %eax     #  w = 1
  subq  %rdx, %rax #   w -= z
  ret
.L8:                   # Default:
  movl  $2, %eax     #  2
  ret
```

# Switch Cases Overview

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi    # x:6
    ja      .L8
    jmp     *.L4(,%rdi,8)
```

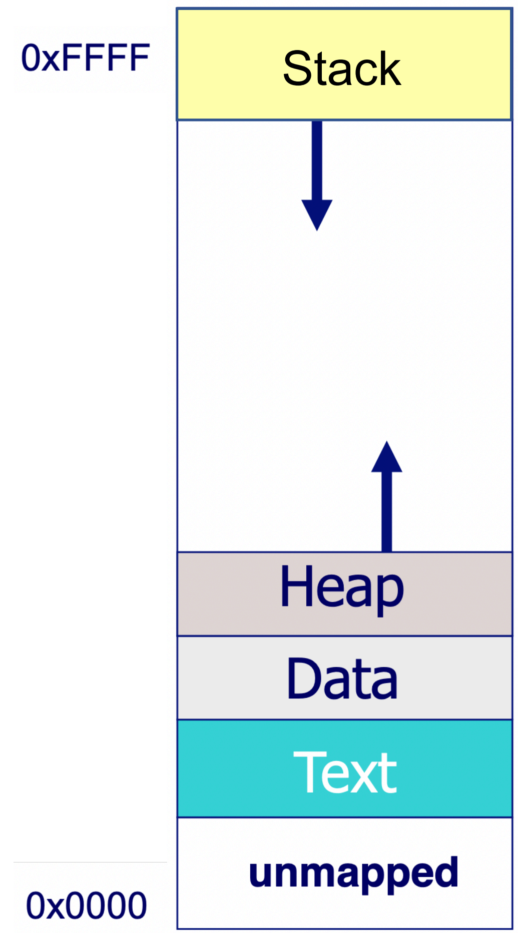
```
.section    .rodata
  .align 8
.L4:
  .quad     .L8 # x = 0
  .quad     .L3 # x = 1
  .quad     .L5 # x = 2
  .quad     .L9 # x = 3
  .quad     .L8 # x = 4
  .quad     .L7 # x = 5
  .quad     .L7 # x = 6
```

| Register | Use(s) |
|---|---|
| `%rdi` | Argument `x` |
| `%rsi` | Argument `y` |
| `%rdx` | Argument `z` |
| `%rax` | Return value |

```
.L3:
   movq    %rsi,   %rax   # y
   imulq   %rdx,   %rax   # y*z
   ret
.L5:                      # Case 2
   movq     %rsi, %rax
   cqto
   idivq    %rcx          #  y/z
   jmp      .L6         #  goto merge
.L9:                      # Case 3
   movl     $1, %eax      #  w = 1
.L6:                      # merge:
   addq     %rcx, %rax #  w += z
   ret
.L7:                   # Case 5,6
  movl   $1, %eax      #  w = 1
  subq   %rdx, %rax #  w -= z
  ret
.L8:                   # Default:
  movl   $2, %eax      #  2
  ret
```

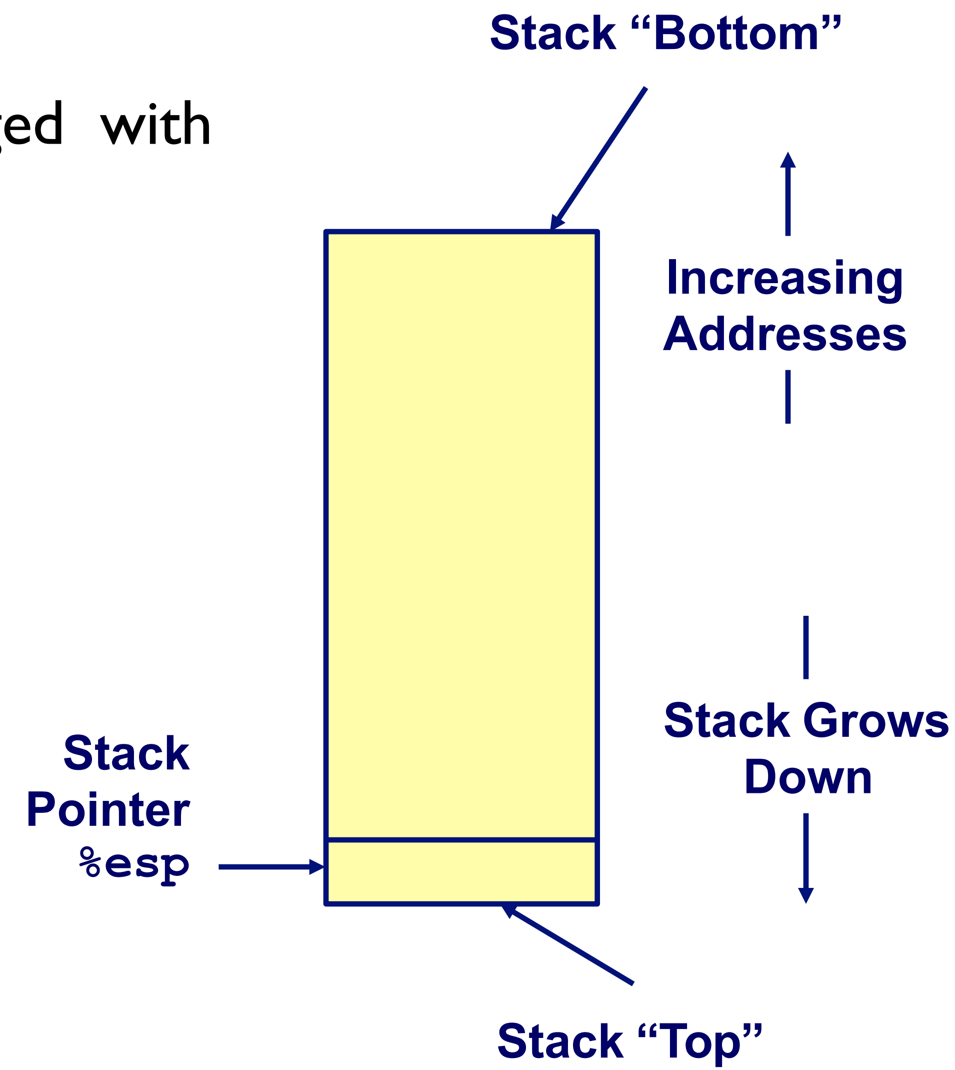
# Memory Segments

- Segments of an executable are laid out in memory
- An application/program's memory has 4 segments
  - Text: instructions of the program
  - Data: global and static data
  - Heap: dynamic allocation
  - Stack: function calls and local data
- Heap and stack Grow dynamically
  - Heap grows up
  - Stack grows down



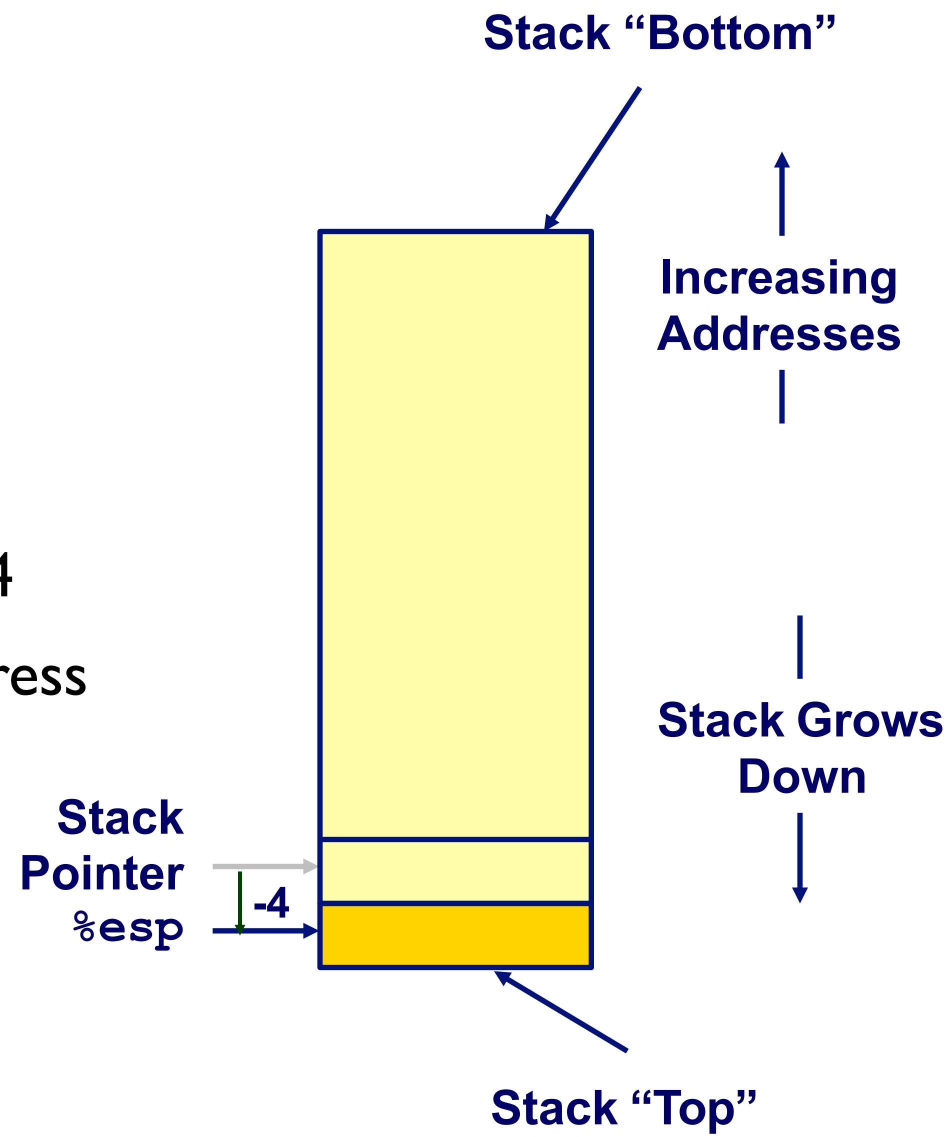
# x86 Stack

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



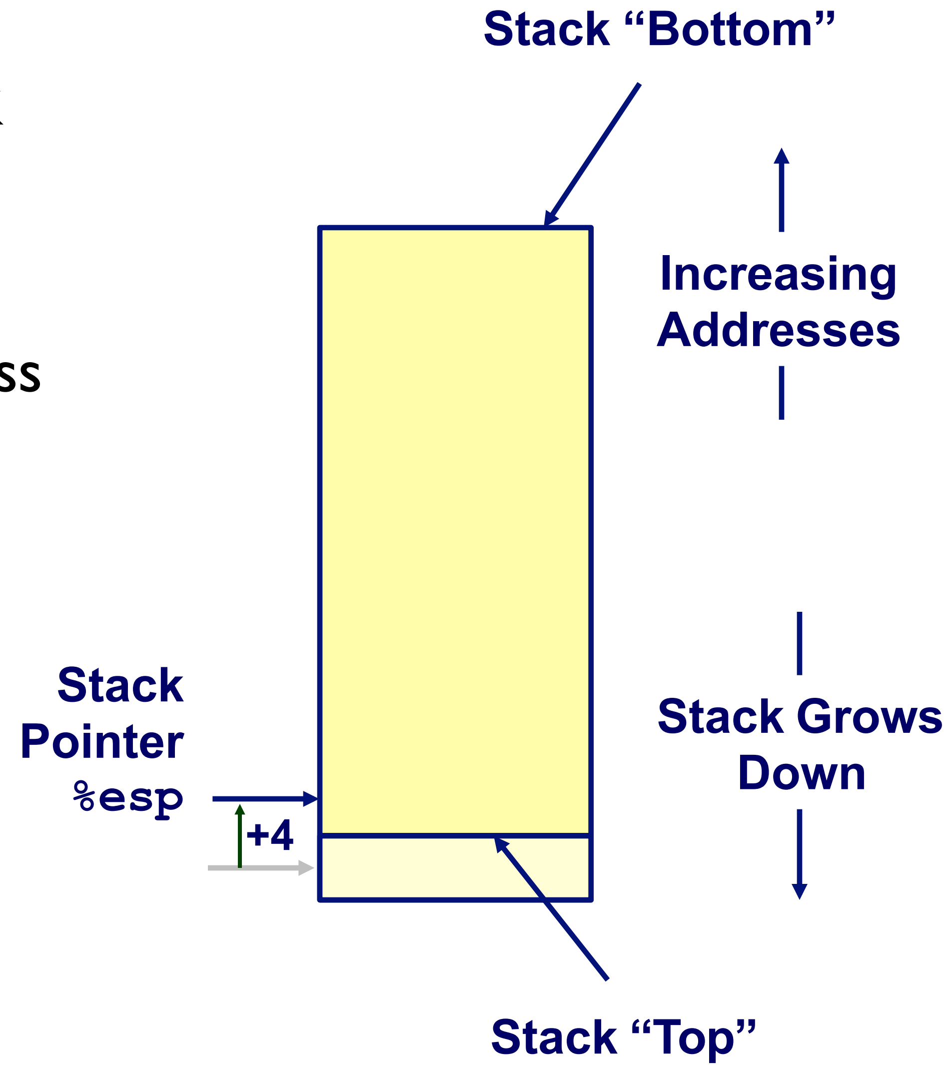
# x86 Stack: Pushing

- Pushing to the stack
- `pushl` *Src*
- What it does:
  - Fetch operand at *Src*
  - Decrement `%esp` by 4
  - Write operand at address given by `%esp`



# x86 Stack: Popping

- Popping from the stack
- `popl` *Dest*
- What it does
  - Read operand at address given by `%esp`
  - Increment `%esp` by 4
  - Write to *Dest*



# Procedure Control Flow

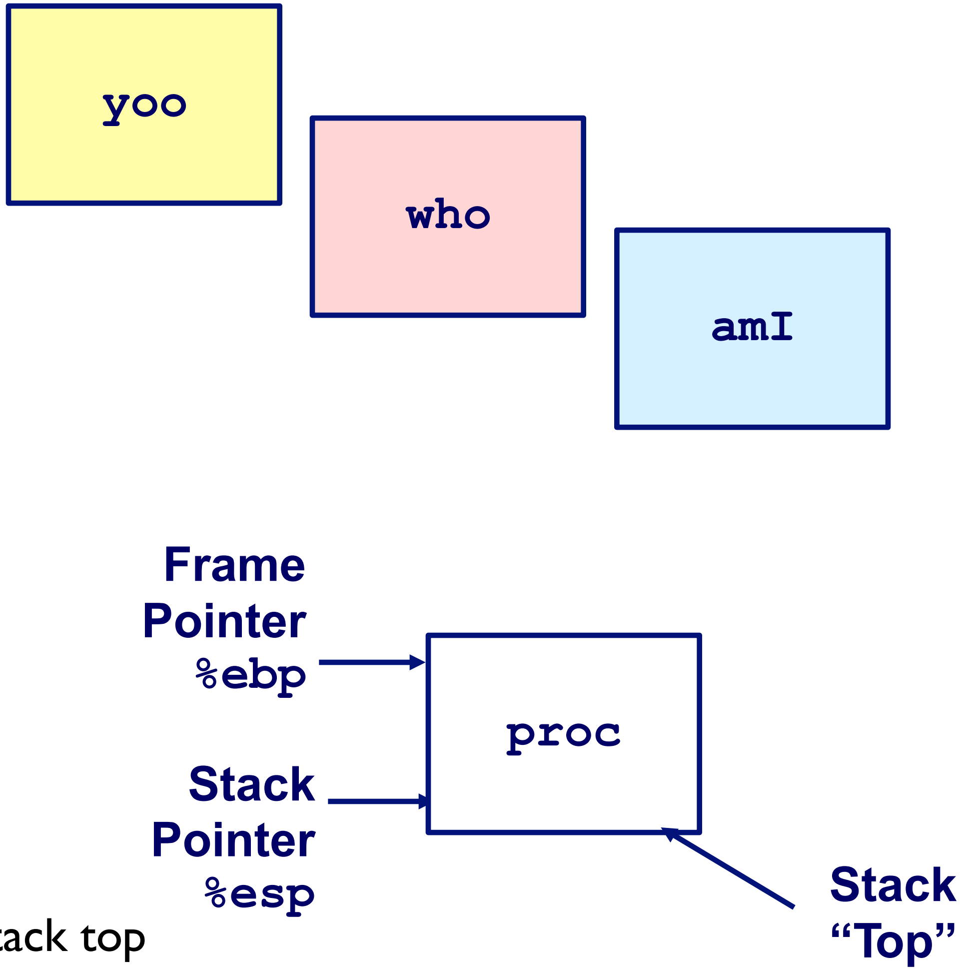
- Use stack to support procedure call and return
- A procedure call involves passing data and control from one part of a program to another
- Procedure call:
  - `call` *`label`*
  - Pushes return address on stack, then jump to *`label`*
- The return address is the address of instruction beyond `call`
- Example:

```
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

# 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



# Call Chain Example

## Code Structure

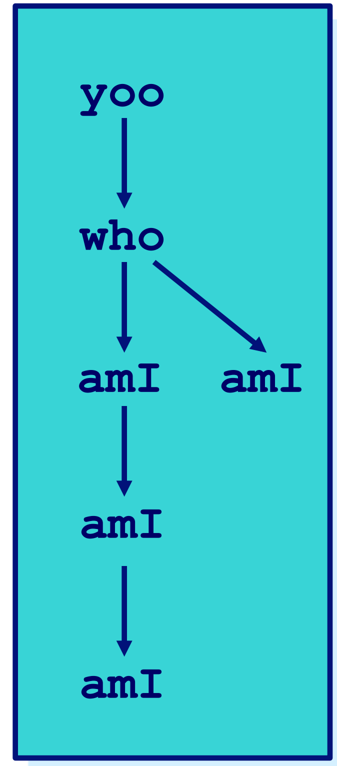
```
yoo(…)
{
    •
    •
    who();
    •
    •
}
```

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

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

- Procedure `amI` recursive

## Call Chain

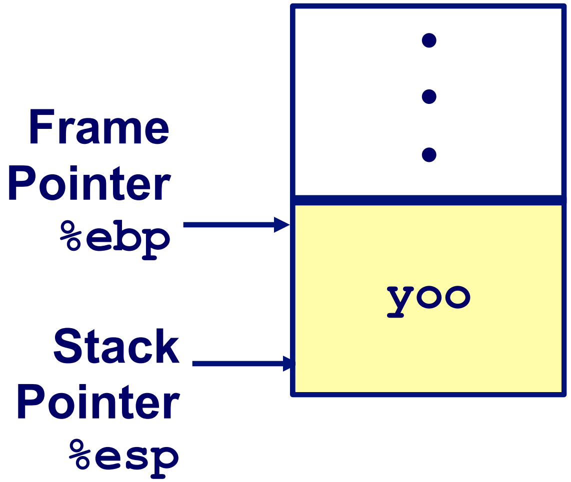


# Stack Operation

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

**Call Chain**

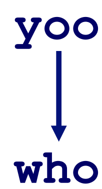
`yoo`

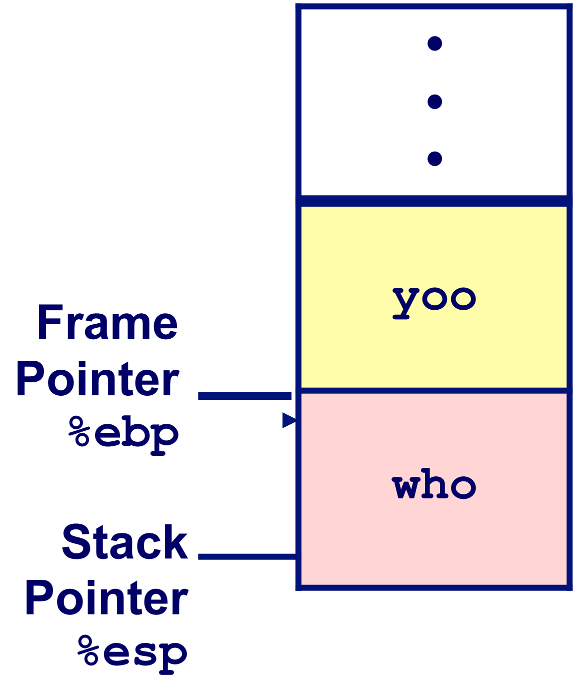


# Stack Operation

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

## Call Chain



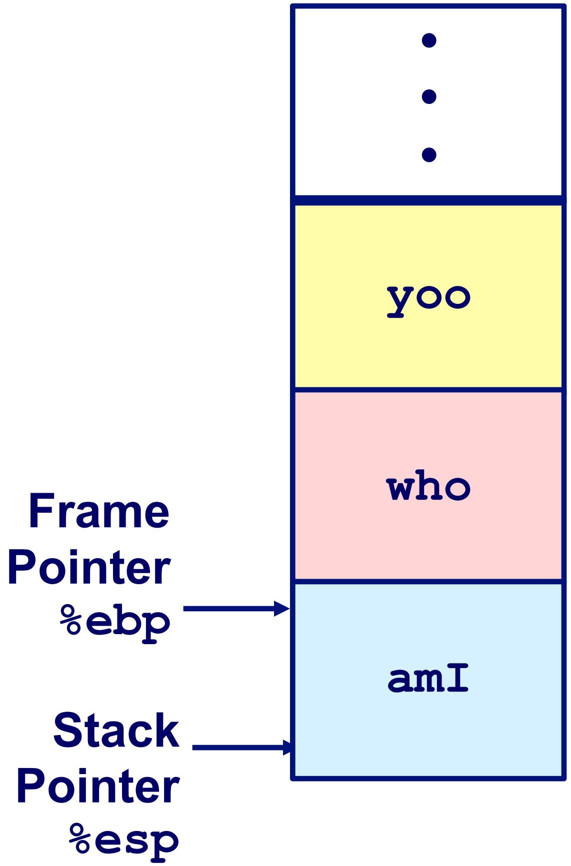



# Stack Operation

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

### Call Chain

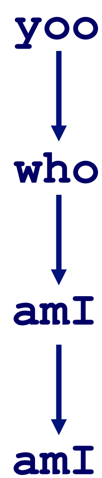
yoo
↓
who
↓
amI

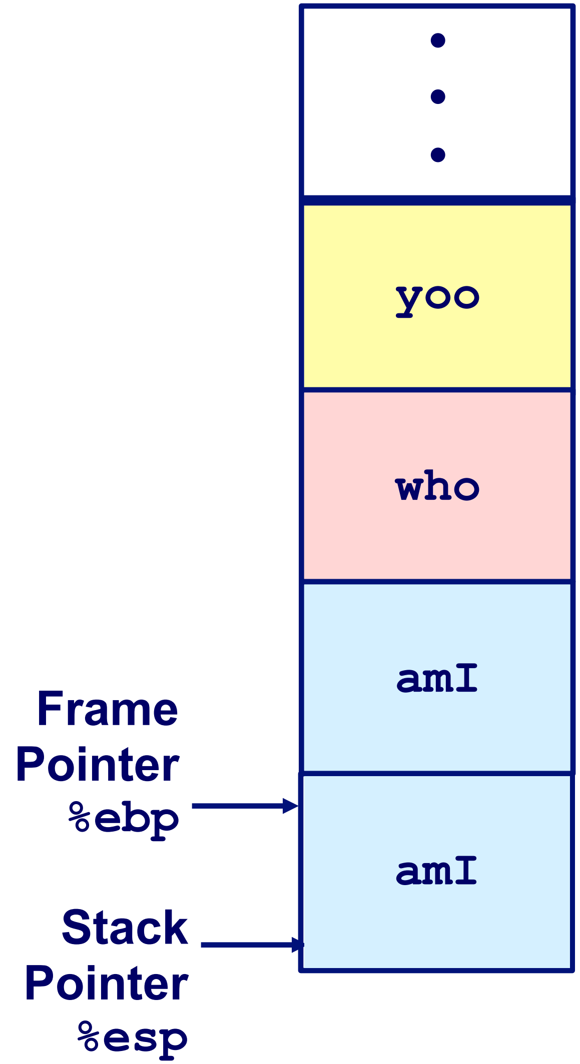


# Stack Operation

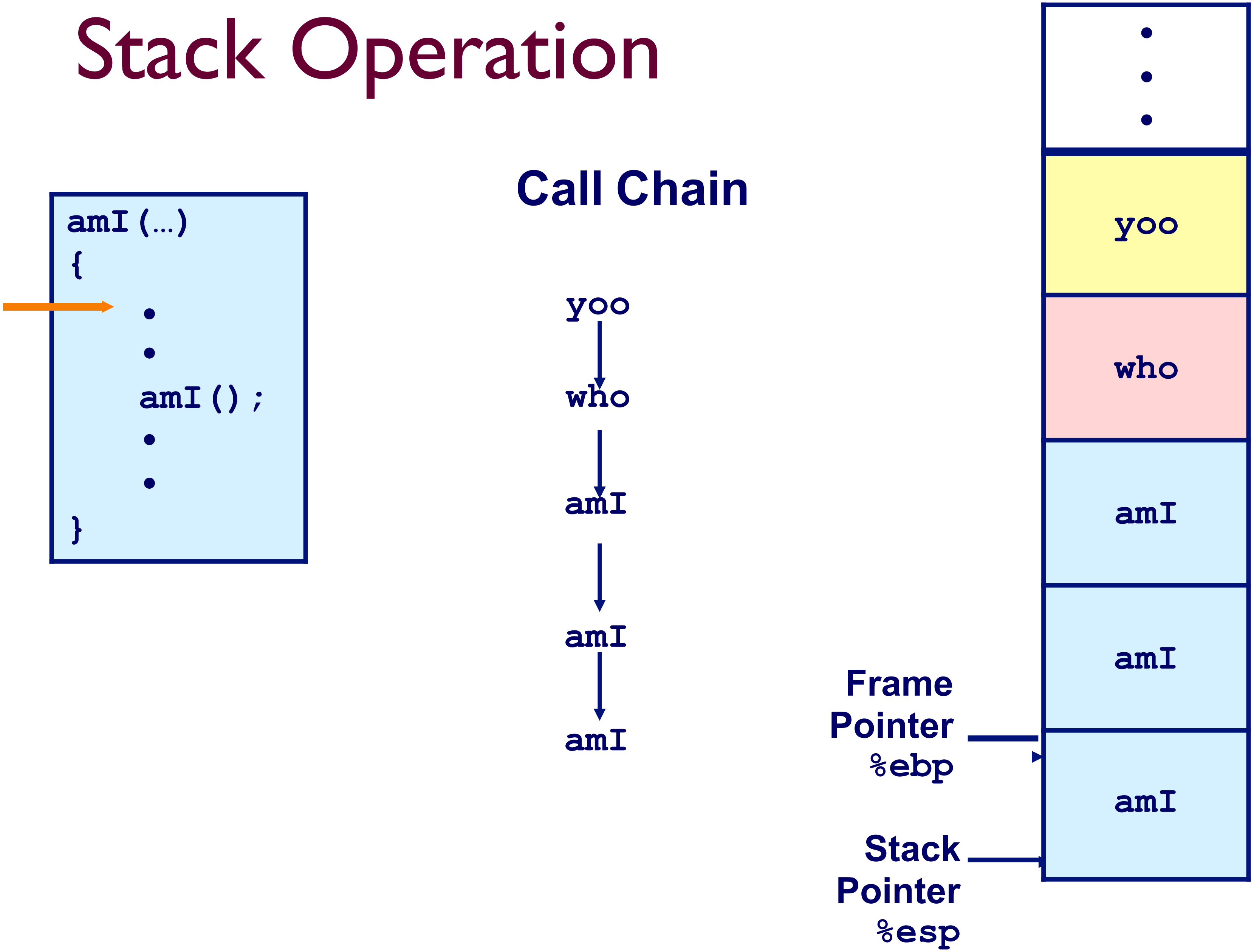
```
amI(…)
{
    •
    •
    amI();
    •
    •
}
```

## Call Chain



# Stack Operation

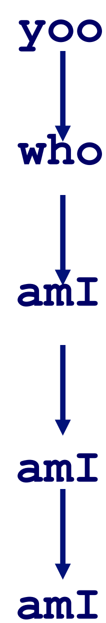


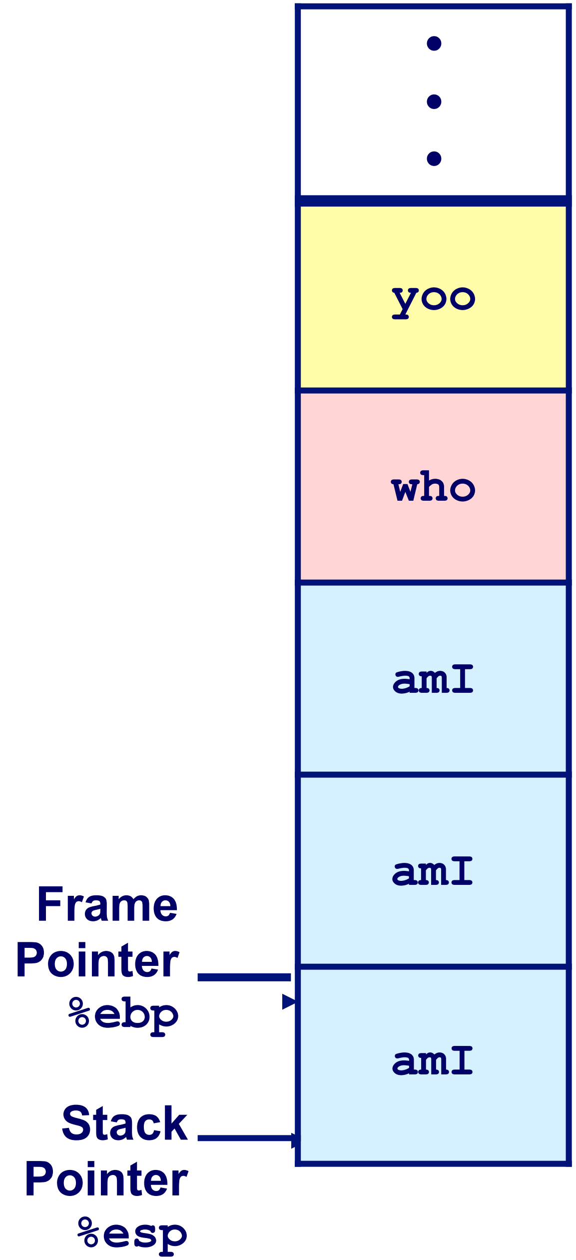
# Stack Operation

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

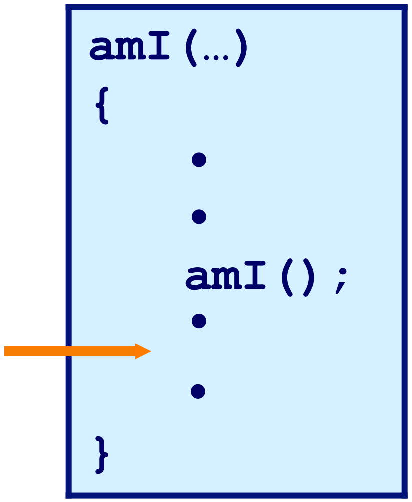
Lets say it ends recursive calling here and start to return

## Call Chain

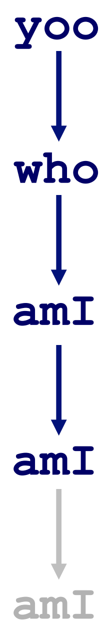


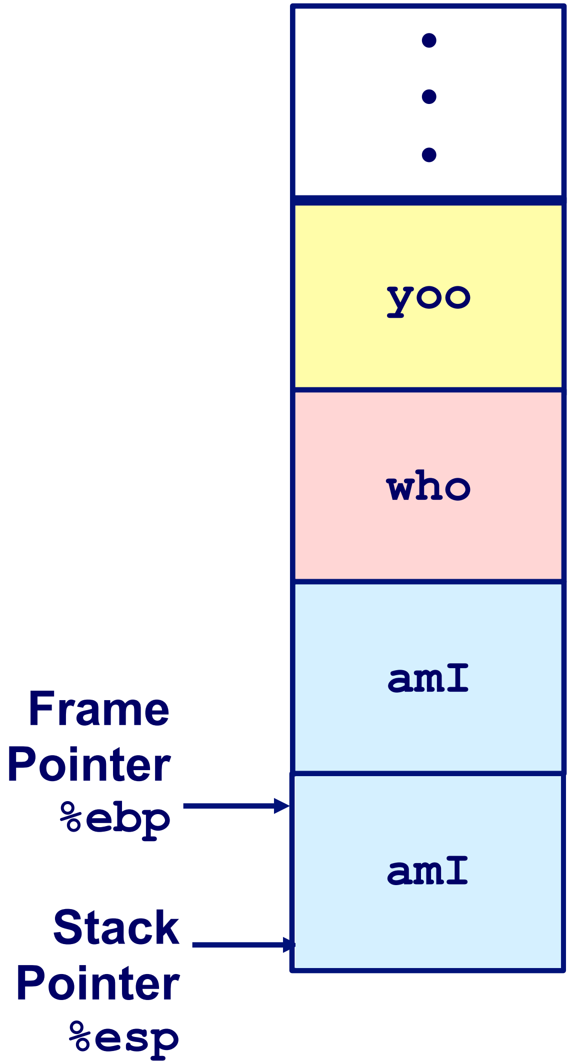



# Stack Operation




**Call Chain**

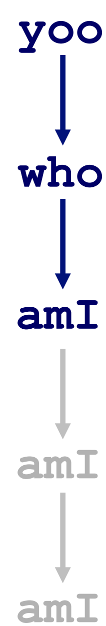


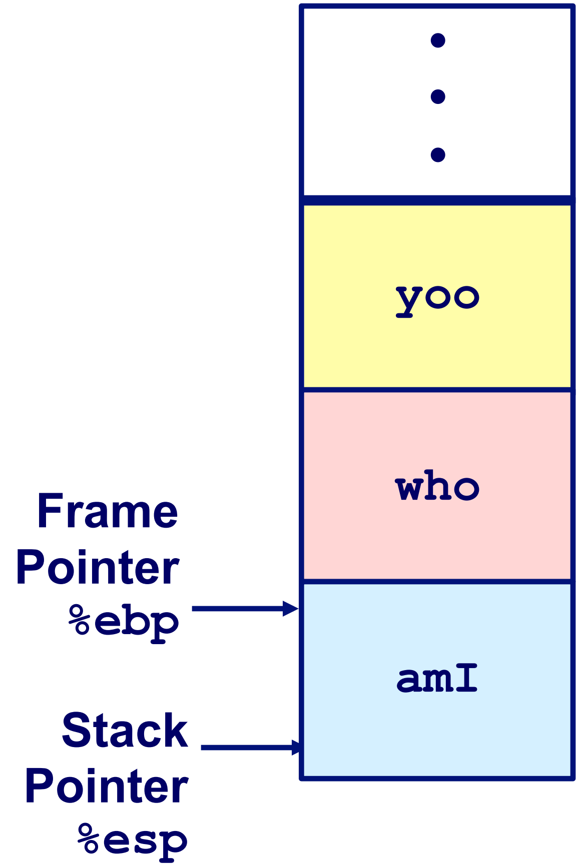



# Stack Operation

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

## Call Chain

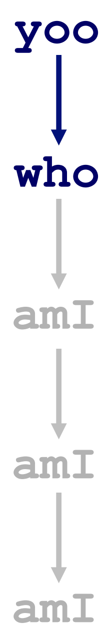


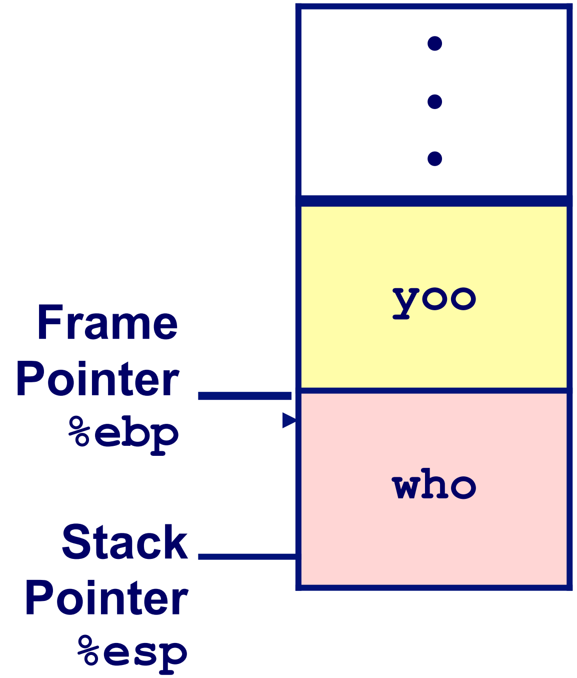



# Stack Operation

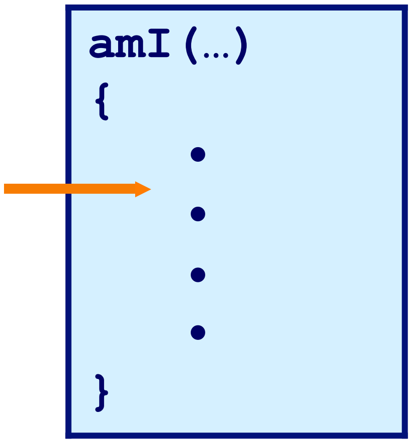
```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

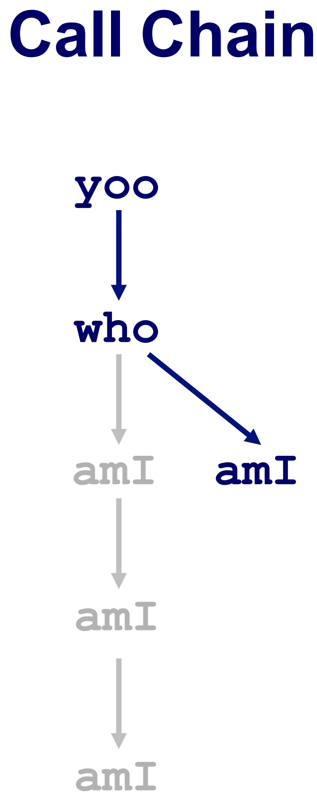
**Call Chain**

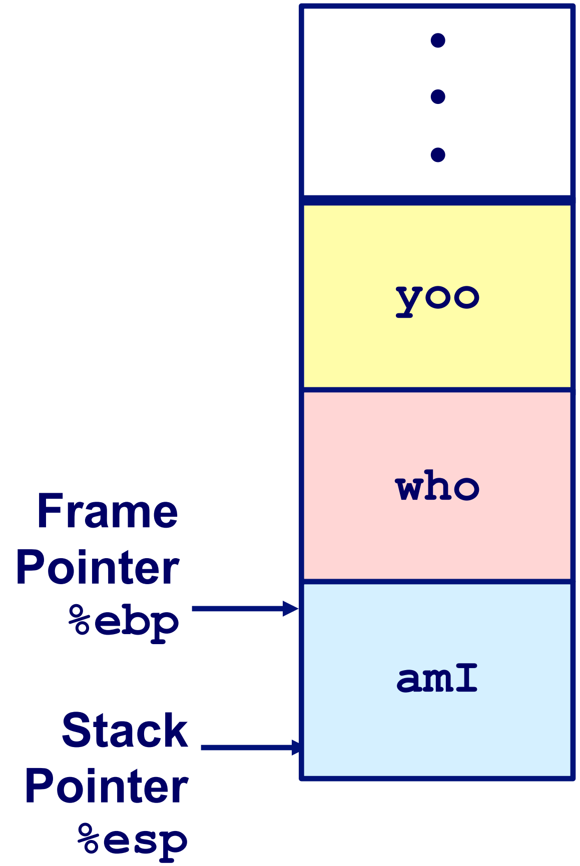


# Stack Operation



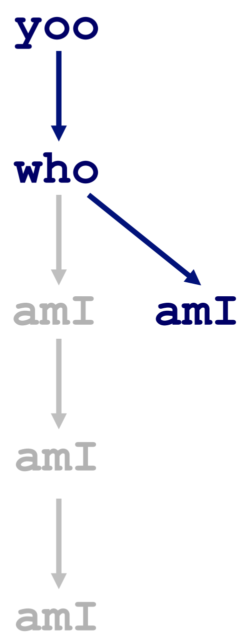



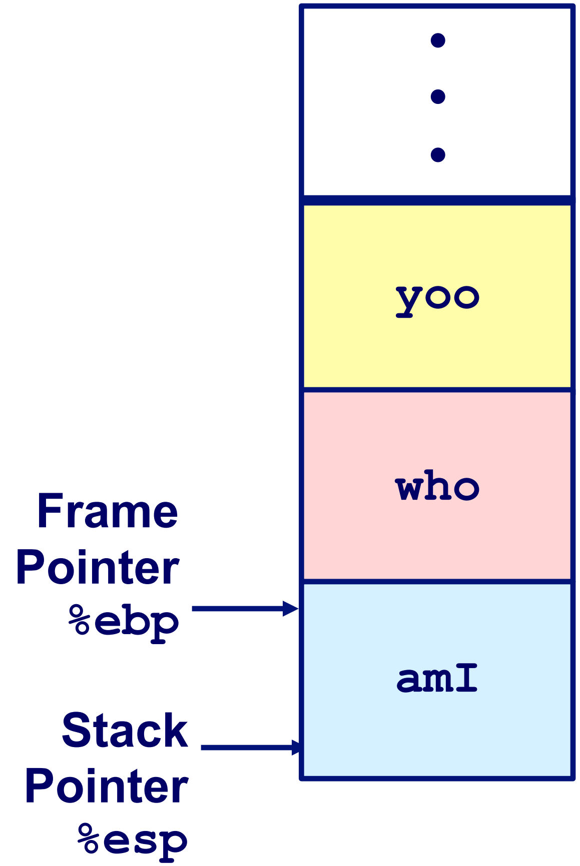
# Stack Operation

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

Lets say it amI() avoids recursive calling here and starts to return

## Call Chain

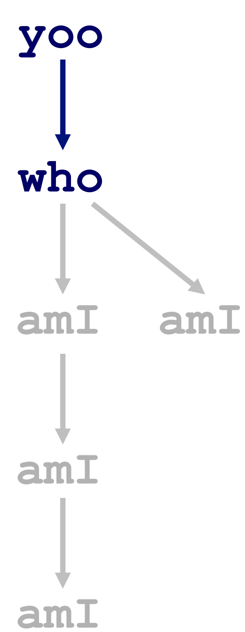


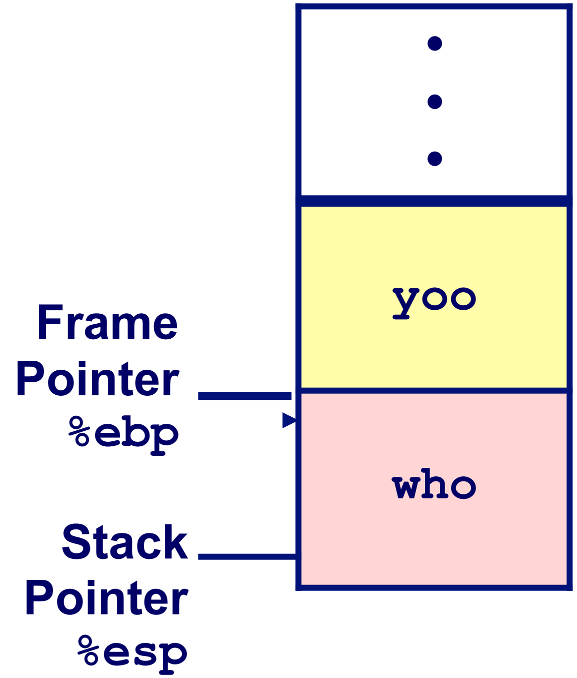



# Stack Operation

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

## Call Chain

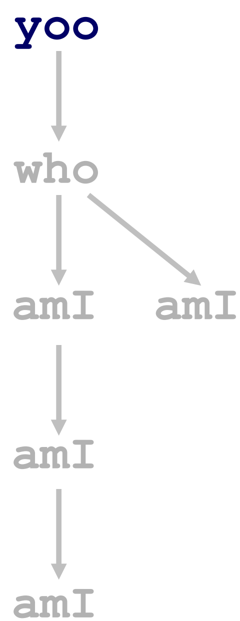


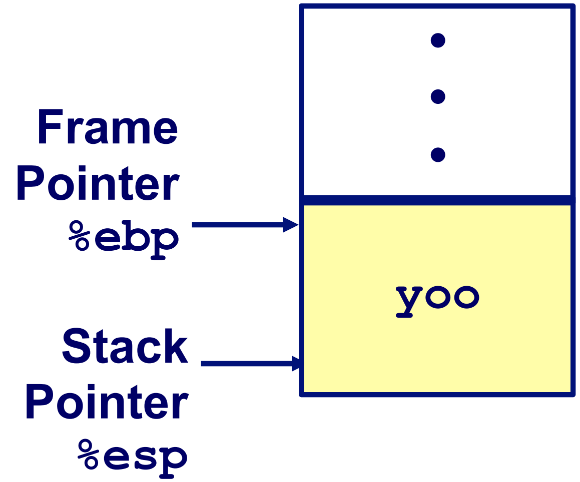



# Stack Operation

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

**Call Chain**

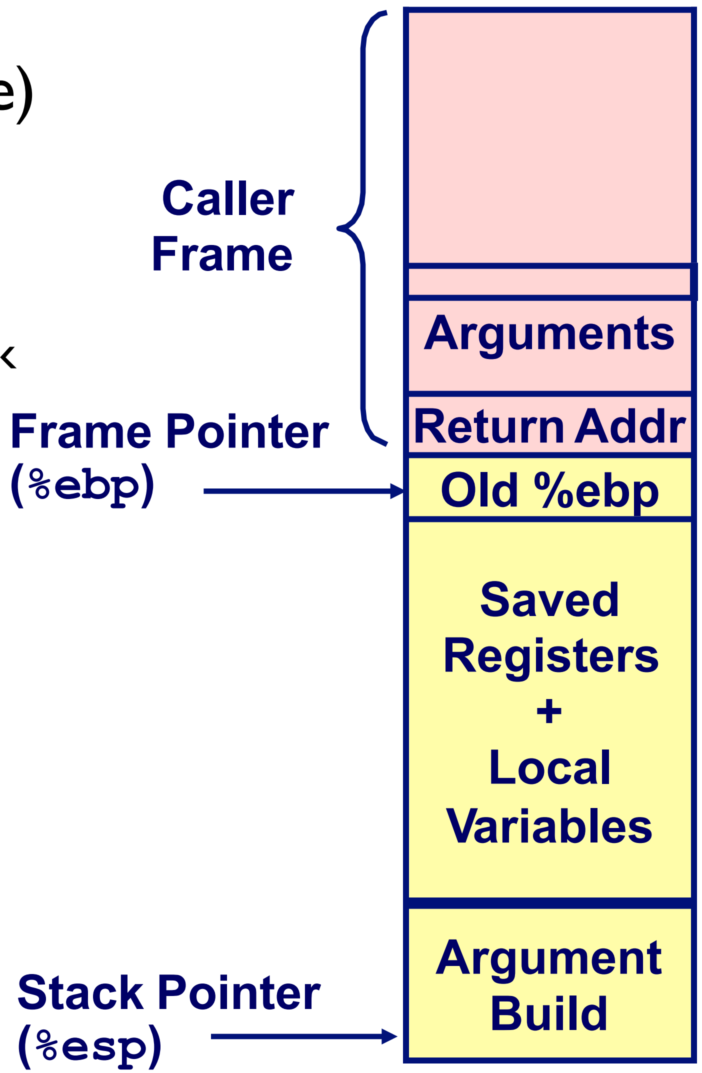


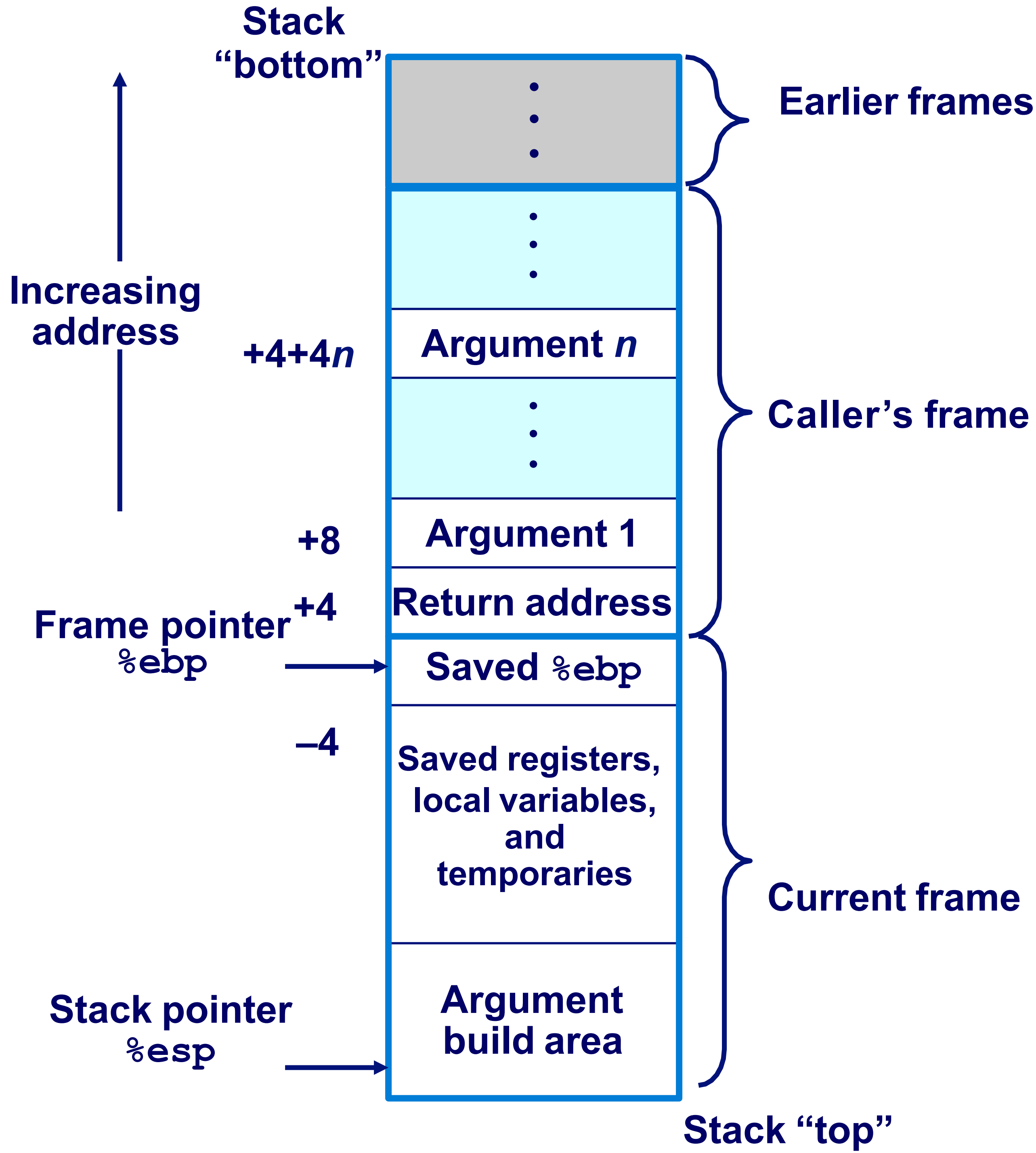



# Assembly:
# Stack and Procedure Control Cont.

# x86 Linux Stack Frame

- Current Stack Frame (Callee's frame)
  - From "Top" to "Bottom"
  - Parameters for function about to call
    - "Argument build"
    - Parameters are pushed on to the stack right to left
  - 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




Stack
"bottom"
Earlier frames
Increasing
address
+4+4n
Argument n
Caller's frame
+8
Argument 1
+4
Return address
Frame pointer
%ebp
Saved %ebp
–4
Saved registers,
local variables,
and
temporaries
Current frame
Stack pointer
%esp
Argument
build area
Stack "top"

# Revisiting `swap`

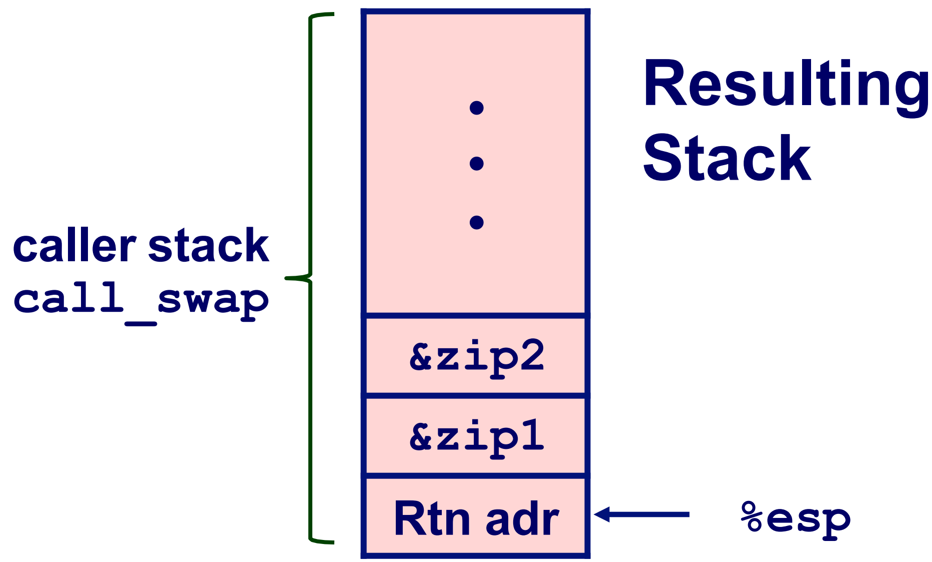
```
int zip1 = 15213;
int zip2 = 91125;

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
   • • •
```

Put return address into stack and jump to label



# Revisiting `swap` in x86

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

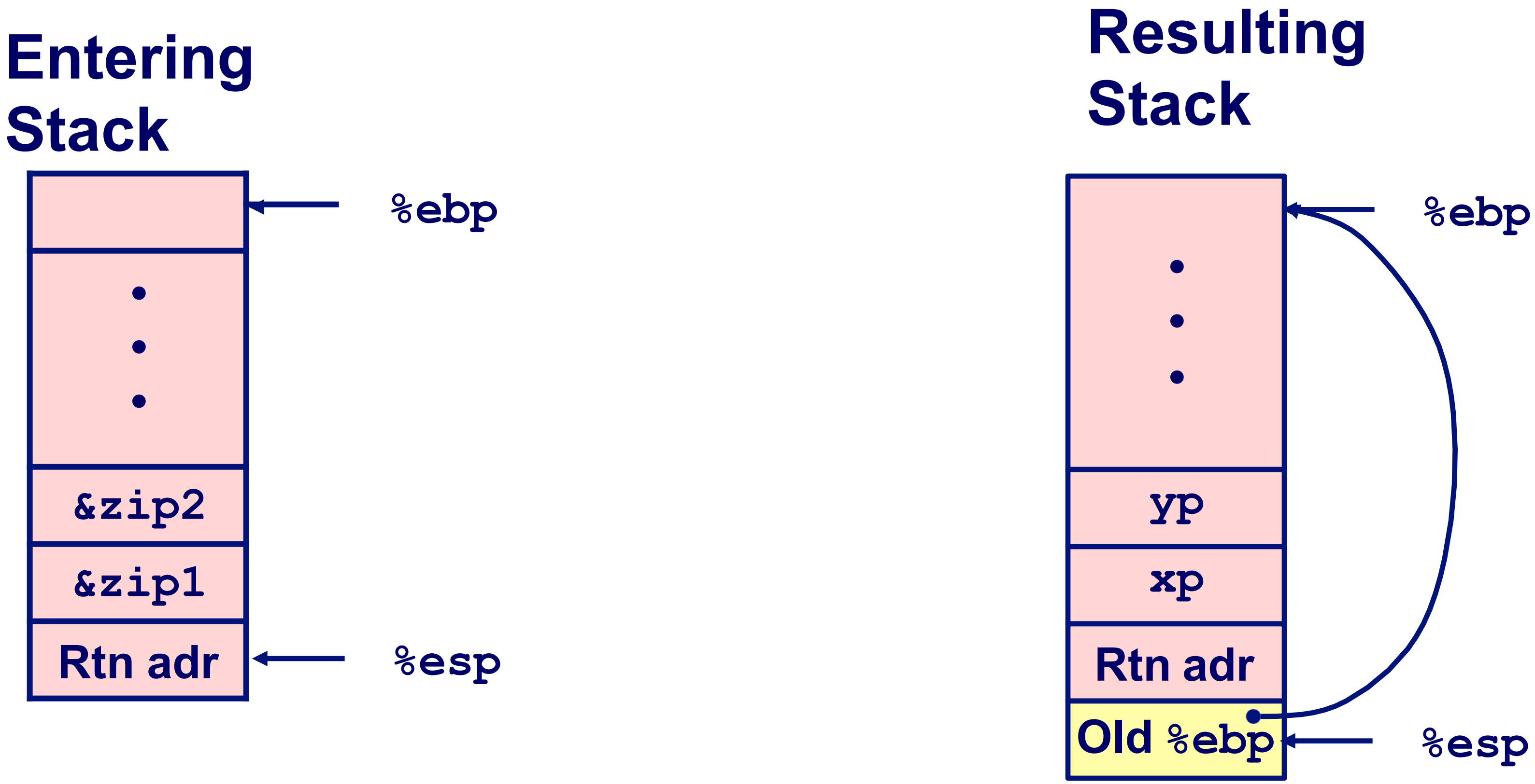
```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl%ebx

    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



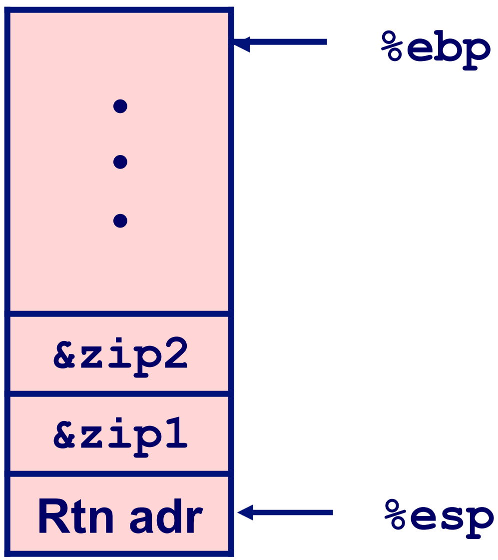

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

## Observation

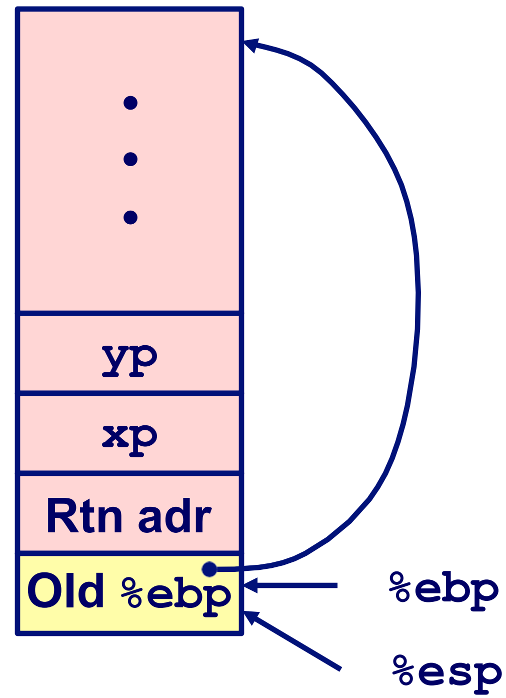
- Save `%ebp`

# swap Setup #2

## Entering Stack




## Resulting Stack



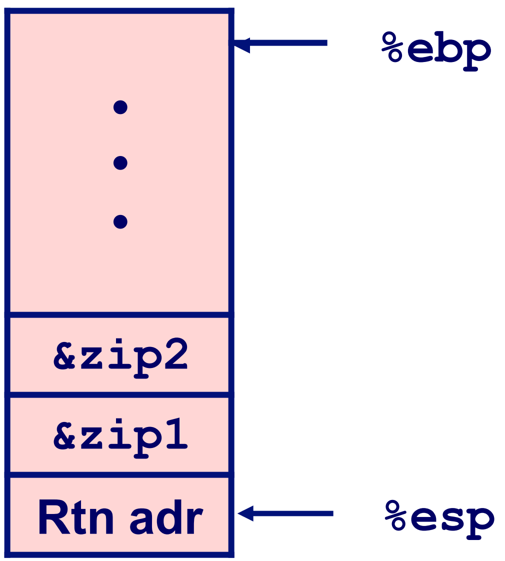

```
swap:
    pushl %ebp
    movl  %esp,%ebp
    pushl %ebx
```

## Observation

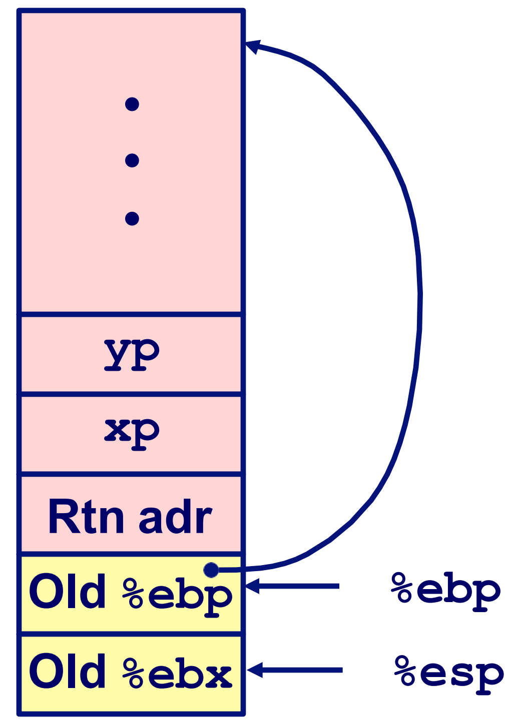
- Saved %ebp

# swap Setup #3

**Entering Stack**




**Resulting Stack**

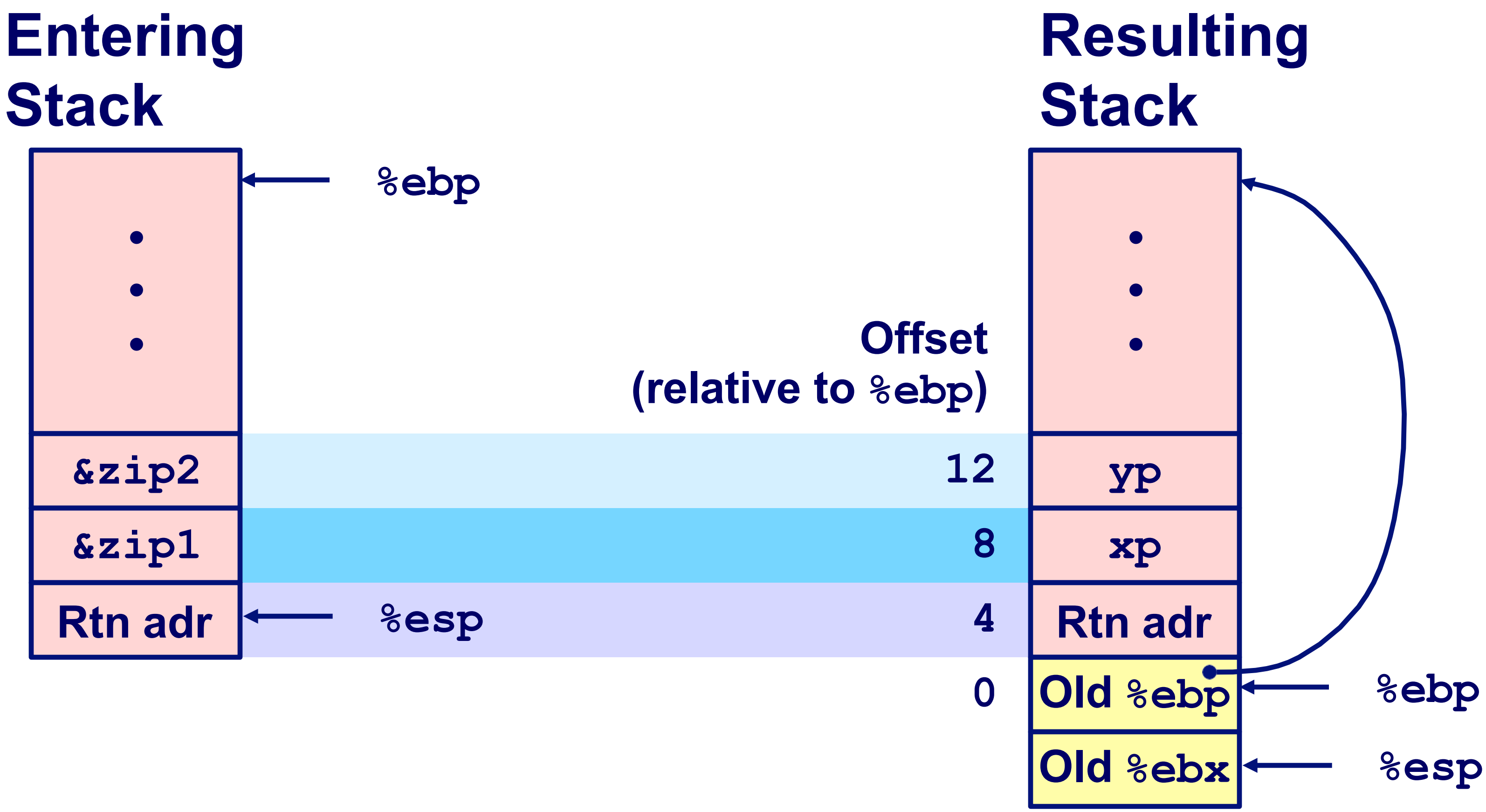



```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

## Observation

- Save register `%ebx`

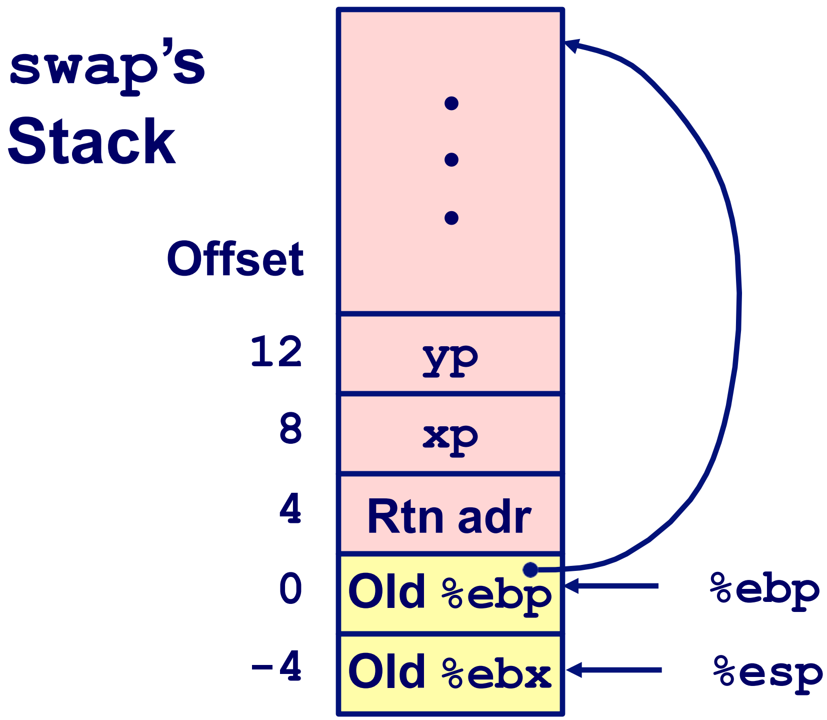
# Effect of `swap` Setup

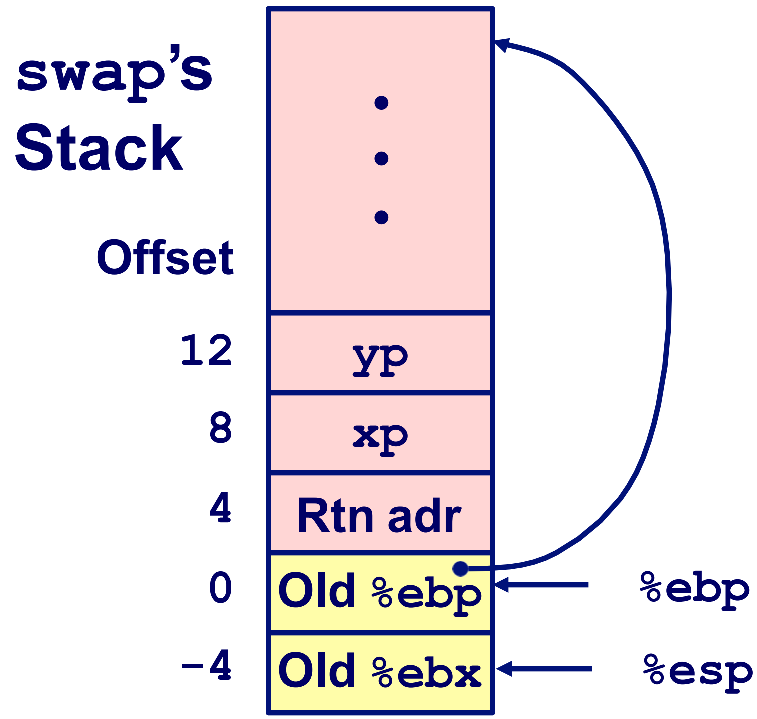



```
movl 12(%ebp),%ecx # get yp
movl 8(%ebp),%edx  # get xp
. . .
```

Body

# `swap` Finish #1



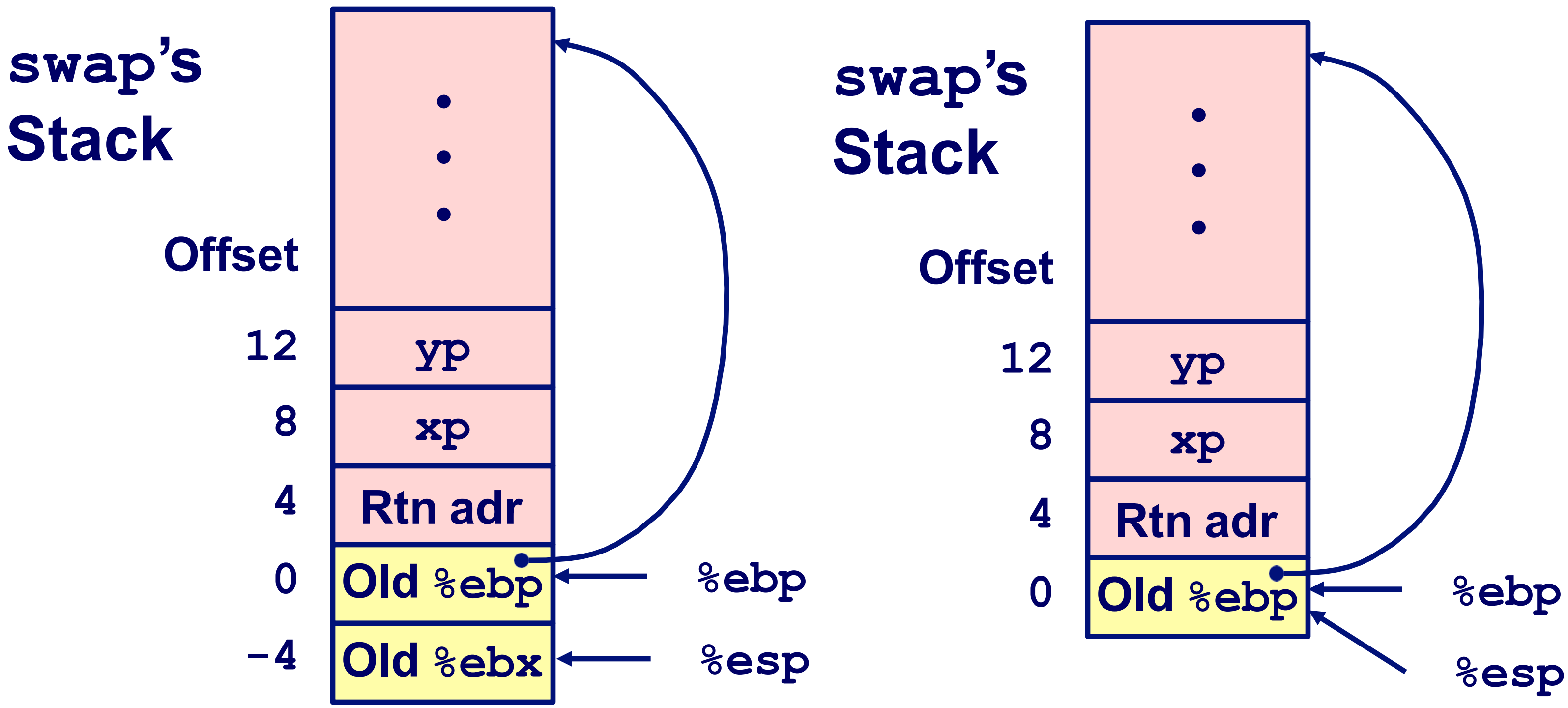




```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

## Observation

- Saved & restored register `%ebx`

# swap Finish #2

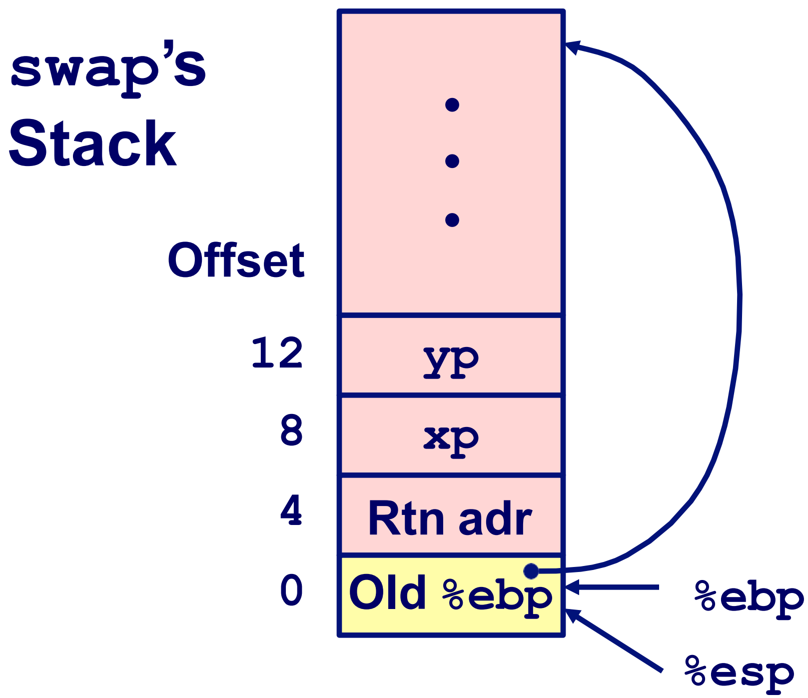


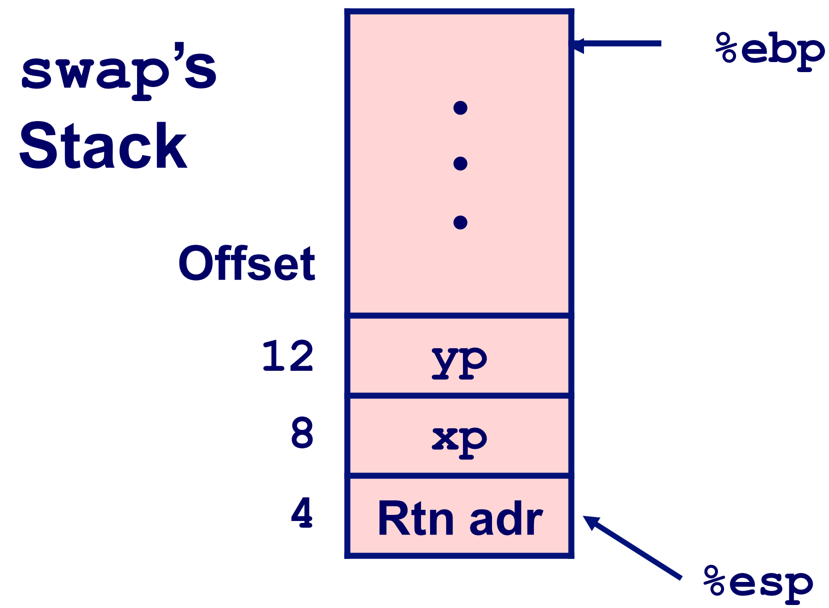

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

## Observation

- Set `%esp` to `%ebp` after restoring any registers

# `swap` Finish #3



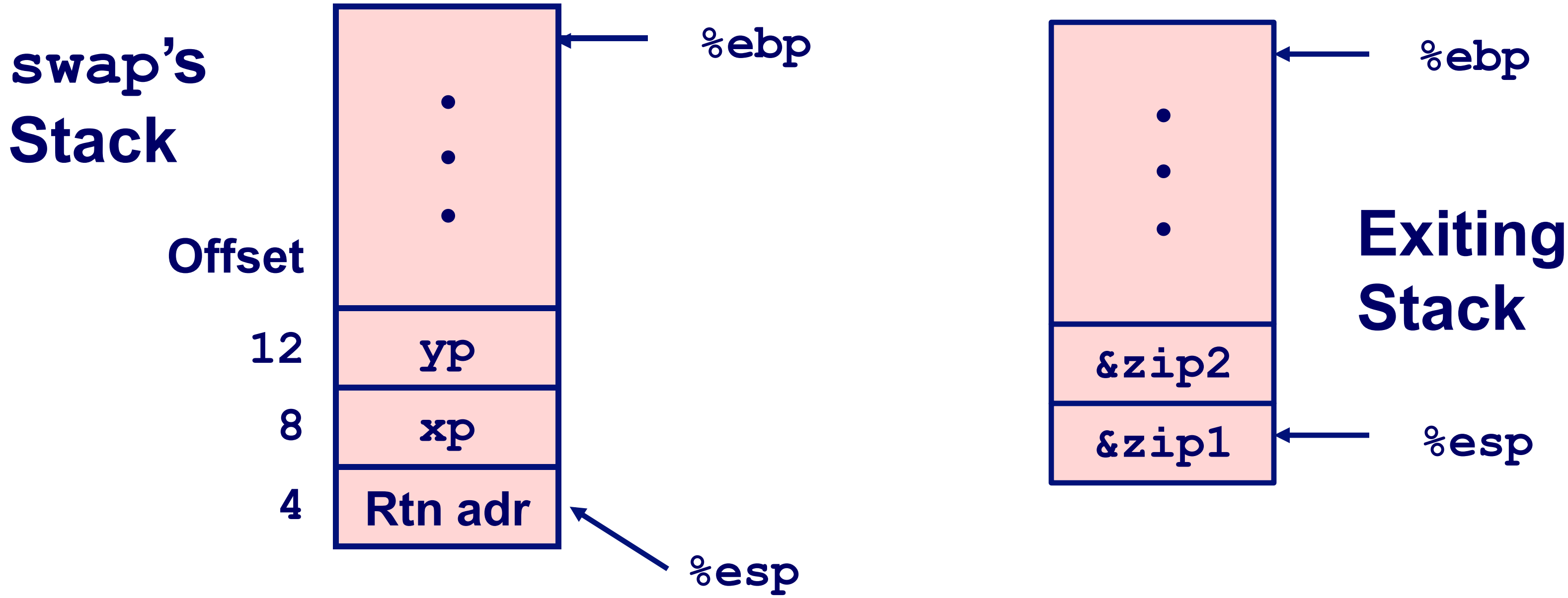




```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Observation

- Restore old `%ebp`

# `swap` Finish #4




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

Overall 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 a 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
```

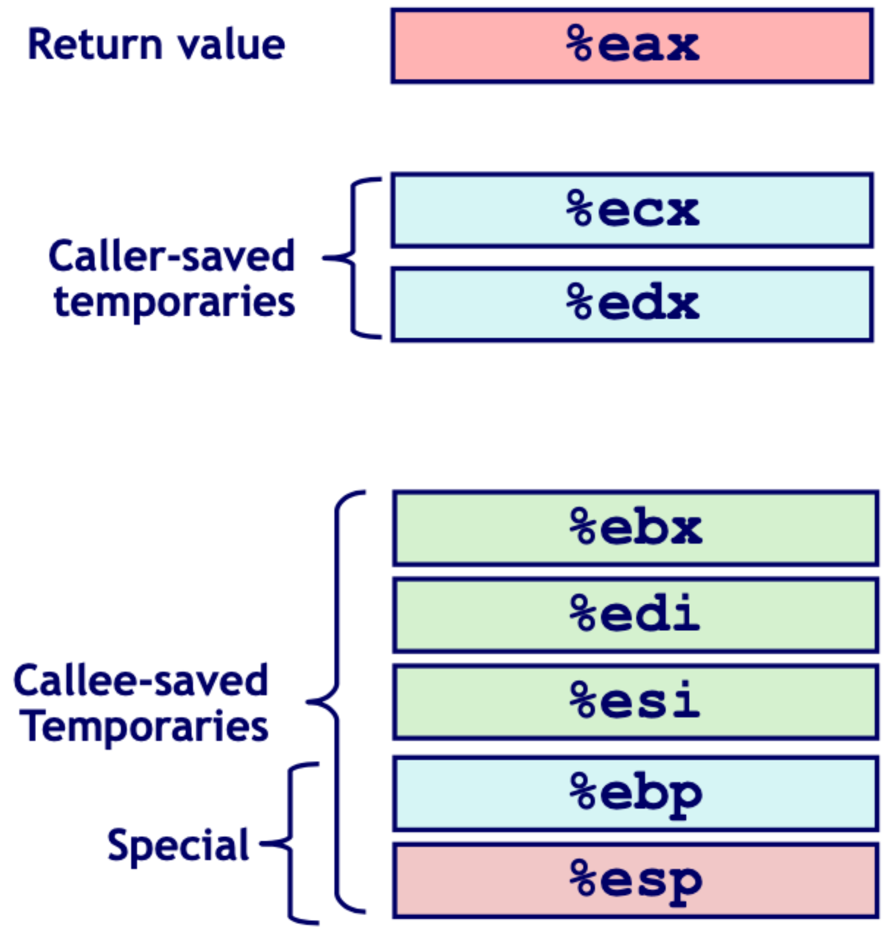
- Contents of register `%edx` overwritten by `who()`

# Register Saving Conventions

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

# x86 Linux Register Usage

- `%eax`
  - Used to store return value
  - Caller saved
  - Can be modified by procedure
- `%ecx, %edx`
  - Caller saved
  - Can be modified by procedure
- `%ebx, %edi, %esi`
  - Callee saved
  - Callee must save & restore
- `%ebp`
  - Callee saved
  - Callee must save & restore
  - May be used as frame pointer
- `%esp`
  - Special form of callee save
  - Restored to original value uipon exit of procedure



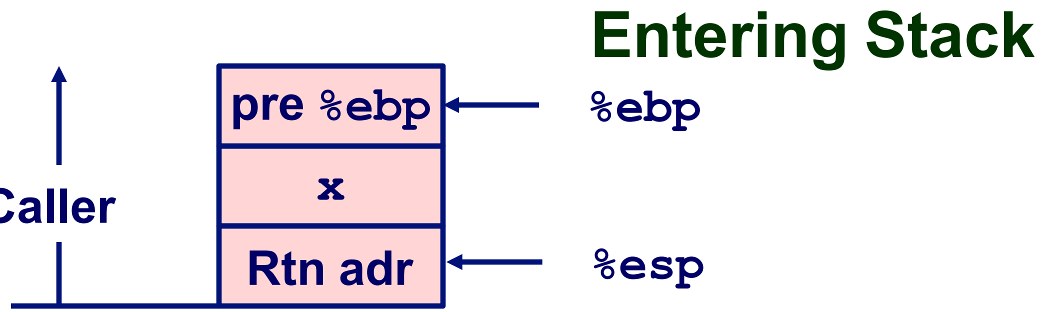
# Recursive Factorial

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

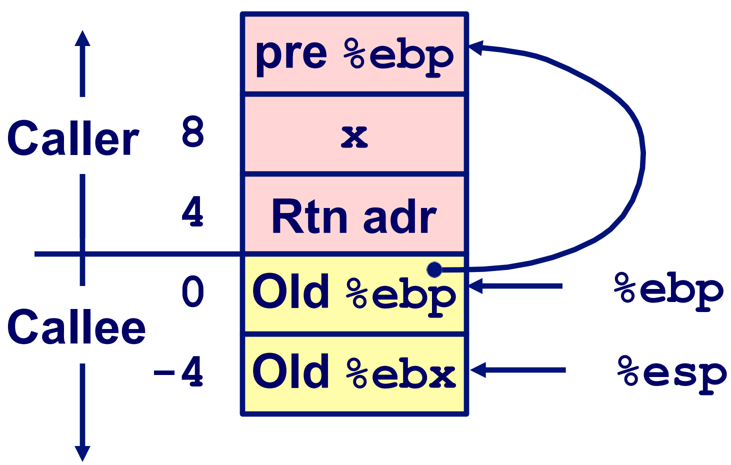
- Registers
  - `%eax` used without first saving
  - `%ebx` used, but save at beginning & restore at end

```
.globl rfact
    .type
rfact,@function
rfact:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 8(%ebp),%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(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

# Rfact Stack  Setup




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



# Rfact Body

Recursion

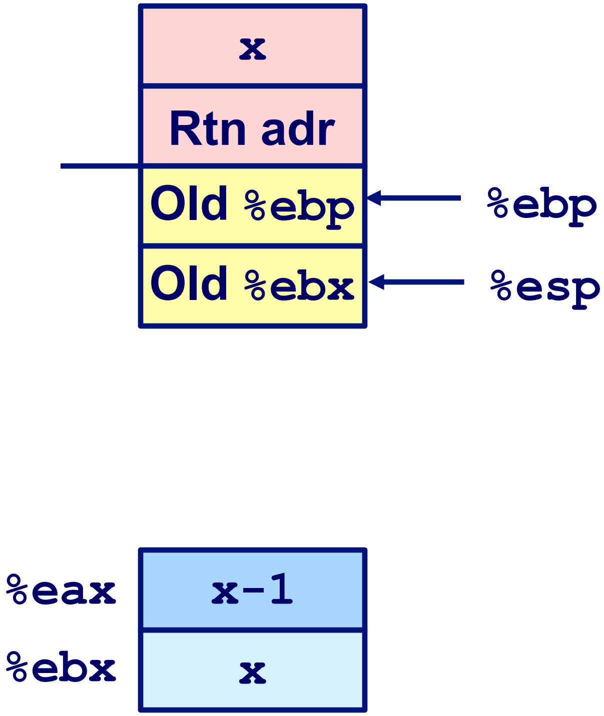
```
  movl 8(%ebp),%ebx    # ebx = x
  cmpl $1,%ebx         # Compare x : 1
  jle .L78             # If <= goto Term
  leal -1(%ebx),%eax   # eax = x-1
  pushl %eax           # Push x-1
  call rfact           # rfact(x-1)
  imull %ebx,%eax      # rval * x
  jmp .L79             # Goto done
.L78:    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

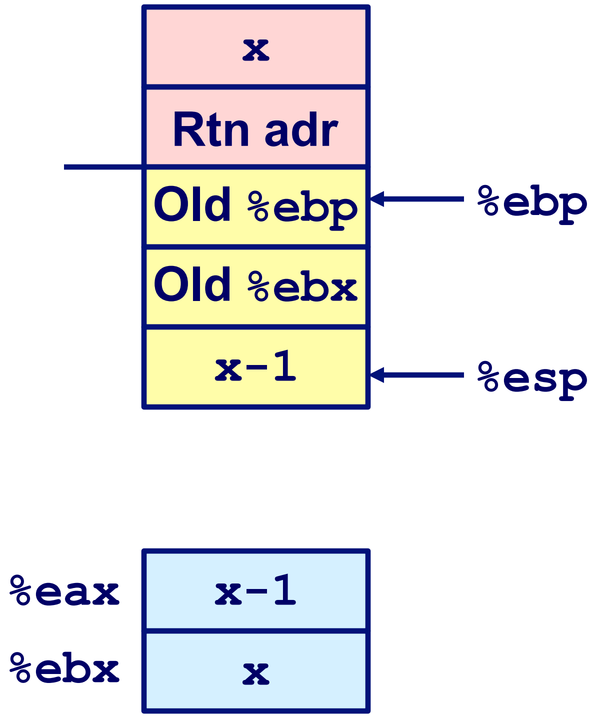
- Registers
  - `%ebx` Stored value of x
  - `%eax`
    - Temporary value of x-1
    - Returned value from rfact(x-1)
    - Returned value from this call

# Rfact Recursion

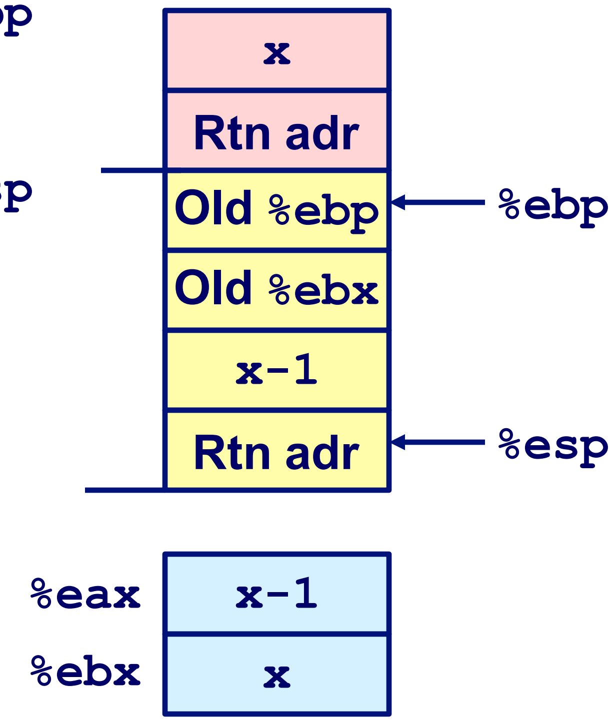
```
leal -1(%ebx),%eax
```




```
pushl %eax
```

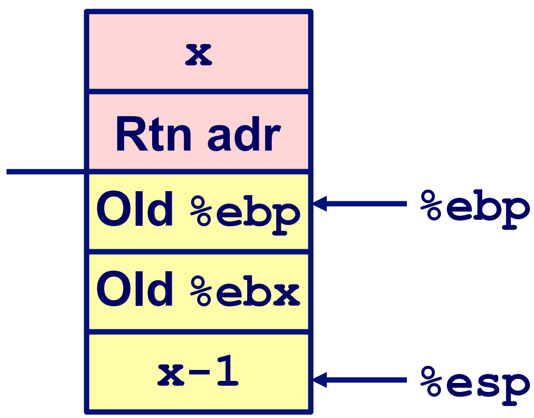



```
call rfact
```



# Rfact Result

**Return from Call**

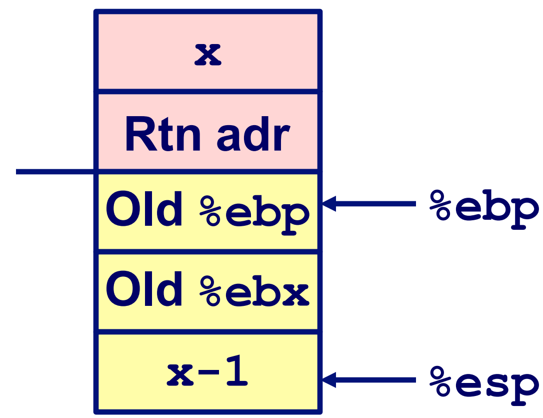


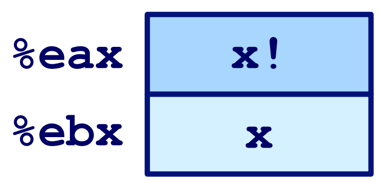

`%eax    (x-1)!`

`%ebx        x`

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

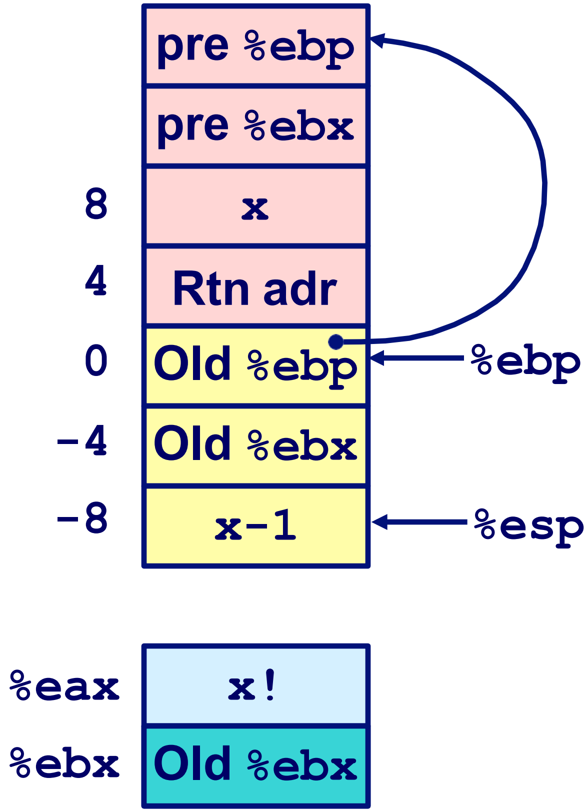
```
imull %ebx,%eax
```

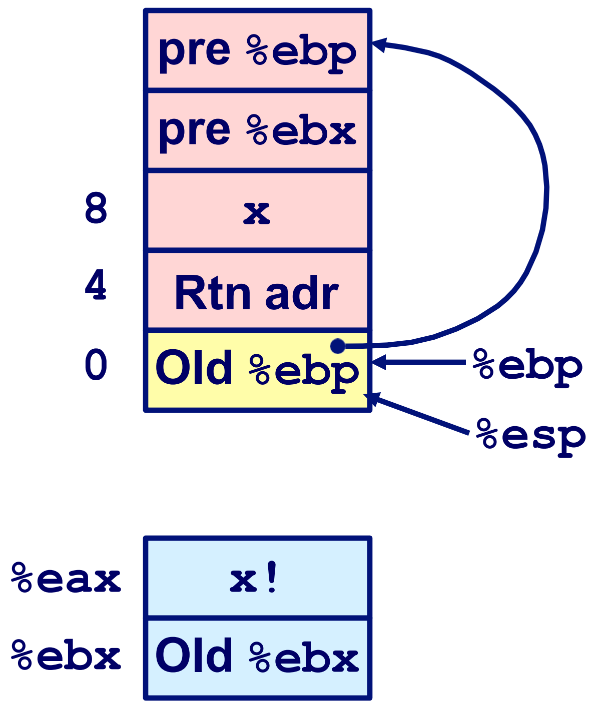


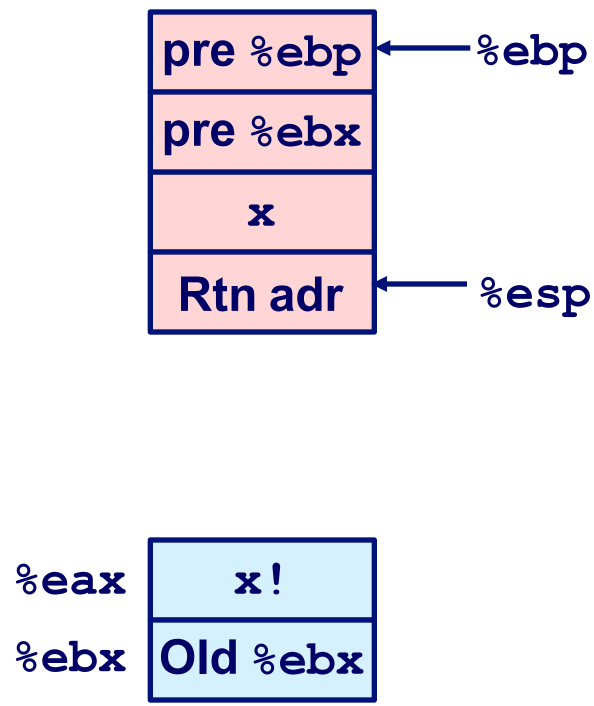



# Rfact Completion

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



# 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
- x86 Procedures Combination of Instructions + Conventions
  - Call / Ret instructions
  - Register usage conventions
    - Caller / Callee save
    - `%ebp` and `%esp`
  - Stack frame organization conventions