

# **Machine-Level Programming III: Switch Statements and IA32 Procedures**

**Introduction to Computer Systems**  
**6<sup>th</sup> Lecture, Sep. 30, 2015**

## **Instructors:**

Xiangqun Chen , Junlin Lu

Guangyu Sun , Xuetao Guan

# Today

- **Switch statements**
- **IA 32 Procedures**
  - Stack Structure
  - Calling Conventions
  - Illustrations of Recursion & Pointers

```
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;
}
```

# Switch Statement Example

- **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*

## Approximate Translation

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

# Switch Statement Example (IA32)

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
        . . .
    }
    return w;
}
```

What range of values takes default?

Setup:

```
switch_eg:
    pushl    %ebp                # Setup
    movl     %esp, %ebp         # Setup
    movl     8(%ebp), %eax       # %eax = x
    cmpl     $6, %eax           # Compare x:6
    ja       .L2                 # If unsigned > goto default
    jmp      *.L7(, %eax, 4)      # Goto *JTab[x]
```

Note that **w** not initialized here


# Switch Statement Example (IA32)

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
        . . .
    }
    return w;
}
```

## Jump table

```
.section      .rodata
    .align 4
.L7:
    .long      .L2 # x = 0
    .long      .L3 # x = 1
    .long      .L4 # x = 2
    .long      .L5 # x = 3
    .long      .L2 # x = 4
    .long      .L6 # x = 5
    .long      .L6 # x = 6
```

## Setup:

```
switch_eg:
    pushl      %ebp                # Setup
    movl      %esp, %ebp          # Setup
    movl      8(%ebp), %eax        # eax = x
    cmpl      $6, %eax            # Compare x:6
    ja        .L2                  # If unsigned > goto default
    Indirect   jump jump      *.L7(, %eax, 4) # Goto *JTab[x]
```

# Assembly Setup Explanation

## ■ Table Structure

- Each target requires 4 bytes
- Base address at `.L7`

## ■ Jumping

- **Direct:** `jmp .L2`
- Jump target is denoted by label `.L2`
- **Indirect:** `jmp *.L7(, %eax, 4)`
- Start of jump table: `.L7`
- Must scale by factor of 4 (labels have 32 bits = 4 Bytes on IA32)
- Fetch target from effective Address `.L7 + eax*4`
  - Only for  $0 \leq x \leq 6$

## Jump table

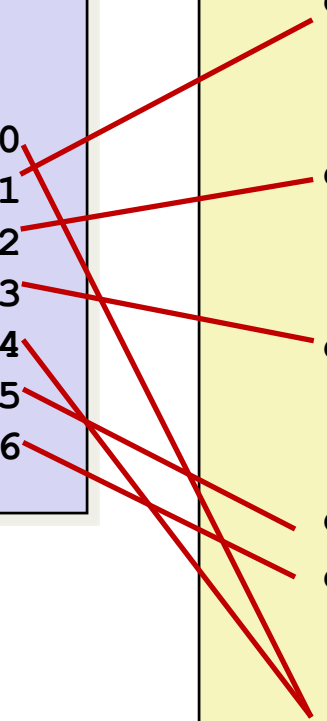
```
.section      .rodata
    .align 4
.L7:
    .long     .L2 # x = 0
    .long     .L3 # x = 1
    .long     .L4 # x = 2
    .long     .L5 # x = 3
    .long     .L2 # x = 4
    .long     .L6 # x = 5
    .long     .L6 # x = 6
```

# Jump Table

## Jump table

```
.section .rodata
.align 4
.L7:
.long .L2 # x = 0
.long .L3 # x = 1
.long .L4 # x = 2
.long .L5 # x = 3
.long .L2 # x = 4
.long .L6 # x = 5
.long .L6 # x = 6
```

```
switch(x) {
case 1:      // .L3
    w = y*z;
    break;
case 2:      // .L4
    w = y/z;
    /* Fall Through */
case 3:      // .L5
    w += z;
    break;
case 5:
case 6:      // .L6
    w -= z;
    break;
default:    // .L2
    w = 2;
}
```





# Code Blocks (x == 1)

```
switch(x) {  
  case 1:      // .L3  
    w = y*z;  
    break;  
  . . .  
}
```

```
.L3:      # x == 1  
    movl 12(%ebp), %eax # y  
    imull 16(%ebp), %eax # w = y*z  
    jmp  .L8           # Goto done
```

# Handling Fall-Through

```
long w = 1;  
    . . .  
switch(x) {  
    . . .  
case 2:   
    w = y/z;  
    /* Fall Through */  
case 3:  
    w += z;  
    break;  
    . . .  
}
```

case 2:  
 w = y/z;  
 goto merge;

case 3: w = 1;
merge: w += z;

# Code Blocks (x == 2, x == 3)

```

long w = 1;
. . .
switch(x) {
. . .
case 2:
    w = y/z;
    /* Fall Through */
case 3:
    w += z;
    break;
. . .
}

```

```

.L5:                                # x == 2
    movl    12(%ebp), %eax # y
    cltd
    idivl    16(%ebp)      # y/z
    jmp     .L6            # goto merge
.L9:                                # x == 3
    movl    $1, %eax      # w = 1
.L6:                                # merge:
    addl    16(%ebp), %eax # += z
    jmp     .L8            # goto done

```

# Code Blocks (x == 5, x == 6, default)

```
switch(x) {  
    . . .  
    case 5:  // .L7  
    case 6:  // .L7  
        w -= z;  
        break;  
    default: // .L8  
        w = 2;  
}
```

```
.L7:                                # x == 5, 6  
    movl    $1, %eax               # w = 1  
    subl    16(%ebp), %eax         # w -= z  
    jmp     .L8                    # goto done  
.L2:                                # default  
    movl    $2, %eax               # w = 2  
.L8:                                # done:
```

# Switch Code (Finish)

```
return w;
```

```
.L8:      # done:  
    popl   %ebp  
    ret
```

## ■ Noteworthy Features

- Jump table avoids sequencing through cases
  - Constant time, rather than linear
- Use jump table to handle holes and duplicate tags
- Use program sequencing to handle fall-through
- Don't initialize  $w = 1$  unless really need it

# x86-64 Switch Implementation

- Same general idea, adapted to 64-bit code
- Table entries 64 bits (pointers)
- Cases use revised code

```
switch(x) {
case 1:      // .L3
    w = y*z;
    break;
    . . .
}
```

```
.L3:
    movq    %rdx, %rax
    imulq   %rsi, %rax
    ret
```

## Jump Table

```
.section .rodata
.align 8
.L7:
    .quad   .L2      # x = 0
    .quad   .L3      # x = 1
    .quad   .L4      # x = 2
    .quad   .L5      # x = 3
    .quad   .L2      # x = 4
    .quad   .L6      # x = 5
    .quad   .L6      # x = 6
```

# IA32 Object Code

## ■ Setup

- Label `.L2` becomes address `0x80484b8`
- Label `.L7` becomes address `0x8048680`

## Assembly Code

```
switch_eg:
    . . .
    ja      .L2          # If unsigned > goto default
    jmp     *.L7(, %eax, 4) # Goto *JTab[x]
```

## Disassembled Object Code

```
08048480 <switch_eg>:
    . . .
8048489: 77 2d                ja 80484b8 <switch_eg+0x38>
804848b: ff 24 85 80 86 04 08 jmp *0x8048680(, %eax, 4)
```

# IA32 Object Code (cont.)

## ■ Jump Table

- Doesn't show up in disassembled code
- Can inspect using GDB
- `gdb switch`
- `(gdb) x/7xw 0x8048680`
  - Examine 7 hexadecimal format "words" (4-bytes each)
  - Use command "**help x**" to get format documentation

<code><b>0x8048680:</b></code>	<code><b>0x080484b8</b></code>	<code><b>0x08048492</b></code>	<code><b>0x0804849b</b></code>	<code><b>0x080484a4</b></code>
<code><b>0x8048690:</b></code>	<code><b>0x080484b8</b></code>	<code><b>0x080484ae</b></code>	<code><b>0x080484ae</b></code>	



# IA32 Object Code (cont.)

## ■ Deciphering Jump Table

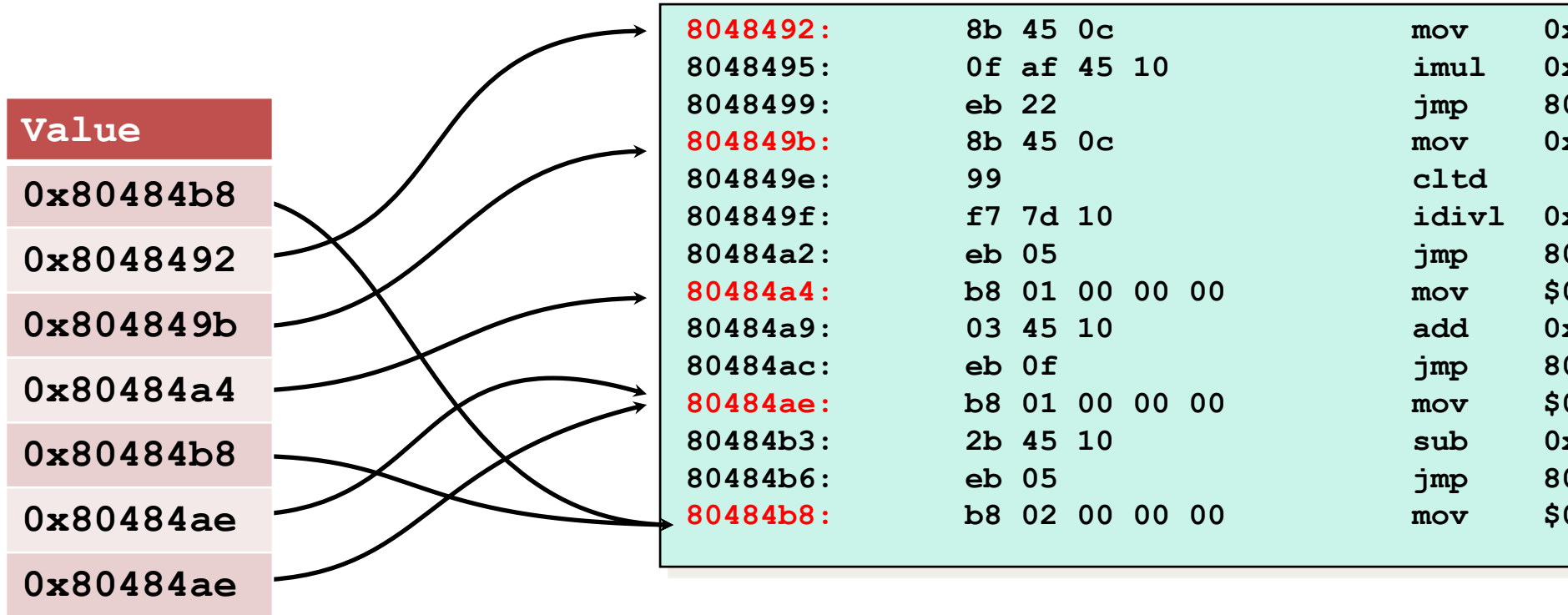
0x8048680:      0x080484b8      0x08048492      0x0804849b      0x080484a4  
0x8048690:      0x080484b8      0x080484ae      0x080484ae

Address	Value	x
0x8048680	0x080484b8	0
0x8048684	0x08048492	1
0x8048688	0x0804849b	2
0x804868c	0x080484a4	3
0x8048690	0x080484b8	4
0x8048694	0x080484ae	5
0x8048698	0x080484ae	6

# Disassembled Targets

<b>8048492:</b>	8b 45 0c	mov	0xc(%ebp),%eax
8048495:	0f af 45 10	imul	0x10(%ebp),%eax
8048499:	eb 22	jmp	80484bd <switch_eg+0x3d>
<b>804849b:</b>	8b 45 0c	mov	0xc(%ebp),%eax
804849e:	99	cltd	
804849f:	f7 7d 10	idivl	0x10(%ebp)
80484a2:	eb 05	jmp	80484a9 <switch_eg+0x29>
<b>80484a4:</b>	b8 01 00 00 00	mov	\$0x1,%eax
80484a9:	03 45 10	add	0x10(%ebp),%eax
80484ac:	eb 0f	jmp	80484bd <switch_eg+0x3d>
<b>80484ae:</b>	b8 01 00 00 00	mov	\$0x1,%eax
80484b3:	2b 45 10	sub	0x10(%ebp),%eax
80484b6:	eb 05	jmp	80484bd <switch_eg+0x3d>
<b>80484b8:</b>	b8 02 00 00 00	mov	\$0x2,%eax

# Matching Disassembled Targets



# Summarizing

## ■ C Control

- if-then-else
- do-while
- while, for
- switch

## ■ Assembler Control

- Conditional jump
- Conditional move
- Indirect jump
- Compiler generates code sequence to implement more complex control

## ■ Standard Techniques

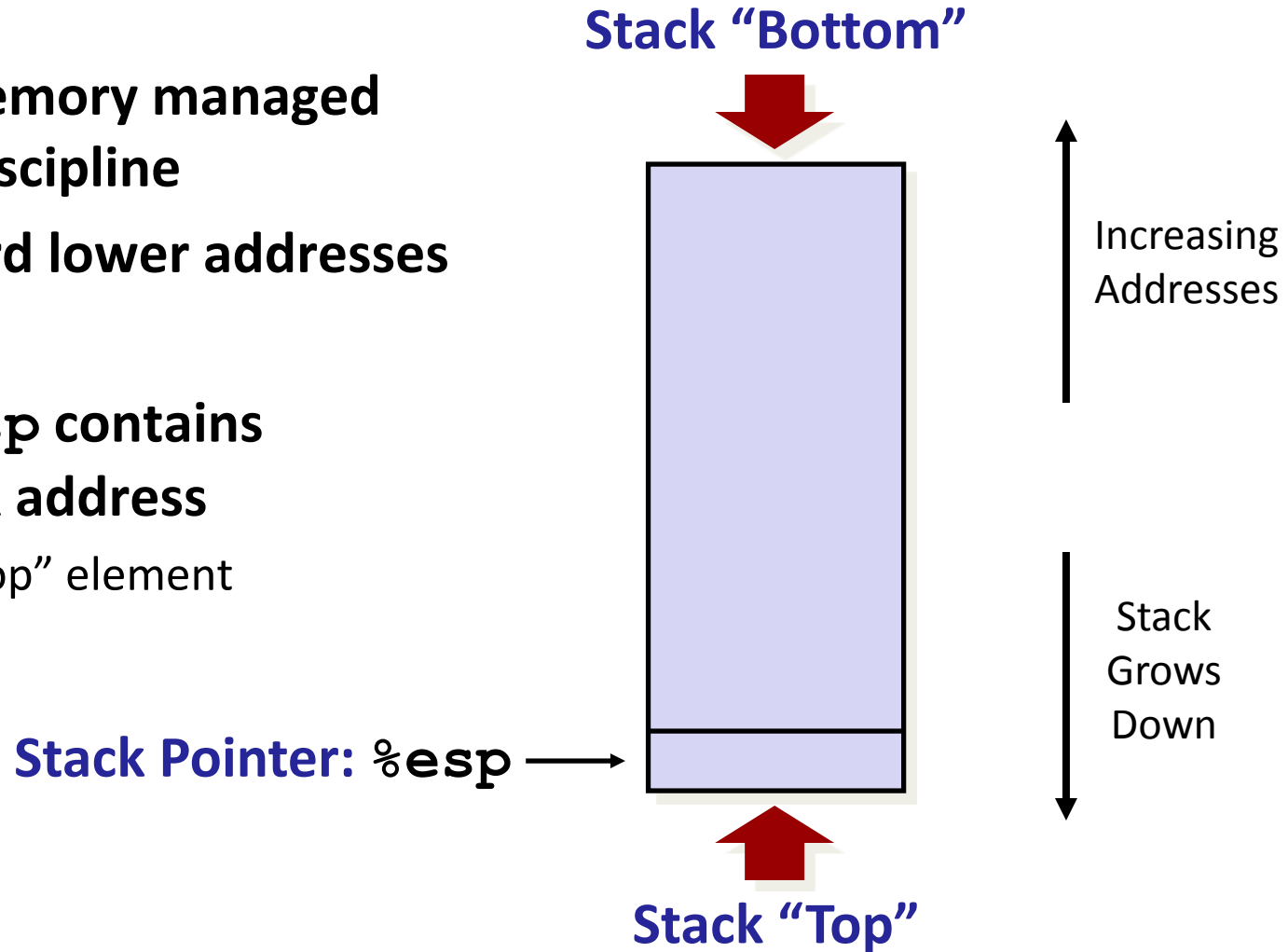
- Loops converted to do-while form
- Large switch statements use jump tables
- Sparse switch statements may use decision trees

# Today

- Switch statements
- **IA 32 Procedures**
  - Stack Structure
  - Calling Conventions
  - Illustrations of Recursion & Pointers

# IA32 Stack

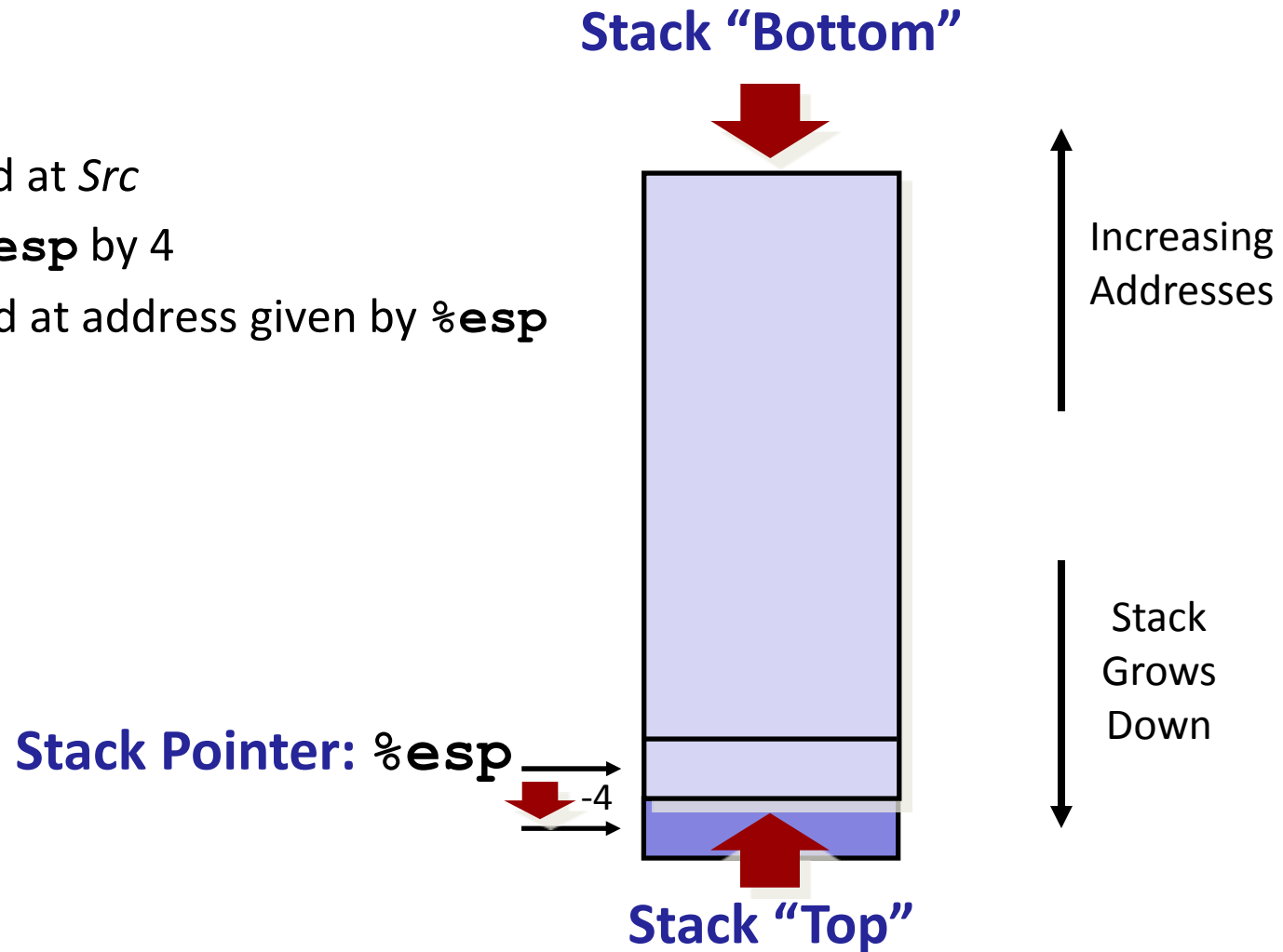
- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%esp` contains lowest stack address
  - address of “top” element



# IA32 Stack: Push

## ■ `pushl Src`

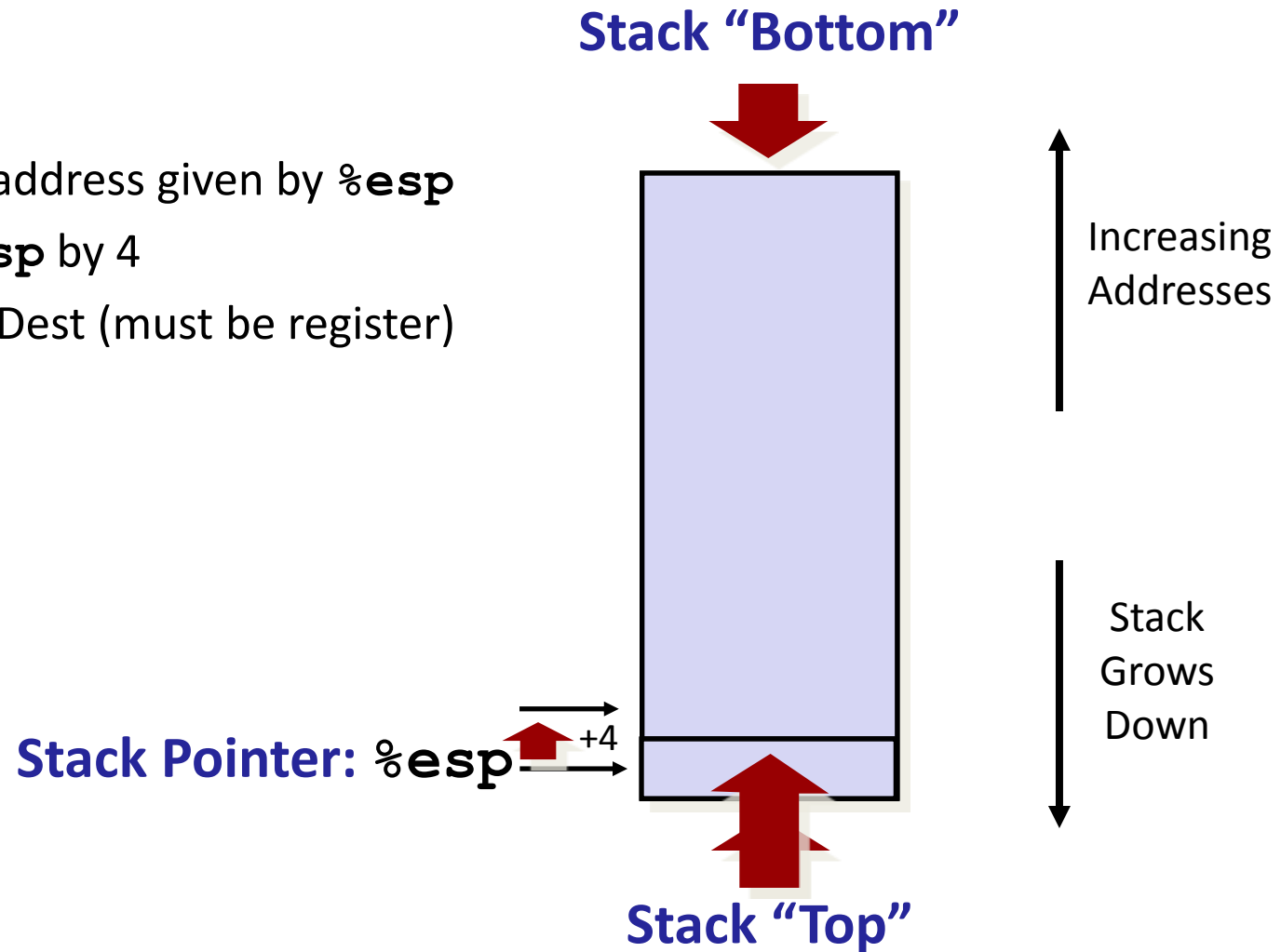
- Fetch operand at *Src*
- Decrement `%esp` by 4
- Write operand at address given by `%esp`



# IA32 Stack: Pop

## ■ `popl Dest`

- Read value at address given by `%esp`
- Increment `%esp` by 4
- Store value at `Dest` (must be register)





# 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

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

- Return address = `0x8048553`

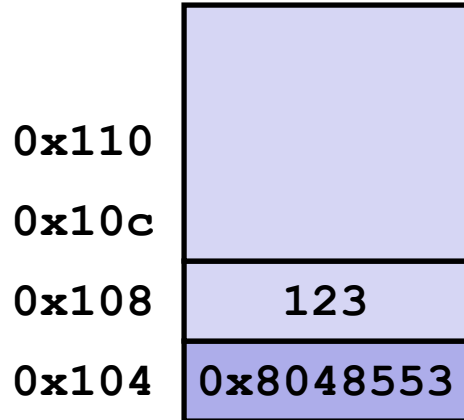
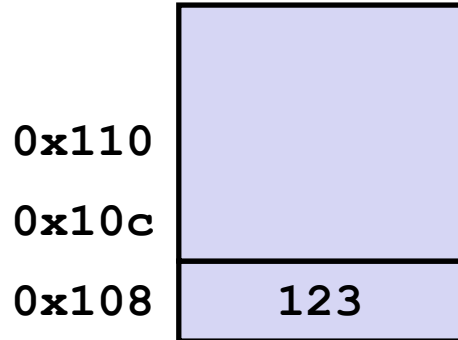
- **Procedure return:** `ret`

- Pop address from stack
- Jump to address

# Procedure Call Example

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

**call 8048b90**



%esp: 0x108

%esp: 0x104

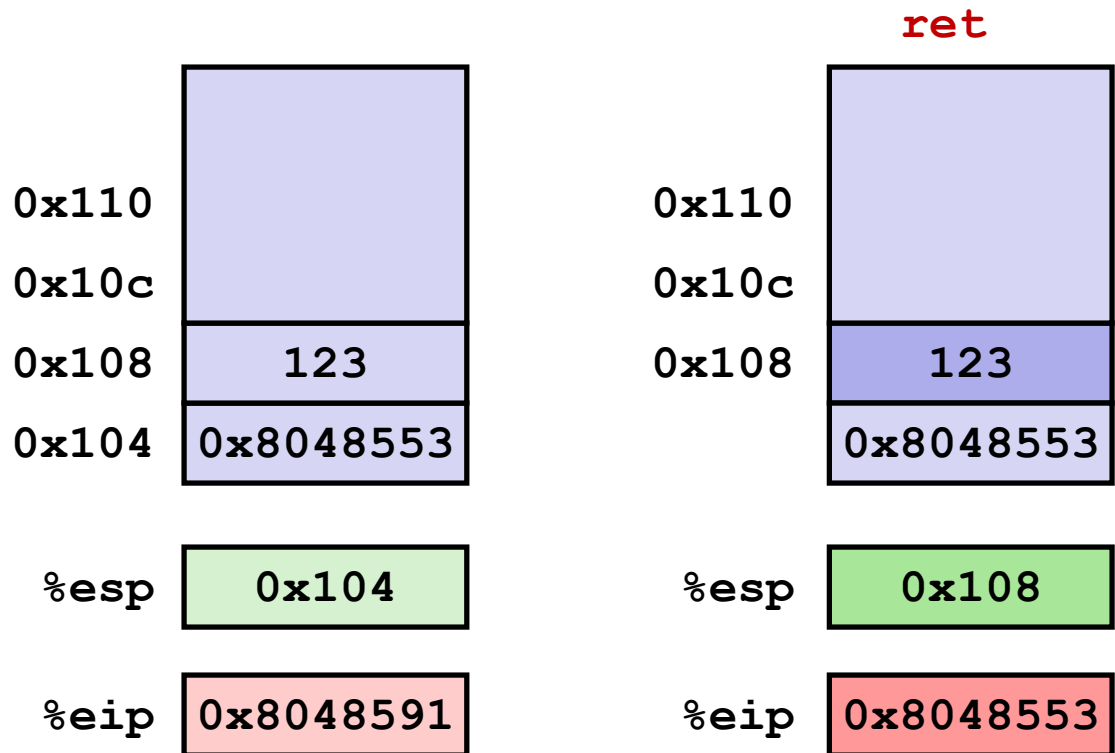
%eip: 0x804854e

%eip: 0x8048b90

*%eip: program counter*

# Procedure Return Example

```
8048591:    c3                ret
```



*%eip: 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 for single procedure instantiation

# Call Chain Example

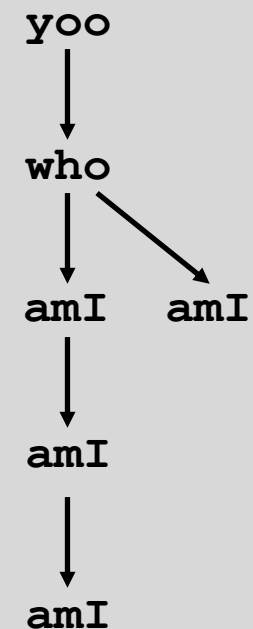
```
yoo (...)  
{  
  .  
  .  
  who ();  
  .  
  .  
}
```

```
who (...)  
{  
  . . .  
  amI ();  
  . . .  
  amI ();  
  . . .  
}
```

```
amI (...)  
{  
  .  
  .  
  amI ();  
  .  
  .  
}
```

**Procedure amI () is recursive**

## Example Call Chain



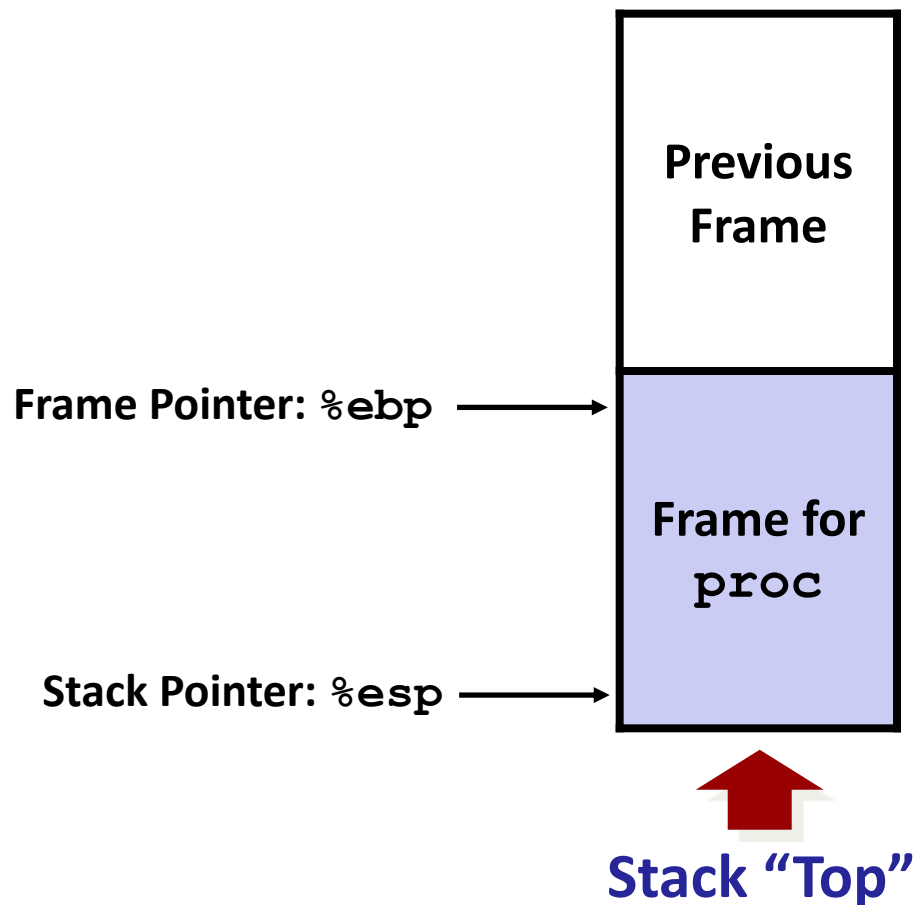
# Stack Frames

## ■ Contents


- Local variables
- Return information
- Temporary space

## ■ Management

- Space allocated when enter procedure
  - “Set-up” code
- Deallocated when return
  - “Finish” code

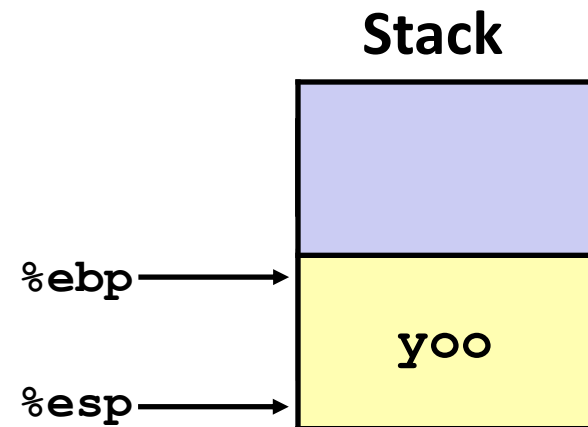


# Example

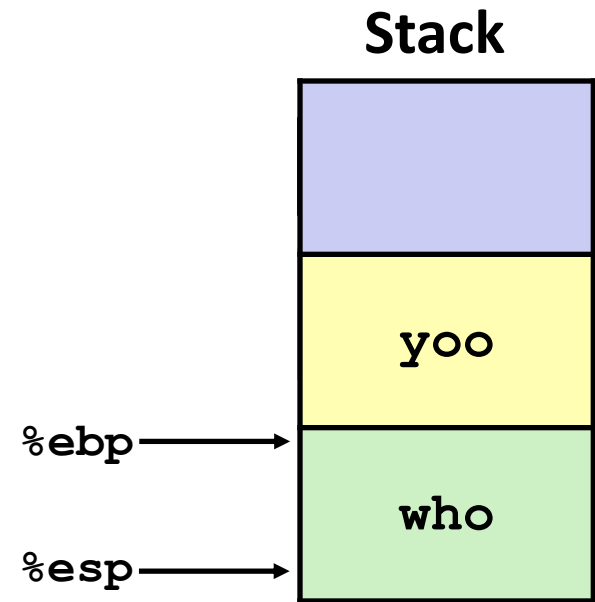
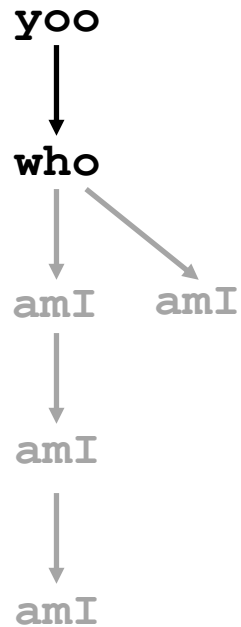
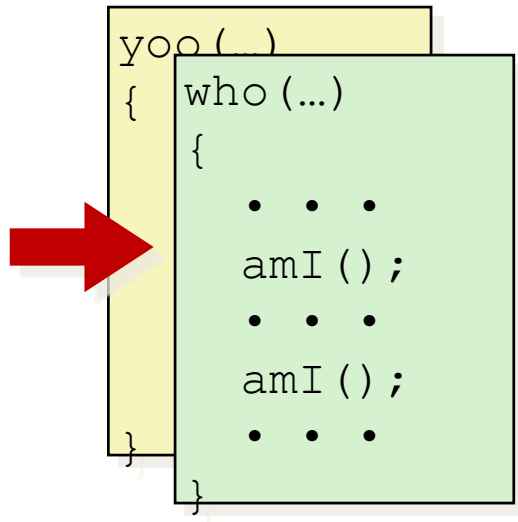


```
yoo (...)  
{  
  .  
  .  
  who ();  
  .  
  .  
}
```

```
yoo  
  ↓  
who  
  ↓  ↘  
amI  amI  
  ↓  
amI  
  ↓  
amI
```

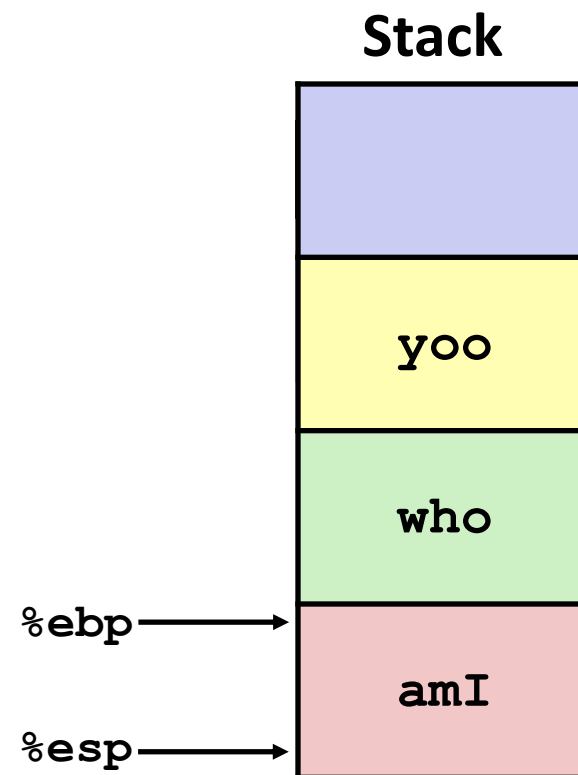
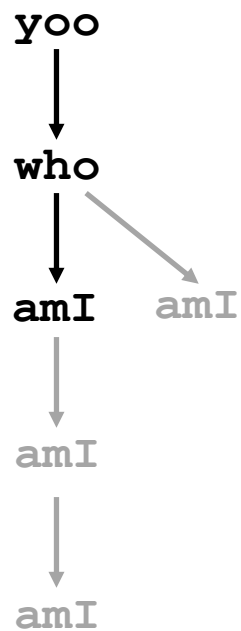
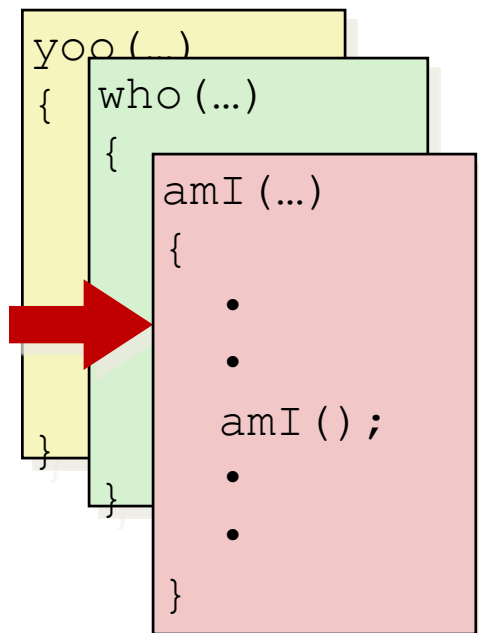


# Example

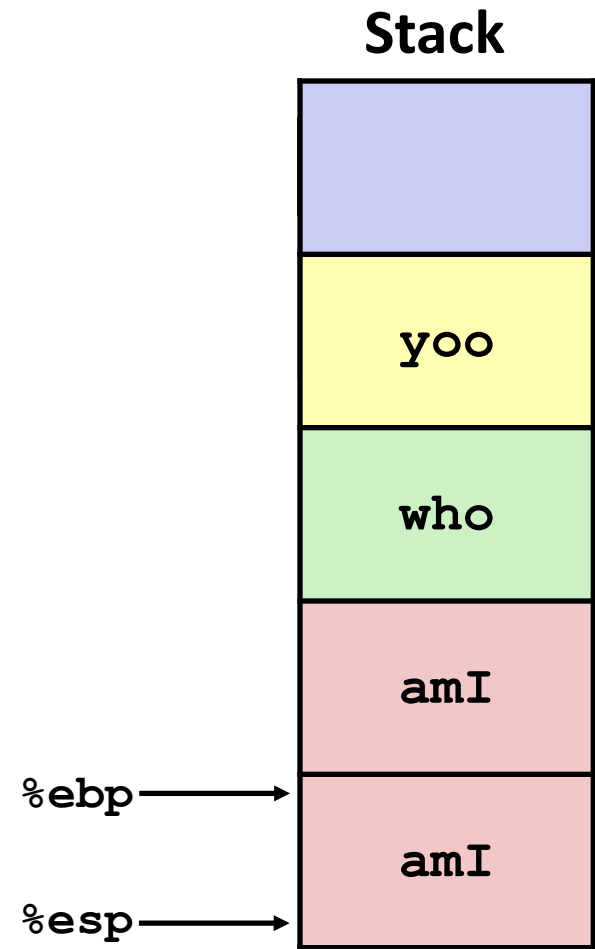
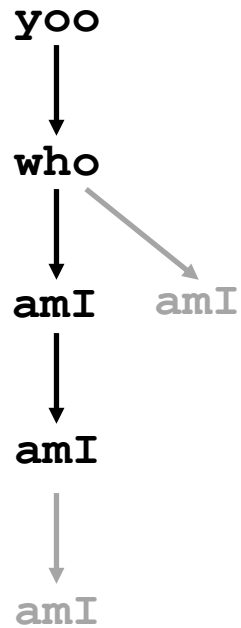
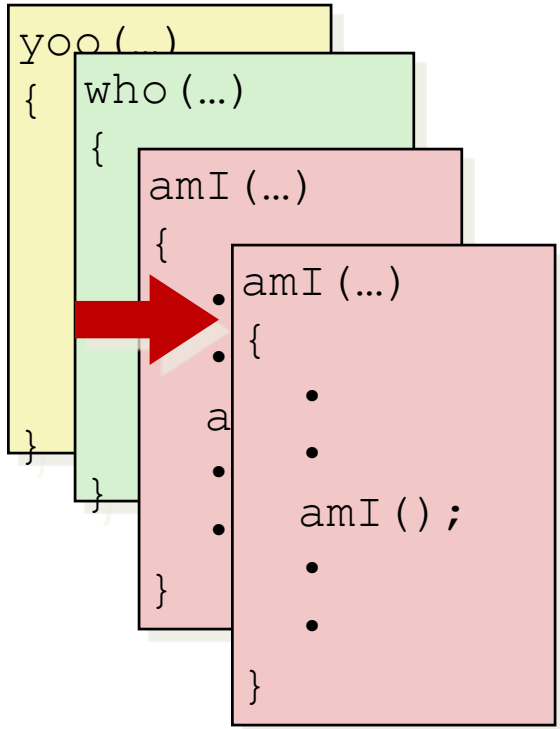




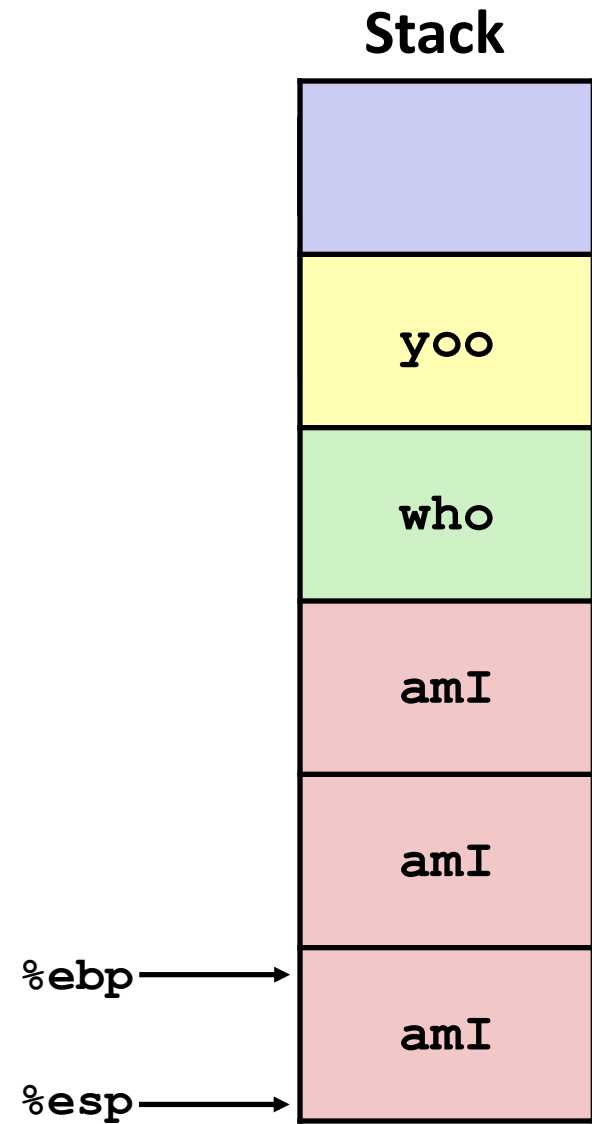
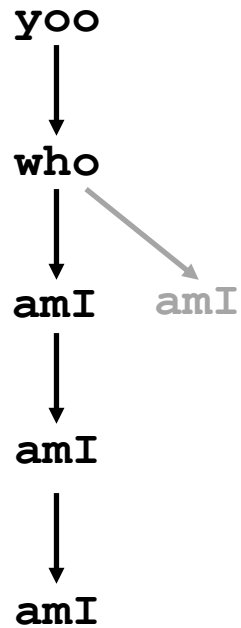
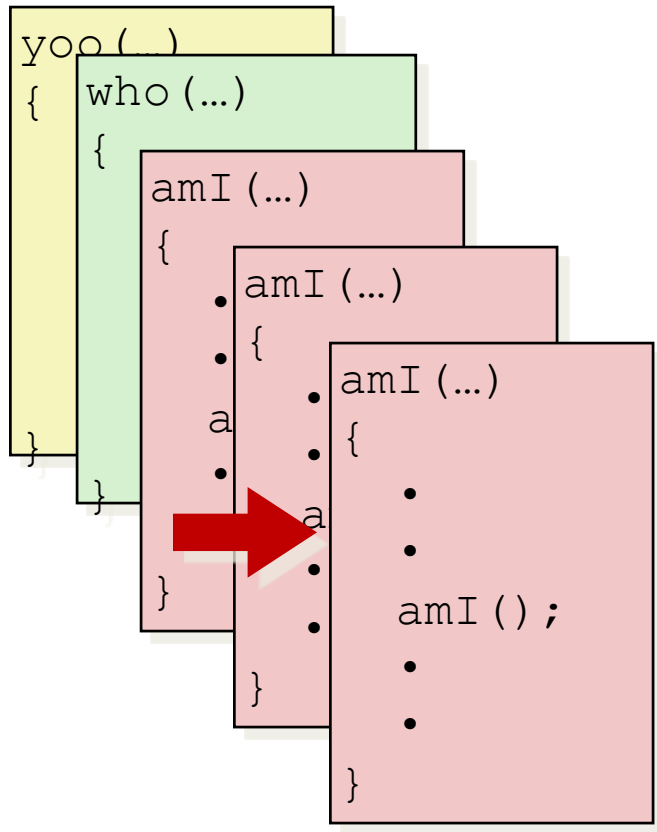
# Example



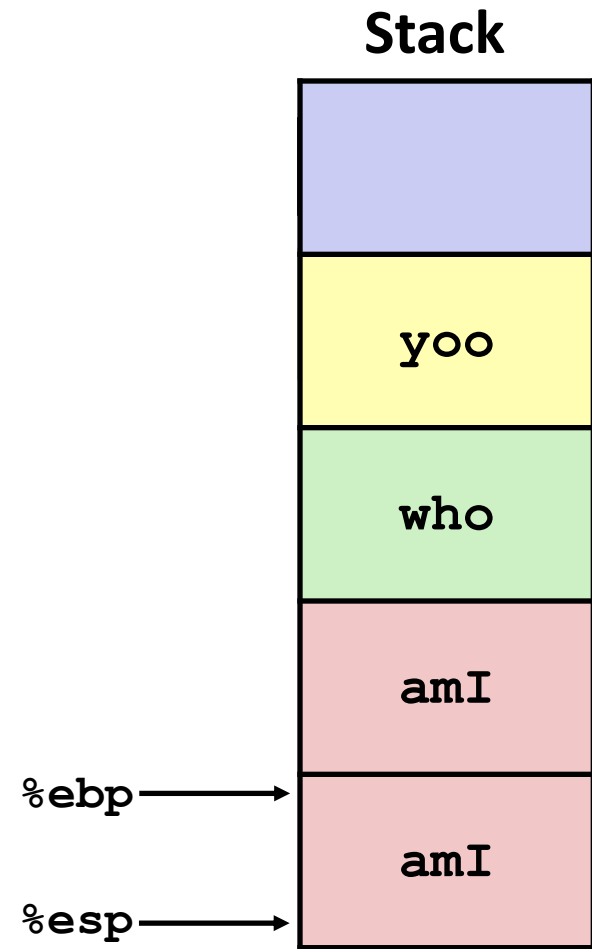
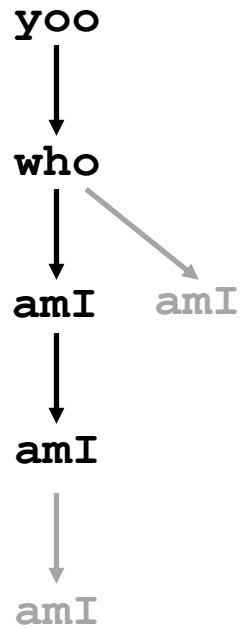
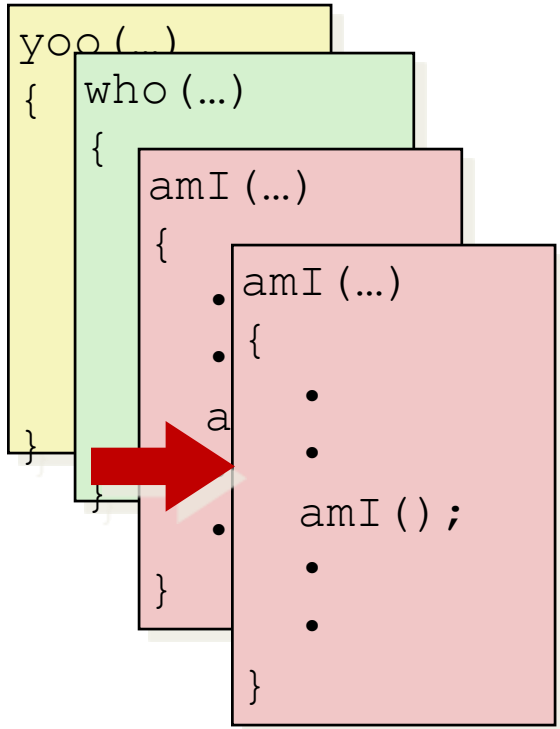
# Example



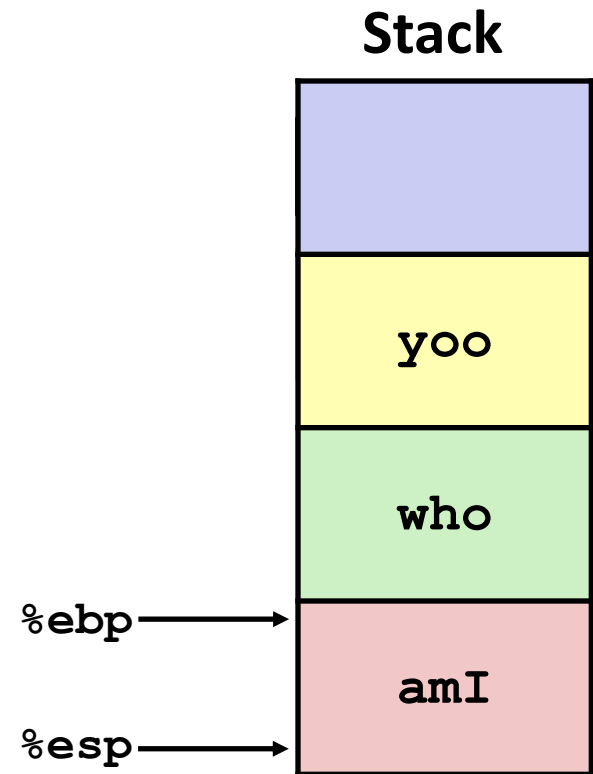
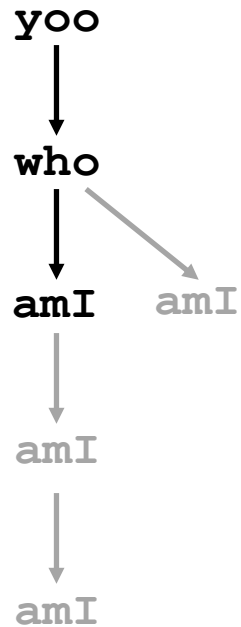
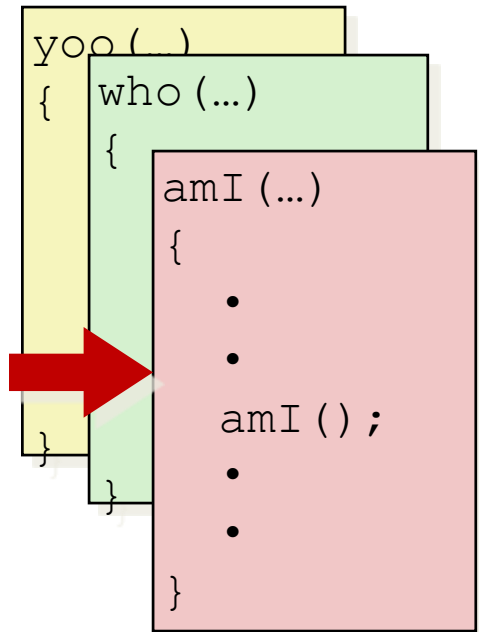
# Example



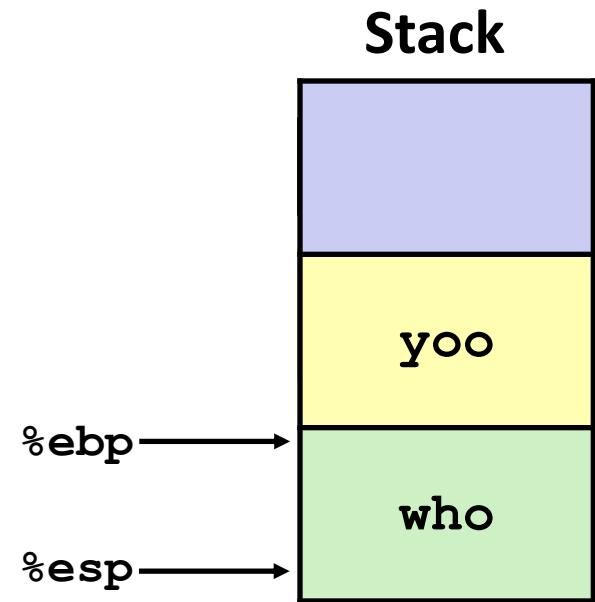
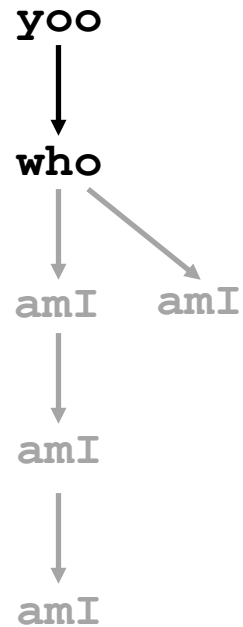
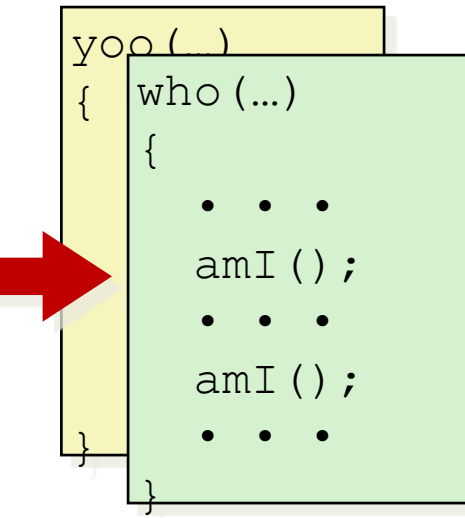
# Example



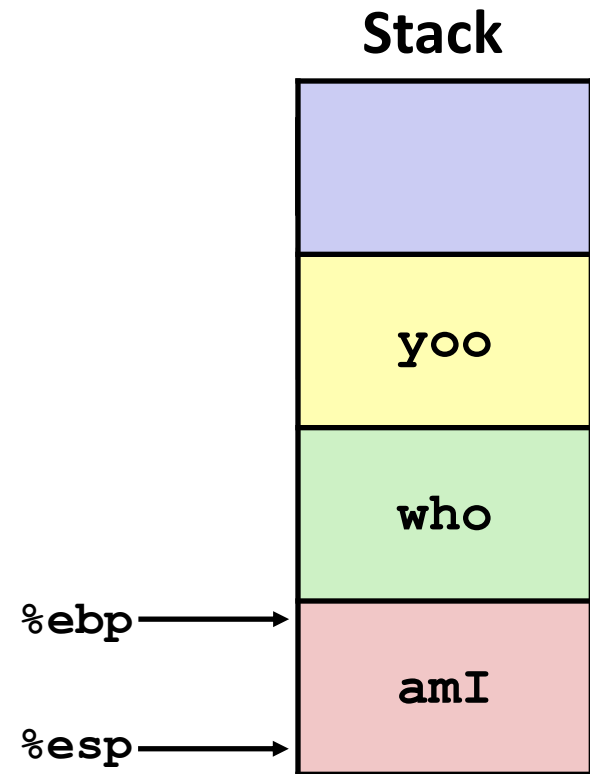
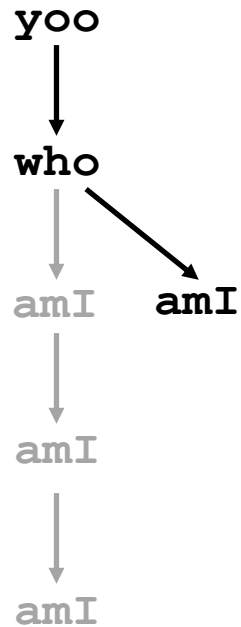
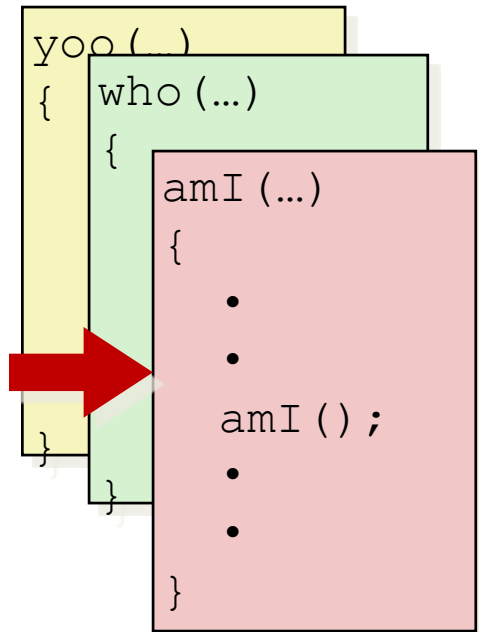
# Example



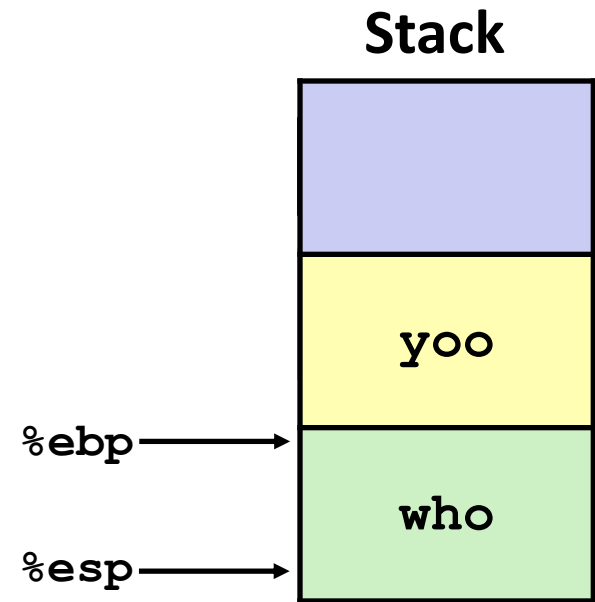
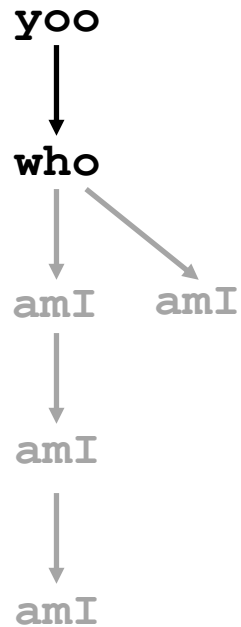
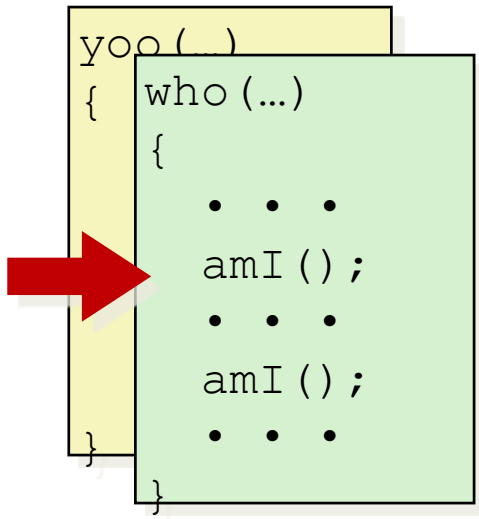
# Example



# Example

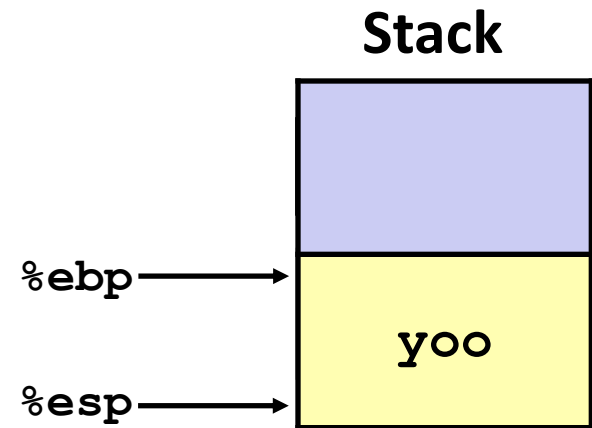
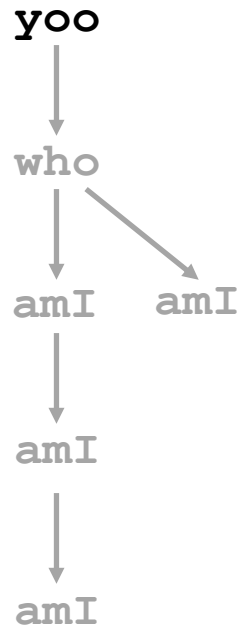
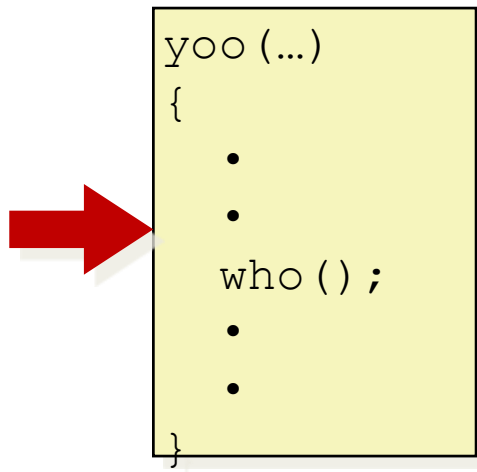


# Example





# Example



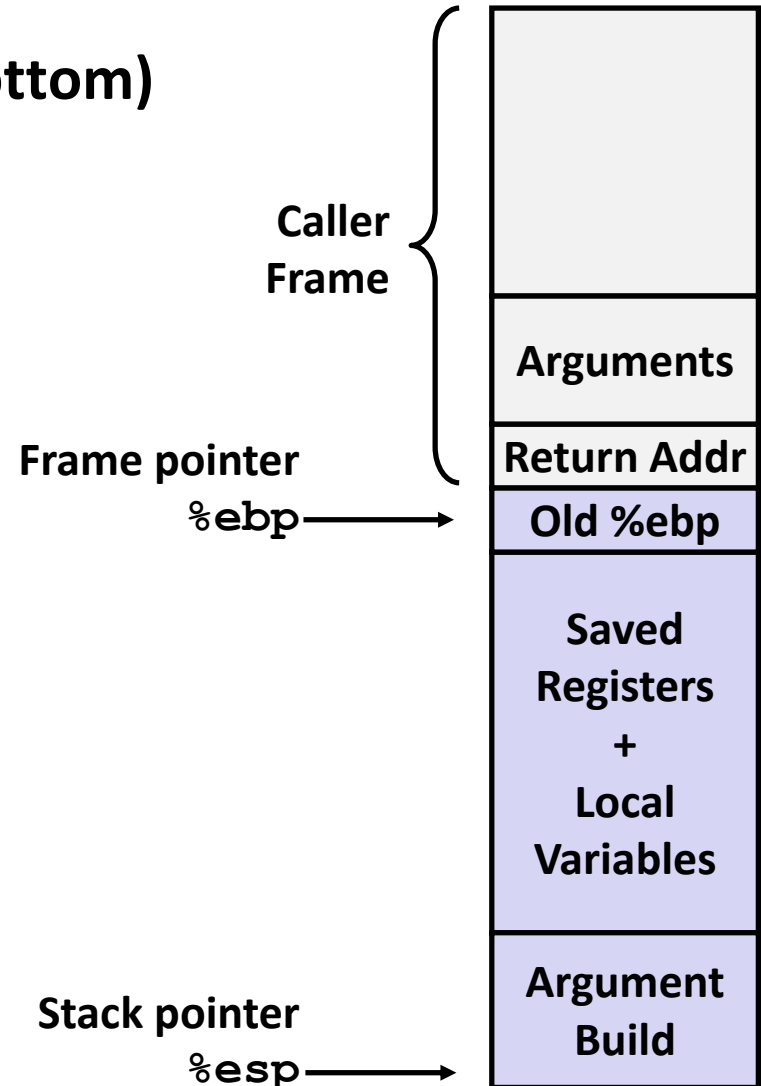
# IA32/Linux Stack Frame

## ■ Current Stack Frame (“Top” to Bottom)

- “Argument build:”  
Parameters for function about to call
- 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

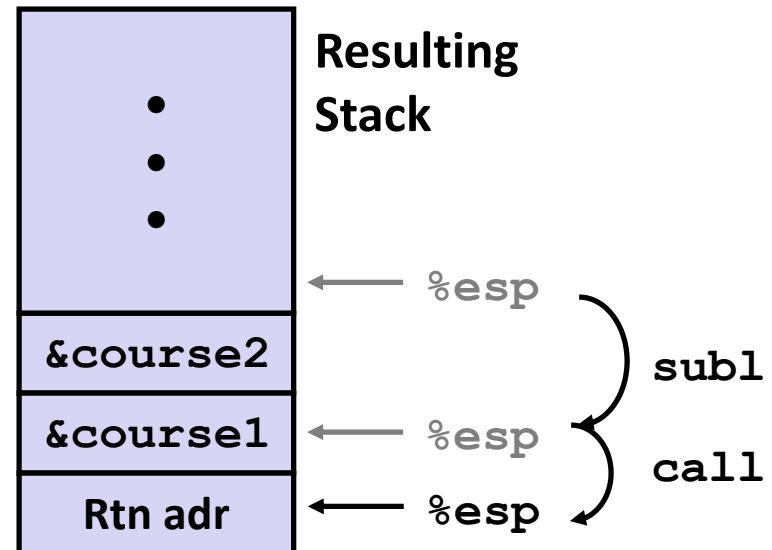
```
int course1 = 15213;
int course2 = 18243;

void call_swap() {
    swap(&course1, &course2);
}
```

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

## Calling swap from call\_swap

```
call_swap:
    . . .
    subl    $8, %esp
    movl    $course2, 4(%esp)
    movl    $course1, (%esp)
    call    swap
    . . .
```



# Revisiting swap

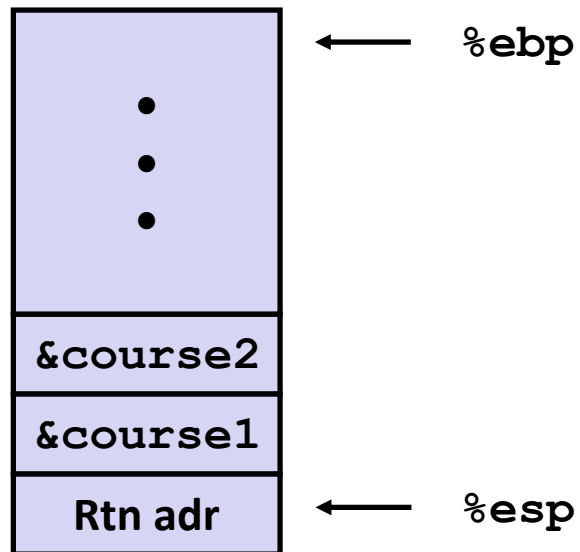
```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

swap:

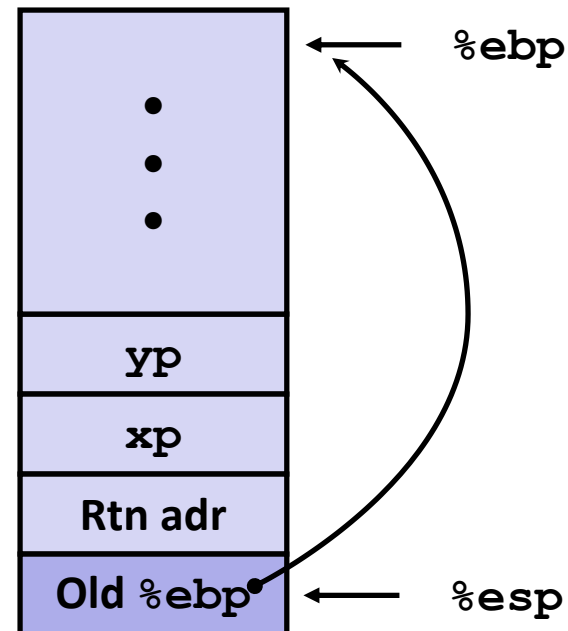
pushl %ebp	}	Set Up
movl %esp, %ebp		
pushl %ebx		
movl 8(%ebp), %edx	}	Body
movl 12(%ebp), %ecx		
movl (%edx), %ebx		
movl (%ecx), %eax		
movl %eax, (%edx)		
movl %ebx, (%ecx)		
popl %ebx	}	Finish
popl %ebp		
ret		

# swap Setup #1

## Entering Stack



## Resulting Stack



`swap:`

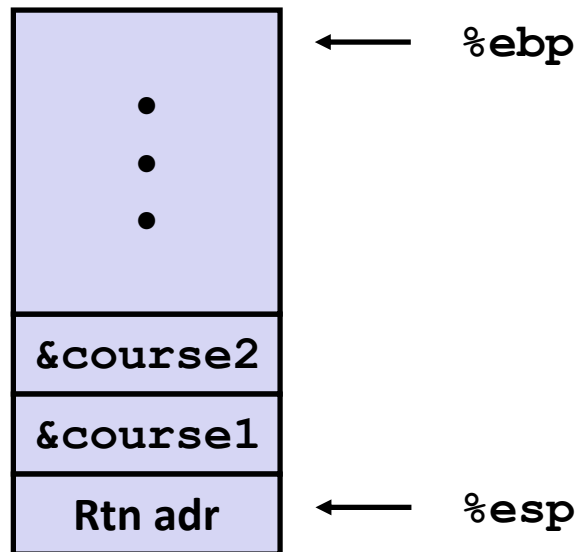
`pushl %ebp`

`movl %esp, %ebp`

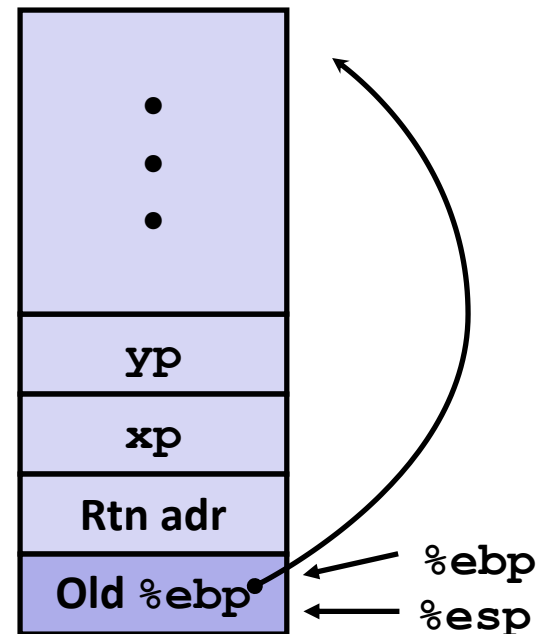
`pushl %ebx`

# swap Setup #2

## Entering Stack



## Resulting Stack



swap:

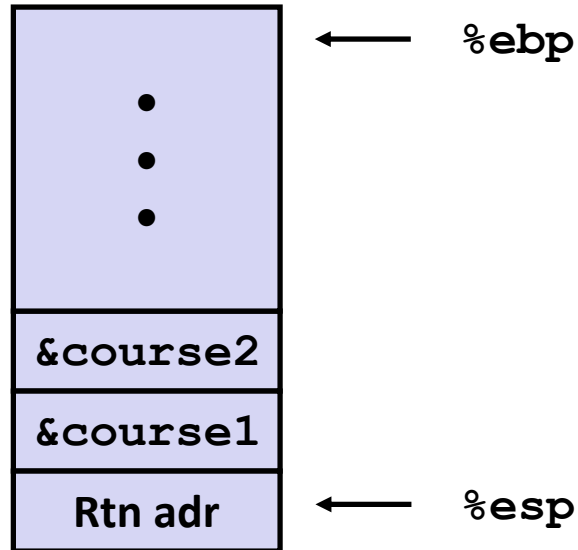
```
pushl %ebp
```

```
movl %esp, %ebp
```

```
pushl %ebx
```

# swap Setup #3

## Entering Stack



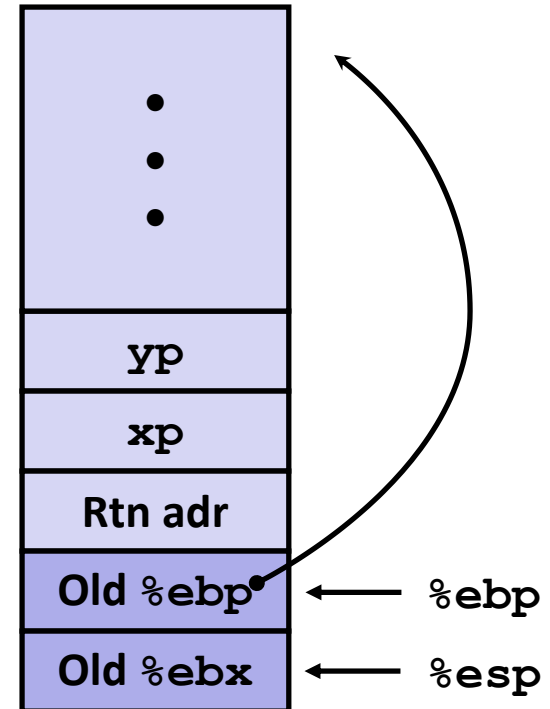
`swap:`

```

pushl %ebp
movl %esp, %ebp
pushl %ebx

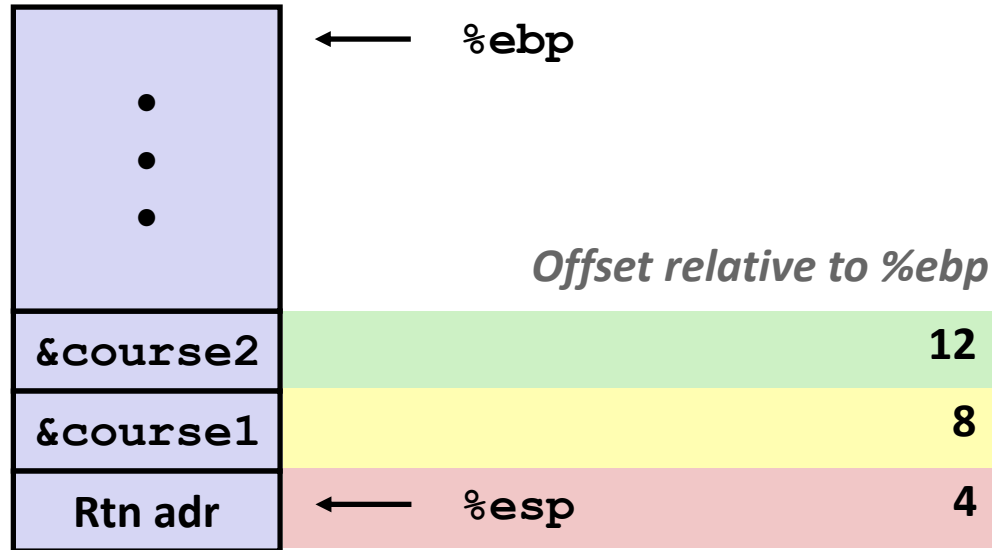
```

## Resulting Stack

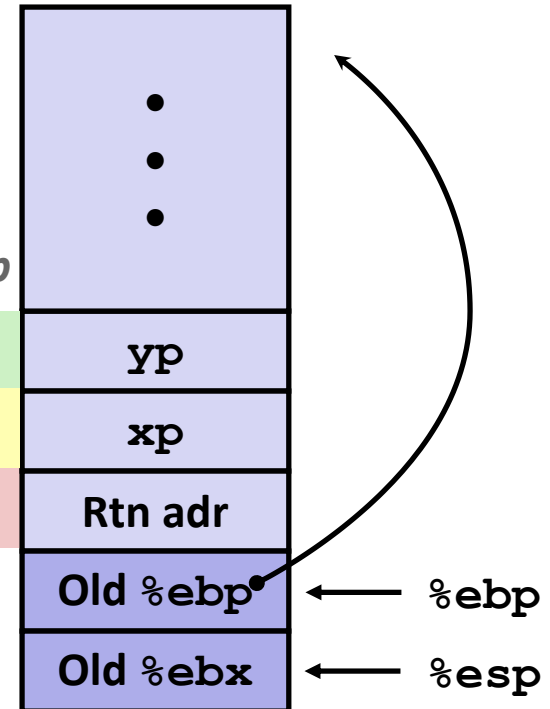


# swap Body

## Entering Stack



## Resulting Stack



```

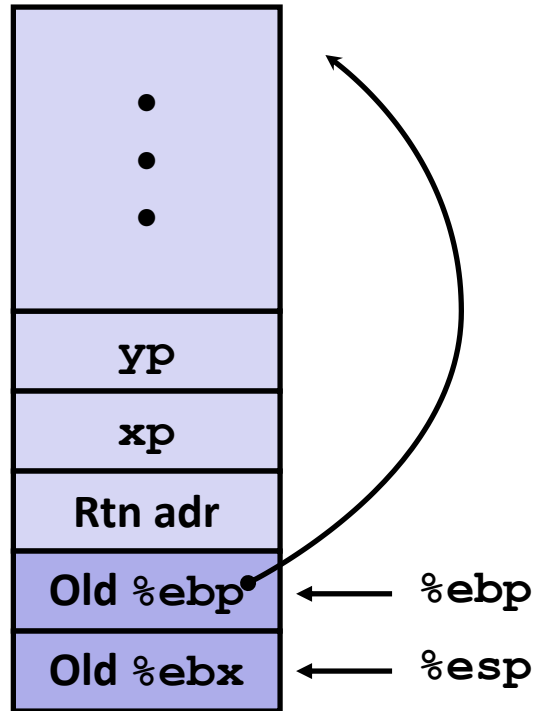
movl 8(%ebp),%edx    # get xp
movl 12(%ebp),%ecx   # get yp
. . .

```



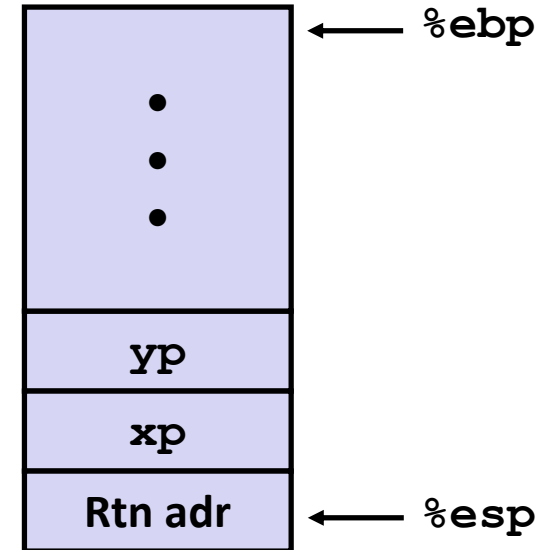
# swap Finish

## Stack Before Finish



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

## Resulting Stack



## ■ Observation

- Saved and restored register %ebx
- Not so for %eax, %ecx, %edx

# Disassembled swap

08048490 <swap>:

8048490:	55	push	%ebp
8048491:	89 e5	mov	%esp, %ebp
8048493:	53	push	%ebx
8048494:	8b 55 08	mov	0x8(%ebp), %edx
8048497:	8b 45 0c	mov	0xc(%ebp), %eax
804849a:	8b 0a	mov	(%edx), %ecx
804849c:	8b 18	mov	(%eax), %ebx
804849e:	89 1a	mov	%ebx, (%edx)
80484a0:	89 08	mov	%ecx, (%eax)
80484a2:	5b	pop	%ebx
80484a3:	5d	pop	%ebp
80484a4:	c3	ret	

## Calling Code

8048426:	c7 44 24 04 18 98 04	movl	\$0x8049818, 0x4(%esp)
804842d:	08		
804842e:	c7 04 24 1c 98 04 08	movl	\$0x804981c, (%esp)
8048435:	e8 56 00 00 00	call	<b>8048490</b> <swap>
804843a:	c9	leave	
804843b:	c3	ret	

# Today

- Switch statements
- **IA 32 Procedures**
  - Stack Structure
  - Calling Conventions
  - Illustrations of Recursion & Pointers

# 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 $18243, %edx
    . . .
    ret
```

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

# 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 values in its frame before the call
  - “*Callee Save*”
    - Callee saves temporary values in its frame before using

# IA32/Linux+Windows Register Usage

## ■ **%eax, %edx, %ecx**

- Caller saves prior to call if values are used later

## ■ **%eax**

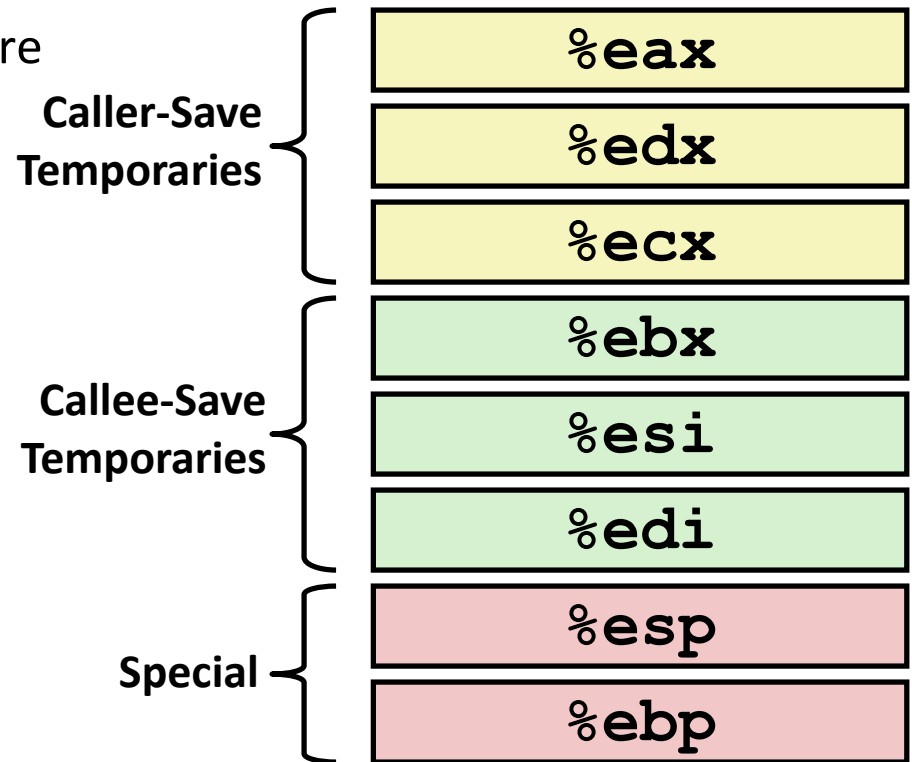
- also used to return integer value

## ■ **%ebx, %esi, %edi**

- Callee saves if wants to use them

## ■ **%esp, %ebp**

- special form of callee save
- Restored to original values upon exit from procedure



# Today

- Switch statements
- **IA 32 Procedures**
  - Stack Structure
  - Calling Conventions
  - Illustrations of Recursion & Pointers

# Recursive Function

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

## ■ Registers

- **%eax, %edx** used without first saving
- **%ebx** used, but saved at beginning & restored at end

```
pcount_r:
    pushl %ebp
    movl %esp, %ebp
    pushl %ebx
    subl $20, %esp
    movl 8(%ebp), %ebx
    movl $0, %eax
    testl %ebx, %ebx
    je .L3
    movl %ebx, %eax
    shrl %eax
    movl %eax, (%esp)
    call pcount_r
    movl %ebx, %edx
    andl $1, %edx
    addl %edx, %eax
.L3:
    addl $20, %esp
    popl %ebx
    popl %ebp
    ret
```



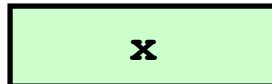
# Recursive Call #1

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

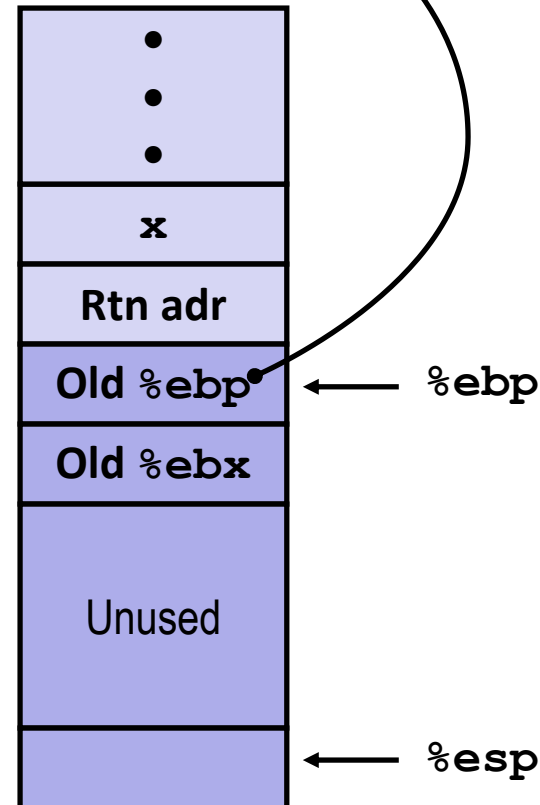
## ■ Actions

- Save old value of **%ebx** on stack
- Allocate space for argument to recursive call
- Store x in **%ebx**

**%ebx**



```
pcount_r:
    pushl %ebp
    movl  %esp, %ebp
    pushl %ebx
    subl  $20, %esp
    movl  8(%ebp), %ebx
    . . .
```



# Recursive Call #2

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

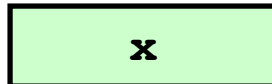
```

    . . .
movl    $0, %eax
testl   %ebx, %ebx
je      .L3
    . . .
.L3:
    . . .
ret
```

## ■ Actions

- If `x == 0`, return
  - with `%eax` set to 0

`%ebx`



# Recursive Call #3

```

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

```

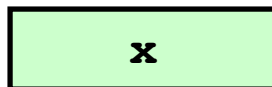
## ■ Actions

- Store  $x \gg 1$  on stack
- Make recursive call

## ■ Effect

- **%eax** set to function result
- **%ebx** still has value of  $x$

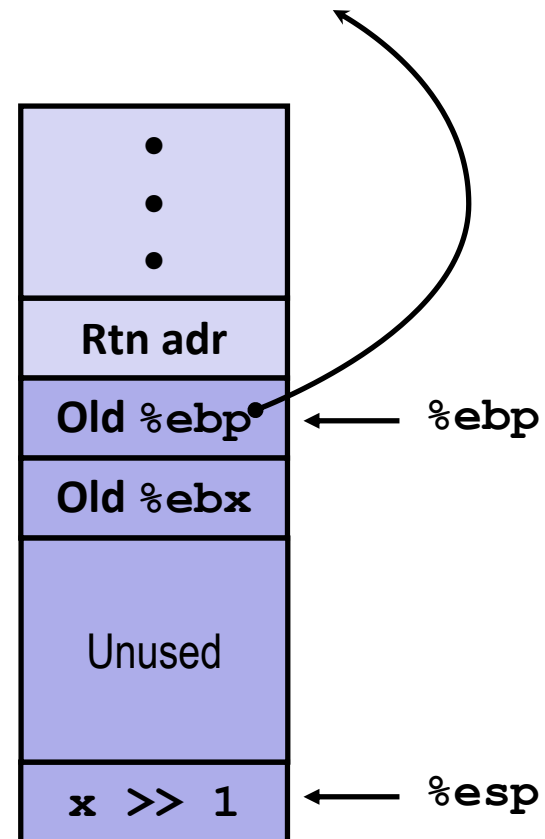
**%ebx**



```

. . .
movl  %ebx, %eax
shrl  %eax
movl  %eax, (%esp)
call  pcount_r
. . .

```



# Recursive Call #4

```

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

```

```

. . .
movl    %ebx, %edx
andl    $1, %edx
leal    (%edx,%eax), %eax
. . .

```

## ■ Assume

- `%eax` holds value from recursive call
- `%ebx` holds `x`

## ■ Actions

- Compute `(x & 1) + computed value`

## ■ Effect

- `%eax` set to function result



# Recursive Call #5

```

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

```

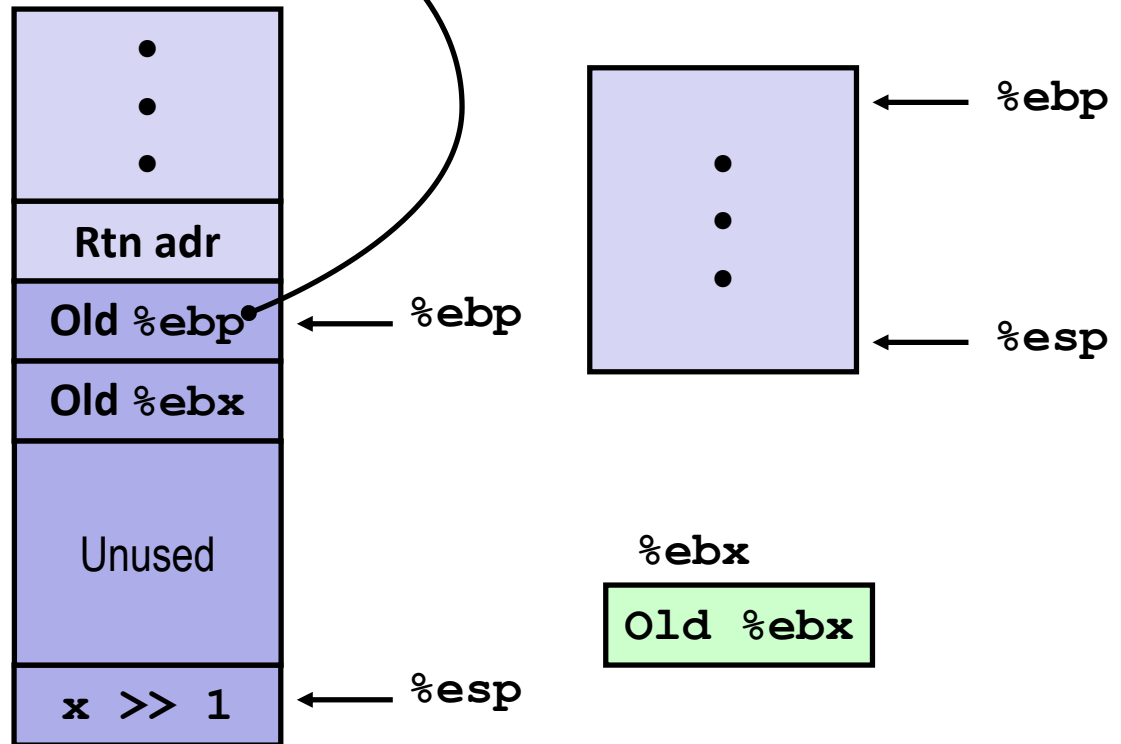
## ■ Actions

- Restore values of `%ebx` and `%ebp`
- Restore `%esp`

```

...
L3:
    addl    $20, %esp
    popl    %ebx
    popl    %ebp
    ret

```



# 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
- 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

# Pointer Code

## Generating Pointer

```
/* Compute x + 3 */  
int add3(int x) {  
    int localx = x;  
    incrk(&localx, 3);  
    return localx;  
}
```

## Referencing Pointer

```
/* Increment value by k */  
void incrk(int *ip, int k) {  
    *ip += k;  
}
```

- **add3** creates pointer and passes it to **incrk**

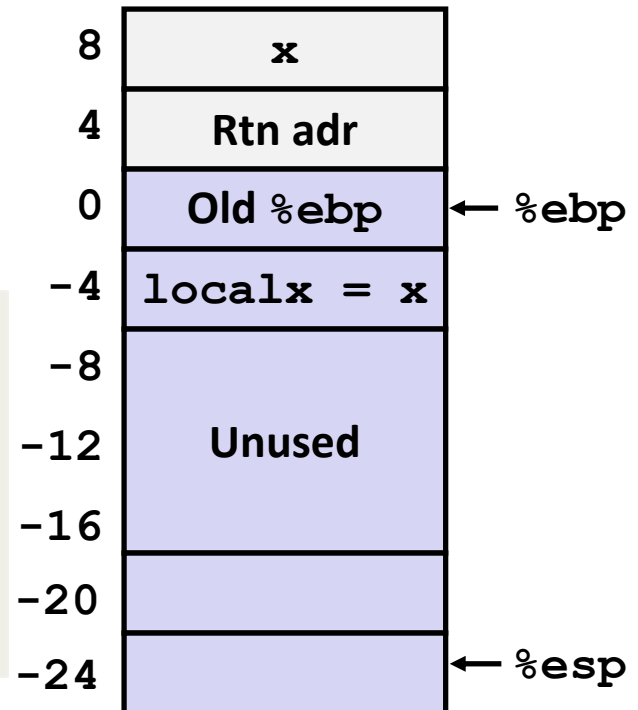
# Creating and Initializing Local Variable

```
int add3(int x) {
    int localx = x;
    incr(&localx, 3);
    return localx;
}
```

- Variable localx must be stored on stack
  - Because: Need to create pointer to it
- Compute pointer as  $-4(\%ebp)$

First part of add3

```
add3:
    pushl %ebp
    movl %esp, %ebp
    subl $24, %esp      # Alloc. 24 bytes
    movl 8(%ebp), %eax
    movl %eax, -4(%ebp) # Set localx to x
```





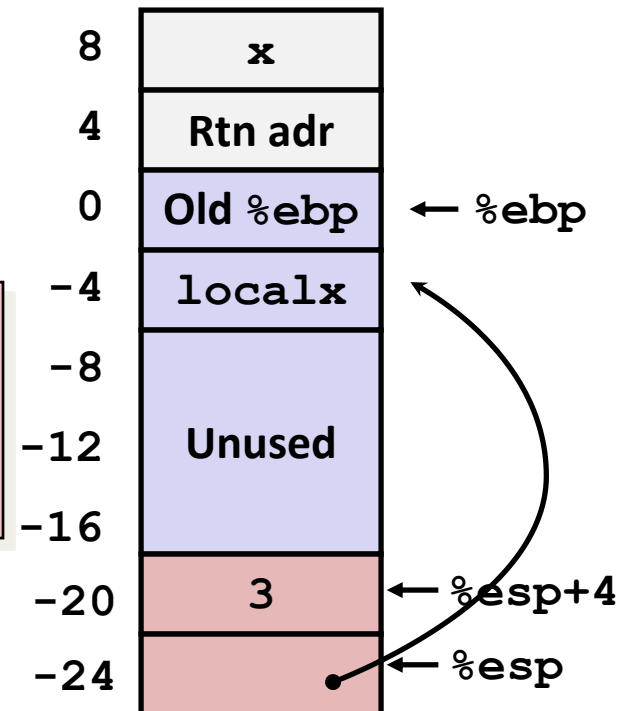
# Creating Pointer as Argument

```
int add3(int x) {
    int localx = x;
    incrk(&localx, 3);
    return localx;
}
```

- Use leal instruction to compute address of localx

## Middle part of add3

```
movl $3, 4(%esp)      # 2nd arg = 3
leal -4(%ebp), %eax   # &localx
movl %eax, (%esp)     # 1st arg = &localx
call incrk
```



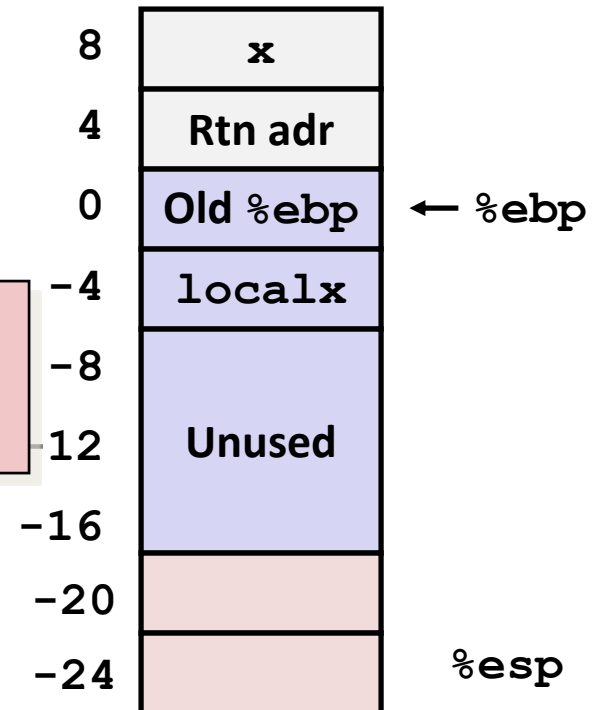
# Retrieving local variable

```
int add3(int x) {
    int localx = x;
    incr(&localx, 3);
    return localx;
}
```

- Retrieve localx from stack as return value

Final part of add3

```
movl -4(%ebp), %eax # Return val= localx
leave
ret
```



# IA32 Optimization: Inlining

## Generating Pointer

```
/* Compute x + 3 */  
int add3(int x) {  
    int localx = x;  
    incrk(&localx, 3);  
    return localx;  
}
```

## Referencing Pointer

```
/* Increment value by k */  
void incrk(int *ip, int k) {  
    *ip += k;  
}
```

- When both functions in same file:

```
add3:  
    pushl    %ebp  
    movl     %esp, %ebp  
    movl     8(%ebp), %eax  
    addl     $3, %eax  
    popl     %ebp  
    ret
```

# How Inlining Works

## Generating Pointer

```
/* Compute x + 3 */
int add3(int x) {
    int localx = x;
    incrk(&localx, 3);
    return localx;
}
```

## Expand callee into caller

```
/* Compute x + 3 */
int add3_inline(int x) {
    int localx = x;
    *(&localx) += 3;
    return localx;
}
```

## Referencing Pointer

```
/* Increment value by k */
void incrk(int *ip, int k) {
    *ip += k;
}
```

# IA 32 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 **%eax**

## ■ Pointers are addresses of values

- On stack or global

