

Introduction to Computer Systems

15-213/18-243, spring 2009

7th Lecture, Feb. 3rd

Instructors:

Gregory Kesden and Markus Püschel

Last Time

■ Complete memory addressing mode

- `(%eax), 17(%eax), 2(%ebx, %ecx, 8), ...`

■ Arithmetic operations

- `subl %eax, %ecx` # `ecx = ecx + eax`
- `sall $4, %edx` # `edx = edx << 4`
- `addl 16(%ebp), %ecx` # `ecx = ecx + Mem[16+ebp]`
- `leal 4(%edx, %eax), %eax` # `eax = 4 + edx + eax`
- `imull %ecx, %eax` # `eax = eax * ecx`

Last Time

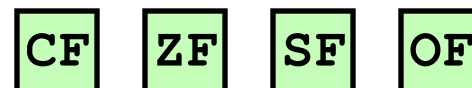
■ x86-64 vs. IA32

- Integer registers: **16 x 64-bit** vs. **8 x 32-bit**
- **movq, addq, ...** vs. **movl, addl, ...**
- Better support for passing function arguments in registers

%rax	%eax	%r8	%r8d
%rbx	%edx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%ebx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

■ Control

- Condition code registers
- Set as side effect or by **cmp, test**
- Used:
 - Read out by setx instructions (**setg, setle, ...**)
 - Or by conditional jumps (**jle .L4, je .L10, ...**)

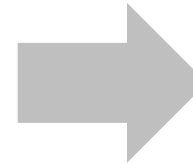


Last Time

■ Do-While loop

C Code

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



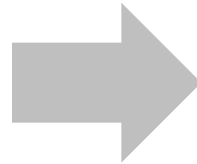
Goto Version

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

■ While-Do loop

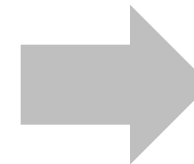
While version

```
while (Test)
  Body
```



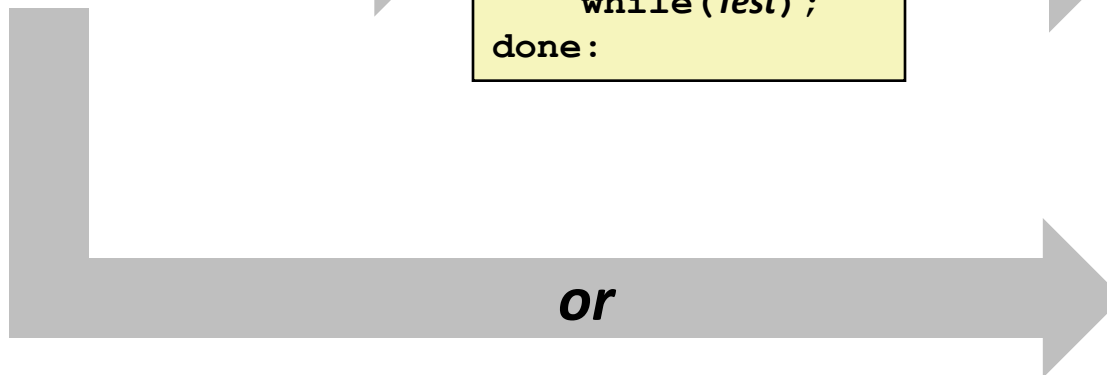
Do-While Version

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



Goto Version

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



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

Today

- For loops
- Switch statements
- Procedures

“For” Loop Example: Square-and-Multiply

```

/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned p)
{
    int result;
    for (result = 1; p != 0; p = p>>1) {
        if (p & 0x1)
            result *= x;
        x = x*x;
    }
    return result;
}

```

■ Algorithm

- Exploit bit representation: $p = p_0 + 2p_1 + 2^2p_2 + \dots + 2^{n-1}p_{n-1}$
- Gives: $x^p = z_0 \cdot z_1^2 \cdot (z_2^2)^2 \cdot \dots \cdot \underbrace{(\dots((z_{n-1}^2)^2)\dots)^2}_{n-1 \text{ times}}$
 - $z_i = 1$ when $p_i = 0$
 - $z_i = x$ when $p_i = 1$
- Complexity $O(\log p)$

Example

$$\begin{aligned}
 3^{10} &= 3^2 * 3^8 \\
 &= 3^2 * ((3^2)^2)^2
 \end{aligned}$$

ipwr Computation

```

/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned p)
{
    int result;
    for (result = 1; p != 0; p = p>>1) {
        if (p & 0x1)
            result *= x;
        x = x*x;
    }
    return result;
}

```

before iteration	result	x=3	p=10
1	1	3	10=1010 ₂
2	1	9	5= 101 ₂
3	9	81	2= 10 ₂
4	9	6561	1= 1 ₂
5	59049	43046721	0

“For” Loop Example

```
int result;
for (result = 1; p != 0; p = p>>1)
{
    if (p & 0x1)
        result *= x;
    x = x*x;
}
```

General Form

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

Test

```
p != 0
```

Init

```
result = 1
```

Update

```
p = p >> 1
```

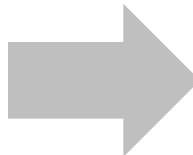
Body

```
{
    if (p & 0x1)
        result *= x;
    x = x*x;
}
```


“For” → “While” → “Do-While”

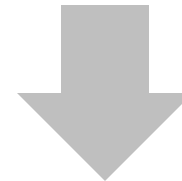
For Version

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



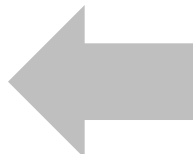
While Version

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



Goto Version

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



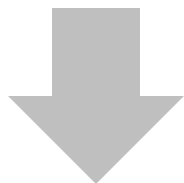
Do-While Version

```
Init;  
if (!Test)  
    goto done;  
do {  
    Body  
    Update ;  
} while (Test)  
done:
```

For-Loop: Compilation #1

For Version

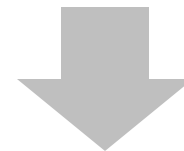
```
for (Init; Test; Update )  
    Body
```



Goto Version

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

```
for (result = 1; p != 0; p = p>>1)  
{  
    if (p & 0x1)  
        result *= x;  
    x = x*x;  
}
```

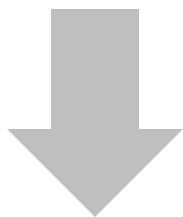


```
result = 1;  
if (p == 0)  
    goto done;  
loop:  
    if (p & 0x1)  
        result *= x;  
    x = x*x;  
    p = p >> 1;  
    if (p != 0)  
        goto loop;  
done:
```

“For” → “While” (Jump-to-Middle)

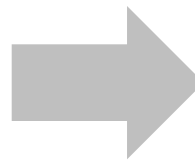
For Version

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



While Version

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



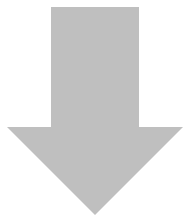
Goto Version

```
Init ;  
goto middle ;  
loop :  
    Body  
    Update ;  
middle :  
    if (Test)  
        goto loop ;  
done :
```

For-Loop: Compilation #2

For Version

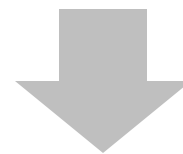
```
for (Init; Test; Update )  
    Body
```



Goto Version

```
Init;  
goto middle;  
loop:  
    Body  
    Update ;  
middle:  
    if (Test)  
        goto loop;  
done:
```

```
for (result = 1; p != 0; p = p>>1)  
{  
    if (p & 0x1)  
        result *= x;  
    x = x*x;  
}
```



```
result = 1;  
goto middle;  
loop:  
    if (p & 0x1)  
        result *= x;  
    x = x*x;  
    p = p >> 1;  
middle:  
    if (p != 0)  
        goto loop;  
done:
```

Today

- For loops
- **Switch statements**
- Procedures

```
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
}
```

Approximate Translation

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

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

Switch Statement Example (IA32)

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

```
Setup:  switch_eg:
        pushl %ebp                # Setup
        movl  %esp, %ebp         # Setup
        pushl %ebx               # Setup
        movl  $1, %ebx
        movl  8(%ebp), %edx
        movl  16(%ebp), %ecx
        cmpl  $6, %edx
        ja   .L61
        jmp  *.L62(, %edx, 4)
```

*Will disappear
Blackboard?*

Switch Statement Example (IA32)

```

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

```

```

Setup:  switch_eg:
        pushl %ebp                # Setup
        movl  %esp, %ebp         # Setup
        pushl %ebx               # Setup
        movl  $1, %ebx           # w = 1
        movl  8(%ebp), %edx       # edx = x
        movl  16(%ebp), %ecx      # ecx = z
        cmpl  $6, %edx           # x:6
        ja   .L61                # if > goto default
        jmp  *.L62(, %edx, 4)     # goto JTab[x]

```

*Indirect
jump*



Jump table

```

.section .rodata
    .align 4
.L62:
    .long   .L61   # x = 0
    .long   .L56   # x = 1
    .long   .L57   # x = 2
    .long   .L58   # x = 3
    .long   .L61   # x = 4
    .long   .L60   # x = 5
    .long   .L60   # x = 6

```

Assembly Setup Explanation

■ Table Structure

- Each target requires 4 bytes
- Base address at `.L62`

■ Jumping

Direct: `jmp .L61`

- Jump target is denoted by label `.L61`

Indirect: `jmp *.L62(, %edx, 4)`

- Start of jump table: `.L62`
- Must scale by factor of 4 (labels have 32-bit = 4 Bytes on IA32)
- Fetch target from effective Address `.L61 + edx*4`
 - Only for $0 \leq x \leq 6$

Jump table

```
.section .rodata
    .align 4
.L62:
    .long    .L61    # x = 0
    .long    .L56    # x = 1
    .long    .L57    # x = 2
    .long    .L58    # x = 3
    .long    .L61    # x = 4
    .long    .L60    # x = 5
    .long    .L60    # x = 6
```

Jump Table

Jump table

```
.section .rodata
    .align 4
.L62:
    .long    .L61    # x = 0
    .long    .L56    # x = 1
    .long    .L57    # x = 2
    .long    .L58    # x = 3
    .long    .L61    # x = 4
    .long    .L60    # x = 5
    .long    .L60    # x = 6
```

```
switch(x) {
case 1:      // .L56
    w = y*z;
    break;
case 2:      // .L57
    w = y/z;
    /* Fall Through */
case 3:      // .L58
    w += z;
    break;
case 5:
case 6:      // .L60
    w -= z;
    break;
default:    // .L61
    w = 2;
}
```

Code Blocks (Partial)

```
switch(x) {  
    . . .  
case 2:      // .L57  
    w = y/z;  
    /* Fall Through */  
case 3:      // .L58  
    w += z;  
    break;  
    . . .  
default:    // .L61  
    w = 2;  
}
```

```
.L61: // Default case  
    movl    $2, %ebx      # w = 2  
    movl    %ebx, %eax    # Return w  
    popl    %ebx  
    leave  
    ret  
  
.L57: // Case 2:  
    movl    12(%ebp), %eax # y  
    cld  
    idivl   %ecx          # Div prep  
    movl    %eax, %ebx    # w = y/z  
  
# Fall through  
  
.L58: // Case 3:  
    addl    %ecx, %ebx    # w+= z  
    movl    %ebx, %eax    # Return w  
    popl    %ebx  
    leave  
    ret
```

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:      // .L50
    w = y*z;
    break;
    . . .
}
```

```
.L50: // Case 1:
    movq    %rsi, %r8    # w = y
    imulq  %rdx, %r8    # w *= z
    movq    %r8, %rax    # Return w
    ret
```

Jump Table

```
.section .rodata
    .align 8
.L62:
    .quad   .L55    # x = 0
    .quad   .L50    # x = 1
    .quad   .L51    # x = 2
    .quad   .L52    # x = 3
    .quad   .L55    # x = 4
    .quad   .L54    # x = 5
    .quad   .L54    # x = 6
```

IA32 Object Code

■ Setup

- Label `.L61` becomes address `0x8048630`
- Label `.L62` becomes address `0x80488dc`

Assembly Code

```
switch_eg:
    . . .
    ja     .L61          # if > goto default
    jmp   *.L62(, %edx, 4) # goto JTab[x]
```

Disassembled Object Code

```
08048610 <switch_eg>:
    . . .
    8048622:  77 0c                ja     8048630
    8048624:  ff 24 95 dc 88 04 08 jmp   *0x80488dc(, %edx, 4)
```

IA32 Object Code (cont.)

■ Jump Table

- Doesn't show up in disassembled code
- Can inspect using GDB

```
gdb asm-cnt1
```

```
(gdb) x/7xw 0x80488dc
```

- Examine 7 hexadecimal format "words" (4-bytes each)
- Use command "**help x**" to get format documentation

```
0x80488dc:
```

```
0x08048630
```

```
0x08048650
```

```
0x0804863a
```

```
0x08048642
```

```
0x08048630
```

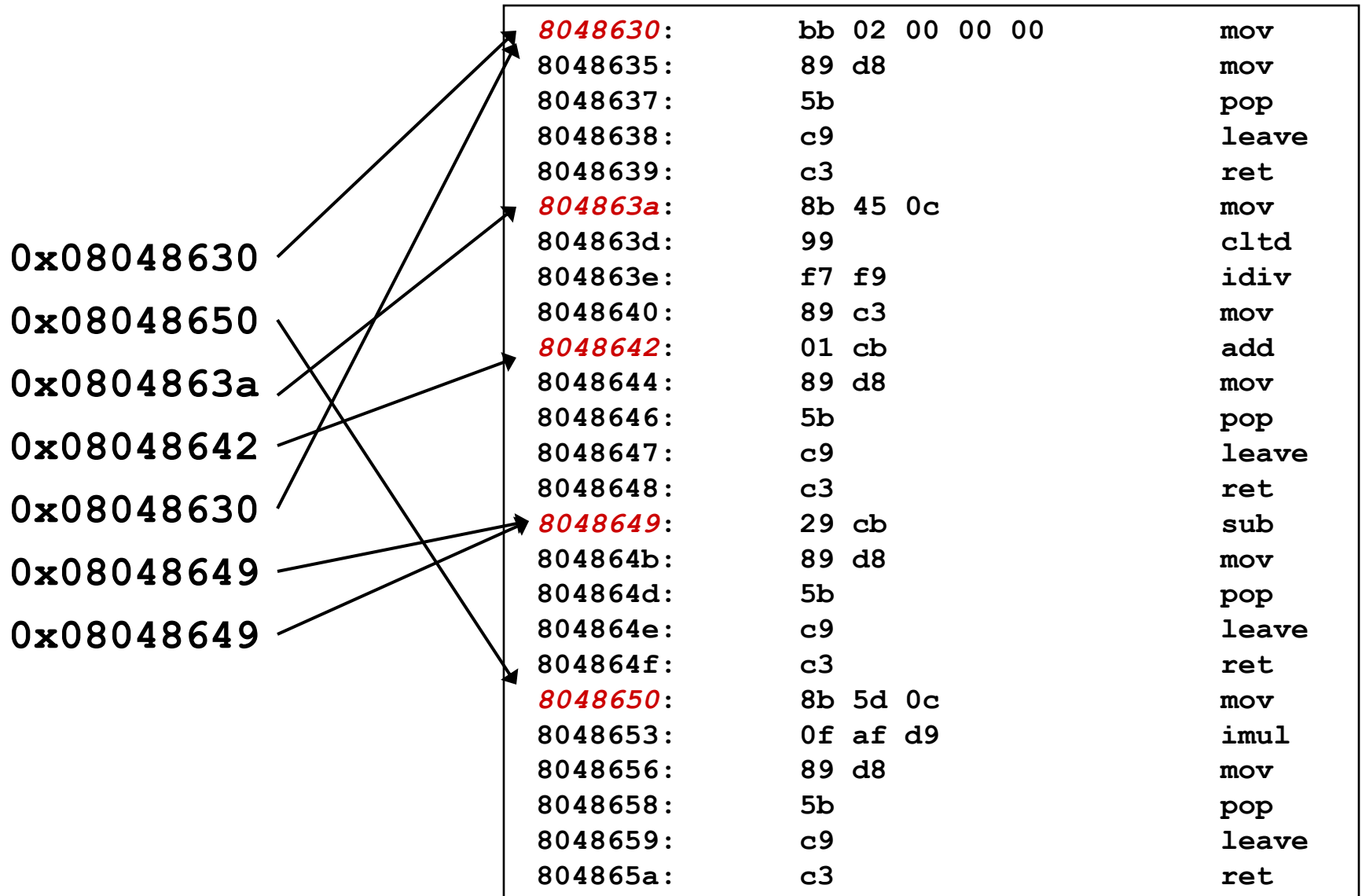
```
0x08048649
```

```
0x08048649
```

Disassembled Targets

8048630:	bb 02 00 00 00	mov	\$0x2, %ebx
8048635:	89 d8	mov	%ebx, %eax
8048637:	5b	pop	%ebx
8048638:	c9	leave	
8048639:	c3	ret	
804863a:	8b 45 0c	mov	0xc(%ebp), %eax
804863d:	99	cld	
804863e:	f7 f9	idiv	%ecx
8048640:	89 c3	mov	%eax, %ebx
8048642:	01 cb	add	%ecx, %ebx
8048644:	89 d8	mov	%ebx, %eax
8048646:	5b	pop	%ebx
8048647:	c9	leave	
8048648:	c3	ret	
8048649:	29 cb	sub	%ecx, %ebx
804864b:	89 d8	mov	%ebx, %eax
804864d:	5b	pop	%ebx
804864e:	c9	leave	
804864f:	c3	ret	
8048650:	8b 5d 0c	mov	0xc(%ebp), %ebx
8048653:	0f af d9	imul	%ecx, %ebx
8048656:	89 d8	mov	%ebx, %eax
8048658:	5b	pop	%ebx
8048659:	c9	leave	
804865a:	c3	ret	

Matching Disassembled Targets



x86-64 Object Code

■ Setup

- Label `.L61` becomes address `0x0000000000400716`
- Label `.L62` becomes address `0x0000000000400990`

Assembly Code

```
switch_eg:
    . . .
    ja     .L55          # if > goto default
    jmp   *.L56(,%rdi,8) # goto JTab[x]
```

Disassembled Object Code

```
0000000000400700 <switch_eg>:
    . . .
    40070d: 77 07          ja     400716
    40070f: ff 24 fd 90 09 40 00 jmpq  *0x400990(,%rdi,8)
```

x86-64 Object Code (cont.)

■ Jump Table

- Can inspect using GDB

```
gdb asm-cnt1
```

```
(gdb) x/7xg 0x400990
```

- Examine 7 hexadecimal format “giant words” (8-bytes each)
- Use command “**help x**” to get format documentation

```
0x400990:
```

```
0x000000000000400716
```

```
0x000000000000400739
```

```
0x000000000000400720
```

```
0x00000000000040072b
```

```
0x000000000000400716
```

```
0x000000000000400732
```

```
0x000000000000400732
```

Summarizing

■ C Control

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

■ Assembler Control

- Conditional jump
- Conditional move
- Indirect jump
- Compiler
- Must generate assembly code to implement more complex control

■ Standard Techniques

- IA32 loops converted to do-while form
- x86-64 loops use jump-to-middle
- Large switch statements use jump tables
- Sparse switch statements may use decision trees (not shown)

■ Conditions in CISC

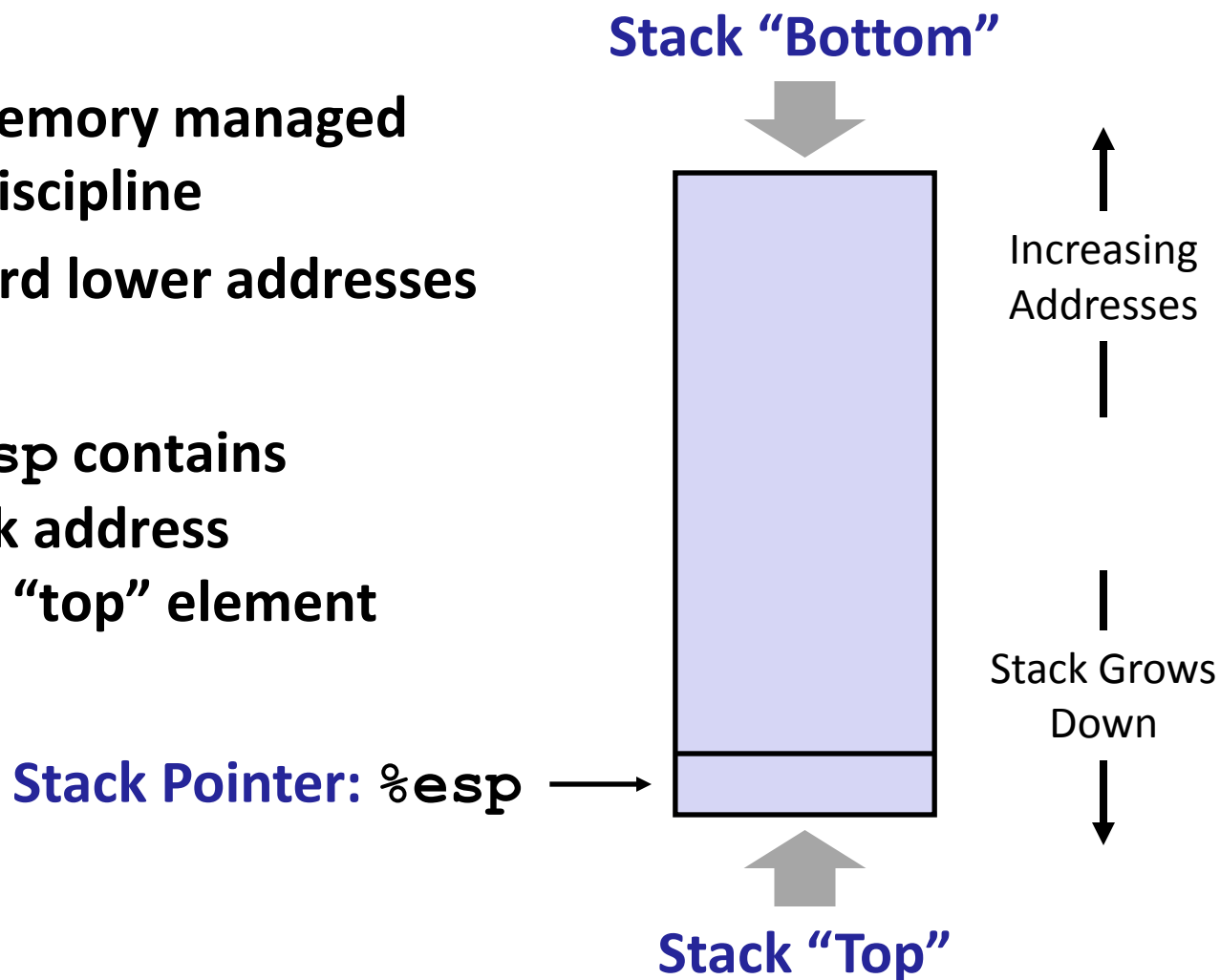
- CISC machines generally have condition code registers

Today

- For loops
- Switch statements
- **Procedures**

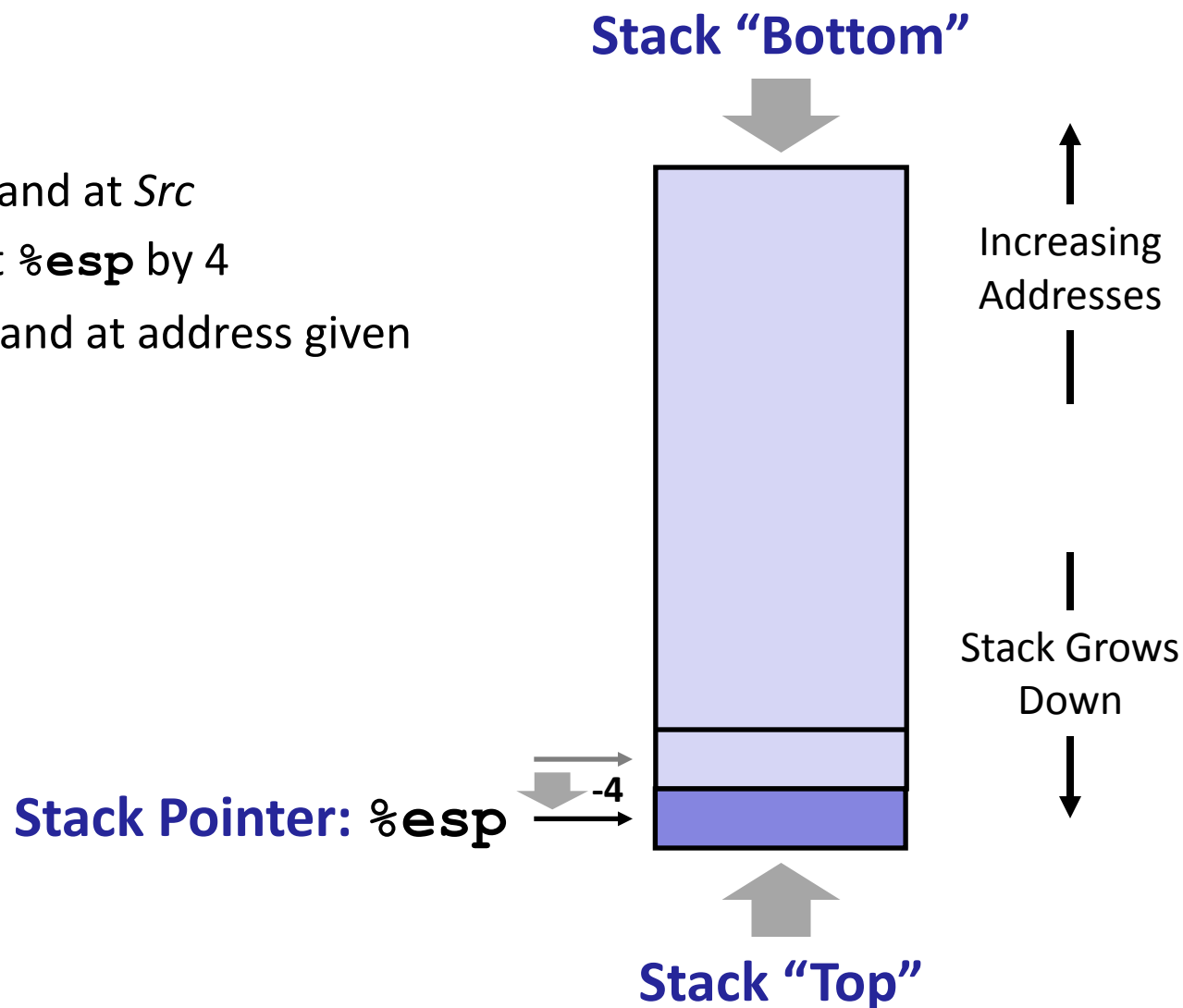
IA32 Stack

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



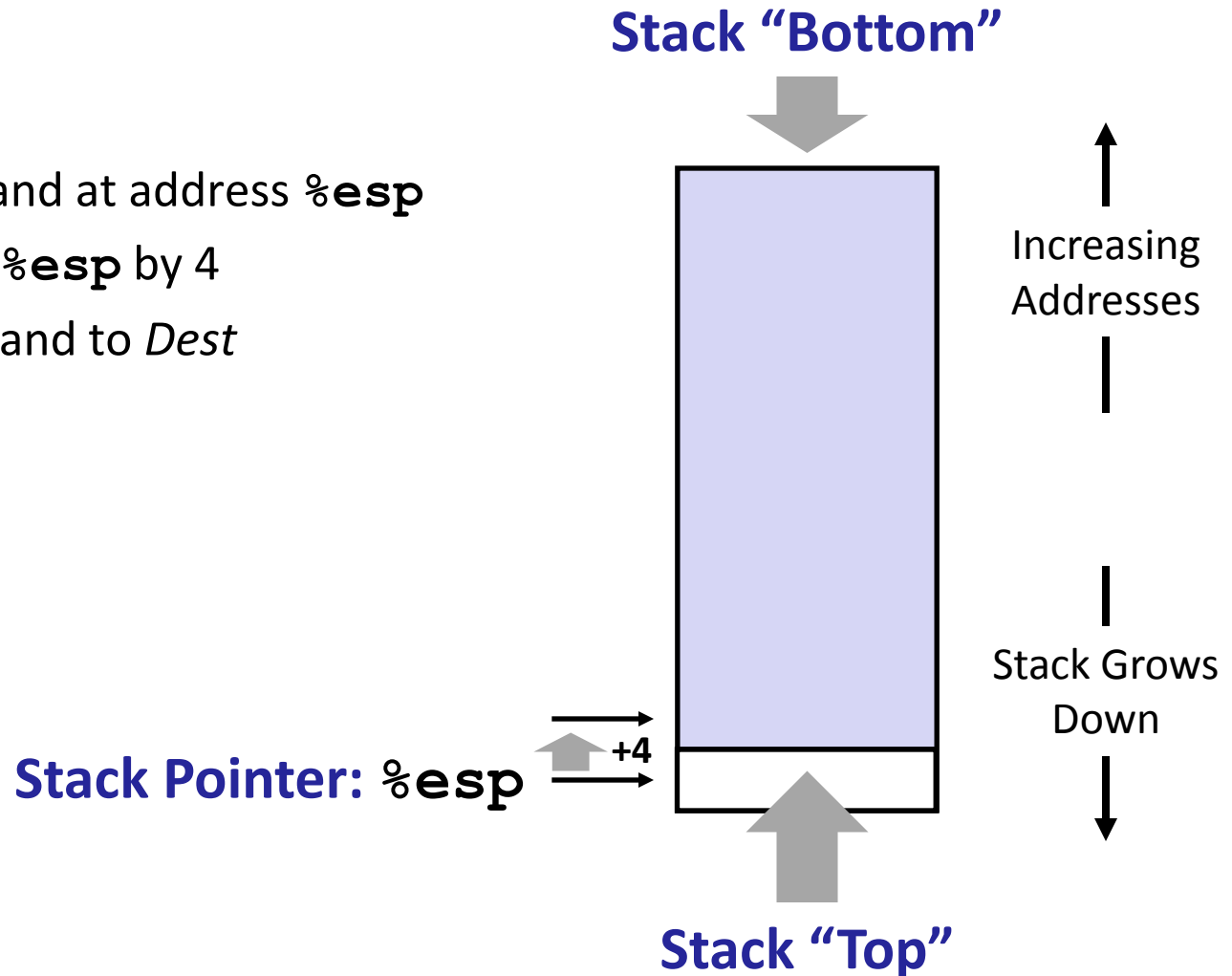
IA32 Stack: Push

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



IA32 Stack: Pop

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



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 instruction beyond `call`
- Example from disassembly

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

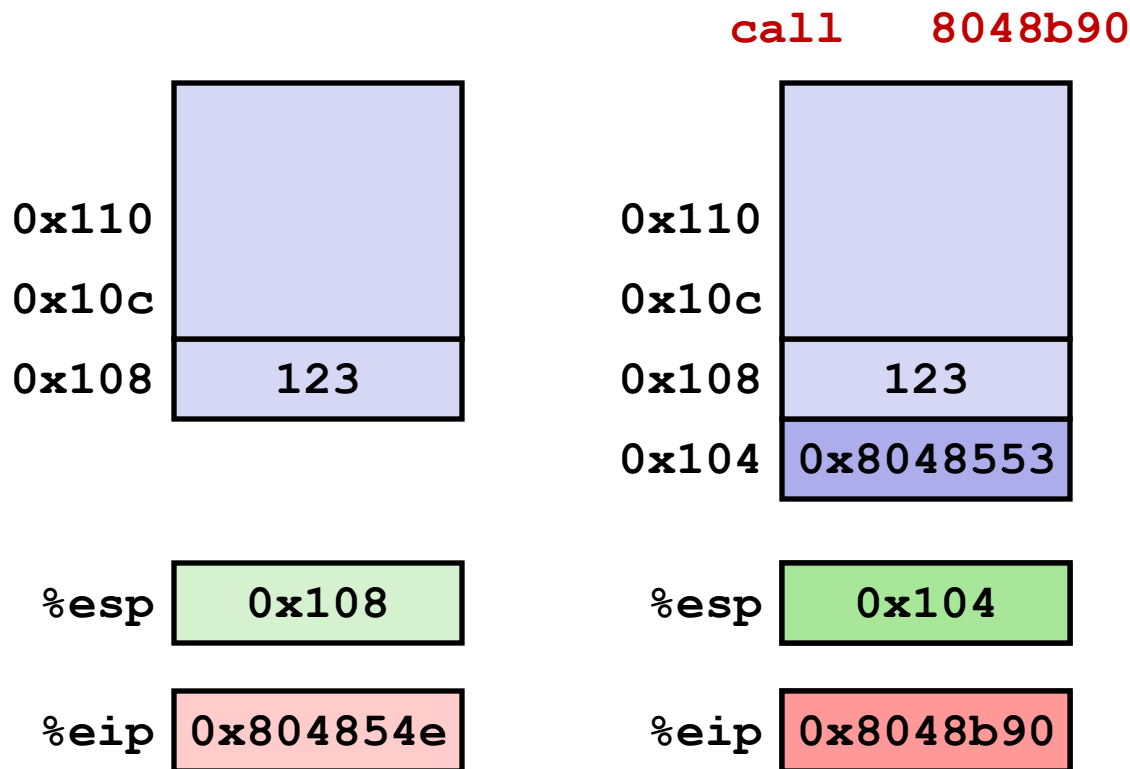
- Return address = `0x8048553`

■ Procedure return: `ret`

- Pop address from stack
- Jump to address

Procedure Call Example

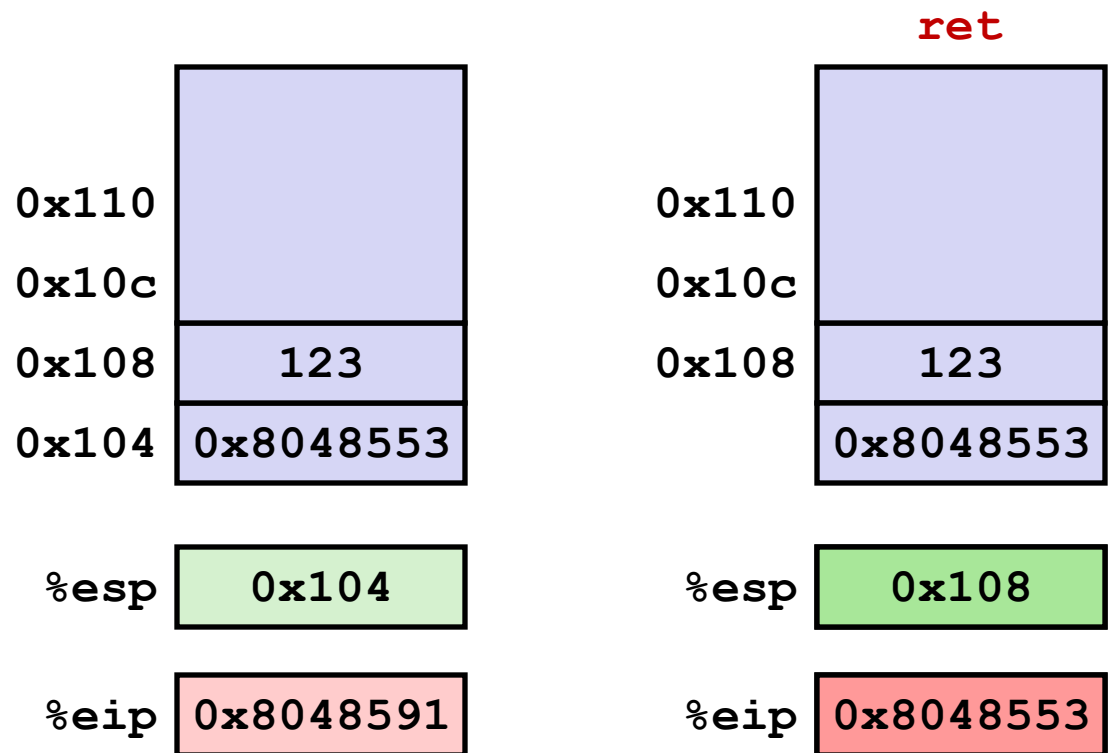
804854e:	e8 3d 06 00 00	call	8048b90 <main>
8048553:	50	pushl	%eax



%eip: 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 ();
  • • •
}

```

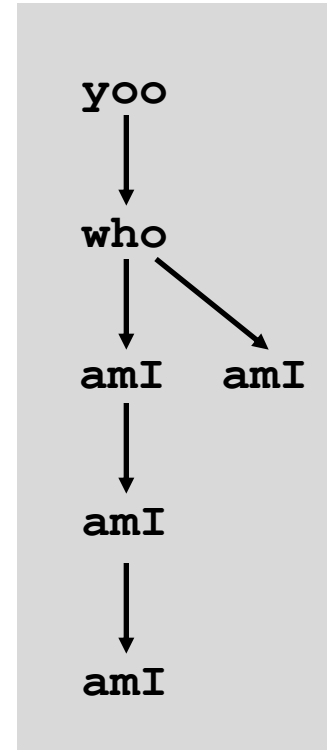
```

amI (...)
{
  •
  •
  amI ();
  •
  •
}

```

Procedure `amI` is recursive

Example Call Chain



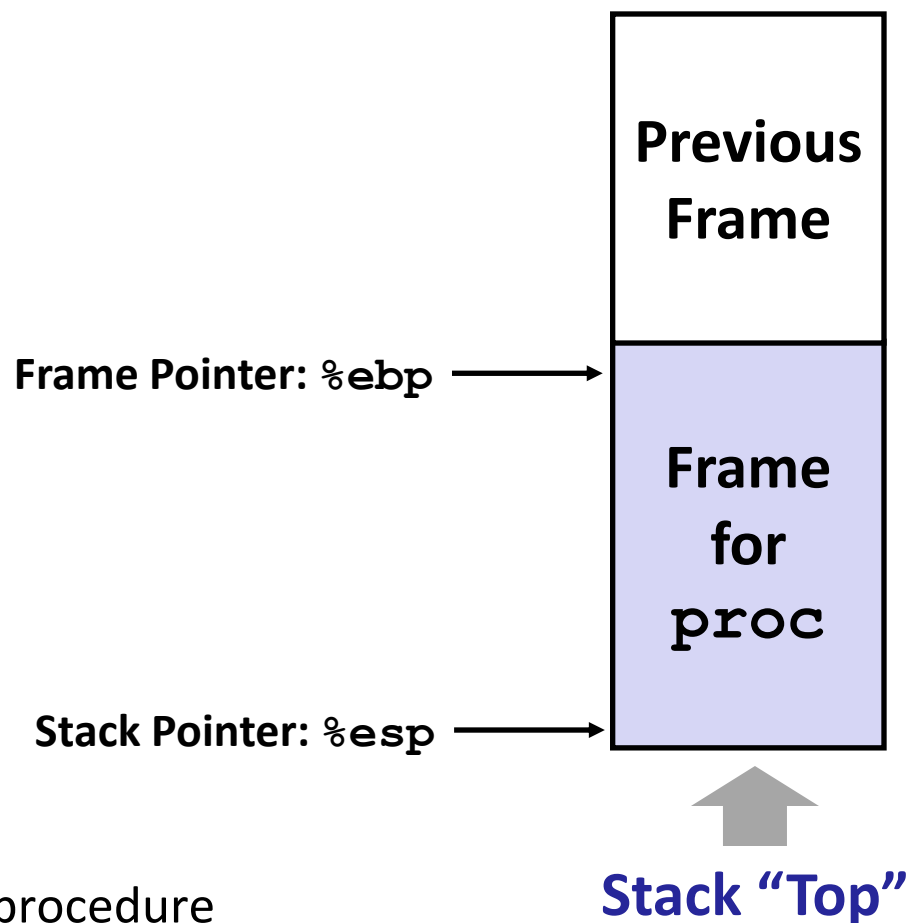
Stack Frames

■ Contents

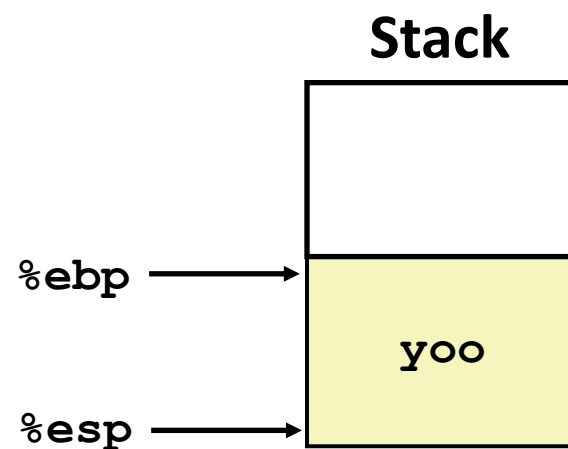
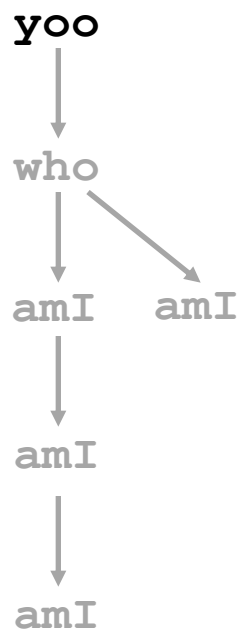
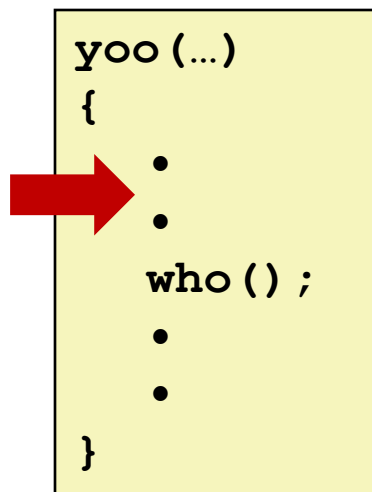
- Local variables
- Return information
- Temporary space

■ Management

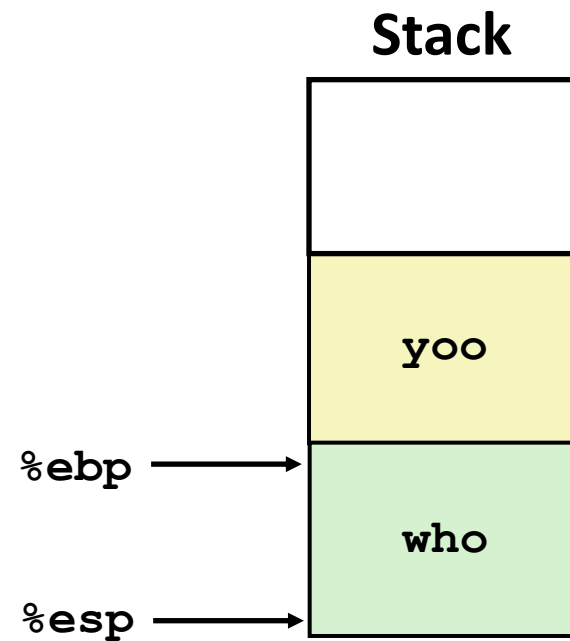
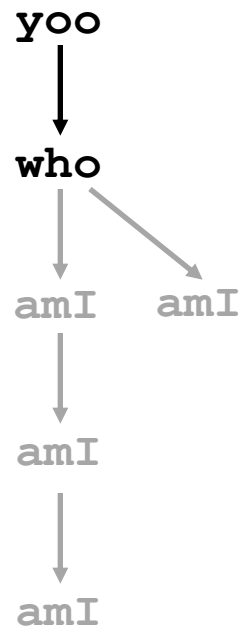
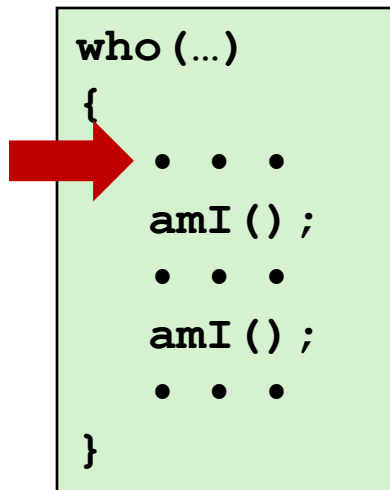
- Space allocated when enter procedure
 - “Set-up” code
- Deallocated when return
 - “Finish” code



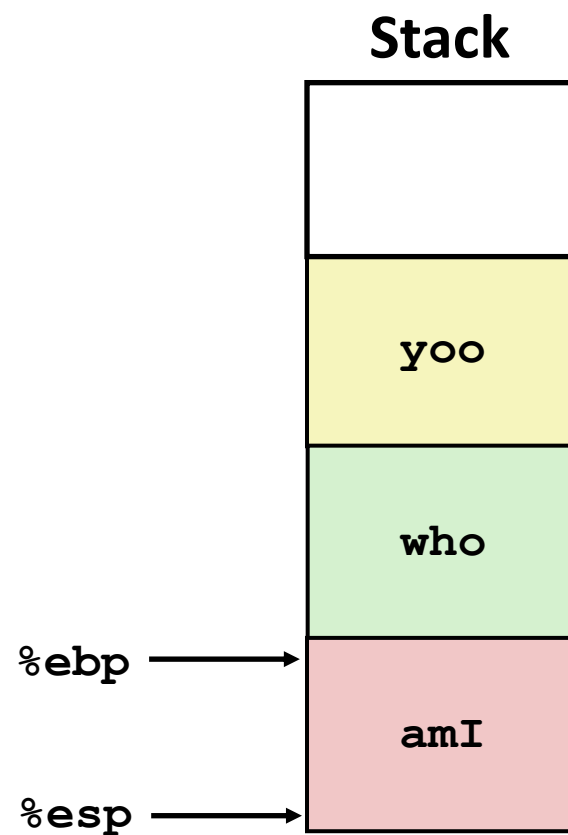
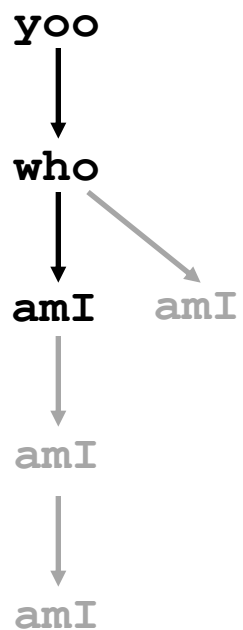
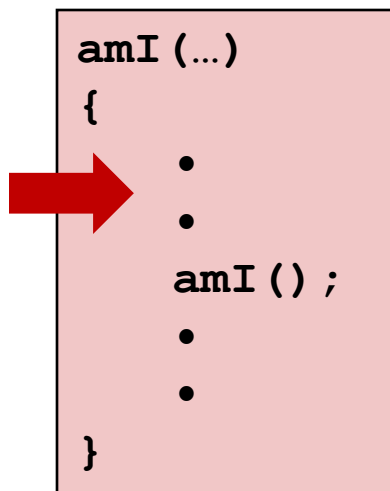
Example



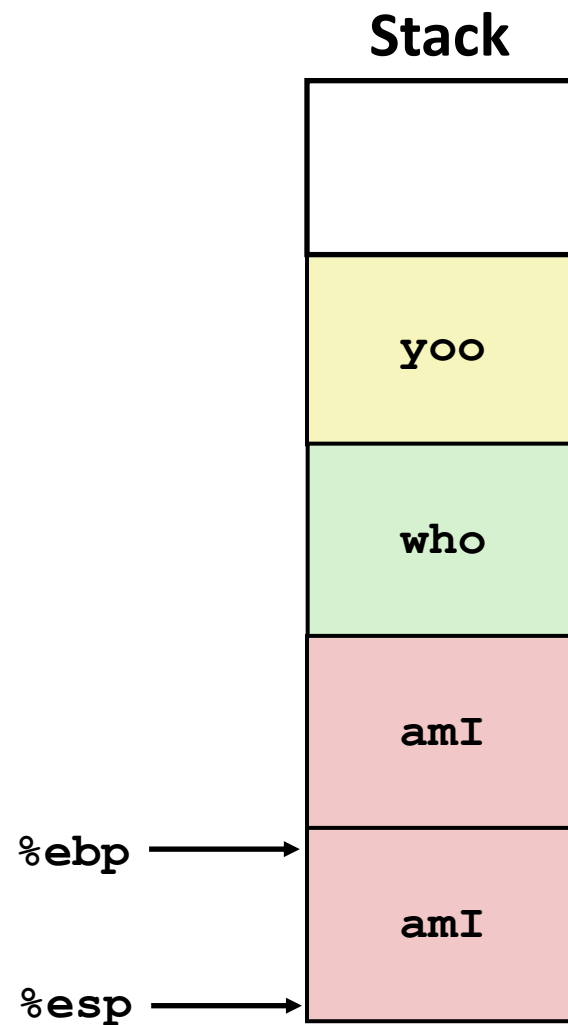
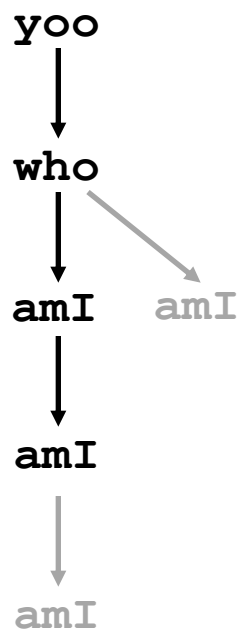
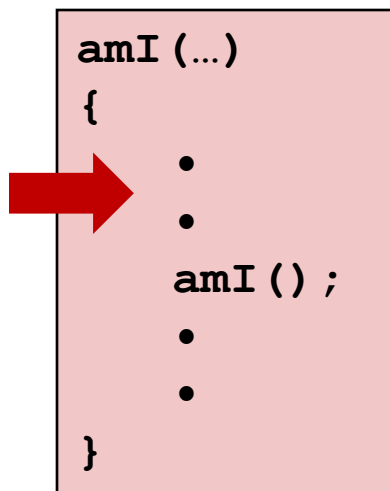
Example



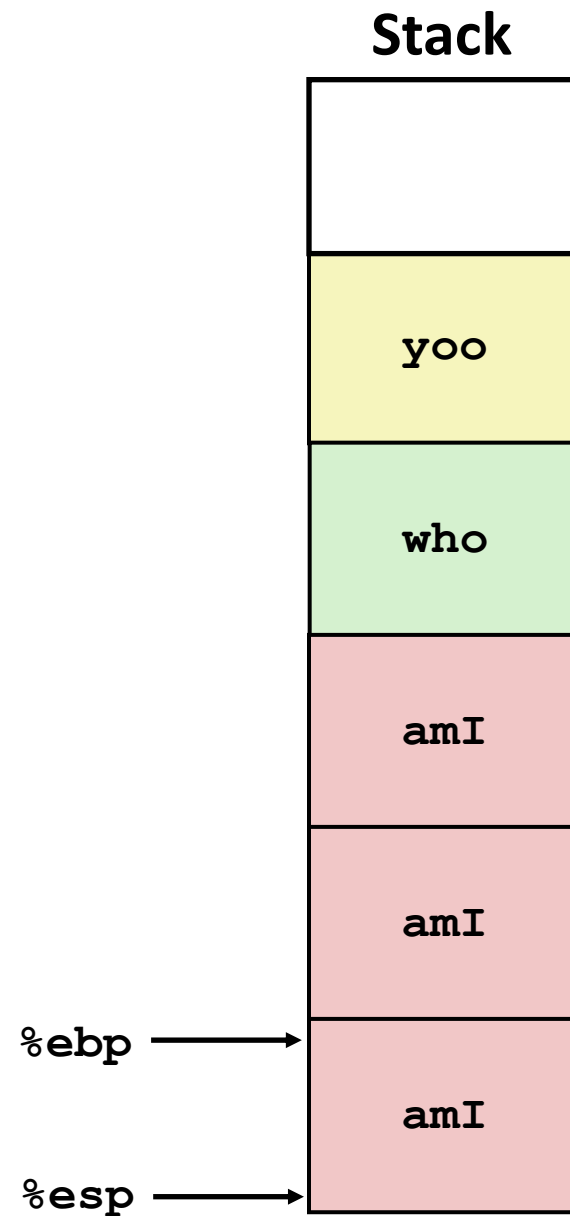
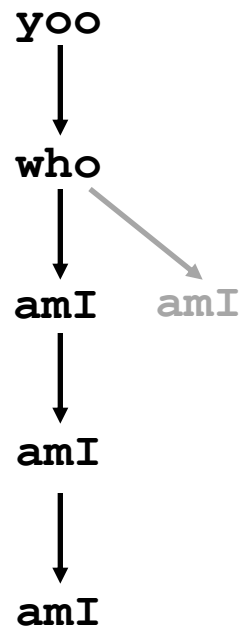
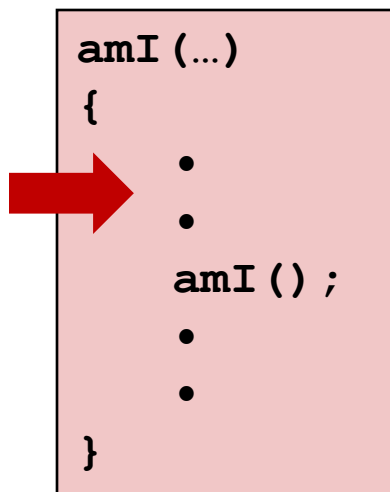
Example



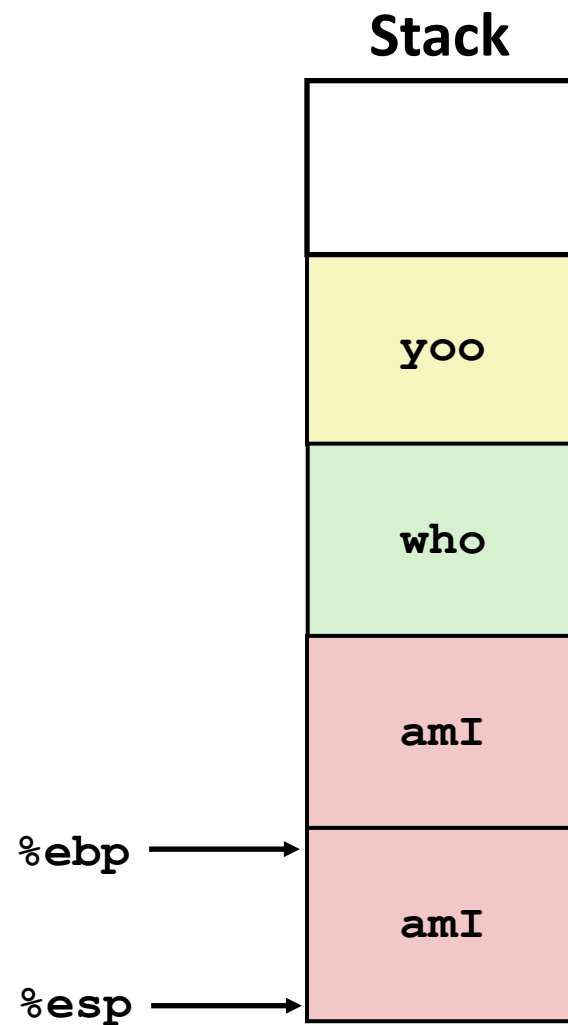
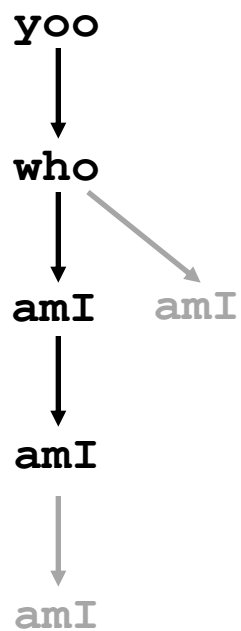
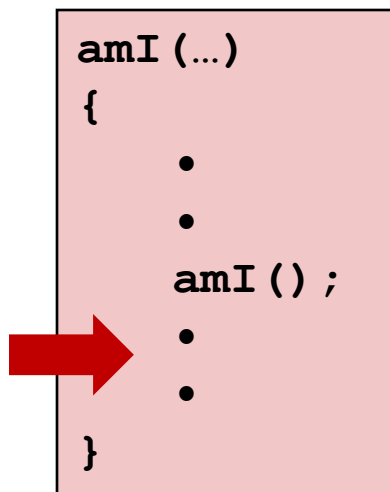
Example



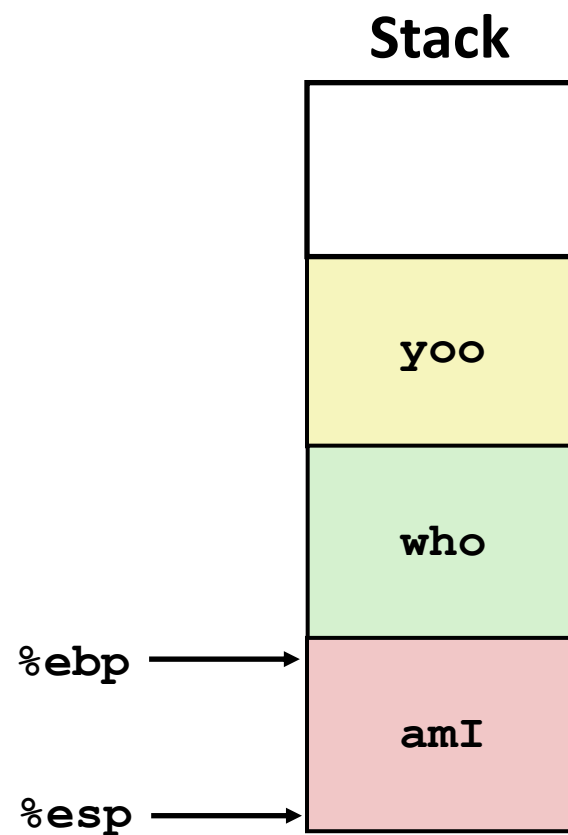
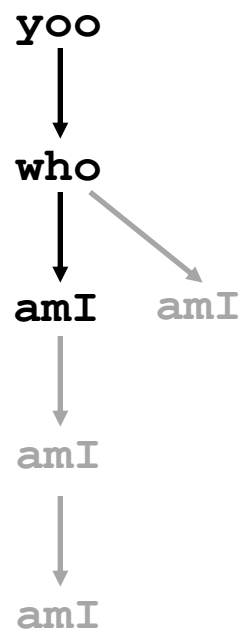
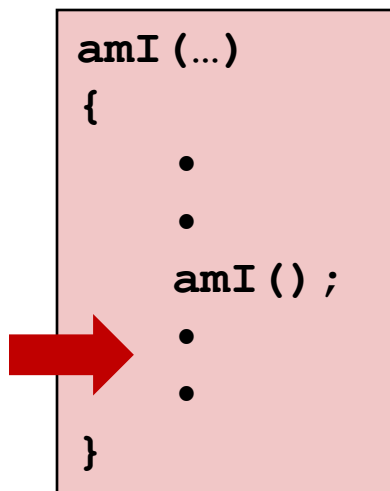
Example



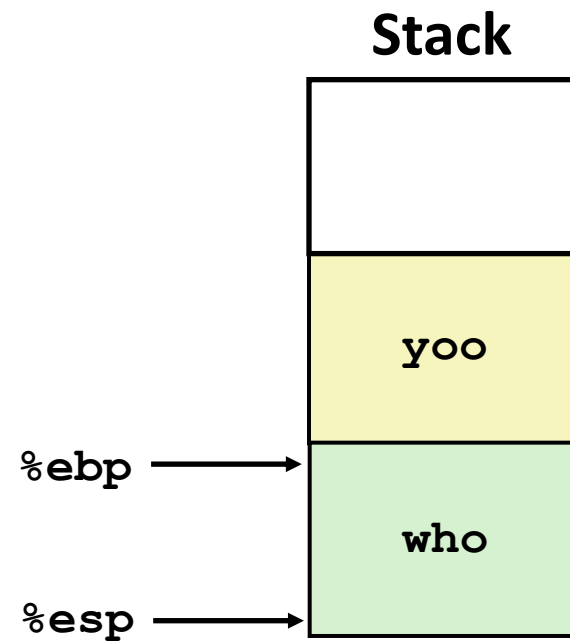
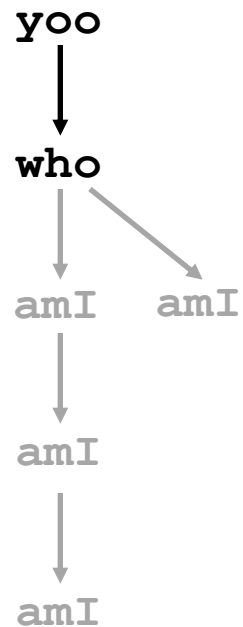
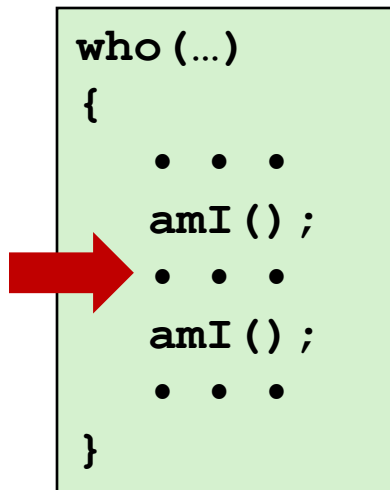
Example



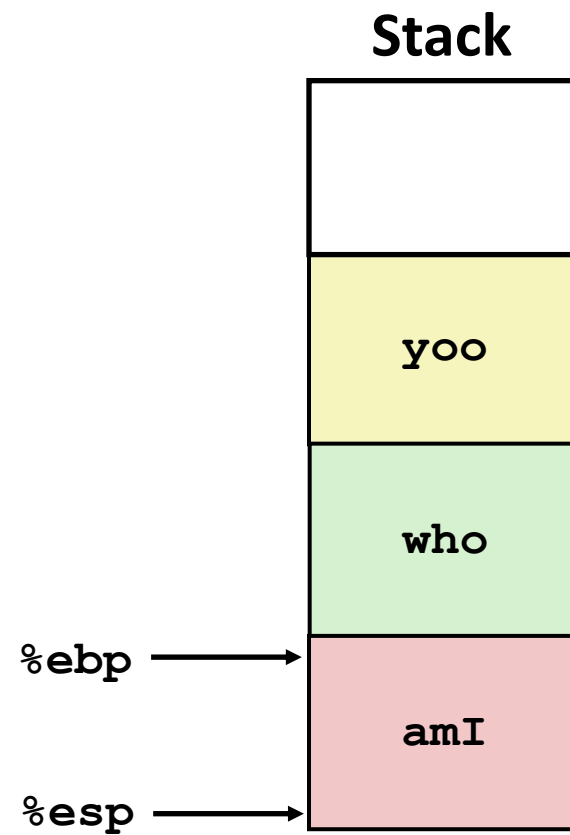
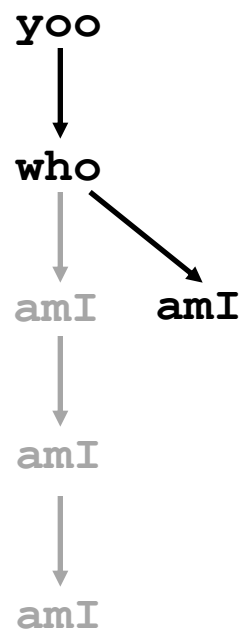
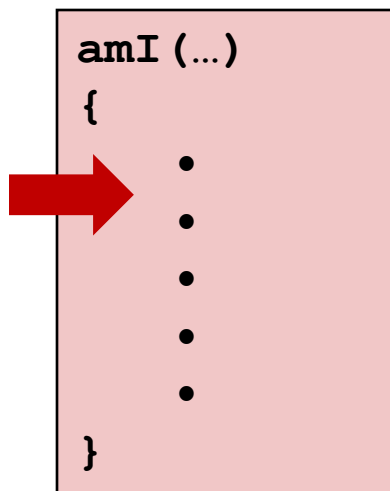
Example



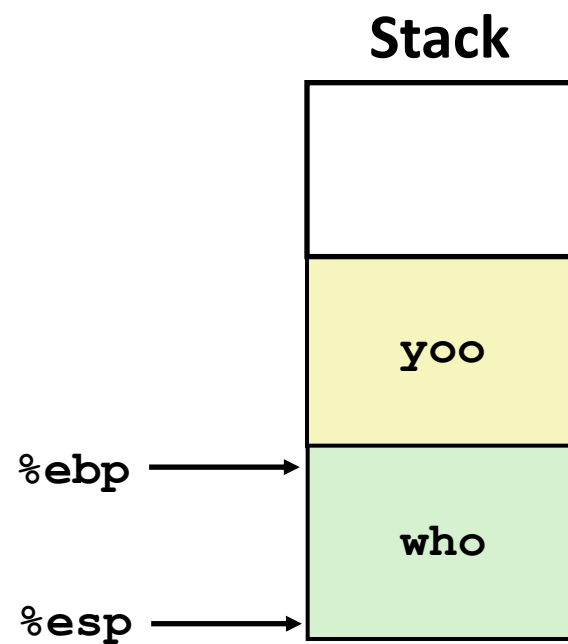
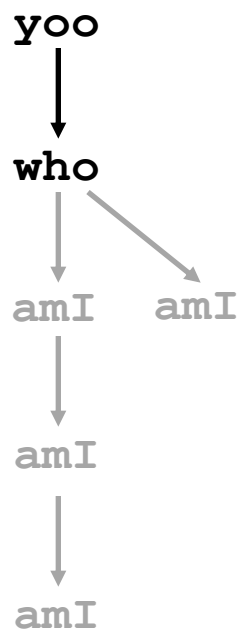
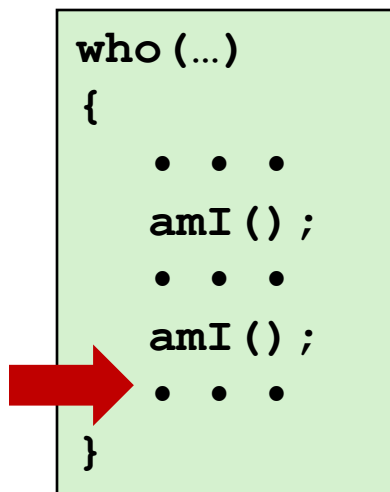
Example



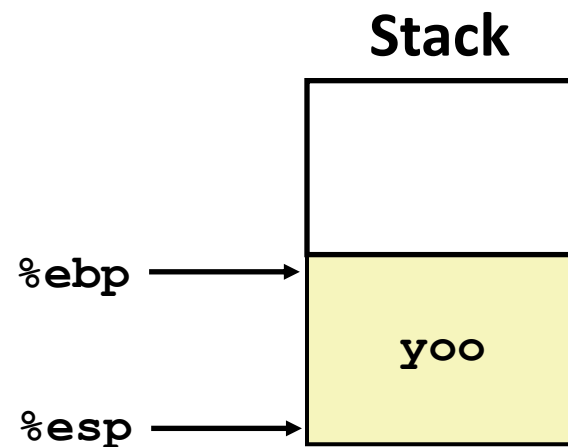
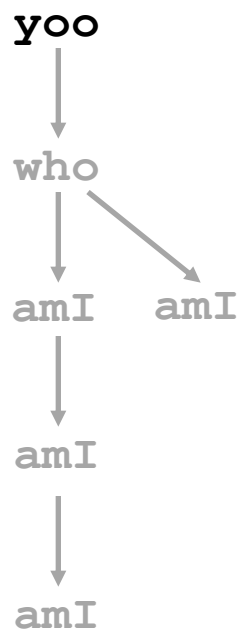
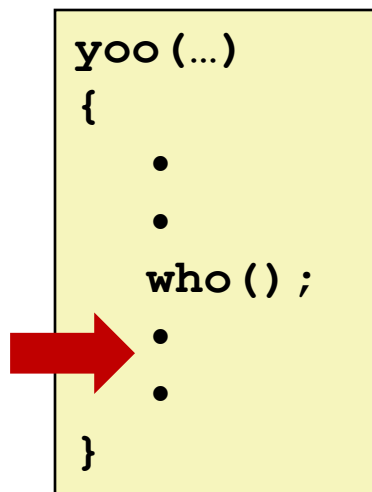
Example



Example



Example



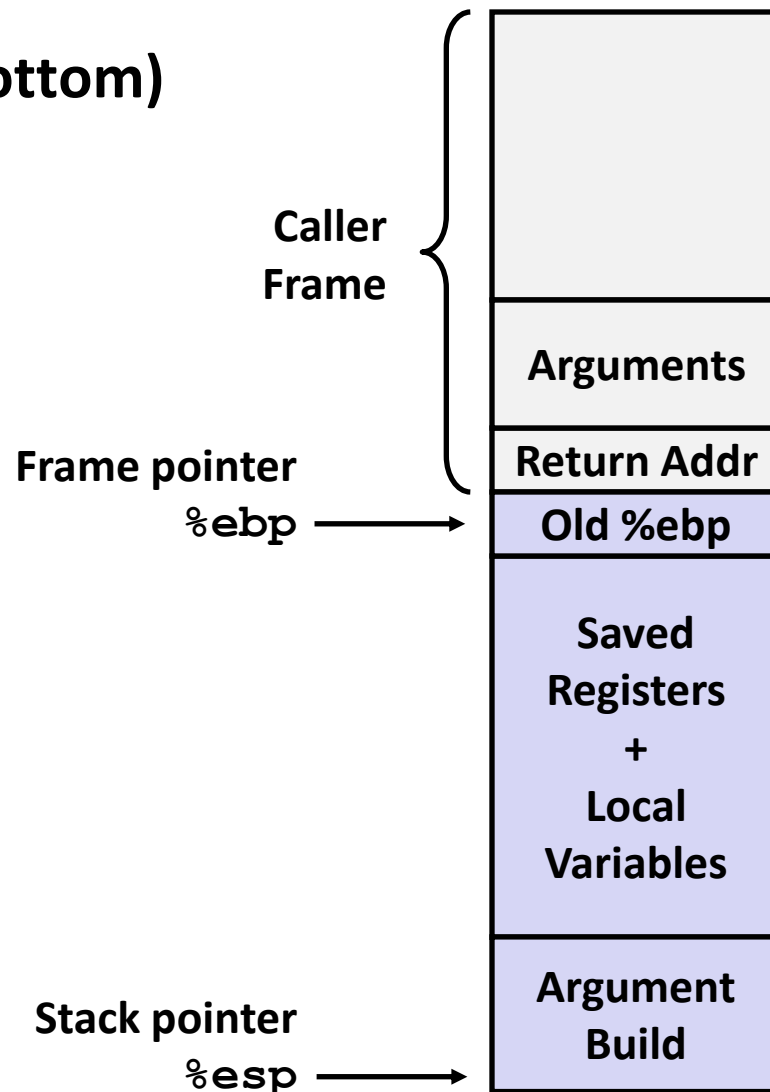
IA32/Linux Stack Frame

■ 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

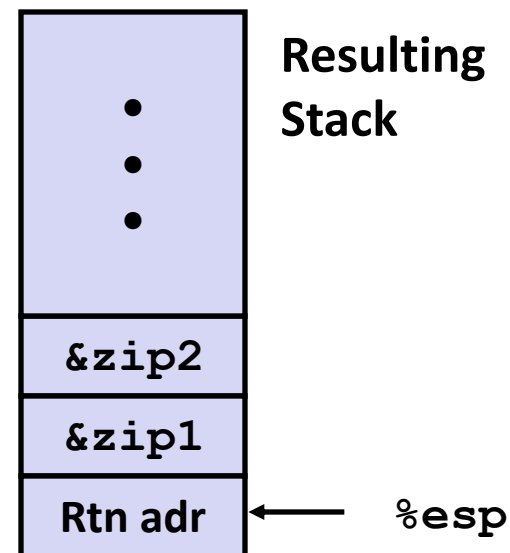
```
int zip1 = 15213;
int zip2 = 91125;

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
    . . .
```



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
} Set Up

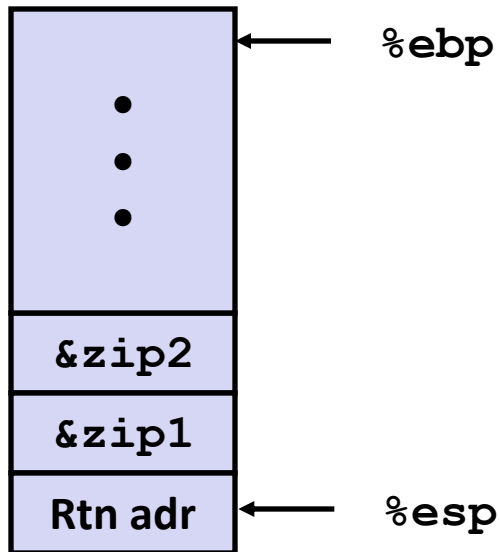
    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)
} Body

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

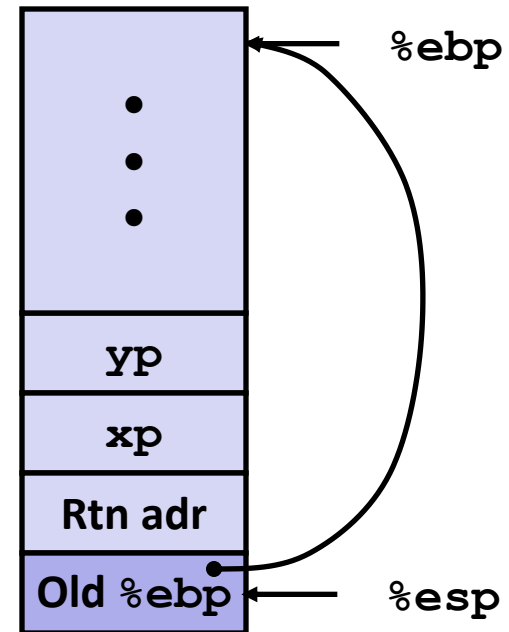
Do on blackboard?

swap Setup #1

Entering Stack



Resulting Stack

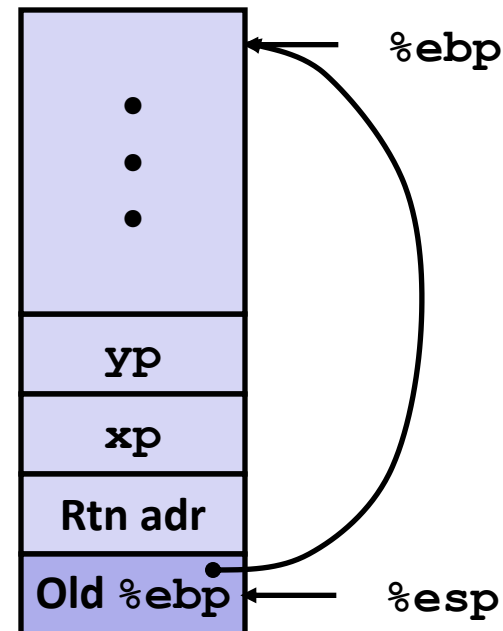
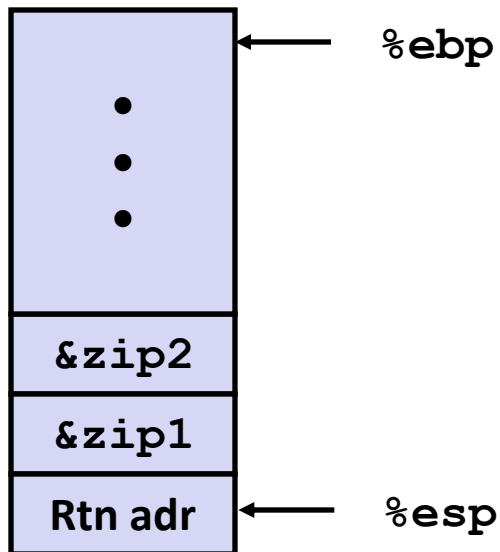


`swap:`

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

swap Setup #1

Entering Stack

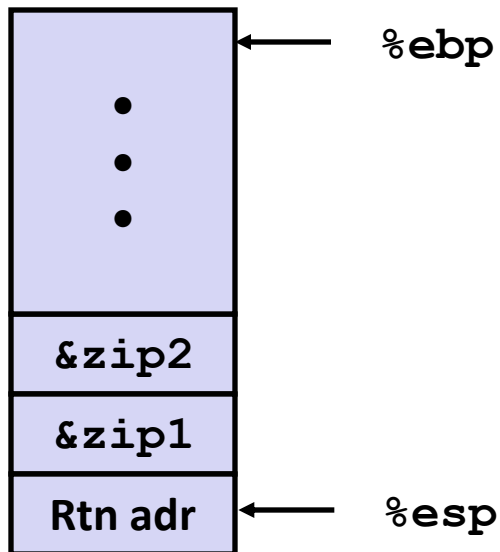


`swap:`

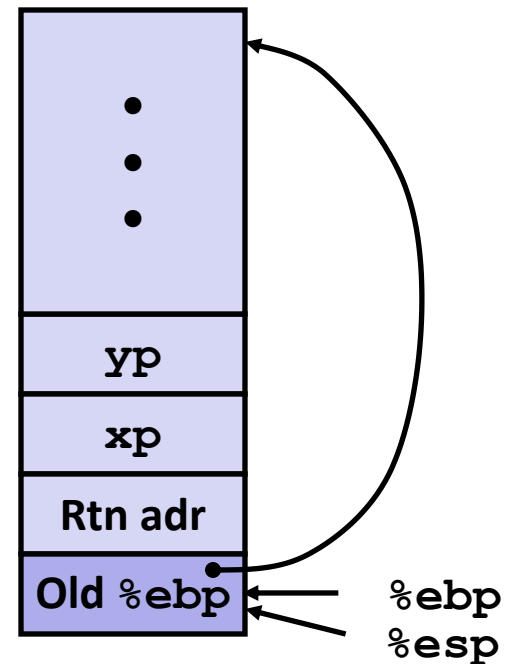
```
    pushl %ebp  
    movl %esp, %ebp  
    pushl %ebx
```

swap Setup #1

Entering Stack



Resulting Stack



swap:

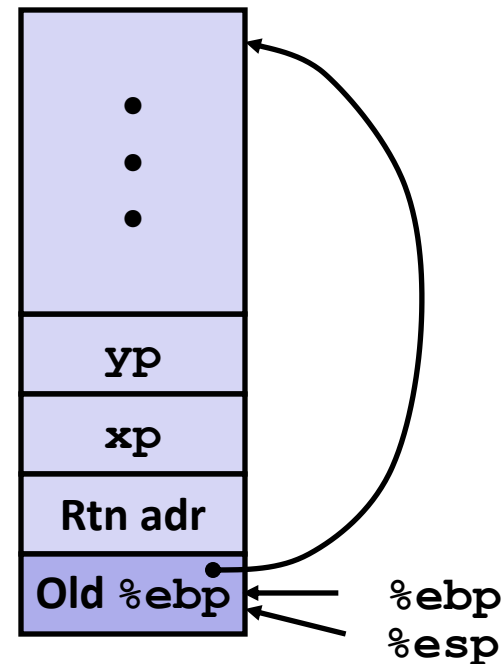
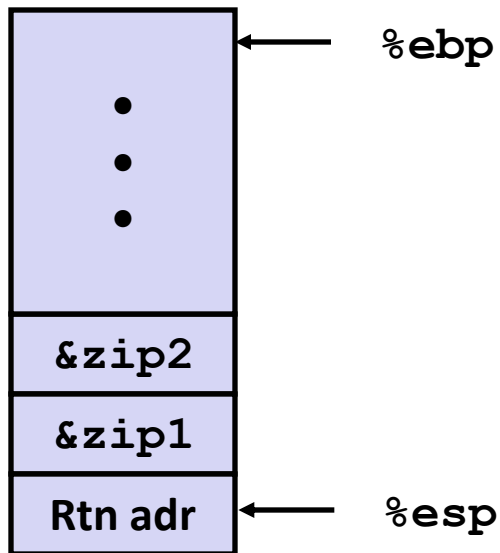
```

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

```

swap Setup #1

Entering Stack

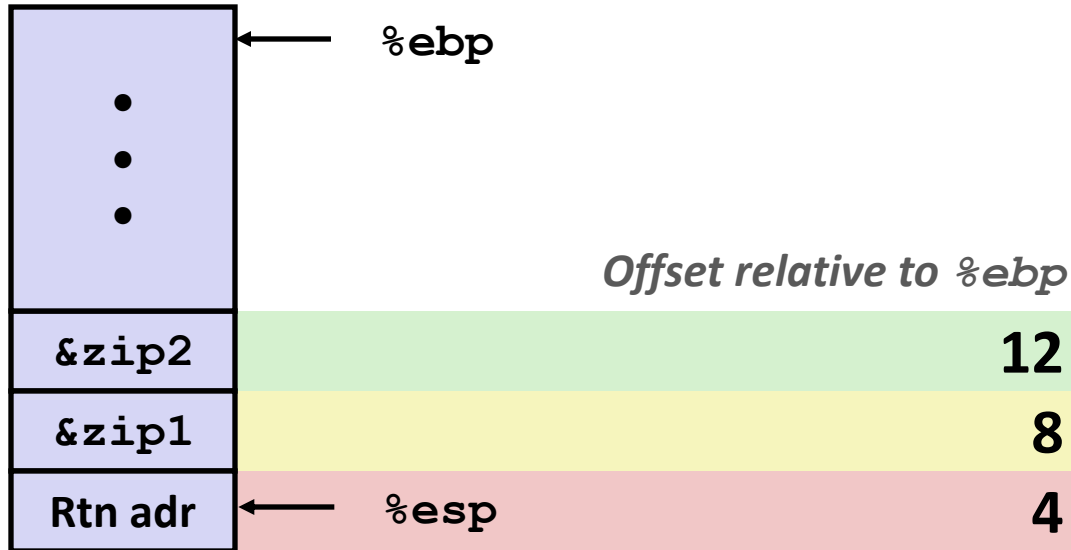


`swap:`

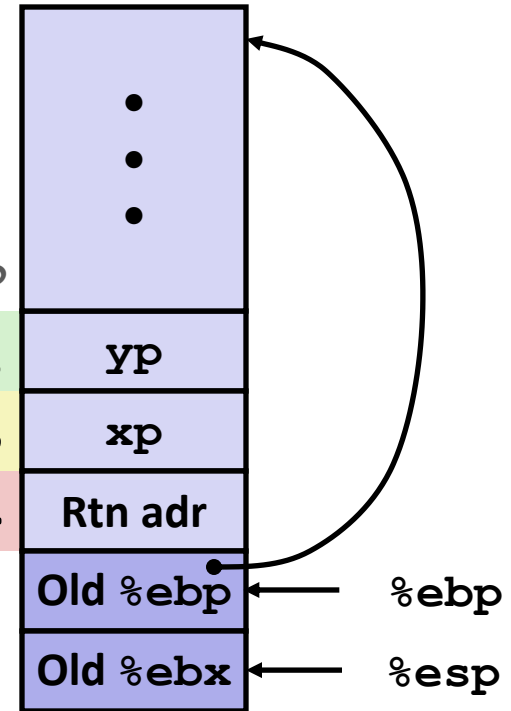
```
    pushl %ebp  
    movl %esp,%ebp  
    pushl %ebx
```


swap Setup #1

Entering Stack



Resulting Stack



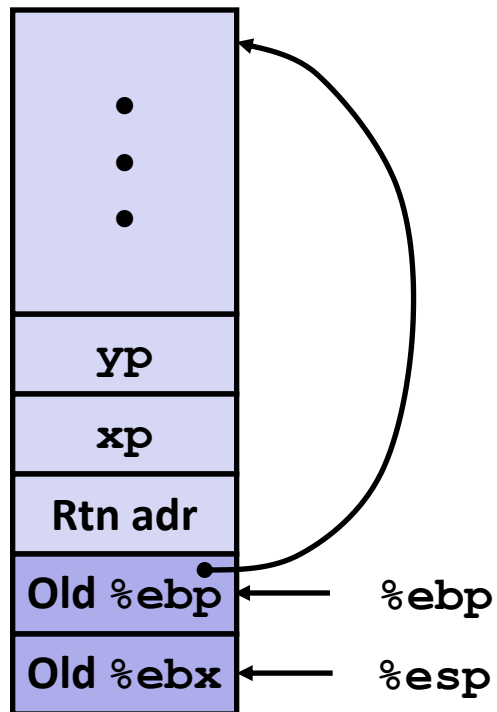
```

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

```

swap Finish #1

swap' s Stack

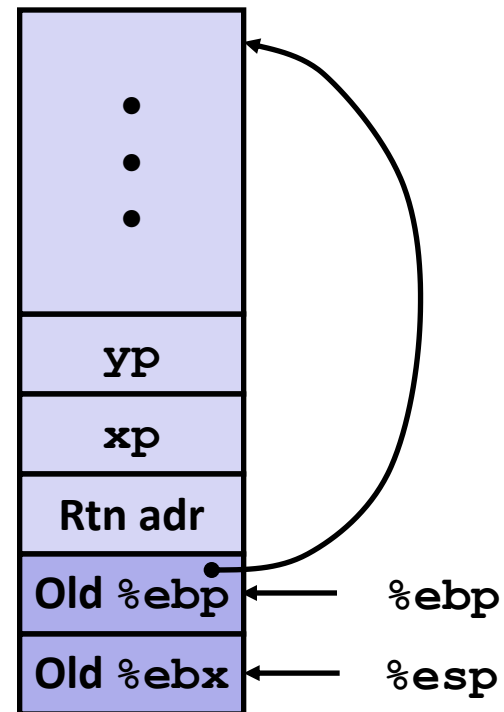


```

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

```

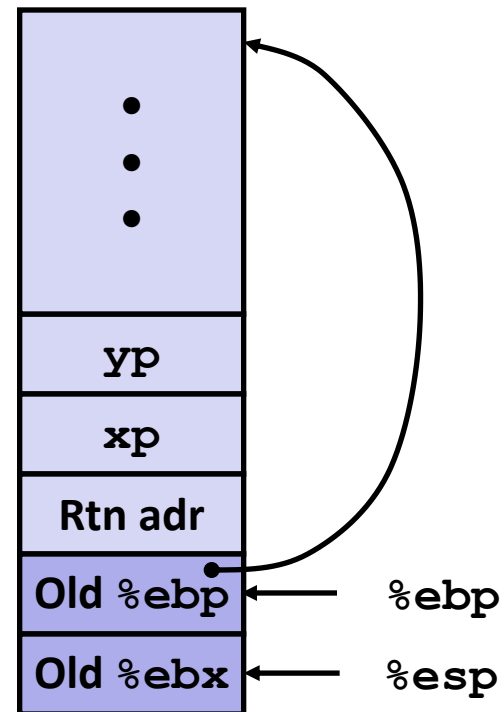
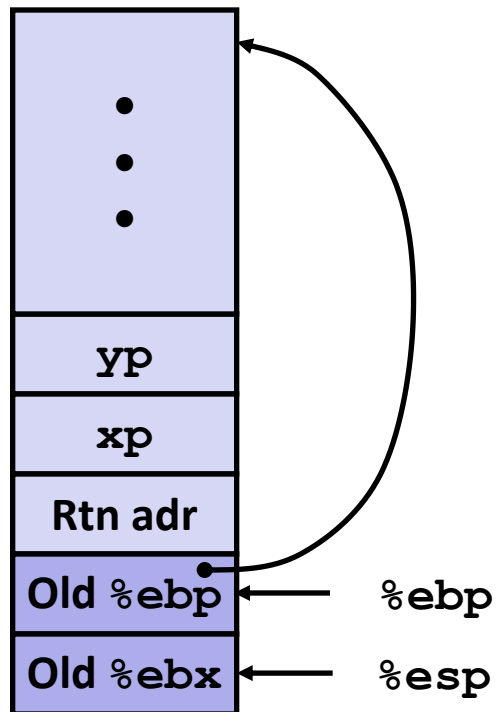
Resulting Stack



Observation: Saved and restored register %ebx

swap Finish #2

swap's Stack



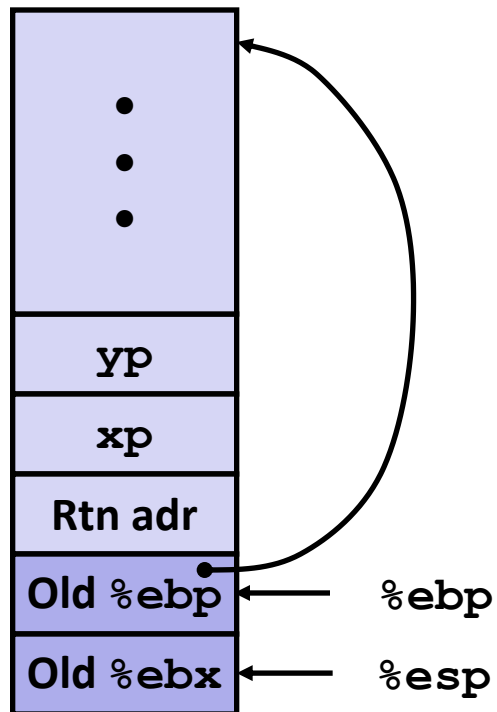
```

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

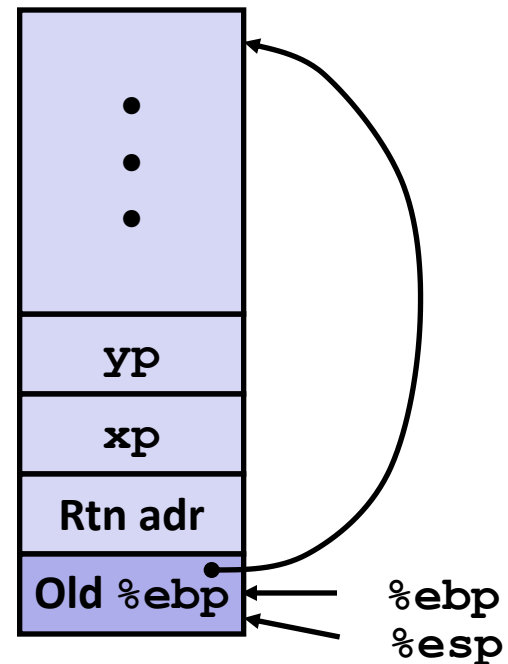
```

swap Finish #2

swap' s Stack



Resulting Stack



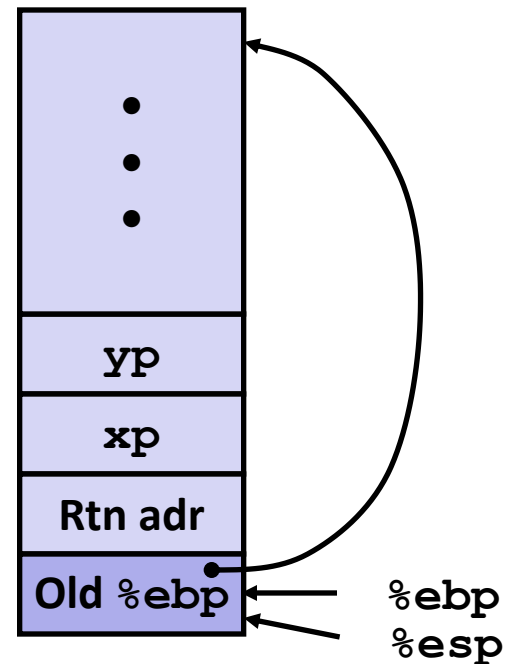
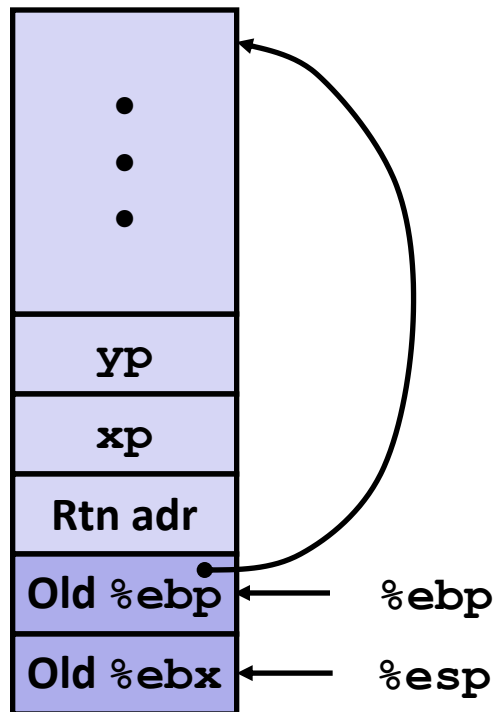
```

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

```

swap Finish #2

swap's Stack



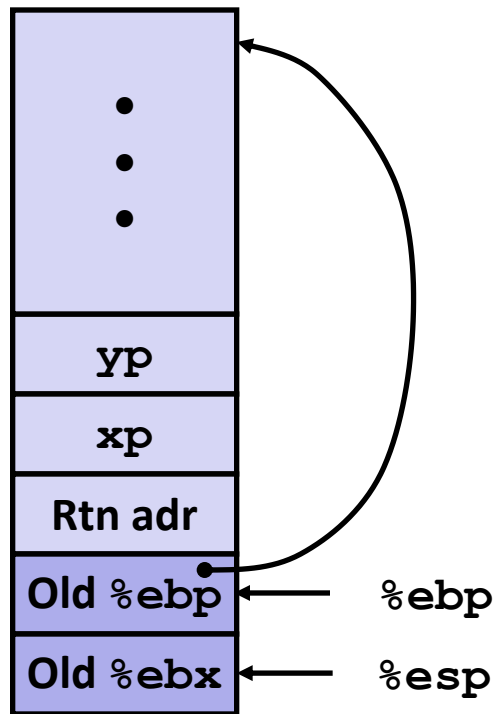
```

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

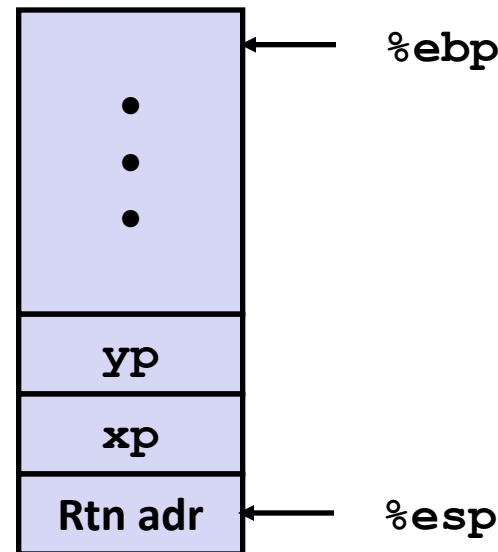
```

swap Finish #3

swap' s Stack



Resulting Stack



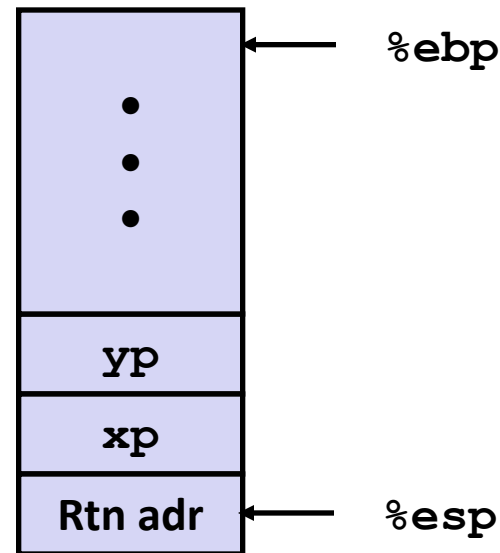
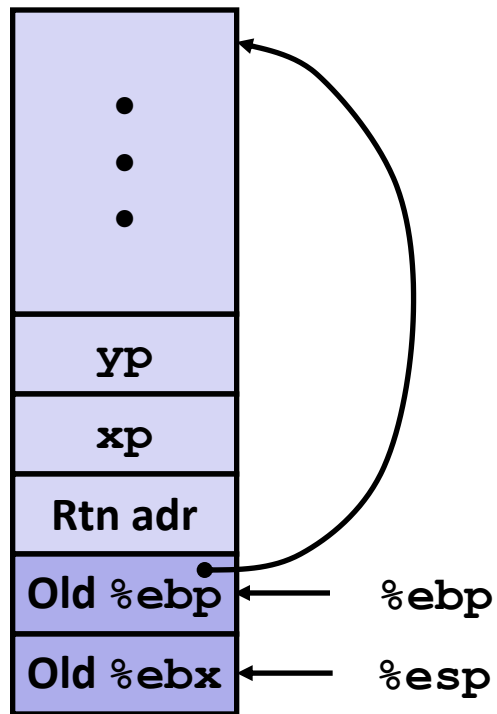
```

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

```

swap Finish #4

swap's Stack



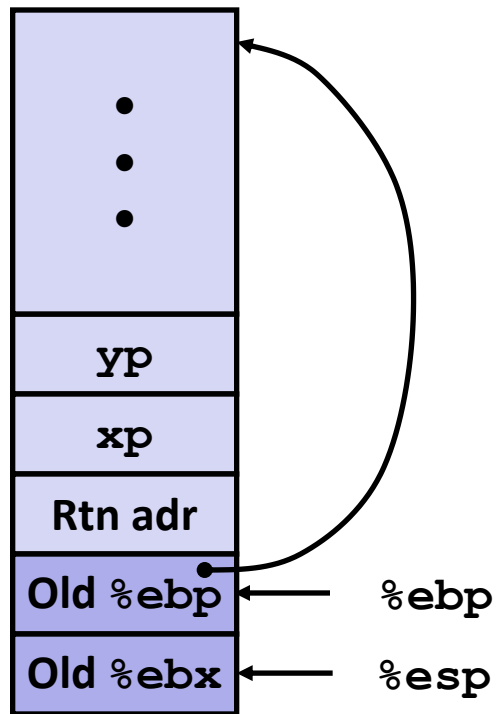
```

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

```

swap Finish #4

swap's Stack

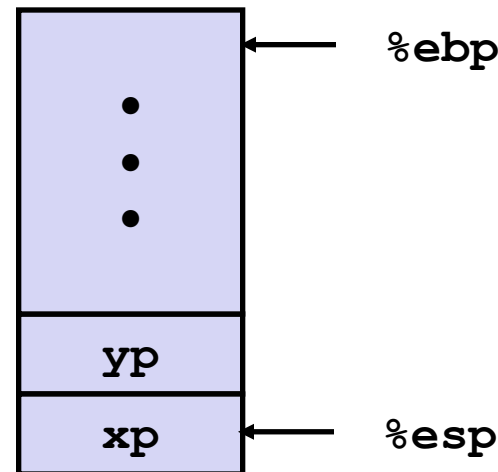


```

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

```

Resulting Stack



■ Observation

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

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
```

Calling Code

```
8048409:  e8 96 ff ff ff  call 80483a4 <swap>
804840e:  8b 45 f8       mov  0xffffffff8(%ebp), %eax
```

Register Saving Conventions

■ When procedure `yoo` calls `who`:

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

■ Can Register be used for temporary storage?

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

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

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

Register Saving Conventions

- When procedure `yoo` calls `who`:
 - `yoo` is the *caller*
 - `who` is the *callee*
- Can register be used for temporary storage?
- Conventions
 - “*Caller Save*”
 - Caller saves temporary in its frame before calling
 - “*Callee Save*”
 - Callee saves temporary in its frame before using

IA32/Linux Register Usage

■ `%eax`, `%edx`, `%ecx`

- Caller saves prior to call if values are used later

■ `%eax`

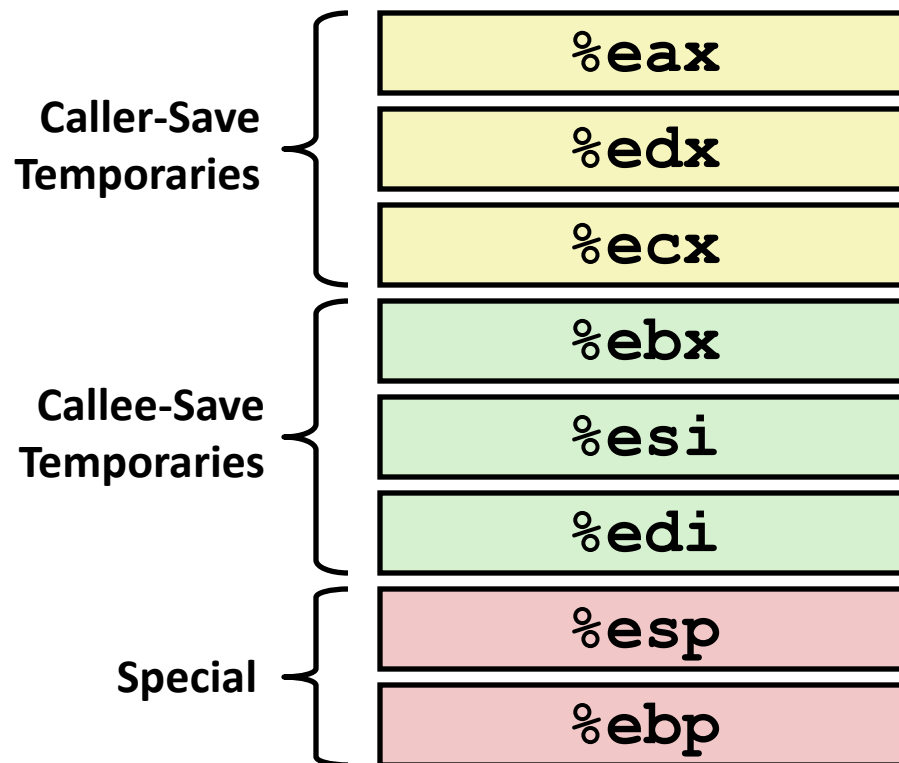
- also used to return integer value

■ `%ebx`, `%esi`, `%edi`

- Callee saves if wants to use them

■ `%esp`, `%ebp`

- special



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 saved 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
```

Pointer Code

Recursive Procedure

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

Top-Level Call

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

- Pass pointer to update location

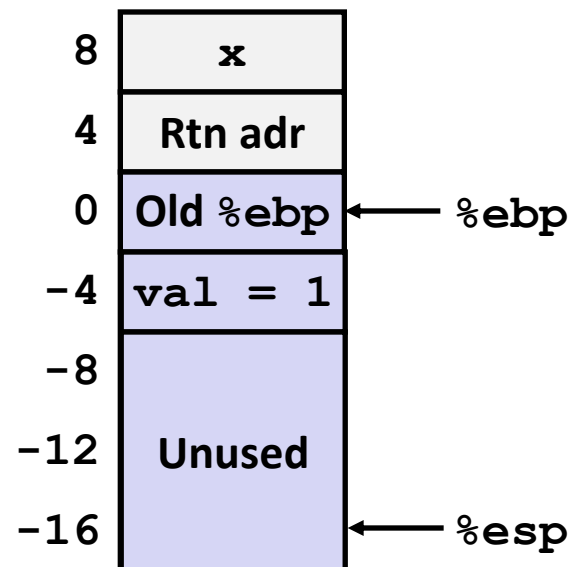
Creating & Initializing Pointer

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

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

Initial part of `sfact`

```
_sfact:
    pushl %ebp
    movl %esp,%ebp
    subl $16,%esp
    movl 8(%ebp),%edx
    movl $1,-4(%ebp)
```



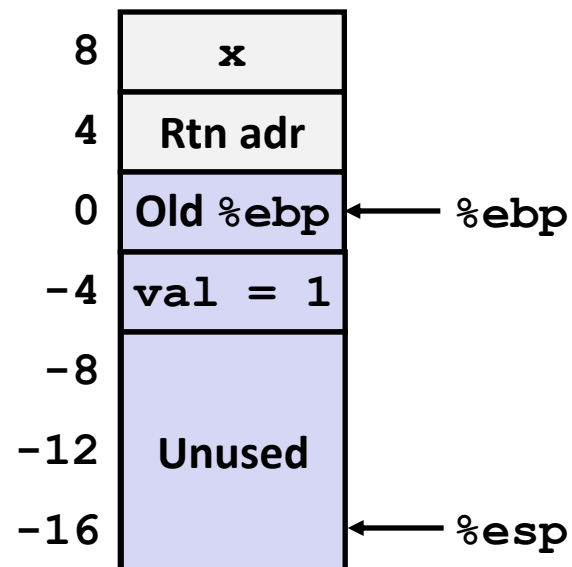
Creating & Initializing Pointer

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

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

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
```



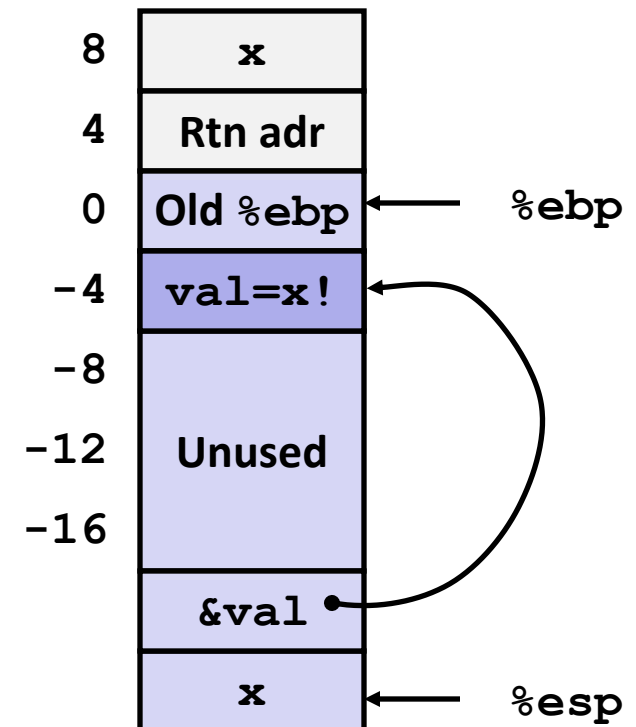
Passing Pointer

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

Calling s_helper from sfact

```
leal -4(%ebp), %eax
pushl %eax
pushl %edx
call s_helper
movl -4(%ebp), %eax
. . .
```

Stack at time of call



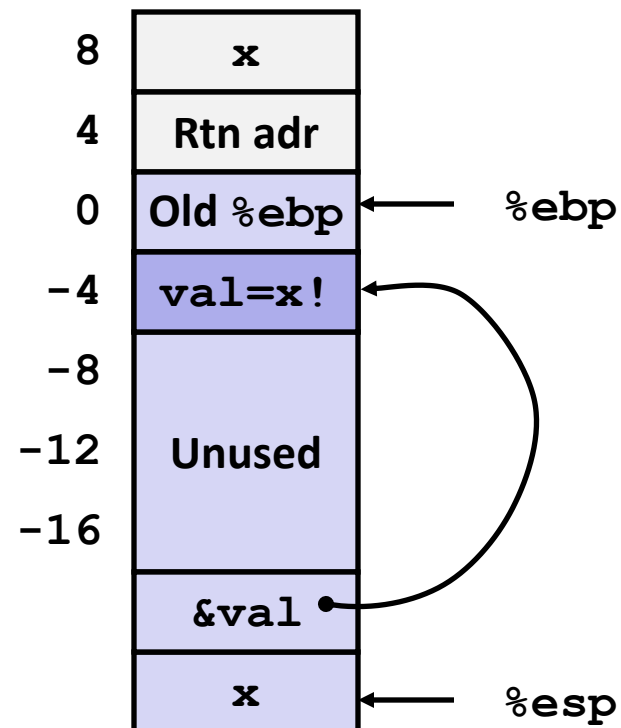
Passing Pointer

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

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
```

Stack at time of call



IA 32 Procedure 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
- 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

