

---

# Machine-Level Representation

CSCI 2021: Machine Architecture and Organization

Antonia Zhai

Department Computer Science and Engineering

University of Minnesota

<http://www.cs.umn.edu/~zhai>

With Slides from Bryant and O'Hallaron



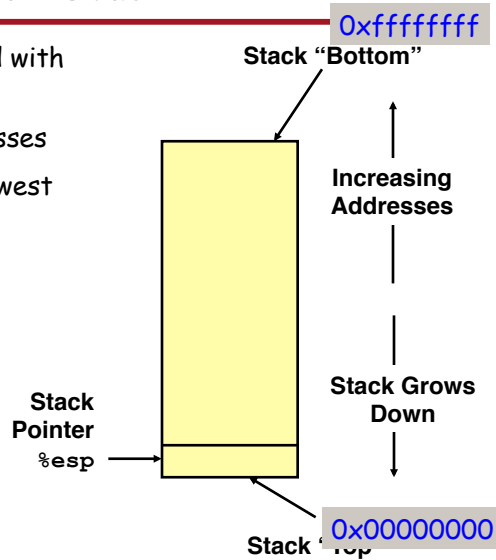
---

## Procedure Calls

With Slides from Bryant and O'Hallaron

## IA-32 Stack

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

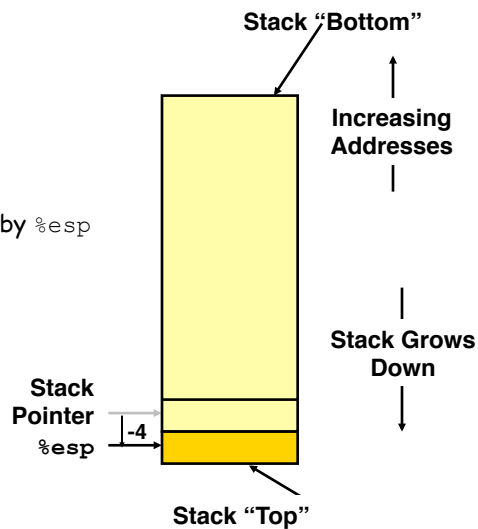


With Slides from Bryant and O'Hallaron

## IA-32 Stack Pushing

### Pushing

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

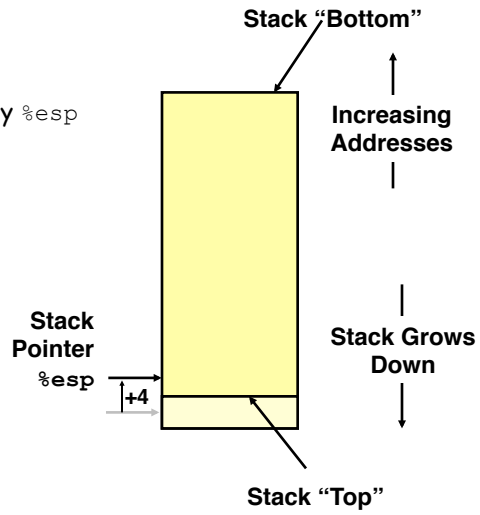


With Slides from Bryant and O'Hallaron

## IA-32 Stack Popping

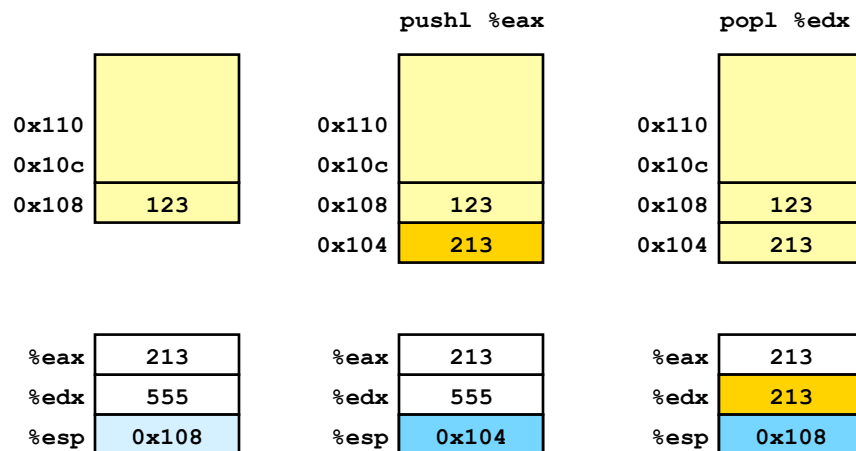
### Popping

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



With Slides from Bryant and O'Hallaron

## Stack Operation Examples



With Slides from Bryant and O'Hallaron

## 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 value

- 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:

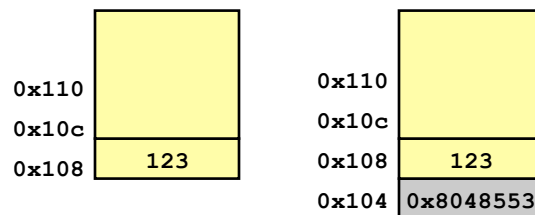
```
ret          Pop address from stack; Jump to address
```

With Slides from Bryant and O'Hallaron

## Procedure Call Example

---

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



%esp 0x108

%esp 0x104

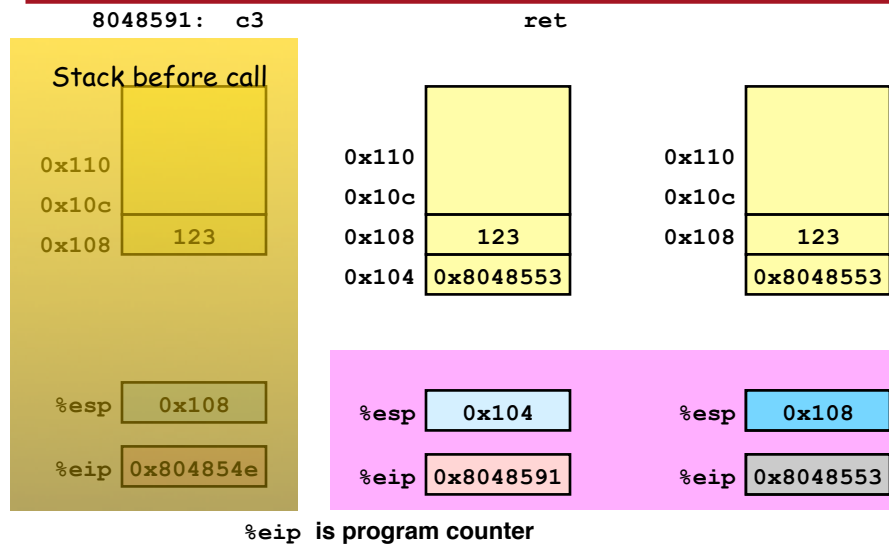
%eip 0x804854e

%eip 0x8048b90

%eip is program counter

With Slides from Bryant and O'Hallaron

## Procedure Return Example



With Slides from Bryant and O'Hallaron

## Call Chain Example

Code Structure

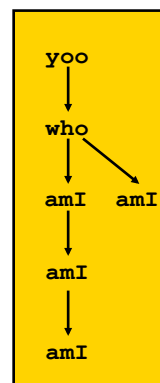
```
yoo (...)  
{  
  .  
  .  
  who ();  
  .  
  .  
}
```

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

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

Procedure amI is recursive

Call Chain



2/10/15

CSCI 2021

10

With Slides from Bryant and O'Hallaron

# Stack Frames

## Contents

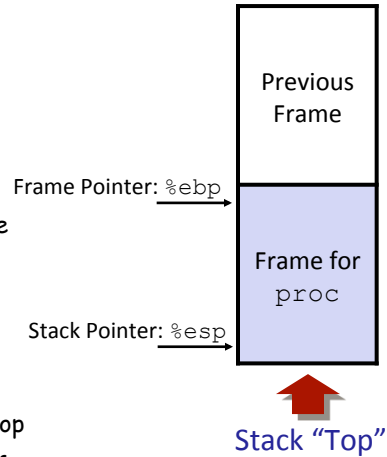
- Local variables
- Return information
- Temporary space

## Management

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

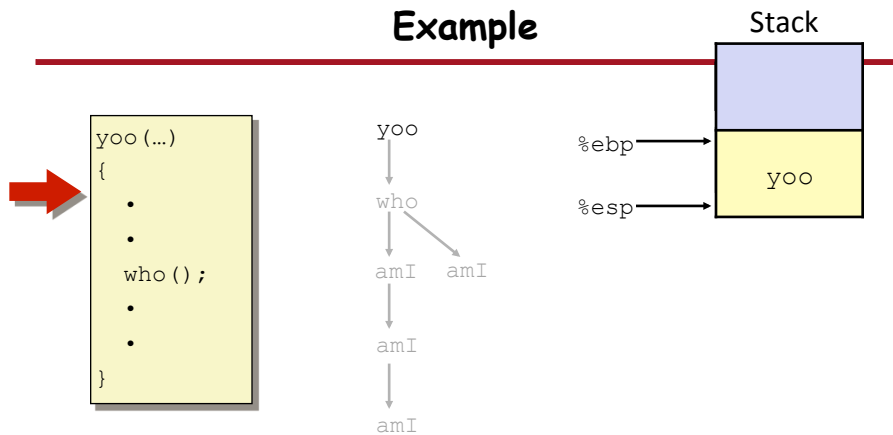
## Pointers

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

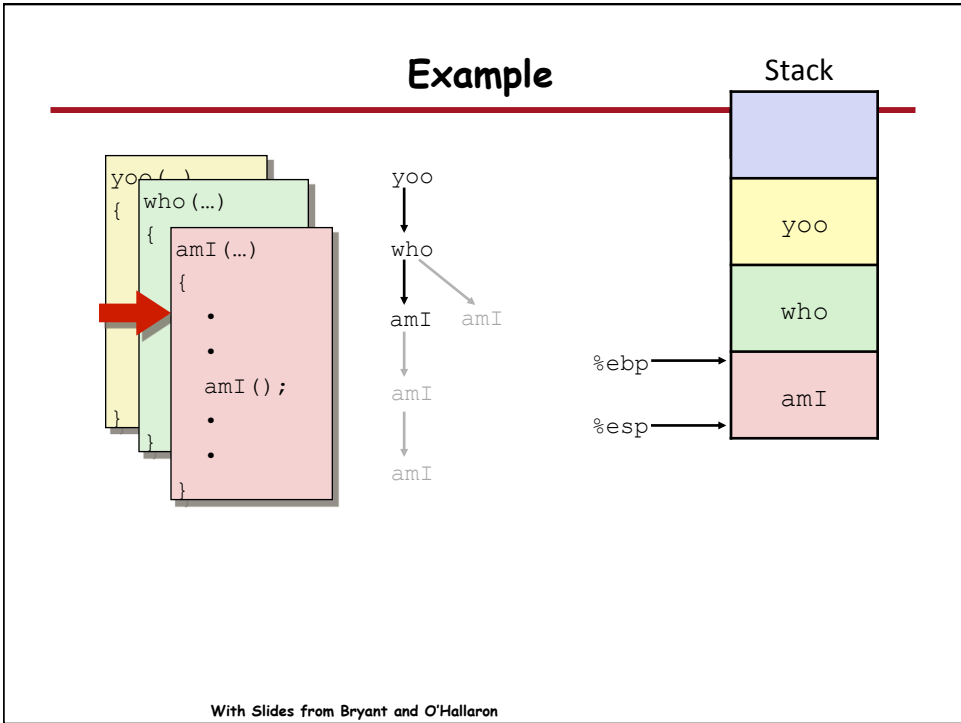
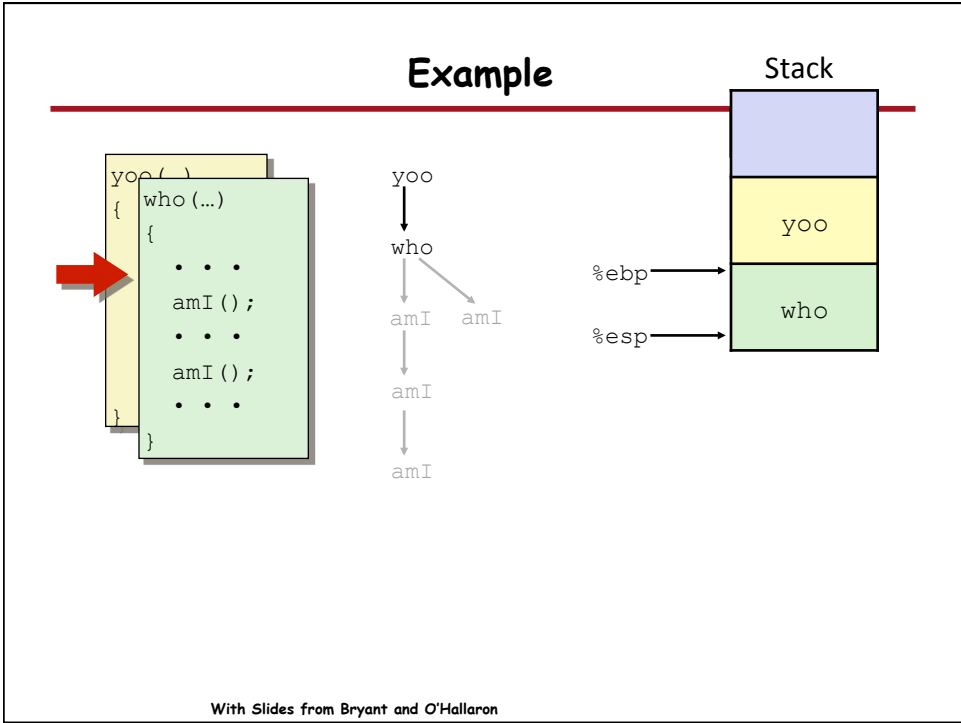


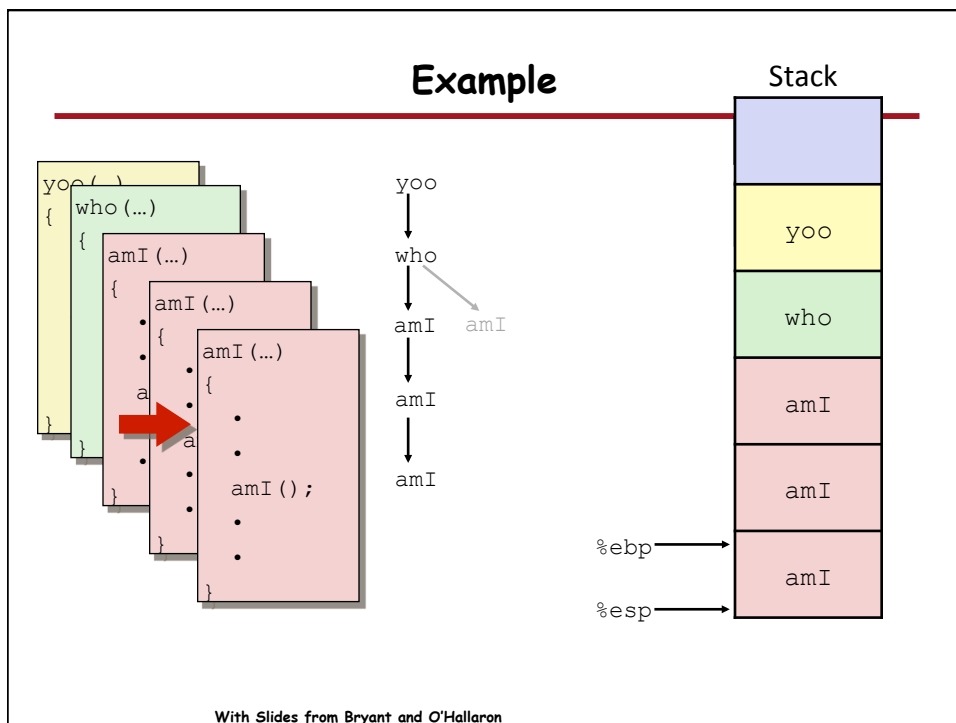
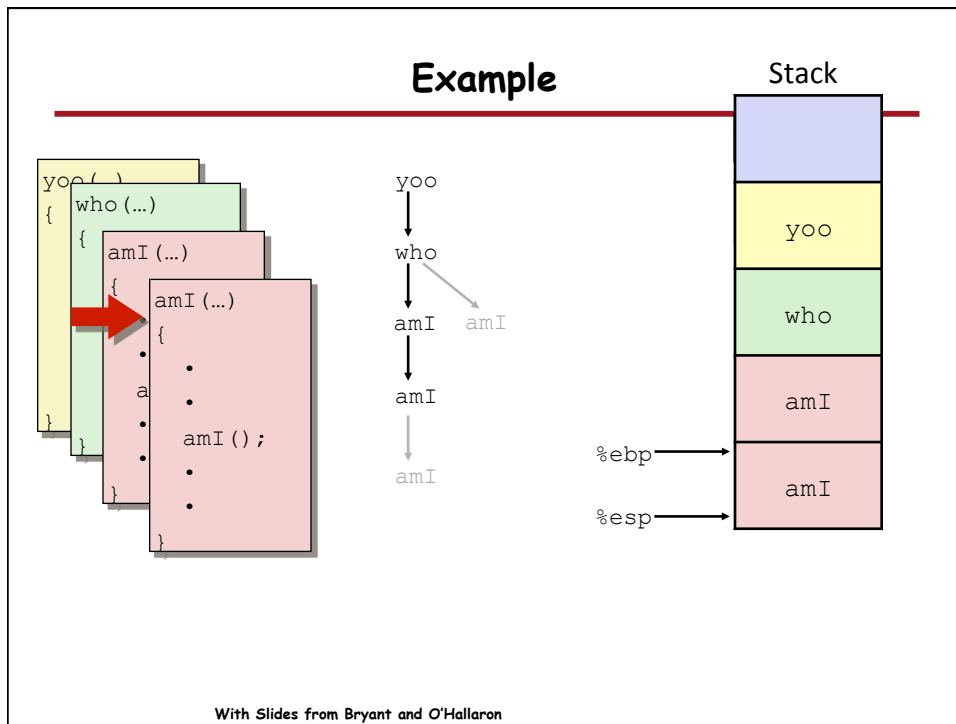
With Slides from Bryant and O'Hallaron

# Example

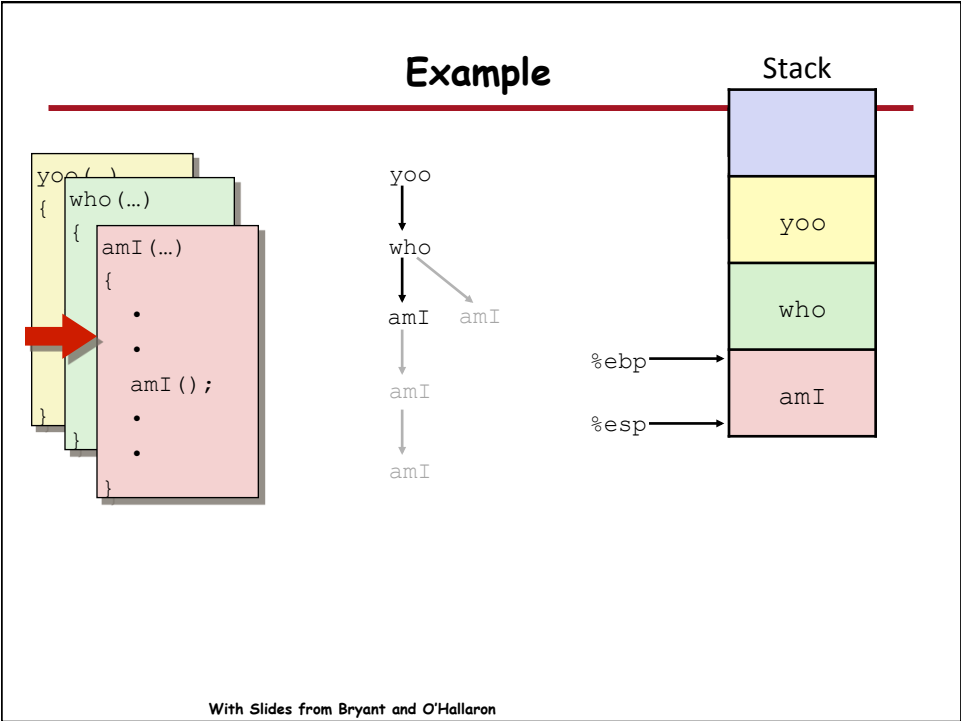
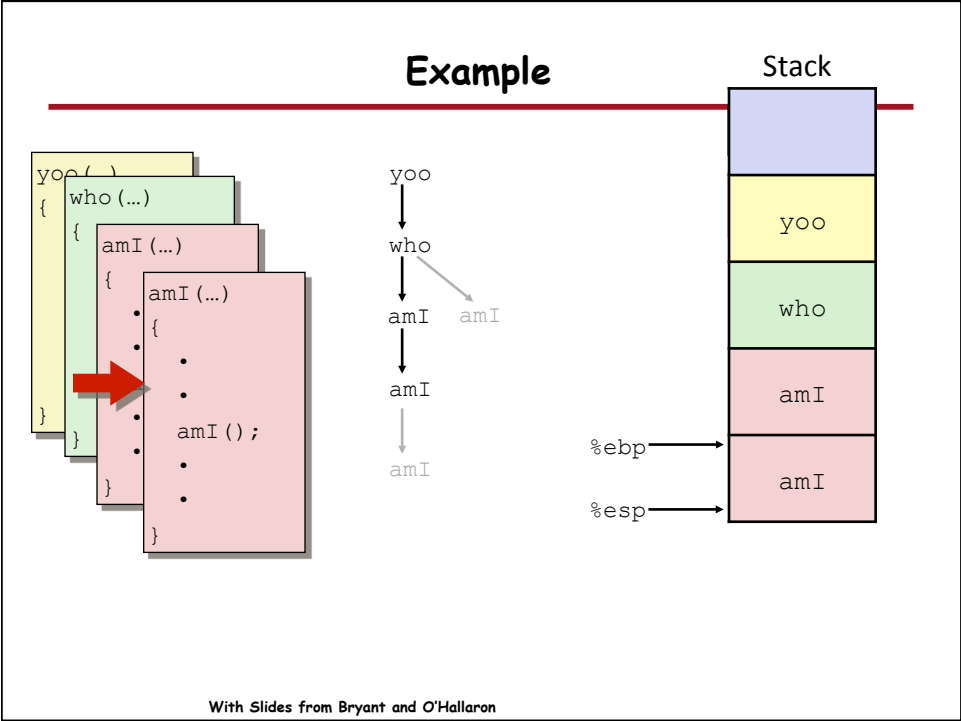


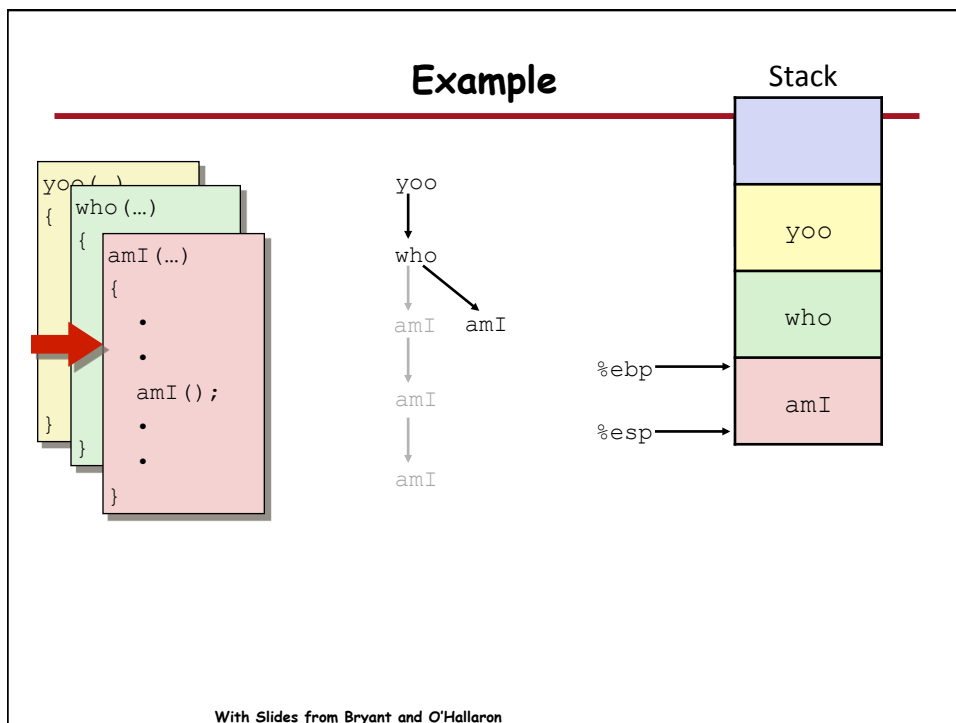
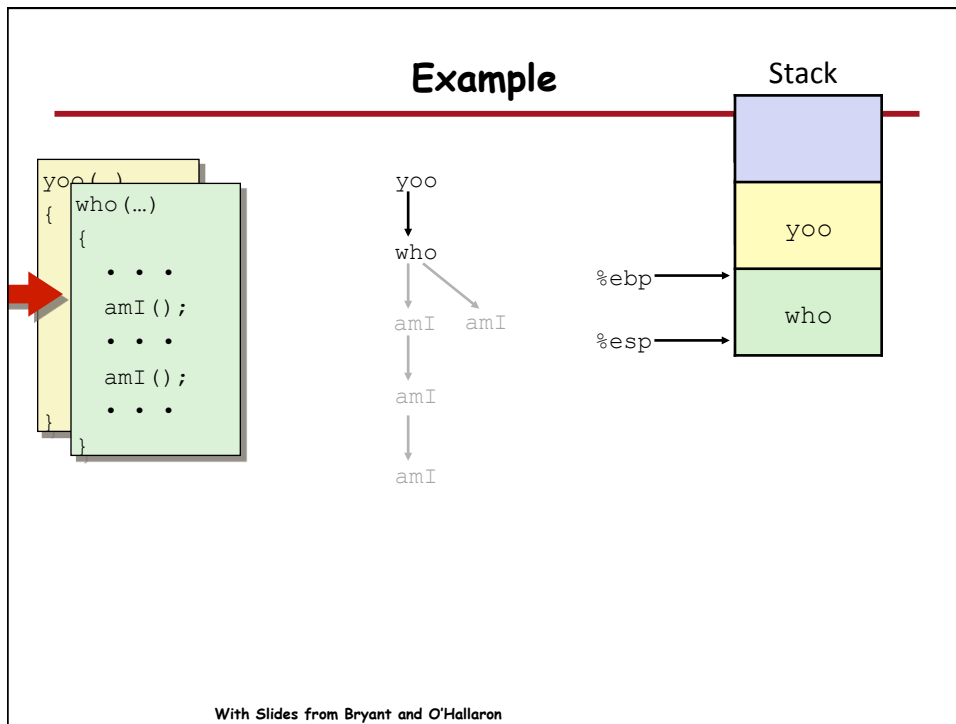
With Slides from Bryant and O'Hallaron

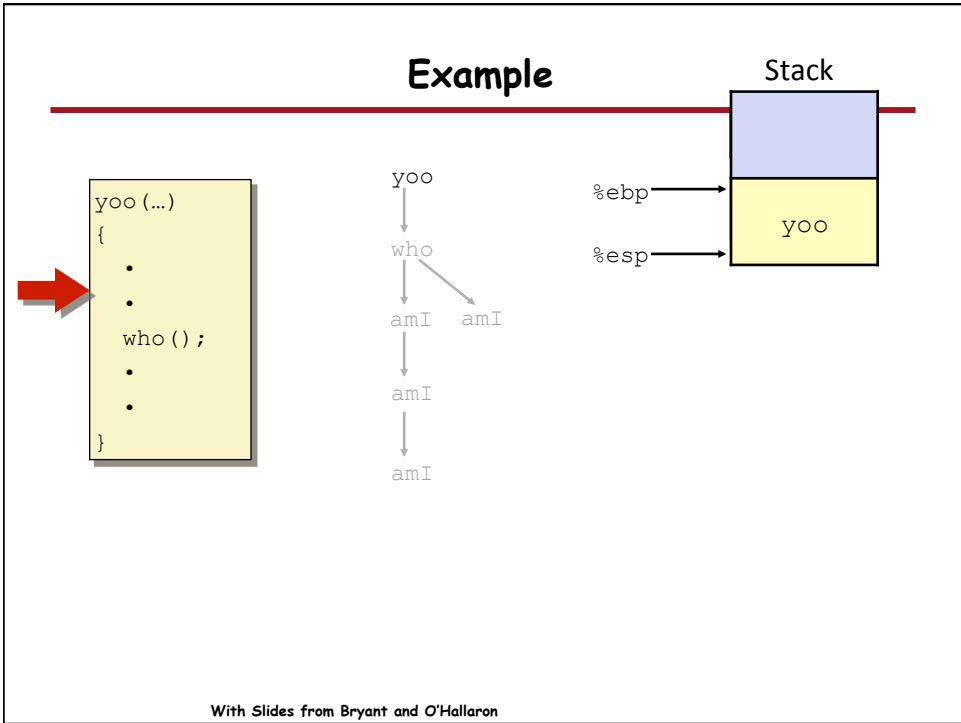
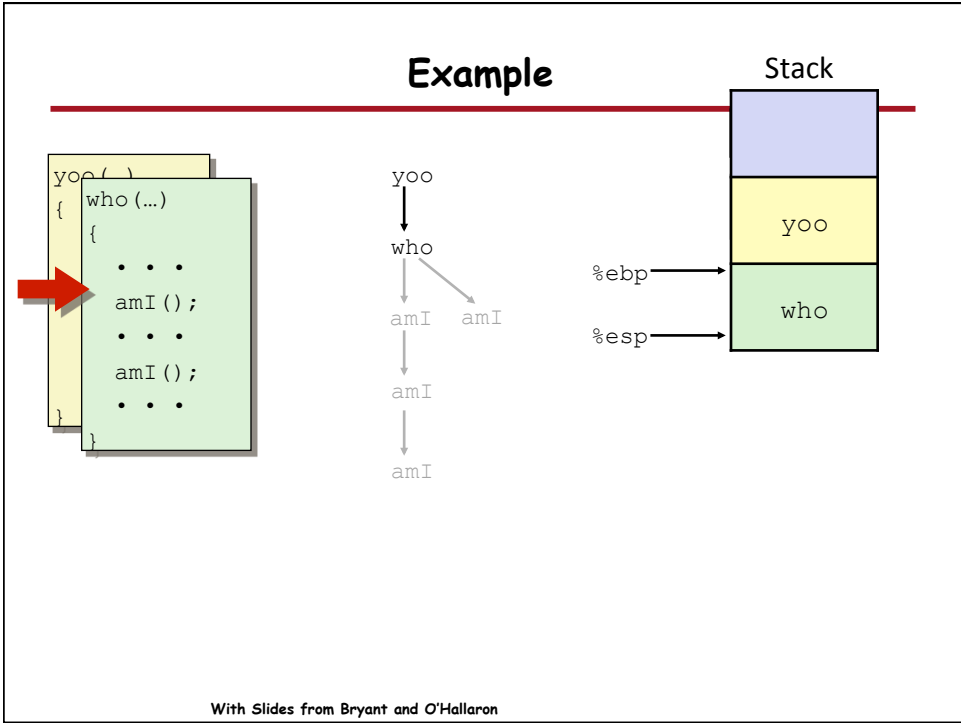












## Stack-Based Languages

- 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

With Slides from Bryant and O'Hallaron

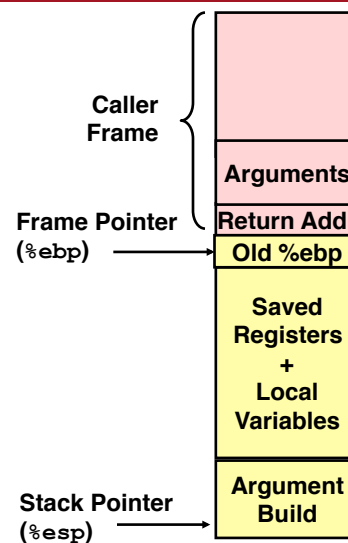
## IA32/Linux Stack Frame

Current Stack Frame ("Top" to Bottom)

- Parameters for function about to call
  - "Argument build"
- Local variables
  - If can't keep in registers
- Saved register context
- Old frame pointer

Caller Stack Frame

- Return address
  - Pushed by `call` instruction
- Arguments for this call



With Slides from Bryant and O'Hallaron

## Revisiting swap

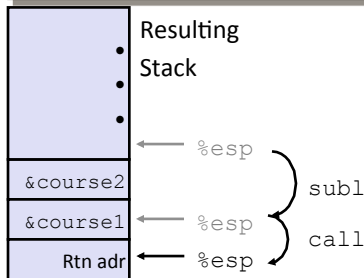
```
int course1 = 15213;
int course2 = 18243;

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
    . . .
```



With Slides from Bryant and O'Hallaron

## 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

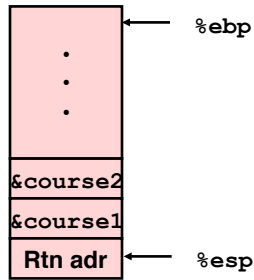
    movl 12(%ebp), %ecx
    movl 8(%ebp), %edx
    movl (%ecx), %eax
    movl (%edx), %ebx
    movl %eax, (%edx)
    movl %ebx, (%ecx)
    } Body

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

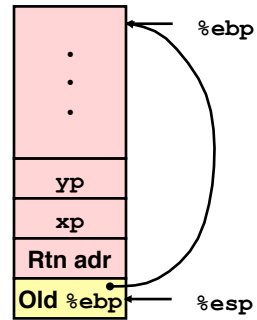
With Slides from Bryant and O'Hallaron

## swap Setup #1

Entering Stack



Resulting Stack

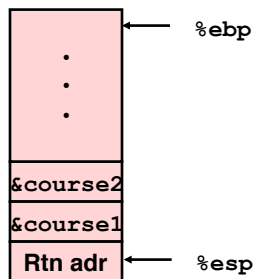


```
swap:  
    pushl %ebp  
    movl %esp,%ebp  
    pushl %ebx
```

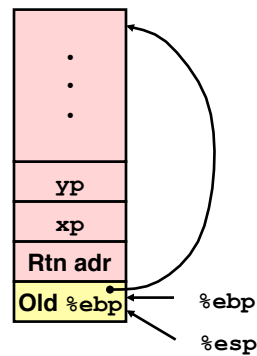
With Slides from Bryant and O'Hallaron

## swap Setup #2

Entering Stack



Resulting Stack

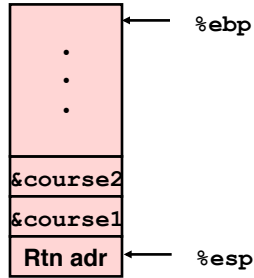


```
swap:  
    pushl %ebp  
    movl %esp,%ebp  
    pushl %ebx
```

With Slides from Bryant and O'Hallaron

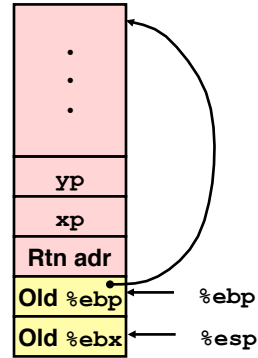
## swap Setup #3

Entering Stack



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

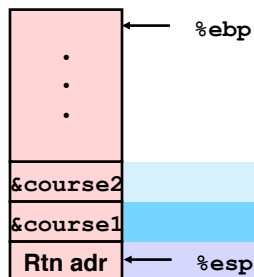
Resulting Stack



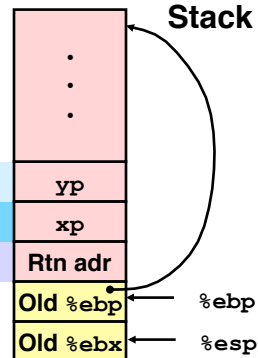
With Slides from Bryant and O'Hallaron

## Effect of swap Setup

Entering Stack



Offset  
(relative to %ebp)

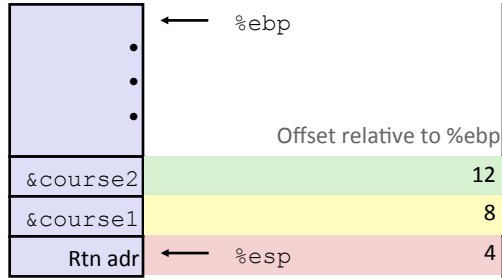


```
movl 12(%ebp),%ecx # get yp
movl 8(%ebp),%edx # get xp
. . . } Body
```

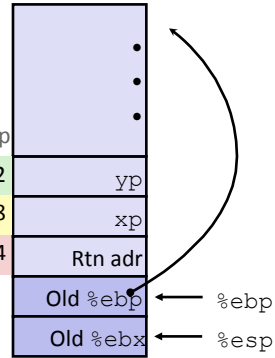
With Slides from Bryant and O'Hallaron

## swap Body

Entering Stack



Resulting Stack



```

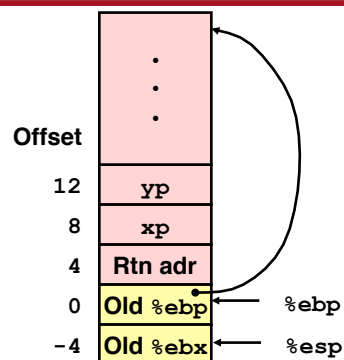
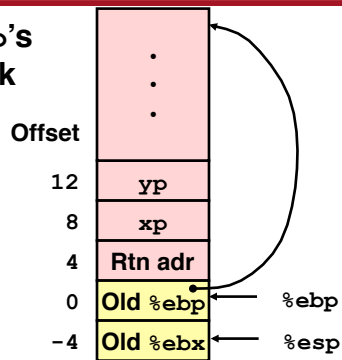
movl 8(%ebp),%edx # get xp
movl 12(%ebp),%ecx # get yp
. . .

```

With Slides from Bryant and O'Hallaron

## swap Finish #1

swap's Stack



- Observation
  - Saved & restored register %ebx

```

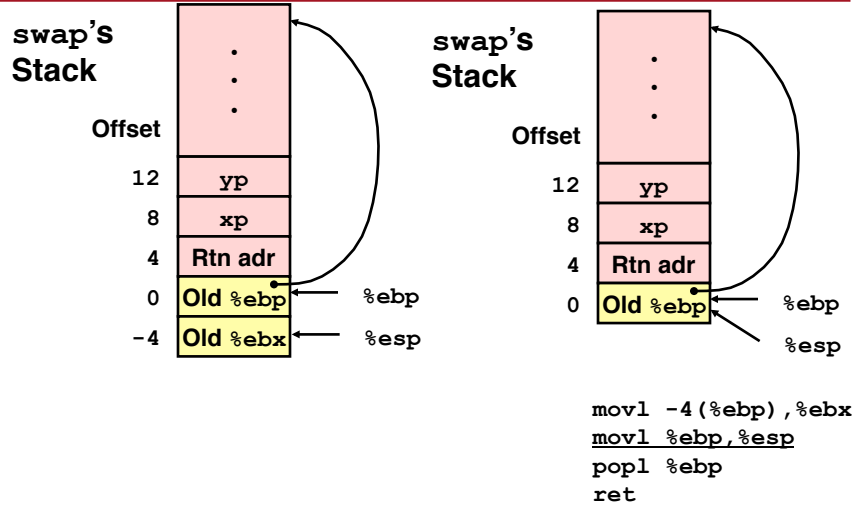
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret

```

With Slides from Bryant and O'Hallaron

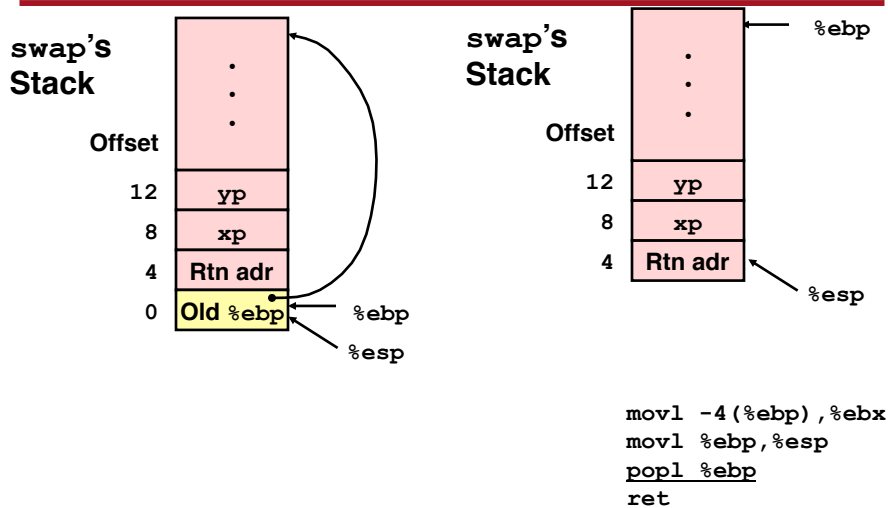


## swap Finish #2



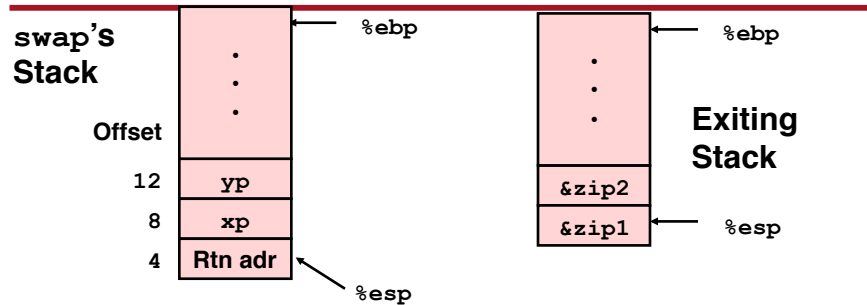
With Slides from Bryant and O'Hallaron

## swap Finish #3



With Slides from Bryant and O'Hallaron

## swap Finish #4



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

```

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

```

With Slides from Bryant and O'Hallaron

## Disassembled swap

```

08048384 <swap>:
8048384: 55          push  %ebp
8048385: 89 e5      mov   %esp,%ebp
8048387: 53        push  %ebx
8048388: 8b 55 08   mov   0x8(%ebp),%edx
804838b: 8b 4d 0c   mov   0xc(%ebp),%ecx
804838e: 8b 1a     mov   (%edx),%ebx
8048390: 8b 01     mov   (%ecx),%eax
8048392: 89 02     mov   %eax,(%edx)
8048394: 89 19     mov   %ebx,(%ecx)
8048396: 5b       pop   %ebx
8048397: 5d       pop   %ebp
8048398: c3       ret

```

### Calling Code

```

80483b4: movl  $0x8049658,0x4(%esp)  # Copy &course2
80483bc: movl  $0x8049654,(%esp)    # Copy &course1
80483c3: call  8048384 <swap>       # Call swap
80483c8: leave                                # Prepare to return
80483c9: ret                                # Return

```

With Slides from Bryant and O'Hallaron

## 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`
- This could be trouble → something should be done!
  - Need some coordination

With Slides from Bryant and O'Hallaron

## 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

Do you have to follow conventions?

2/10/15

CSCI 2021

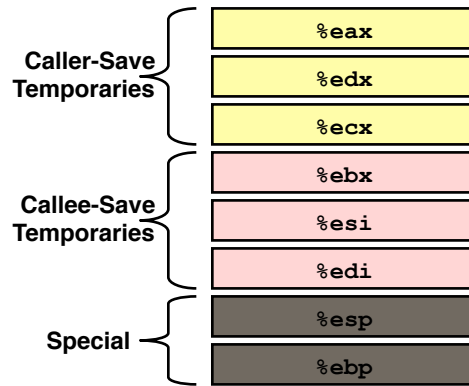
38

With Slides from Bryant and O'Hallaron

## IA32/Linux Register Usage

### Integer Registers

- Two have special uses  
%ebp, %esp
- Three managed as callee-save  
%ebx, %esi, %edi
  - Old values saved on stack prior to using
- Three managed as caller-save  
%eax, %edx, %ecx
  - Do what you please, but expect any callee to do so, as well
- Register %eax also stores returned value



2/10/15

CSCI 2021

39

With Slides from Bryant and O'Hallaron

## 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  $4, %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
    leal (%edx,%eax), %eax
.L3:
    addl  $4, %esp
    popl  %ebx
    popl  %ebp
    ret
```

With Slides from Bryant and O'Hallaron

## Recursive Call #1

```

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

```

### Actions

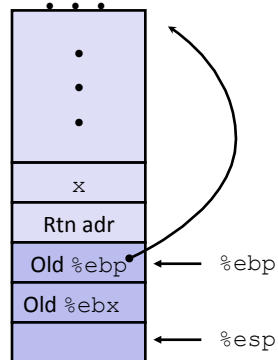
- Save old value of %ebx on stack
- Allocate space for argument to recursive call
- Store x in %ebx

%ebx x

```

pcount_r:
    pushl %ebp
    movl  %esp, %ebp
    pushl %ebx
    subl  $4, %esp
    movl  8(%ebp), %ebx

```



With Slides from Bryant and O'Hallaron

## Recursive Call #2

```

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

```

### Actions

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

%ebx x

```

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

```

With Slides from Bryant and O'Hallaron

## 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

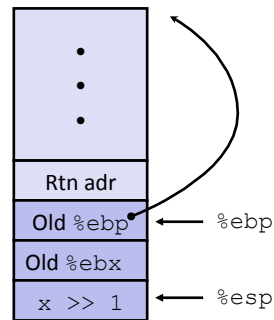
- `%eax` set to function result
- `%ebx` still has value of  $x$

`%ebx` x

```

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

```



With Slides from Bryant and O'Hallaron

## Recursive Call #4

```

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

```

### Assume

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

### Actions

- Compute  $(x \& 1) +$  computed value

### Effect

- `%eax` set to function result

`%ebx` x

```

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

```

With Slides from Bryant and O'Hallaron

## Recursive Call #5

```

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

```

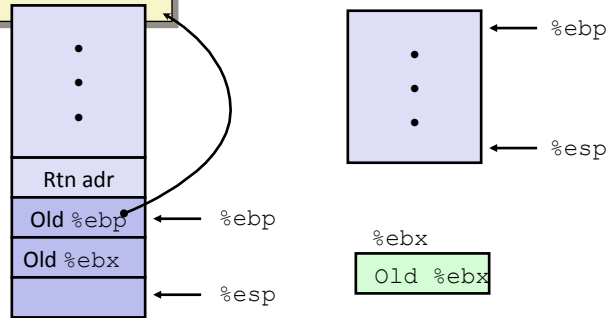
### Actions

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

```

...
L3:
    addl$4, %esp
    popl%ebx
    popl%ebp
    ret

```



With Slides from Bryant and O'Hallaron

## 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

With Slides from Bryant and O'Hallaron

## 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

With Slides from Bryant and O'Hallaron

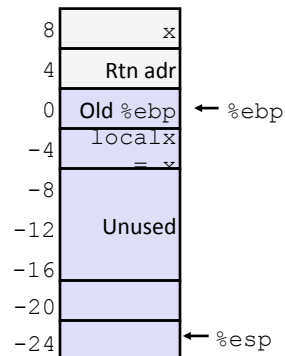
## Creating and Initializing Local Variable

```
int add3(int x) {
    int localx = x;
    incrk(&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
```



With Slides from Bryant and O'Hallaron



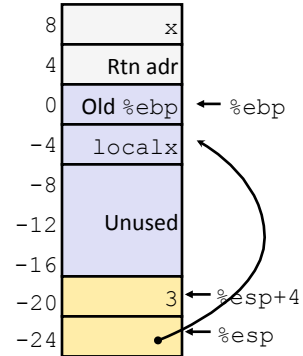
## Creating Pointer as Argument

```
int add3(int x) {
    int localx = x;
    incr(&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 incr
```



With Slides from Bryant and O'Hallaron

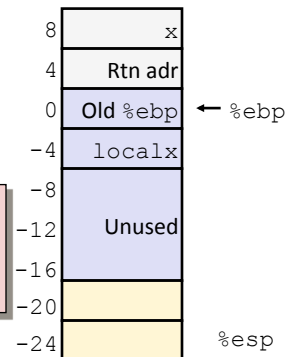
## Retrieving local variable

```
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
```



What happens with C code "return &x"?  
Can you return pointer to a local variable?

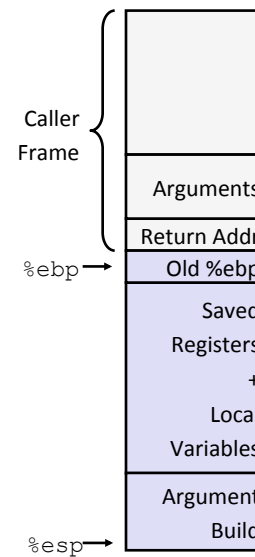
## Returning the address of a local variable

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

With Slides from Bryant and O'Hallaron

## 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



With Slides from Bryant and O'Hallaron