

# Machine-Level Programming: Procedures

These slides based on some provided by the authors of our  
textbook:  
Randal Bryant & David O'Hallaron

## Topics

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

# IA32 Stack

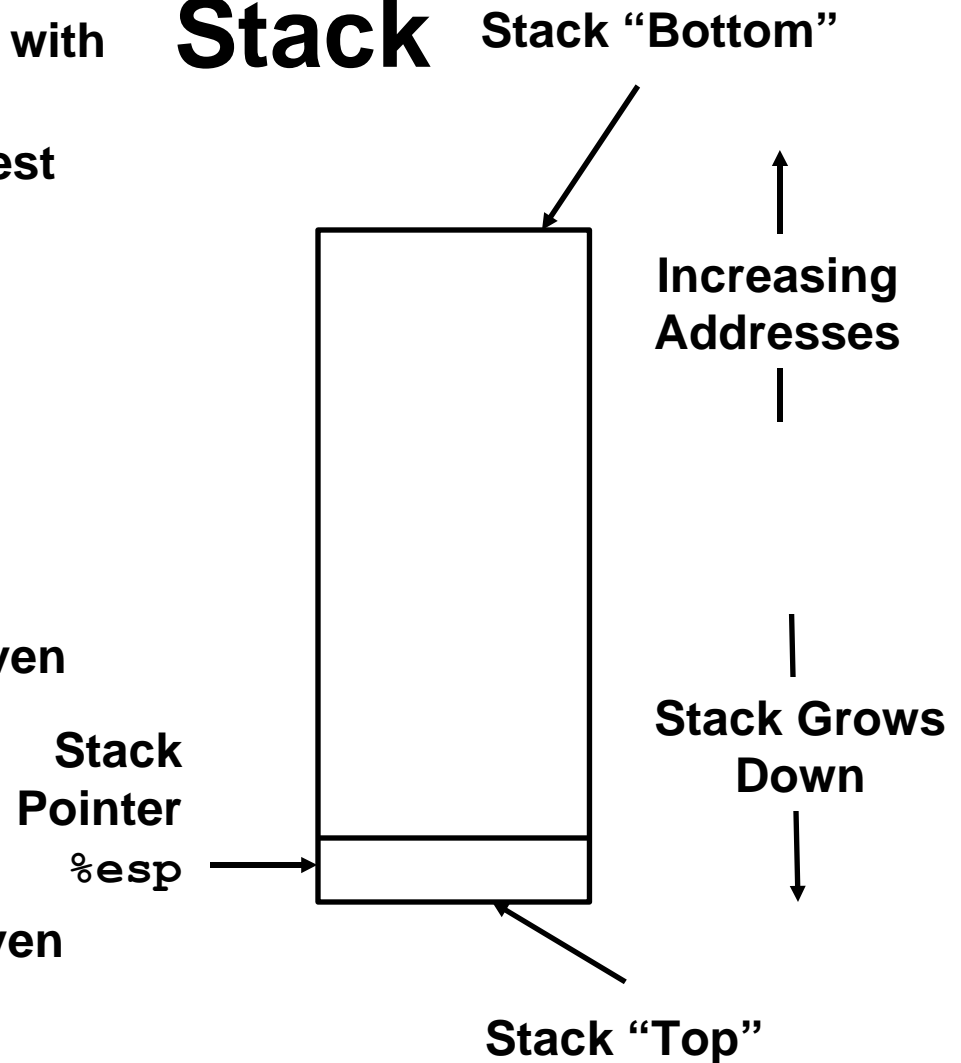
- Region of memory managed with stack discipline
- Register `%esp` indicates lowest allocated position in stack
  - i.e., address of top element

## Pushing

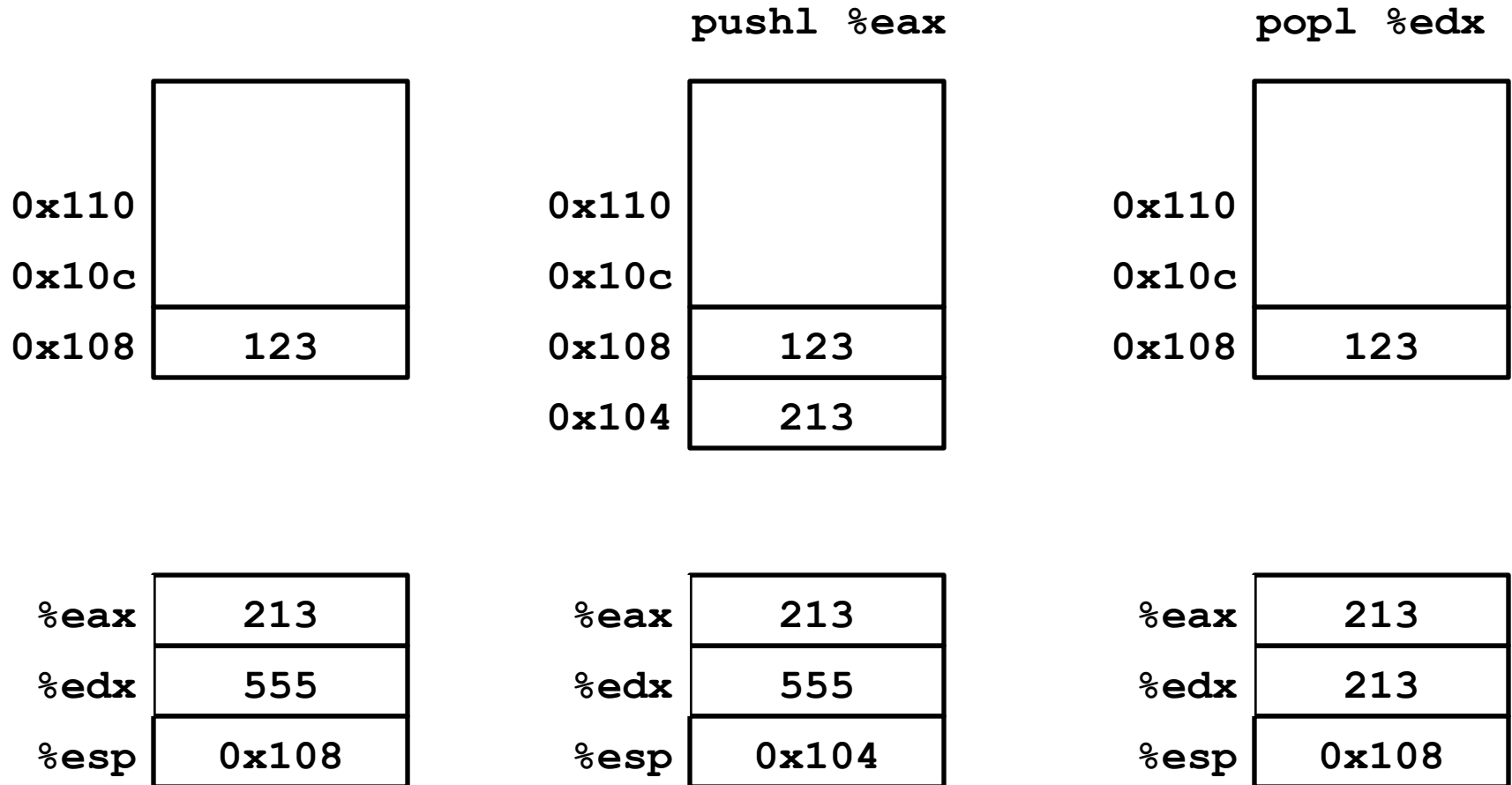
- `pushl Src`
- Fetch operand at `Src`
- Decrement `%esp` by 4
- Write operand at address given by `%esp`

## Popping

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



# Stack Operation Examples



# Procedure Control Flow

Use stack to support procedure call and return

## Procedure call:

`call label` Push return address on stack; Jump to `label`

## Return address value

- Address of instruction beyond `call`
- Example from disassembly

```
804854e:  e8 3d 06 00 00  call    8048b90 <main>
```

```
8048553:  50                pushl   %eax
```

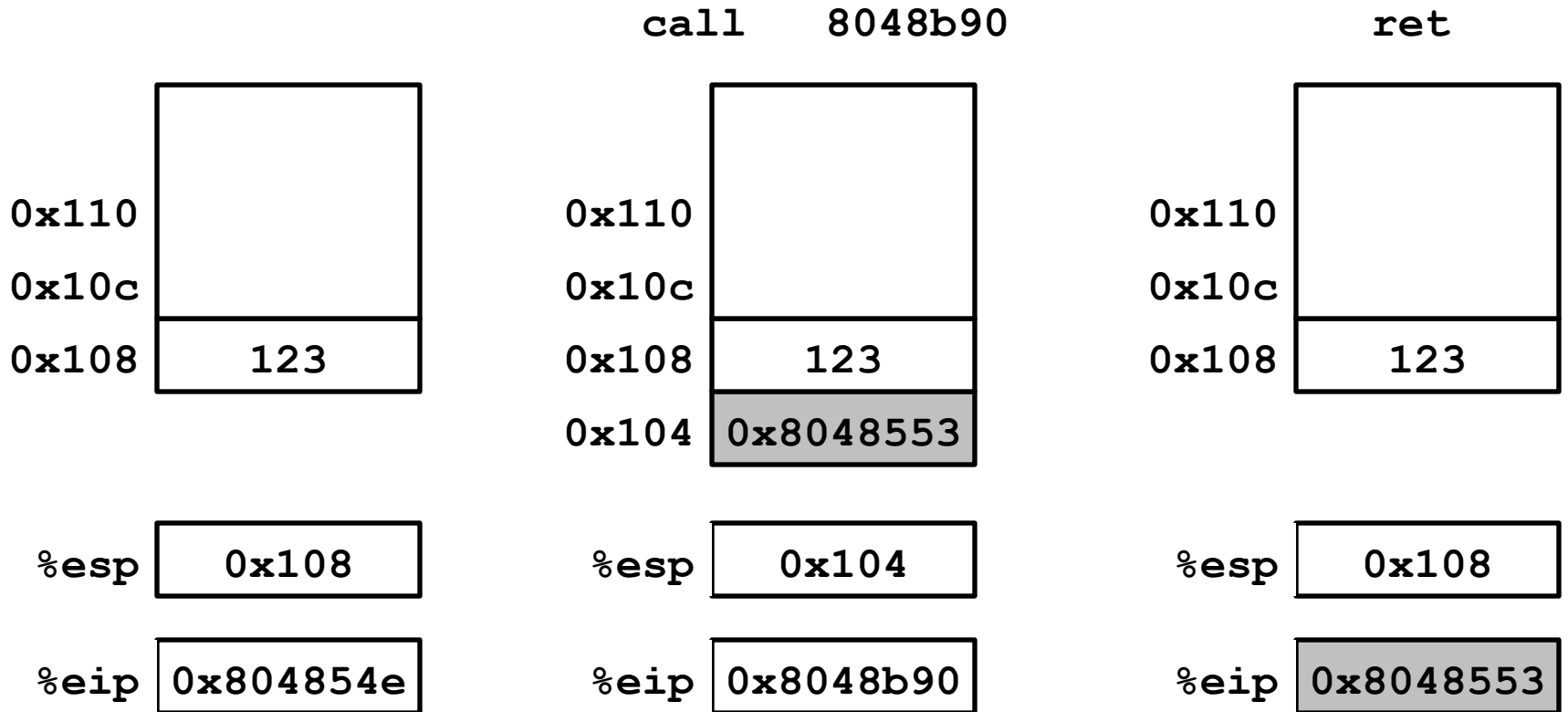
– Return address = `0x8048553`

## Procedure return:

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

# Procedure Call / Return Example

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



`%eip` is program counter

# Stack-Based Languages

## Languages that Support Recursion

- e.g., C, Pascal, Java
- Code must be “*Reentrant*”
  - Multiple simultaneous instantiations of single procedure
- **Need some place to store state of each instantiation**
  - Arguments
  - Local variables
  - Return pointer

## Stack Discipline

- **State for given procedure needed for limited time**
  - From when called to when return
- **Callee returns before caller does**

## Stack Allocated in *Frames*

- **state for single procedure instantiation**

# Call Chain Example

## Code Structure

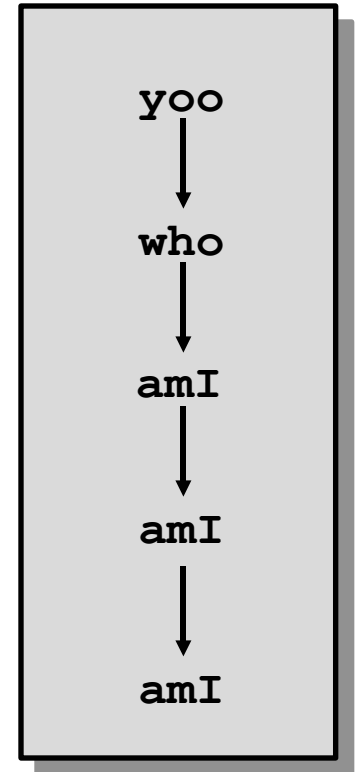
```
yoo (...)  
{  
  •  
  •  
  who ();  
  •  
  •  
}
```

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

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

- Procedure `amI` recursive

## Call Chain



# IA32 Stack Structure

## Stack Growth

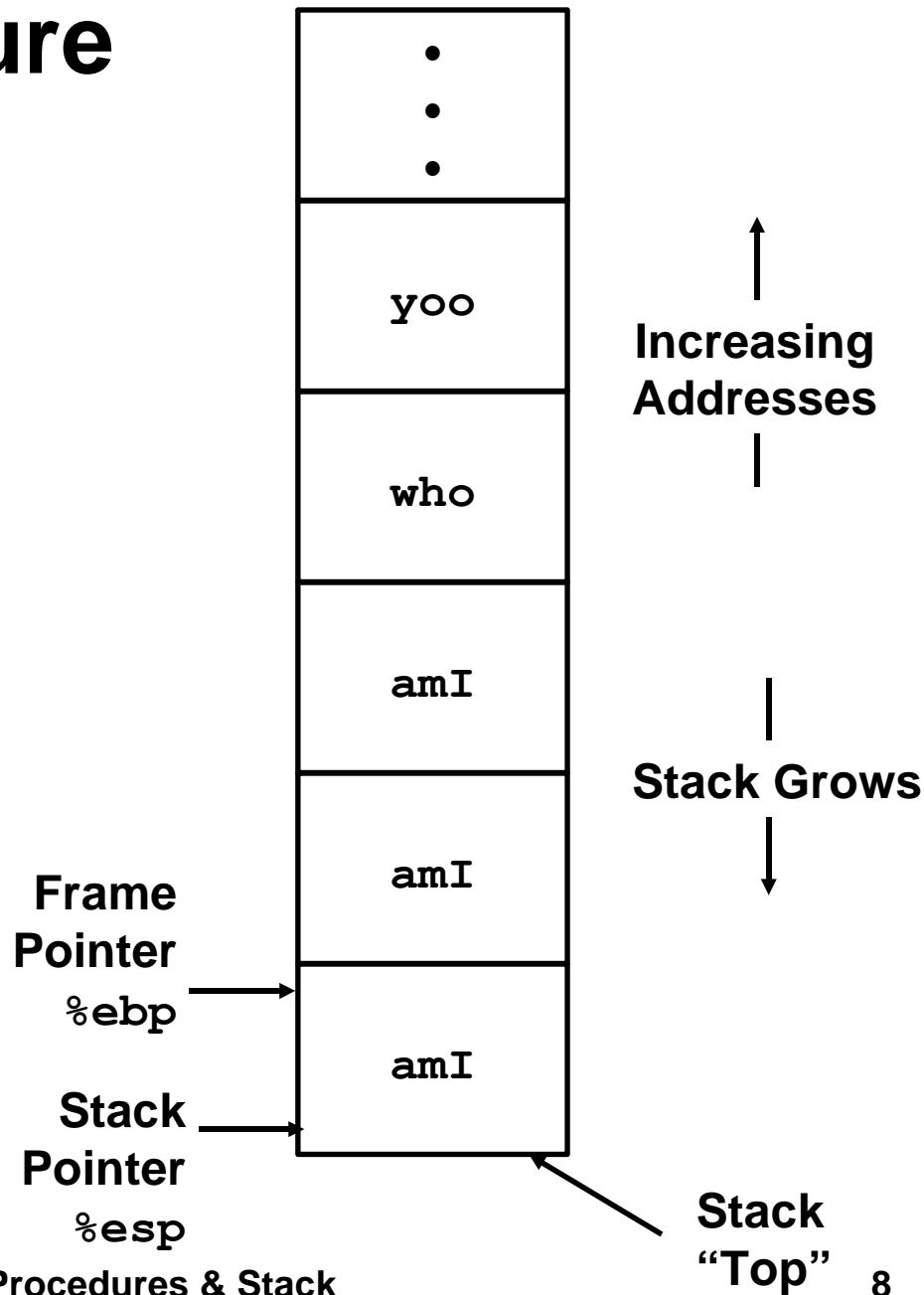
- Toward lower addresses

## Stack Pointer

- Address of next available location in stack
- Use register `%esp`

## Frame Pointer

- Start of current stack frame
- Use register `%ebp`



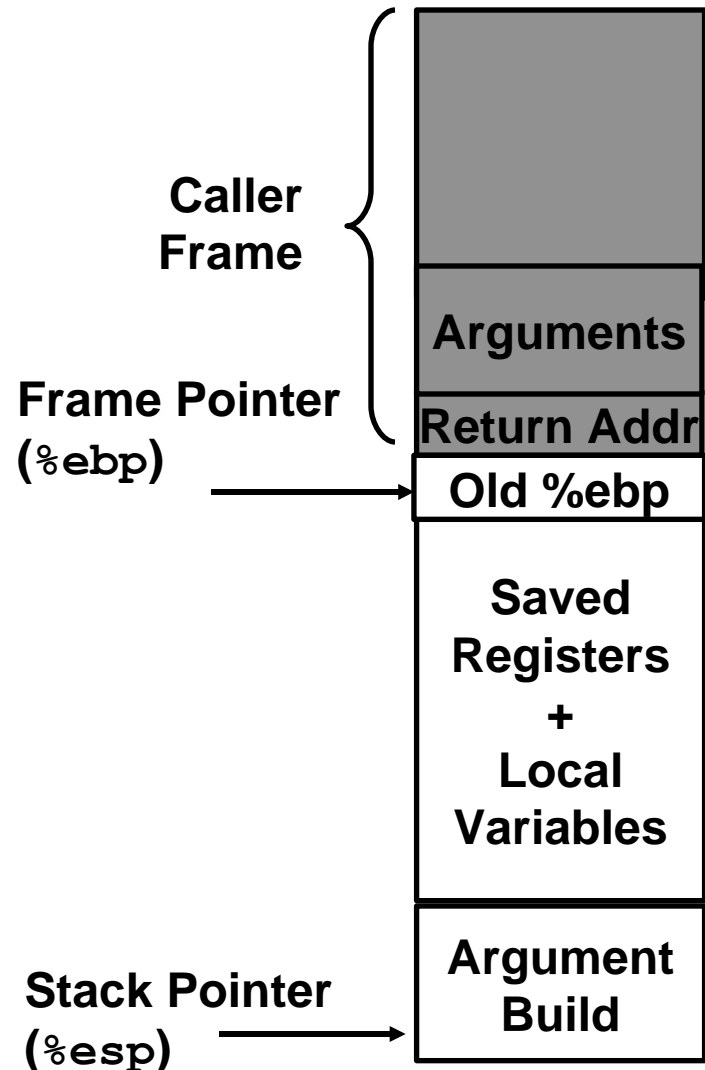
# IA32/Linux Stack Frame

## Callee Stack Frame (“Top” to Bottom)

- Parameters for called functions
- 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



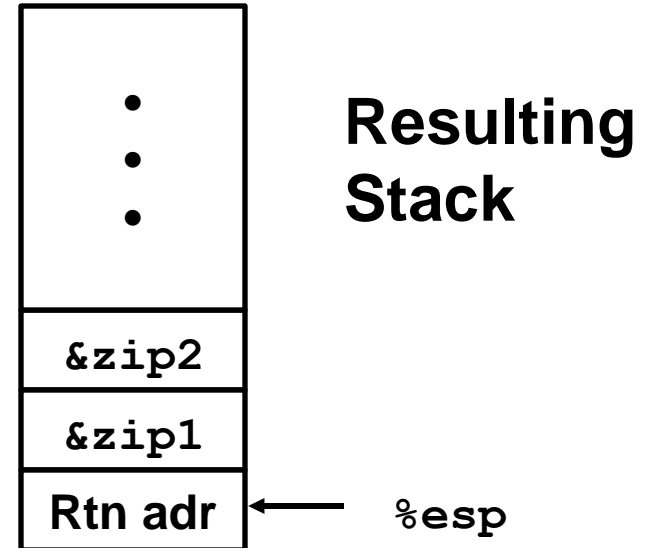
# swap

```
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;
}
```

```
call_swap:
    . . .
    pushl $zip2    # Global Var
    pushl $zip1    # Global Var
    call swap
    . . .
```



# swap

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

swap:

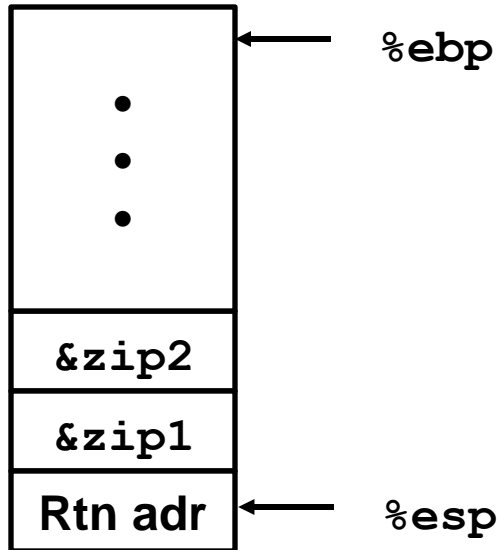
```
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
} Set Up

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

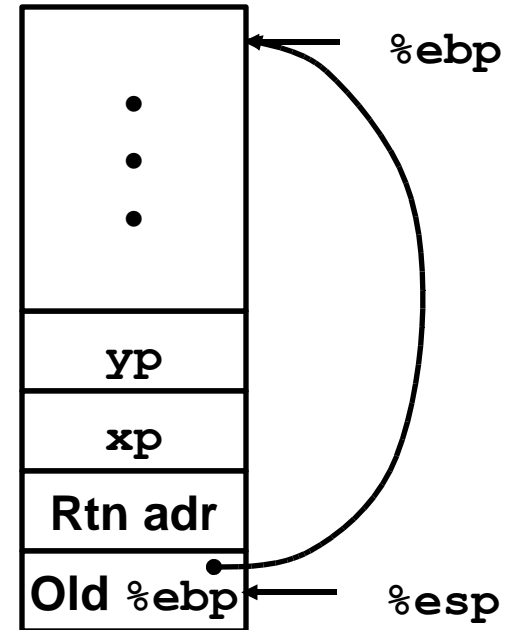
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
} Finish
```

# swap Setup #1

## Entering Stack



## Resulting Stack



`swap:`

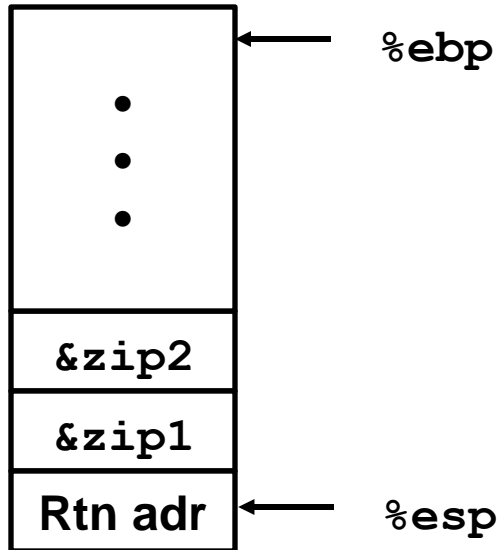
```
pushl %ebp
```

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

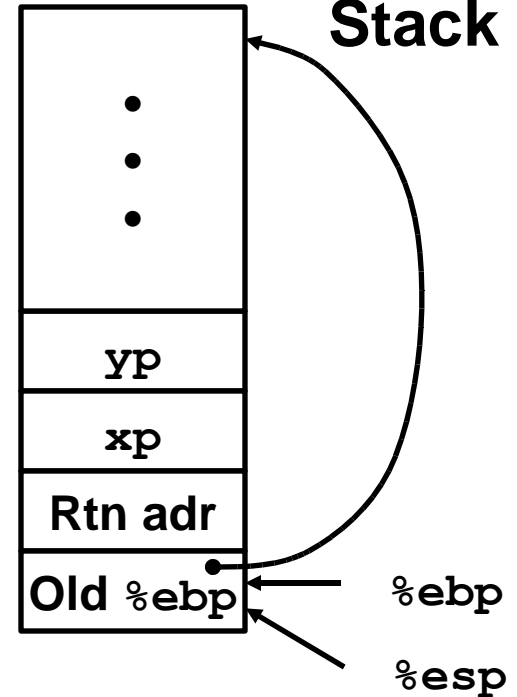
```
pushl %ebx
```

# swap Setup #2

Entering Stack



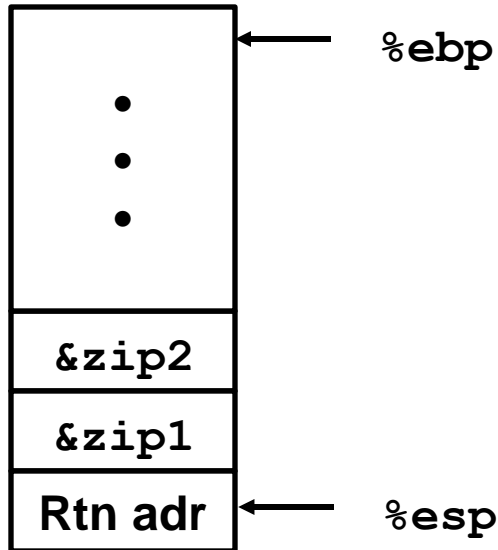
Resulting Stack



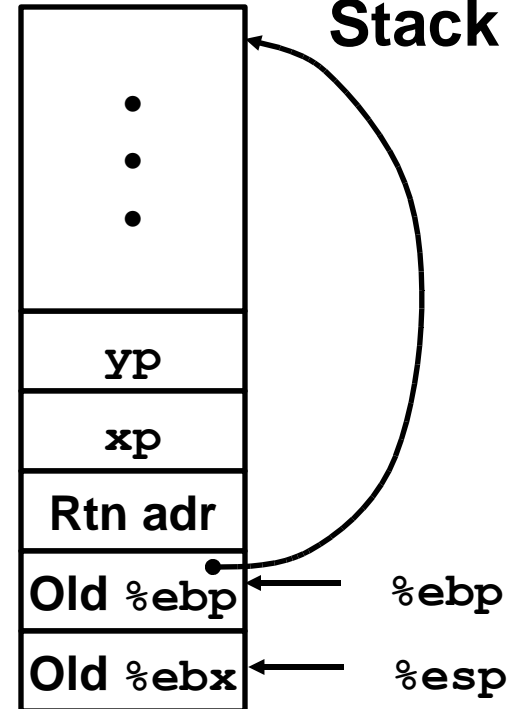
```
swap:  
    pushl %ebp  
    movl %esp, %ebp  
    pushl %ebx
```

# swap Setup #3

Entering  
Stack



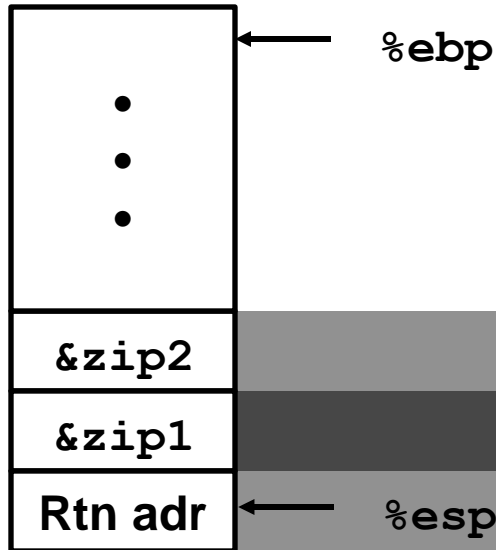
Resulting  
Stack



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

# Effect of swap Setup

Entering Stack



Offset

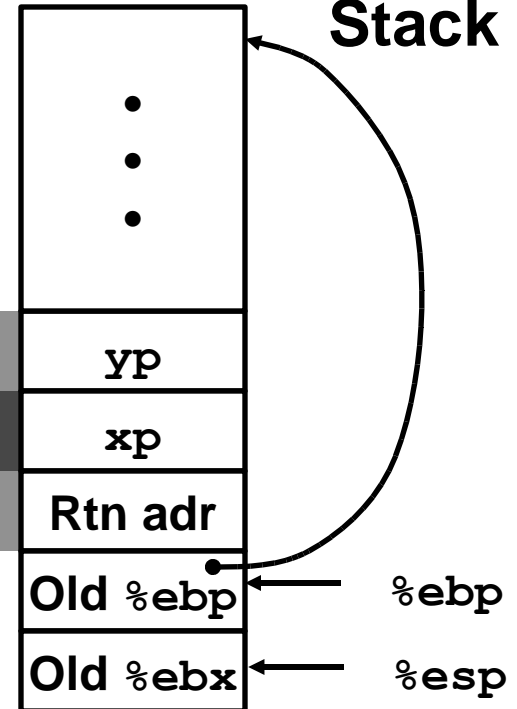
12

8

4

0

Resulting Stack



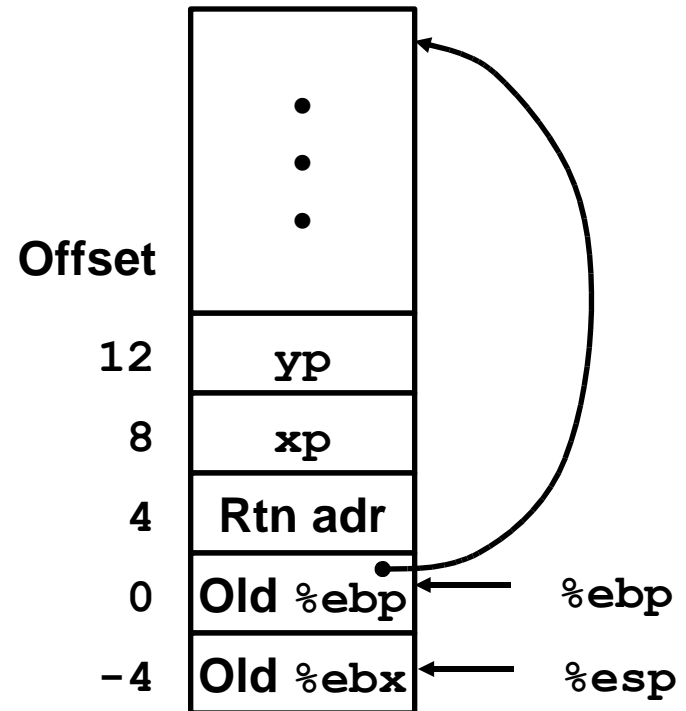
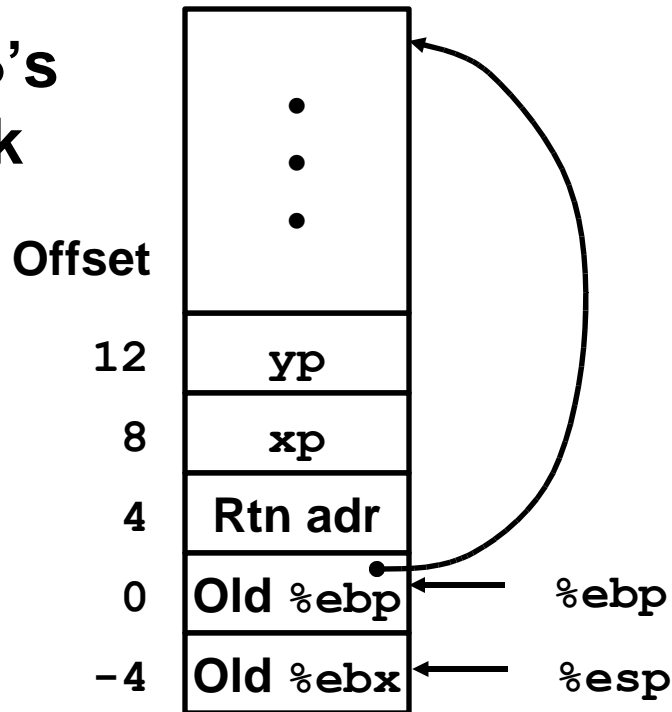
swap:

```

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

# swap Finish #1

swap's  
Stack

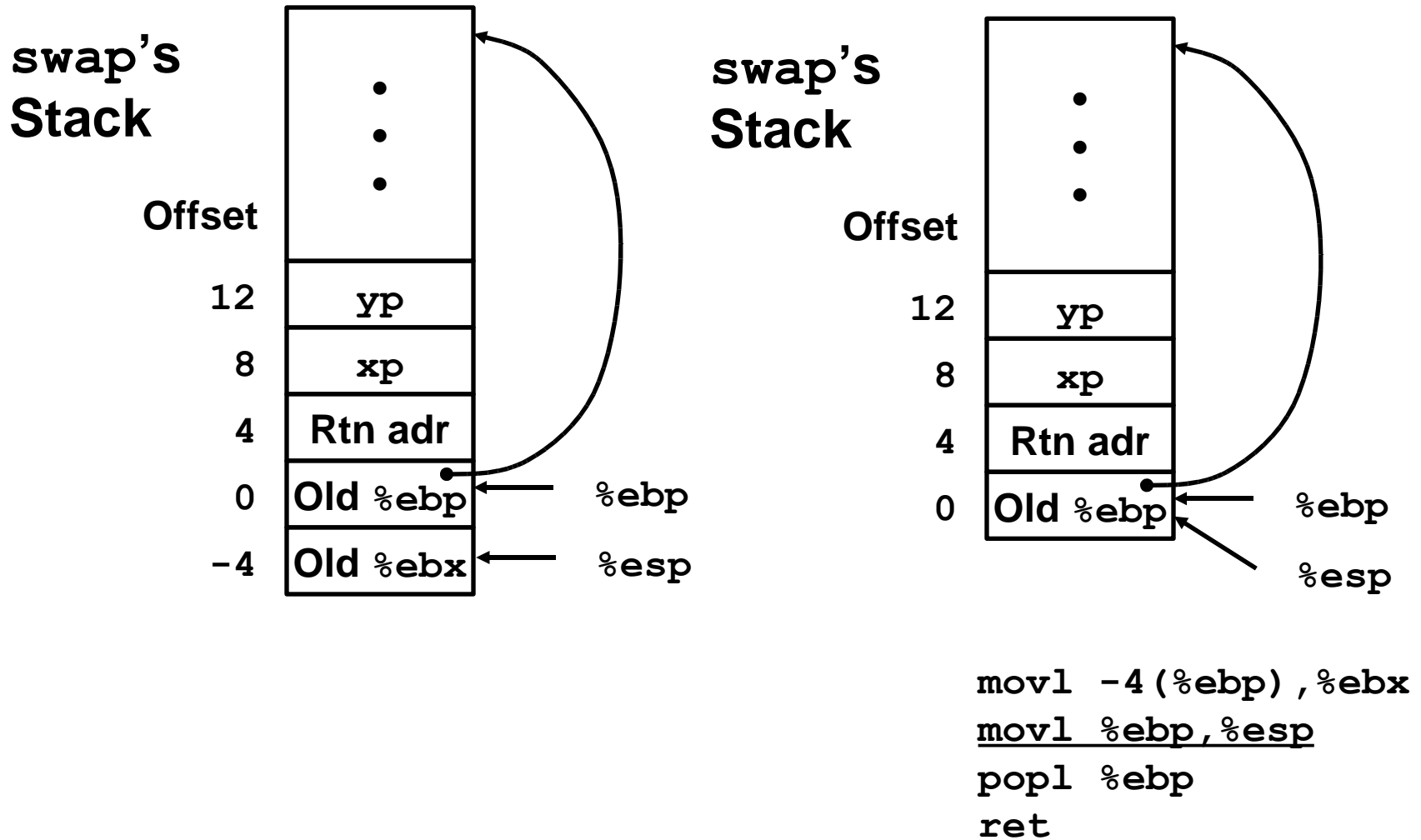


```
movl -4(%ebp), %ebx  
movl %ebp, %esp  
popl %ebp  
ret
```

## Observation

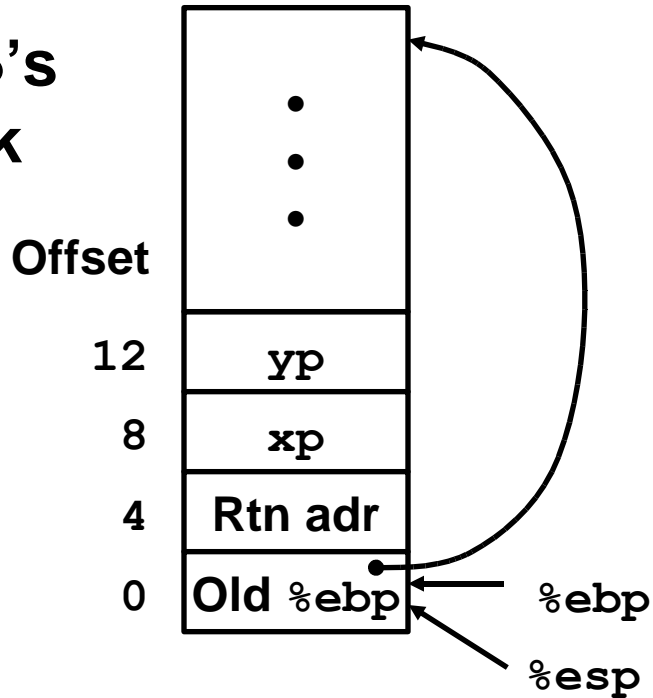
- Saved & restored register %ebx

# swap Finish #2

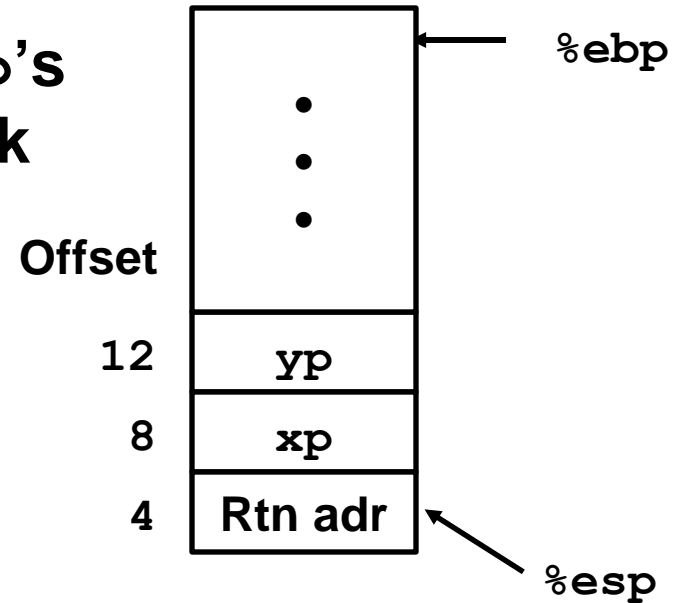


# swap Finish #3

swap's  
Stack

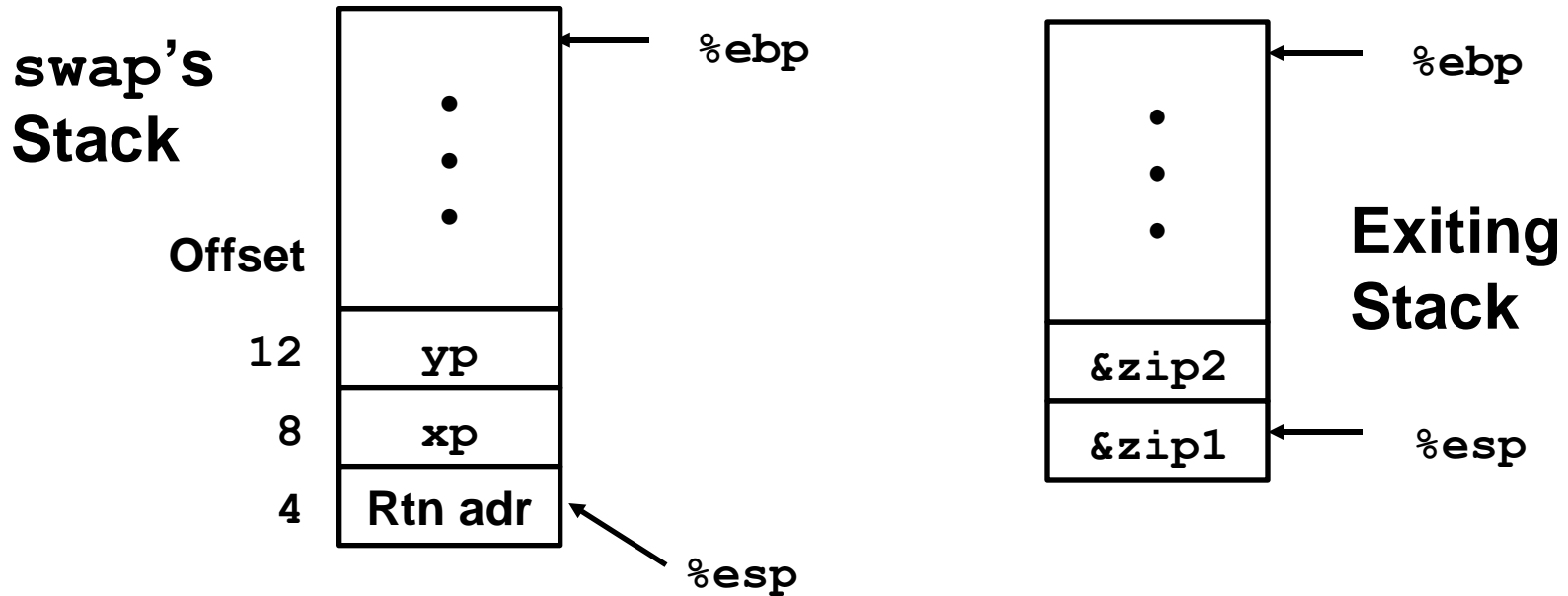


swap's  
Stack



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

# 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
```

# 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`

## Conventions

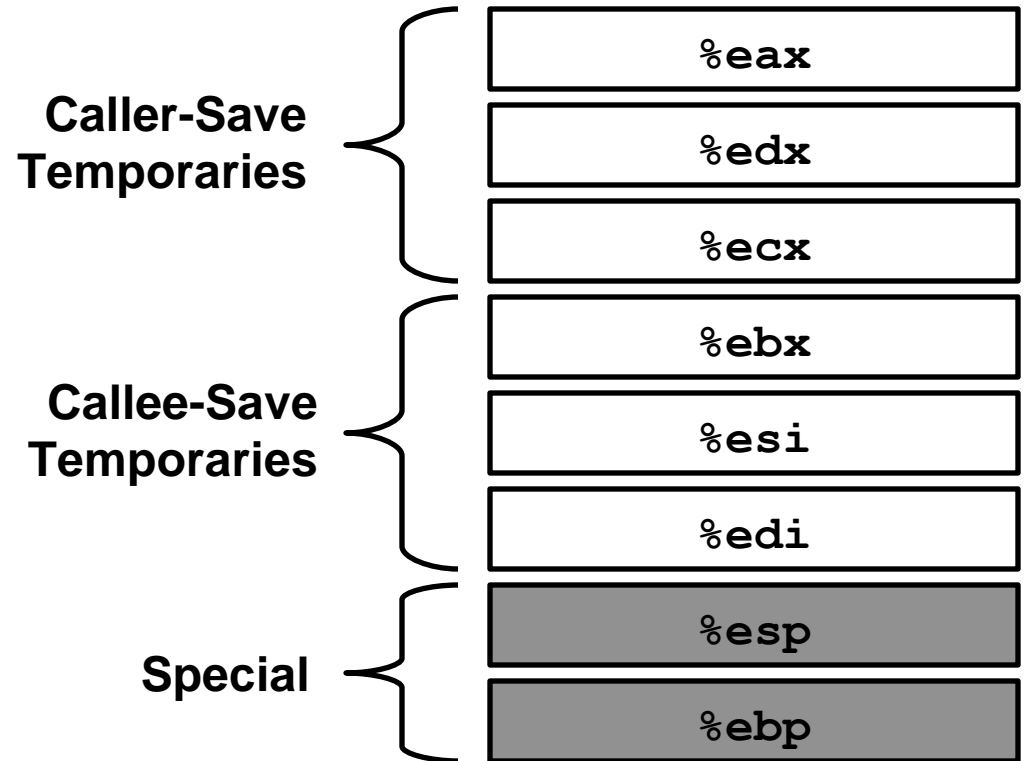
- “**Caller Save**”
  - Caller saves temporary in its frame before calling
- “**Callee Save**”
  - Callee saves temporary in its frame before using

# IA32/Linux Register Usage

- Surmised by looking at code examples

## Integer Registers

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



# Recursive Factorial

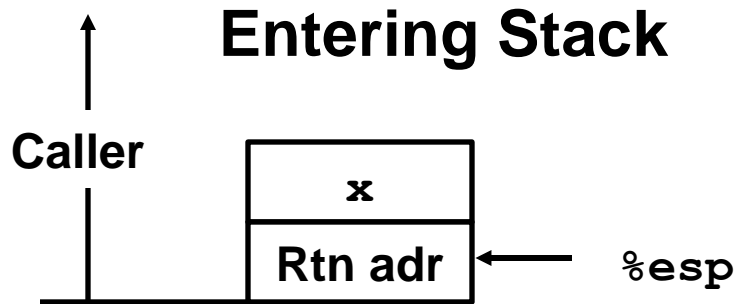
```
int rfact(int x)
{
    int rval;
    if (x <= 1)
        return 1;
    rval = rfact(x-1);
    return rval * x;
}
```

## Complete Assembly

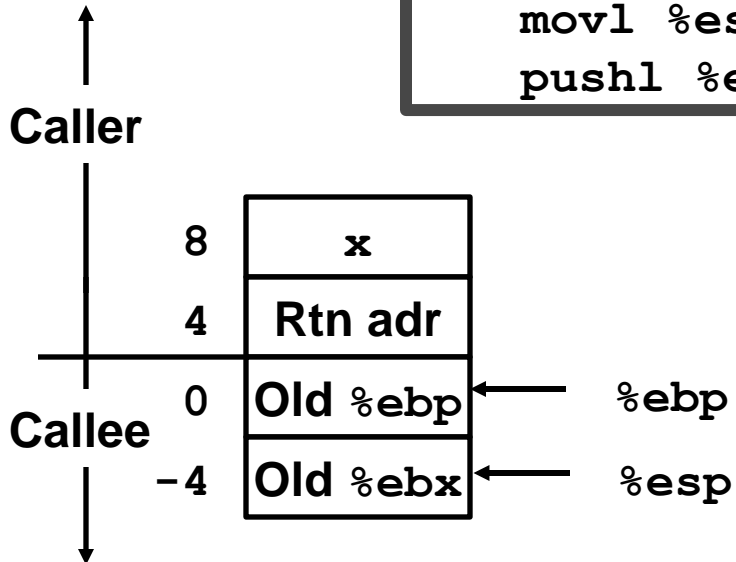
- **Assembler directives**
  - Lines beginning with “.”
  - Not of concern to us
- **Labels**
  - .Lxx
- **Actual instructions**

```
.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
```

# Rfact Stack Setup



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



# Rfact Body

Recursion

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

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

## Registers

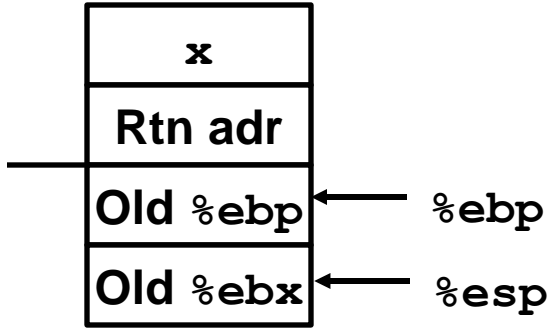
\$ebx Stored value of x

\$eax

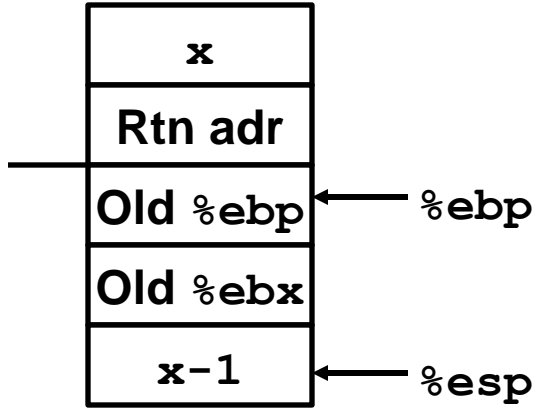
- Temporary value of x-1
- Returned value from rfact(x-1)
- Returned value from this call

# Rfact Recursion

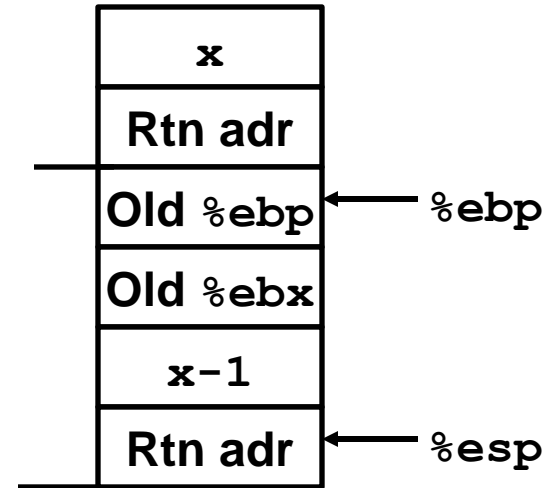
```
leal -1(%ebx), %eax
```



```
pushl %eax
```

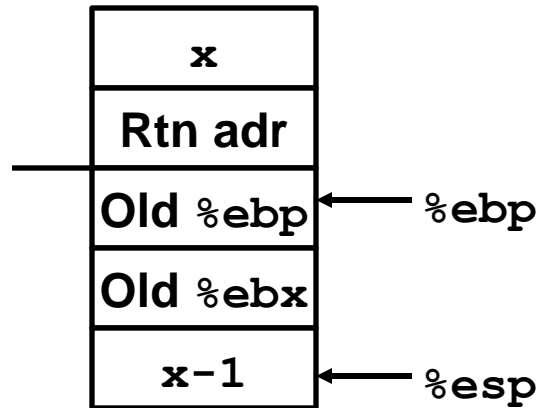


```
call rfact
```

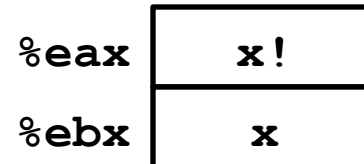
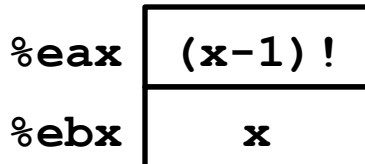
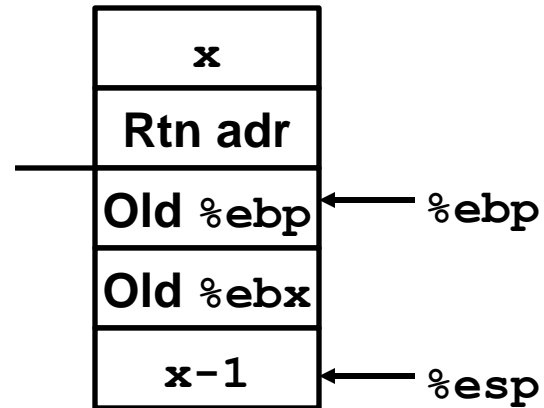


# Rfact Result

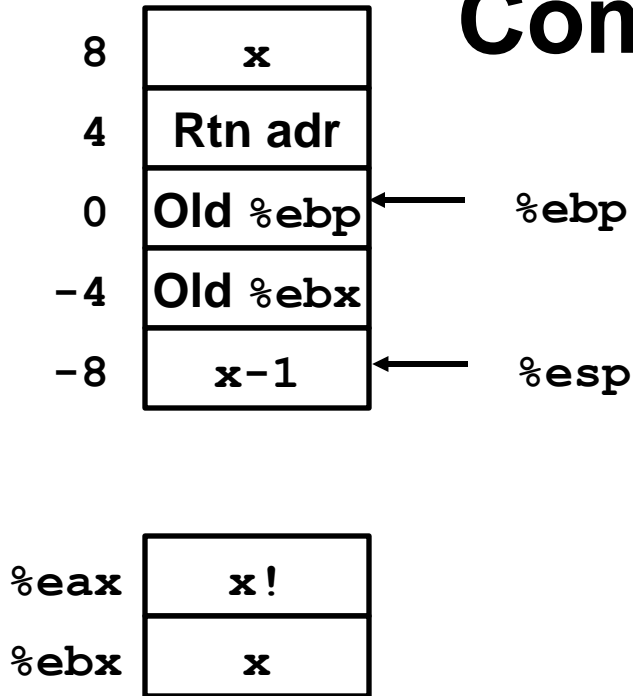
Return from Call



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



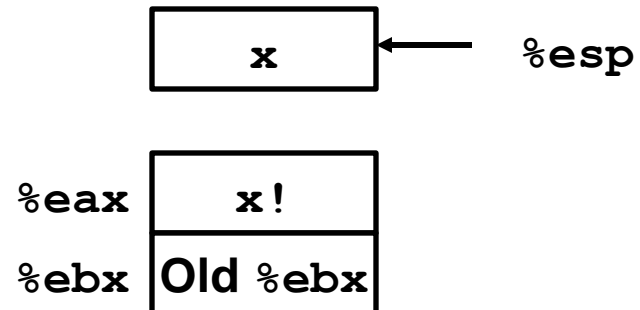
# Rfact Completion



```

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

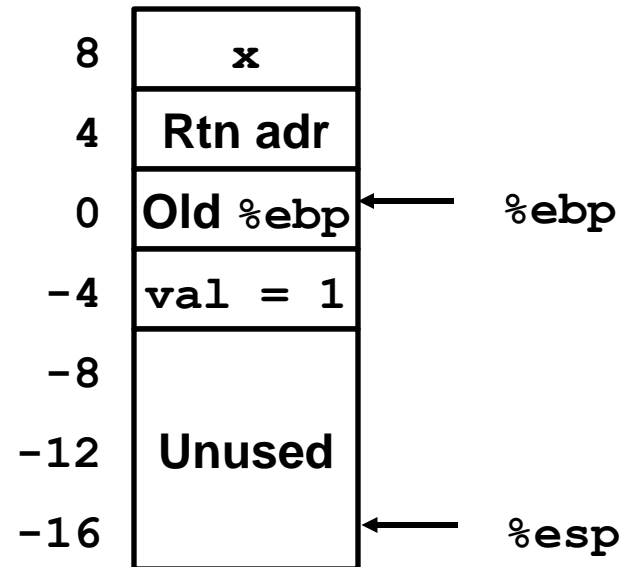
```
int sfact(int x)
{
    int val = 1;
    s_helper(x, &val);
    return val;
}
```

- Pass pointer to update location
- Uses tail recursion
  - But GCC only partially optimizes it

# Creating & Initializing Pointer

## 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
```



## Using Stack for Local Variable

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

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

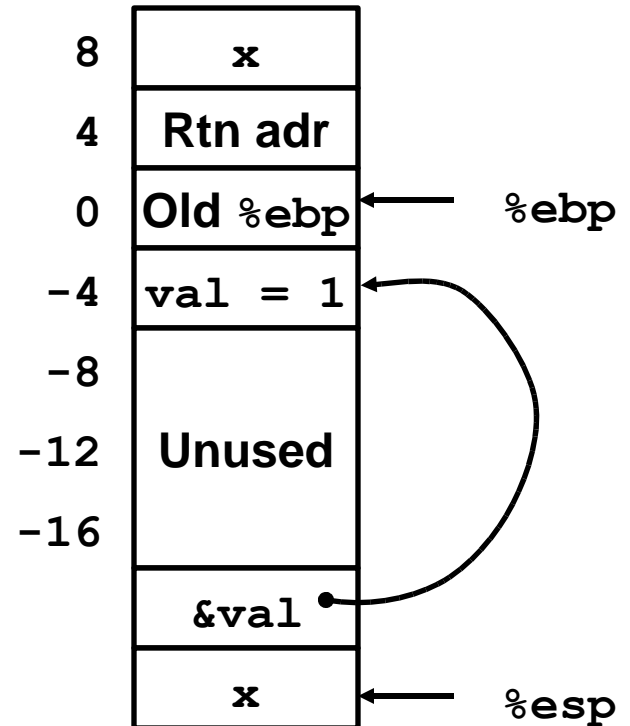
# Passing Pointer

## 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
```

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

## Stack at time of call



# Using Pointer

```
void s_helper
(int x, int *accum)
{
    • • •
    int z = *accum * x;
    *accum = z;
    • • •
}
```

```
• • •
movl %ecx,%eax    # z = x
imull (%edx),%eax # z *= *accum
movl %eax,(%edx)  # *accum = z
• • •
```

- Register `%ecx` holds `x`
- Register `%edx` holds `accum`
  - Use access `(%edx)` to reference memory

# Tail Recursion

## Tail Recursive Procedure

```
int t_helper
  (int x, int val)
{
  if (x <= 1)
    return val;
  return
    t_helper(x-1, val*x);
}
```

## General Form

```
t_helper(x, val)
{
  . . .
  return
    t_helper(Xexpr, Vexpr)
}
```

## Top-Level Call

```
int tfact(int x)
{
  return t_helper(x, 1);
}
```

## Form

- Directly return value returned by recursive call

## Consequence

- Can convert into loop

# Removing Tail Recursion

## Optimized General Form

```
t_helper(x, val)
{
  start:
    . . .
    val = Vexpr;
    x = Xexpr;
    goto start;
}
```

## Resulting Code

```
int t_helper
(int x, int val)
{
  start:
    if (x <= 1)
      return val;
    val = val*x;
    x = x-1;
    goto start;
}
```

## Effect of Optimization

- Turn recursive chain into single procedure
- No stack frame needed
- Constant space requirement
  - Vs. linear for recursive version

# Generated Code for Tail Recursive Proc.

## Optimized Form

```
int t_helper
(int x, int val)
{
  start:
  if (x <= 1)
    return val;
  val = val*x;
  x = x-1;
  goto start;
}
```

## Code for Loop

```
# %edx = x
# %ecx = val
L53:                                     # start:
  cmp1 $1,%edx                          # x : 1
  jle L52                                # if <= goto done
  mov1 %edx,%eax                          # eax = x
  imull %ecx,%eax                         # eax = val * x
  decl %edx                                # x--
  mov1 %eax,%ecx                           # val = val * x
  jmp L53                                  # goto start
L52:                                     # done:
```

## Registers

```
$edx x
$ecx val
```

# Summary

## The Stack Makes Recursion Work

- **Private storage for each *instance* of procedure call**
  - Instantiations don't clobber each other
  - Addressing of locals + arguments can be relative to stack positions
- **Can be managed by stack discipline**
  - Procedures return in inverse order of calls

## IA32 Procedures Combination of Instructions + Conventions

- **Call / Ret instructions**
- **Register usage conventions**
  - Caller / Callee save
  - `%ebp` and `%esp`
- **Stack frame organization conventions**