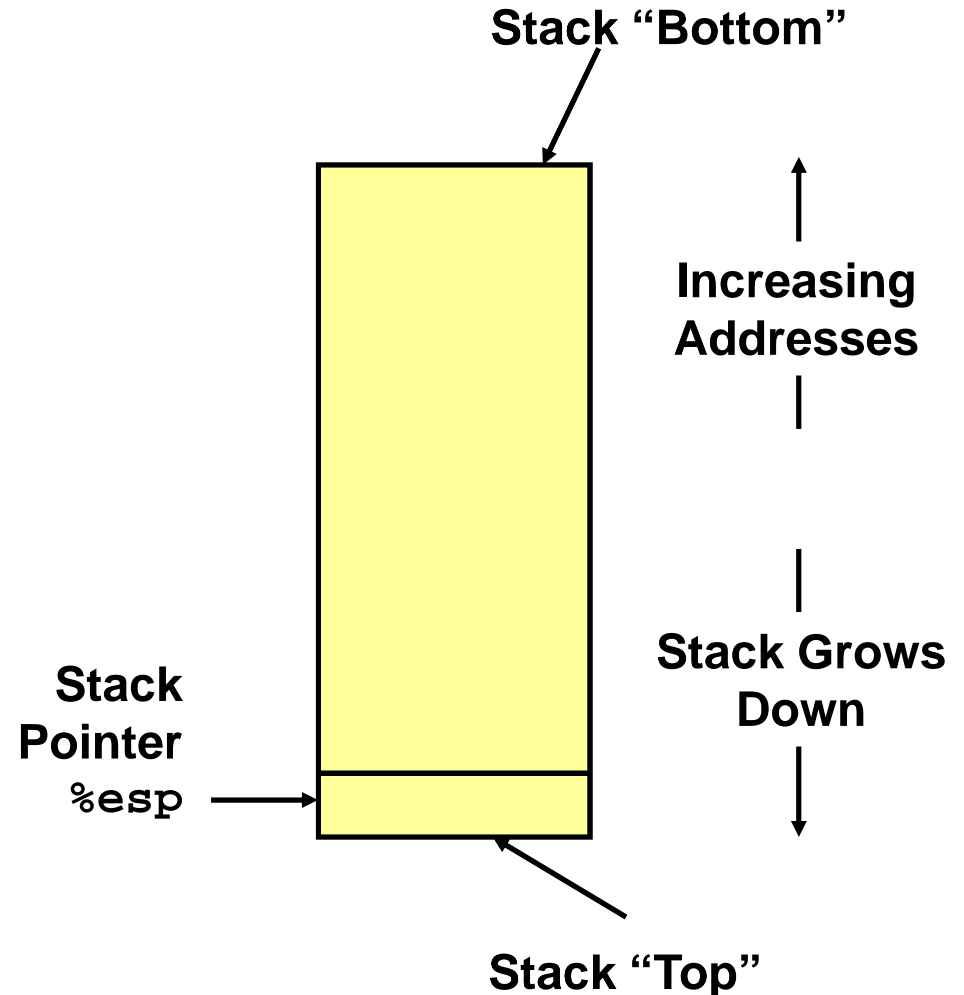
Assembly III: Procedures

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu

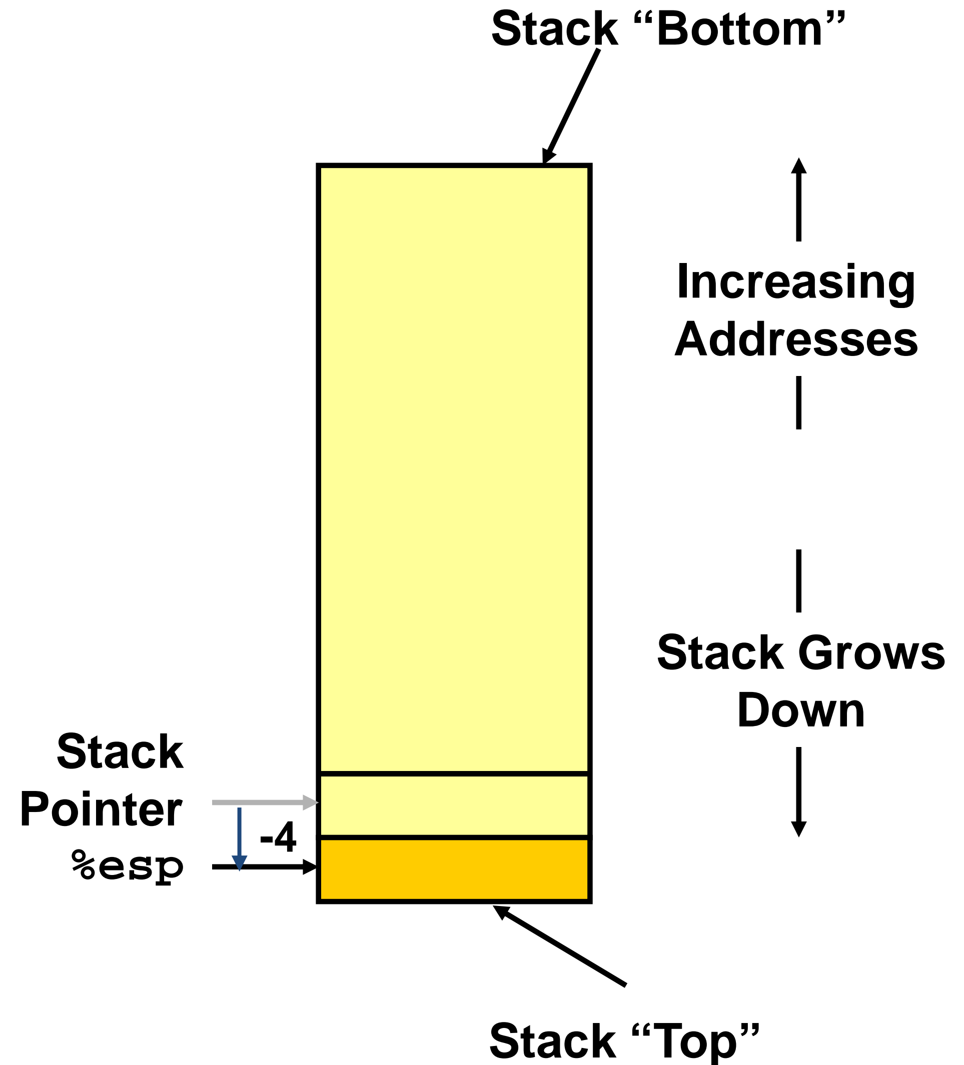
IA-32 Stack (1)

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



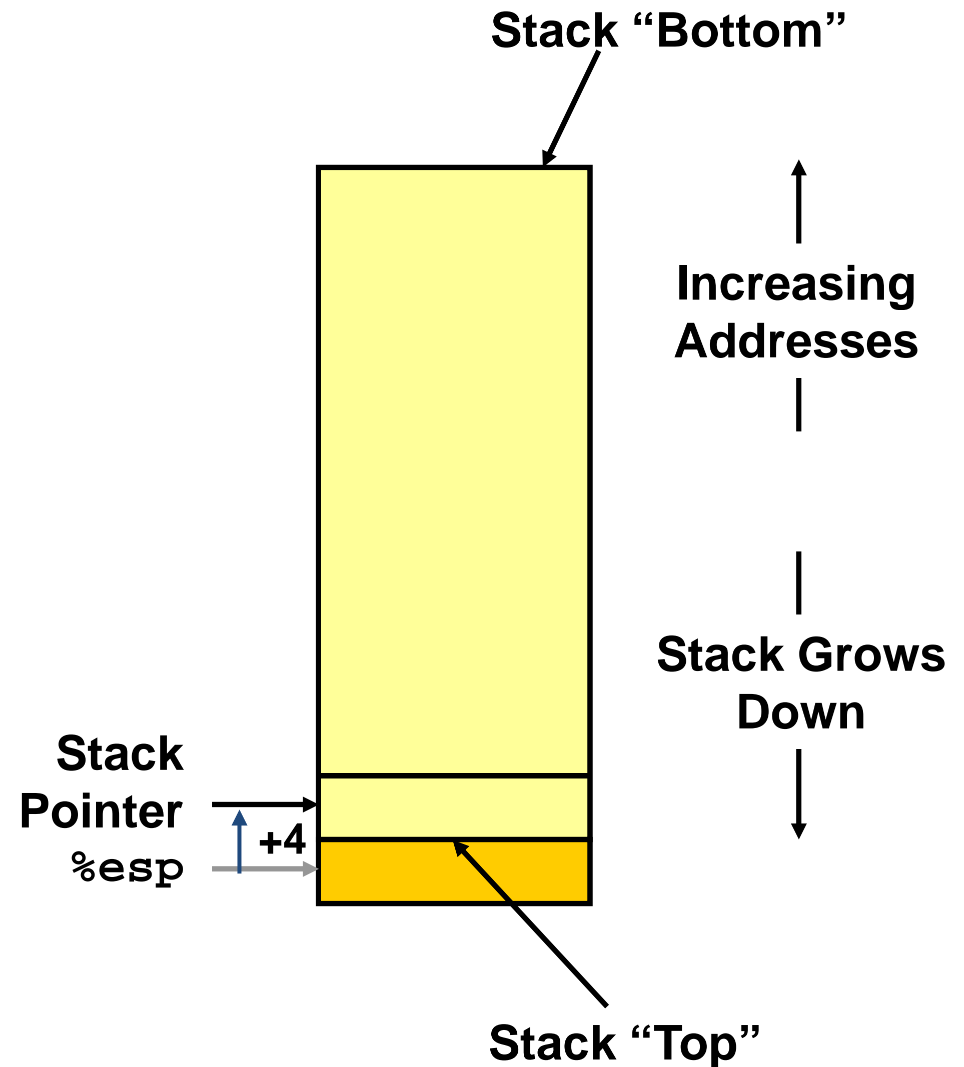
IA-32 Stack (2)

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



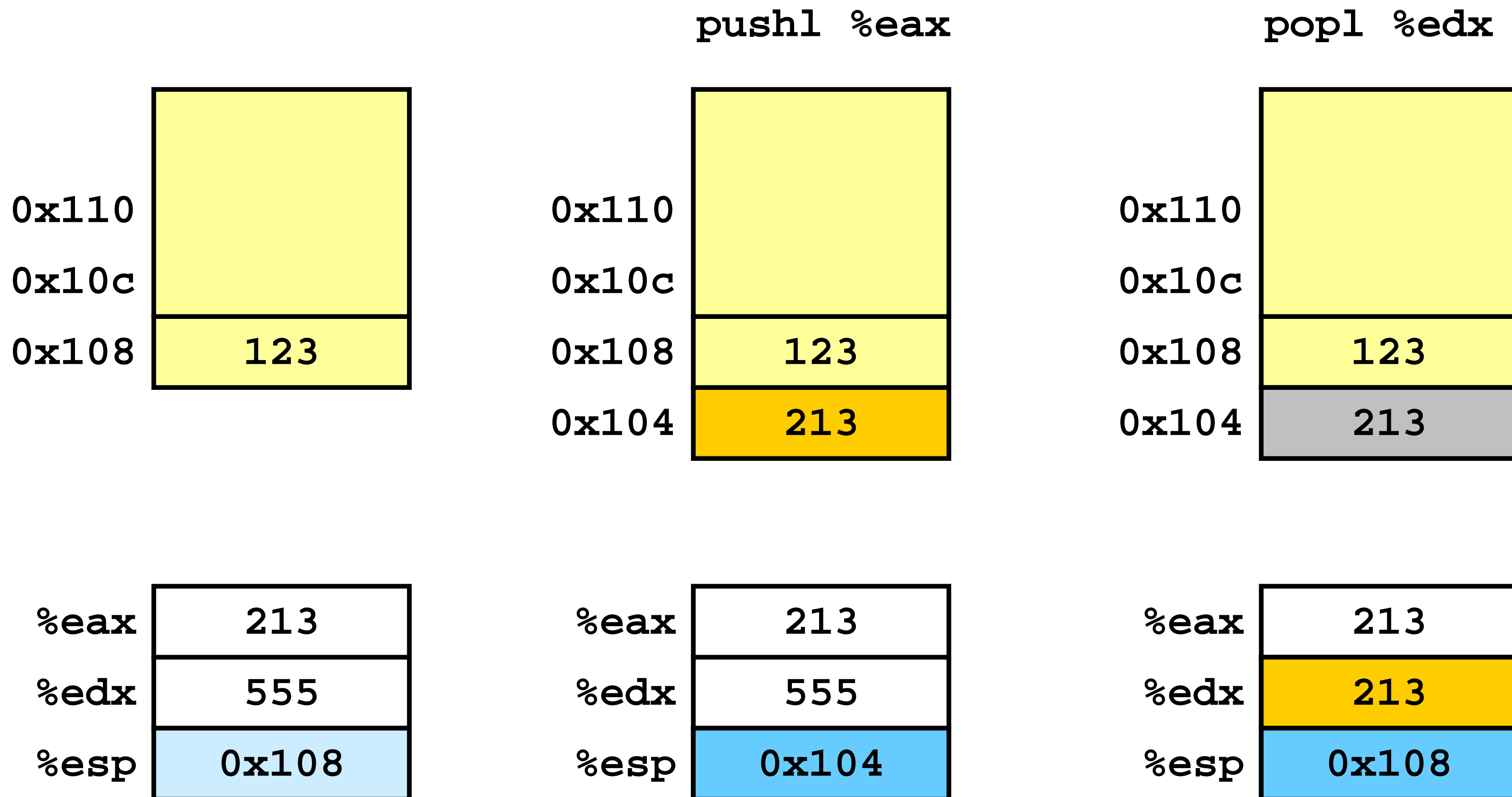
IA-32 Stack (3)

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



IA-32 Stack (4)

- **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
- **Procedure return**
  - ret
    - Pop address from stack
    - Jump to address

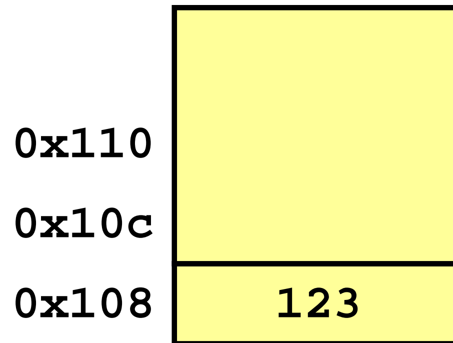
Procedure Call Example

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

0x08048553

+0x0000063d

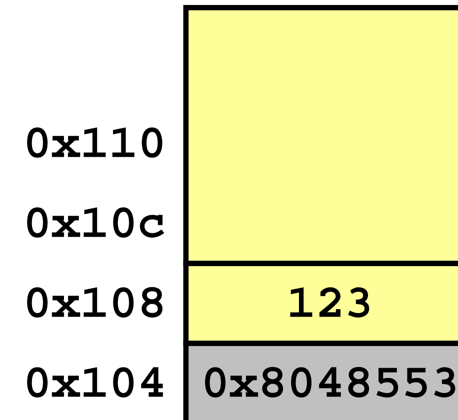
=0x08048b90



%esp 0x108

%eip 0x804854e

call 8048b90



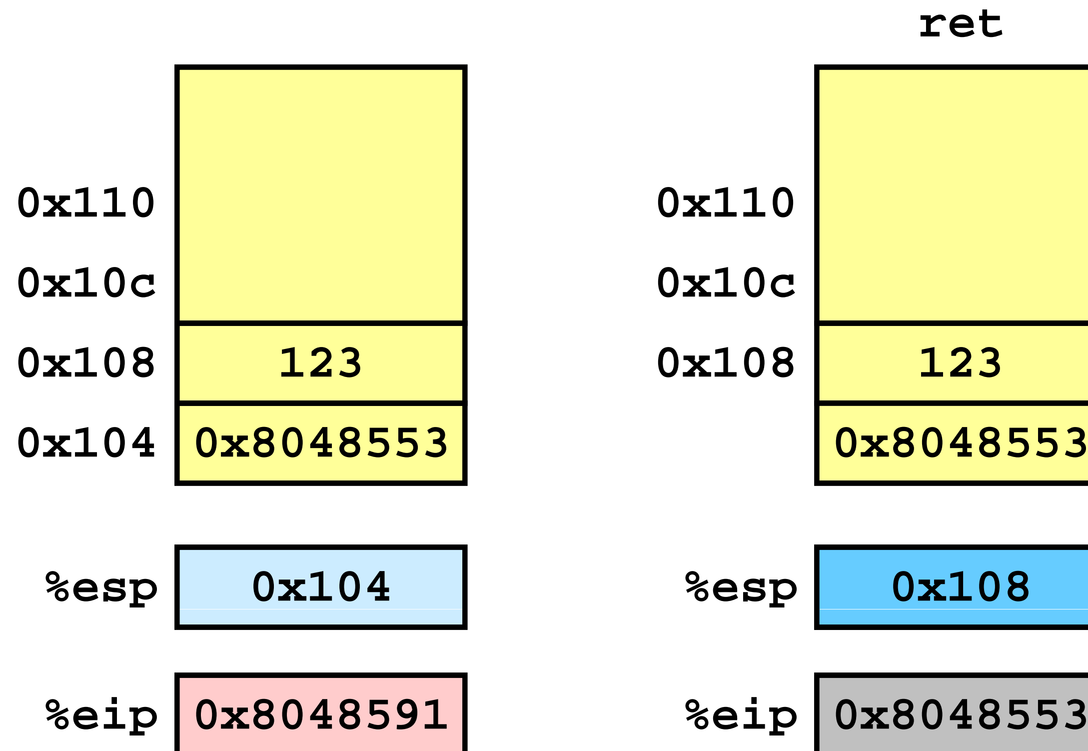
%esp 0x104

%eip 0x8048b90

%eip is program counter

Procedure Return Example

```
8048591:   c3                ret
```



`%eip` **is program counter**

Stack-based Languages

- **Languages that support recursion**
  - e.g., C, Pascal, Java, etc.
  - 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

Stack Frames (1)

Code Structure

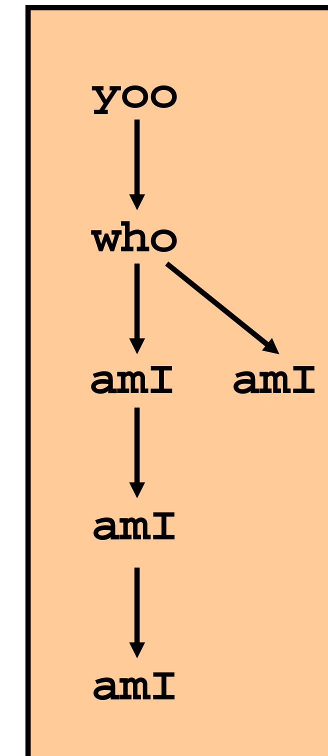
```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

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

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

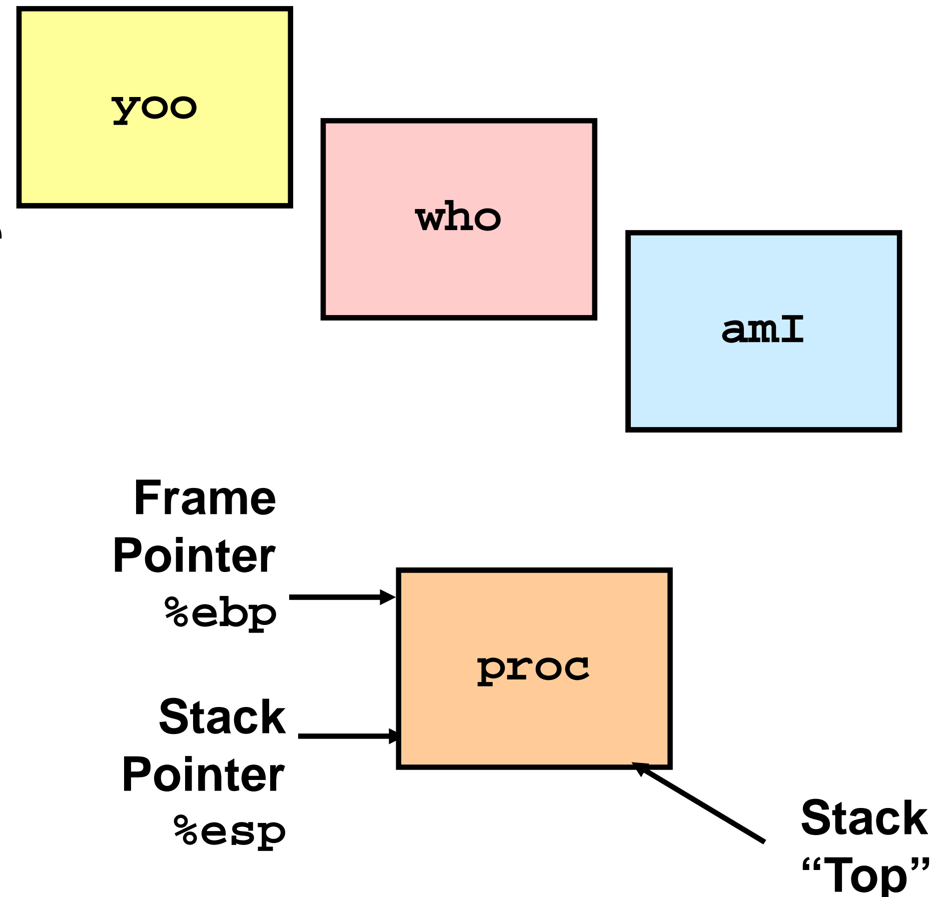
- Procedure **`amI`** recursive

Call Chain

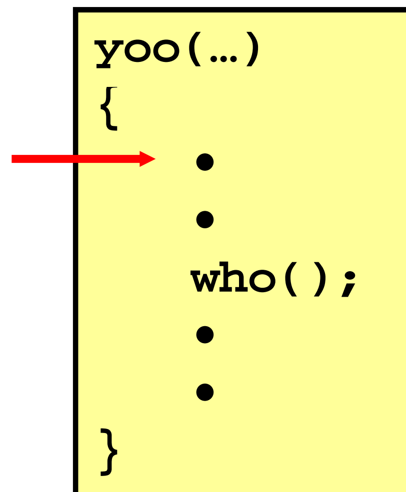


Stack Frames (2)

- **Contents**
  - Return information
  - Arguments
  - Local variables & temp 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

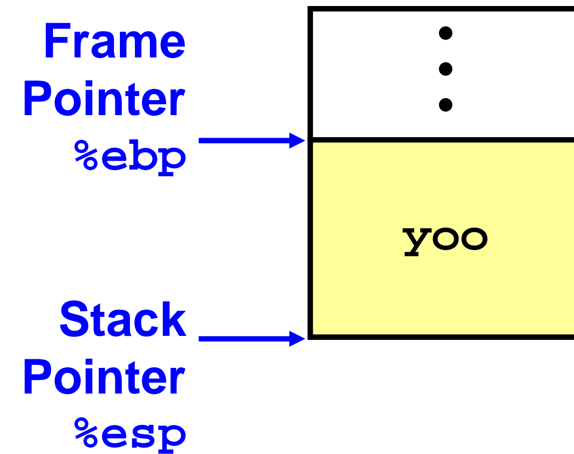


Stack Frames (3)

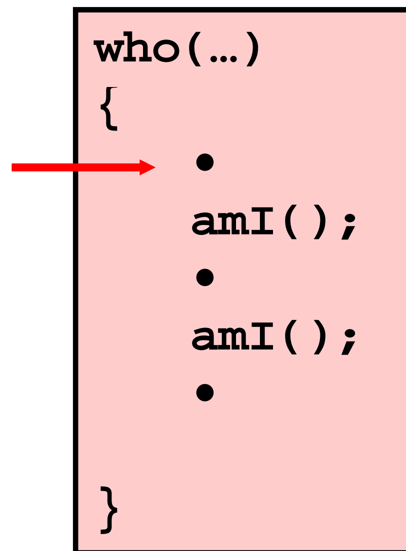


Call Chain

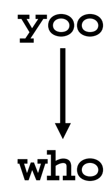
yoo

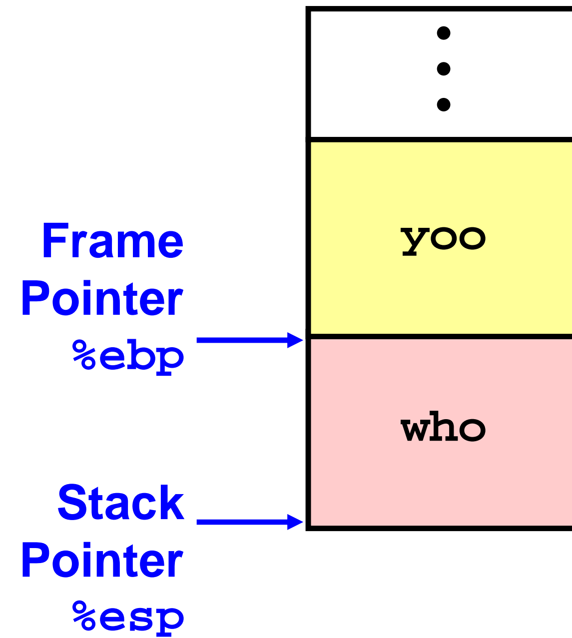


Stack Frames (4)

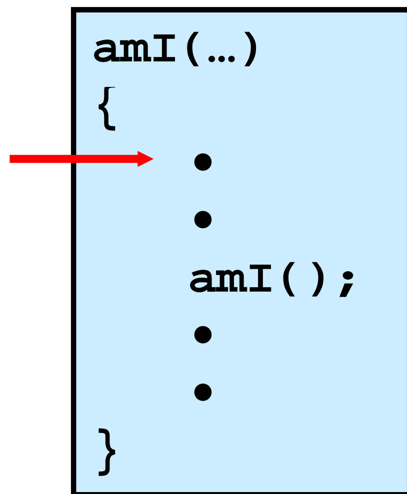


Call Chain

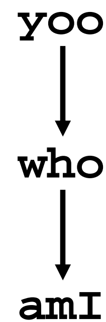


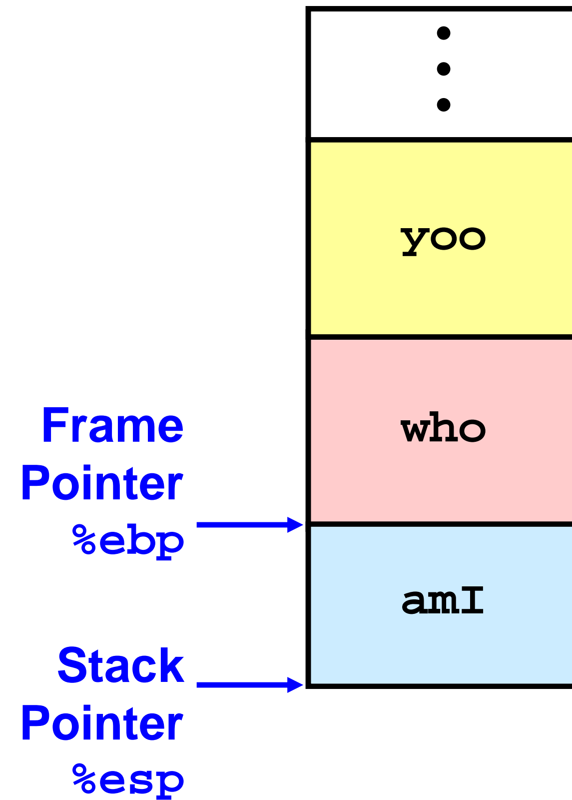


Stack Frames (5)

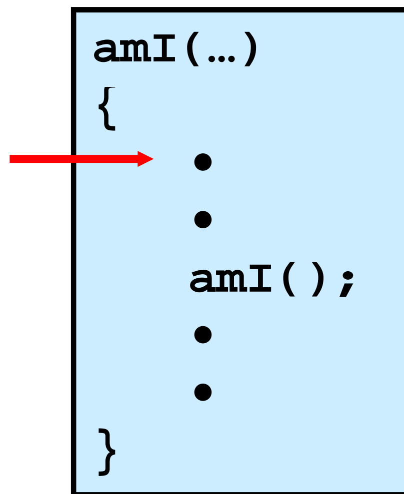


Call Chain

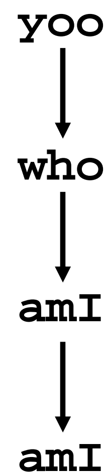


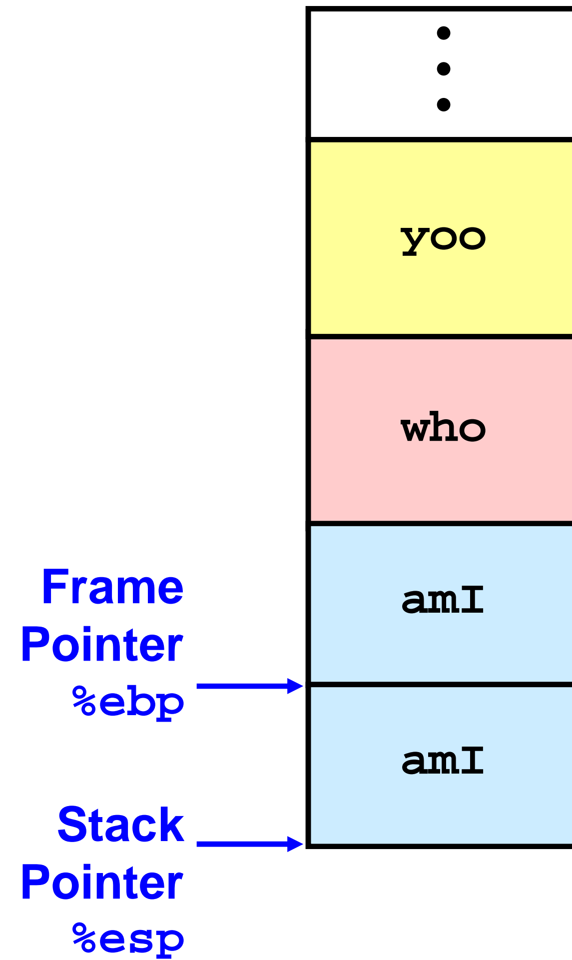


Stack Frames (6)

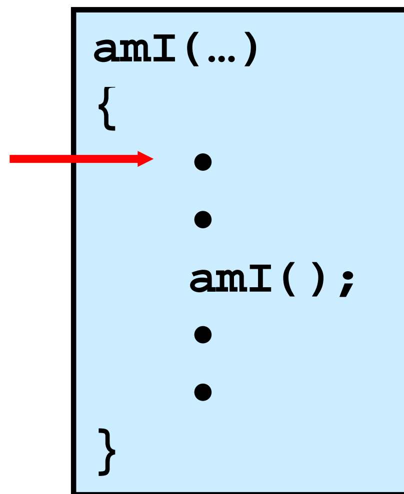


Call Chain

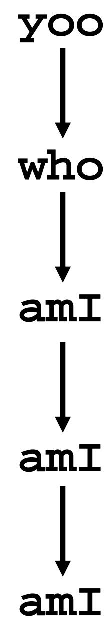


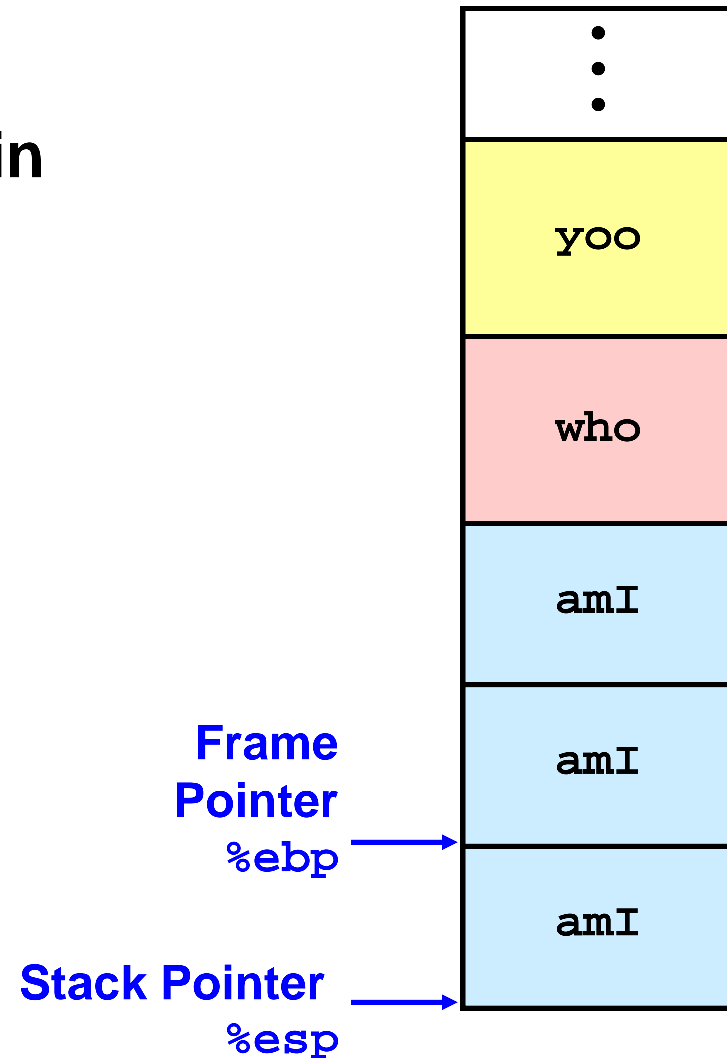


Stack Frames (7)

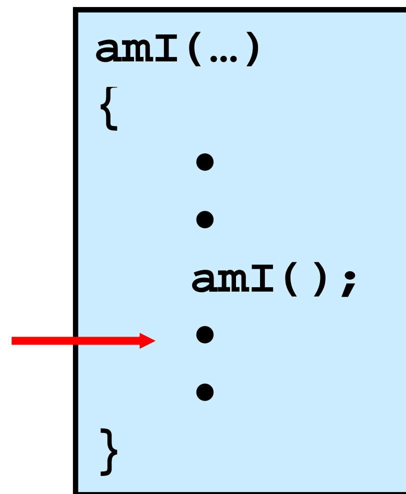


Call Chain

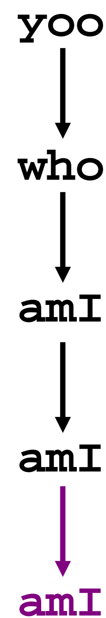


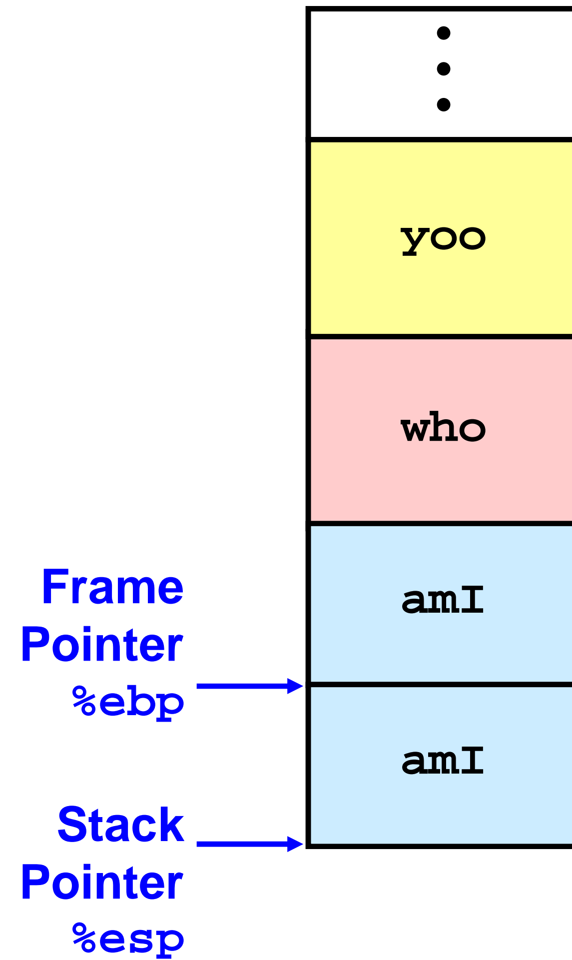


Stack Frames (8)

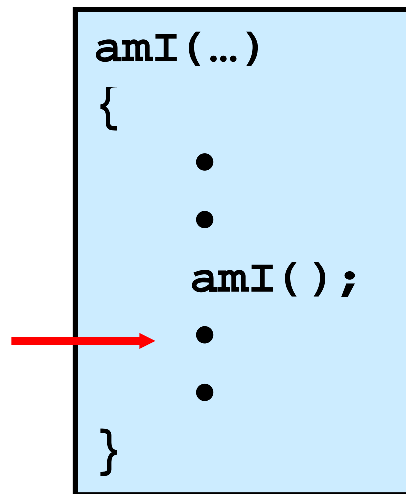


Call Chain

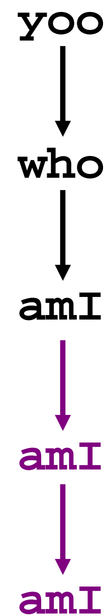


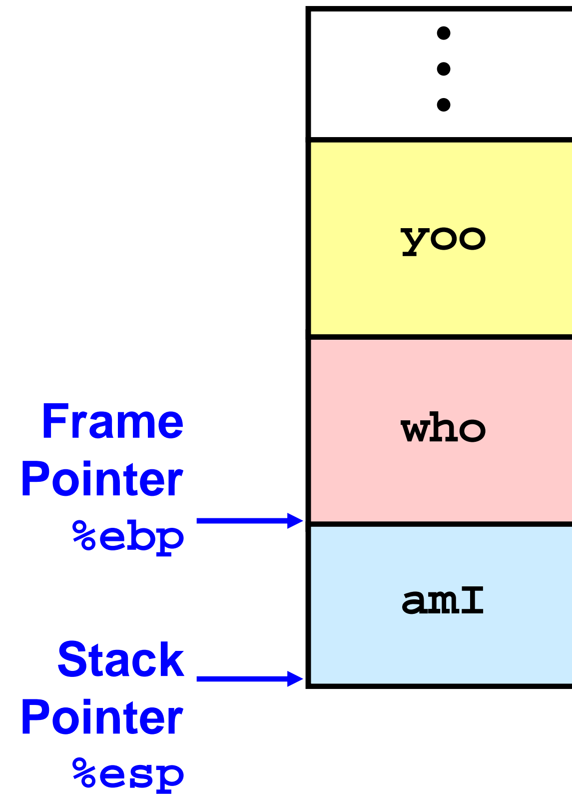


Stack Frames (9)

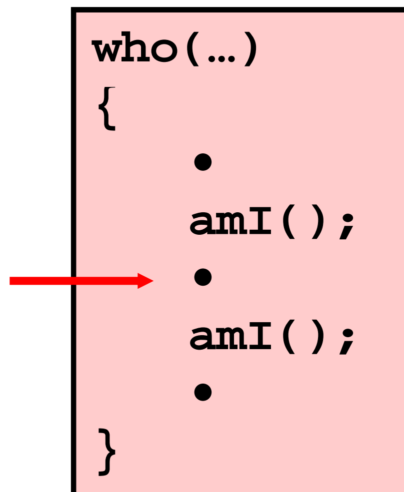


Call Chain

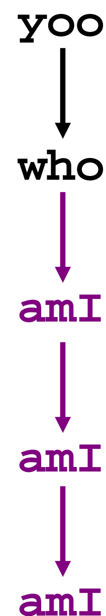


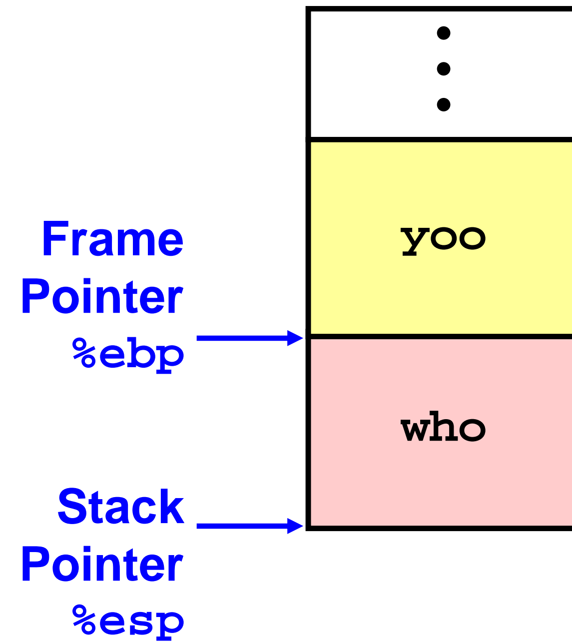


Stack Frames (10)

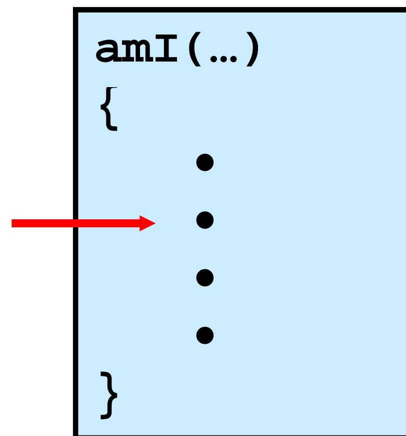


Call Chain

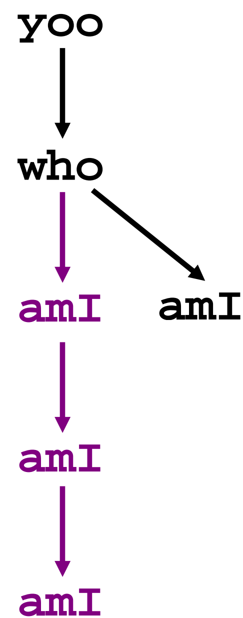


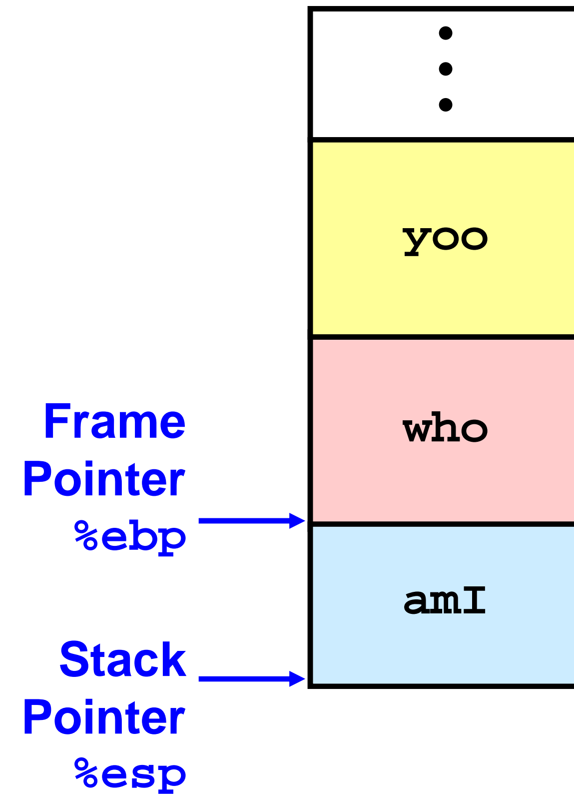


Stack Frames (11)

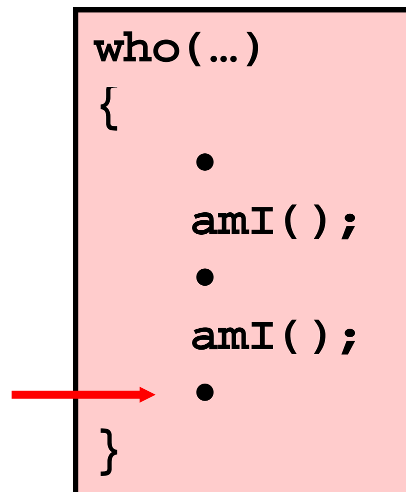


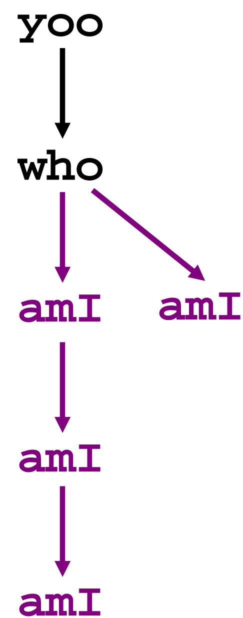
Call Chain

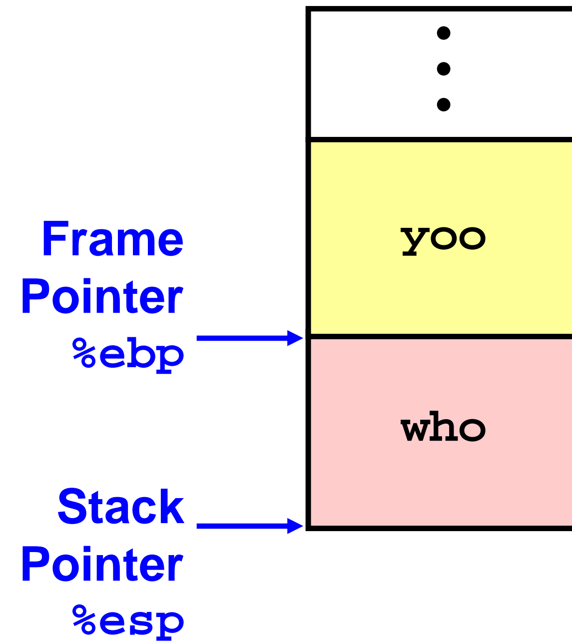




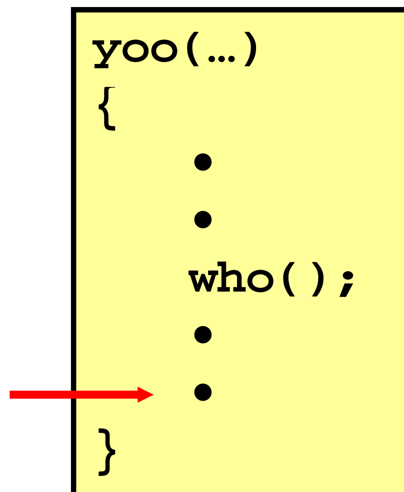
Stack Frames (12)



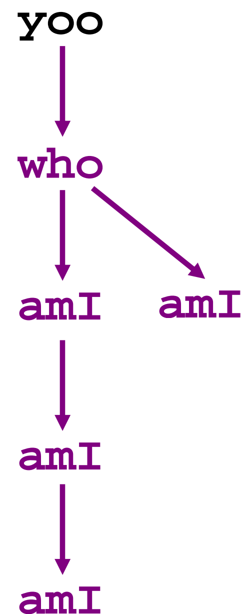


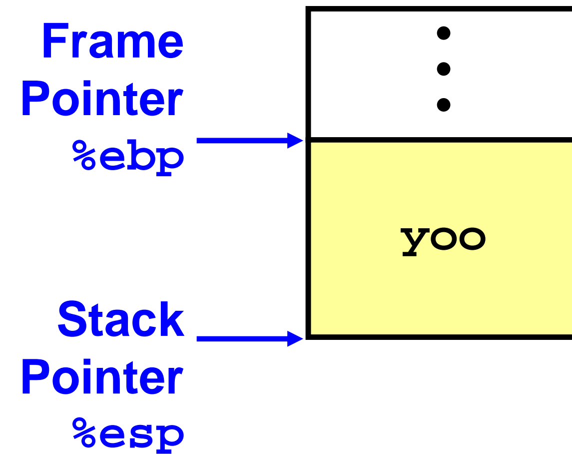


Stack Frames (13)



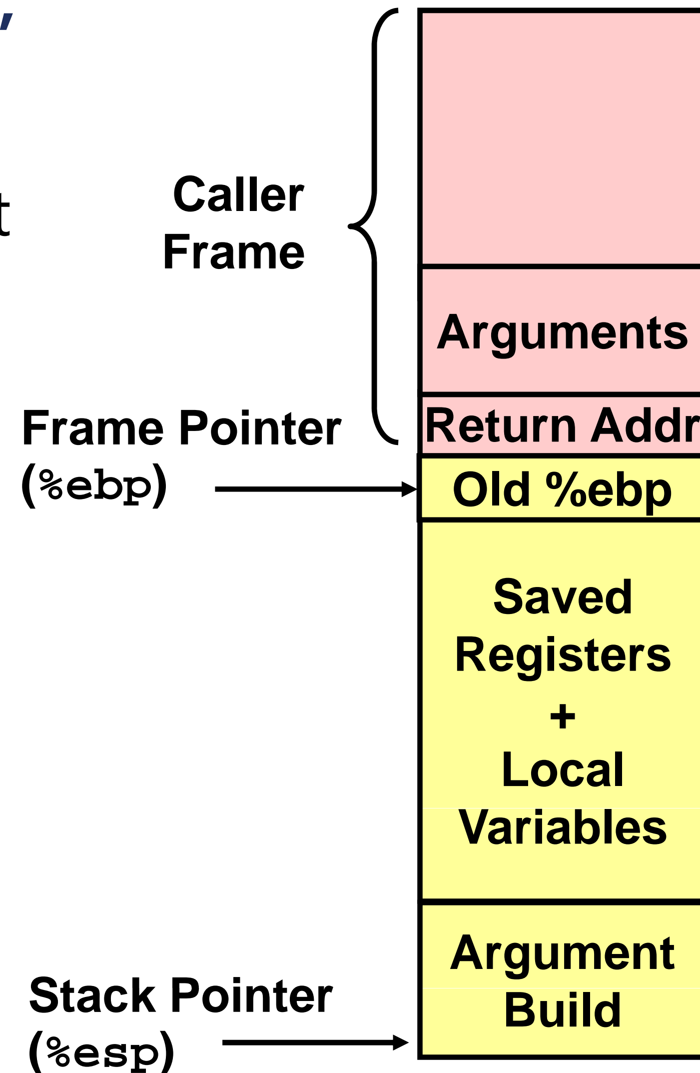
Call Chain





IA-32/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



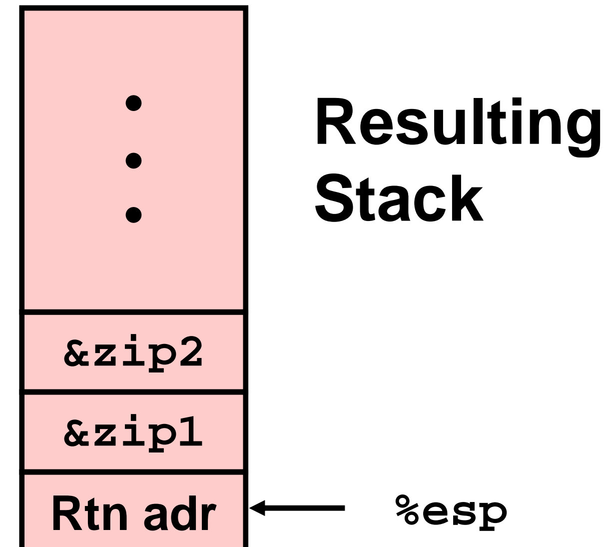
Revisiting swap (1)

```
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
    • • •
```



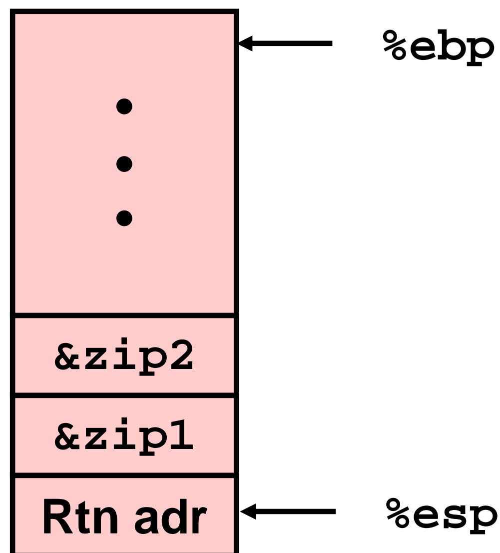
Revisiting swap (2)

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

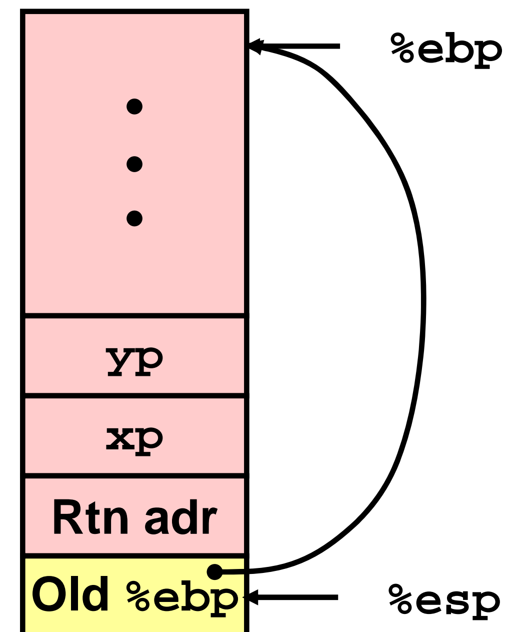
```
swap:
    pushl %ebp              } Setup
    movl %esp,%ebp
    pushl %ebx

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

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

Swap Setup (1)

Entering Stack



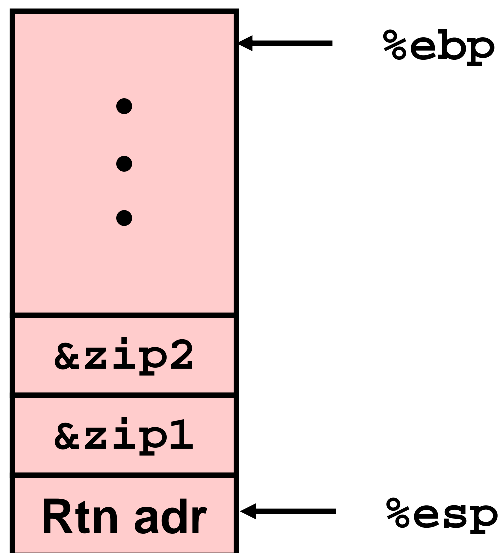
Resulting Stack




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

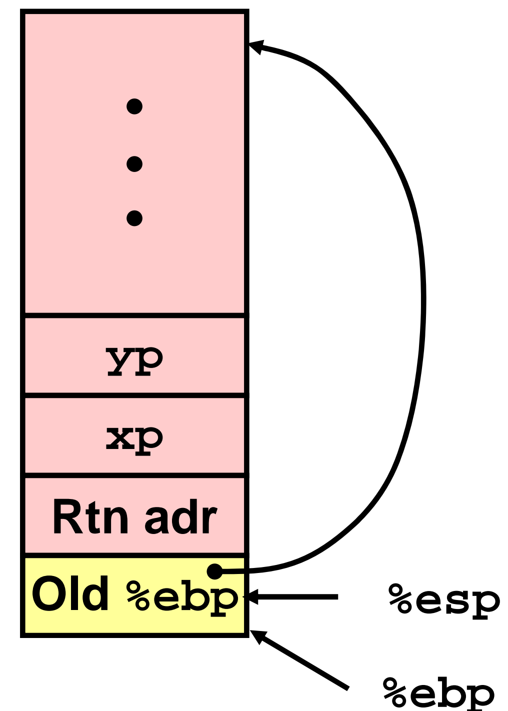
Swap Setup (2)

Entering Stack

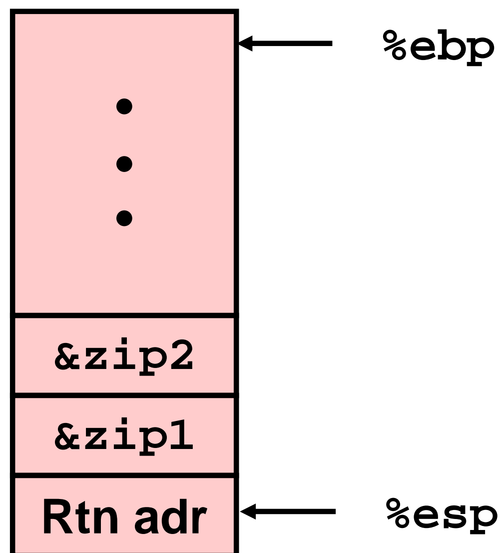


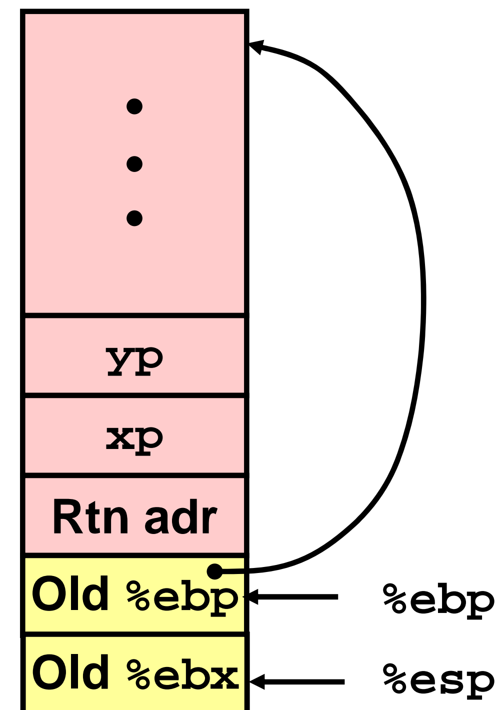

```
swap:
    pushl  %ebp
    movl  %esp,%ebp
    pushl  %ebx
```

Resulting Stack



Swap Setup (3)



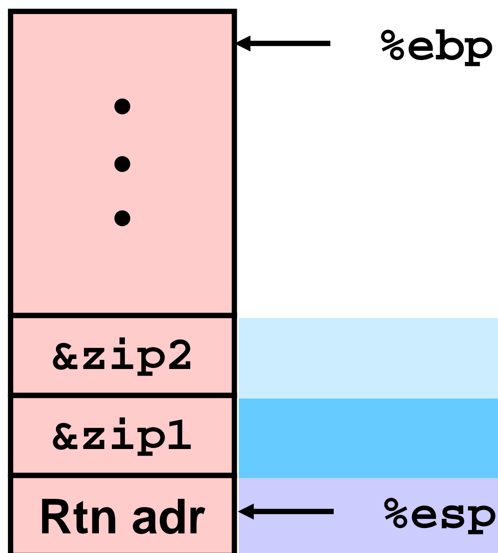


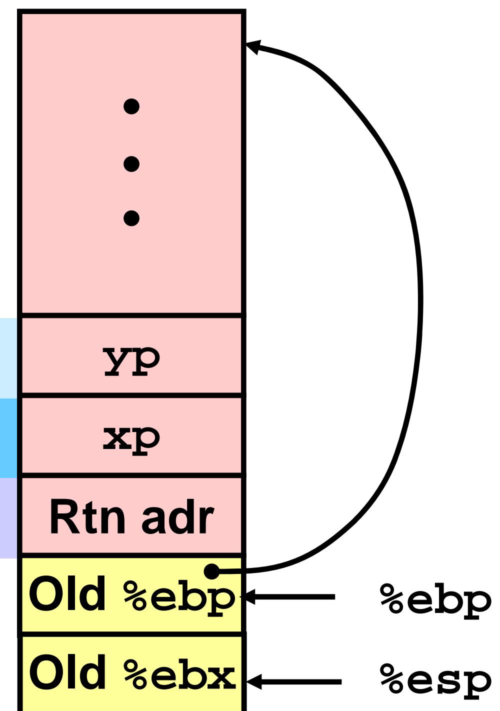

```
swap:
    pushl  %ebp
    movl  %esp,%ebp
    pushl  %ebx
```

Effect of swap Setup

Entering Stack

Resulting Stack



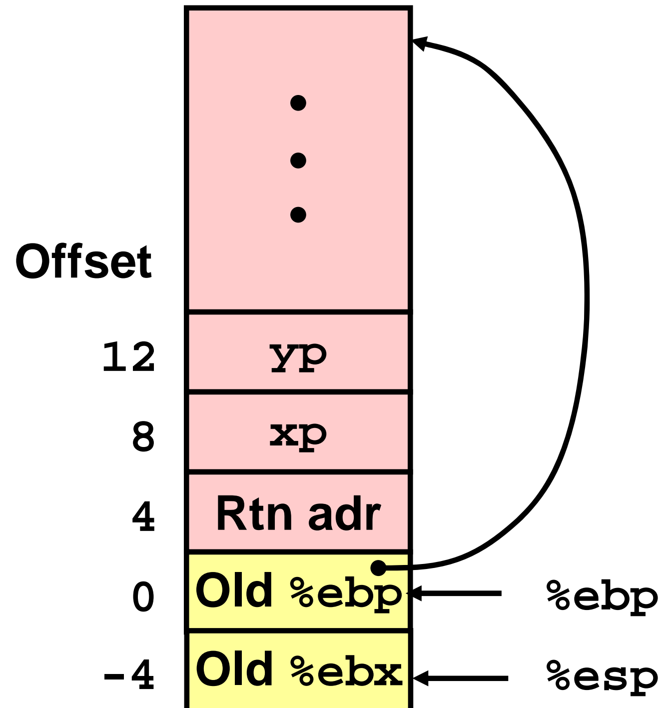


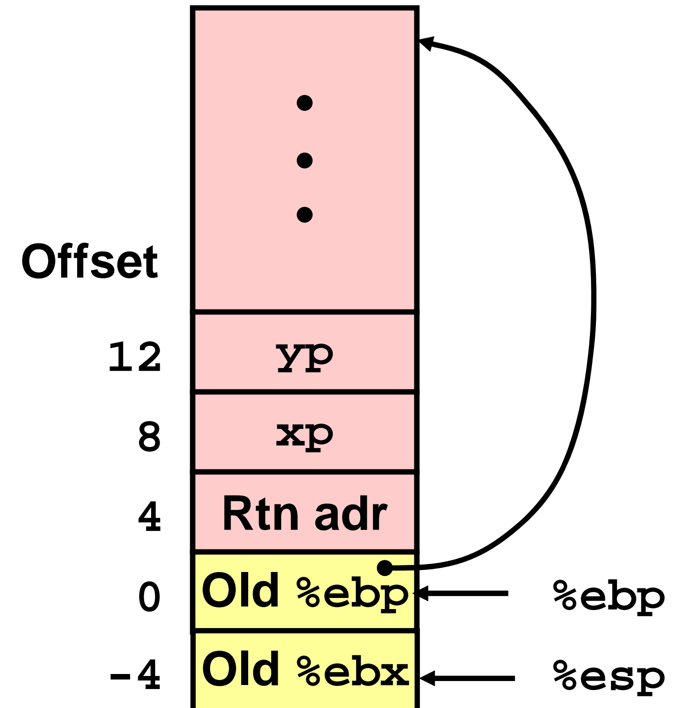

```
movl 12(%ebp),%ecx      # get yp
movl 8(%ebp),%edx       # get xp
    . . .
```

Body

swap Finish (1)

swap's **Stack**



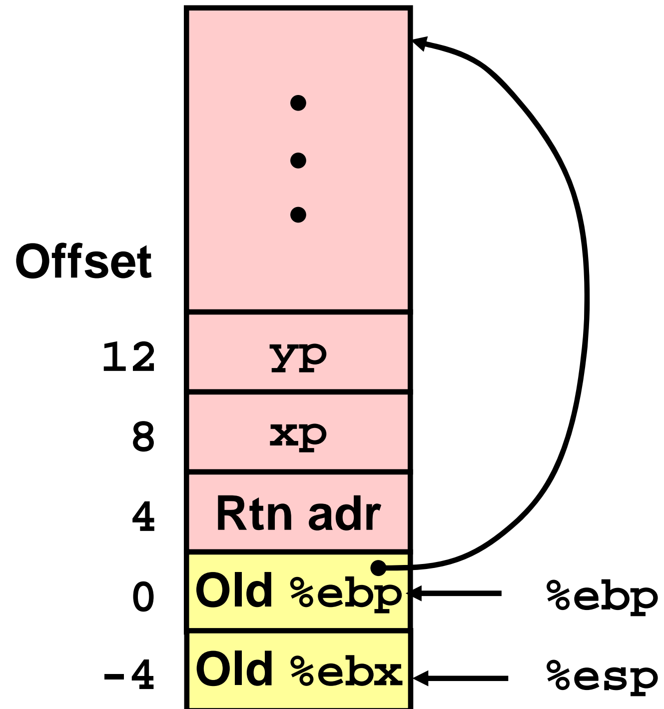


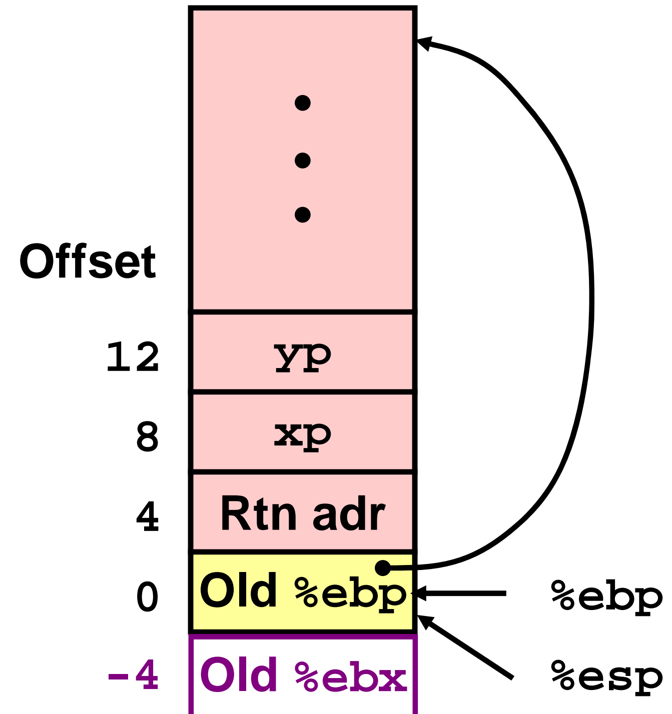

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

- **Observation**
  - Saved & restored register %ebx

swap Finish (2)

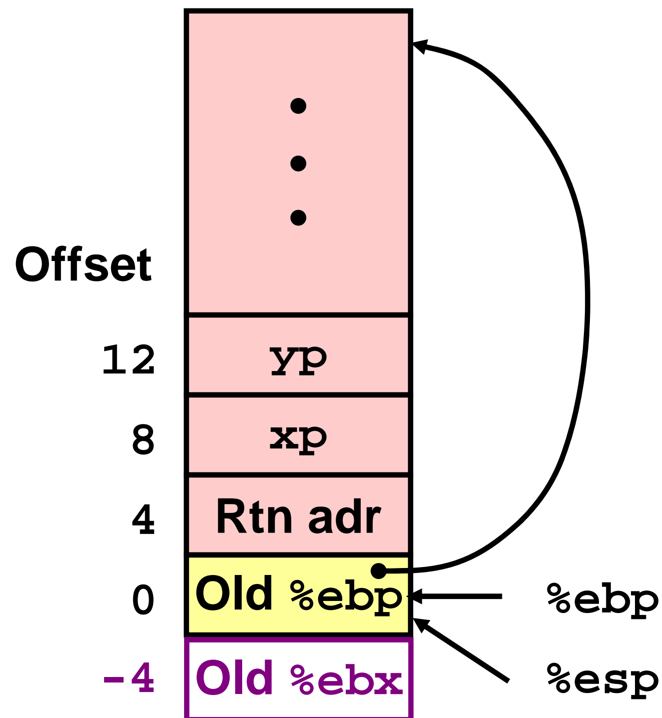
swap's Stack

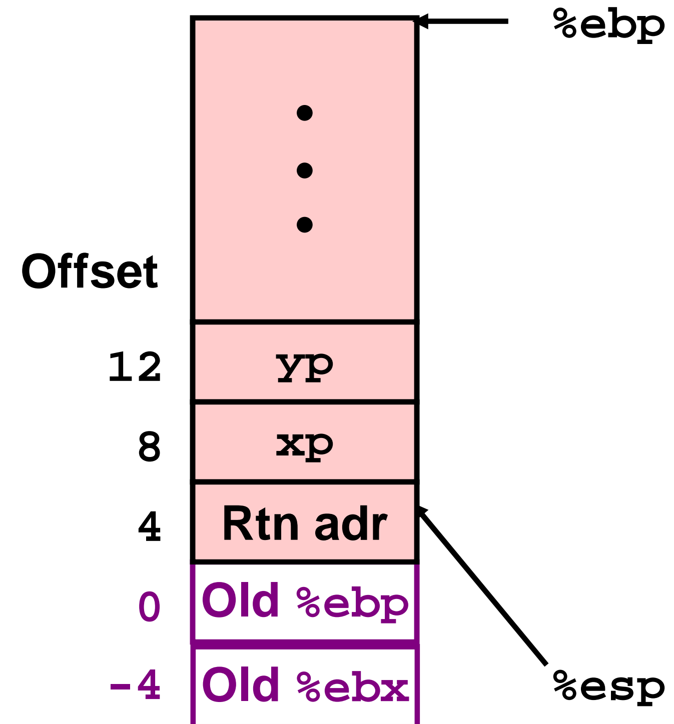





```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

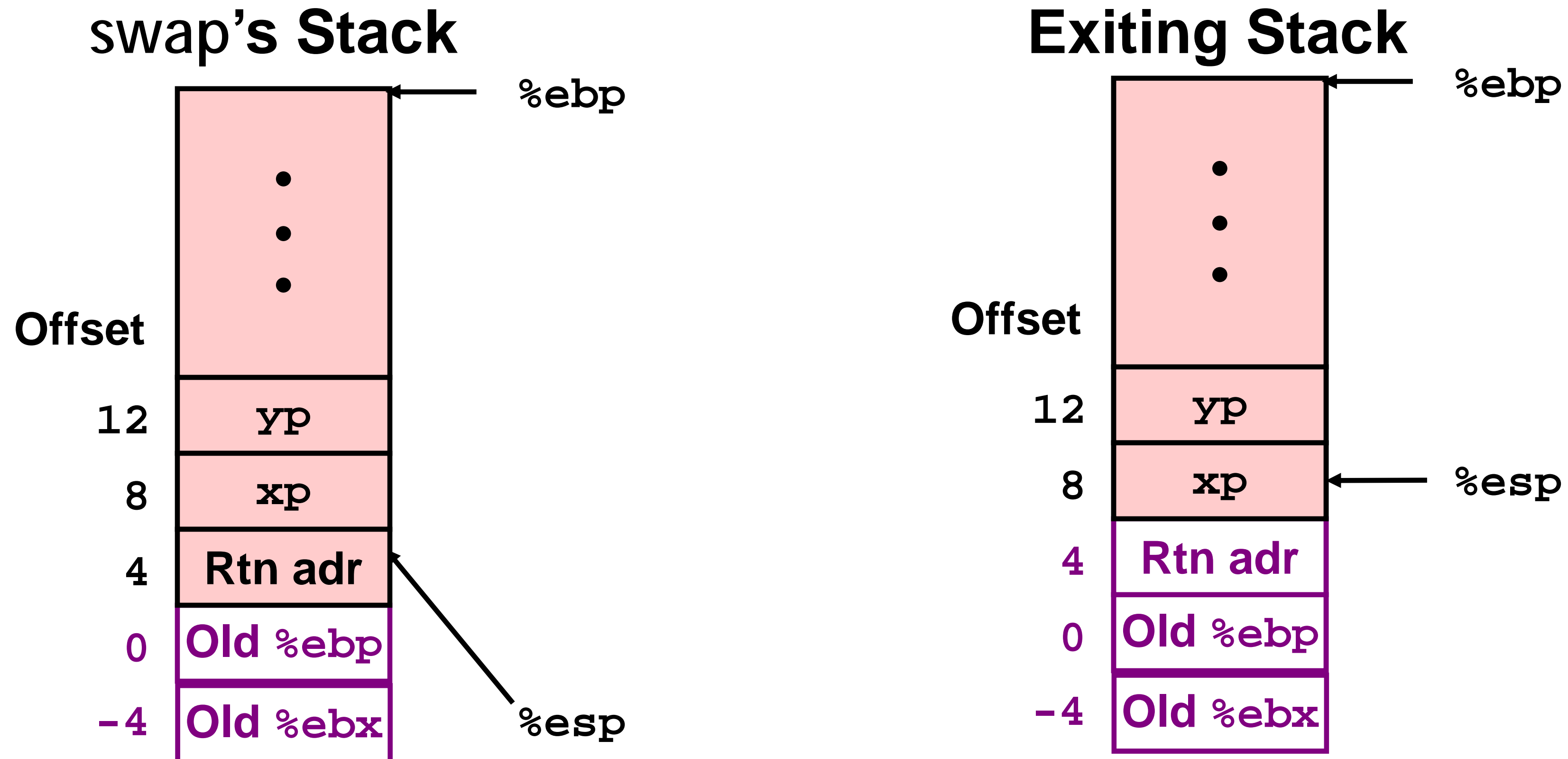
swap Finish (3)






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

swap Finish (4)




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

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

Register Saving Conventions (1)

- **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
```

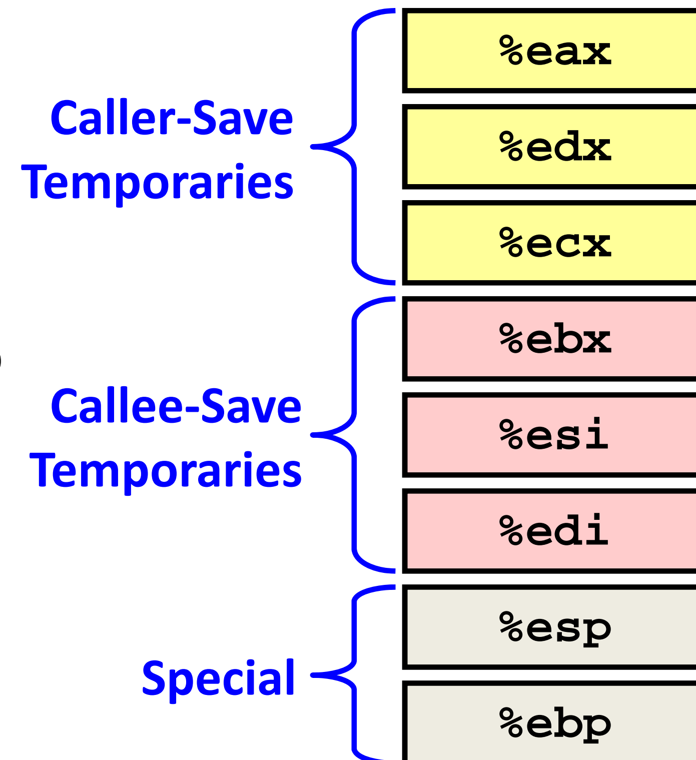
- Contents of register **%edx** overwritten by **who**

Register Saving Conventions (2)

- **Conventions**
  - "Caller save"
    - Caller saves temporary in its frame before calling
  - "Callee save"
    - Callee saves temporary in its frame before using

IA-32/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



Recursive Factorial: rfact

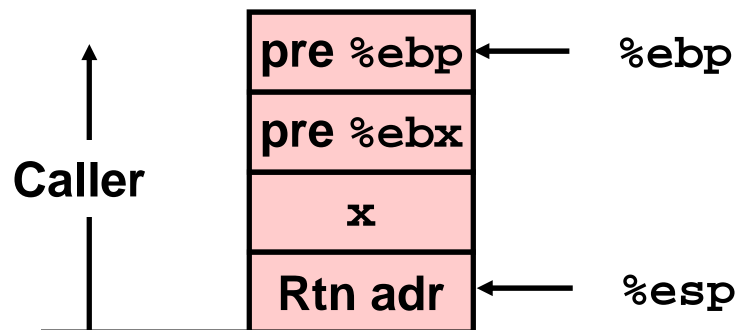
- **Registers**
  - %eax used without first saving
  - %ebx used, but save at beginning & restore at end

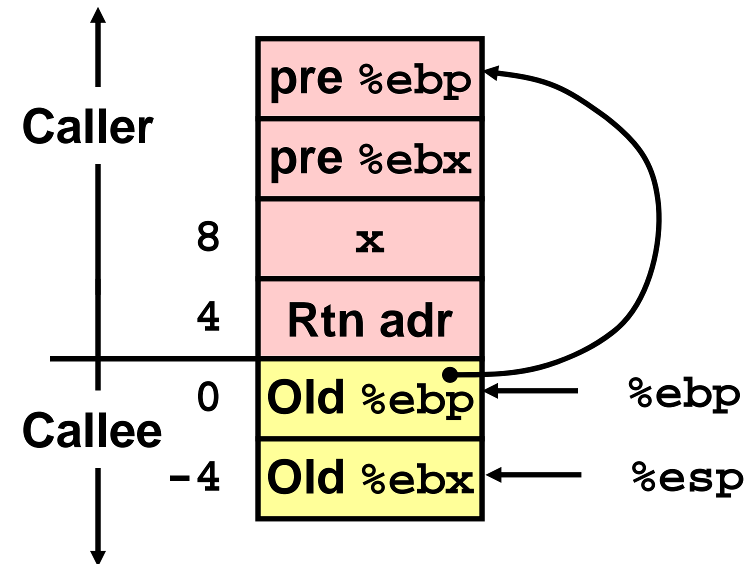
```
int rfact(int x)
{
  int rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1);
  return rval * x;
}
```

```
rfact:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 8(%ebp),%ebx
    cmpl $1,%ebx
    jle .L78
    leal -1(%ebx),%eax
    pushl %eax
    call rfact
    imull %ebx,%eax
    jmp .L79
    .align 4
.L78:
    movl $1,%eax
.L79:
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

rfact Stack Setup

Entering Stack






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

rfact Body

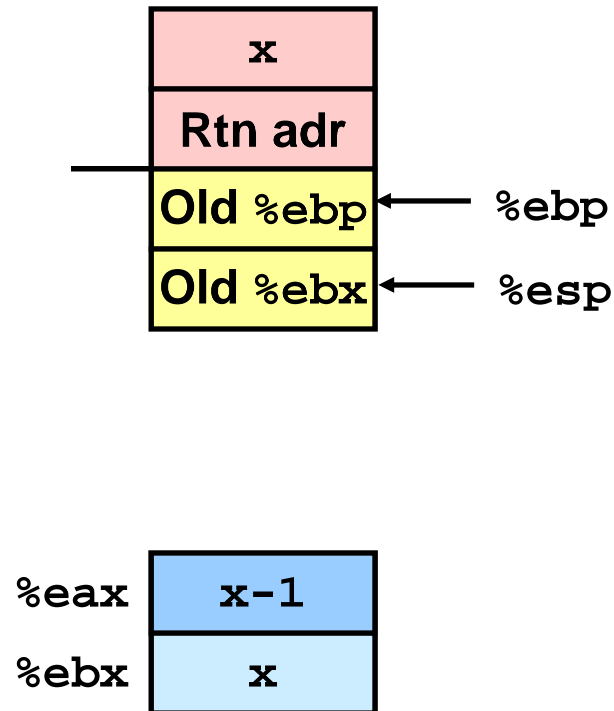
- **Registers**
  - %ebx: stored value of **x**
  - %eax
    - Temporary value of **x-1**
    - Returned value from **rfact(x-1)**
    - Returned value from this call

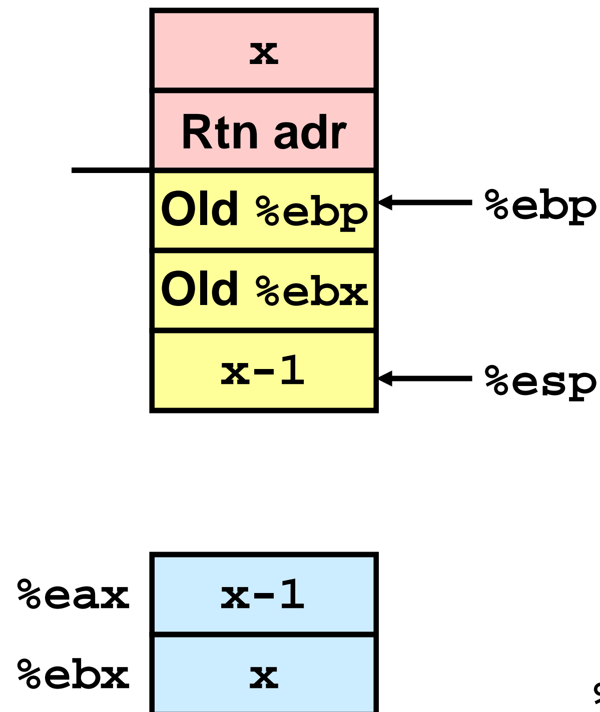
```
int rfact(int x)
{
  int rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1);
  return rval * x;
}
```

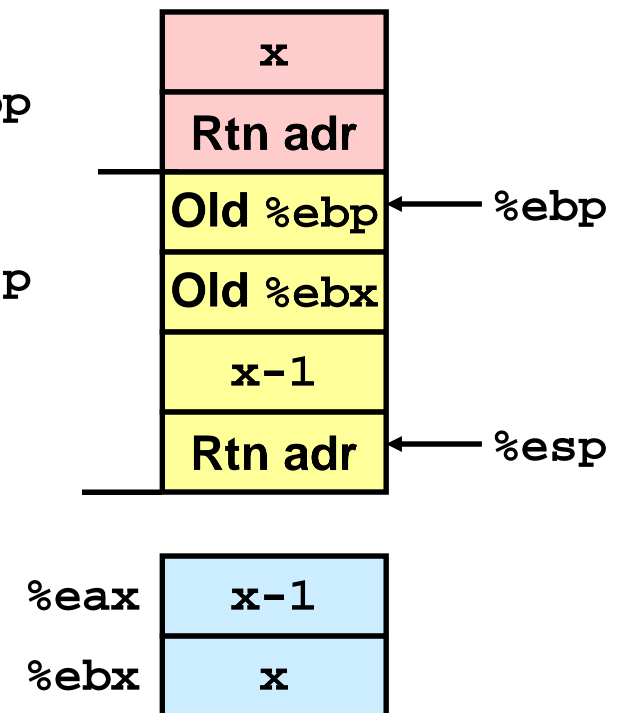
Recursion (bracketed lines: leal through imull)

```
  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:
```

rfact Recursion

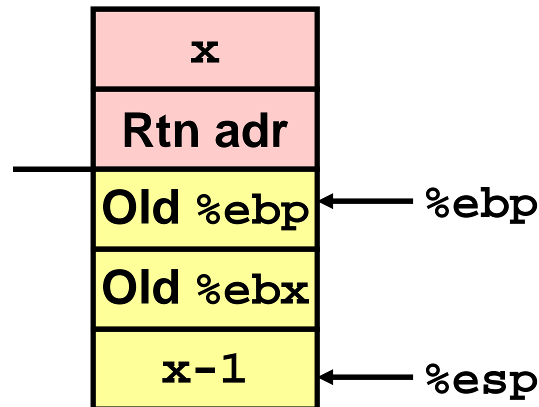


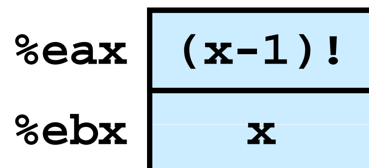




rfact Result

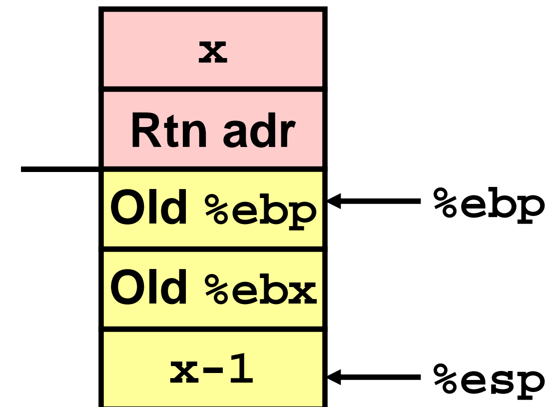
Return from Call

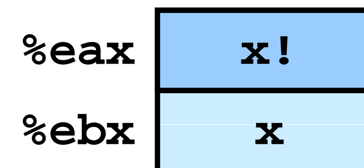




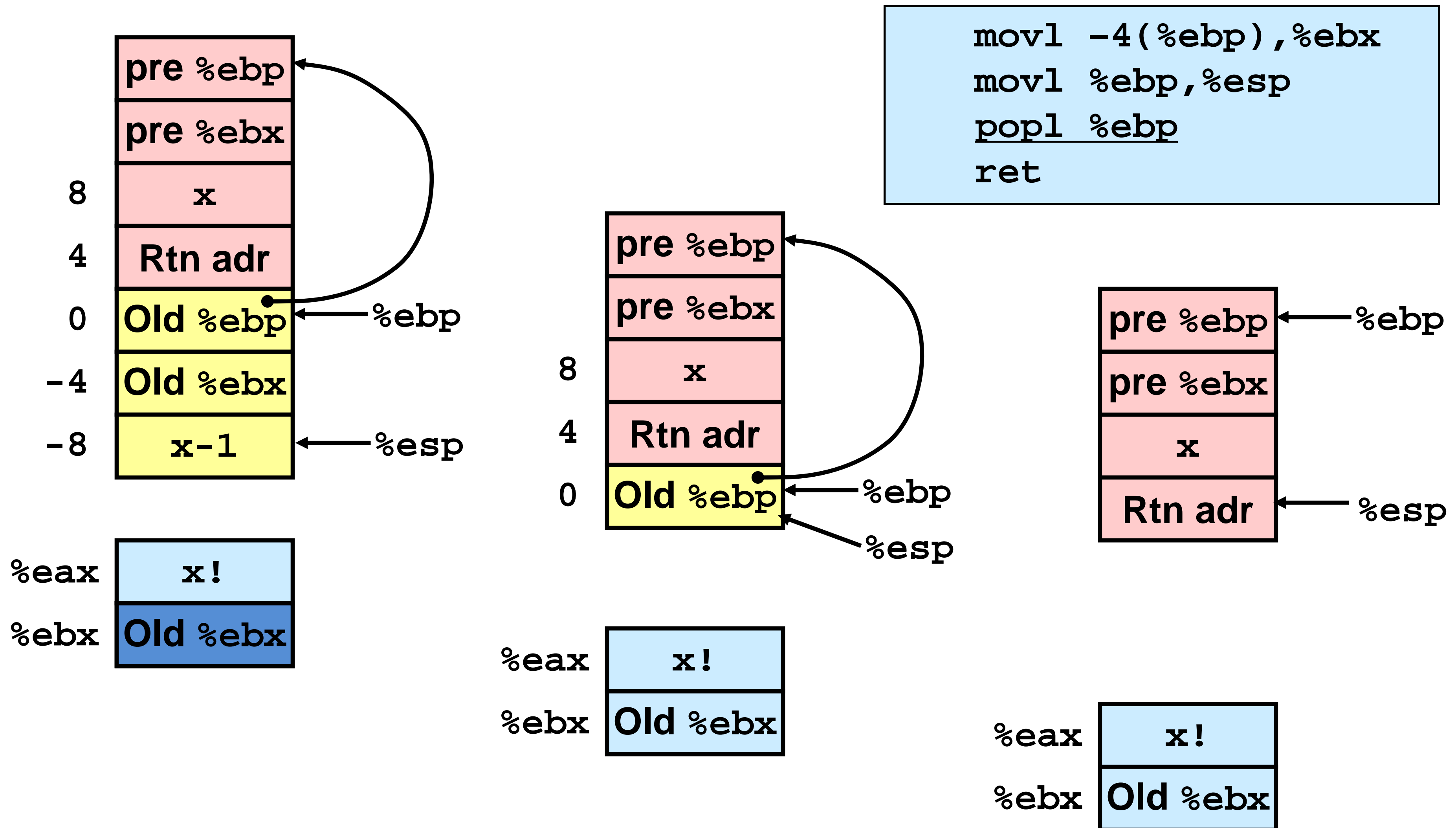
Assume that `rfact(x-1)` returns `(x-1)!` in register `%eax`

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





rfact Completion



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
- **Procedures = Instructions + Conventions**
  - Call / Ret instructions
  - Register usage conventions
    - Caller / Callee save
    - %ebp and %esp
  - Stack frame organization conventions