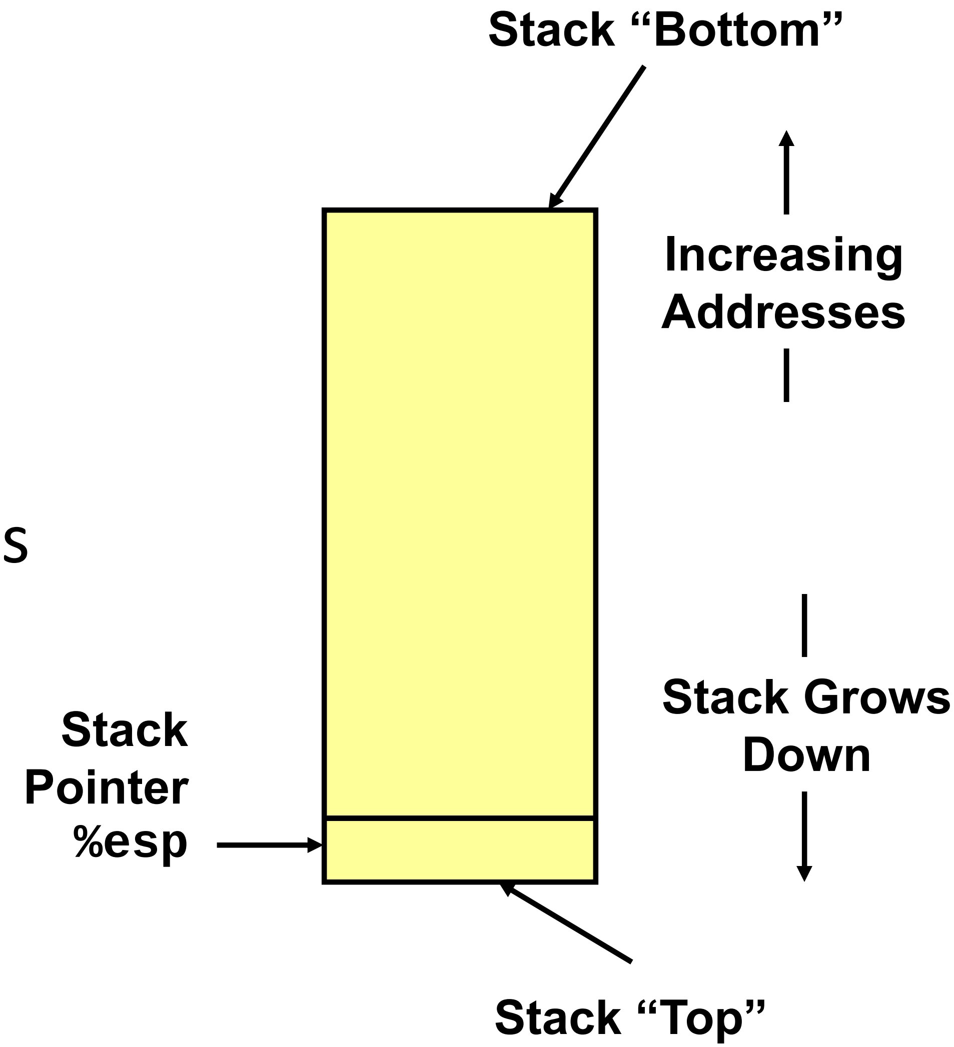
Machine-Level Programming: Procedures

Chapter 3 of B&O

Some notes adopted from Bryant and O'Hallaron

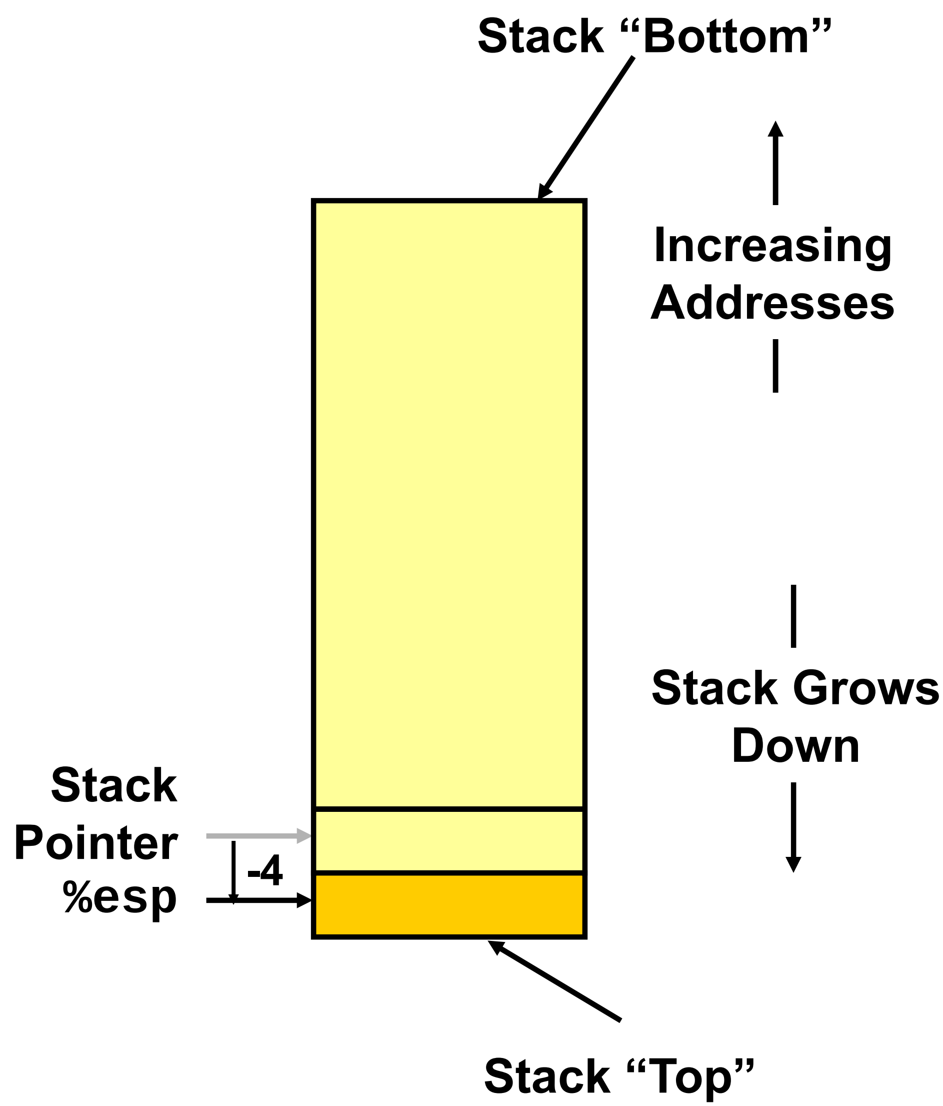
IA32 Stack

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



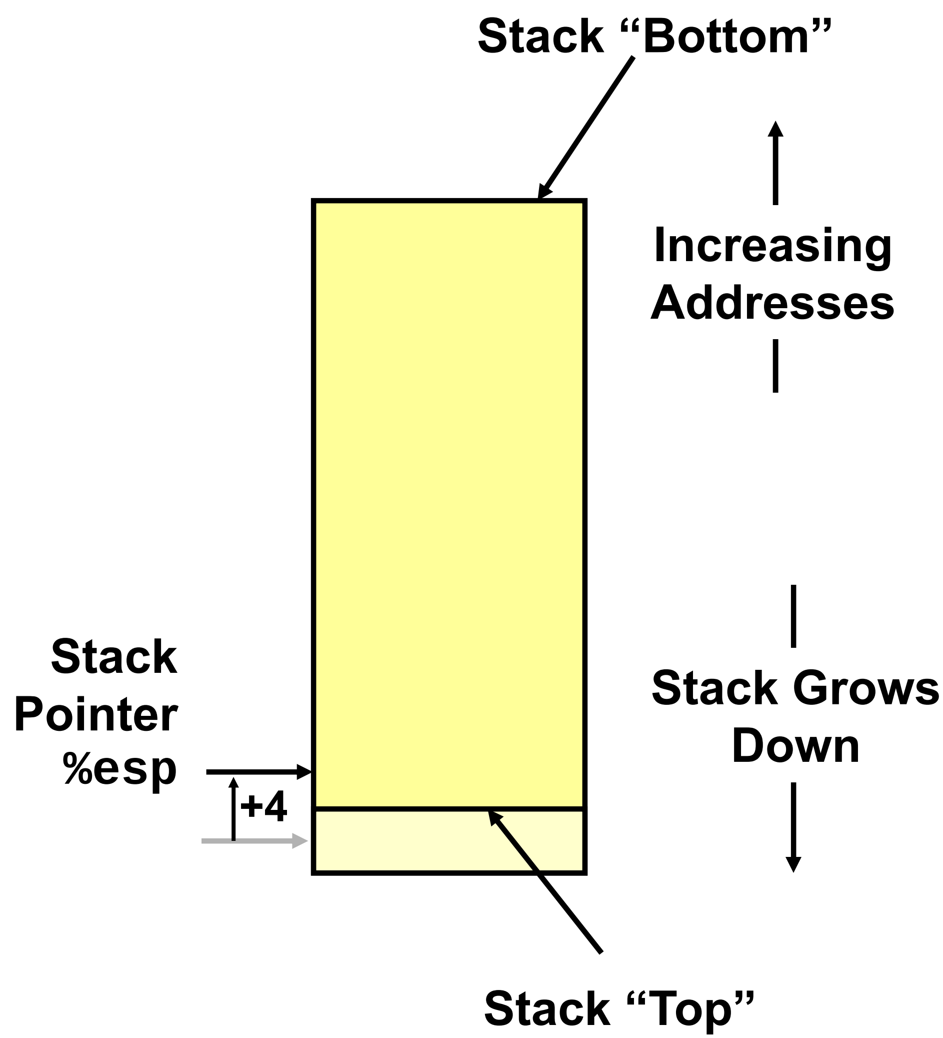
IA32 Stack Pushing

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



IA32 Stack Popping

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



Procedure Control Flow

- Use stack to support procedure call and return

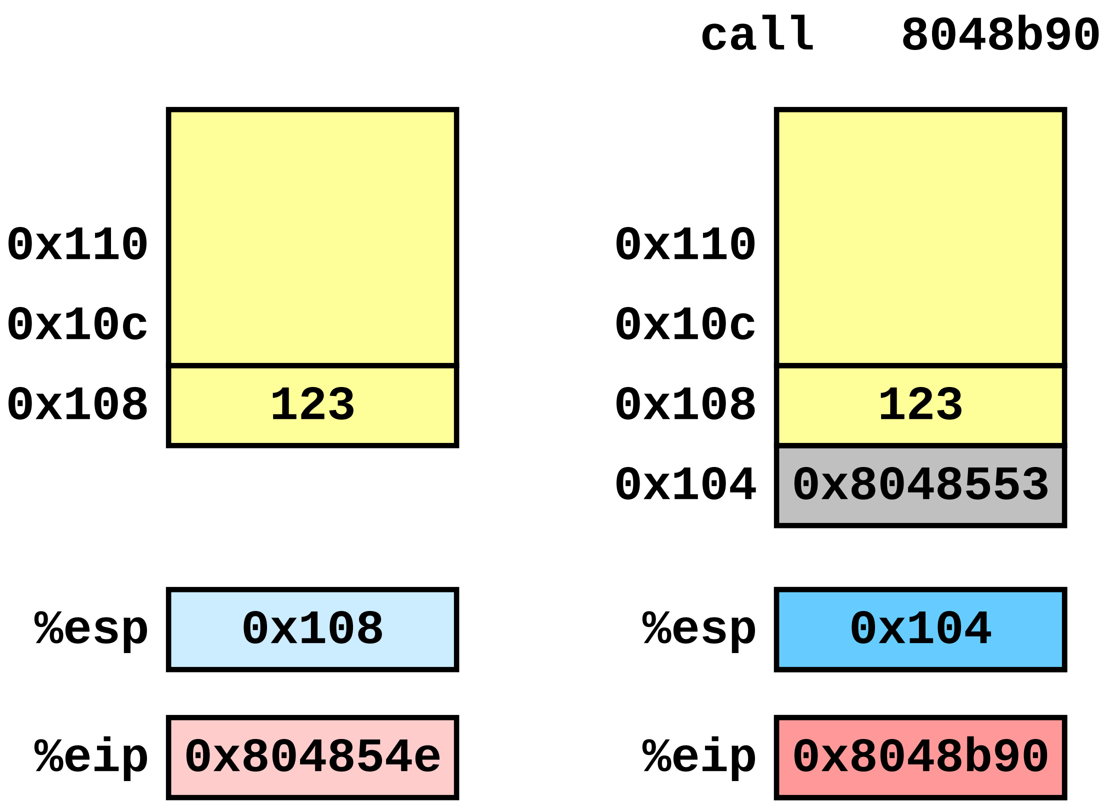
- Procedure call:
  - call *label*: Push return address on stack; Jump to *label*
- Return address value
  - Address of instruction beyond call
  - Example from disassembly

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 Example

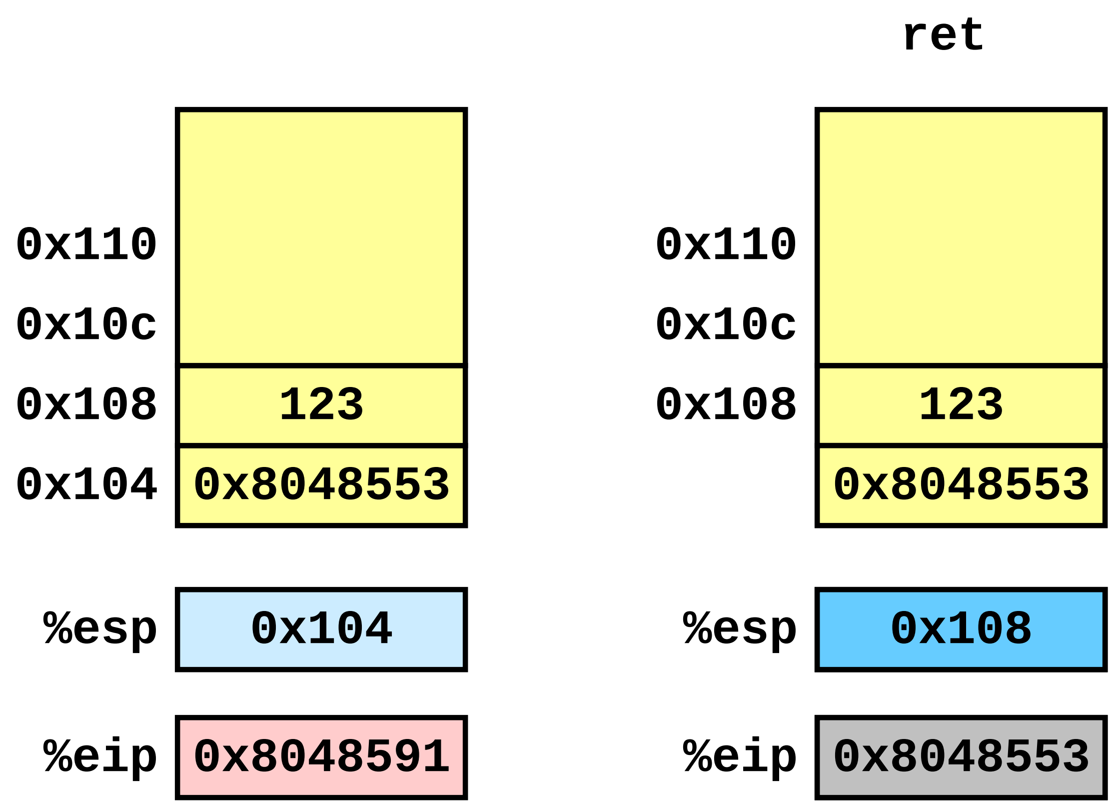
```
804854e:   e8 3d 06 00 00       call    8048b90 <main>
8048553:   50                   pushl   %eax
```



%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
  - 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

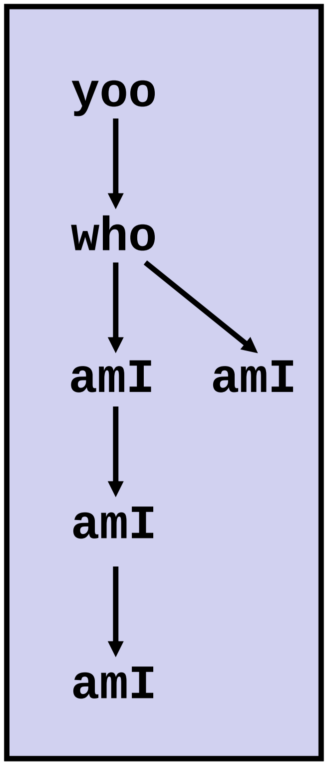
```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

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

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

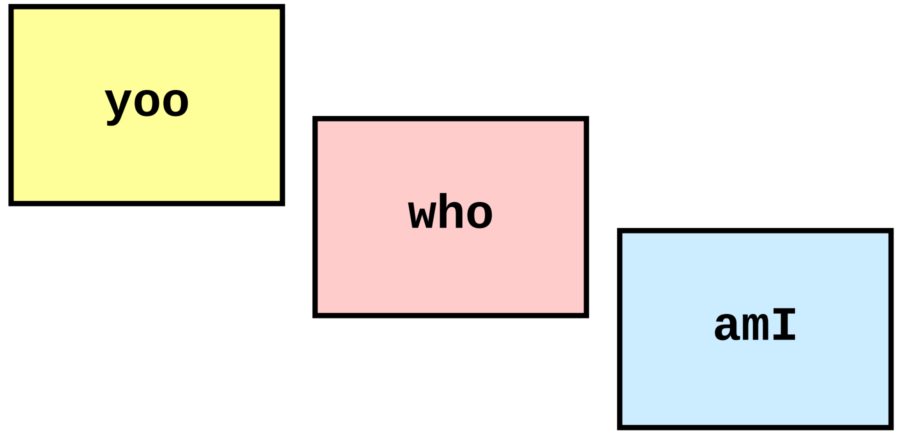
– Procedure amI recursive

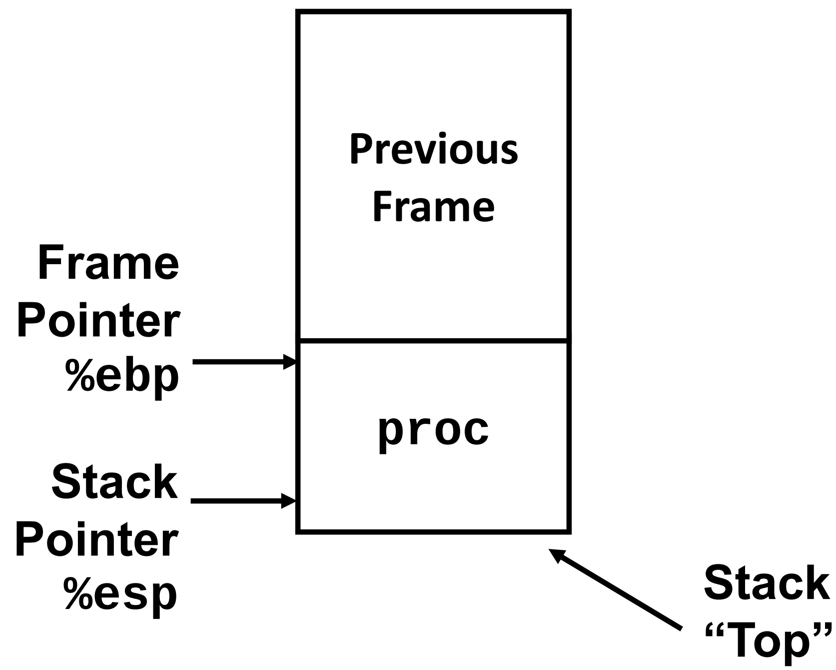
Call Chain



Stack Frames

- Contents
  - Local variables
  - Return information
  - Temporary space
- Management
  - Space allocated when enter procedure
    - "Set-up" code
  - Deallocated when return
    - "Finish" code
- Pointers
  - Stack pointer `%esp` indicates stack top
  - Frame pointer `%ebp` indicates start of current frame

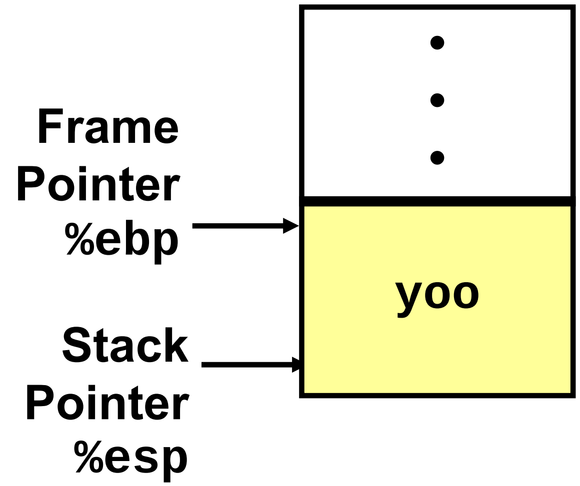




```
yoo(…)
{
  •
  •
  who();
  •
  •
}
```

Call Chain

yoo

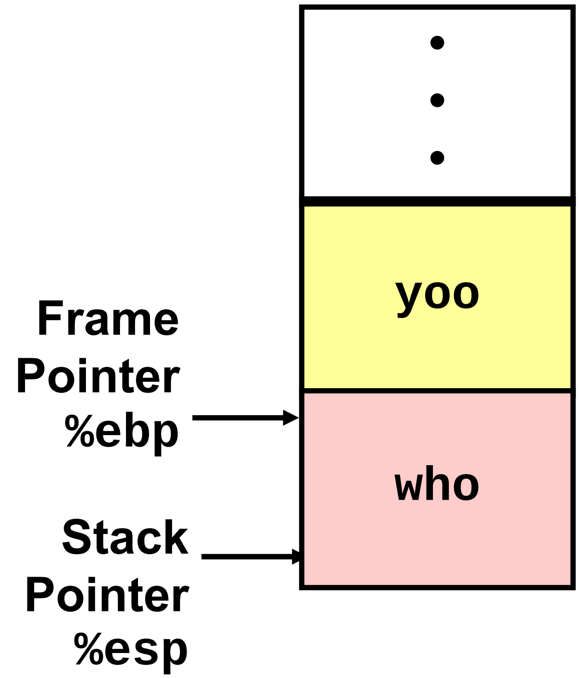


```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

Call Chain

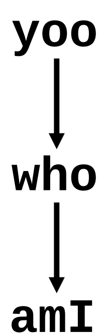
yoo

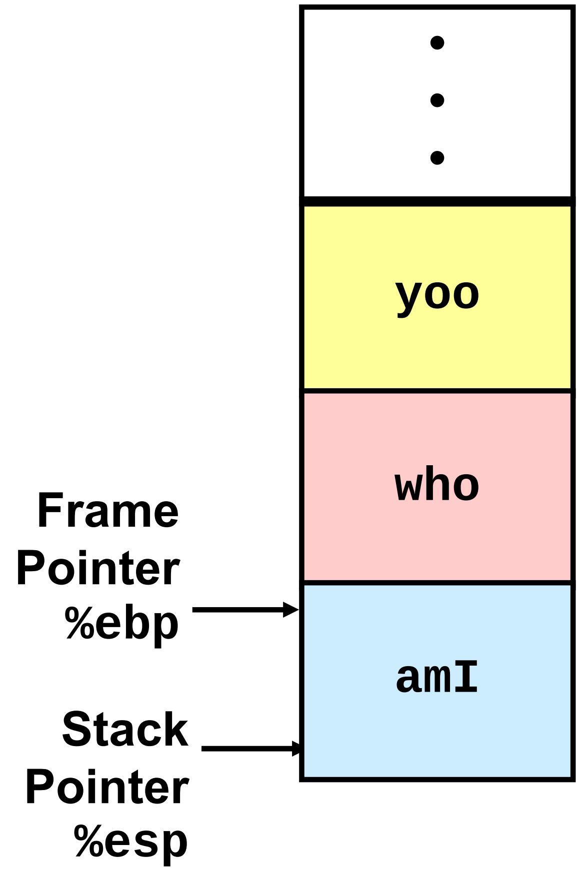
who



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

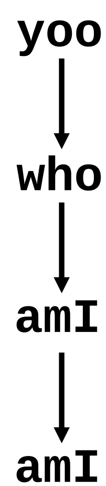
Call Chain

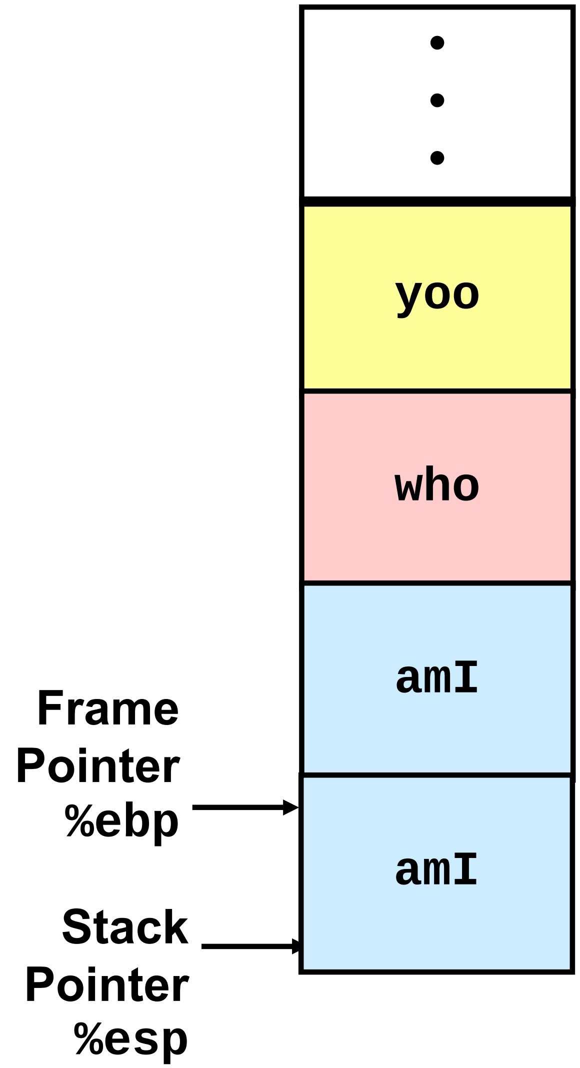




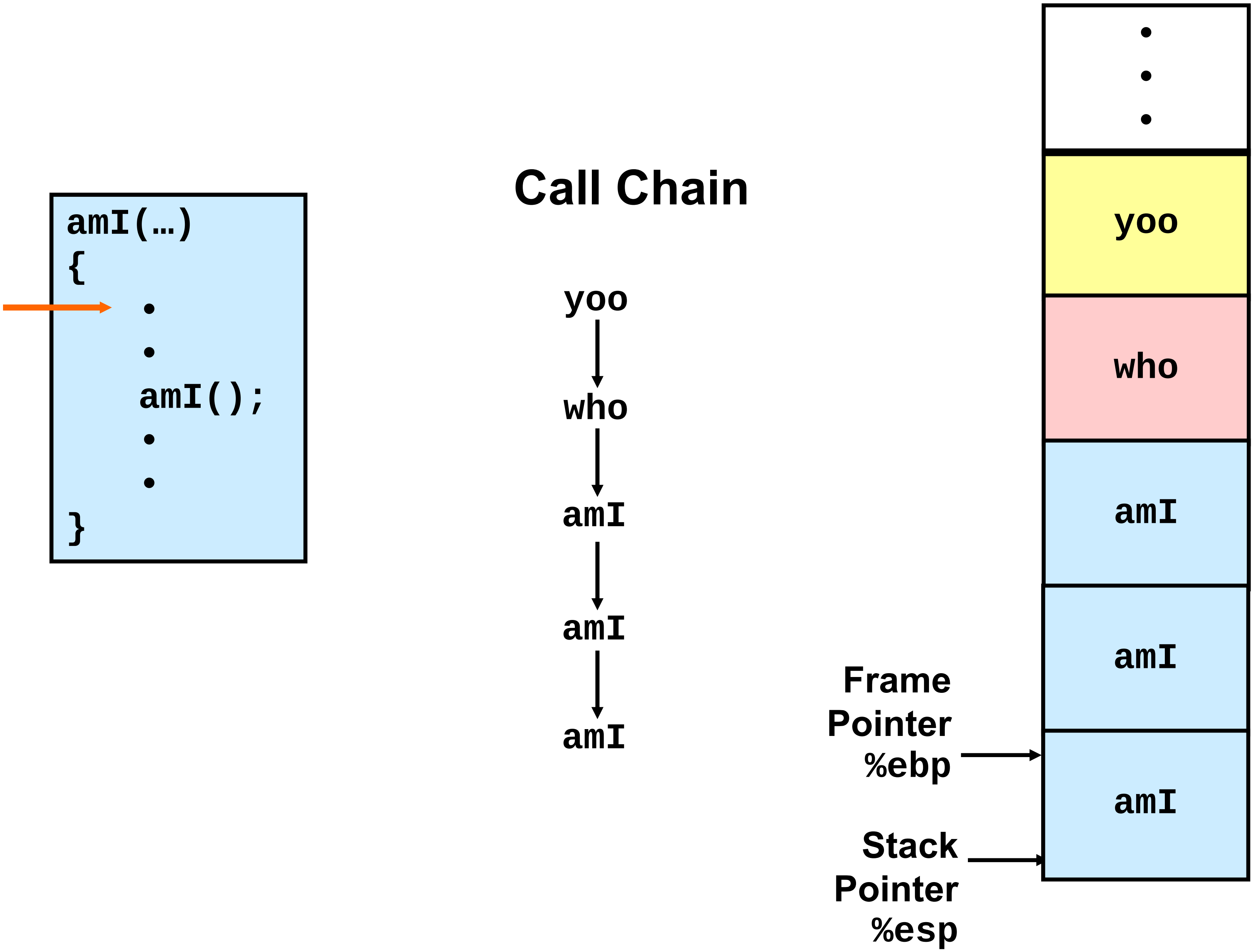
Call Chain

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



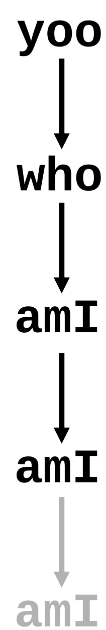


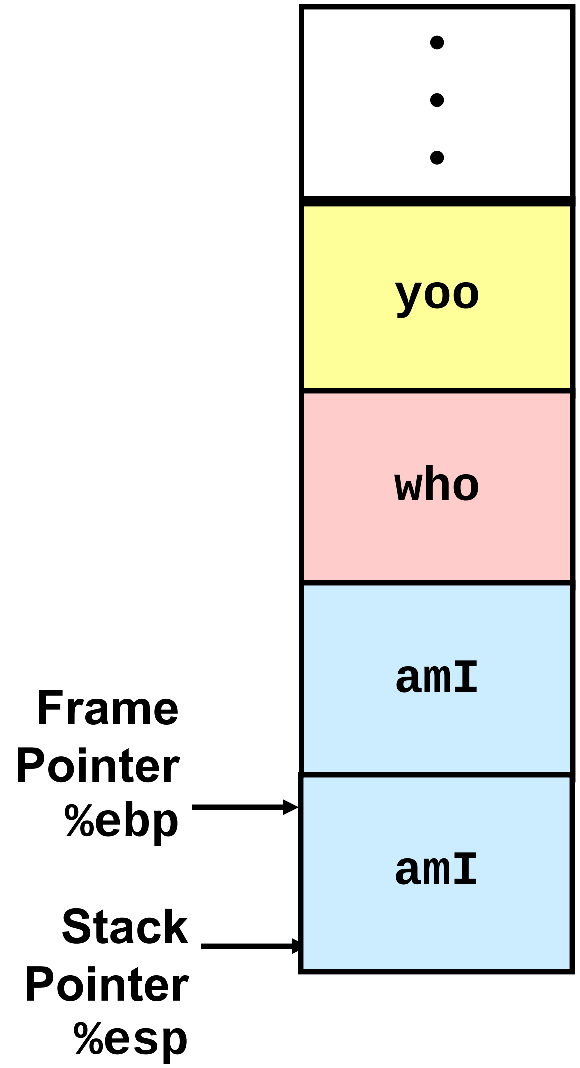
Call Chain


amI(…)
{
amI();
}
yoo
who
amI
amI
amI
yoo
who
amI
amI
amI
Frame Pointer %ebp
Stack Pointer %esp

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

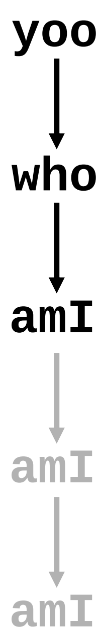
Call Chain

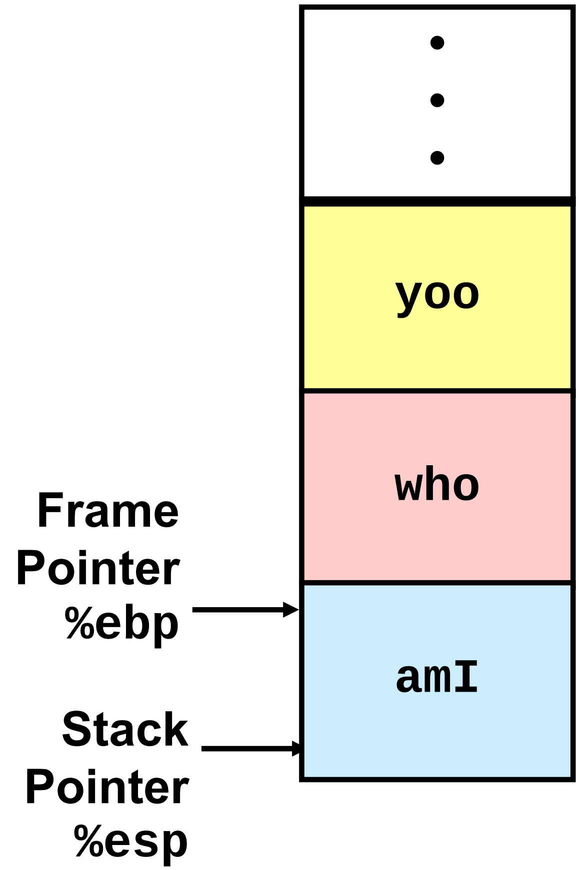




Call Chain

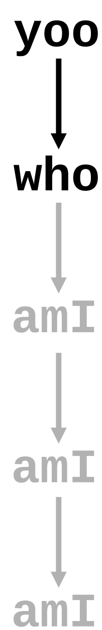
```
amI(…)
{
    •
    •
    amI();
    •
    •
}
```

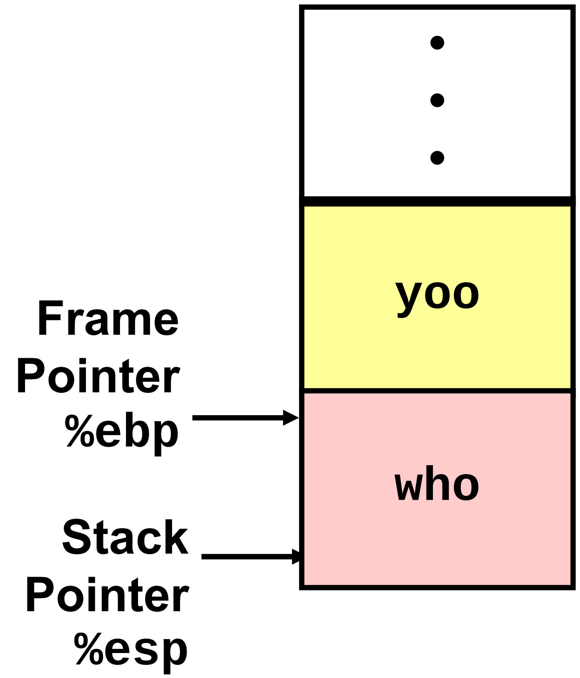


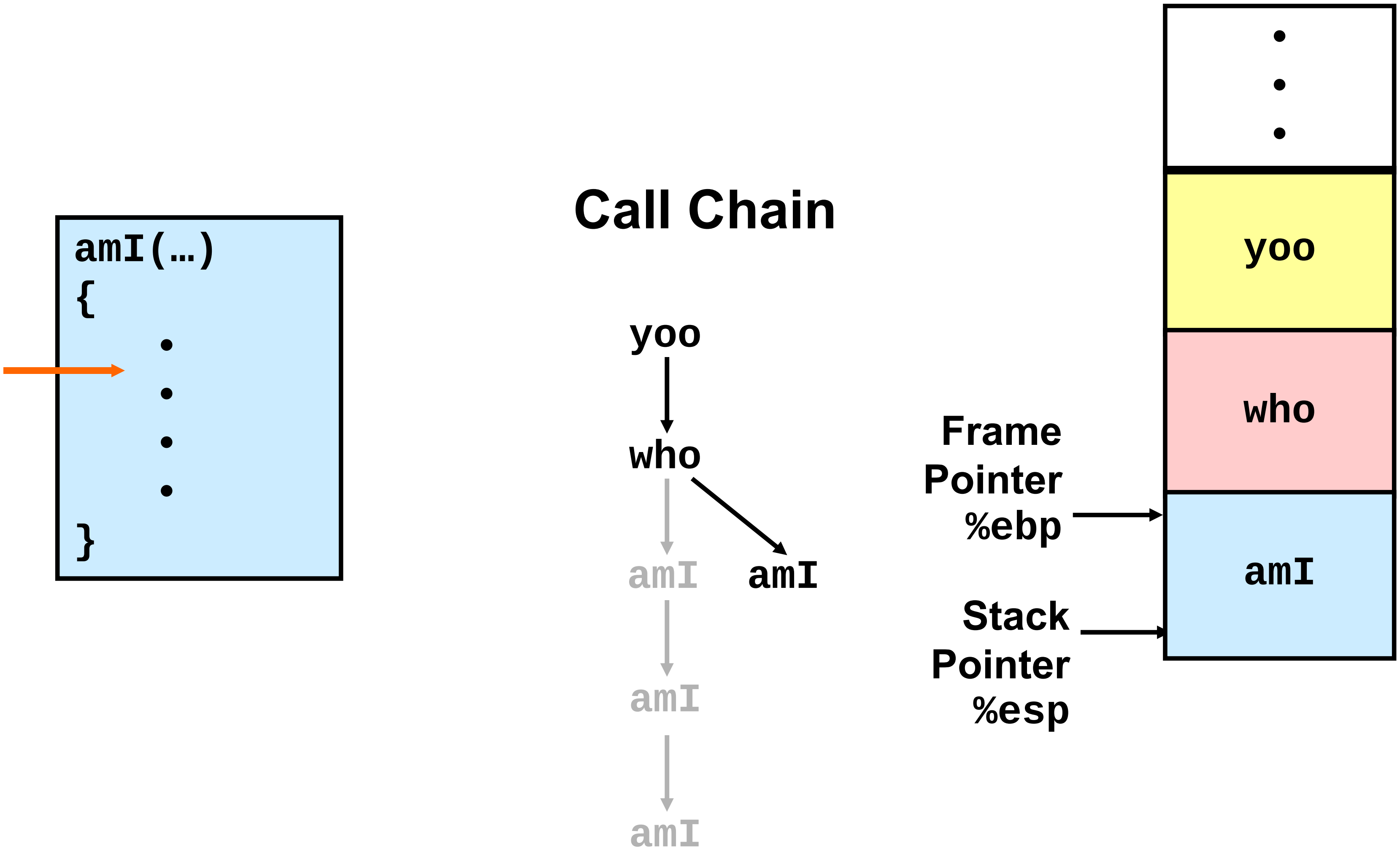


```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

Call Chain

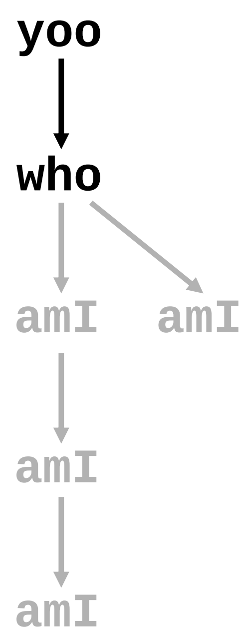


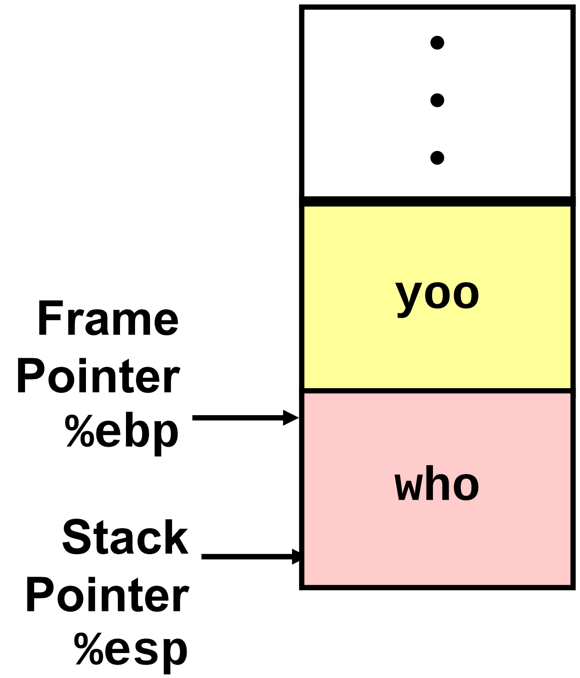



amI(…)
{
}
Call Chain
yoo
who
amI
amI
amI
amI
yoo
who
amI
Frame Pointer %ebp
Stack Pointer %esp

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

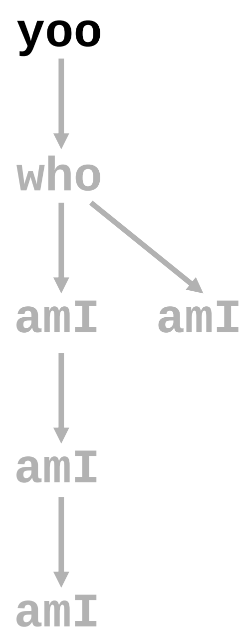
Call Chain

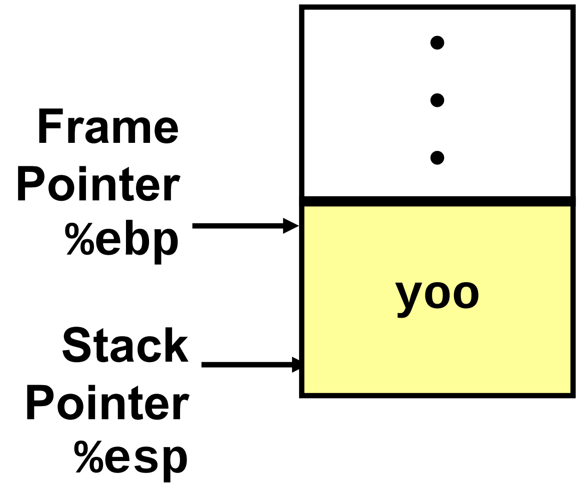




```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

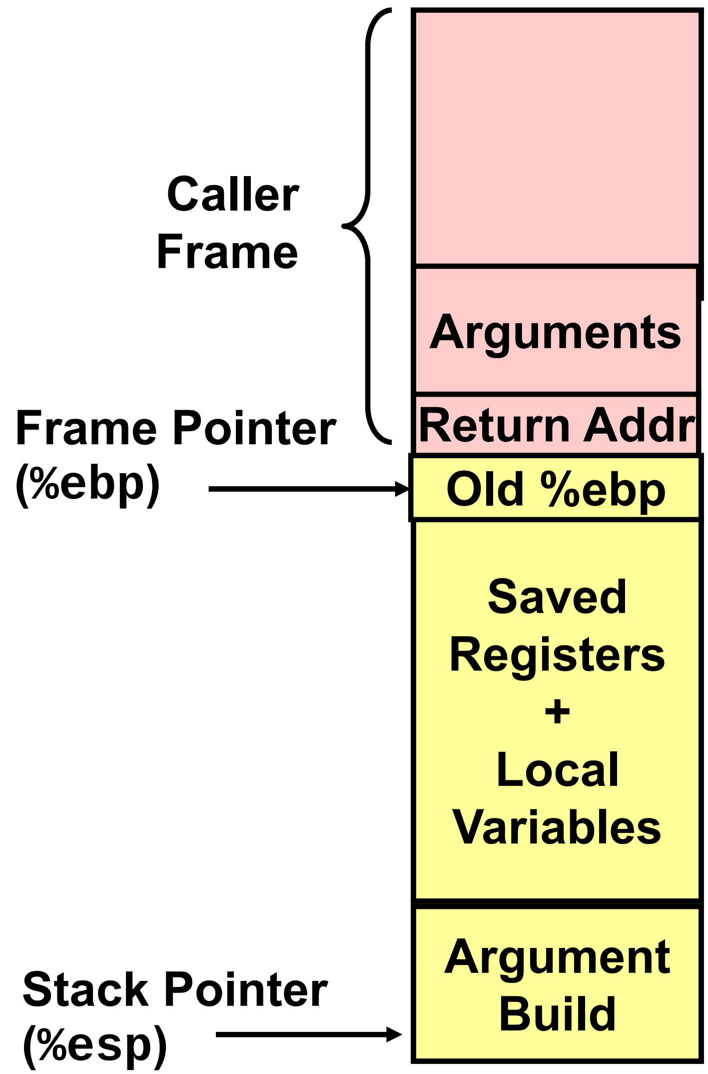
Call Chain





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



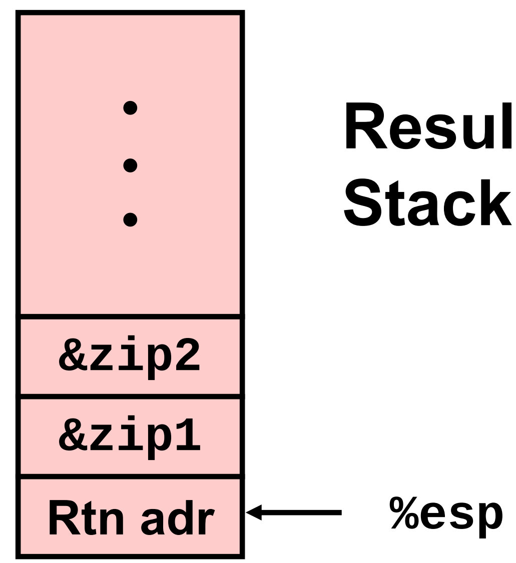
Revisiting swap

```
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
   • • •
```



Resulting Stack

Revisiting swap

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

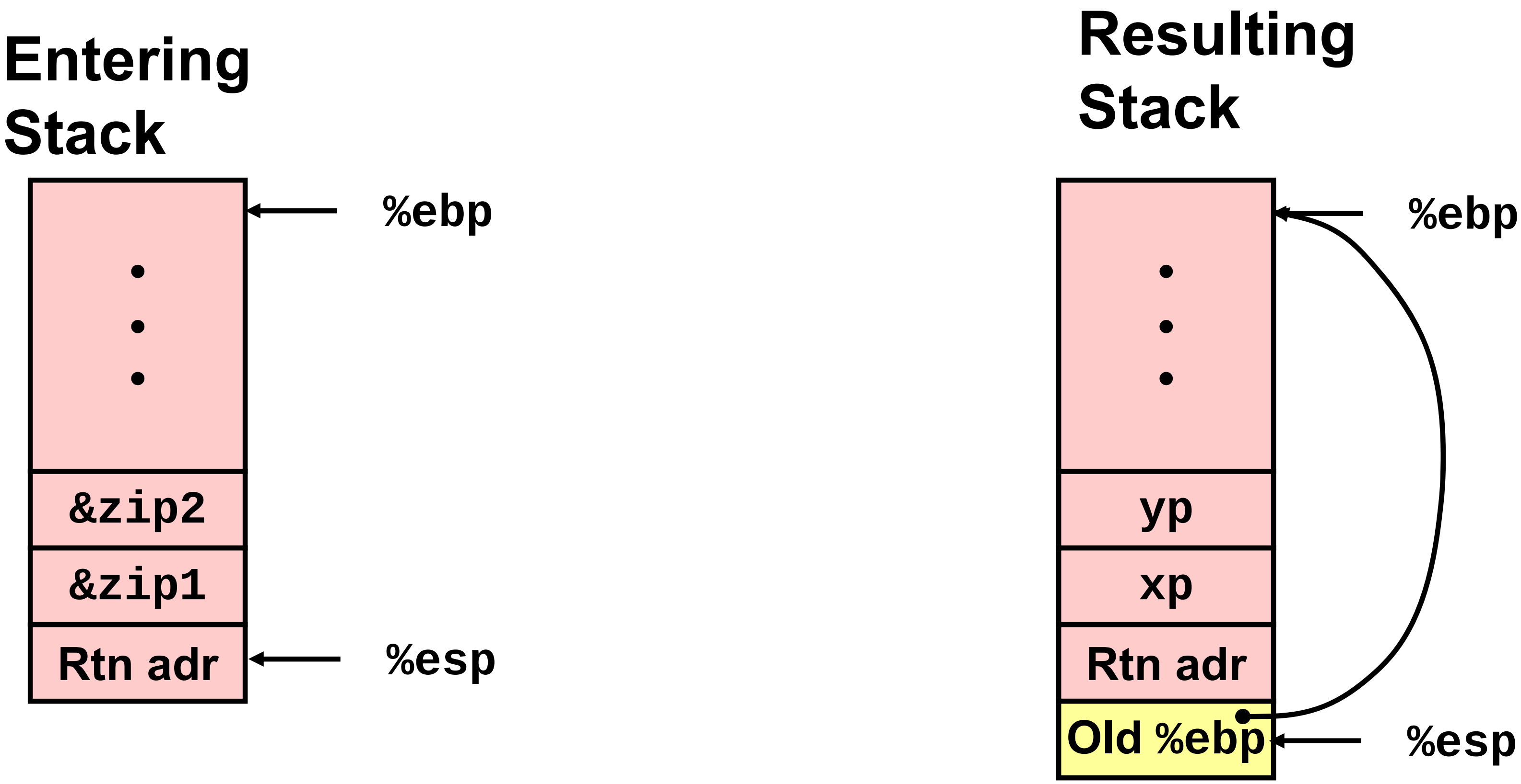
```
swap:
   pushl %ebp
   movl %esp,%ebp
   pushl %ebx

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

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

Set Up

Body

Finish

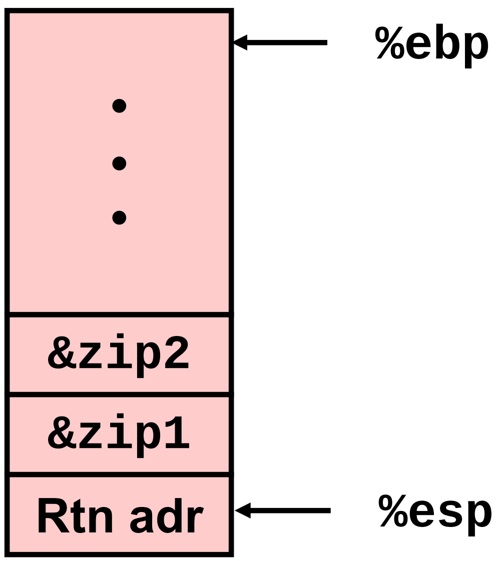
swap Setup #1



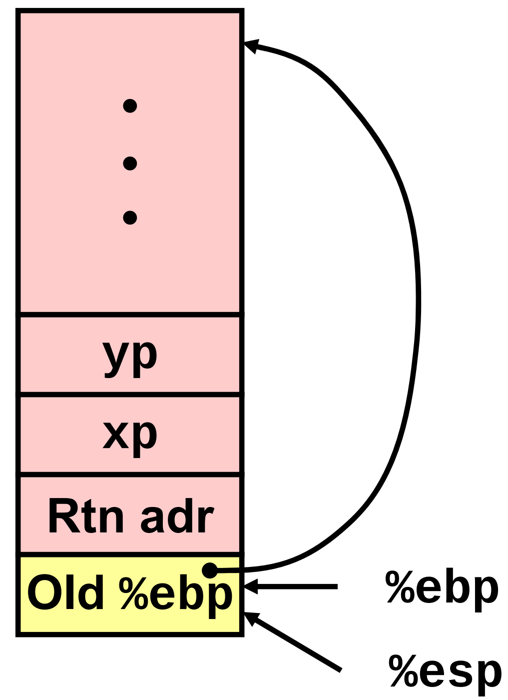

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

swap Setup #2

Entering Stack

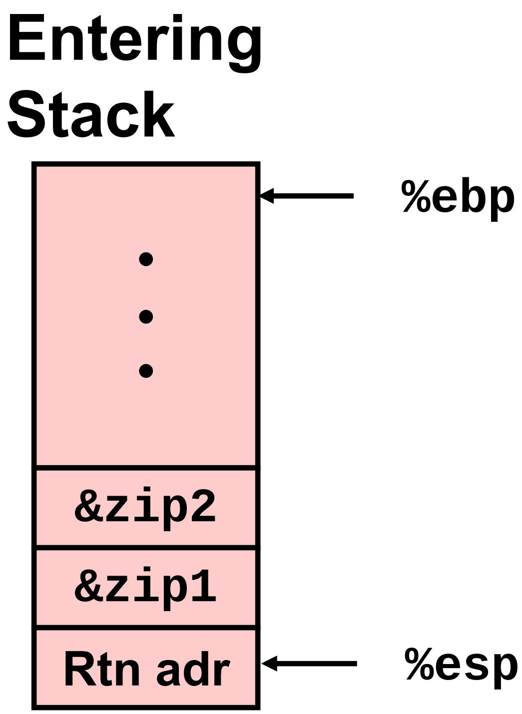


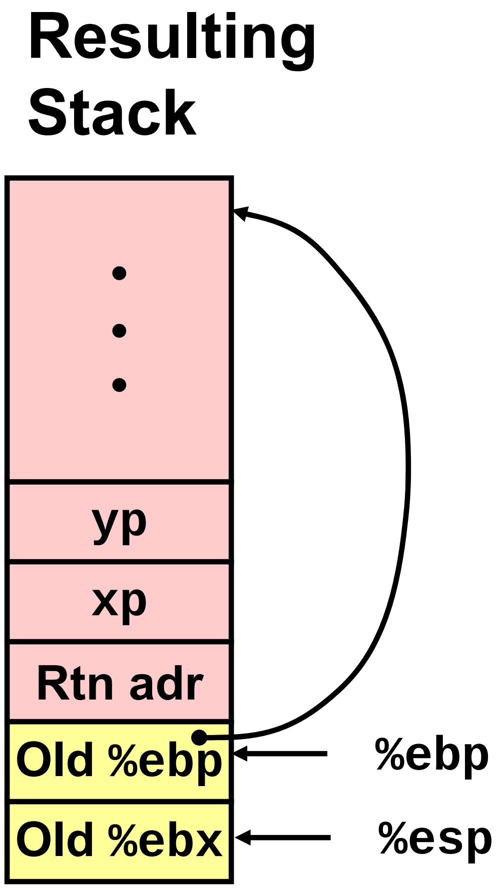
Resulting Stack




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

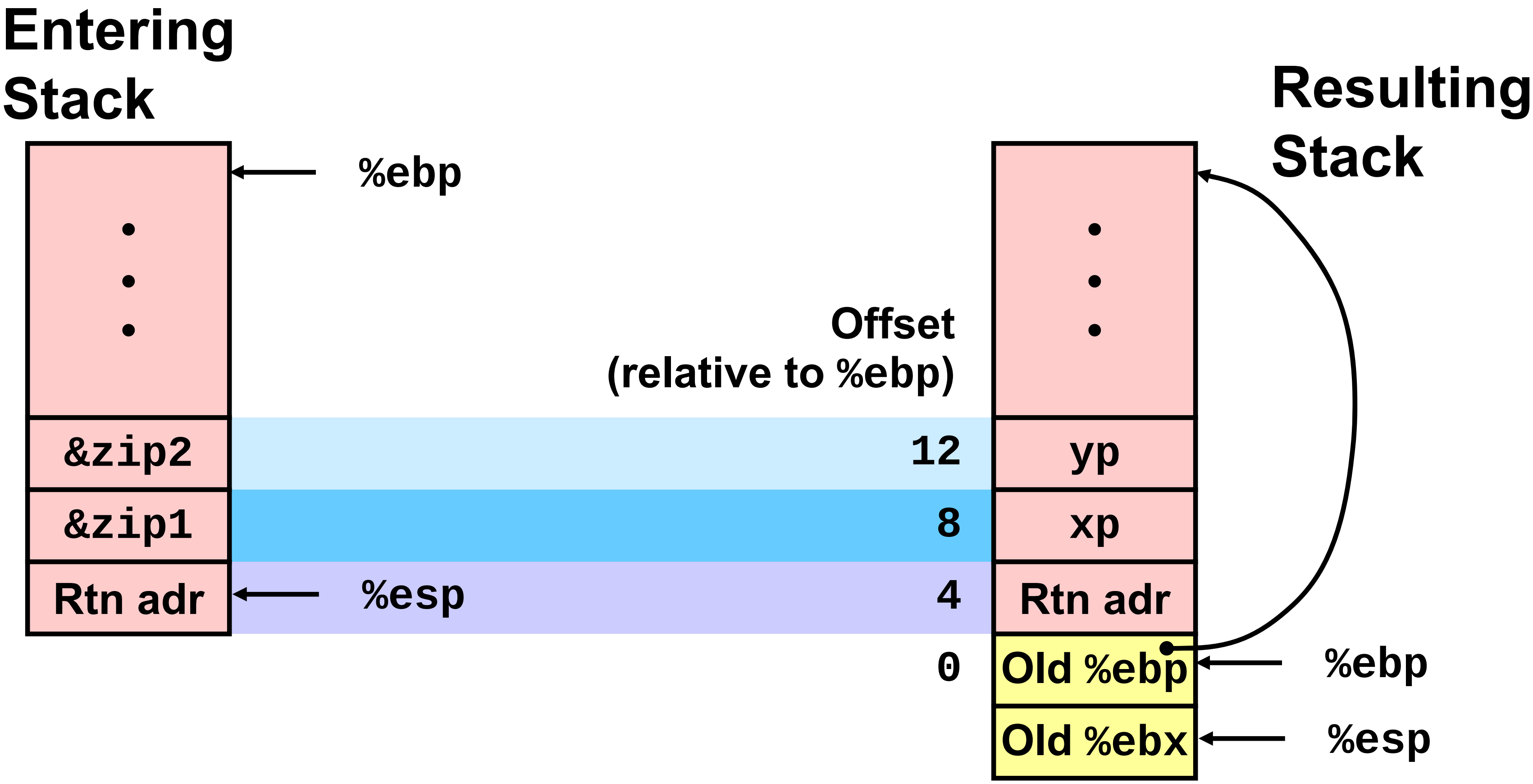
swap Setup #3






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

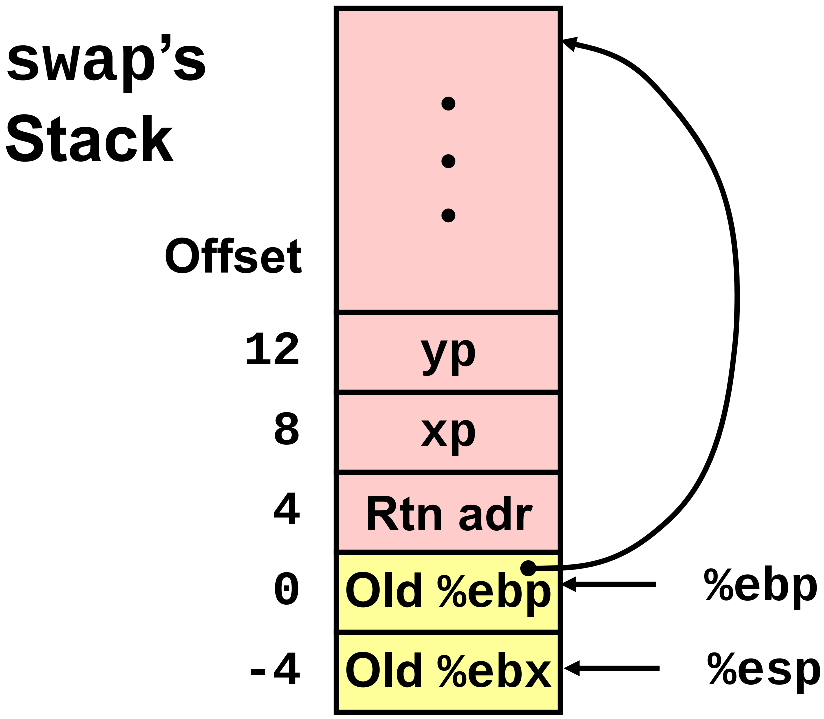
Effect of swap Setup

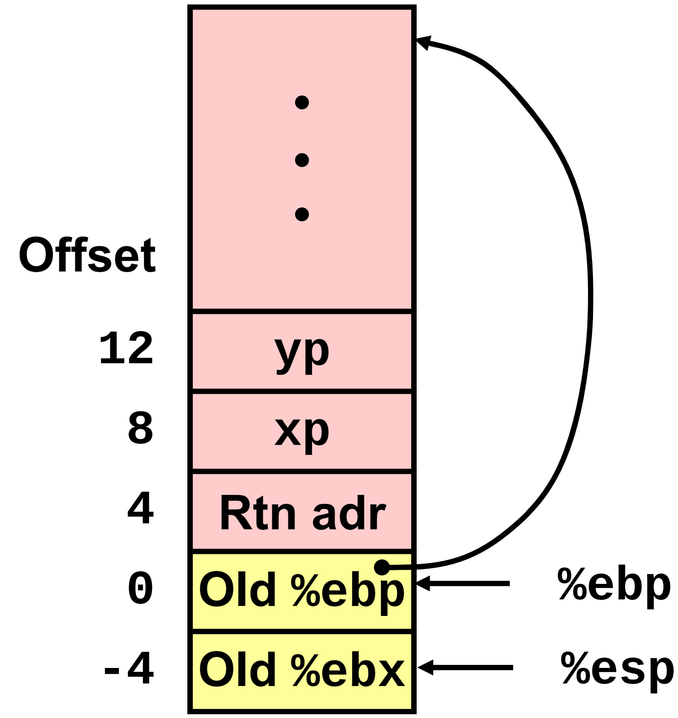



```
movl 12(%ebp),%ecx # get yp
movl 8(%ebp),%edx  # get xp
. . .
```

Body

swap Finish #1

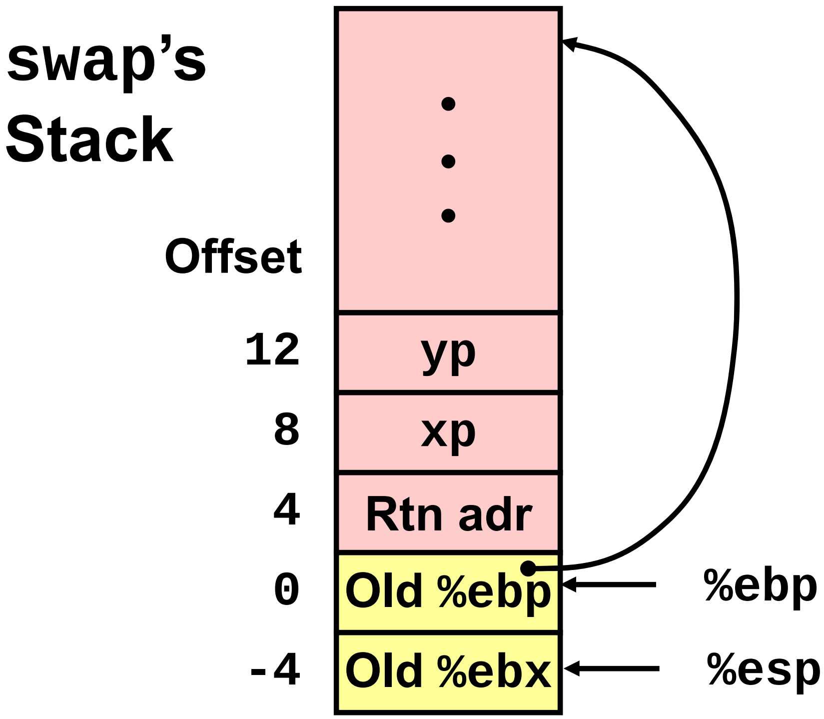


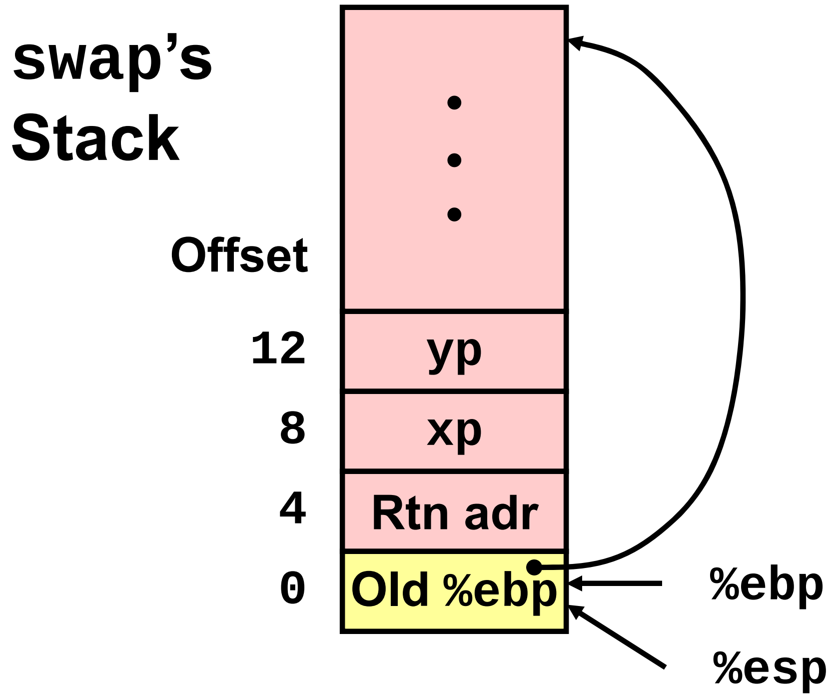



```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

- Observation
  - Saved & restored register %ebx

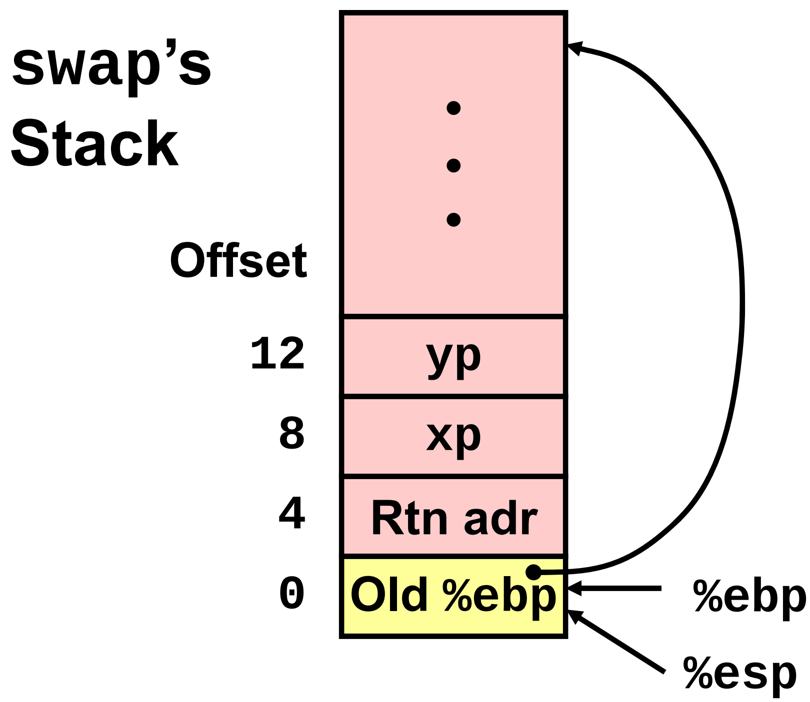
swap Finish #2

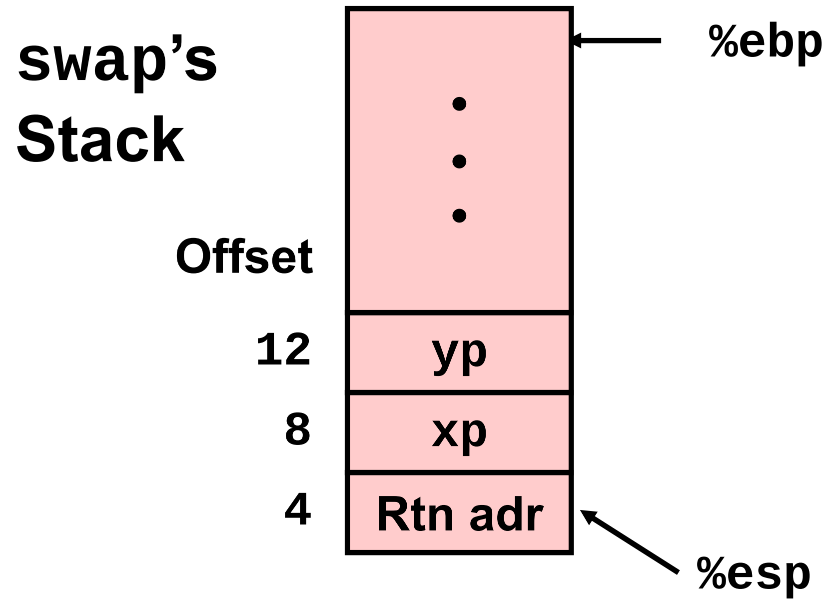





```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

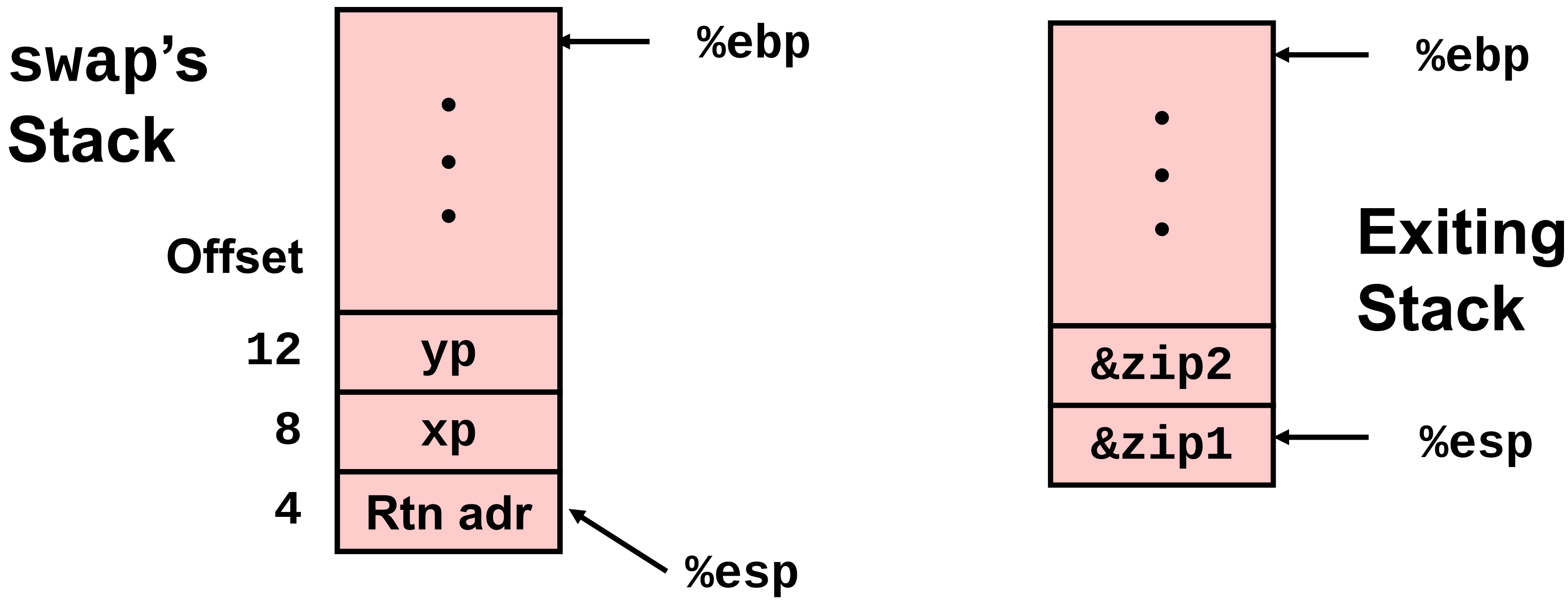
swap Finish #3






```
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

- When procedure `yoo` calls `who`:
  - `yoo` is the *caller*, `who` is the *callee*
- Can Registers 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 Registers 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

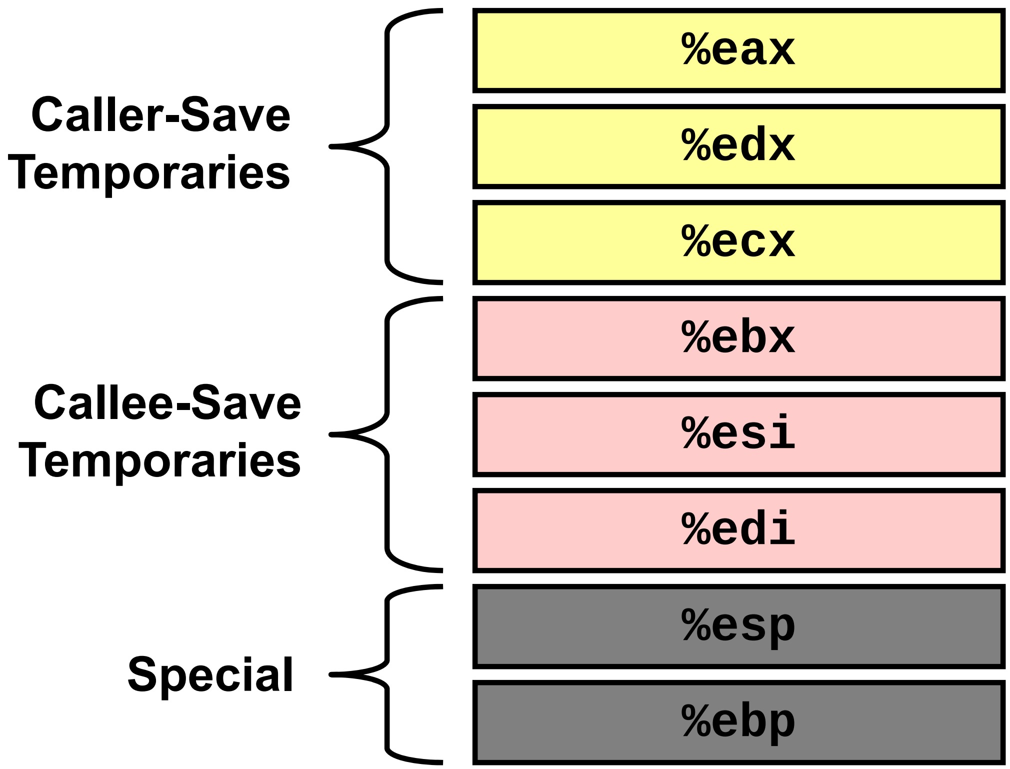
IA32/Linux Register Usage

- 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



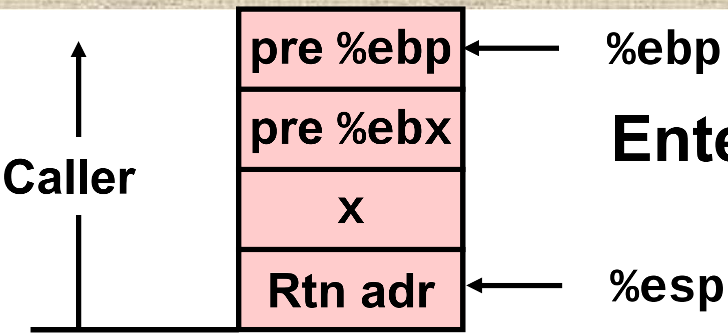
Recursive Factorial

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

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

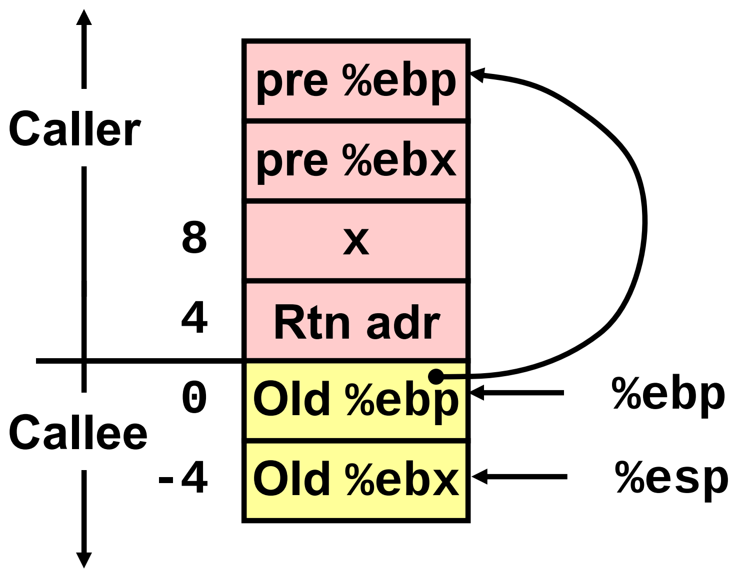
```
.globl rfact
    .type
rfact,@function
rfact:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 8(%ebp),%ebx
    cmpl $1,%ebx
    jle .L78
    leal -1(%ebx),%eax
    pushl %eax
    call rfact
    imull %ebx,%eax
    jmp .L79
    .align 4
.L78:
    movl $1,%eax
.L79:
    movl -4(%ebp),%ebx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

Rfact Stack Setup



Entering Stack

```
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;
}
```

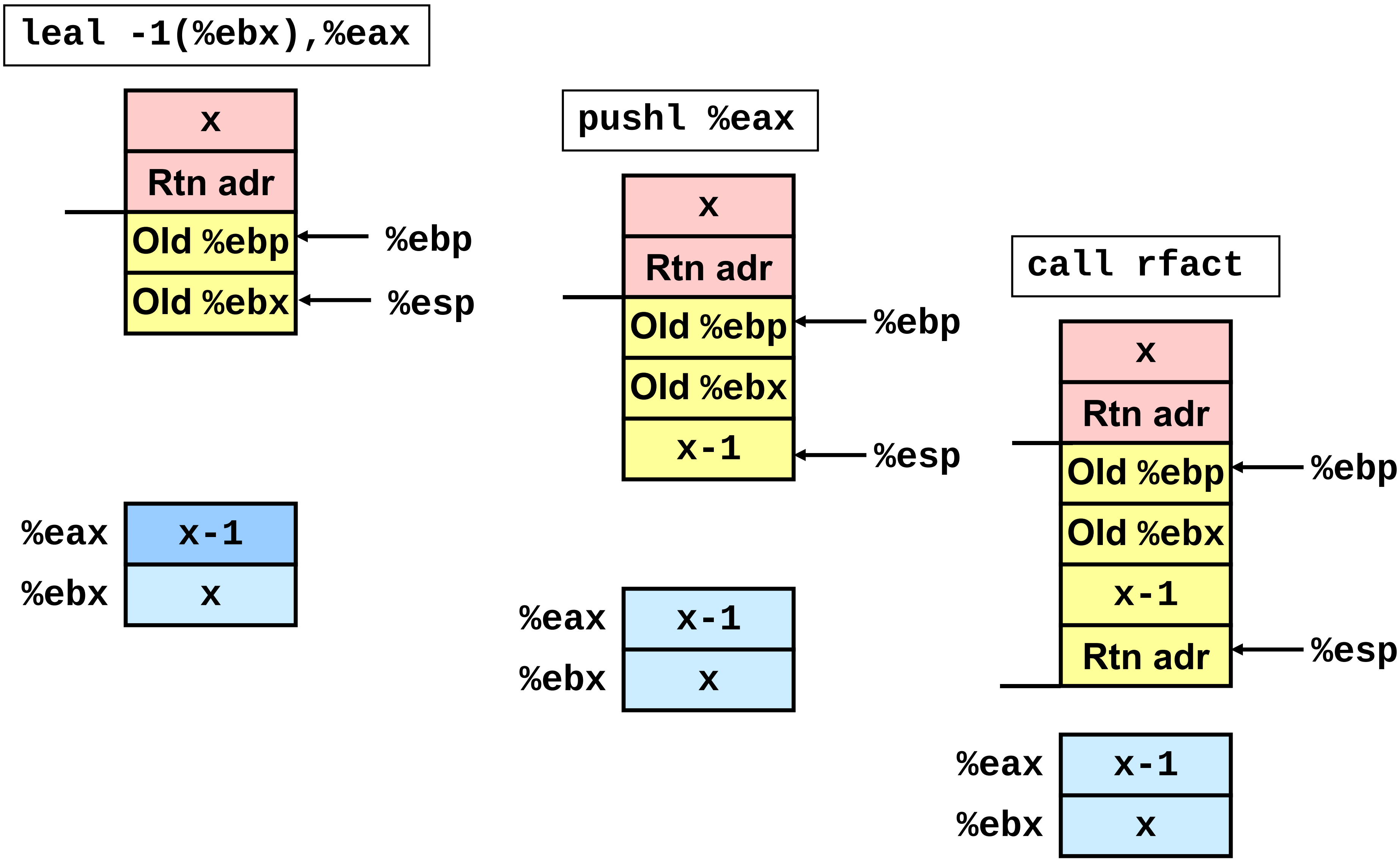
- Registers

  %ebx Stored value of x

  %eax

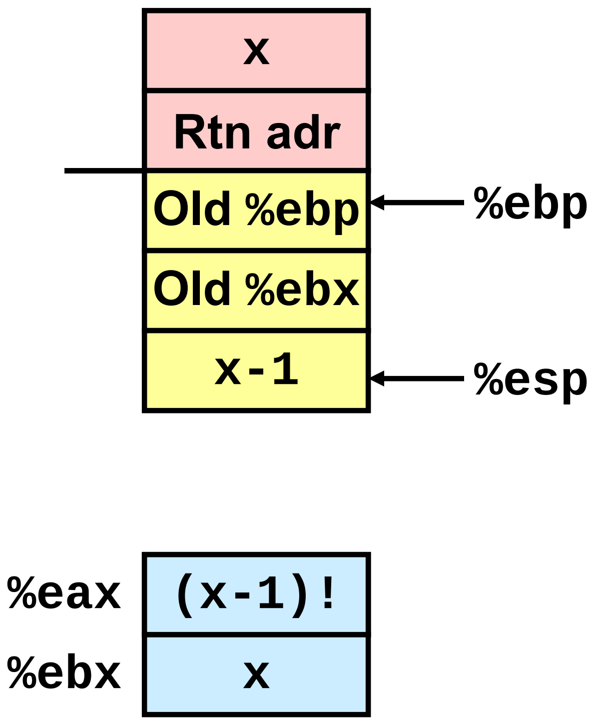
  - Temporary value of `x-1`
  - Returned value from `rfact(x-1)`
  - Returned value from this call

Rfact Recursion



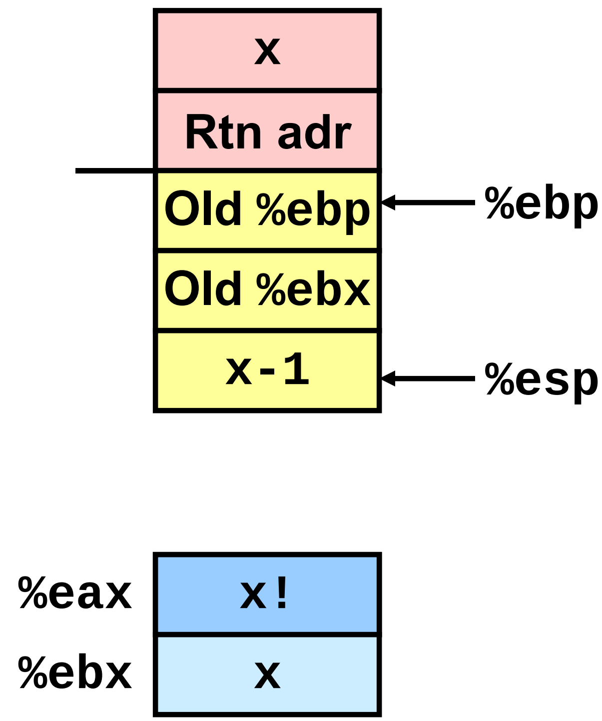
Rfact Result

Return from Call



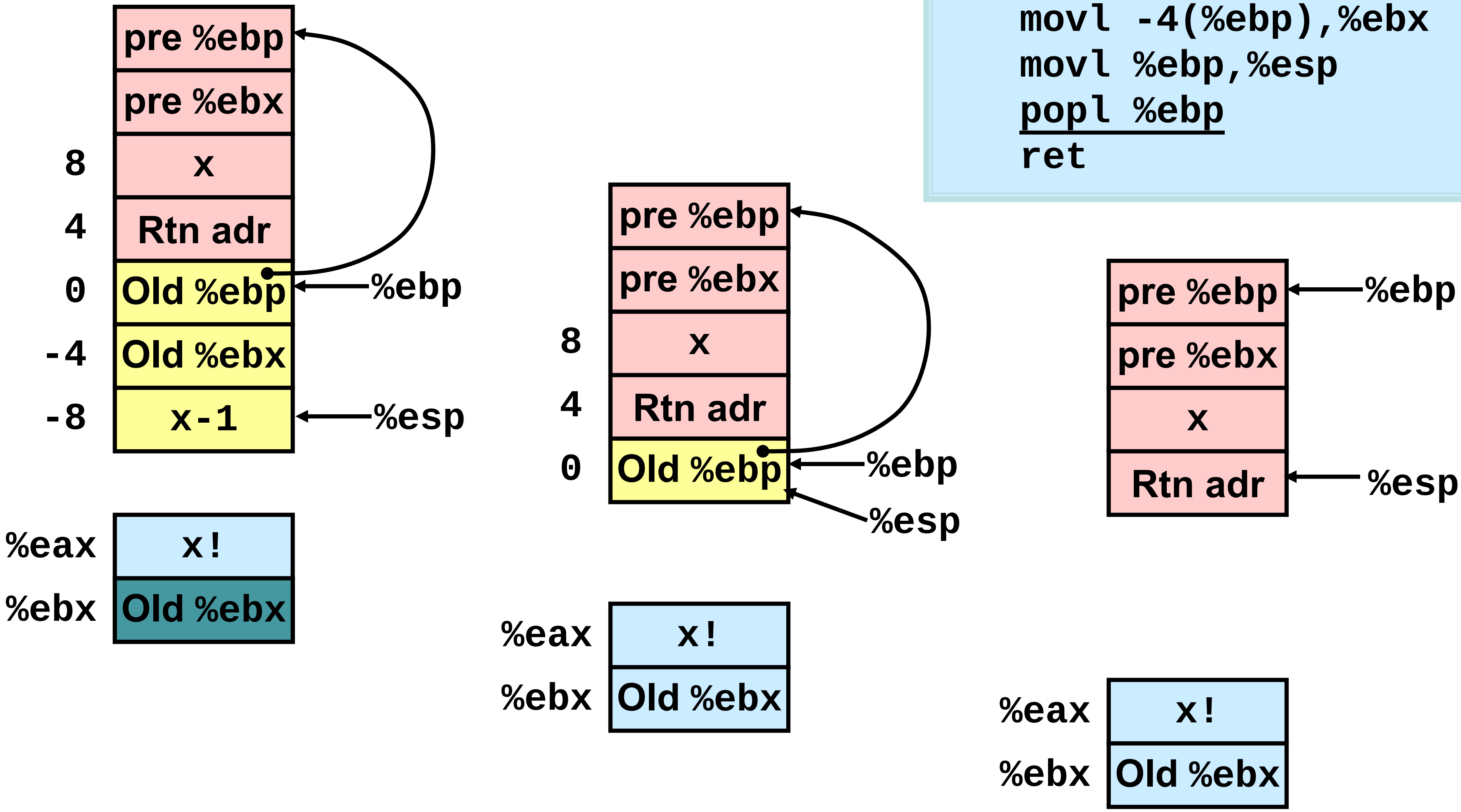
Assume that `rfact(x-1)` returns `(x-1)!` in register `%eax`

```
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

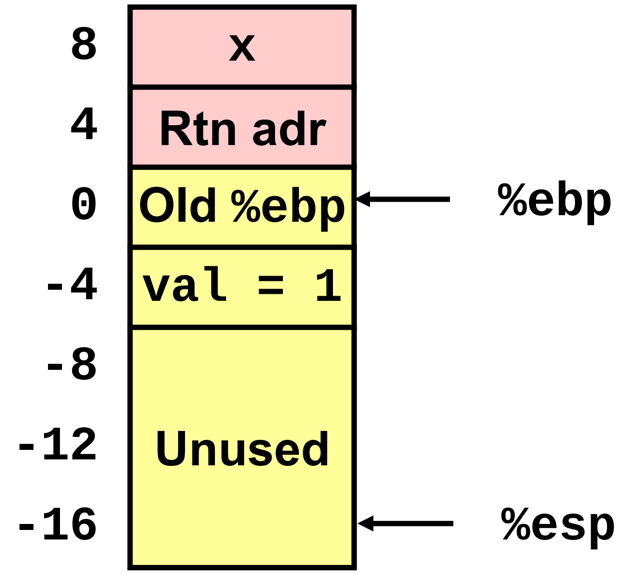
```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

– Pass pointer to update location

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;
}
```

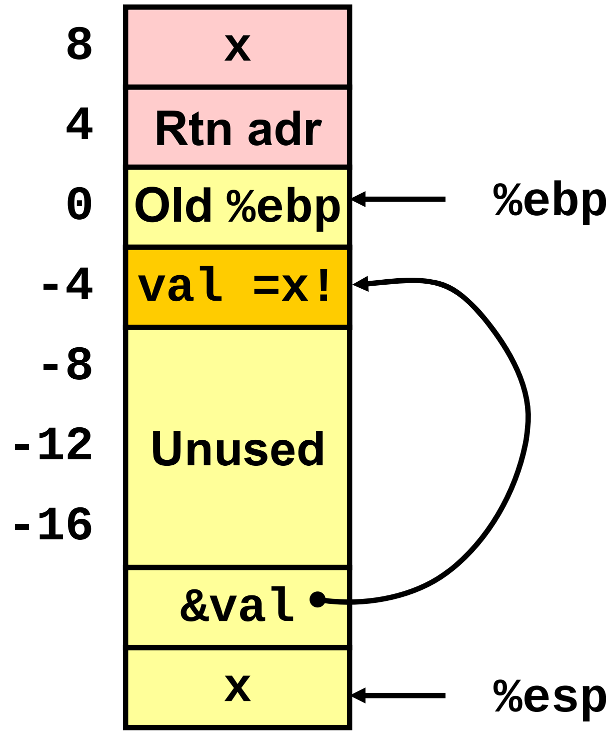
Passing Pointer

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
```

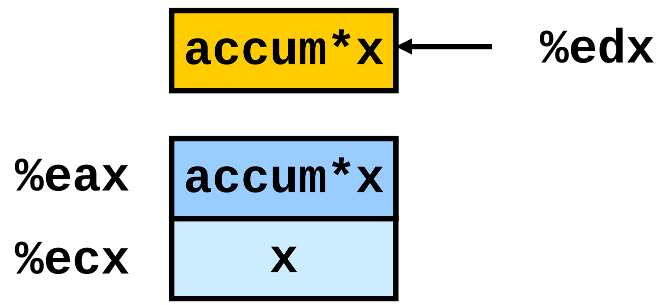
```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

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 pointer to accum

- Use access (%edx) to reference memory

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

x86-64 Integer Registers

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

– Twice the number of registers

– Accessible as 8, 16, 32, 64 bits

x86-64 Integer Registers

`%rax`	Return value	`%r8`	Argument #5
`%rbx`	Callee saved	`%r9`	Argument #6
`%rcx`	Argument #4	`%r10`	Callee saved
`%rdx`	Argument #3	`%r11`	Used for linking
`%rsi`	Argument #2	`%r12`	C: Callee saved
`%rdi`	Argument #1	`%r13`	Callee saved
`%rsp`	Stack pointer	`%r14`	Callee saved
`%rbp`	Callee saved	`%r15`	Callee saved

x86-64 Registers

- Arguments passed to functions via registers
  - If more than 6 integral parameters, then pass rest on stack
  - These registers can be used as caller-saved as well
- All references to stack frame via stack pointer
  - Eliminates need to update **`%ebp/%rbp`**
- Other Registers
  - 6+1 callee saved
  - 2 or 3 have special uses

x86-64 Long Swap

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

```
swap:
   movq    (%rdi), %rdx
   movq    (%rsi), %rax
   movq    %rax, (%rdi)
   movq    %rdx, (%rsi)
   ret
```

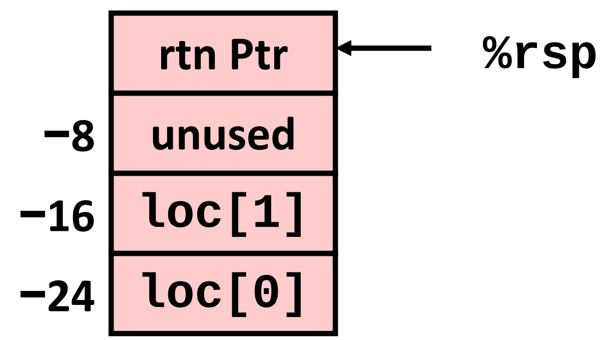
- Operands passed in registers
  - First (`xp`) in `%rdi`, second (`yp`) in `%rsi`
  - 64-bit pointers
- No stack operations required (except `ret`)
- Avoiding stack
  - Can hold all local information in registers

x86-64 Locals in the Red Zone

```
/* Swap, using local array */
void swap_a(long *xp, long *yp)
{
    volatile long loc[2];
    loc[0] = *xp;
    loc[1] = *yp;
    *xp = loc[1];
    *yp = loc[0];
}
```

```
swap_a:
   movq   (%rdi), %rax
   movq   %rax, -24(%rsp)
   movq   (%rsi), %rax
   movq   %rax, -16(%rsp)
   movq   -16(%rsp), %rax
   movq   %rax, (%rdi)
   movq   -24(%rsp), %rax
   movq   %rax, (%rsi)
   ret
```

- Avoiding Stack Pointer Change
  - Can hold all information within small window beyond stack pointer



x86-64 NonLeaf without Stack Frame

```
long scount = 0;

/* Swap a[i] & a[i+1] */
void swap_ele_se
  (long a[], int i)
{
    swap(&a[i], &a[i+1]);
    scount++;
}
```

- No values held while swap being invoked
- No callee save registers needed

```
swap_ele_se:
  movslq %esi,%rsi                # Sign extend i
  leaq   (%rdi,%rsi,8), %rdi # &a[i]
  leaq   8(%rdi), %rsi            # &a[i+1]
  call   swap                     # swap()
  incq   scount(%rip)             # scount++;
  ret
```

x86-64 Call using Jump

```
long scount = 0;

/* Swap a[i] & a[i+1] */
void swap_ele(long a[], int i)
{
    swap(&a[i], &a[i+1]);
}
```

- When `swap` executes `ret`, it will return from `swap_ele`
- Possible since `swap` is a "tail call" (no instructions afterwards)

```
swap_ele:
  movslq %esi,%rsi              # Sign extend i
  leaq   (%rdi,%rsi,8), %rdi # &a[i]
  leaq   8(%rdi), %rsi          # &a[i+1]
  jmp    swap                   # swap()
```

x86-64 Stack Frame Example

```
long sum = 0;
/* Swap a[i] & a[i+1] */
void swap_ele_su
  (long a[], int i)
{
    swap(&a[i], &a[i+1]);
    sum += a[i];
}
```

- Keeps values of a and i in callee save registers
- Must set up stack frame to save these registers

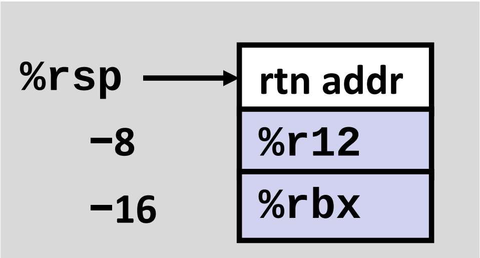
```
swap_ele_su:
  movq    %rbx, -16(%rsp)
  movslq  %esi,%rbx
  movq    %r12, -8(%rsp)
  movq    %rdi, %r12
  leaq    (%rdi,%rbx,8), %rdi
  subq    $16, %rsp
  leaq    8(%rdi), %rsi
  call    swap
  movq    (%r12,%rbx,8), %rax
  addq    %rax, sum(%rip)
  movq    (%rsp), %rbx
  movq    8(%rsp), %r12
  addq    $16, %rsp
  ret
```

Understanding x86-64 Stack Frame

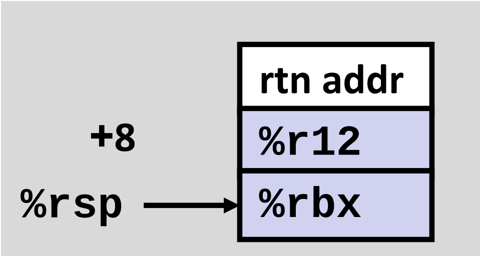
```
swap_ele_su:
  movq    %rbx, -16(%rsp)       # Save %rbx
  movslq  %esi,%rbx             # Extend & save i
  movq    %r12, -8(%rsp)        # Save %r12
  movq    %rdi, %r12            # Save a
  leaq    (%rdi,%rbx,8), %rdi   # &a[i]
  subq    $16, %rsp             # Allocate stack frame
  leaq    8(%rdi), %rsi         #     &a[i+1]
  call    swap                  # swap()
  movq    (%r12,%rbx,8), %rax   # a[i]
  addq    %rax, sum(%rip)       # sum += a[i]
  movq    (%rsp), %rbx          # Restore %rbx
  movq    8(%rsp), %r12         # Restore %r12
  addq    $16, %rsp             # Deallocate stack frame
  ret
```

Understanding x86-64 Stack Frame

```
swap_ele_su:
  movq    %rbx, -16(%rsp)        # Save %rbx

  movq    %r12, -8(%rsp)         # Save %r12
```




```
  subq    $16, %rsp              # Allocate stack frame
```




```
  movq    (%rsp), %rbx           # Restore %rbx
  movq    8(%rsp), %r12          # Restore %r12
  addq    $16, %rsp              # Deallocate stack frame
```

Interesting Features of Stack Frame

- Allocate entire frame at once
  - All stack accesses can be relative to **`%rsp`**
  - Do by decrementing stack pointer
  - Can delay allocation, since safe to temporarily use red zone

- Simple deallocation
  - Increment stack pointer
  - No base/frame pointer needed

x86-64 Procedure Summary

- Heavy use of registers
  - Parameter passing
  - More temporaries since more registers
- Minimal use of stack
  - Sometimes none
  - Allocate/deallocate entire block
- Many tricky optimizations
  - What kind of stack frame to use
  - Calling with jump
  - Various allocation techniques