Lecture 8:
Assembly Cont..

Announcements

- Project 2
  - Released, will be due in two weeks (July 28th)
- Exam Survey
  - Survey on resources available (do by July 17th)
- Midterm Exam
  - July 22nd , Next Wednesday
  - Format: Online 80 minute exam via Sakai Quizzes/Exams
  - We'll release sample questions later this week
- Recitation Today:
  - Project 2: BombLab Project Description and getting started

General Midterm Topics

- C Programming
  - Basic C syntax
  - Basic standard library function
    - Ex. printf(), scanf(), file I/O
  - Arrays, Data structures
  - Dynamic allocation
  - C Memory layout (text, data, stack, heap)
- Data Representation
  - Bit representation
  - Unsigned Int
  - Signed Int
  - Floating Point Representation
  - ASCII
- Assembly
  - Moving Data
  - Memory Addressing
  - Arithmetic
  - Procedure Control (Stack Discipline)

Today's Outline

- Recap
- Control Flow
  - Conditional Branching
  - Loops
  - Switch Case
- Procedure Control

Recap:

Recap: Moving Data

- Moving Data
  - `movq` ***Source, Dest***
- Operand Types
  - ***Immediate:*** Constant integer data
    - Example: **`$0x400`**, **`$-533`**
    - Like C constant, but prefixed with **`'$'`**
    - Encoded with 1, 2, or 4 bytes
  - ***Register:*** One of 16 integer registers
    - Example: **`%rax, %r13`**
    - But **`%rsp`** reserved for special use
    - Others have special uses for particular instructions
  - ***Memory:*** 8 consecutive bytes of memory at address given by register
    - Simplest example: **`(%rax)`**
    - Various other "address modes"

`%rax`
`%rcx`
`%rdx`
`%rbx`
`%rsi`
`%rdi`
`%rsp`
`%rbp`

`%rN`

Recap: Simple Memory Addressing Modes

- Normal (R) Mem[Reg[R]]
  - Register R specifies memory address
  - Example
    - `movq (%rcx),%rax`
- Displacement D(R) Mem[Reg[R]+D]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset
  - Example:
    - `movq 8(%rbp),%rdx`

Recap: Complete Memory Addressing Modes

- Most General Form
  - D(Rb,Ri,S)   Mem[Reg[Rb]+S*Reg[Ri]+ D]
  - D: Constant "displacement" 1, 2, or 4 bytes
  - Rb: Base register: Any of 16 integer registers
  - Ri: Index register: Any, except for **`%rsp`**
  - S: Scale: 1, 2, 4, or 8 (*why these numbers?*)
- Special Cases
  - (Rb,Ri)   Mem[Reg[Rb]+Reg[Ri]]
  - D(Rb,Ri)   Mem[Reg[Rb]+Reg[Ri]+D]
  - (Rb,Ri,S)   Mem[Reg[Rb]+S*Reg[Ri]]

Recap: Address Computation Instruction (LEAQ)

- **`leaq`** *Src*, *Dst*
  - *Src* is address mode expression
  - Set *Dst* to address denoted by expression
- Uses
  - Computing addresses without a memory reference
    - E.g., translation of **`p = &x[i];`**
  - Computing arithmetic expressions of the form x + k*y
    - k = 1, 2, 4, or 8
    - Example:

Register	Value
`%eax`	`x`
`%ecx`	`y`

Instruction	Result
`leaq 6(%eax),%edx`	`6 + x`
`leaq (%eax,%ecx), %edx`	`x + y`
`leaq (%eax,%ecx,4), %edx`	`x + 4y`
`leaq 7(%eax,%eax,8), %edx`	`7 + 9x`
`leaq 0xA (,%ecx,4), %edx`	`10 + 4y`
`leaq 9(%eax,%ecx,2), %edx`	`9 + x + 2y`

Recap: Some Arithmetic Operations

- Two Operand Instructions:

Format		***Computation***	
`addq`	*Src,Dest*	Dest = Dest + Src	
`subq`	*Src,Dest*	Dest = Dest – Src	
`imulq`	*Src,Dest*	Dest = Dest * Src	
`salq`	*Src,Dest*	Dest = Dest << Src	***Also called shlq***
`sarq`	*Src,Dest*	Dest = Dest >> Src	***Arithmetic***
`shrq`	*Src,Dest*	Dest = Dest >> Src	***Logical***
`xorq`	*Src,Dest*	Dest = Dest ^ Src	
`andq`	*Src,Dest*	Dest = Dest & Src	
`orq`	*Src,Dest*	Dest = Dest | Src	

- Watch out for argument order!
- No distinction between signed and unsigned int (why?)

Recap: Some Other Arithmetic Operations

- One Operand Instructions

`incq`	*Dest*	*Dest* = *Dest* + *1*
`decq`	*Dest*	*Dest* = *Dest* – *1*
`negq`	*Dest*	*Dest* = – *Dest*
`notq`	*Dest*	*Dest* = ~*Dest*

- See book for more instructions

Recap: Condition Codes

- Single Bit Registers (set after each instruction)
  - `CF` Carry Flag: instruction generated a carry out
  - `SF` Sign Flag: instruction yielded a negative value
  - `ZF` Zero Flag: instruction yielded zero
  - `OF` Overflow Flag: instruction caused 2's complement overflow
- Can be set either implicitly or explicitly.
  - Implicitly by almost all logic and arithmetic operations
  - Explicitly by specific comparison operations
- *Not* Set by `leaq/leal` instruction
  - Intended for use in address computation only

Conditionals/Control Flow

- ~~Control: Condition codes~~
- Conditional branches
- Loops
- Switch Statements

Assembly: Conditional Branching

Jumping

- jX Instructions
  - Jump to different part of code depending on condition codes

jX	Condition	Description
`jmp`	`1`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned)
`jb`	`CF`	Below (unsigned)

Conditional Branch Example

Generation

```
gcc –Og -S –fno-if-conversion control.c
```

```c
long absdiff
  (long x, long y)
{
  long result;
  if (x > y)
    result = x-y;
  else
    result = y-x;
  return result;
}
```

```
absdiff:
   cmpq    %rsi, %rdi  # x:y
   jle     .L4
   movq    %rdi, %rax
   subq    %rsi, %rax
   ret
.L4:       # x <= y
   movq    %rsi, %rax
   subq    %rdi, %rax
   ret
```

Register	Use(s)
`%rdi`	Argument **x**
`%rsi`	Argument **y**
`%rax`	Return value

Expressing with Goto Code

C allows **goto** statement

Jump to position designated by label

```
long absdiff
   (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff_j
   (long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest)
       goto Else;
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```

General Conditional Expression Translation (Using Branches)

C Code

```
val = Test ? Then_Expr : Else_Expr;
```

```
val = x>y ? x-y : y-x;
```

Goto Version

```
  ntest = !Test;
  if (ntest) goto Else;
  val = Then_Expr;
  goto Done;
Else:
  val = Else_Expr;
Done:
  . . .
```

- Create separate code regions for then & else expressions
- Execute appropriate one
- Can it be better?

Using Conditional Moves

- More efficient assignment with conditional move
- Conditional Move Instructions
  - ex. `cmovle` *src, dest*
  - Instruction supports:
    - if (Test) Dest <- Src
  - Supported in post-1995 x86 processors
  - GCC tries to use them
  - But, only when known to be safe
- Why?
  - Branches are very disruptive to instruction flow through pipelines
  - Conditional moves do not require control transfer

C Code

```
val = Test
    ? Then_Expr
    : Else_Expr;
```

Goto Version

```
  result = Then_Expr;
  eval = Else_Expr;
  nt = !Test;
  if (nt) result = eval;
  return result;
```

Conditional Move Example

```
long absdiff
  (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

Register	Use(s)
`%rdi`	Argument `x`
`%rsi`	Argument `y`
`%rax`	Return value

```
absdiff:
   movq    %rdi, %rax  # x
   subq    %rsi, %rax  # result = x-y
   movq    %rsi, %rdx
   subq    %rdi, %rdx  # eval = y-x
   cmpq    %rsi, %rdi  # x:y
   cmovle  %rdx, %rax  # if <=, result = eval
   ret
```

Conditionals/Control Flow

- ~~Control: Condition codes~~
- ~~Conditional branches~~
- Loops
- Switch Statements

Assembly:
Loops

“Do-While” Loop Example

C Code

```
long pcount_do
  (unsigned long x) {
  long result = 0;
  do {
    result += x & 0x1;
    x >>= 1;
  } while (x);
  return result;
}
```

Goto Version

```
long pcount_goto
  (unsigned long x) {
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

Count number of 1’s in argument `x` (“popcount”)

Use conditional branch to either continue looping or to exit loop

"Do-While" Loop Compilation

Goto Version

```
long pcount_goto
  (unsigned long x) {
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

Register	Use(s)
%rdi	Argument x
%rax	result

```
        movl    $0, %eax      #  result = 0
.L2:                          # loop:
        movq    %rdi, %rdx
        andl    $1, %edx      # t = x & 0x1
        addq    %rdx, %rax    # result += t
        shrq    %rdi          # x >>= 1
        jne     .L2           # if (x) goto loop
        rep; ret
```

General "Do-While" Translation

C Code

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

Goto Version

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

Body:

```
{
  Statement1;
  Statement2;
    …
  Statementn;
}
```

General "While" Translation #1

- "Jump-to-middle" translation
- Used with `–Og`

While version

```
while (Test)
  Body
```

Goto Version

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

While Loop Example #1

C Code

```
long pcount_while
  (unsigned long x) {
  long result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

Jump to Middle

```
long pcount_goto_jtm
  (unsigned long x) {
  long result = 0;
  goto test;
 loop:
  result += x & 0x1;
  x >>= 1;
 test:
  if(x) goto loop;
  return result;
}
```

- Compare to do-while version of function
- Initial goto starts loop at test

General "While" Translation #2

- "Do-while" conversion
- Used with –O1

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:
```

While Loop Example #2

C Code

```
long pcount_while
  (unsigned long x) {
  long result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

Do-While

```
long pcount_goto_dw
  (unsigned long x) {
  long result = 0;
  if (!x) goto done;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
 done:
  return result;
}
```

- Compare to do-while version of function
- Initial conditional guards entrance to loop

"For" Loop Form

General Form

```
for (Init; Test; Update )

    Body
```

```
size_t WSIZE = 8*sizeof(int)
long pcount_for
  (unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
    (x >> i) & 0x1;
  result += bit;
}
```

"For" Loop -> While Loop

For Version

```
for (Init; Test; Update)

        Body
```

While Version

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

For-While Conversion

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
      (x >> i) & 0x1;
  result += bit;
}
```

```
long pcount_for_while
  (unsigned long x)
{
  size_t i;
  long result = 0;
  i = 0;
  while (i < WSIZE)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
    i++;
  }
  return result;
}
```

"For" Loop Do-While Conversion

C Code

```
long pcount_for
  (unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

Goto Version

```
long pcount_for_goto_dw
  (unsigned long x) {
  size_t i;
  long result = 0;
  i = 0;                          Init
  if (!(i < WSIZE))               ! Test
    goto done;
 loop:
  {
    unsigned bit =
      (x >> i) & 0x1;             Body
    result += bit;
  }
  i++;                            Update
  if (i < WSIZE)                  Test
    goto loop;
 done:
  return result;
}
```

Conditionals/Control Flow

- ~~Control: Condition codes~~
- ~~Conditional branches~~
- ~~Loops~~
- Switch Statements

Assembly: Switch Cases

Switch Statement Example

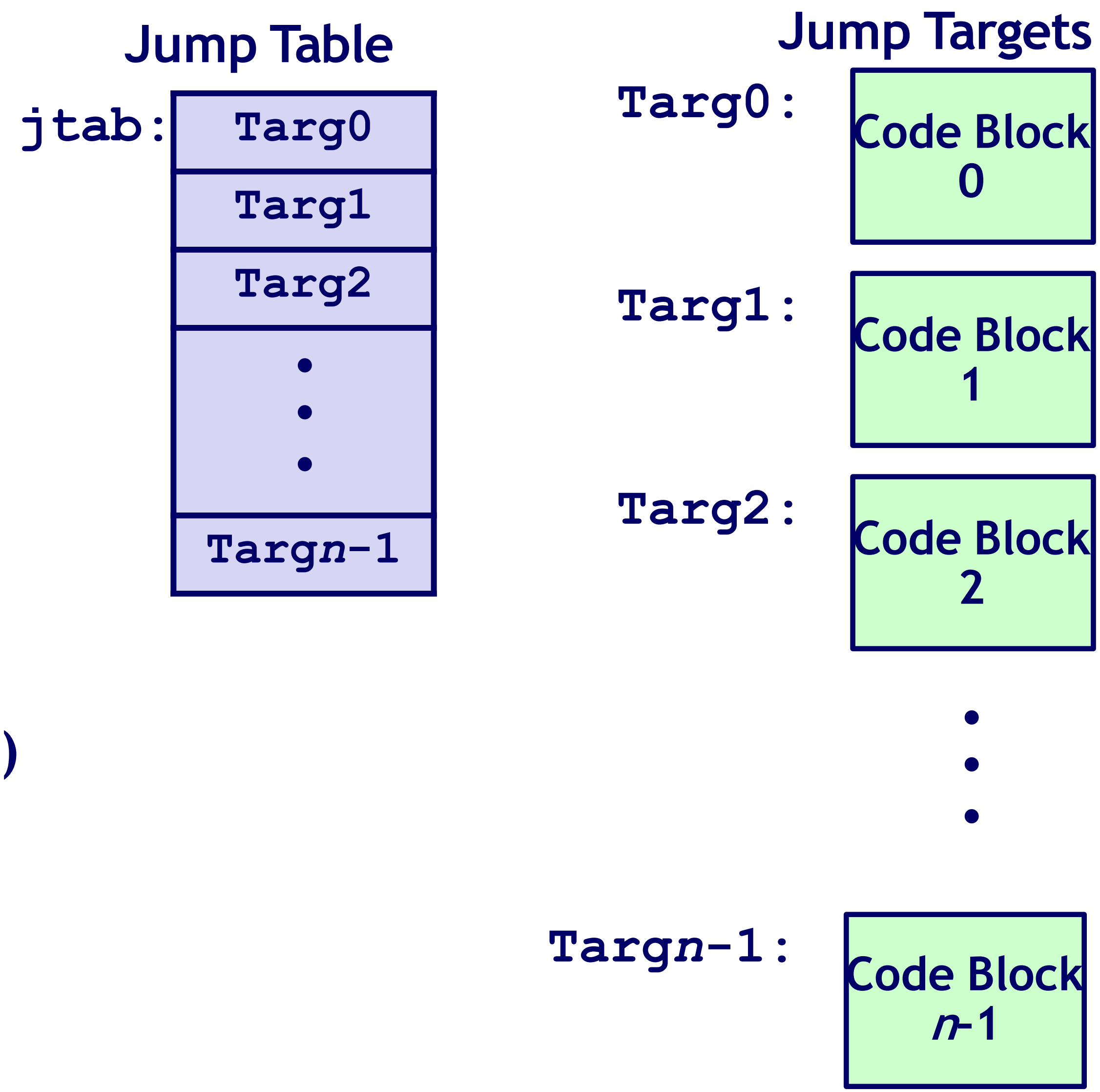
```
long switch_eg
    (long x, long y, long z)
{
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

- 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
}
```



Translation (Extended C)

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

Switch Statement Example

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

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi     # x:6
    ja      .L8
    jmp     *.L4(,%rdi,8)
```

What range of values takes default?

Register	Use(s)
`%rdi`	Argument `x`
`%rsi`	Argument `y`
`%rdx`	Argument `z`
`%rax`	Return value

Note : w not initialized here, optimized out in assembly (later)

Switch Statement Example

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

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8 # x = 0
  .quad     .L3 # x = 1
  .quad     .L5 # x = 2
  .quad     .L9 # x = 3
  .quad     .L8 # x = 4
  .quad     .L7 # x = 5
  .quad     .L7 # x = 6
```

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi          # x:6
    ja      .L8               # Use default
    jmp     *.L4(,%rdi,8)     # goto *JTab[x]
```

Indirect jump

Assembly Setup Explanation

- Table Structure
  - Each target requires 8 bytes
  - Base address at **`.L4`**
- Jumping
  - **Direct:** **`jmp .L8`**
    - Jump target is denoted by label **`.L8`**
  - **Indirect:** **`jmp *.L4(,%rdi,8)`**
    - Start of jump table: **`.L4`**
    - Must scale by factor of 8 (addresses are 8 bytes)
    - Fetch target from effective address **`.L4 + x*8`**
    - Only for $0 \le$ **`x`** ≤ 6

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8 # x = 0
  .quad     .L3 # x = 1
  .quad     .L5 # x = 2
  .quad     .L9 # x = 3
  .quad     .L8 # x = 4
  .quad     .L7 # x = 5
  .quad     .L7 # x = 6
```

Jump Table

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8  # x = 0
  .quad     .L3  # x = 1
  .quad     .L5  # x = 2
  .quad     .L9  # x = 3
  .quad     .L8  # x = 4
  .quad     .L7  # x = 5
  .quad     .L7  # x = 6
```

```
switch(x) {
case 1:        // .L3
    w = y*z;
    break;
case 2:        // .L5
    w = y/z;
    /* Fall Through */
case 3:        // .L9
    w += z;
    break;
case 5:
case 6:        // .L7
    w -= z;
    break;
default:       // .L8
    w = 2;
}
```

Code Blocks (x == 1)

```
  switch(x) {
  case 1:       // .L3
        w = y*z;
        break;
   . . .
}
```

```
.L3:
   movq    %rsi, %rax   # y
   imulq   %rdx, %rax   # y*z
   ret
```

Register	Use(s)
`%rdi`	Argument `x`
`%rsi`	Argument `y`
`%rdx`	Argument `z`
`%rax`	Return value

Handling Fall-Through

```
long w = 1;
   . . .
switch(x) {
   . . .
case 2:
    w = y/z;
    /* Fall Through */
case 3:
    w += z;
    break;
   . . .
}
```

```
case 2:
    w = y/z;
    goto merge;
```

```
case 3:
        w = 1;

merge:
        w += z;
```

Code Blocks (x == 2, x == 3)

```
long w = 1;
   . . .
switch(x) {
   . . .
case 2:
     w = y/z;
     /* Fall Through */
case 3:
     w += z;
     break;
   . . .
}
```

```
.L5:                       # Case 2
   movq    %rsi, %rax
   cqto
   idivq   %rcx            #   y/z
   jmp     .L6             #   goto merge
.L9:                       # Case 3
   movl    $1, %eax        #   w = 1
.L6:                       # merge:
   addq    %rcx, %rax      #   w += z
   ret
```

Register	Use(s)
`%rdi`	Argument `x`
`%rsi`	Argument `y`
`%rdx`	Argument `z`
`%rax`	Return value

Code Blocks (x == 5, x == 6, default)

```
  switch(x) {
    . . .
    case 5:  // .L7
    case 6:  // .L7
        w -= z;
        break;
    default: // .L8
        w = 2;
}
```

```
.L7:                      # Case 5,6
    movl  $1, %eax        # w = 1
    subq  %rdx, %rax      # w -= z
    ret
.L8:                      # Default:
    movl  $2, ret         # 2
    ret
```

Register	Use(s)
`%rdi`	Argument **`x`**
`%rsi`	Argument **`y`**
`%rdx`	Argument **`z`**
`%rax`	Return value

Switch Cases Overview

```
    case 1:         // .L3
        w = y*z;
        break;
    case 2:         // .L5
        w = y/z;
        /* Fall Through */
    case 3:         // .L9
        w += z;
        break;
    case 5:
    case 6:         // .L7
        w -= z;
        break;
    default:        // .L8
        w = 2;
```

```
.L3:
   movq   %rsi,  %rax   # y
   imulq  %rdx,  %rax   # y*z
   ret
.L5:                    # Case 2
   movq    %rsi, %rax
   cqto
   idivq   %rcx         #  y/z
   jmp     .L6        #  goto merge
.L9:                    # Case 3
   movl    $1, %eax     #  w = 1
.L6:                    # merge:
   addq    %rcx, %rax #  w += z
   ret
.L7:                  # Case 5,6
  movl  $1, %eax      #  w = 1
  subq  %rdx, %rax #  w -= z
  ret
.L8:                  # Default:
  movl  $2, %eax      #  2
  ret
```

Switch Cases Overview

```
switch_eg:
    movq      %rdx, %rcx
    cmpq      $6, %rdi     # x:6
    ja        .L8
    jmp       *.L4(,%rdi,8)
```

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8 # x = 0
  .quad     .L3 # x = 1
  .quad     .L5 # x = 2
  .quad     .L9 # x = 3
  .quad     .L8 # x = 4
  .quad     .L7 # x = 5
  .quad     .L7 # x = 6
```

Register	Use(s)
`%rdi`	Argument `x`
`%rsi`	Argument `y`
`%rdx`	Argument `z`
`%rax`	Return value

```
.L3:
   movq   %rsi,   %rax   # y
   imulq  %rdx,   %rax   # y*z
   ret
.L5:                     # Case 2
   movq     %rsi, %rax
   cqto
   idivq    %rcx         #  y/z
   jmp      .L6        #  goto merge
.L9:                     # Case 3
   movl     $1, %eax     #  w = 1
.L6:                     # merge:
   addq     %rcx, %rax #  w += z
   ret
.L7:                  # Case 5,6
  movl  $1, %eax      #  w = 1
  subq  %rdx, %rax #  w -= z
  ret
.L8:                  # Default:
  movl  $2, %eax      #  2
  ret
```

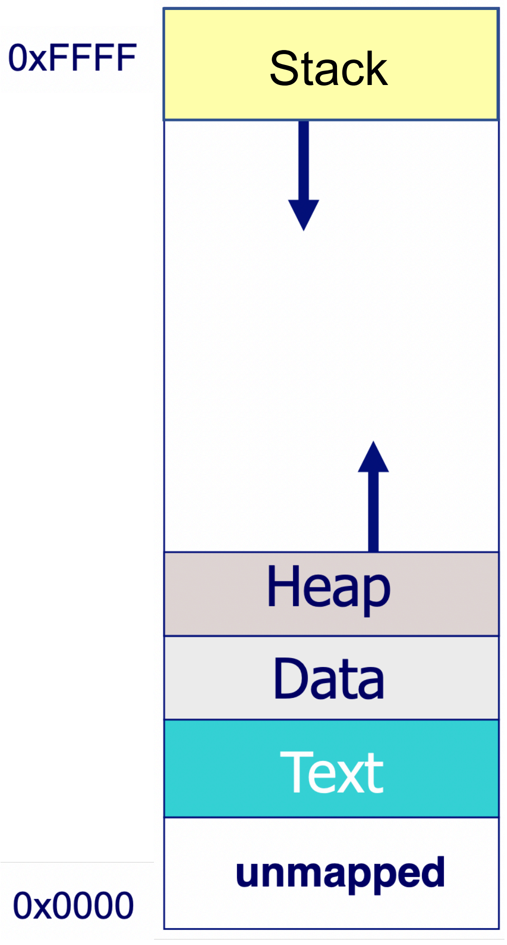
Assembly: Stack and Procedure Control

What are we trying to understand?

- We know about the stack
  - But how is the stack controlled in assembly?
- How is control passed between functions
- How is data passed between functions
- We'll learn using x86 (32-bit) convention as that is what you'll see in the bomblab
  - Registers look like `%eax, %ebx, %ecx`, etc.
    - Register names start with "e" to denote 32-bit register
  - Instructions look like *movl, leal, addl*, etc
    - Instructions end in "l" to denote dealing with "long" values (3 byte/32-bit values)

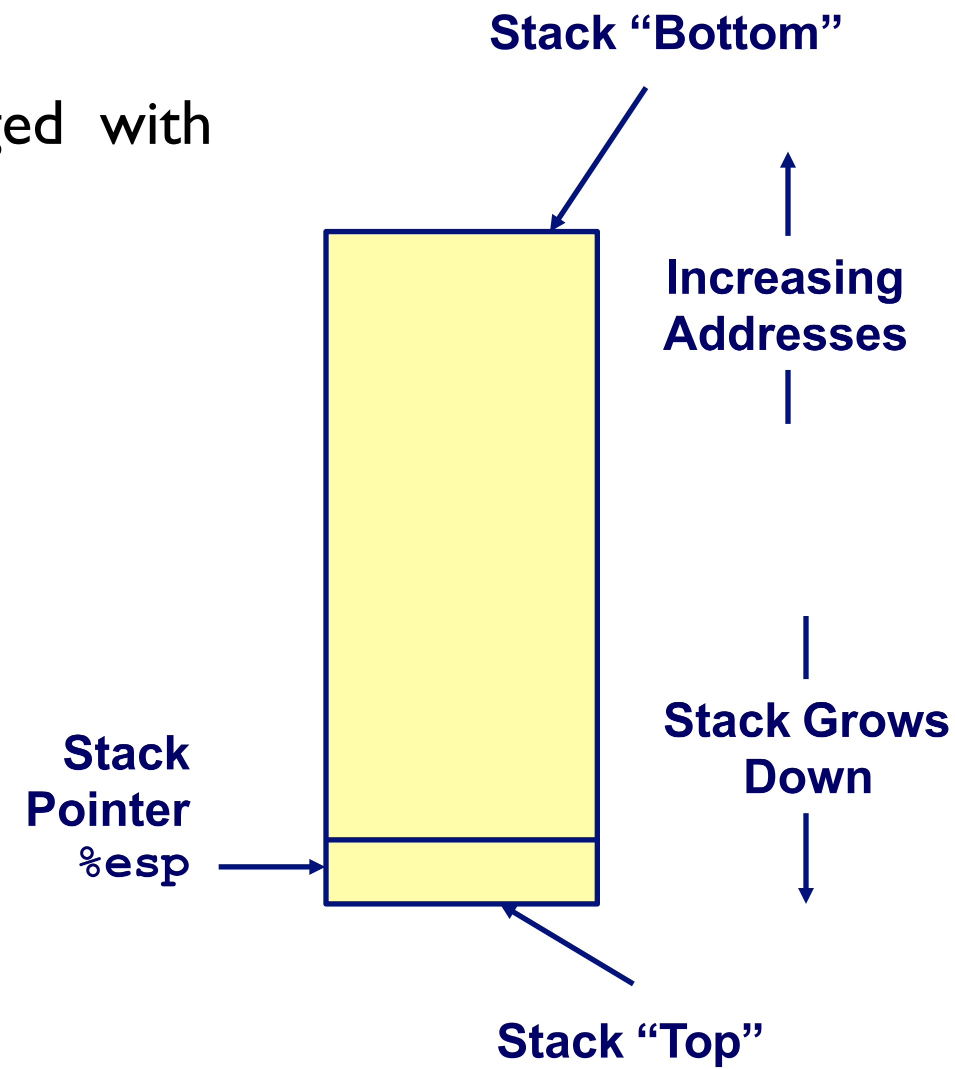
Recall: Memory Segments

- Segments of an executable are laid out in memory
- An application/program's memory has 4 segments
  - Text: instructions of the program
  - Data: global and static data
  - Heap: dynamic allocation
  - Stack: function calls and local data
- Heap and stack Grow dynamically
  - Heap grows up
  - Stack grows down



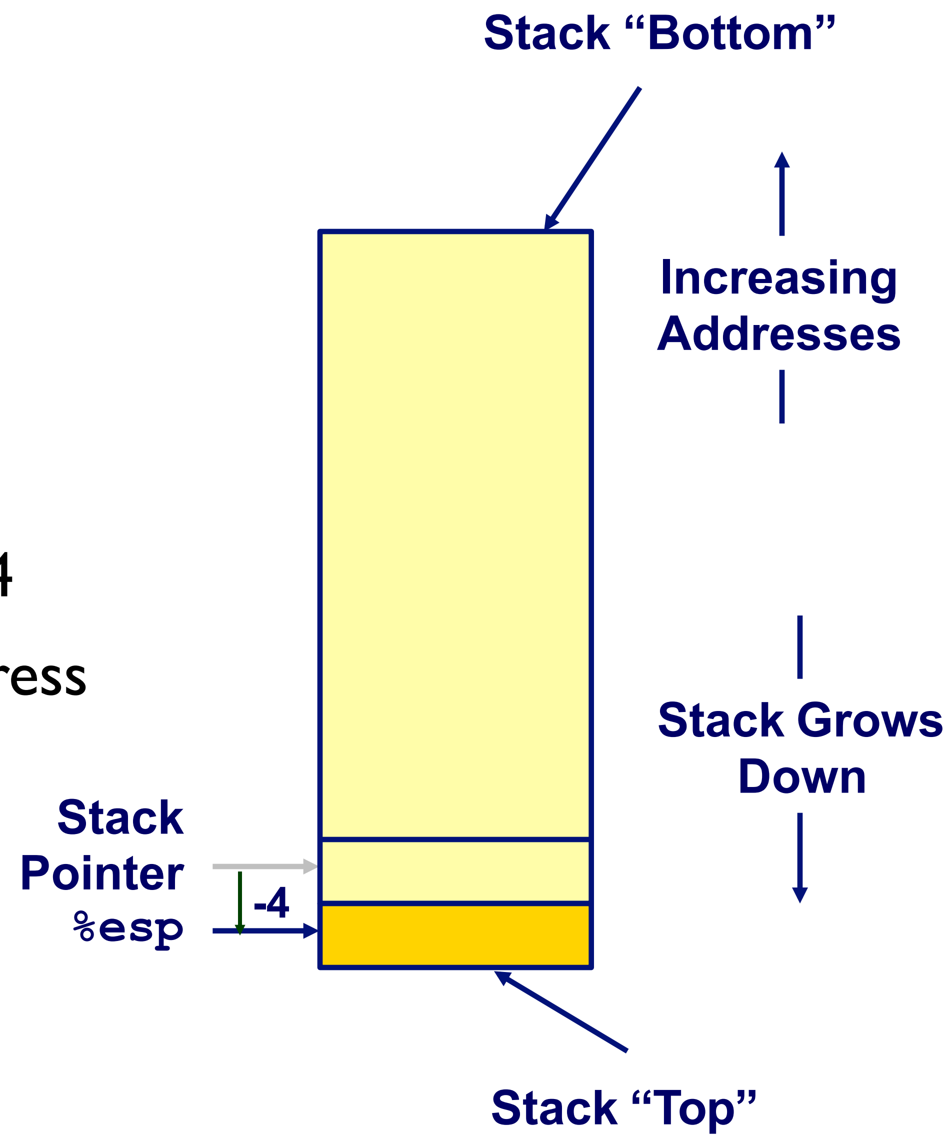
x86 Stack

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



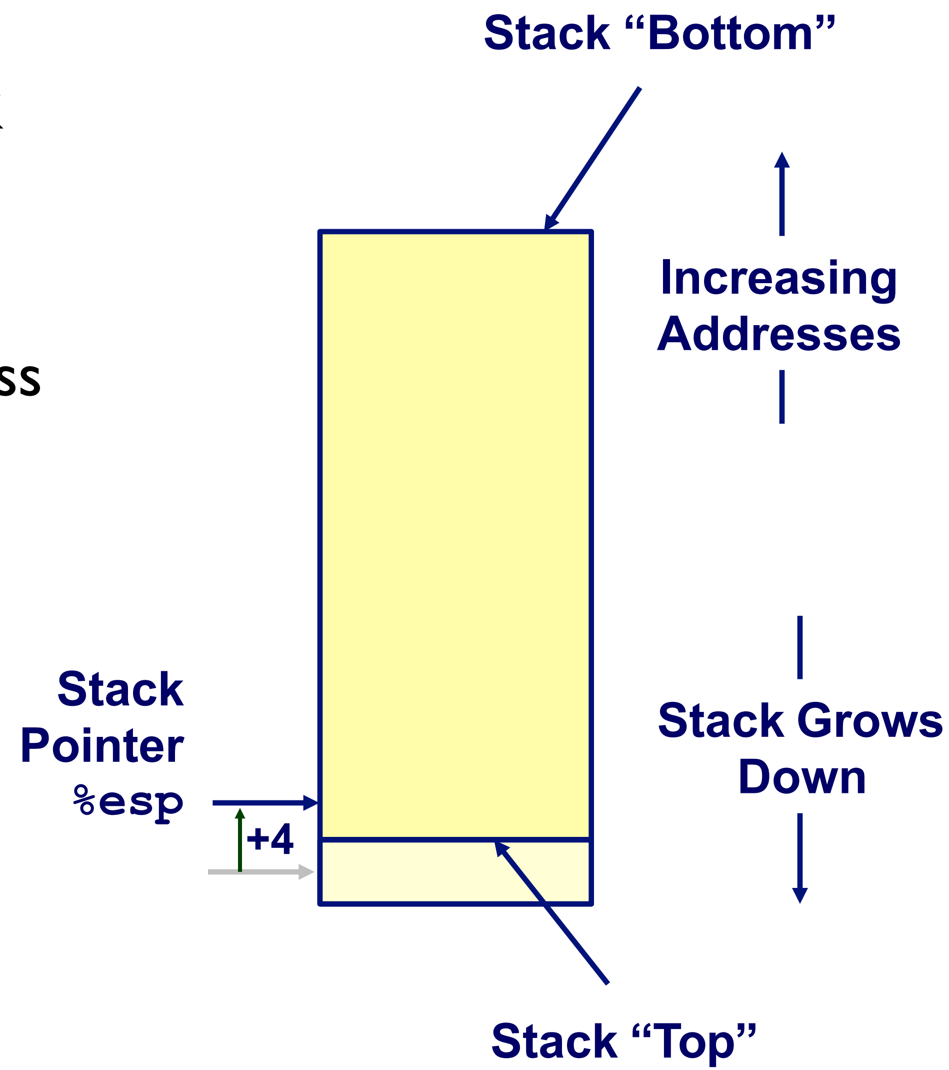
x86 Stack: Pushing

- Pushing to the stack
- `pushl` *Src*
- What it does:
  - Fetch operand at *Src*
  - Decrement `%esp` by 4
  - Write operand at address given by `%esp`

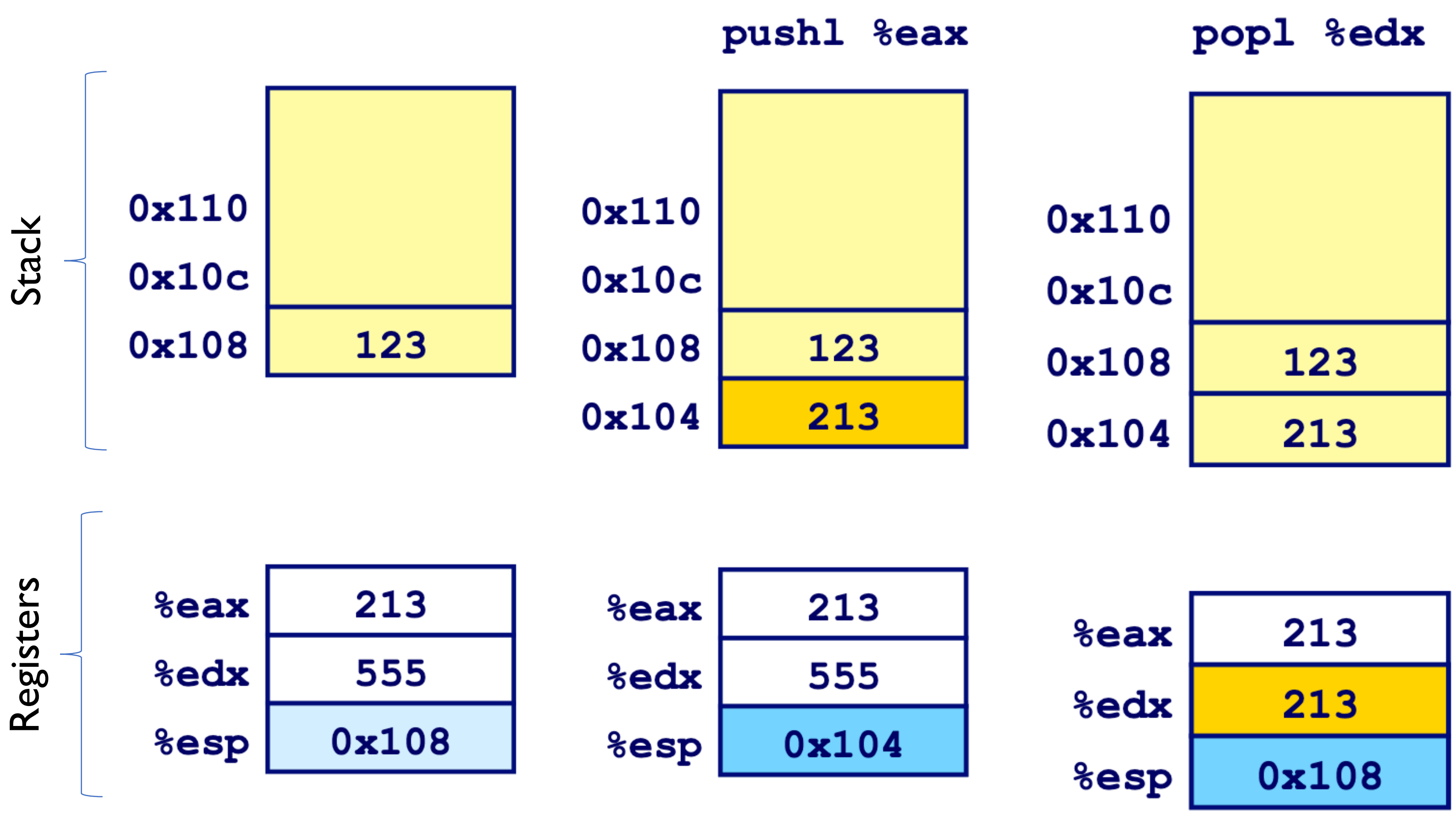


x86 Stack: Popping

- Popping from the stack
- `popl` *Dest*
- What it does
  - Read operand at address given by `%esp`
  - Increment `%esp` by 4
  - Write to *Dest*



Push and Pop Example



Procedure Control Flow

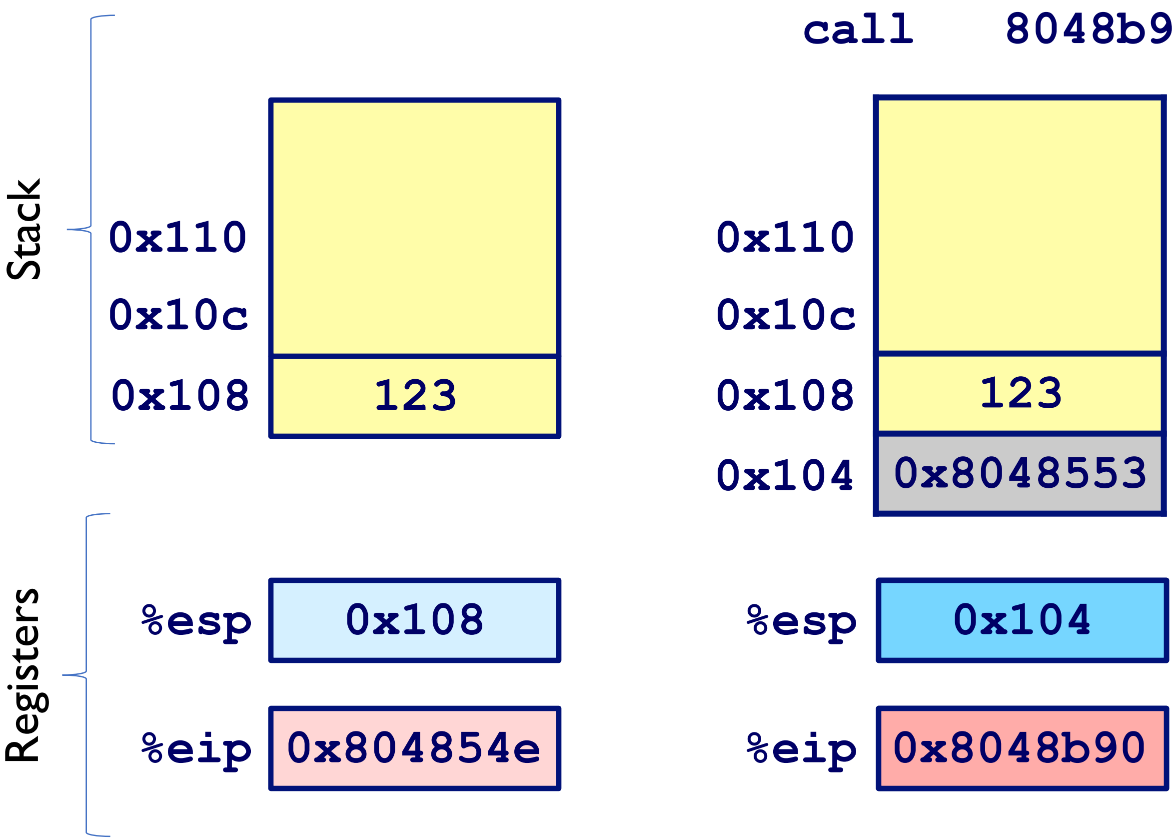
- Use stack to support procedure call and return
- A procedure call involves passing data and control from one part of a program to another
- Procedure call:
  - `call` *`label`*
  - Pushes return address on stack, then jump to *`label`*
- The return address is the address of instruction beyond `call`
- Example:

```
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

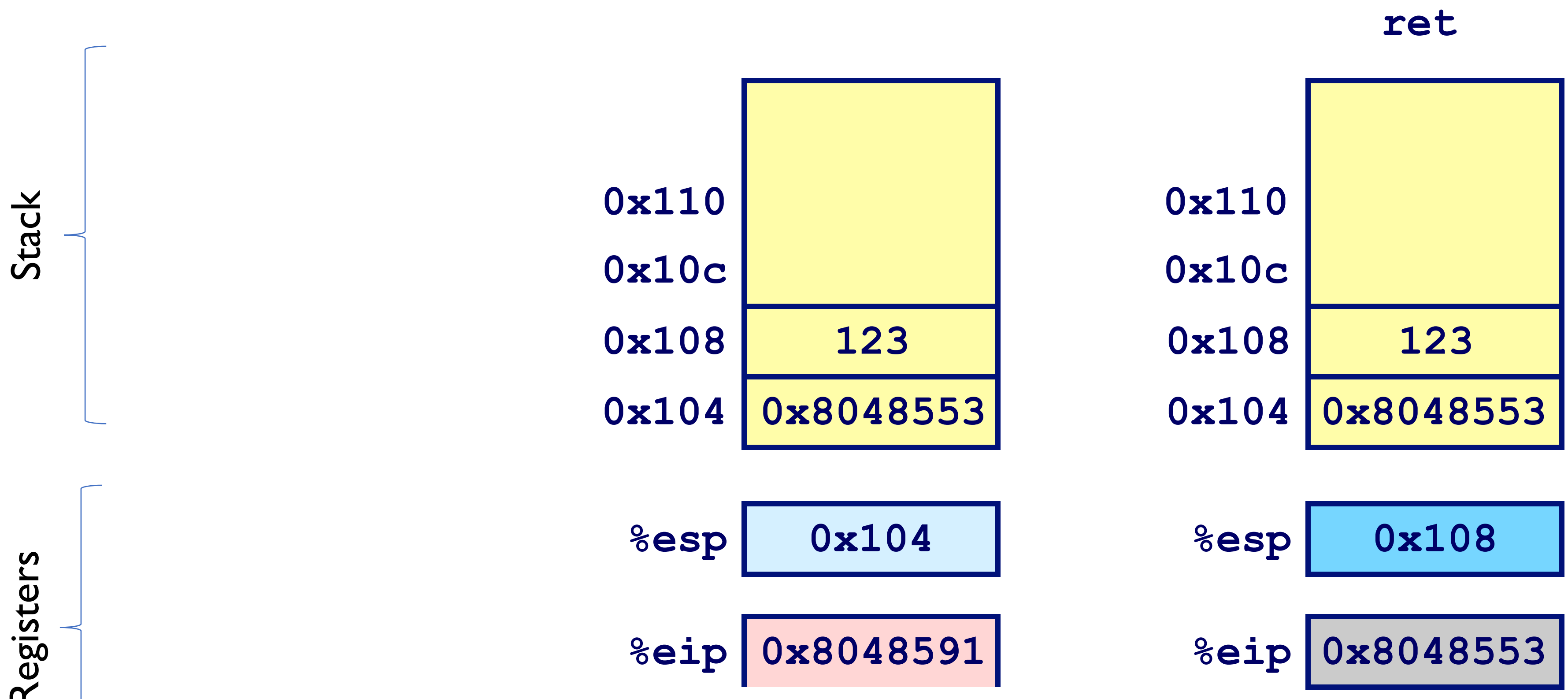
```
804854e:   e8 3d 06 00 00        call    8048b90 <main>
8048553:   50                    pushl   %eax
```



%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

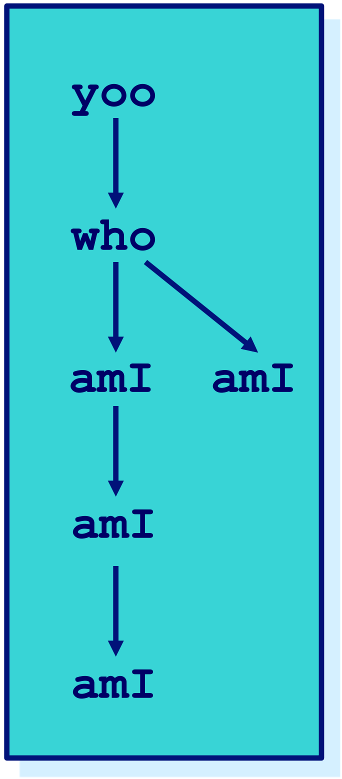
```
yoo(…)
{
    •
    •
    who();
    •
    •
}
```

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

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

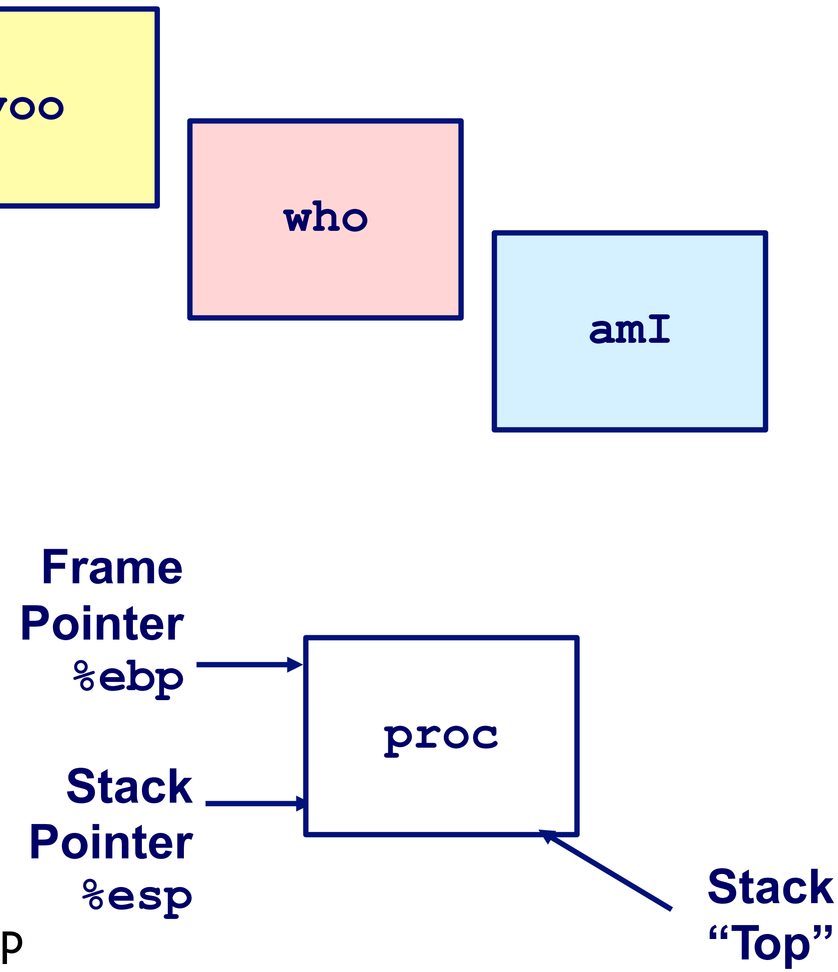
- Procedure `amI` recursive

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

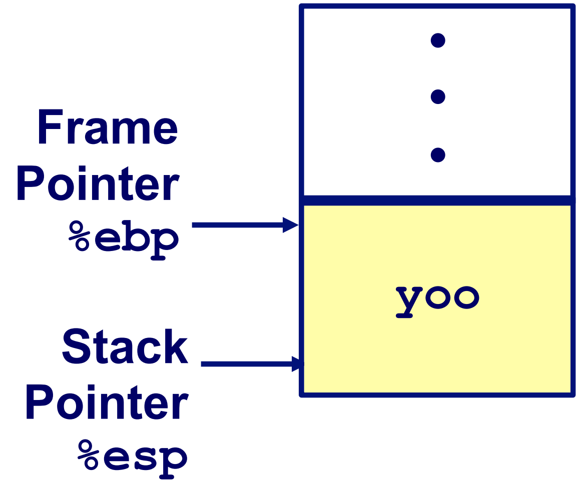


Stack Operation

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

Call Chain

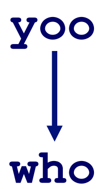
`yoo`

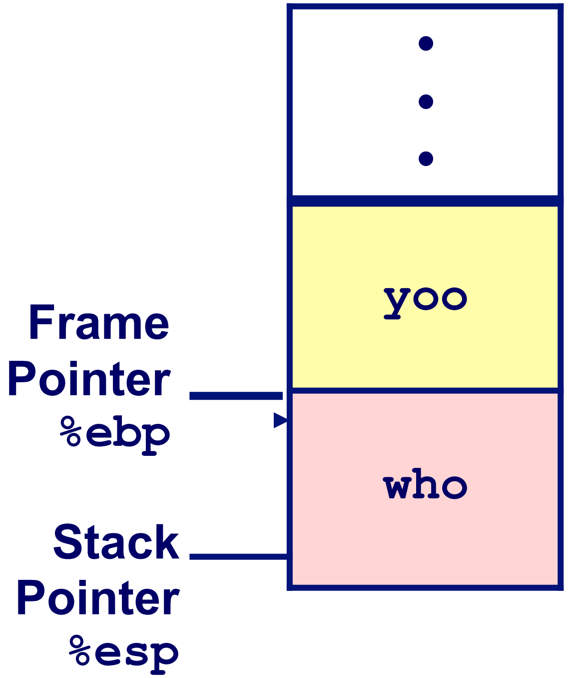


Stack Operation

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

Call Chain

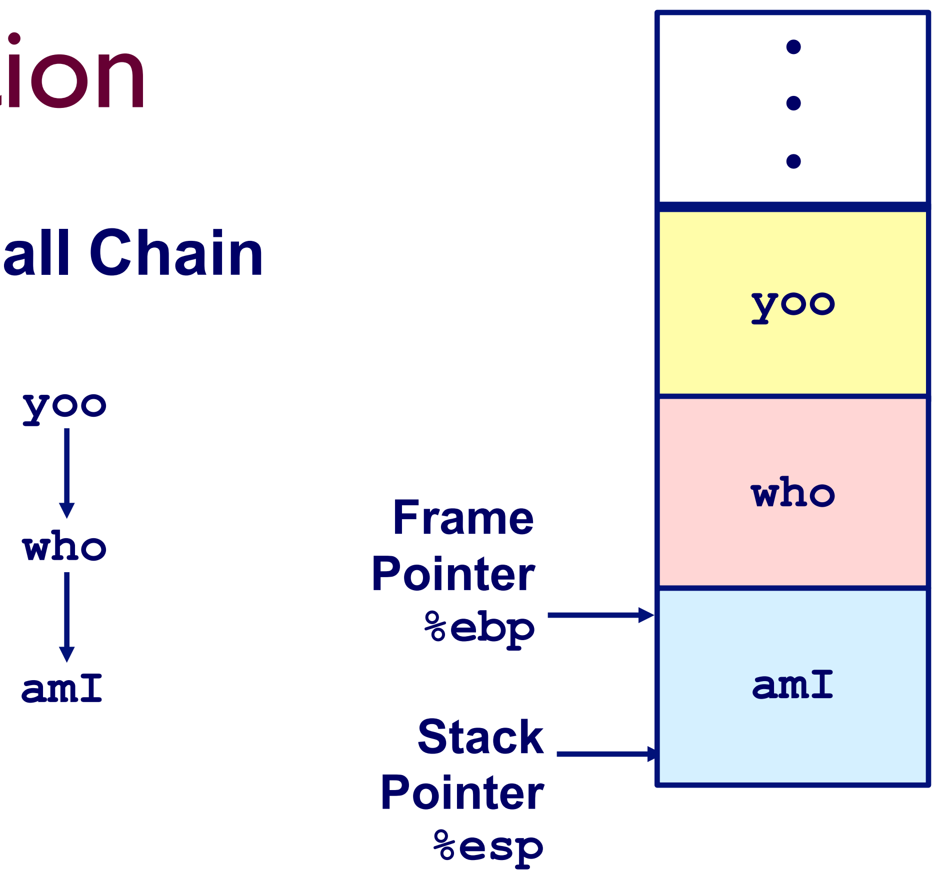




Stack Operation

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

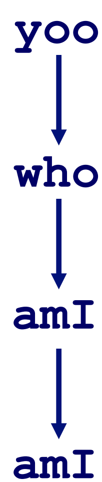
Call Chain

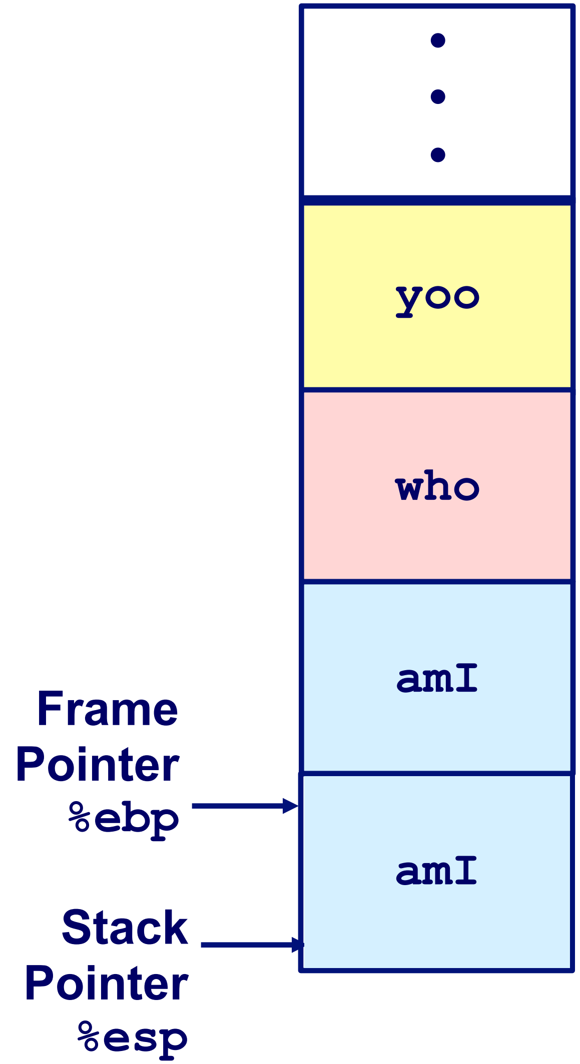


Stack Operation

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

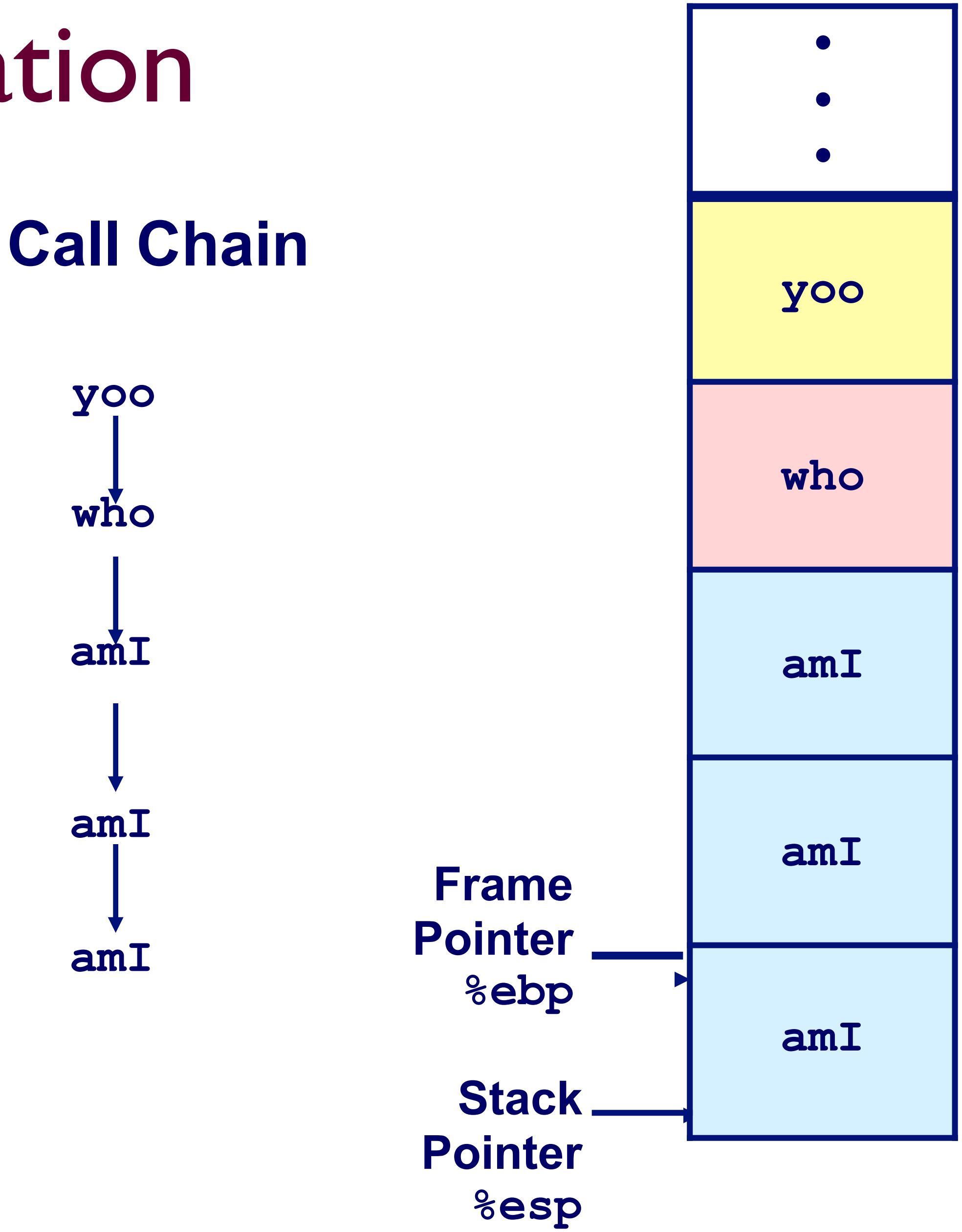
Call Chain





Stack Operation

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

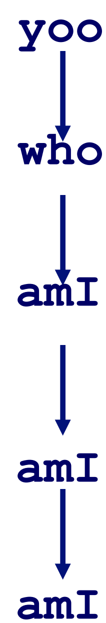


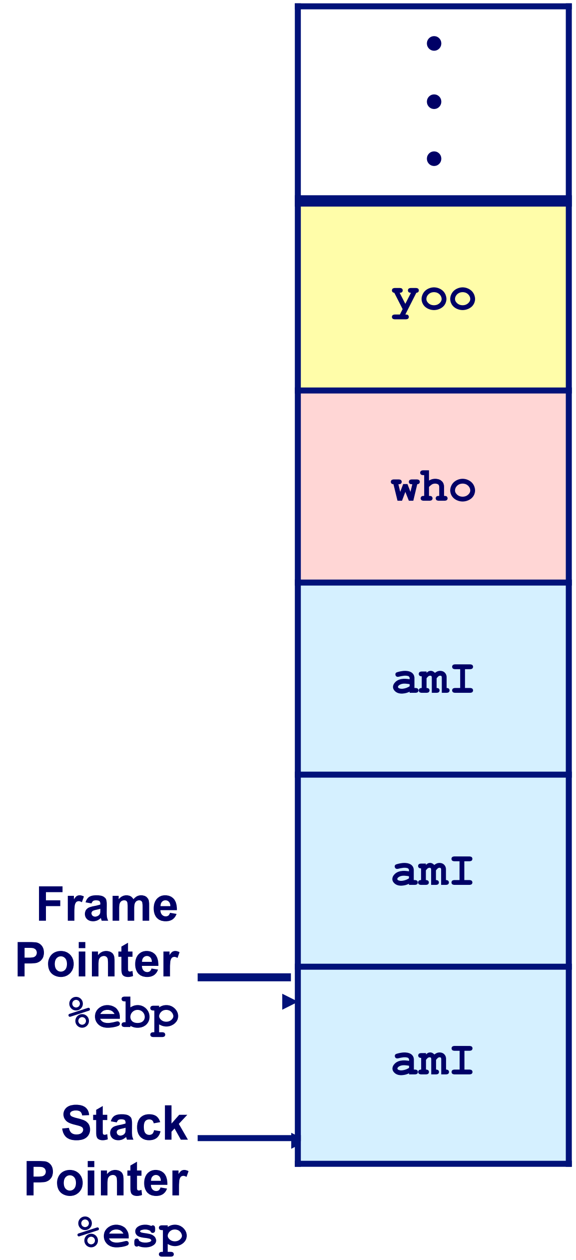
Stack Operation

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

Lets say it ends recursive calling here and start to return

Call Chain

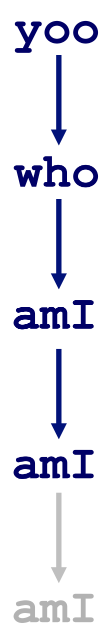


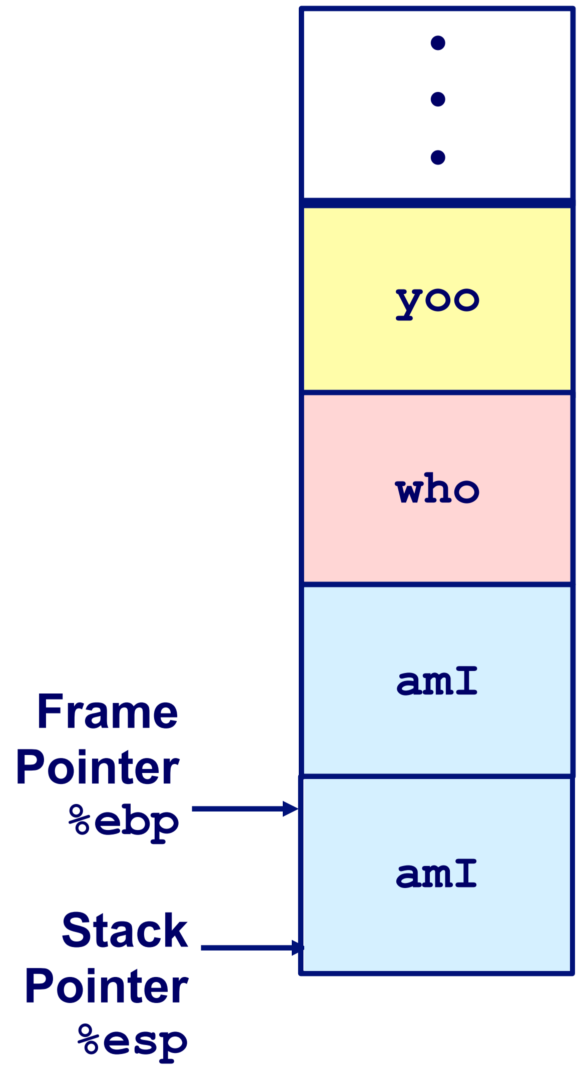


Stack Operation

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

Call Chain

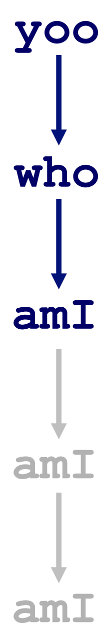


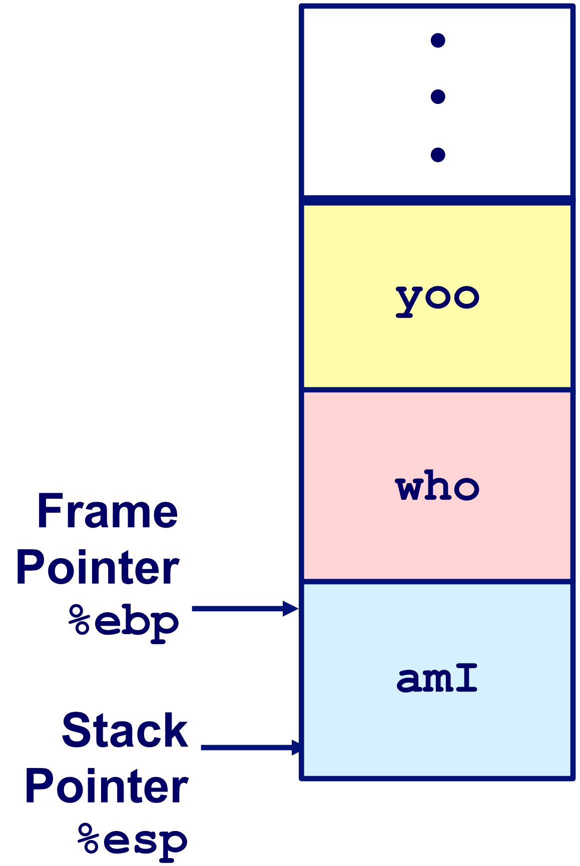


Stack Operation

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

Call Chain

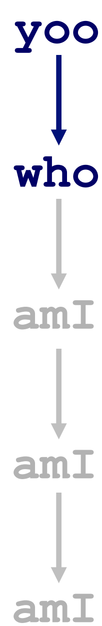


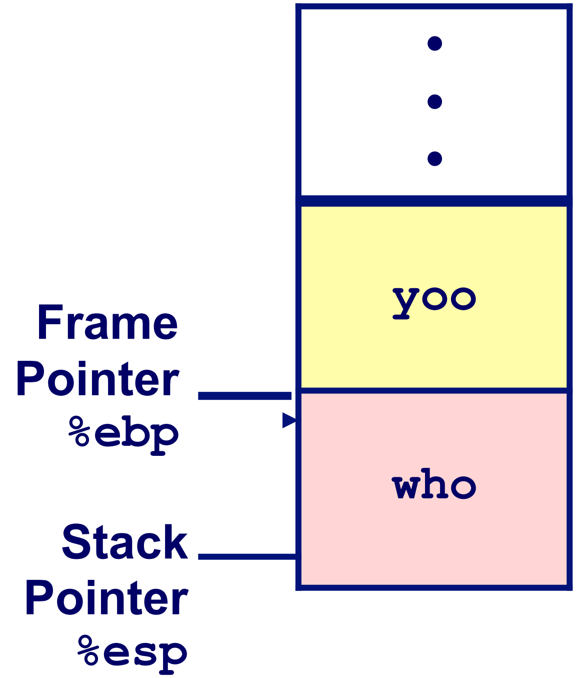


Stack Operation

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

Call Chain

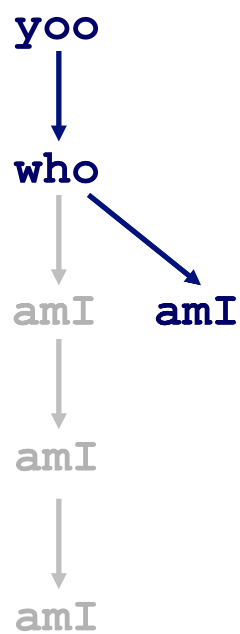


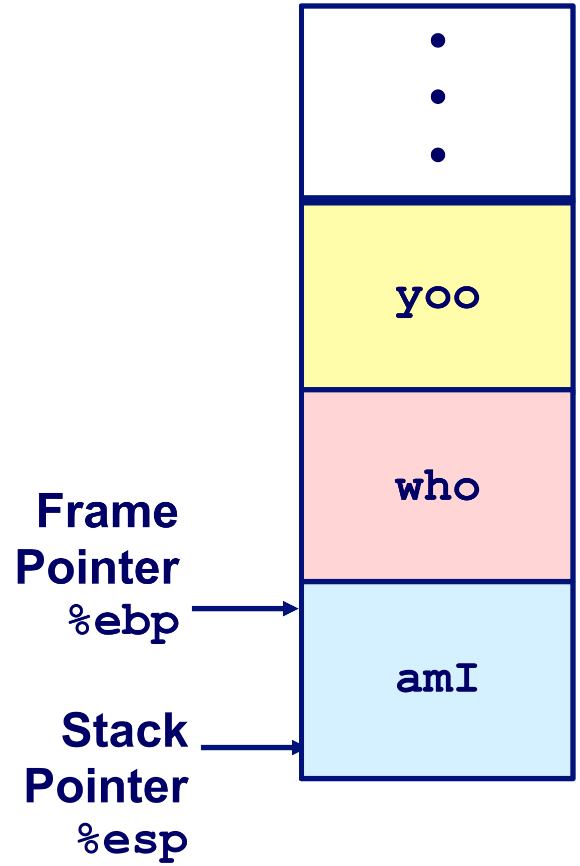


Stack Operation

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

Call Chain



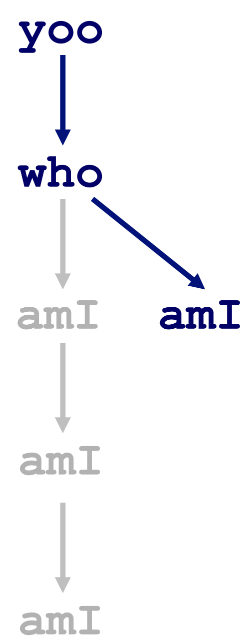


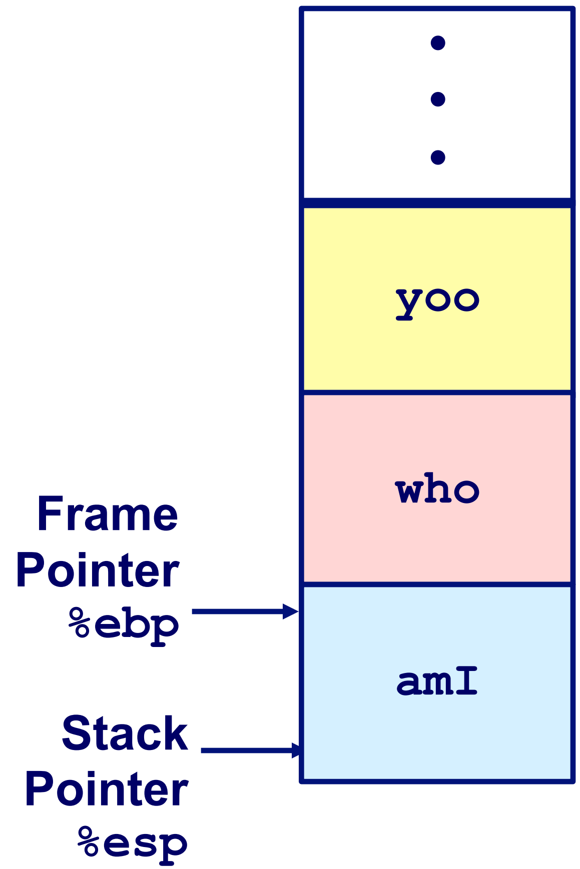
Stack Operation

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

Lets say it amI() avoids recursive calling here and starts to return

Call Chain

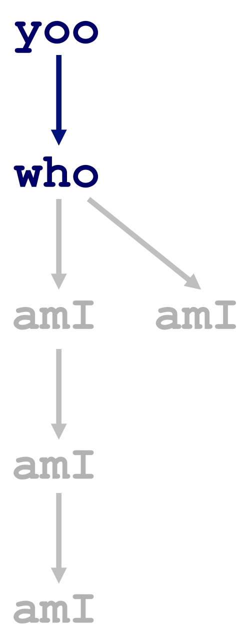


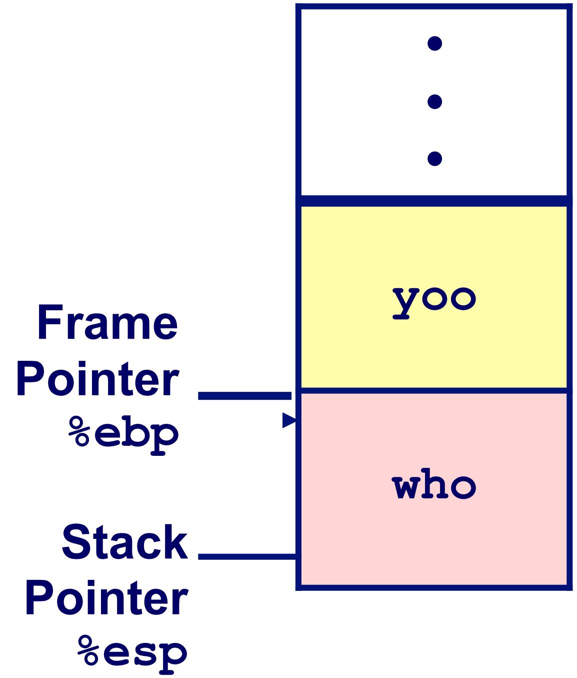


Stack Operation

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

Call Chain

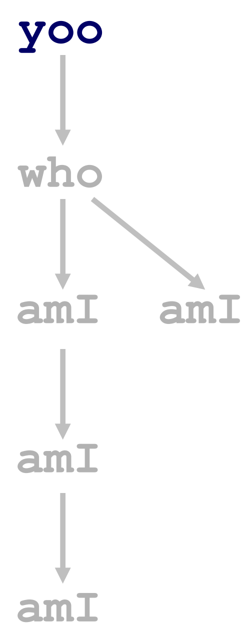


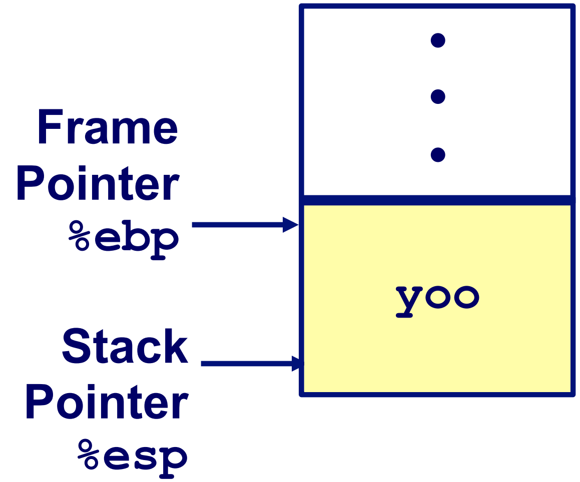


Stack Operation

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

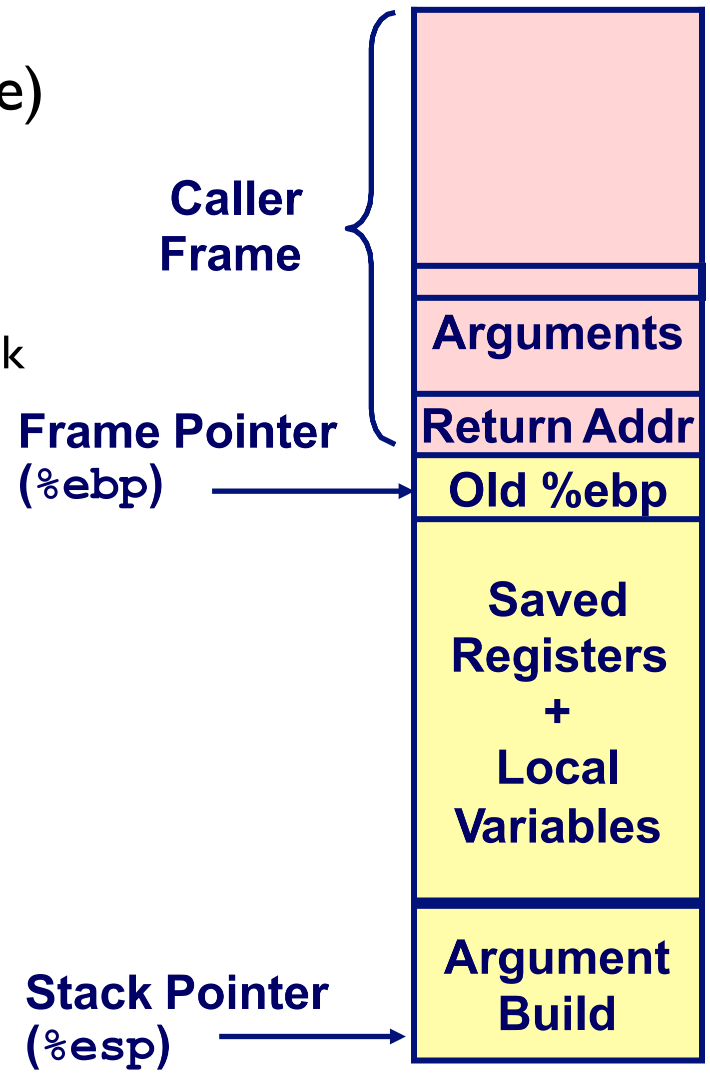
Call Chain

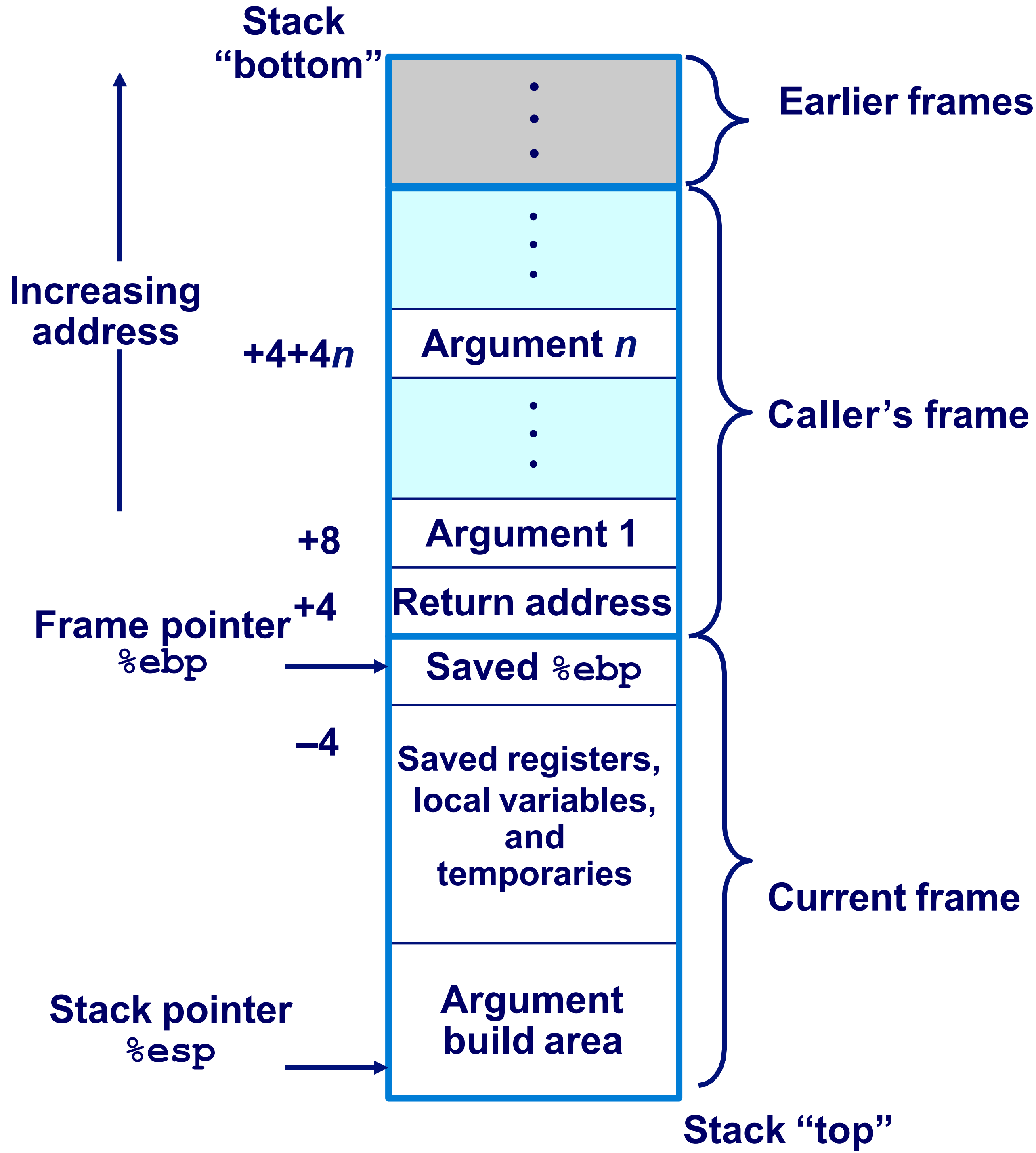




x86 Linux Stack Frame

- Current Stack Frame (Callee's frame)
  - From "Top" to "Bottom"
  - Parameters for function about to call
    - "Argument build"
    - Parameters are pushed on to the stack right to left
  - 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





Stack
"bottom"
Earlier frames
Increasing address
+4+4n
Argument n
Caller's frame
+8
Argument 1
+4
Return address
Frame pointer
%ebp
Saved %ebp
–4
Saved registers, local variables, and temporaries
Current frame
Stack pointer
%esp
Argument build area
Stack "top"

Revisiting `swap`

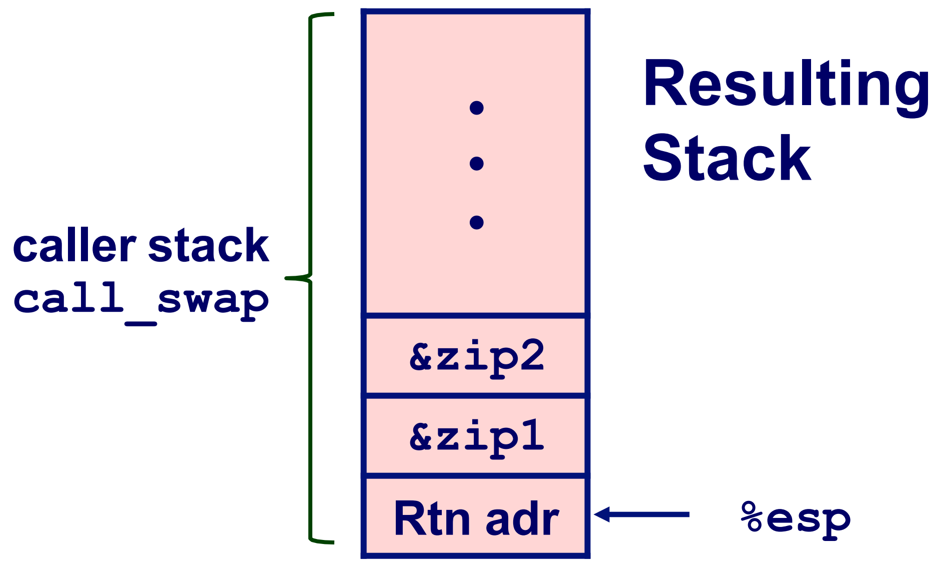
```
int zip1 = 15213;
int zip2 = 91125;

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
   • • •
```

Put return address into stack and jump to label



Revisiting `swap` in x86

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

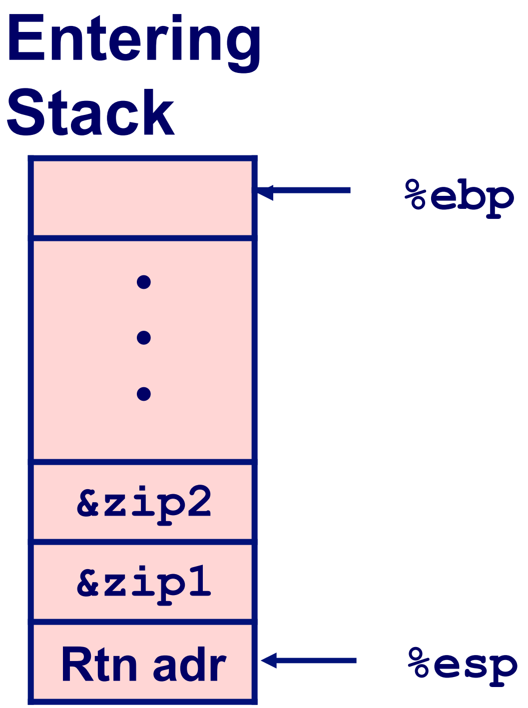
```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl%ebx

    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
```

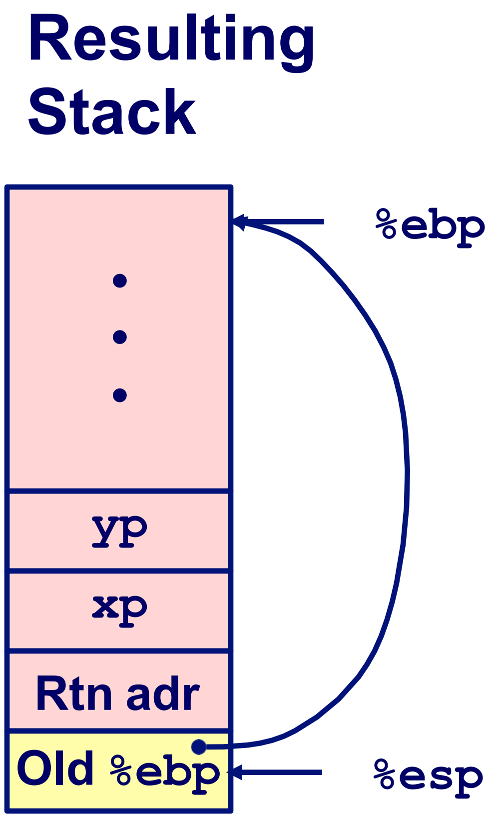
Set Up

Body

Finish

`swap` Setup #1



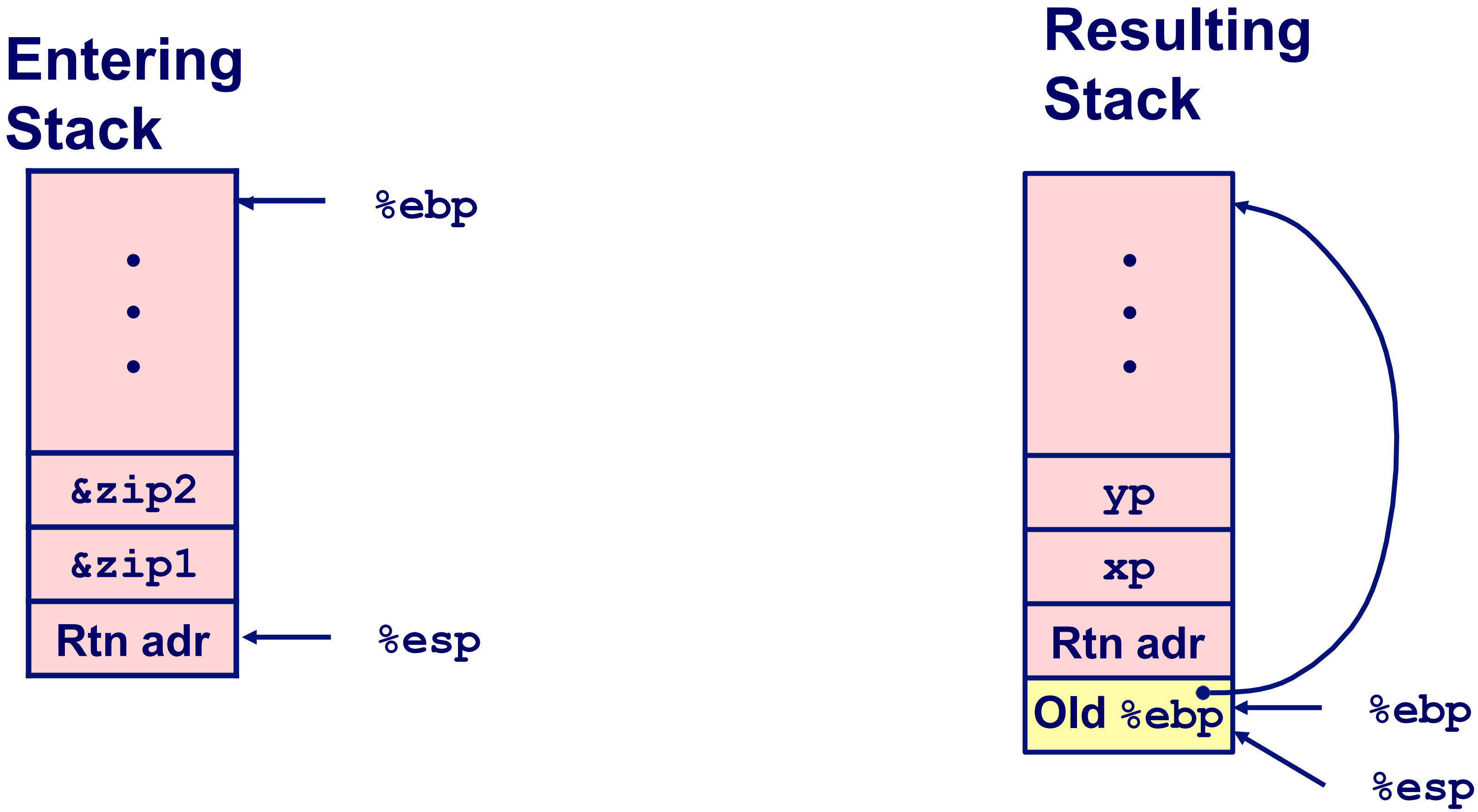



```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

Observation

- Save `%ebp`

swap Setup #2

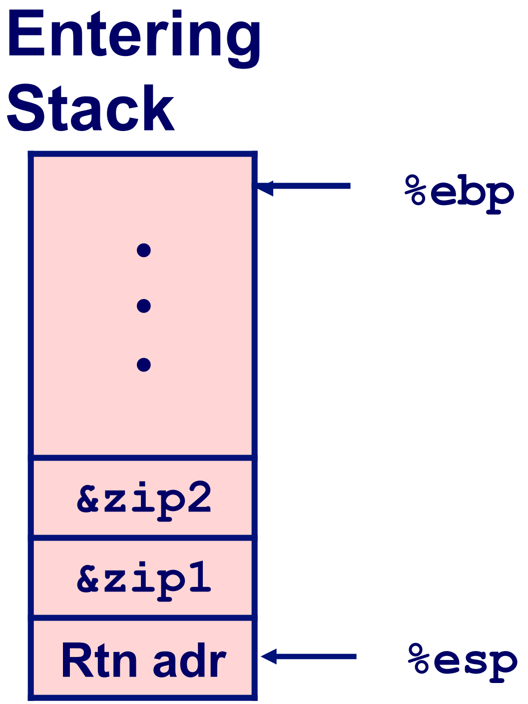


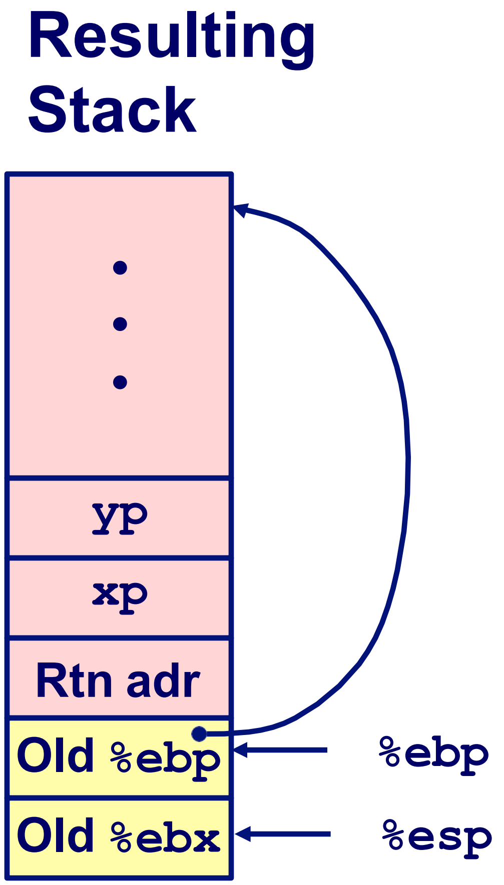

```
swap:
    pushl %ebp
    movl  %esp,%ebp
    pushl %ebx
```

Observation

- Saved `%ebp`

swap Setup #3



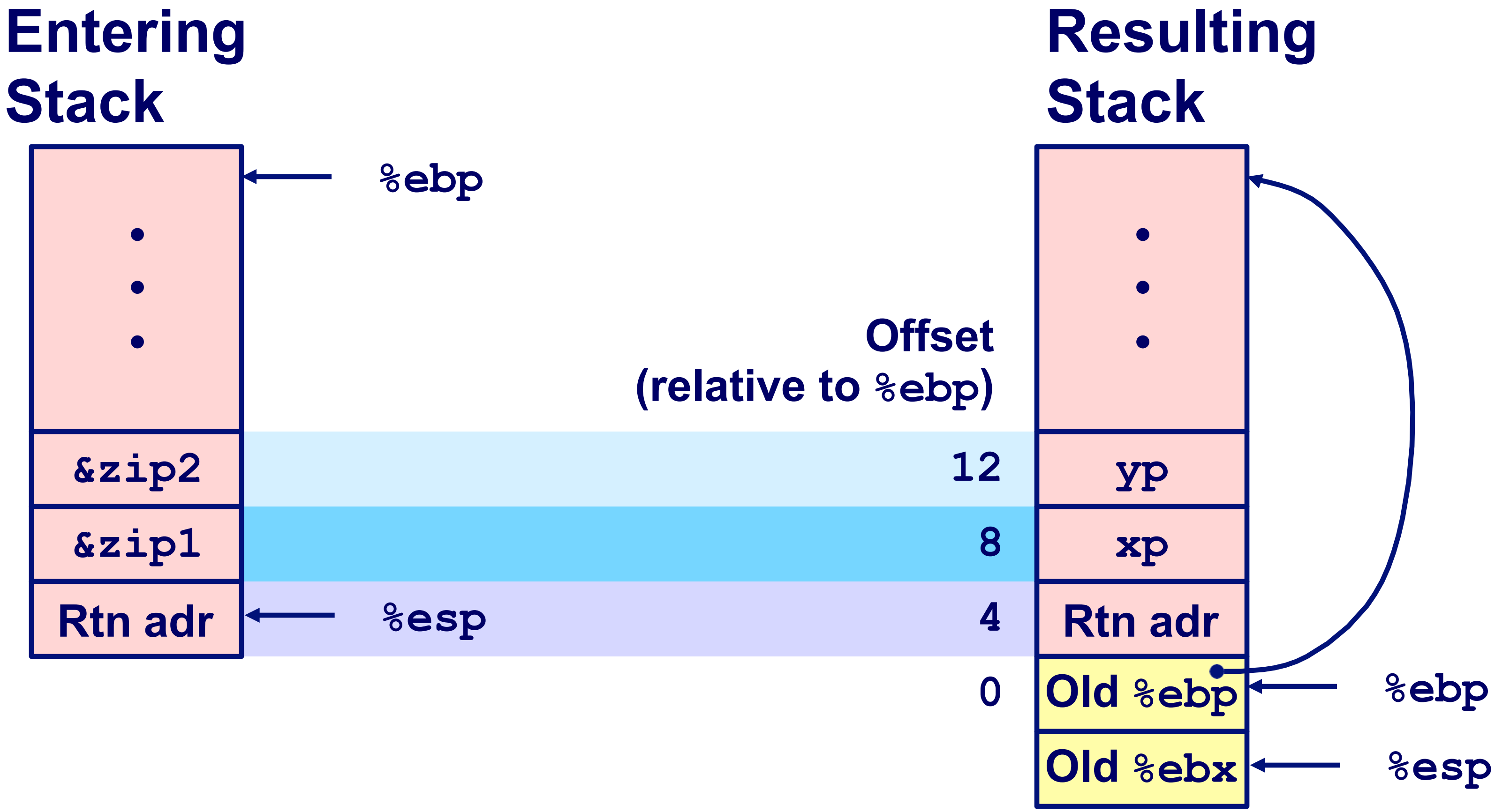



```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

Observation

- Save register `%ebx`

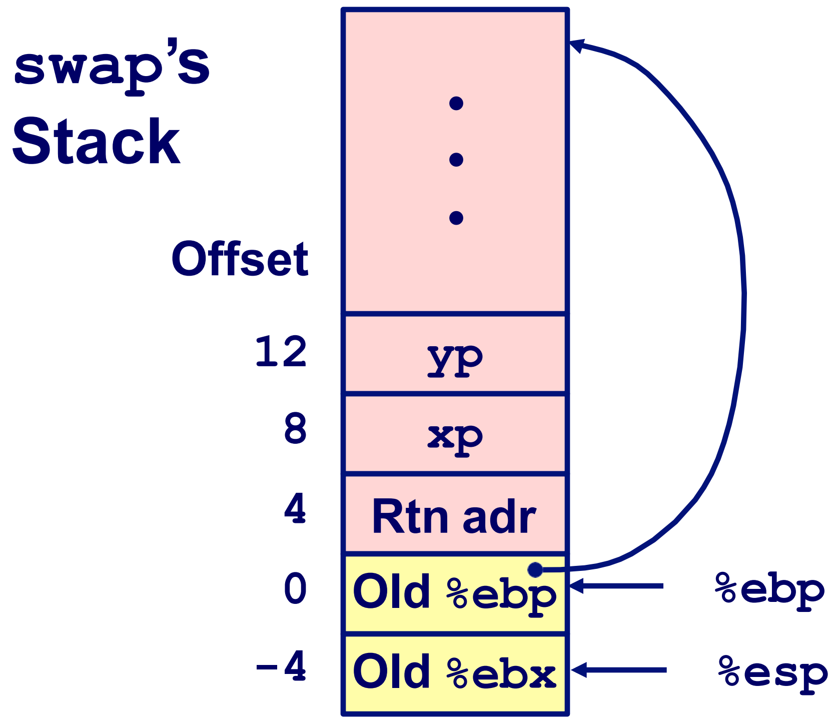
Effect of `swap` Setup

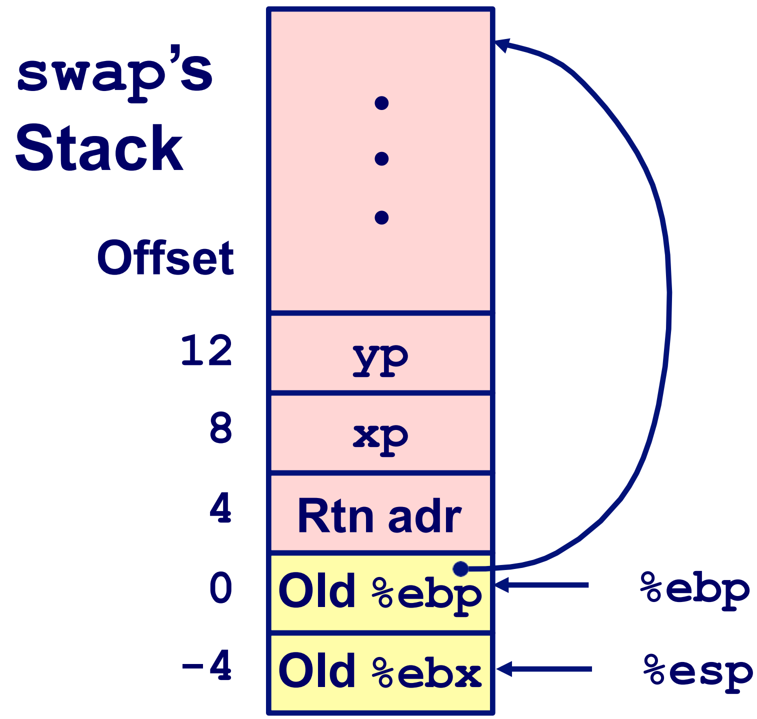



```
movl 12(%ebp),%ecx # get yp
movl 8(%ebp),%edx  # get xp
. . .
```

Body

`swap` Finish #1



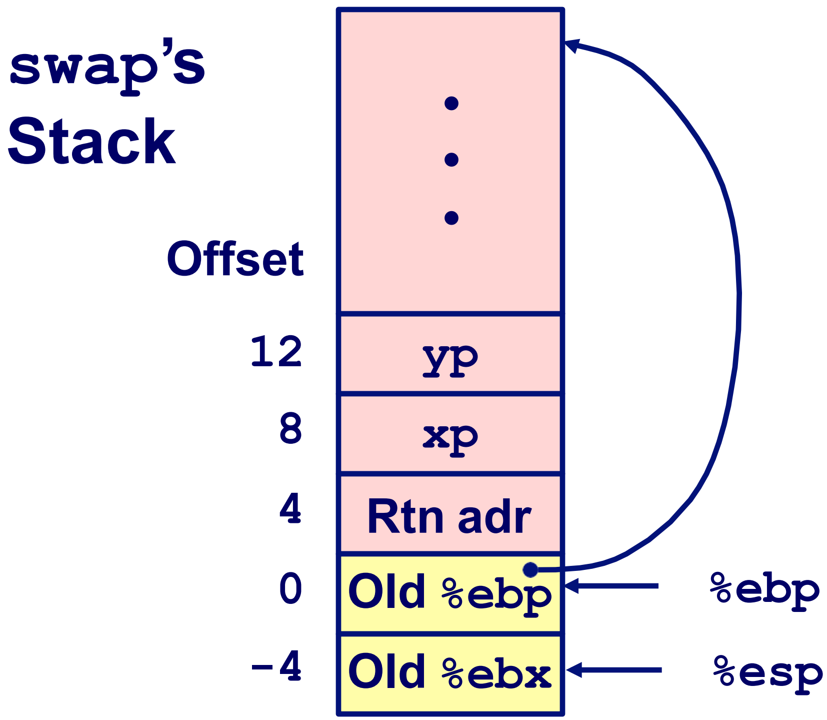


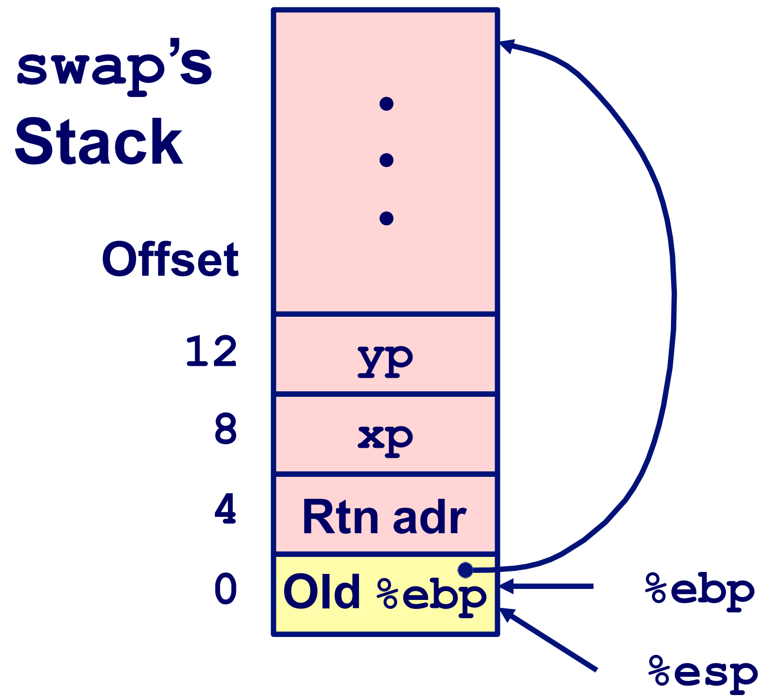

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Observation

- Saved & restored register `%ebx`

`swap` Finish #2



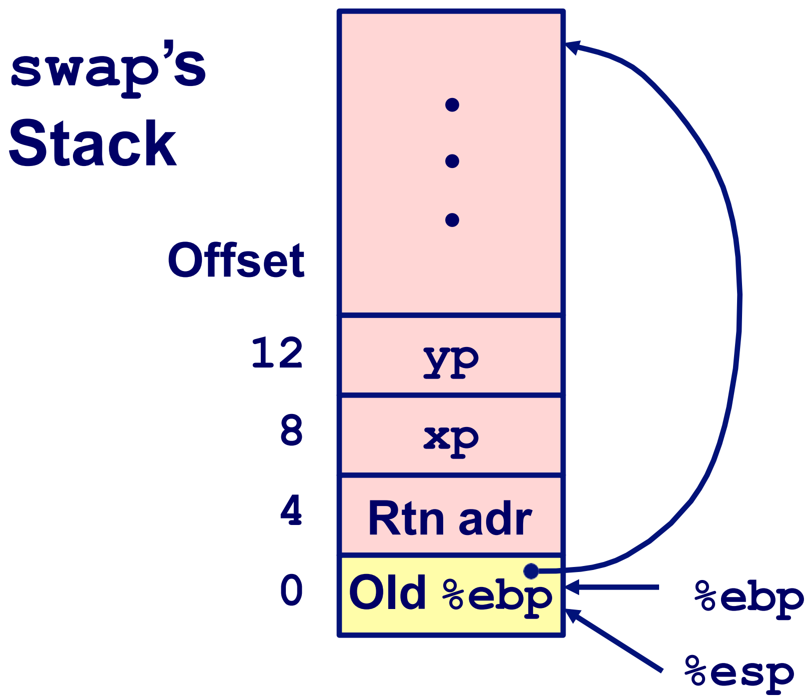


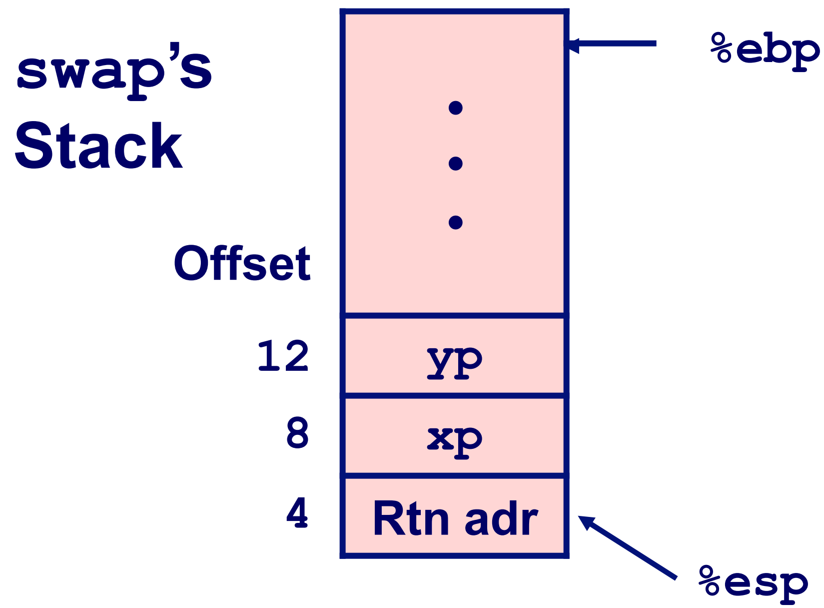

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Observation

- Set `%esp` to `%ebp` after restoring any registers

swap Finish #3



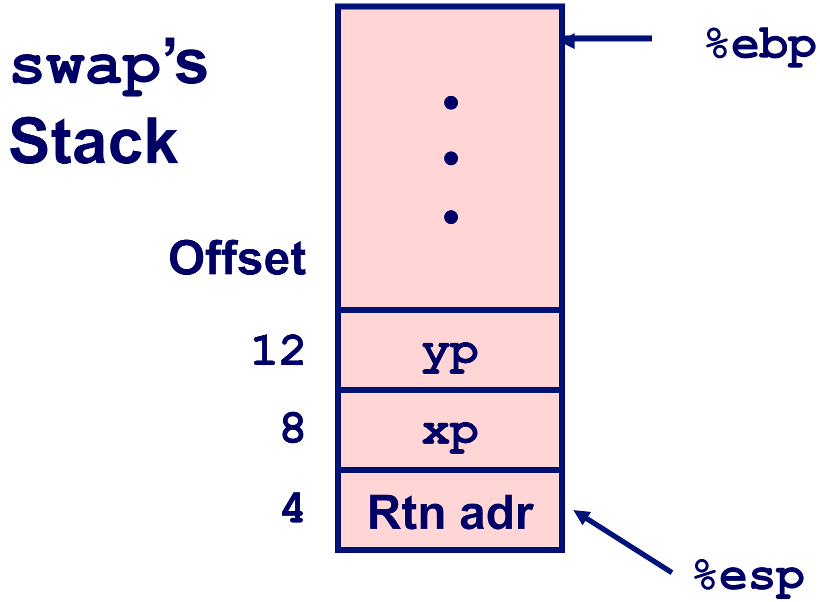


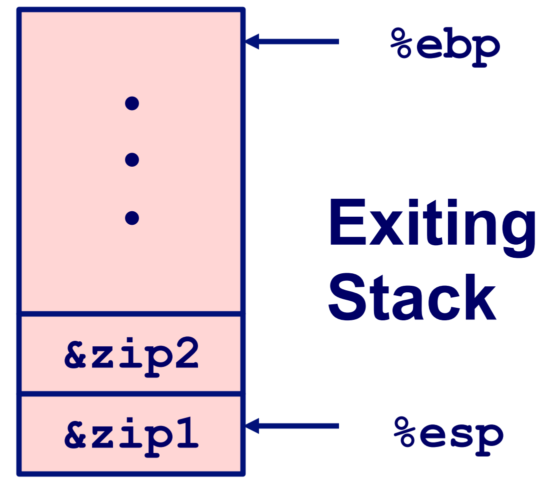

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Observation

- Restore old `%ebp`

`swap` Finish #4






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

Overall 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 a 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
```

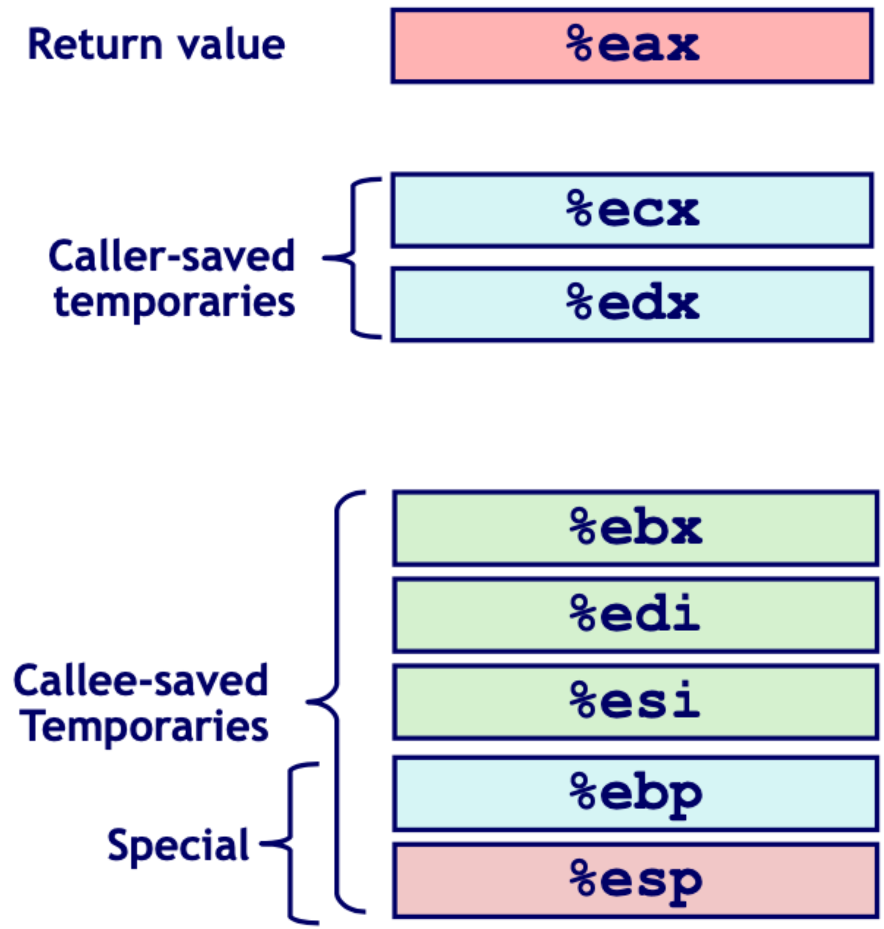
- Contents of register `%edx` overwritten by `who()`

Register Saving Conventions

- When procedure `yoo()` calls who:
  - `yoo()` is the *caller*, `who()` is the *callee*
- Can a 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

x86 Linux Register Usage

- `%eax`
  - Used to store return value
  - Caller saved
  - Can be modified by procedure
- `%ecx, %edx`
  - Caller saved
  - Can be modified by procedure
- `%ebx, %edi, %esi`
  - Callee saved
  - Callee must save & restore
- `%ebp`
  - Callee saved
  - Callee must save & restore
  - May be used as frame pointer
- `%esp`
  - Special form of callee save
  - Restored to original value uipon exit of procedure



Recursive Factorial

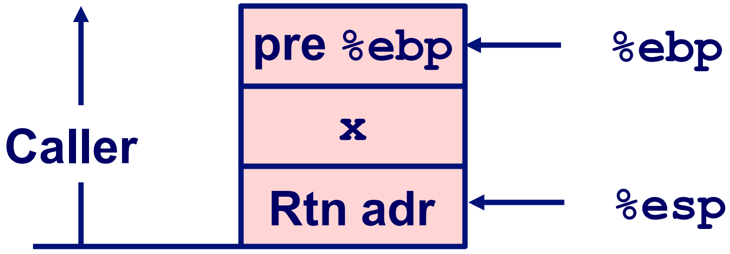
```
int rfact(int x)
{
  int rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1);
  return rval * x;
}
```

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

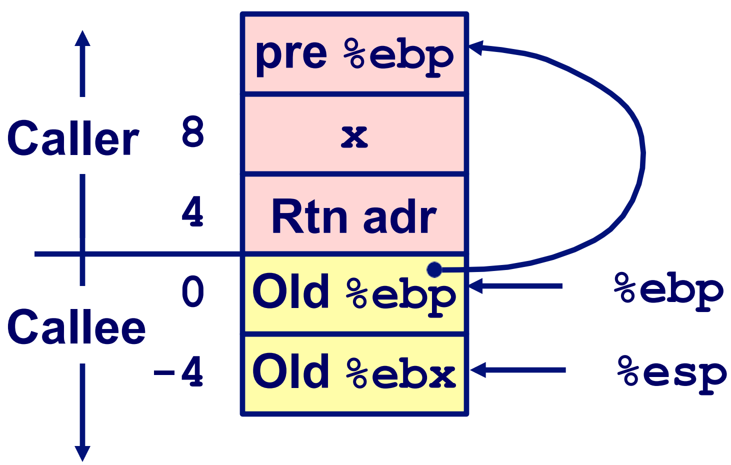
```
.globl rfact
    .type
rfact,@function
rfact:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 8(%ebp),%ebx
    cmpl$1,%ebx
    jle .L78
    leal -1(%ebx),%eax
    pushl %eax
    call rfact
    imull %ebx,%eax
    jmp .L79
    .align 4
.L78:
    movl $1,%eax
.L79:
    movl -4(%ebp),%ebx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

Rfact Stack Setup

Entering Stack




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



Rfact Body

Recursion

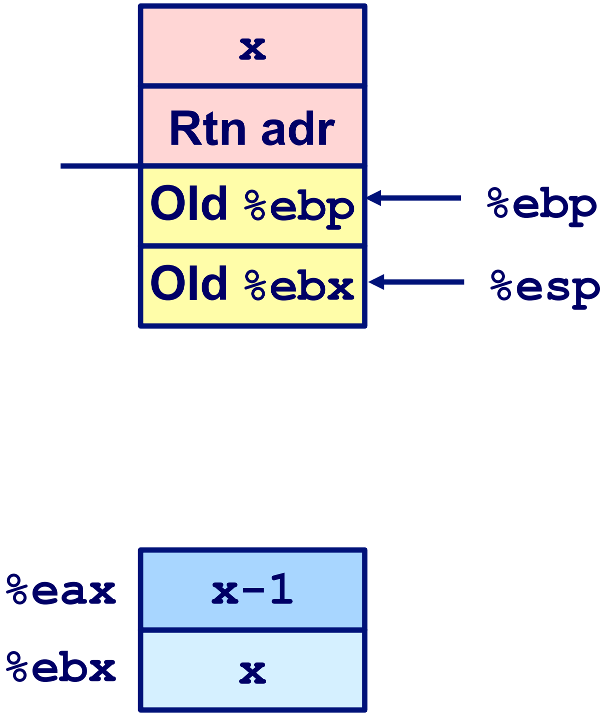
```
  movl 8(%ebp),%ebx   # ebx = x
  cmpl $1,%ebx        # Compare x : 1
  jle .L78            # If <= goto Term
  leal -1(%ebx),%eax  # eax = x-1
  pushl %eax          # Push x-1
  call rfact          # rfact(x-1)
  imull %ebx,%eax     # rval * x
  jmp .L79            # Goto done
.L78:            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

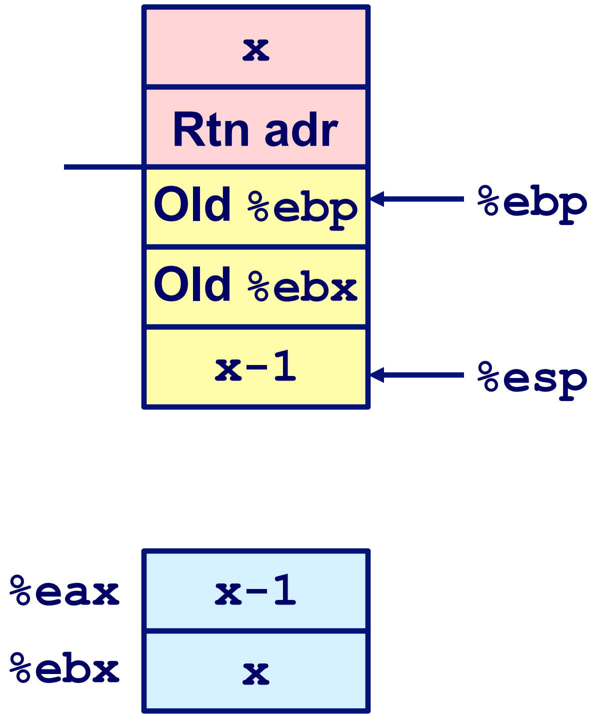
- Registers
  - `%ebx` Stored value of x
  - `%eax`
    - Temporary value of x-1
    - Returned value from rfact(x-1)
    - Returned value from this call

Rfact Recursion

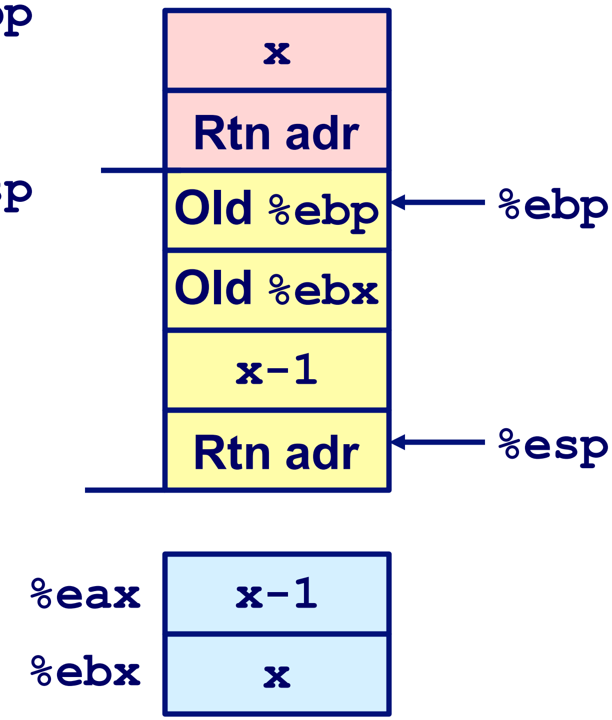
```
leal -1(%ebx),%eax
```




```
pushl %eax
```

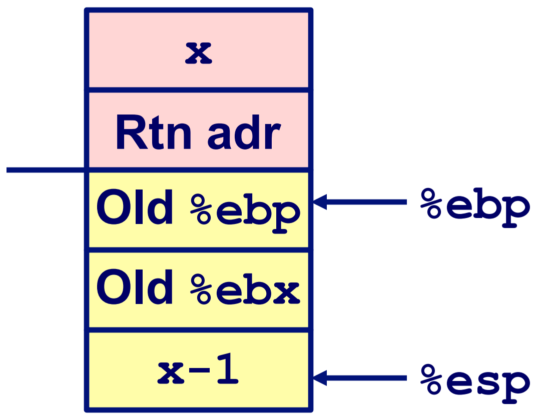



```
call rfact
```



Rfact Result

Return from Call

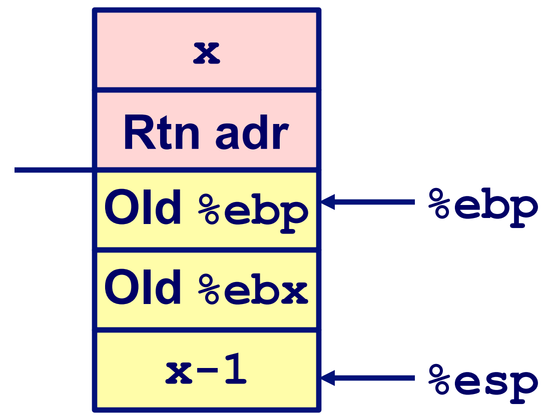


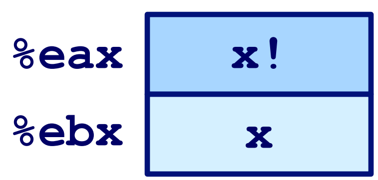
`%eax   (x-1)!`

`%ebx      x`

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

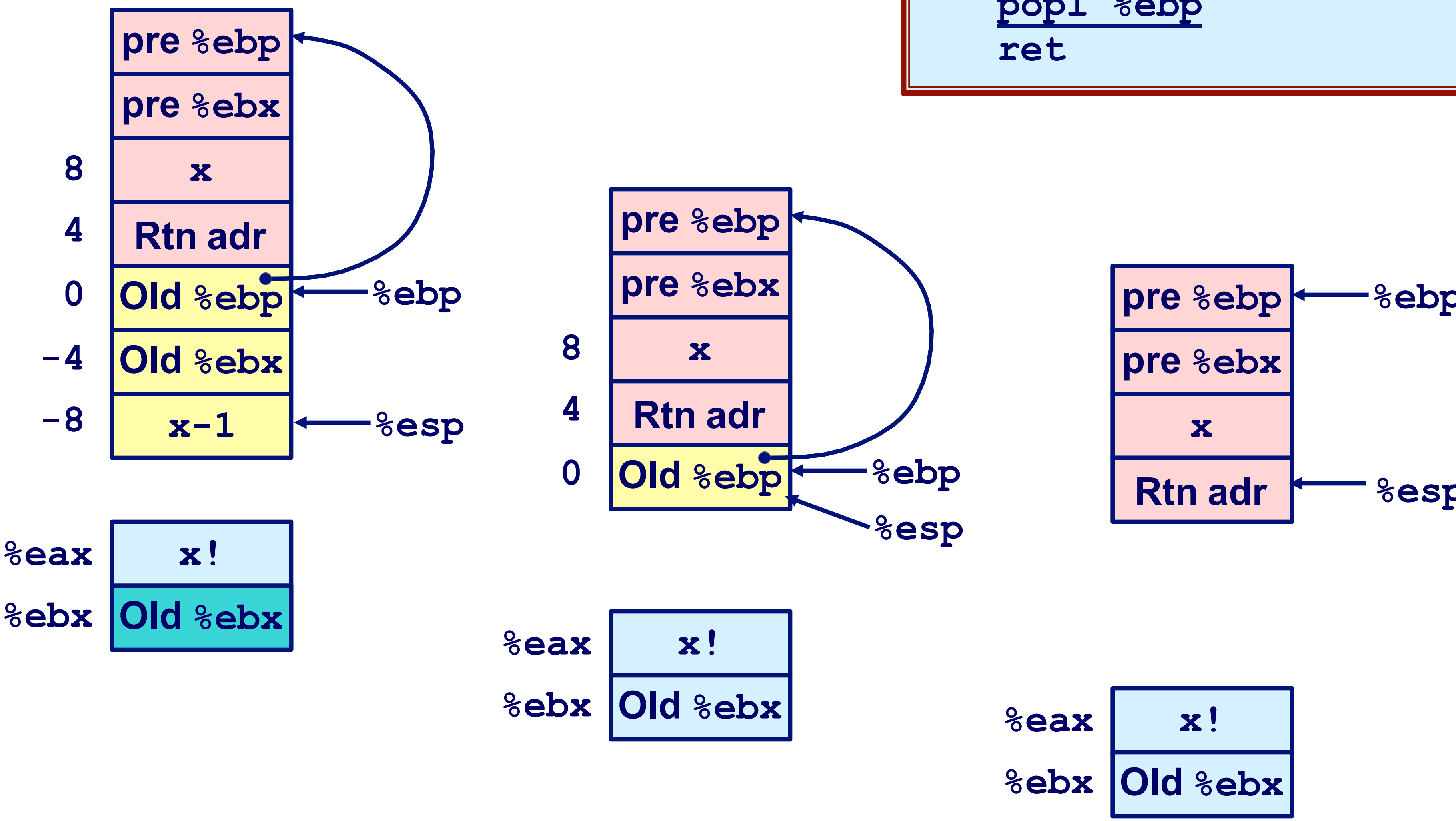
```
imull %ebx,%eax
```





Rfact Completion

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



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
- x86 Procedures Combination of Instructions + Conventions
  - Call / Ret instructions
  - Register usage conventions
    - Caller / Callee save
    - `%ebp` and `%esp`
  - Stack frame organization conventions