Machine-Level Programming II: Control Structures and Procedures

Fall 2012

Instructors:

Aykut & Erkut Erdem

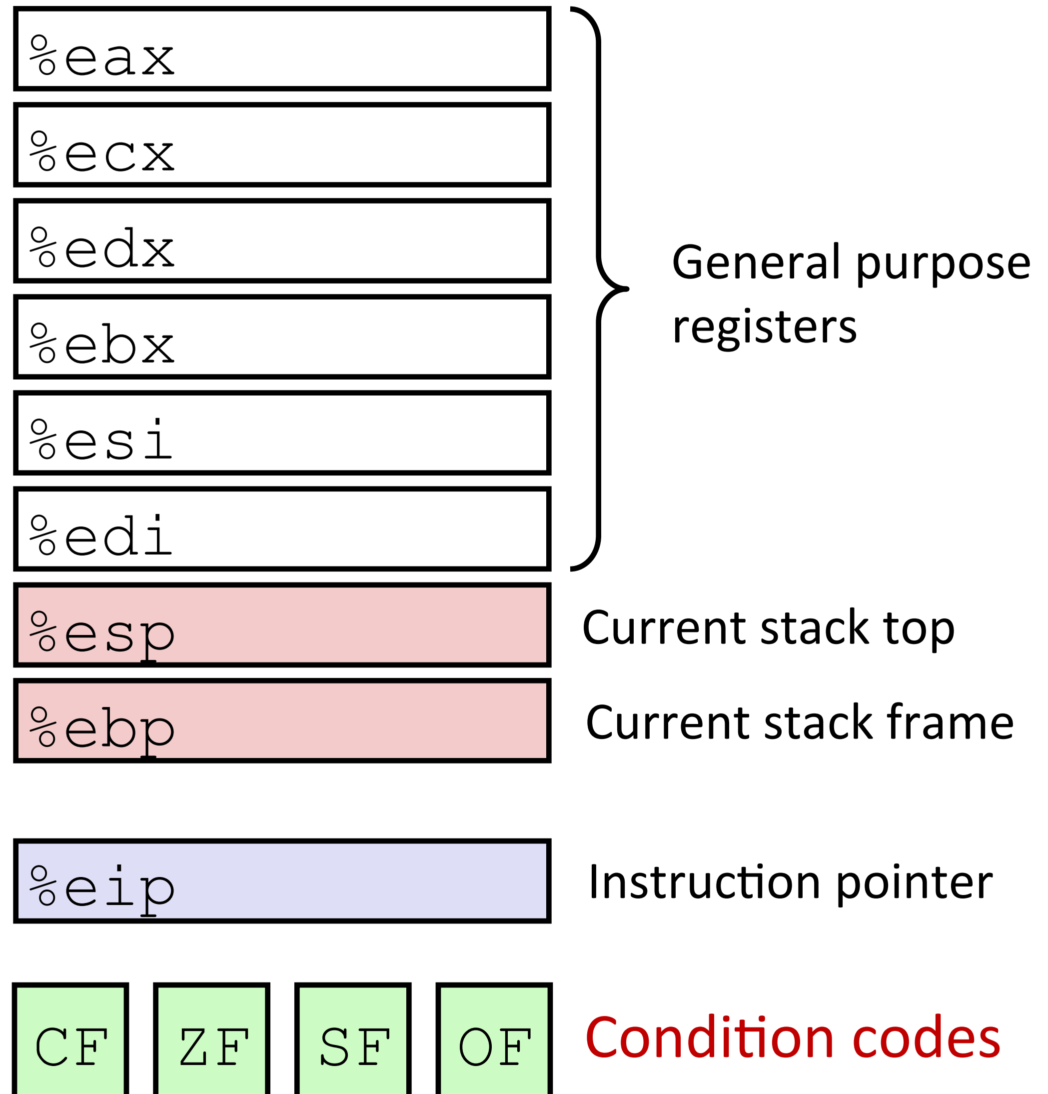
Acknowledgement: The course slides are adapted from the slides prepared by R.E. Bryant, D.R. O’Hallaron, G. Kesden and Markus Püschel of Carnegie-Mellon Univ.

Today

- **Control: Condition codes**
- **Conditional branches**
- **Loops**
- **Switch statements**
- **IA 32 Procedures**
  - Stack Structure
  - Calling Conventions
  - Illustrations of Recursion & Pointers

Processor State (IA32, Partial)

- **Information about currently executing program**
  - Temporary data (%eax, …)
  - Location of runtime stack (%ebp,%esp)
  - Location of current code control point (%eip, …)
  - Status of recent tests (CF, ZF, SF, OF)



Condition Codes (Implicit Setting)

- **Single bit registers**
  - CF Carry Flag (for unsigned) SF Sign Flag (for signed)
  - ZF Zero Flag OF Overflow Flag (for signed)
- **Implicitly set (think of it as side effect) by arithmetic operations**

  Example: `addl/addq` *Src, Dest* ↔ `t = a+b`

  CF set if carry out from most significant bit (unsigned overflow)

  ZF set if `t == 0`

  SF set if `t < 0` (as signed)

  OF set if two's-complement (signed) overflow

  `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`
- **Not set by `lea` instruction**
- **Full documentation (IA32), link on course website**

Condition Codes (Explicit Setting: Compare)

- **Explicit Setting by Compare Instruction**
  - `cmpl/cmpq` Src2, Src1
  - `cmpl b,a` like computing `a-b` without setting destination

  - CF set if carry out from most significant bit (used for unsigned comparisons)
  - ZF set if `a == b`
  - SF set if `(a-b) < 0` (as signed)
  - OF set if two's-complement (signed) overflow

```
(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)
```

Condition Codes (Explicit Setting: Test)

- **Explicit Setting by Test instruction**
  - `testl`/`testq` Src2, Src1
    `testl b,a` like computing `a&b` without setting destination
  - Sets condition codes based on value of Src1 & Src2
  - Useful to have one of the operands be a mask
  - ZF set when `a&b == 0`
  - SF set when `a&b < 0`

Reading Condition Codes

- **SetX Instructions**
  - Set single byte based on combinations of condition codes

e.g.,
```
cmpl %eax,%edx
setl %al
movzbl %al,%eax
```

SetX	Condition	Description
`sete`	`ZF`	Equal / Zero
`setne`	`~ZF`	Not Equal / Not Zero
`sets`	`SF`	Negative
`setns`	`~SF`	Nonnegative
`setg`	`~(SF^OF)&~ZF`	Greater (Signed)
`setge`	`~(SF^OF)`	Greater or Equal (Signed)
`setl`	`(SF^OF)`	Less (Signed)
`setle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`seta`	`~CF&~ZF`	Above (unsigned)
`setb`	`CF`	Below (unsigned)

Reading Condition Codes (Cont.)

`addl/addq` *Src, Dest* ↔ `t = a+b`

CF set if carry out from most significant bit (unsigned overflow)

ZF set if `t == 0`

SF set if `t < 0` (as signed)

OF set if two's-complement (signed) overflow

`(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`

OF	SF	SF^OF	
0	0	**0**	*No overflow, sign bit is correct.*
0	1	**1**	*No overflow, sign bit is correct.*
1	0	**1**	*Overflow, sign bit is reversed.*
1	1	**0**	*Overflow, sign bit is reversed.*

Reading Condition Codes (Cont.)

- **SetX Instructions:**
  - Set single byte based on combination of condition codes
- **One of 8 addressable byte registers**
  - Does not alter remaining 3 bytes
  - Typically use `movzbl` to finish job

```
int gt (int x, int y)
{
  return x > y;
}
```

Body

```
movl 12(%ebp),%eax    # eax = y
cmpl %eax,8(%ebp)     # Compare x : y
setg %al              # al = x > y
movzbl %al,%eax       # Zero rest of %eax
```

%eax	%ah	%al
%ecx	%ch	%cl
%edx	%dh	%dl
%ebx	%bh	%bl
%esi		
%edi		
%esp		
%ebp		

Reading Condition Codes: x86-64

- **SetX Instructions:**
  - Set single byte based on combination of condition codes
  - Does not alter remaining 3 bytes

```
int gt (long x, long y)
{
  return x > y;
}
```

```
long lgt (long x, long y)
{
  return x > y;
}
```

Bodies

```
cmpl %esi, %edi
setg %al
movzbl %al, %eax
```

```
cmpq %rsi, %rdi
setg %al
movzbl %al, %eax
```

Is `%rax` zero?

Yes: 32-bit instructions set high order 32 bits to 0!

Today

- Control: Condition codes
- **Conditional branches & Moves**
- Loops
- Switch statements
- IA 32 Procedures
  - Stack Structure
  - Calling Conventions
  - Illustrations of Recursion & Pointers

Jumping

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

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

```
jmp .L1
jmp *%eax
```

Conditional Branch Example

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

```
absdiff:
    pushl   %ebp               } Setup
    movl    %esp, %ebp
    movl    8(%ebp), %edx      } Body1
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6
    subl    %eax, %edx         } Body2a
    movl    %edx, %eax
    jmp .L7
.L6:
    subl %edx, %eax            } Body2b
.L7:
    popl %ebp                  } Finish
    ret
```

Conditional Branch Example (Cont.)

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
  goto Exit;
Else:
  result = y-x;
Exit:
  return result;
}
```

```
absdiff:
    pushl   %ebp               } Setup
    movl    %esp, %ebp
    movl    8(%ebp), %edx      } Body1
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6
    subl    %eax, %edx         } Body2a
    movl    %edx, %eax
    jmp .L7
.L6:
    subl %edx, %eax            } Body2b
.L7:
    popl %ebp                  } Finish
    ret
```

- **C allows "goto" as means of transferring control**
  - Closer to machine-level programming style
- **Generally considered bad coding style**

Conditional Branch Example (Cont.)

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
  goto Exit;
Else:
  result = y-x;
Exit:
  return result;
}
```

```
absdiff:
    pushl   %ebp            } Setup
    movl    %esp, %ebp
    movl    8(%ebp), %edx   } Body1
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6
    subl    %eax, %edx      } Body2a
    movl    %edx, %eax
    jmp .L7
.L6:
    subl %edx, %eax         } Body2b
.L7:
    popl %ebp               } Finish
    ret
```

Conditional Branch Example (Cont.)

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
  goto Exit;
Else:
  result = y-x;
Exit:
  return result;
}
```

```
absdiff:
    pushl   %ebp              } Setup
    movl    %esp, %ebp
    movl    8(%ebp), %edx     } Body1
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6
    subl    %eax, %edx        } Body2a
    movl    %edx, %eax
    jmp .L7
.L6:
    subl %edx, %eax           } Body2b
.L7:
    popl %ebp                 } Finish
    ret
```

Conditional Branch Example (Cont.)

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
  goto Exit;
Else:
  result = y-x;
Exit:
  return result;
}
```

```
absdiff:
    pushl   %ebp              } Setup
    movl    %esp, %ebp
    movl    8(%ebp), %edx     } Body1
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6
    subl    %eax, %edx        } Body2a
    movl    %edx, %eax
    jmp .L7
.L6:
    subl %edx, %eax           } Body2b
.L7:
    popl %ebp                 } Finish
    ret
```

General Conditional Expression Translation

C Code

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

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

Goto Version

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

- Test is expression returning integer
  - = 0 interpreted as false
  - ≠ 0 interpreted as true
- Create separate code regions for then & else expressions
- Execute appropriate one

Using Conditional Moves

- **Conditional Move Instructions**
  - Instruction supports:

    if (Test) Dest ← Src
  - Supported in post-1995 x86 processors
  - GCC does not always use them
    - Wants to preserve compatibility with ancient processors
    - Enabled for x86-64
    - Use switch `-march=686` for IA32
- **Why?**
  - Branches are very disruptive to instruction flow through pipelines
  - Conditional move do not require control transfer

C Code

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

Goto Version

```
  tval = Then_Expr;
  result = Else_Expr;
  t = Test;
  if (t) result = tval;
  return result;
```

Conditional Move Example: x86-64

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

x in %edi

y in %esi

```
absdiff:
  movl    %edi, %edx
  subl    %esi, %edx   # tval = x-y
  movl    %esi, %eax
  subl    %edi, %eax   # result = y-x
  cmpl    %esi, %edi   # Compare x:y
  cmovg   %edx, %eax   # If >, result = tval
  ret
```

Bad Cases for Conditional Move

Expensive Computations

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

- **Both values get computed**
- **Only makes sense when computations are very simple**

Risky Computations

```
val = p ? *p : 0;
```

- Both values get computed
- May have undesirable effects

Computations with side effects

```
val = x > 0 ? x*=7 : x+=3;
```

- Both values get computed
- Must be side-effect free

Today

- Control: Condition codes
- Conditional branches and moves
- **Loops**
- Switch statements
- IA 32 Procedures
  - Stack Structure
  - Calling Conventions
  - Illustrations of Recursion & Pointers

"Do-While" Loop Example

C Code

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

Goto Version

```
int pcount_do(unsigned x)
{
  int 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

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

Registers:

%edx	x
%ecx	result

```
    movl   $0, %ecx      # result = 0
.L2:                     # loop:
    movl   %edx, %eax
    andl   $1, %eax      # t = x & 1
    addl   %eax, %ecx    # result += t
    shrl   %edx          # x >>= 1
    jne    .L2           # If !0, goto loop
```

General "Do-While" Translation

C Code

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

Goto Version

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

- **Body:**

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

- **Test returns integer**
  - = 0 interpreted as false
  - ≠ 0 interpreted as true

"While" Loop Example

C Code

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

Goto Version

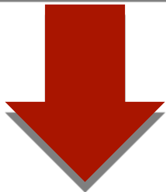
```
int pcount_do(unsigned x) {
  int result = 0;
  if (!x) goto done;
loop:
  result += x & 0x1;
  x >>= 1;
  if (x)
    goto loop;
done:
  return result;
}
```

- **Is this code equivalent to the do-while version?**

General "While" Translation

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:
```

"For" Loop Example

C Code

```
#define WSIZE 8*sizeof(int)
int pcount_for(unsigned x) {
  int i;
  int result = 0;
  for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  return result;
}
```

- Is this code equivalent to other versions?

"For" Loop Form

General Form

```
for (Init; Test; Update )

    Body
```

```
for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

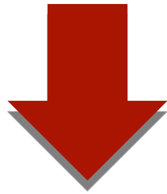
Body

```
{
  unsigned mask = 1 << i;
  result += (x & mask) != 0;
}
```

"For" Loop → While Loop

For Version

```
for (Init; Test; Update )

    Body
```



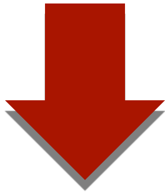
While Version

```
Init;

while (Test ) {

    Body

    Update;

}
```

"For" Loop → … → Goto

For Version

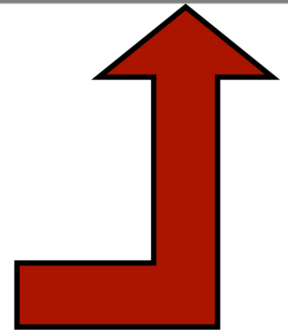
```
for (Init; Test; Update )
    Body
```



While Version

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

```
  Init;
  if (!Test)
    goto done;
  do
    Body
    Update
  while(Test);
done:
```



```
  Init;
  if (!Test)
    goto done;
loop:
  Body
  Update
  if (Test)
    goto loop;
done:
```

"For" Loop Conversion Example

C Code

```
#define WSIZE 8*sizeof(int)
int pcount_for(unsigned x) {
  int i;
  int result = 0;
  for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  return result;
}
```

- Initial test can be optimized away

Goto Version

```
int pcount_for_gt(unsigned x) {
  int i;
  int result = 0;
  i = 0;                          Init
  if (!(i < WSIZE))               ! Test
    goto done;
 loop:
  {                               Body
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  i++;                            Update
  if (i < WSIZE)                  Test
    goto loop;
 done:
  return result;
}
```

(The lines `if (!(i < WSIZE))` and `goto done;` are crossed out.)

Today

- Control: Condition codes
- Conditional branches
- Loops
- **Switch statements**
- IA 32 Procedures
  - Stack Structure
  - Calling Conventions
  - Illustrations of Recursion & Pointers

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;
}
```

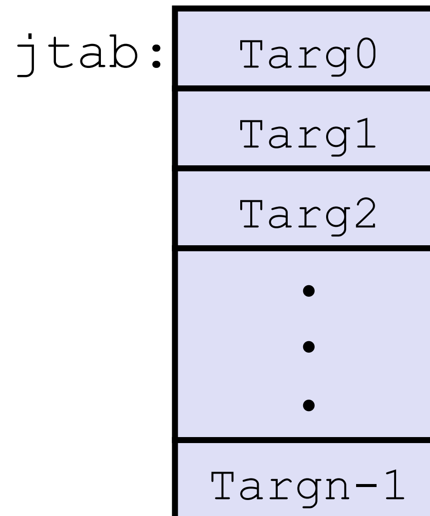
- **Multiple case labels**
  - Here: 5 & 6
- **Fall through cases**
  - Here: 2
- **Missing cases**
  - Here: 4

Jump Table Structure

Switch Form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

Jump Table



Jump Targets

Targ0: Code Block 0

Targ1: Code Block 1

Targ2: Code Block 2

Targn-1: Code Block n–1

Approximate Translation

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

Switch Statement Example (IA32)

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

What range of values takes default?

Setup:

```
switch_eg:
   pushl  %ebp             # Setup
   movl   %esp, %ebp       # Setup
   movl   8(%ebp), %eax    # %eax = x
   cmpl   $6, %eax         # Compare x:6
   ja     .L2              # If unsigned > goto default
   jmp    *.L7(,%eax,4)    # Goto *JTab[x]
```

Note that **w** not initialized here

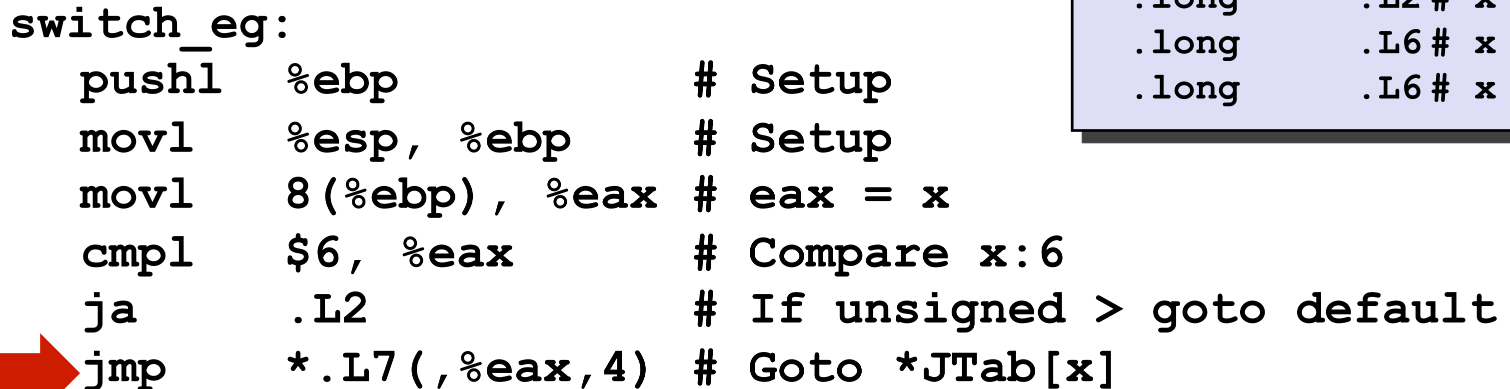
Switch Statement Example (IA32)

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

Jump table

```
.section    .rodata
  .align 4
.L7:
  .long     .L2 # x = 0
  .long     .L3 # x = 1
  .long     .L4 # x = 2
  .long     .L5 # x = 3
  .long     .L2 # x = 4
  .long     .L6 # x = 5
  .long     .L6 # x = 6
```

Setup:



Assembly Setup Explanation

- **Table Structure**
  - Each target requires 4 bytes
  - Base address at `.L7`
- **Jumping**
  - Direct: **`jmp .L2`**
  - Jump target is denoted by label `.L2`
  - Indirect: **`jmp *.L7(,%eax,4)`**
  - Start of jump table: `.L7`
  - Must scale by factor of 4 (labels have 32-bits = 4 Bytes on IA32)
  - Fetch target from effective Address `.L7 + eax*4`
    - Only for $0 \le x \le 6$

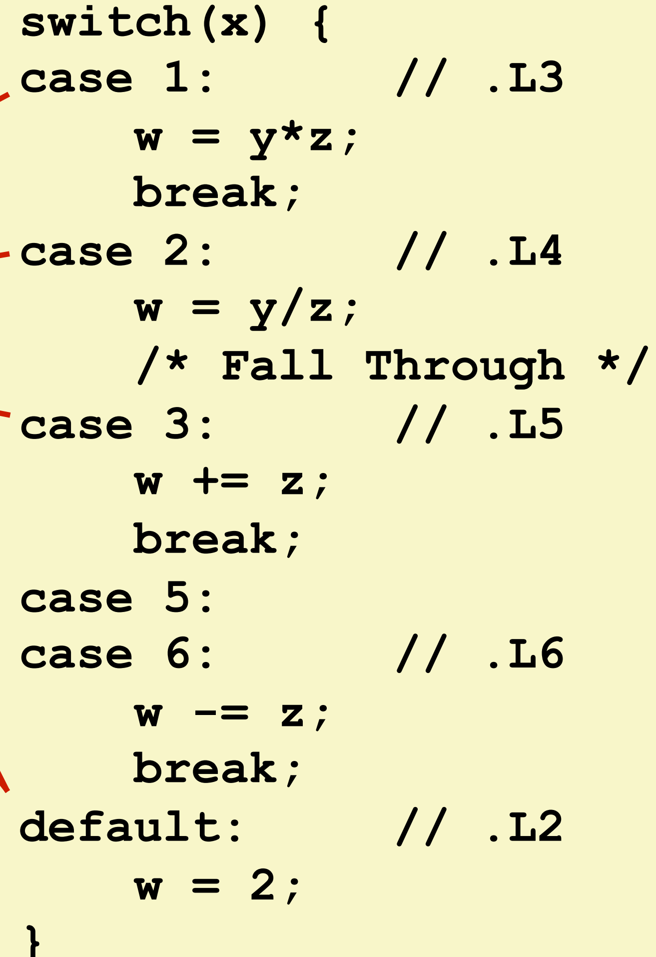
Jump table

```
.section    .rodata
  .align 4
.L7:
  .long     .L2 # x = 0
  .long     .L3 # x = 1
  .long     .L4 # x = 2
  .long     .L5 # x = 3
  .long     .L2 # x = 4
  .long     .L6 # x = 5
  .long     .L6 # x = 6
```

Jump Table

Jump table

```
.section   .rodata
  .align 4
.L7:
  .long    .L2 # x = 0
  .long    .L3 # x = 1
  .long    .L4 # x = 2
  .long    .L5 # x = 3
  .long    .L2 # x = 4
  .long    .L6 # x = 5
  .long    .L6 # x = 6
```



Handling Fall-Through

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

```
case 3:
        w = 1;
        goto merge;
```

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

```
merge:
        w += z;
```

Code Blocks (Partial)

```
switch(x) {
case 1:        // .L3
      w = y*z;
      break;
 . . .
case 3:        // .L5
    w += z;
    break;
  . . .
default:       // .L2
    w = 2;
}
```

```
.L2:                    # Default
   movl  $2, %eax       # w = 2
   jmp   .L8            # Goto done

.L5:                    # x == 3
   movl  $1, %eax       # w = 1
   jmp   .L9            # Goto merge

.L3:                    # x == 1
   movl  16(%ebp), %eax   # z
   imull 12(%ebp), %eax   # w = y*z
   jmp   .L8             # Goto done
```

Code Blocks (Rest)

```
switch(x) {
  . . .
  case 2:  // .L4
      w = y/z;
      /* Fall Through */
  merge:    // .L9
      w += z;
      break;
  case 5:
  case 6:  // .L6
      w -= z;
      break;
}
```

```
.L4:                    # x == 2
   movl  12(%ebp), %edx
   movl  %edx, %eax
   sarl  $31, %edx
   idivl 16(%ebp)      # w = y/z

.L9:                    # merge:
   addl  16(%ebp), %eax # w += z
   jmp   .L8            # goto done

.L6:                    # x == 5, 6
   movl  $1, %eax          # w = 1
   subl  16(%ebp), %eax # w = 1-z
```

Switch Code (Finish)

```
return w;
```

```
.L8:                 # done:
   popl  %ebp
   ret
```

- **Noteworthy Features**
  - Jump table avoids sequencing through cases
    - Constant time, rather than linear
  - Use jump table to handle holes and duplicate tags
  - Use program sequencing to handle fall-through
  - Don't initialize w = 1 unless really need it

x86-64 Switch Implementation

- Same general idea, adapted to 64-bit code
- Table entries 64 bits (pointers)
- Cases use revised code

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

```
.L3:
  movq    %rdx, %rax
  imulq   %rsi, %rax
  ret
```

Jump Table

```
  .section  .rodata
  .align 8
.L7:
  .quad     .L2     # x = 0
  .quad     .L3     # x = 1
  .quad     .L4     # x = 2
  .quad     .L5     # x = 3
  .quad     .L2     # x = 4
  .quad     .L6     # X = 5
  .quad     .L6     # x = 6
```

IA32 Object Code

- **Setup**
  - Label `.L2` becomes address `0x8048422`
  - Label `.L7` becomes address `0x8048660`

Assembly Code

```
switch_eg:
  . . .
  ja      .L2              # If unsigned > goto default
  jmp     *.L7(,%eax,4)  # Goto *JTab[x]
```

Disassembled Object Code

```
08048410 <switch_eg>:
 . . .
 8048419:77 07                ja     8048422 <switch_eg+0x12>
 804841b:ff 24 85 60 86 04 08 jmp    *0x8048660(,%eax,4)
```

IA32 Object Code (cont.)

- **Jump Table**
  - Doesn't show up in disassembled code
  - Can inspect using GDB
  - `gdb switch`
  - `(gdb) x/7xw 0x8048660`
    - Examine 7 hexadecimal format "words" (4-bytes each)
    - Use command "`help x`" to get format documentation

```
0x8048660:    0x08048422    0x08048432    0x0804843b    0x08048429
0x8048670:    0x08048422    0x0804844b    0x0804844b
```

IA32 Object Code (cont.)

- Deciphering Jump Table

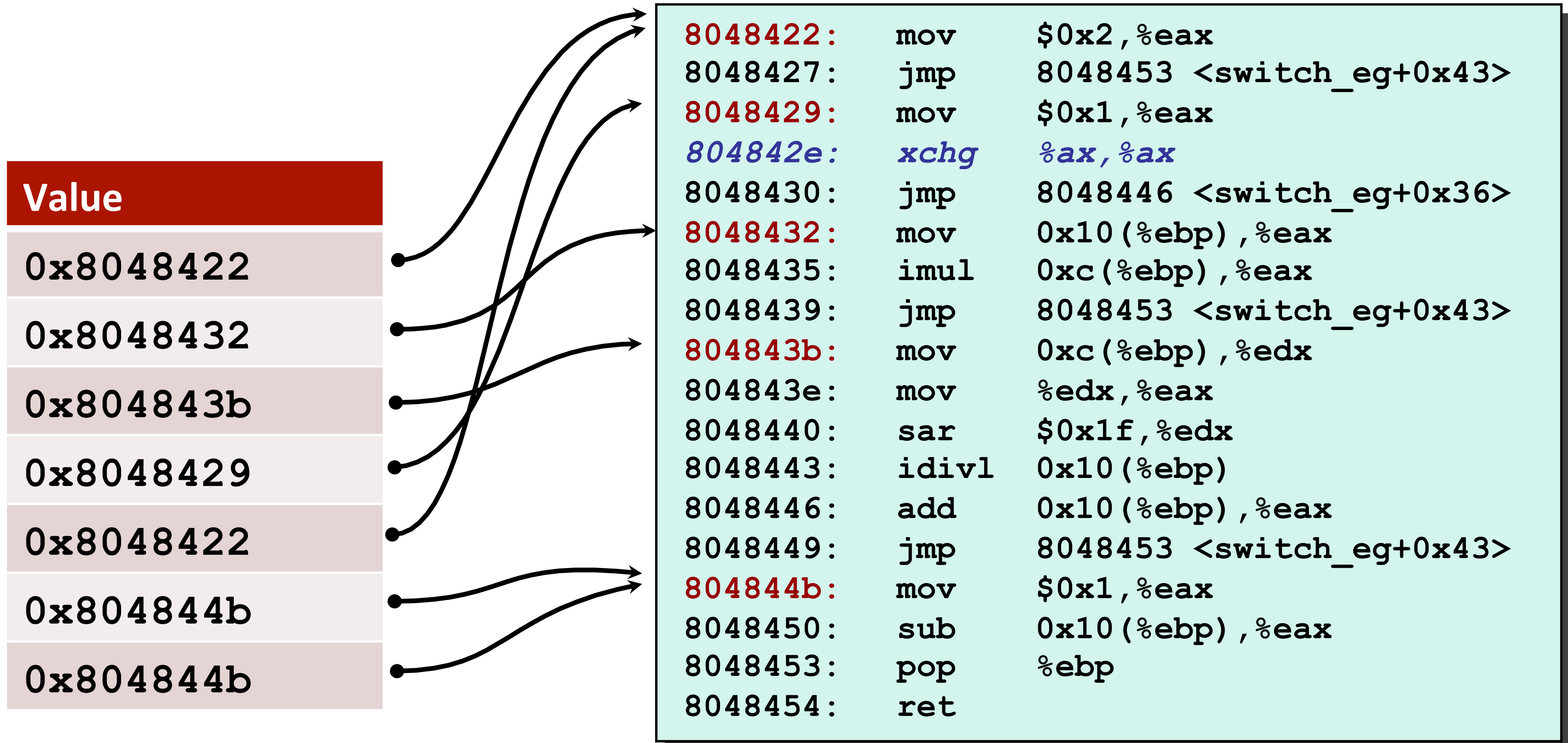
```
0x8048660:      0x08048422      0x08048432      0x0804843b      0x08048429
0x8048670:      0x08048422      0x0804844b      0x0804844b
```

Address	Value	x
0x8048660	0x8048422	0
0x8048664	0x8048432	1
0x8048668	0x804843b	2
0x804866c	0x8048429	3
0x8048670	0x8048422	4
0x8048674	0x804844b	5
0x8048678	0x804844b	6

Disassembled Targets

```
8048422:   b8 02 00 00 00        mov    $0x2,%eax
8048427:   eb 2a                 jmp    8048453 <switch_eg+0x43>
8048429:   b8 01 00 00 00        mov    $0x1,%eax
804842e:   66 90                 xchg   %ax,%ax # noop
8048430:   eb 14                 jmp    8048446 <switch_eg+0x36>
8048432:   8b 45 10              mov    0x10(%ebp),%eax
8048435:   0f af 45 0c           imul   0xc(%ebp),%eax
8048439:   eb 18                 jmp    8048453 <switch_eg+0x43>
804843b:   8b 55 0c              mov    0xc(%ebp),%edx
804843e:   89 d0                 mov    %edx,%eax
8048440:   c1 fa 1f              sar    $0x1f,%edx
8048443:   f7 7d 10              idivl  0x10(%ebp)
8048446:   03 45 10              add    0x10(%ebp),%eax
8048449:   eb 08                 jmp    8048453 <switch_eg+0x43>
804844b:   b8 01 00 00 00        mov    $0x1,%eax
8048450:   2b 45 10              sub    0x10(%ebp),%eax
8048453:   5d                    pop    %ebp
8048454:   c3                    ret
```

Matching Disassembled Targets



Summarizing

- **C Control**
  - if-then-else
  - do-while
  - while, for
  - switch
- **Assembler Control**
  - Conditional jump
  - Conditional move
  - Indirect jump
  - Compiler generates code sequence to implement more complex control
- **Standard Techniques**
  - Loops converted to do-while form
  - Large switch statements use jump tables
  - Sparse switch statements may use decision trees

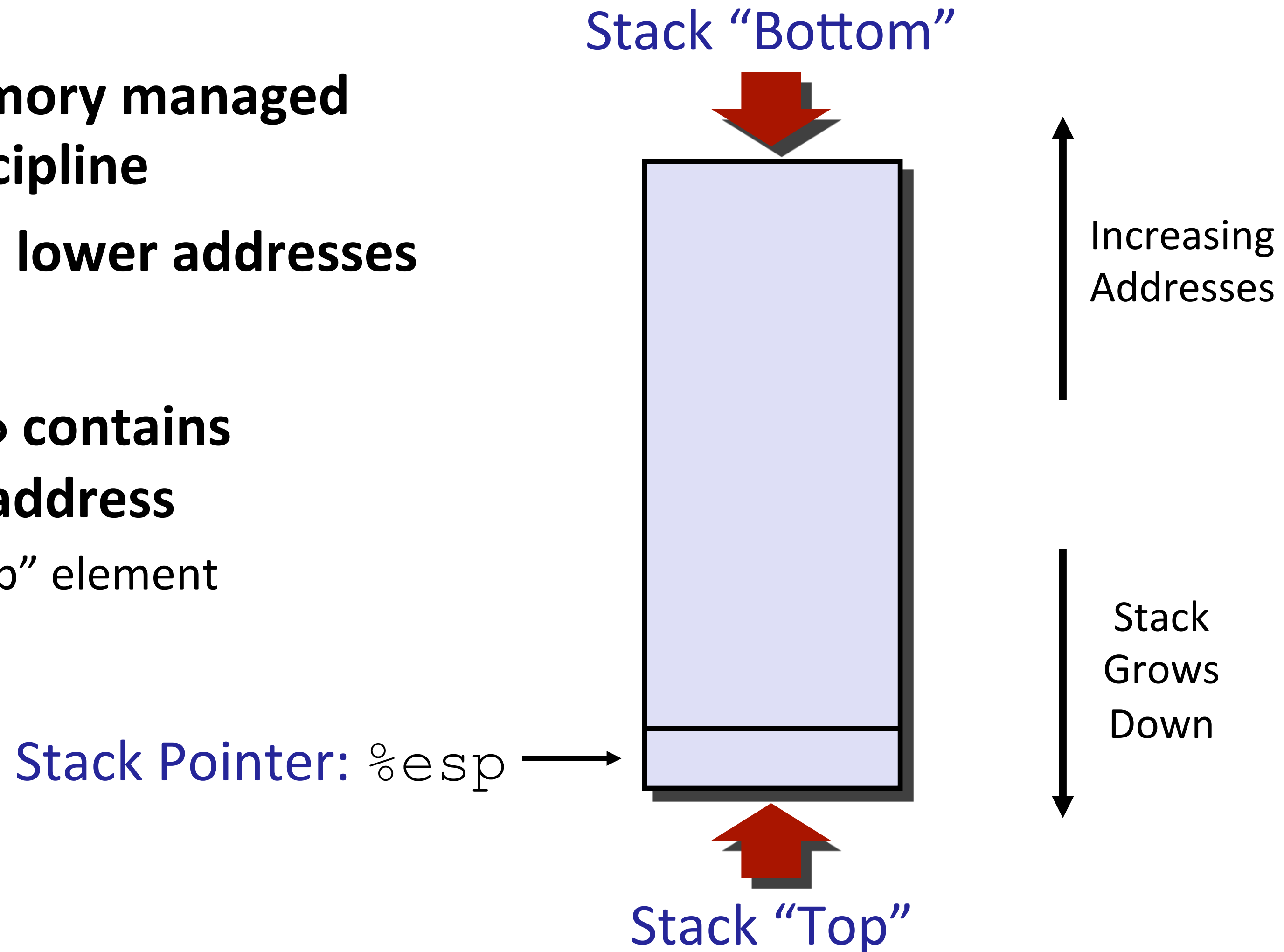
Today

- Control: Condition codes
- Conditional branches
- Loops
- Switch statements
- **IA 32 Procedures**
  - Stack Structure
  - Calling Conventions
  - Illustrations of Recursion & Pointers

IA32 Stack

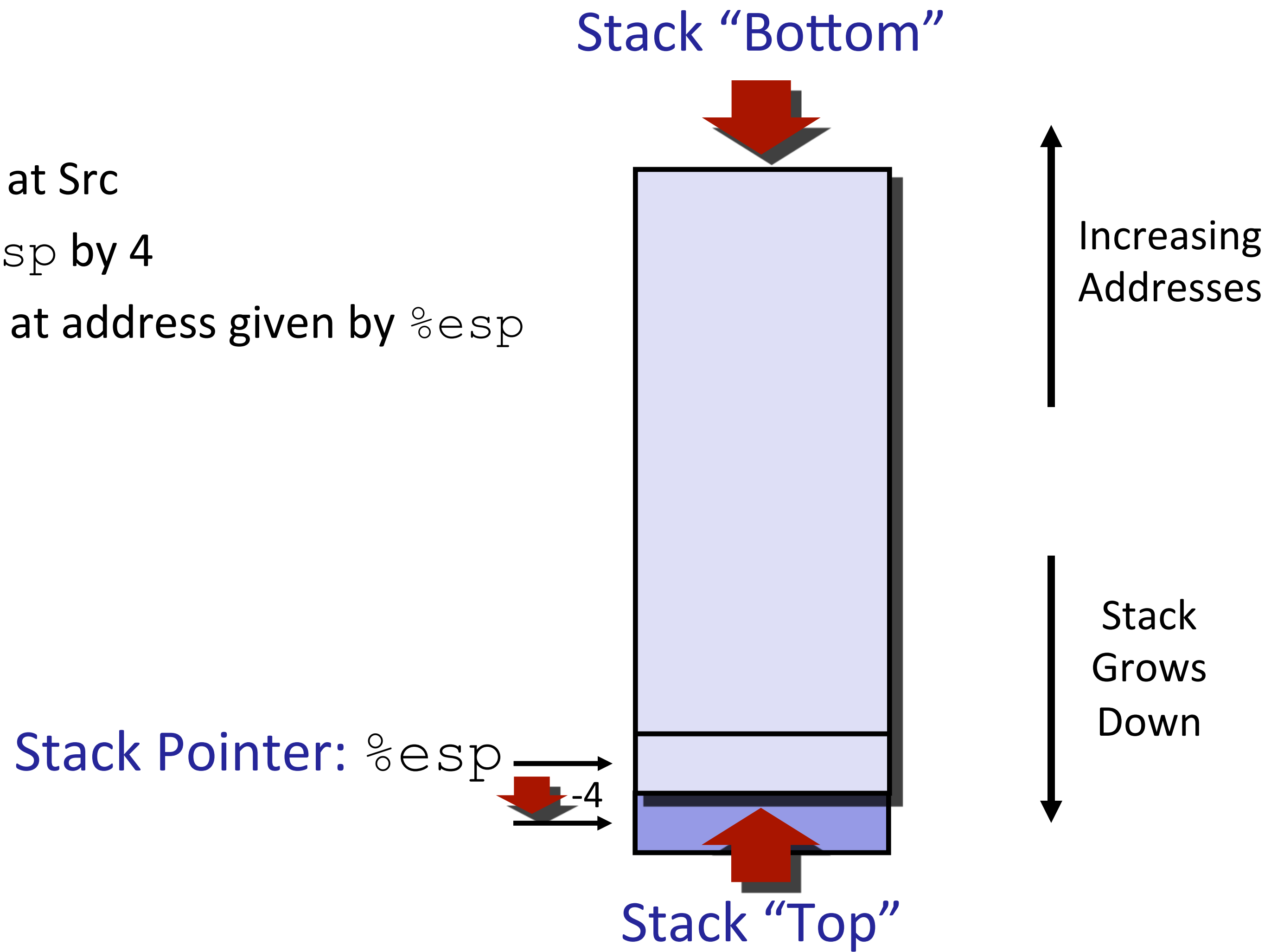
- **Region of memory managed with stack discipline**
- **Grows toward lower addresses**

- **Register `%esp` contains lowest stack address**
  - address of "top" element



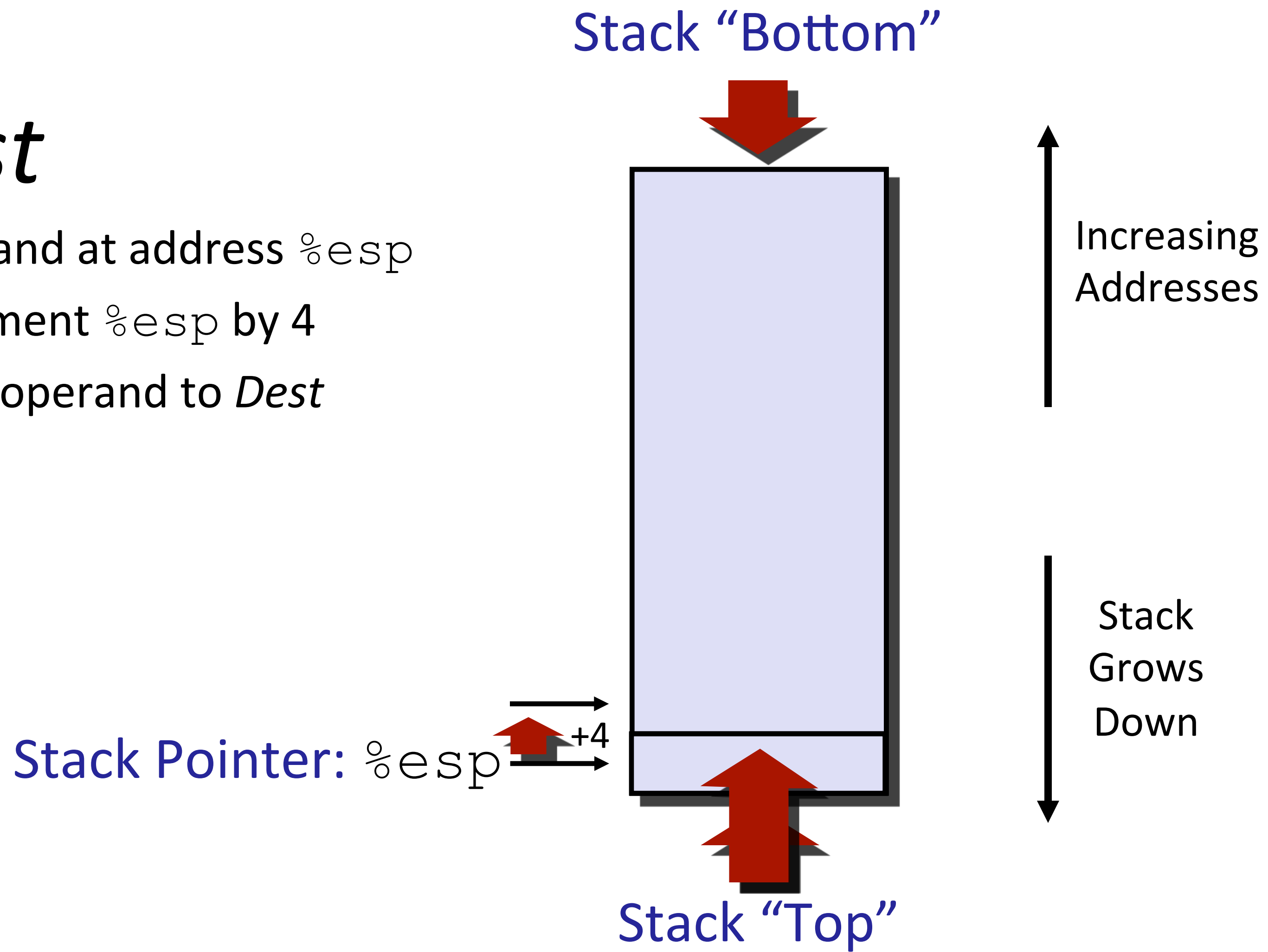
IA32 Stack: Push

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



IA32 Stack: Pop

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



Procedure Control Flow

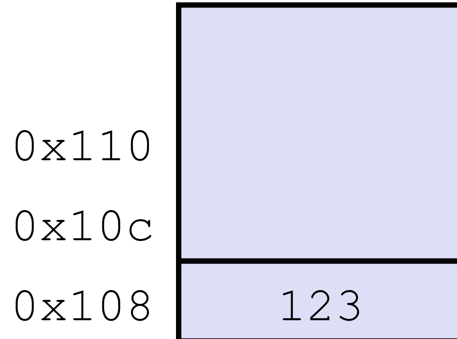
- **Use stack to support procedure call and return**
- **Procedure call: `call label`**
  - Push return address on stack
  - Jump to label
- **Return address:**
  - Address of the next instruction right after call
  - Example from disassembly

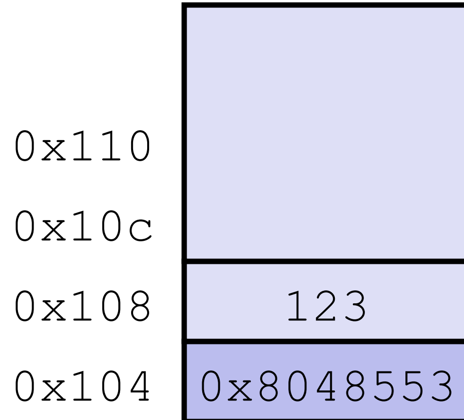
```
804854e:   e8 3d 06 00 00     call    8048b90 <main>
8048553:   50                 pushl   %eax
```

  - 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
```

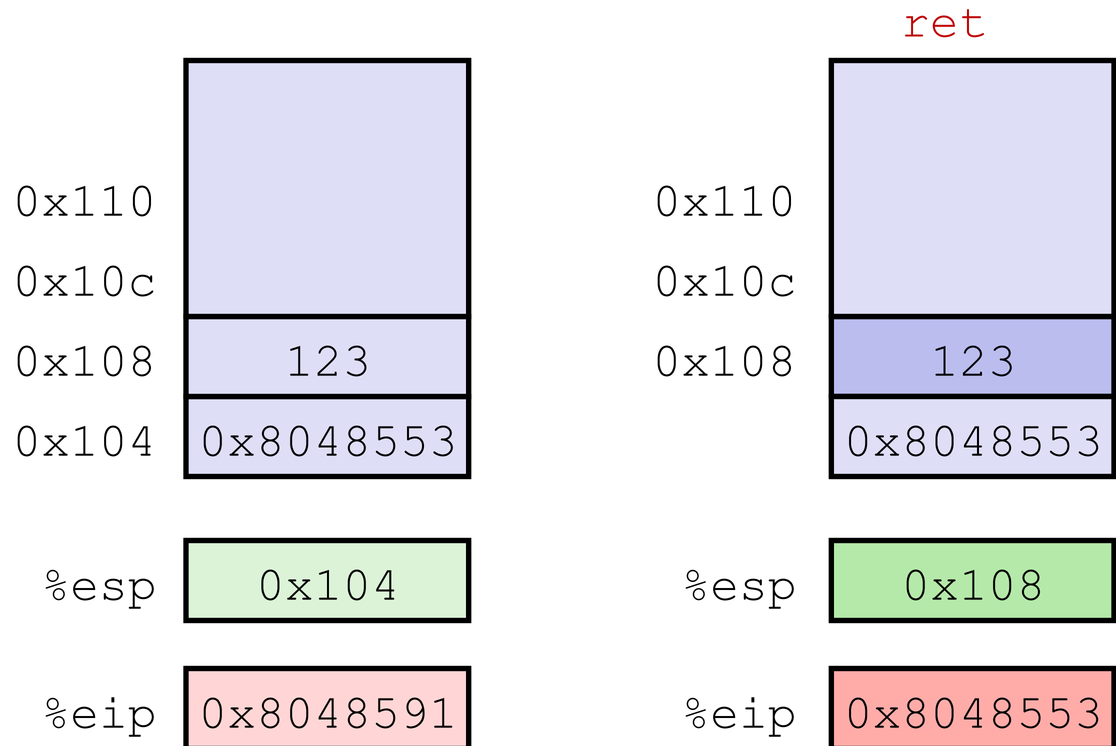




%eip: program counter

Procedure Return Example

```
8048591:      c3                      ret
```



`%eip`: 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

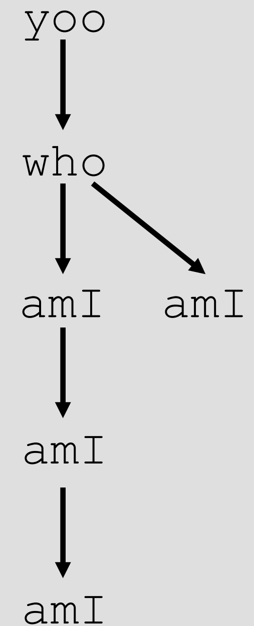
Call Chain Example

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

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

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

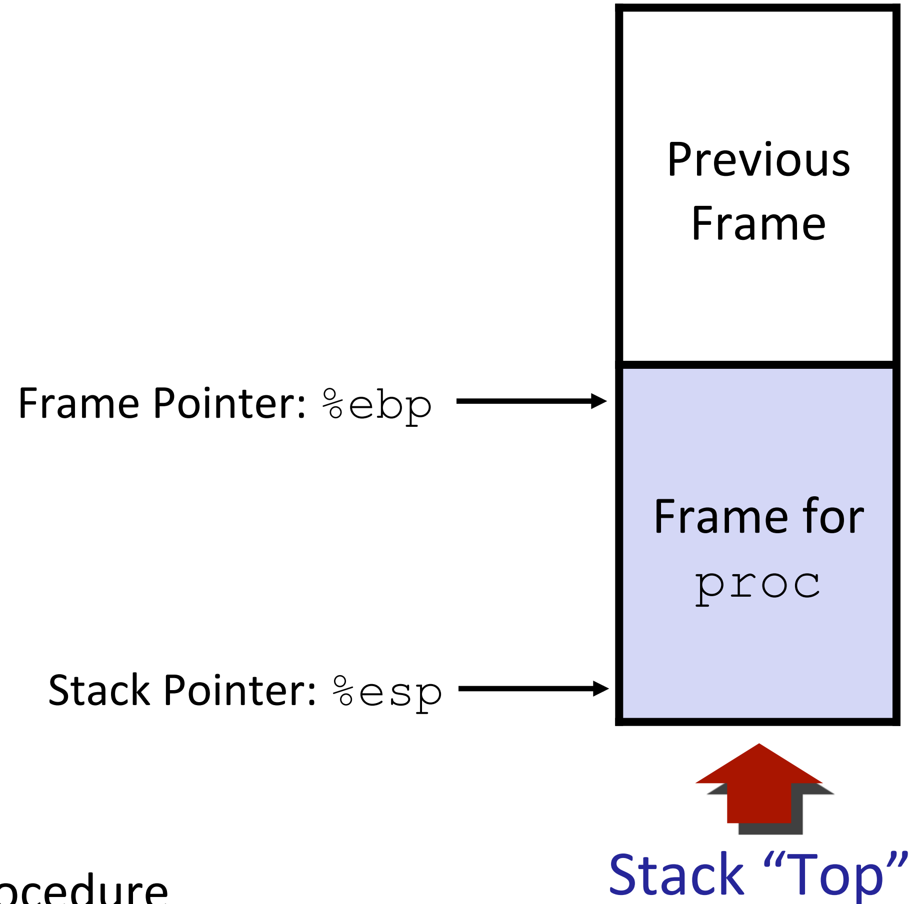
Example
Call Chain



Procedure `amI()` is recursive

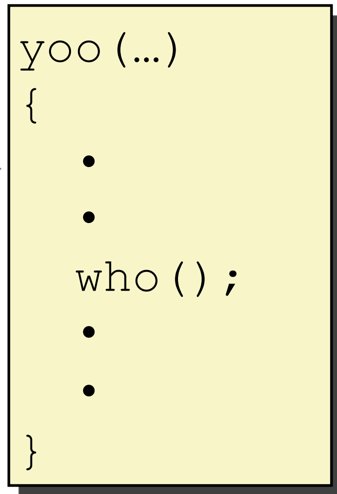
Stack Frames

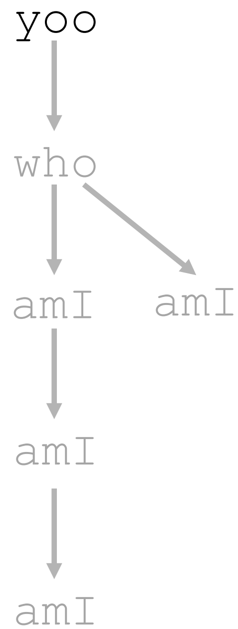
- **Contents**
  - Local variables
  - Return information
  - Temporary space

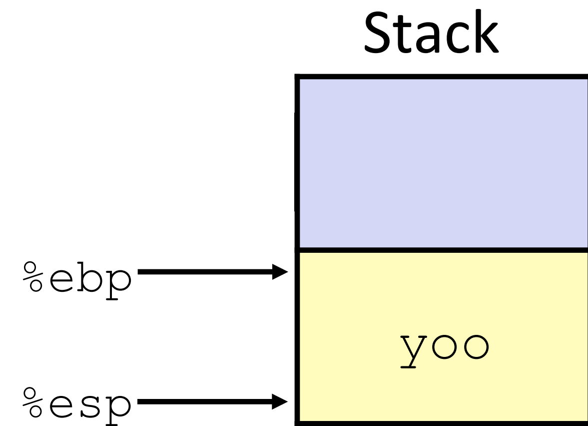


- **Management**
  - Space allocated when enter procedure
    - "Set-up" code
  - Deallocated when return
    - "Finish" code

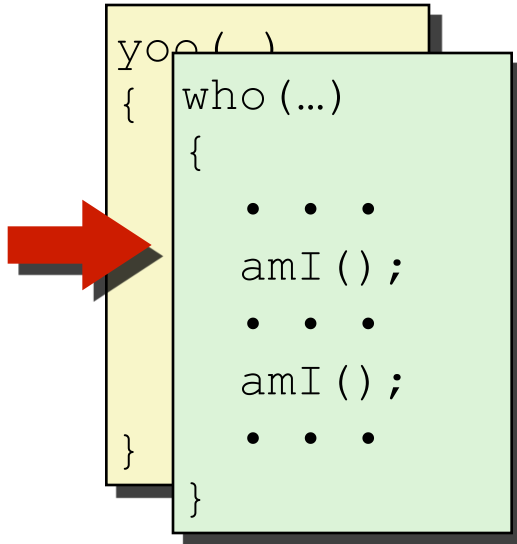
Example

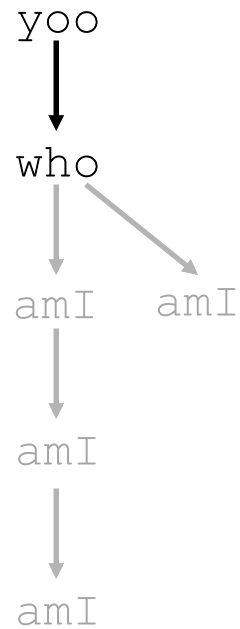


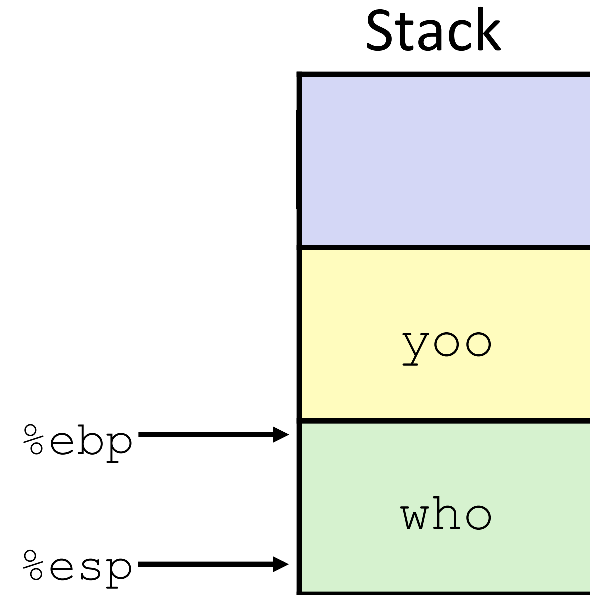




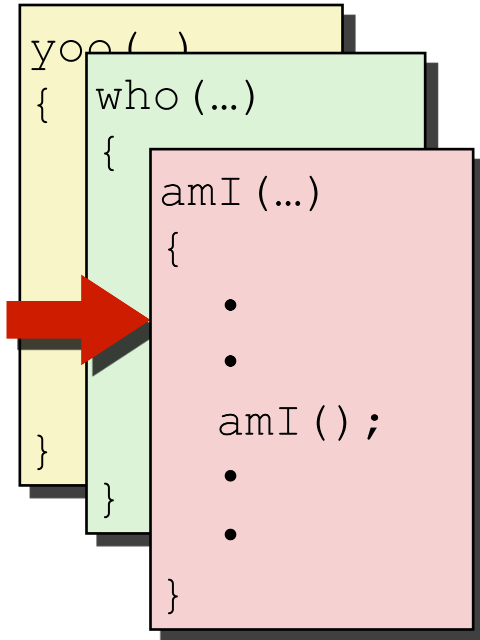
Example

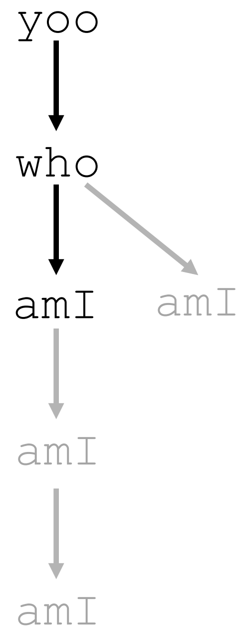


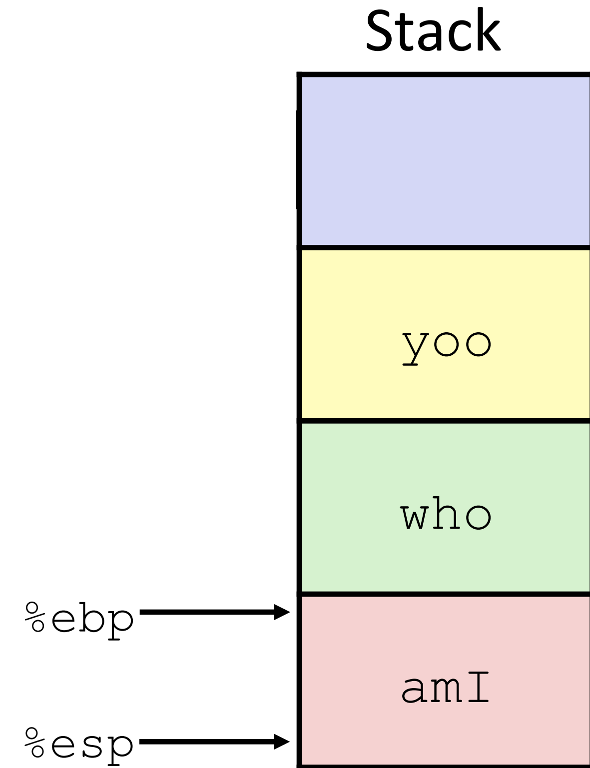




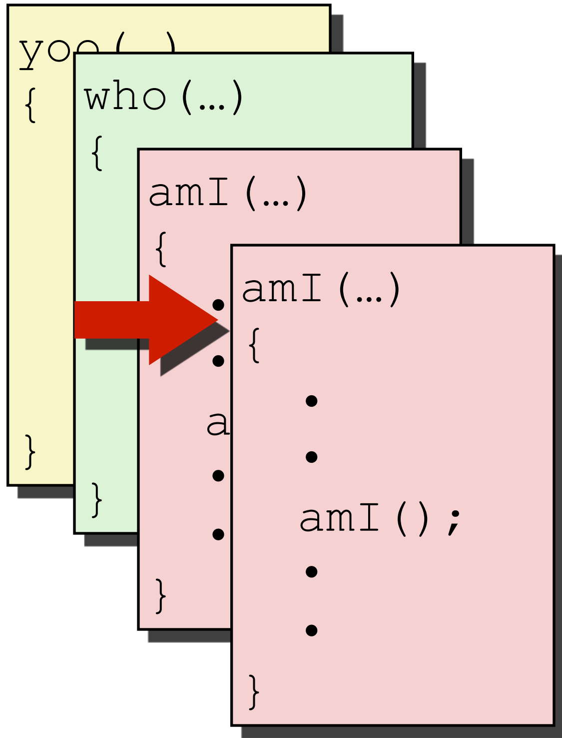
Example

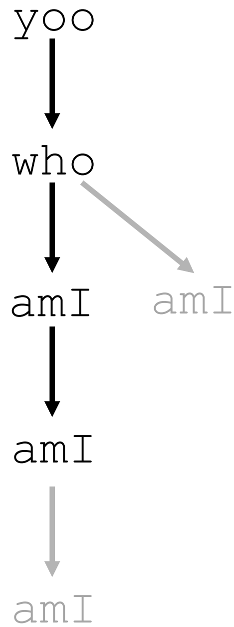


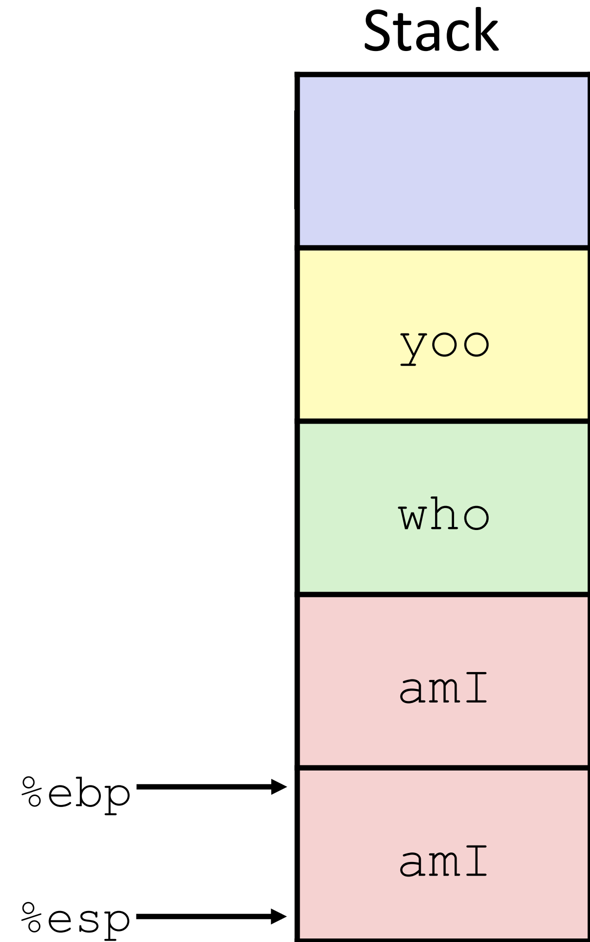




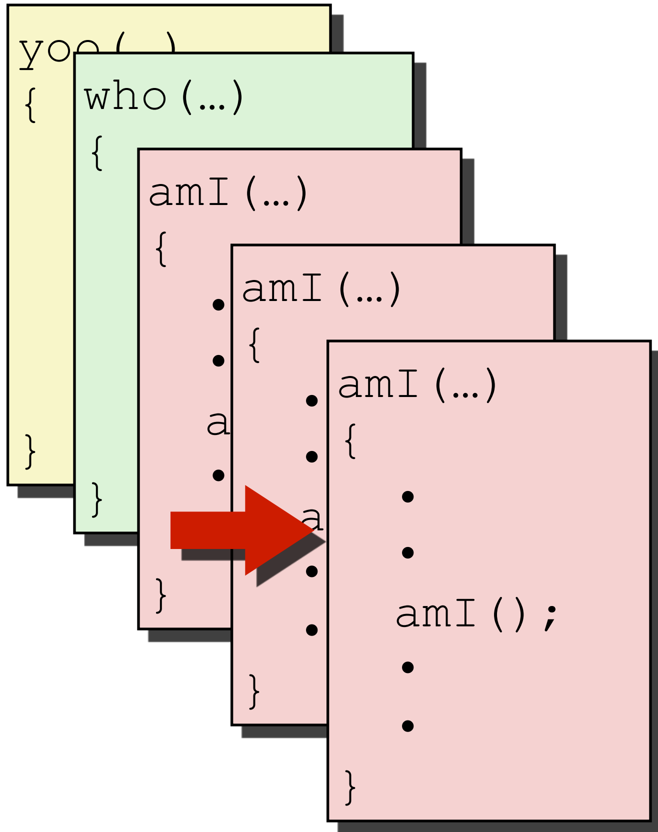
Example

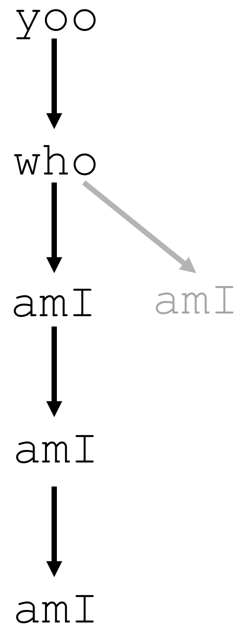


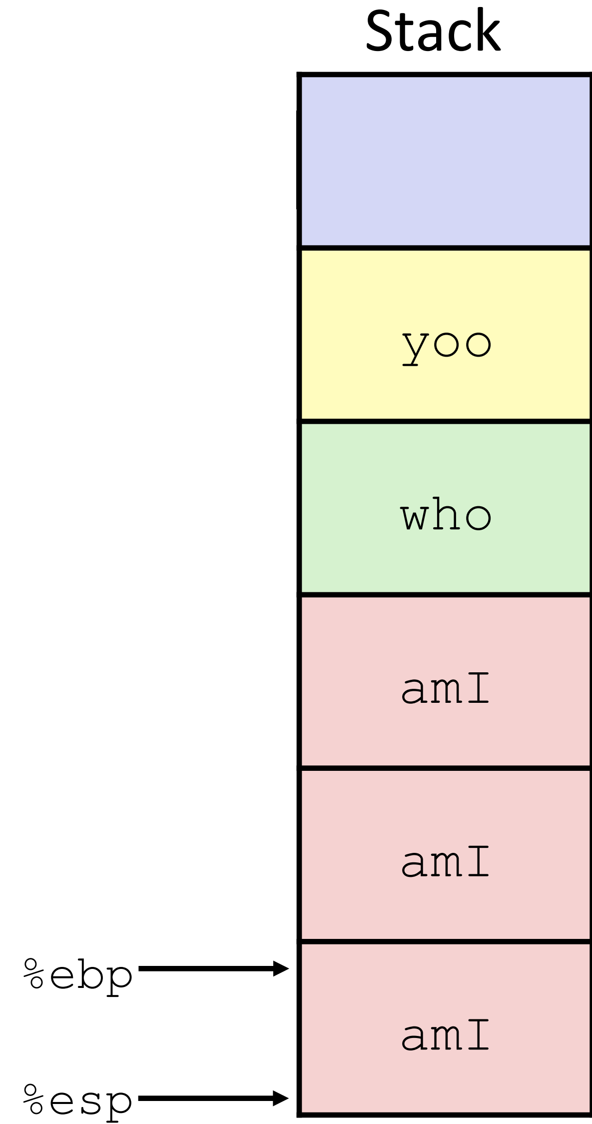




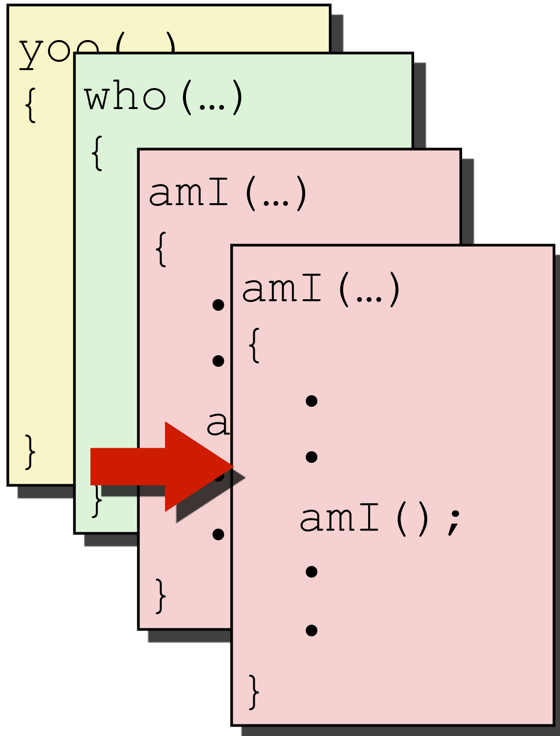
Example

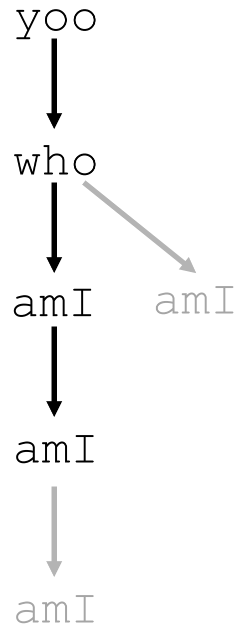


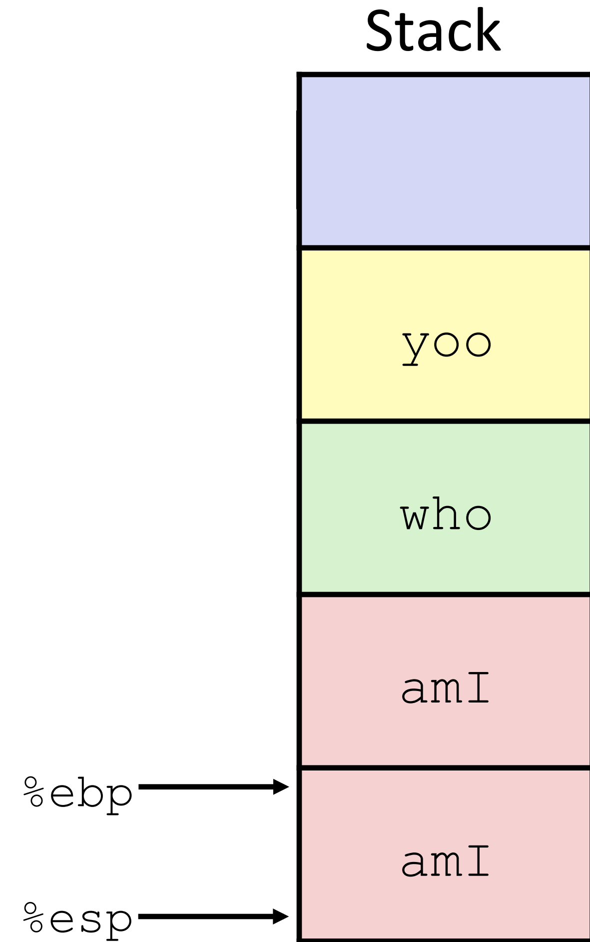




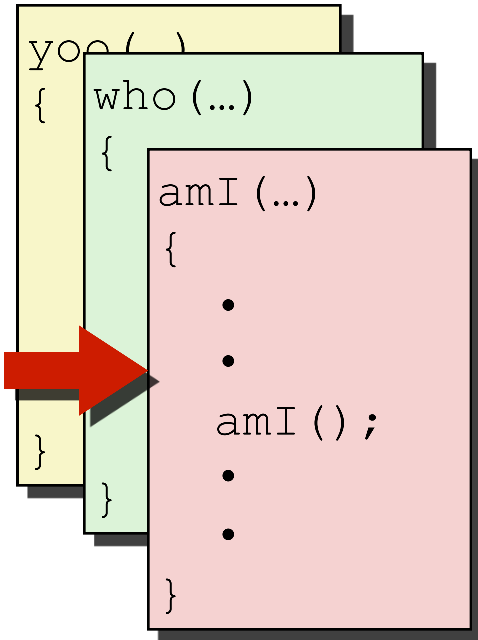
Example

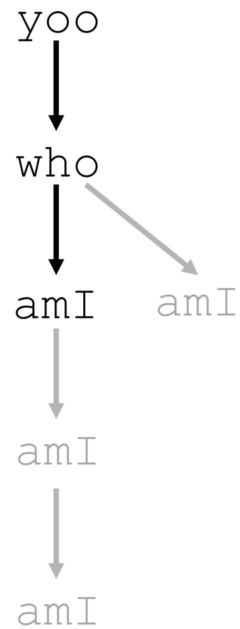


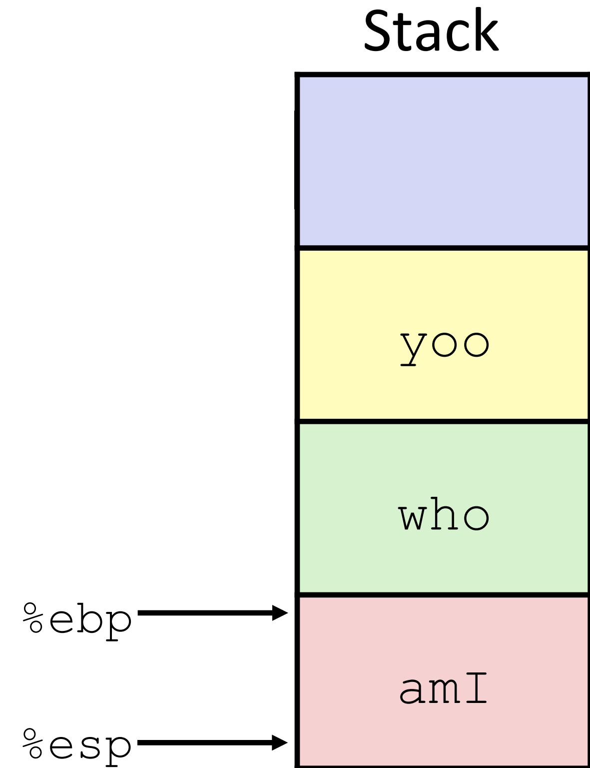




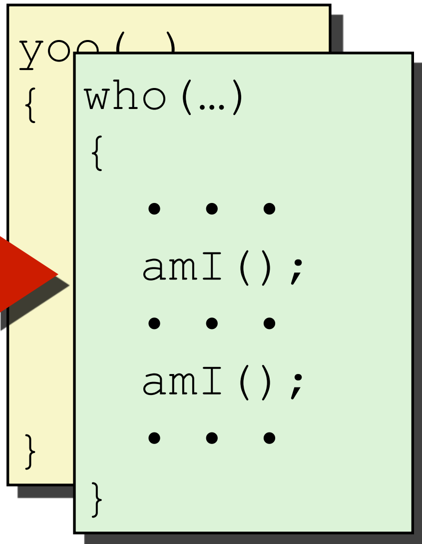
Example

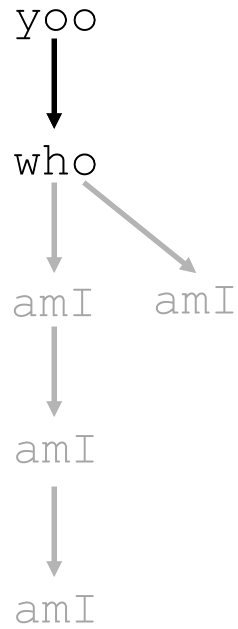


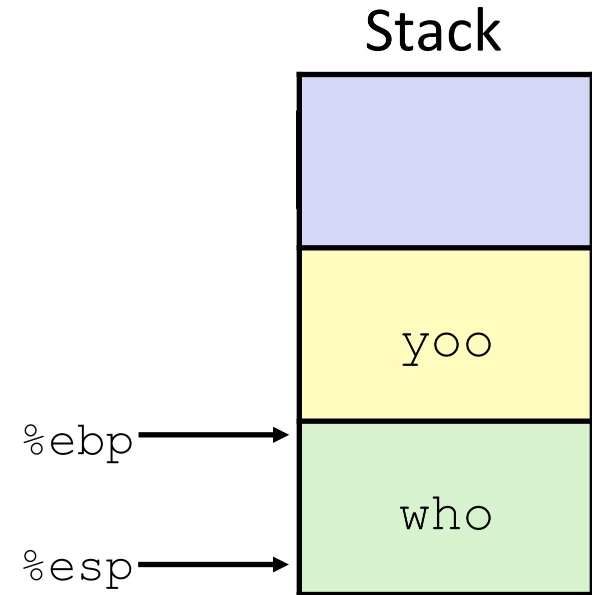




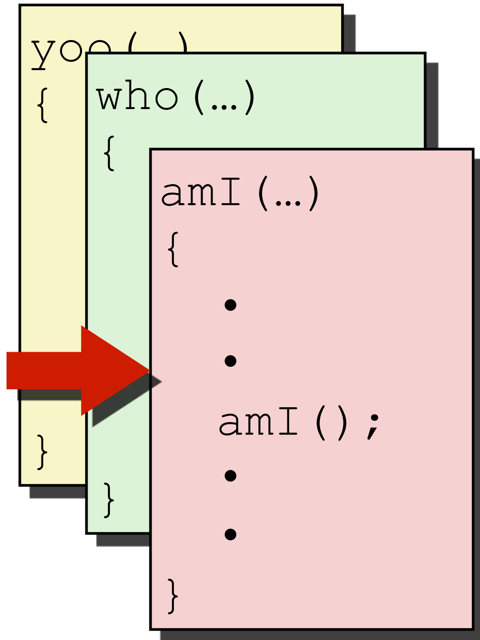
Example

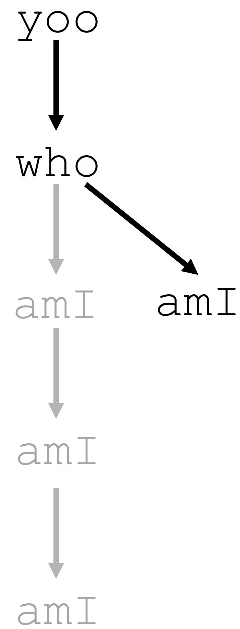


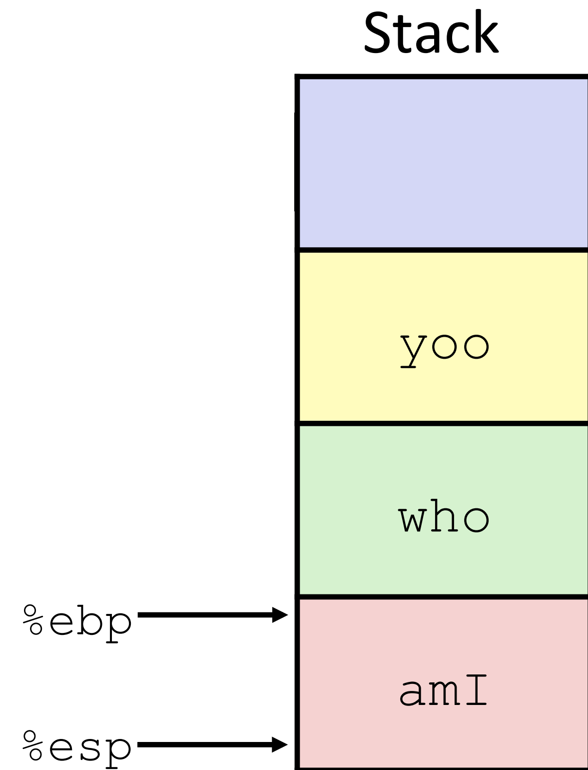




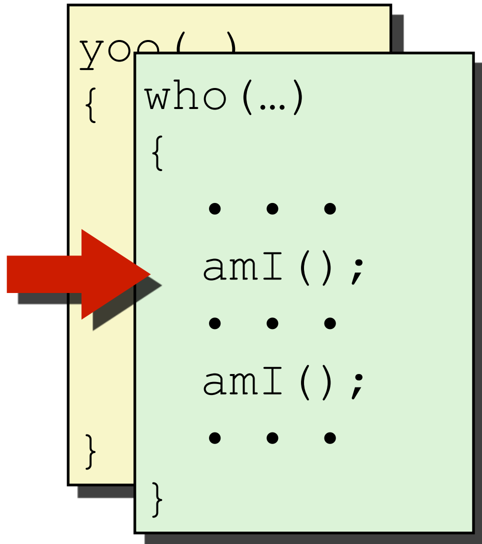
Example

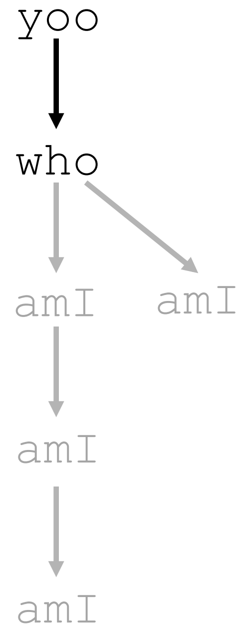


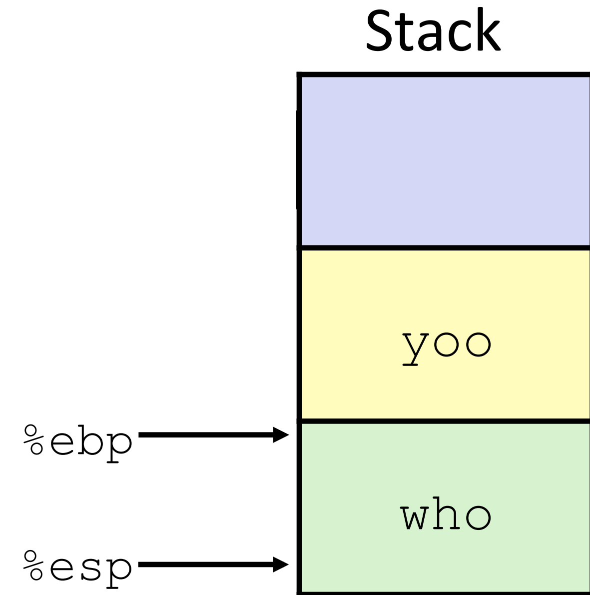




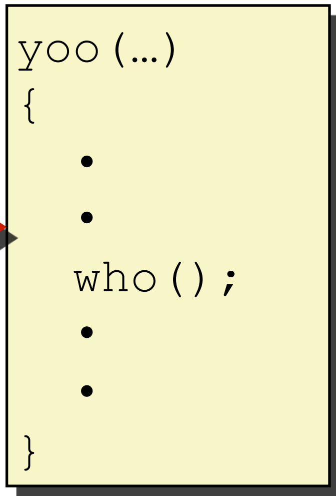
Example

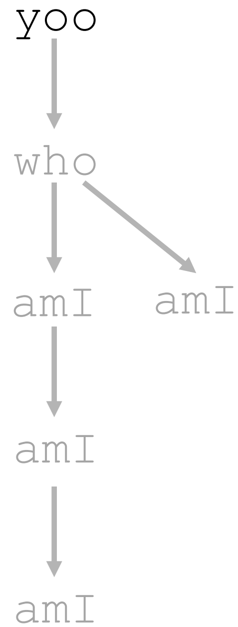


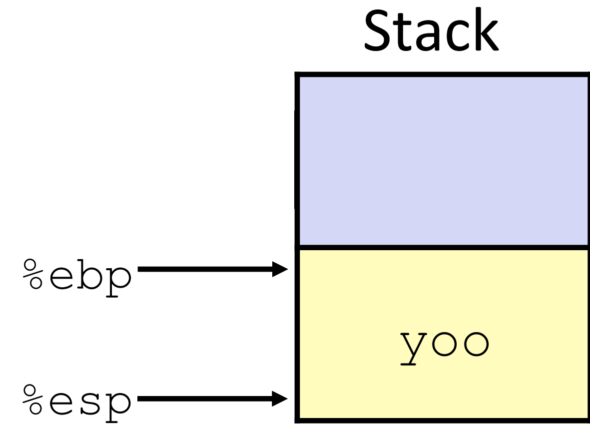




Example



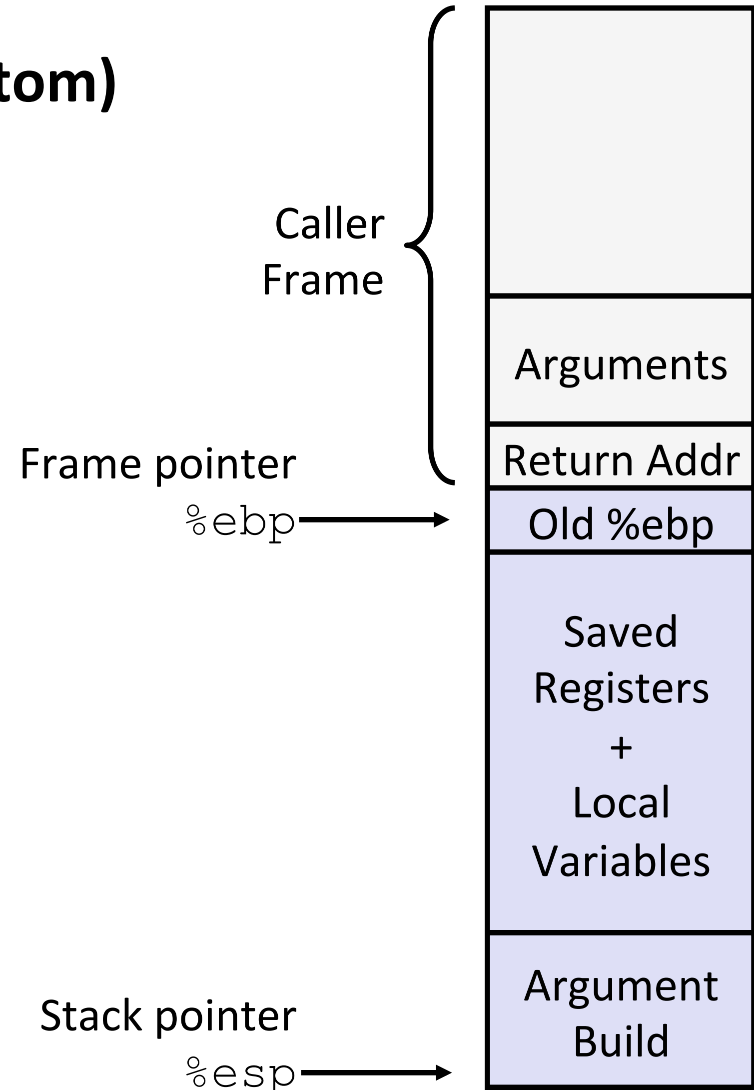




IA32/Linux Stack Frame

- **Current Stack Frame ("Top" to Bottom)**
  - "Argument build:"
    Parameters for function about to call
  - 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



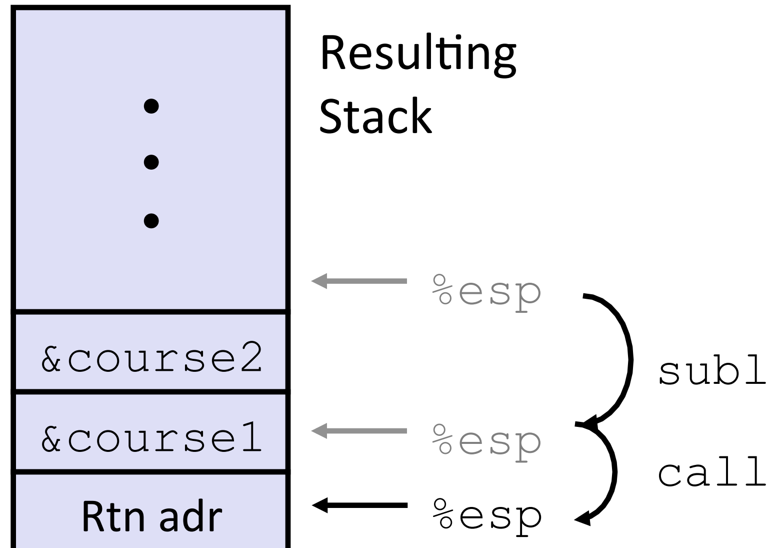
Revisiting swap

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

void call_swap() {
  swap(&course1, &course2);
}
```

Calling `swap` from `call_swap`

```
call_swap:
    • • •
    subl    $8, %esp
    movl    $course2, 4(%esp)
    movl    $course1, (%esp)
    call    swap
    • • •
```

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



Revisiting swap

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

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

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

    popl  %ebx
    popl  %ebp
    ret
```

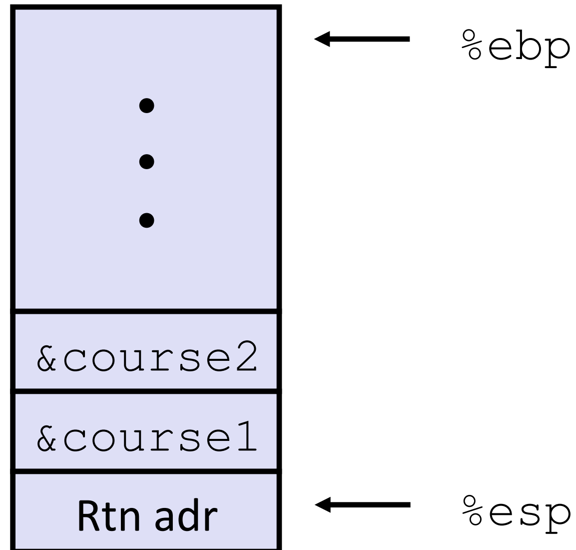
Set Up

Body

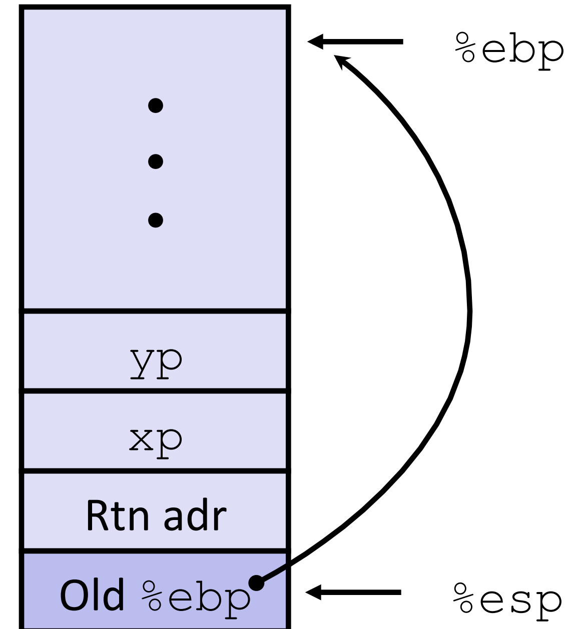
Finish

swap Setup #1

Entering Stack



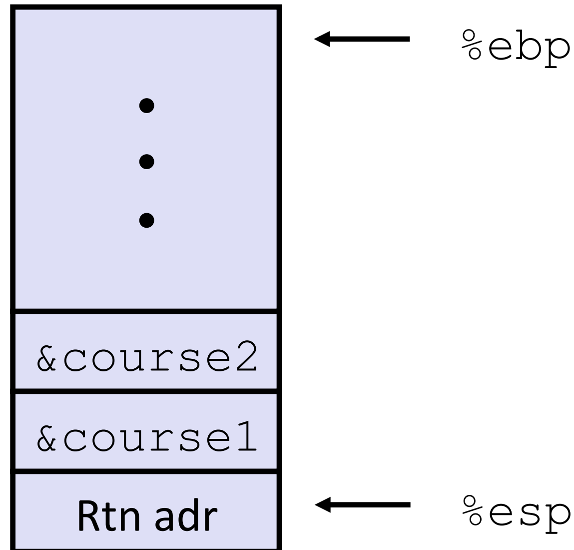
Resulting Stack



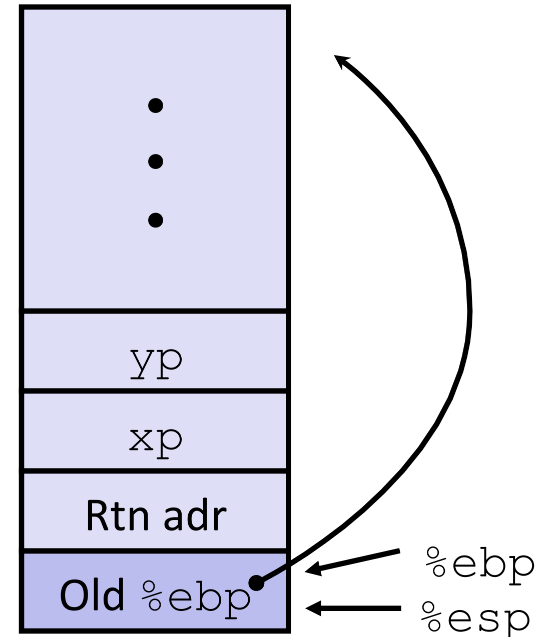

```
swap:
    pushl  %ebp
    movl  %esp,%ebp
    pushl  %ebx
```

swap Setup #2

Entering Stack



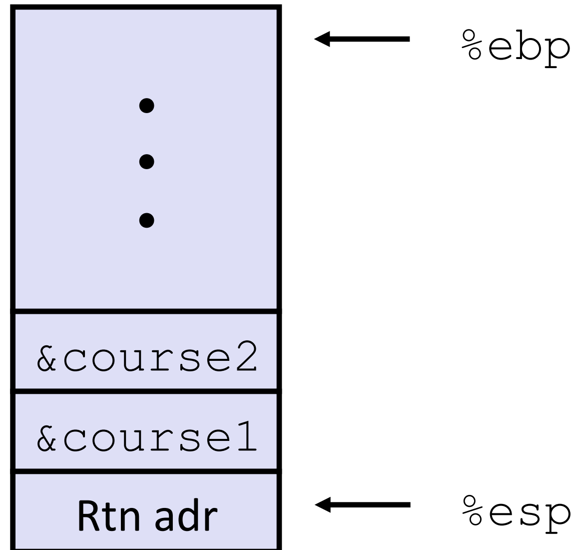
Resulting Stack



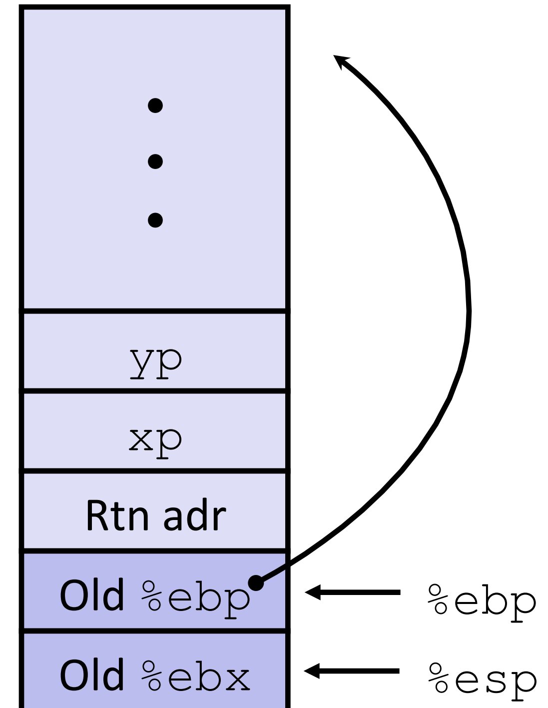

```
swap:
    pushl %ebp
    movl  %esp,%ebp
    pushl %ebx
```

swap Setup #3

Entering Stack

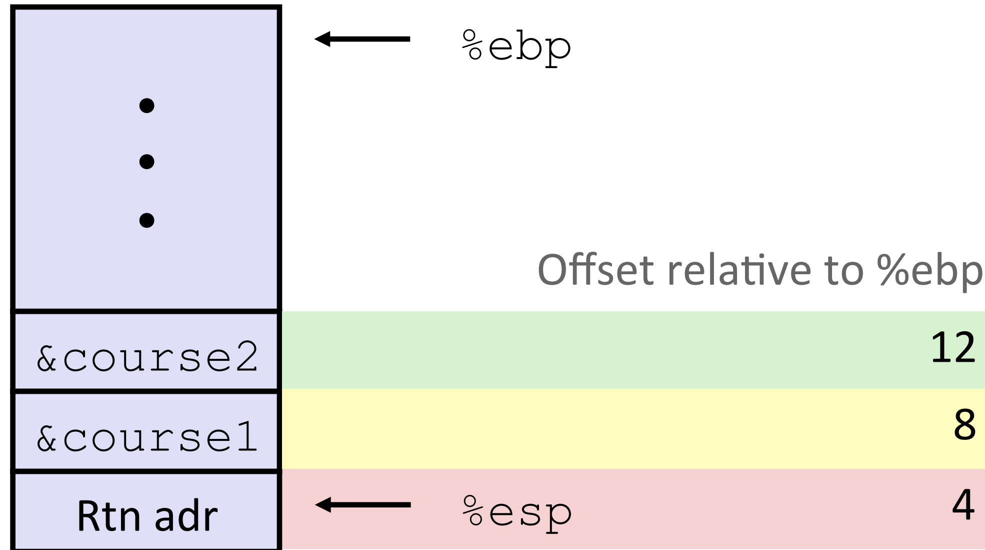


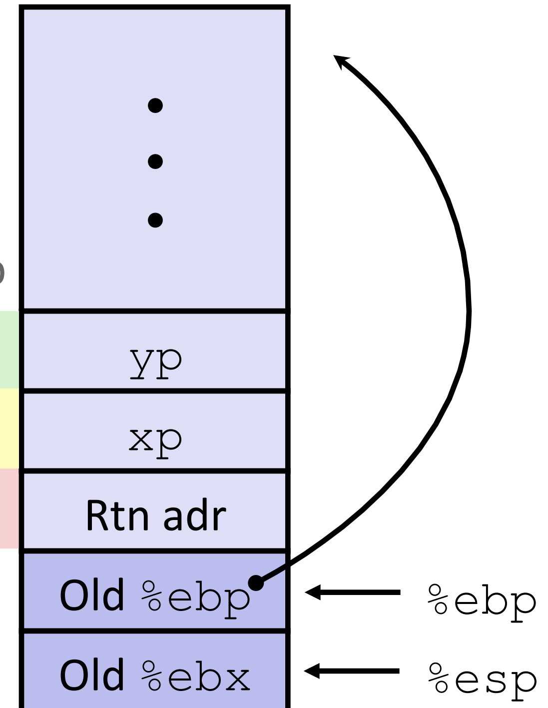
Resulting Stack




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

swap Body

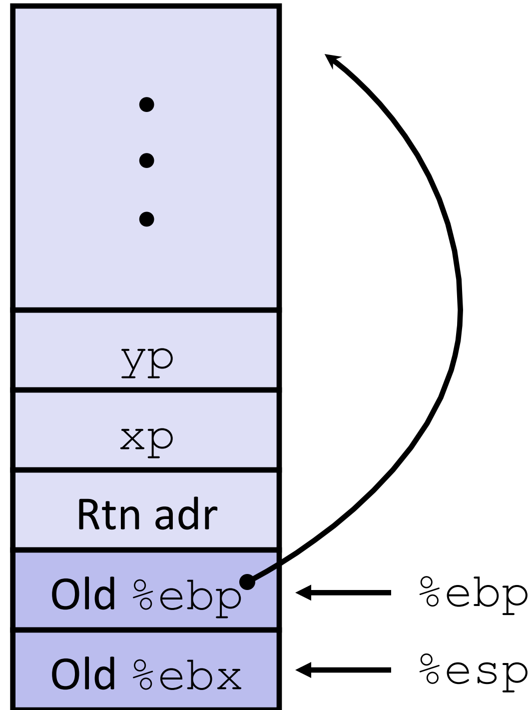





```
movl 8(%ebp),%edx    # get xp
movl 12(%ebp),%ecx   # get yp
. . .
```

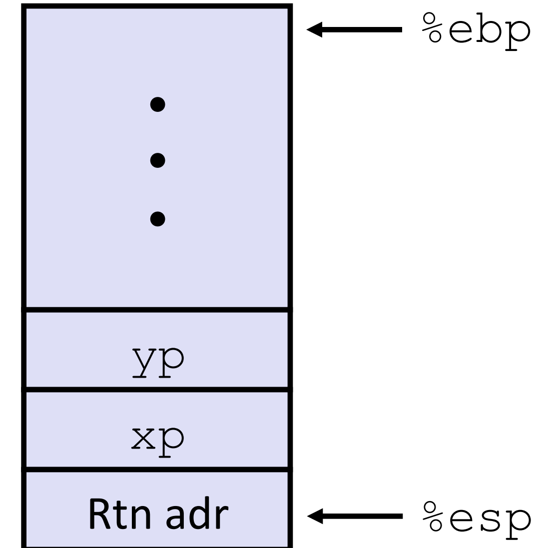
swap Finish

Stack Before Finish




```
popl    %ebx
popl    %ebp
```

Resulting Stack



- Observation
  - Saved and restored register %ebx
  - Not so for %eax, %ecx, %edx

Disassembled swap

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

Calling Code

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

Today

- Control: Condition codes
- Conditional branches
- Loops
- Switch statements
- **IA 32 Procedures**
  - Stack Structure
  - Calling Conventions
  - Illustrations of Recursion & Pointers

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 $18243, %edx
    • • •
    ret
```

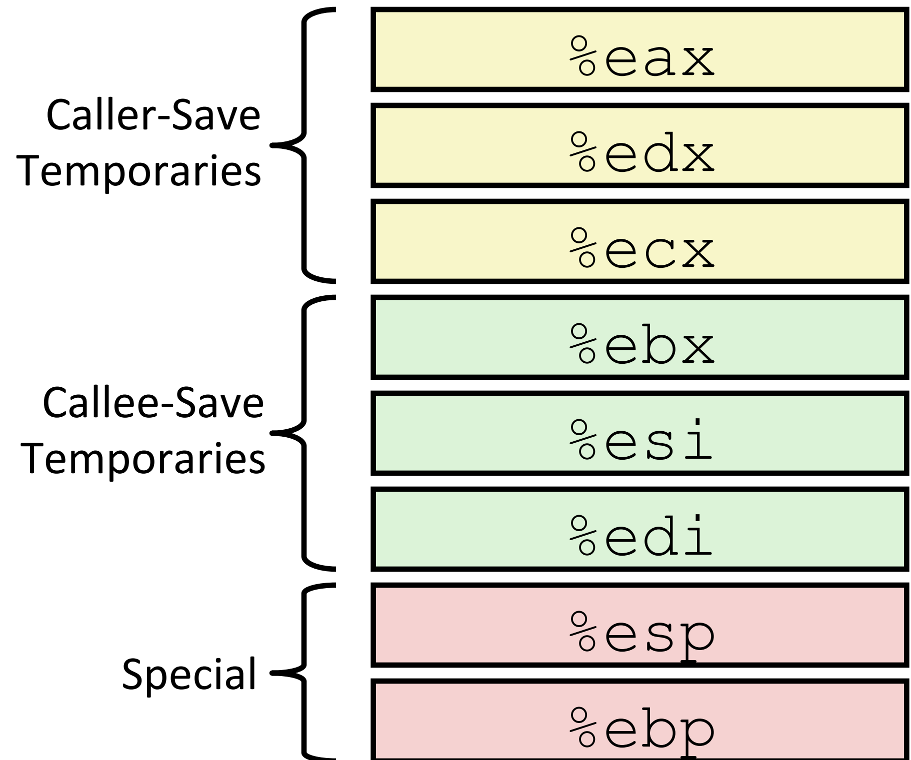
  - Contents of register `%edx` overwritten by `who`
  - This could be trouble ➙ something should be done!
    - Need some coordination

Register Saving Conventions

- **When procedure `yoo` calls `who`:**
  - `yoo` is the caller
  - `who` is the callee
- **Can register be used for temporary storage?**
- **Conventions**
  - "Caller Save"
    - Caller saves temporary values in its frame before the call
  - "Callee Save"
    - Callee saves temporary values in its frame before using

IA32/Linux+Windows Register Usage

- **%eax, %edx, %ecx**
  - Caller saves prior to call if values are used later
- **%eax**
  - also used to return integer value
- **%ebx, %esi, %edi**
  - Callee saves if wants to use them
- **%esp, %ebp**
  - special form of callee save
  - Restored to original values upon exit from procedure



Today

- Control: Condition codes
- Conditional branches
- Loops
- Switch statements
- IA 32 Procedures
  - Stack Structure
  - Calling Conventions
  - Illustrations of Recursion & Pointers

Recursive Function

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

- **Registers**
  - `%eax, %edx` used without first saving
  - `%ebx` used, but saved at beginning & restored at end

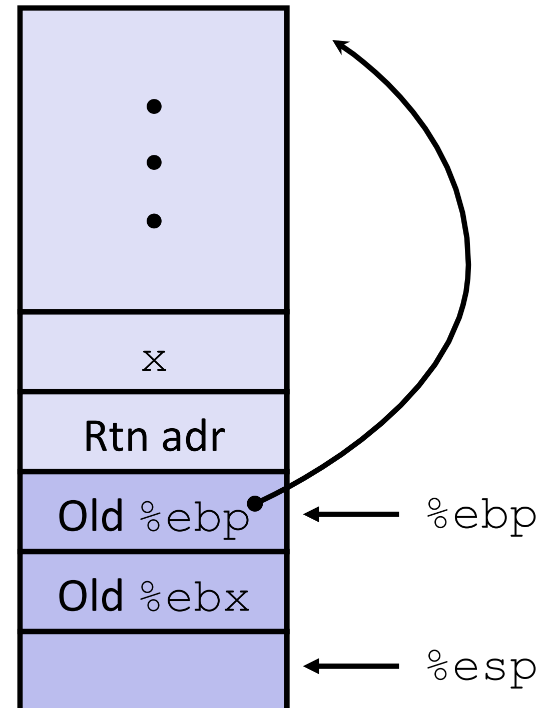
```
pcount_r:
    pushl  %ebp
    movl   %esp, %ebp
    pushl  %ebx
    subl   $4, %esp
    movl   8(%ebp), %ebx
    movl   $0, %eax
    testl  %ebx, %ebx
    je   .L3
    movl   %ebx, %eax
    shrl   %eax
    movl   %eax, (%esp)
    call   pcount_r
    movl   %ebx, %edx
    andl   $1, %edx
    leal   (%edx,%eax), %eax
.L3:
    addl   $4, %esp
    popl   %ebx
    popl   %ebp
    ret
```

Recursive Call #1

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

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

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



Recursive Call #2

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

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

- **Actions**
  - If x == 0, return
    - with `%eax` set to 0

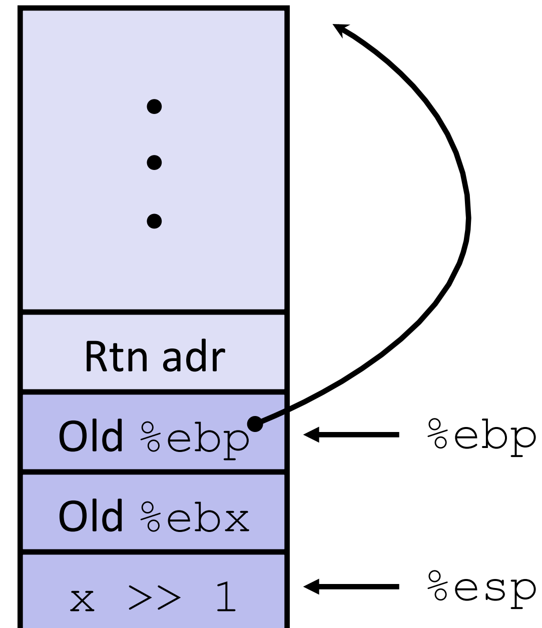
`%ebx` | x

Recursive Call #3

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

```
  • • •
movl   %ebx, %eax
shrl   %eax
movl   %eax, (%esp)
call   pcount_r
  • • •
```

- **Actions**
  - Store x >> 1 on stack
  - Make recursive call
- **Effect**
  - `%eax` set to function result
  - `%ebx` still has value of x



Recursive Call #4

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

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

- **Assume**
  - %eax holds value from recursive call
  - %ebx holds x

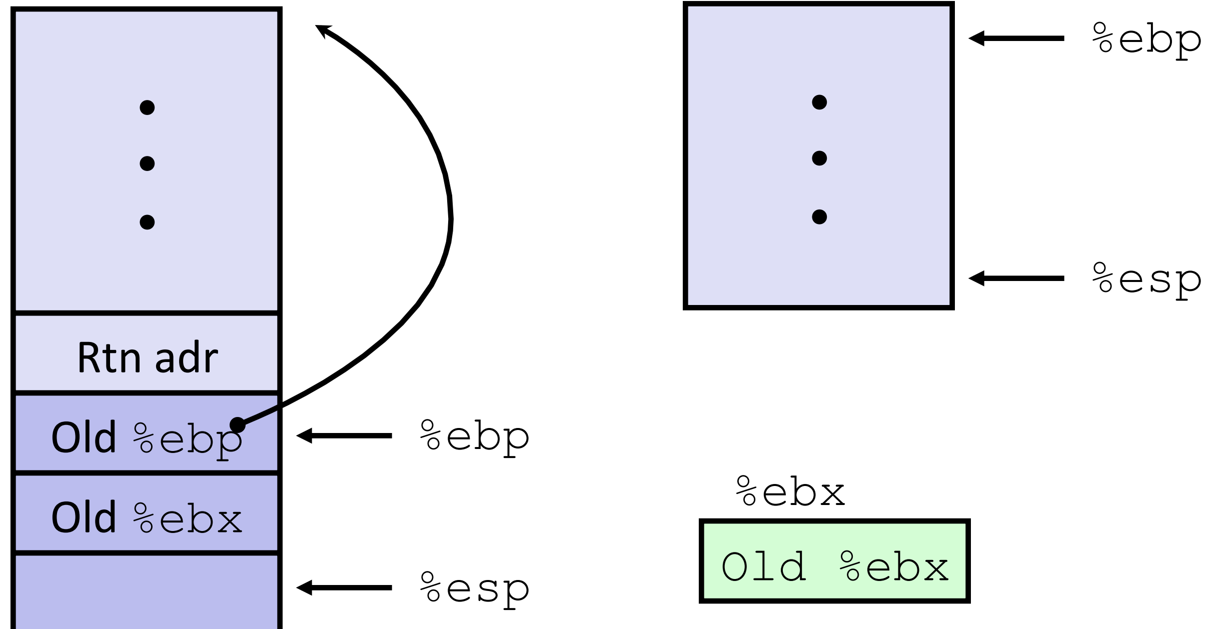
- **Actions**
  - Compute (x & 1) + computed value
- **Effect**
  - %eax set to function result

Recursive Call #5

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

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

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



Observations About Recursion

- **Handled Without Special Consideration**
  - Stack frames mean that each function call has private storage
    - Saved registers & local variables
    - Saved return pointer
  - Register saving conventions prevent one function call from corrupting another's data
  - Stack discipline follows call / return pattern
    - If P calls Q, then Q returns before P
    - Last-In, First-Out
- **Also works for mutual recursion**
  - P calls Q; Q calls P

Pointer Code

Generating Pointer

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

Referencing Pointer

```
/* Increment value by k */
void incrk(int *ip, int k) {
  *ip += k;
}
```

- **add3 creates pointer and passes it to `incrk`**

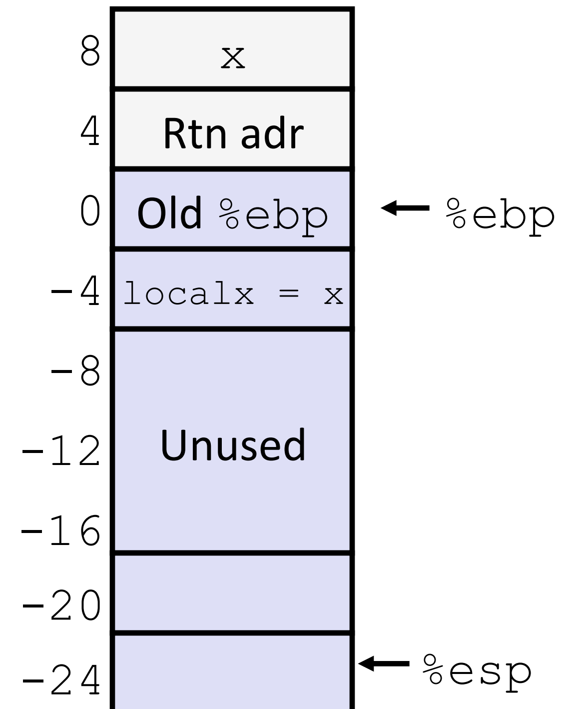
Creating and Initializing Local Variable

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

- **Variable localx must be stored on stack**
  - Because: Need to create pointer to it
- **Compute pointer as -4(%ebp)**

First part of add3

```
add3:
  pushl%ebp
  movl %esp, %ebp
  subl $24, %esp      # Alloc. 24 bytes
  movl 8(%ebp), %eax
  movl %eax, -4(%ebp)# Set localx to x
```



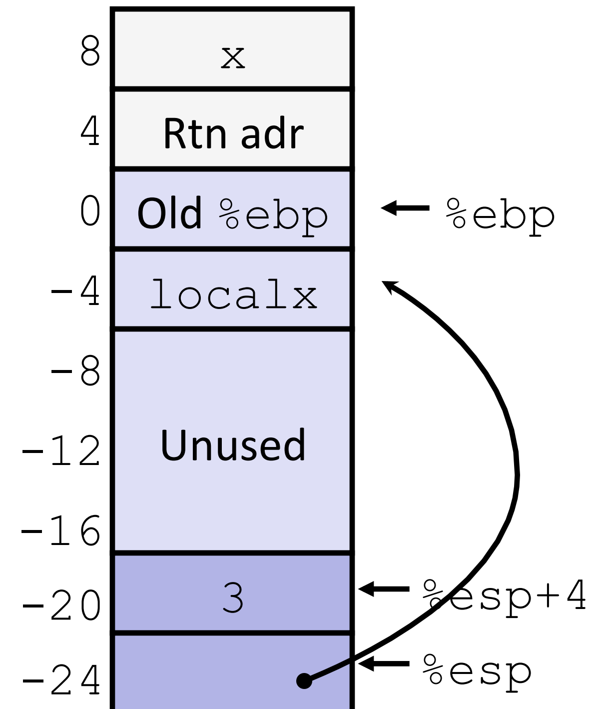
Creating Pointer as Argument

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

- Use leal instruction to compute address of localx

Middle part of add3

```
  movl $3, 4(%esp)     # 2nd arg = 3
  leal -4(%ebp), %eax# &localx
  movl %eax, (%esp)    # 1st arg = &localx
  call incrk
```



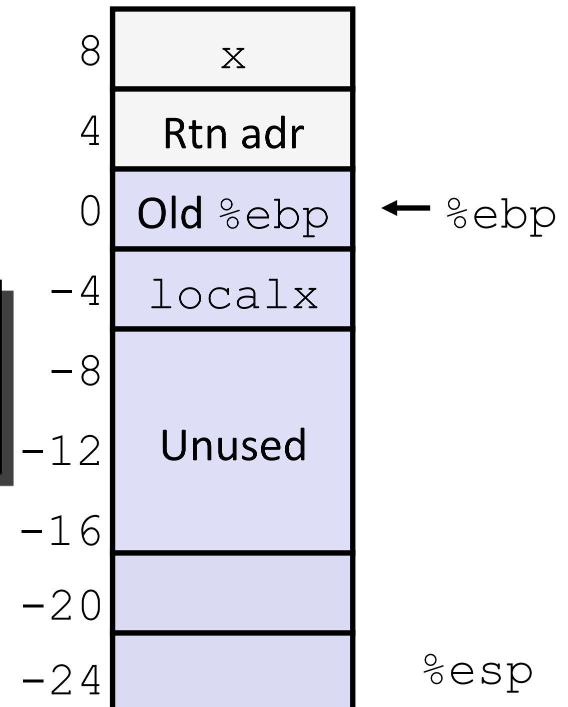
Retrieving local variable

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

- Retrieve localx from stack as return value

Final part of add3

```
movl -4(%ebp), %eax # Return val= localx
leave
ret
```



IA 32 Procedure Summary

- **Important Points**
  - Stack is the right data structure for procedure call / return
    - If P calls Q, then Q returns before P
- **Recursion (& mutual recursion) handled by normal calling conventions**
  - Can safely store values in local stack frame and in callee-saved registers
  - Put function arguments at top of stack
  - Result return in `%eax`
- **`Pointers are addresses of values`**
  - `On stack or global`