BBC Headline: Robots join the fight against cancer

Slashdot Headline: Robot Snakes To Fight Cancer Via Natural Orifice Surgery

"This is a medical robot, guided by a skilled surgeon and designed to get to places doctors are unable to reach without opening a patient up.

It is still only a prototype and has not yet been used on real patients - only in the lab.

But its designers, from OC Robotics in Bristol, are convinced that once ready and approved, it could help find and remove tumours."


http://www.bbc.co.uk/news/health-19653105

# Questions?

# Address Computation Examples

| | |
|---|---|
| %edx | 0xf000 |
| %ecx | 0x100 |

D(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+ D]

| Expression | Address Computation | Address |
|---|---|---|
| 0xc(%ecx,0xdb,2) | 100 + db*2 + c | 0x2C2 |

0x100 + (0xdb * 2) + 0xc = 0x2C2

More about calculator.

# Simple leal example

t1: %ecx
x: %edx
y: %eax

```
int t1 = x+y;

leal (%edx,%eax),%ecx
```

Same form as instruction that reads from memory to a register, but doesn't reference memory

movl S,D: D <- S
leal S,D: D <- &S

%edx contains value x

```
leal 7(%edx, %edx, 4), %eax      //sets %eax to 5x+7
```

%edx contains some address in memory

```
movl 7(%edx, %edx, 4), %eax      //sets %eax to M[7+ 5*(%edx)]
```

# Practice 3.9

In the following variant of the function of Figure 3.8(a), the expressions have been replaced by blanks:

```
int arith(int x,
          int y,
          int z)
{
    int t1 = _________;
    int t2 = _________;
    int t3 = _________;
    int t4 = _________;
    return t4;
}
```

The portion of the generated assembly code implementing these expressions is as follows:

```
x at %ebp+8, y at %ebp+12, z at %ebp+16
  movl    12(%ebp), %eax
  xorl    8(%ebp), %eax
  sarl    $3, %eax
  notl    %eax
  subl    16(%ebp), %eax
```

Return value from function stored in %eax

Based on this assembly code, fill in the missing portions of the C code.

# Practice 3.9

In the following variant of the function of Figure 3.8(a), the expressions have been replaced by blanks:

```
int arith(int x,
          int y,
          int z)
{
    int t1 = _________;
    int t2 = _________;
    int t3 = _________;
    int t4 = _________;
    return t4;
}
```

```
t1 = x^y
t2 = t1 >> 3
t3 = ~t2
t4 = t3-z
```

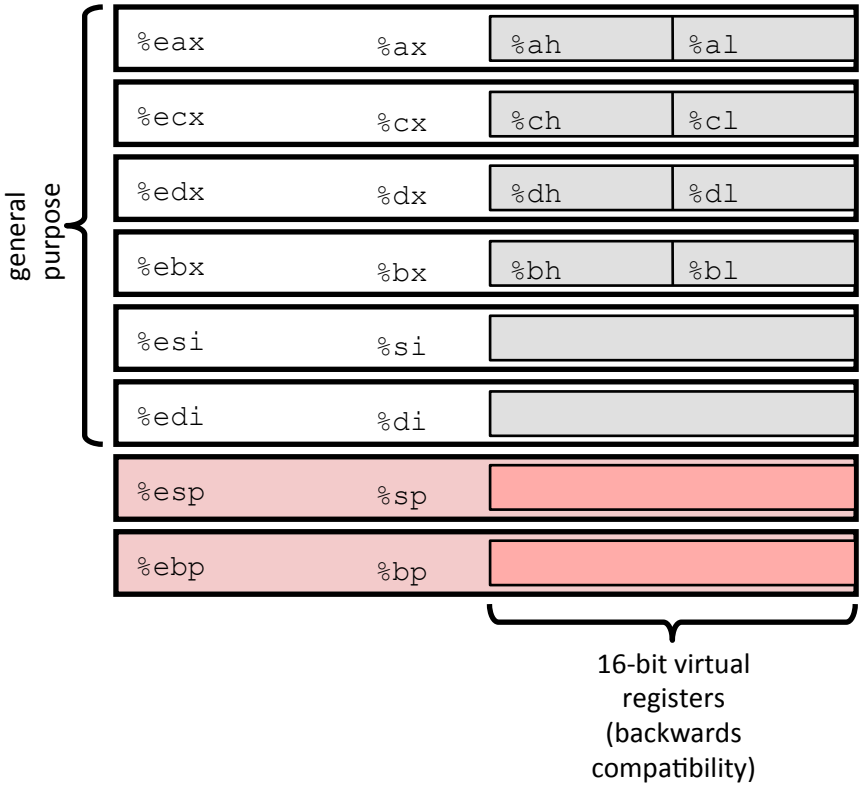
The portion of the generated assembly code implementing these expressions is as follows:

```
x at %ebp+8, y at %ebp+12, z at %ebp+16
  movl    12(%ebp), %eax
  xorl    8(%ebp), %eax
  sarl    $3, %eax
  notl    %eax
  subl    16(%ebp), %eax
```

Return value from function stored in %eax

Based on this assembly code, fill in the missing portions of the C code.

# Integer Registers (IA32)

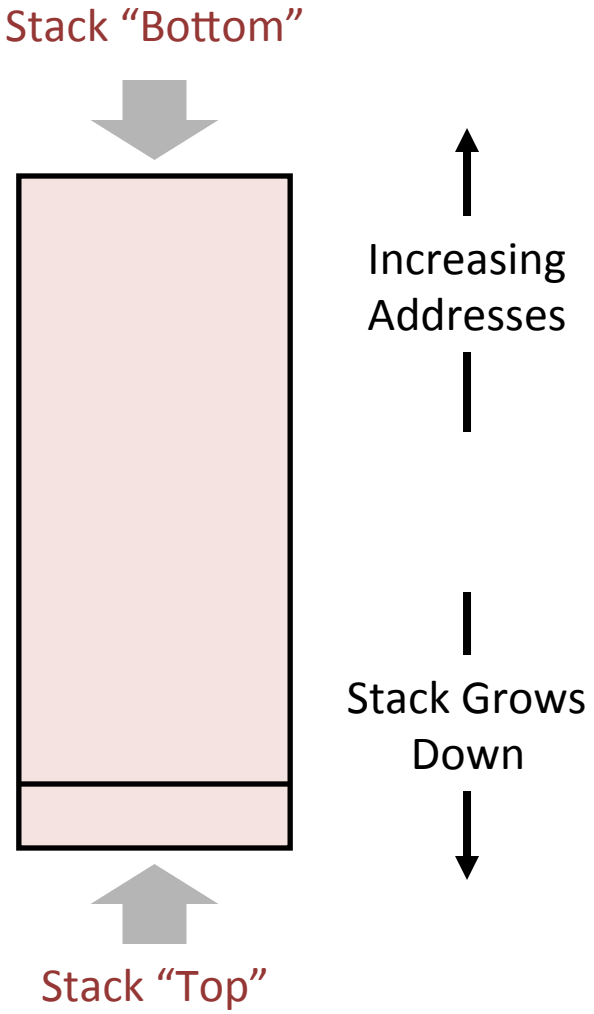



%dh = CD
%eax = 98765432

movb %dh, %al
    %eax = 987654CD  set byte
movsbl %dh, %eax
    %eax = FFFFFFCD  sign extend
movzbl %dh, %eax
    %eax = 000000CD  zero extend

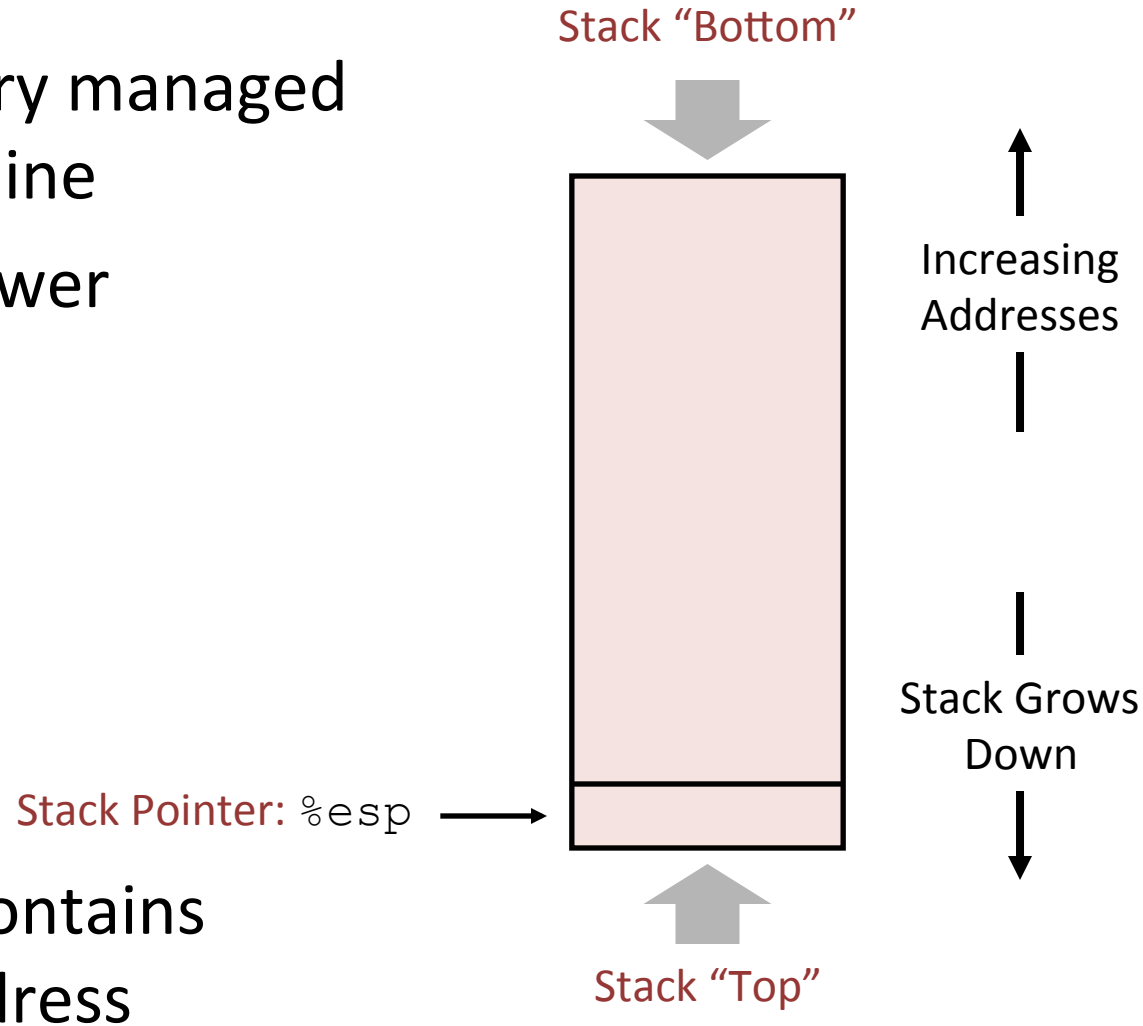
# IA32 Stack

- LIFO *(not LMAO)*
- In RAM
- Grows "downward"
- Nothing explicitly points to the bottom (e.g., highest memory address)
- Stack overflow: pushing too much info onto the stack
  - Blue screen of Death!



# IA32 Stack

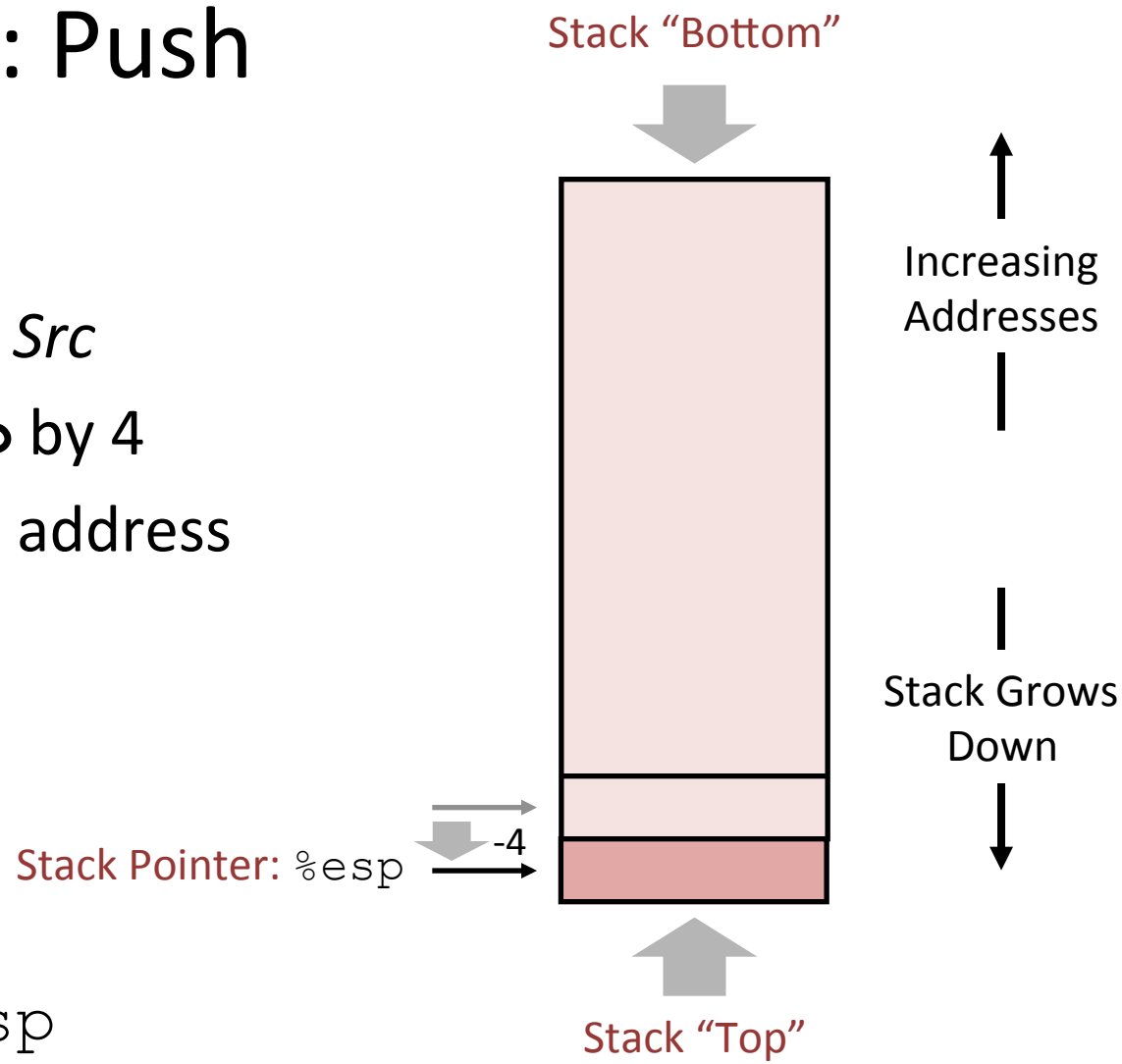
- Region of memory managed with stack discipline
- Grows toward lower addresses




- 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`**

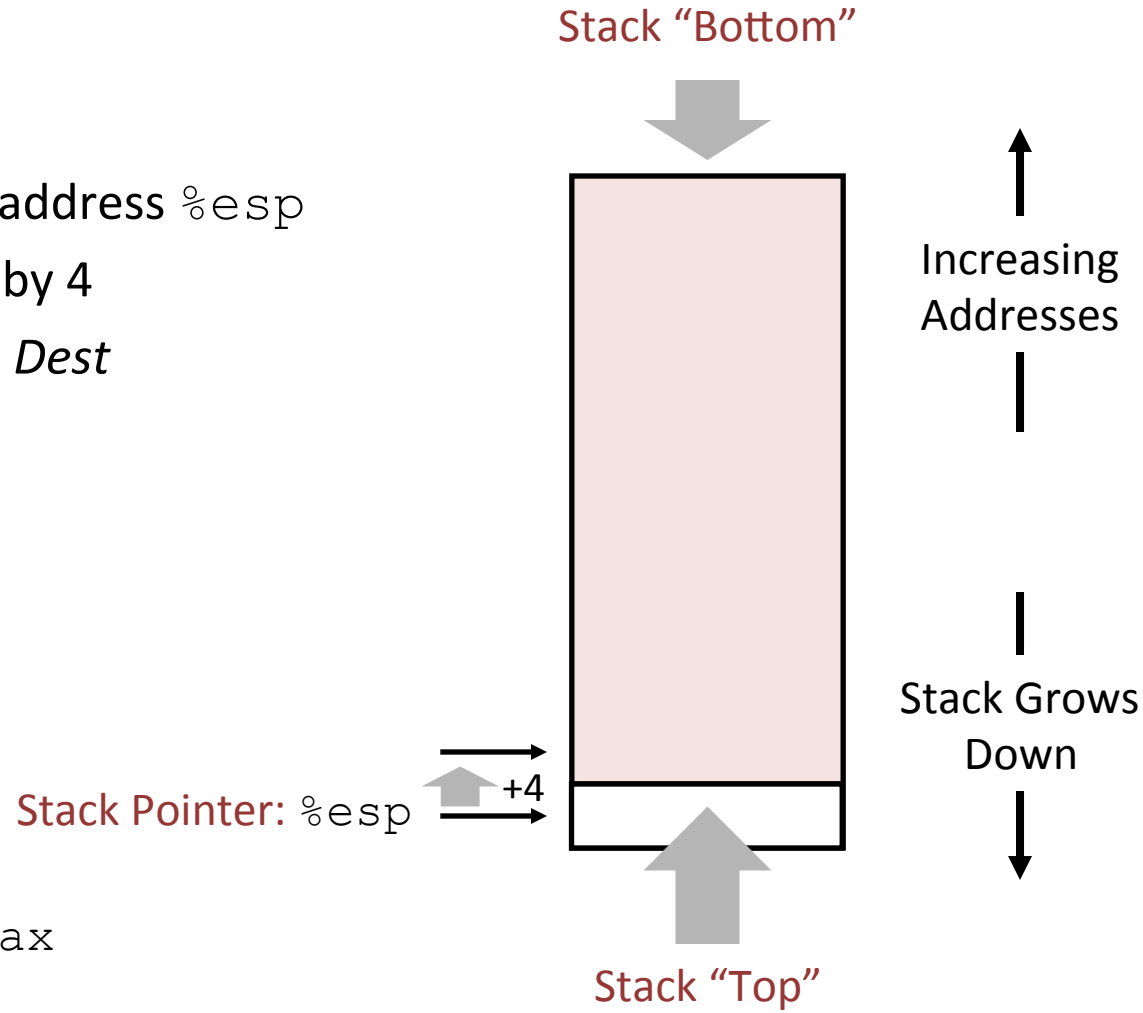



`pushl %ebp:`

- `subl $4, %esp`
- `movl %ebp, (%esp)`

# IA32 Stack: Pop

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




```
popl %eax:
    movl (%esp), %eax
    addl $4, %esp
```

# Condition Codes (Explicit Setting: Compare)

- Explicit Setting by Compare Instruction

  **`cmpl`** *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 Compare Instruction

  **`testl`** *Src2,Src1*

  **`testl 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)`**

# Conditional Branch Example

```
int absdiff(int x, int y)
{
  if (x <= y)
    return y-x;
  return x-y;
}
```

# Conditional Branch Example

```
int absdiff(int x, int y)
{
  if (x <= y)
    return y-x;
  return x-y;
}
```

```
int absdiff2(int x, int y)
{
  if (x <= y)
    return y-x;
  else
    return x-y;
}
```

# Conditional Branch Example

```
int absdiff(int x, int y)
{
  if (x <= y)
    return y-x;
  return x-y;
}
```

```
int absdiff2(int x, int y)
{
  if (x <= y)
    return y-x;
  else
    return x-y;
}
```

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

# Conditional Branch Example (Cont.)

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

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave
    ret
.L7:
    subl    %edx, %eax
    jmp     .L8
```

# 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;
Done:
  . . .
Else:
  val = Else-Expr;
  goto 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

# General Form with Conditional Move


## C Code

```
val = Test ? Then-Expr : Else-Expr;
```

## Conditional Move Version

```
val1  = Then-Expr;
val2  = Else-Expr;
val1  = val2 if !Test;
```

- Both values get computed
- Overwrite then-value with else-value if condition doesn't hold
- Don't use when:
  - Then or else expression have side effects
  - Then and else expression are to expensive

# Practice: X86 -> C

```
08048454 <_Z4funcPi>:
 8048454:    55                    push   %ebp
 8048455:    89 e5                 mov    %esp,%ebp
 8048457:    8b 45 08              mov    0x8(%ebp),%eax
 804845a:    c7 00 6d 01 00 00     movl   $0x16d,(%eax)
 8048460:    b8 00 00 00 00        mov    $0x0,%eax
 8048465:    5d                    pop    %ebp
 8048466:    c3                    ret
```

| Instruction | | Synonym | Jump condition | Description |
|---|---|---|---|---|
| jmp | *Label* | | 1 | Direct jump |
| jmp | **Operand* | | 1 | Indirect jump |
| je | *Label* | jz | ZF | Equal / zero |
| jne | *Label* | jnz | ~ZF | Not equal / not zero |
| js | *Label* | | SF | Negative |
| jns | *Label* | | ~SF | Nonnegative |
| jg | *Label* | jnle | ~(SF ^ OF) & ~ZF | Greater (signed >) |
| jge | *Label* | jnl | ~(SF ^ OF) | Greater or equal (signed >=) |
| jl | *Label* | jnge | SF ^ OF | Less (signed <) |
| jle | *Label* | jng | (SF ^ OF) \| ZF | Less or equal (signed <=) |
| ja | *Label* | jnbe | ~CF & ~ZF | Above (unsigned >) |
| jae | *Label* | jnb | ~CF | Above or equal (unsigned >=) |
| jb | *Label* | jnae | CF | Below (unsigned <) |
| jbe | *Label* | jna | CF \| ZF | Below or equal (unsigned <=) |

- jmp *%eax: jump to the value in %eax
- jmp *(%eax): jump to the value at the address pointed to by %eax

```
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;
}
```

# Switch Statement Example

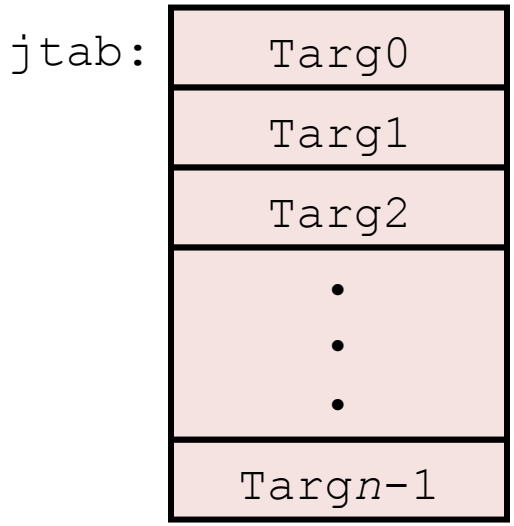
- Multiple case labels
  - Here: 5, 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4

# Jump Table Structure

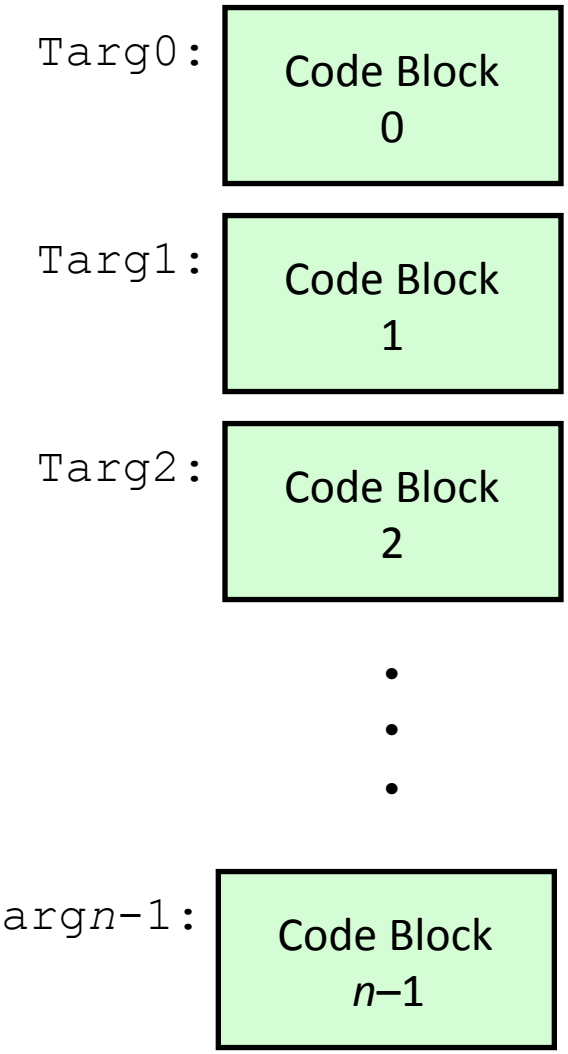
Switch Form

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

Jump Table




Jump Targets




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;
}
```

Setup:

```
switch_eg:
   pushl %ebp              # Setup
   movl  %esp, %ebp        # Setup
   pushl %ebx              # Setup
   movl  $1, %ebx          # w = 1
   movl  8(%ebp), %edx     # edx = x
   movl  16(%ebp), %ecx    # ecx = z
   cmpl  $6, %edx
   ja    .L61
   jmp   *.L62(,%edx,4)
```

# 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
.L62:
  .long   .L61   # x = 0
  .long   .L56   # x = 1
  .long   .L57   # x = 2
  .long   .L58   # x = 3
  .long   .L61   # x = 4
  .long   .L60   # x = 5
  .long   .L60   # x = 6
```

Setup:

```
switch_eg:
   pushl %ebp              # Setup
   movl  %esp, %ebp        # Setup
   pushl %ebx              # Setup
   movl  $1, %ebx          # w = 1
   movl  8(%ebp), %edx     # edx = x
   movl  16(%ebp), %ecx    # ecx = z
   cmpl  $6, %edx          # x:6
   ja    .L61              # if > goto default
   jmp   *.L62(,%edx,4)    # goto JTab[x]
```

*Indirect jump*

# Assembly Setup Explanation

- Table Structure
  - Each target requires 4 bytes
  - Base address at **`.L62`**

Jump table

```
.section .rodata
   .align 4
.L62:
  .long    .L61   # x = 0
  .long    .L56   # x = 1
  .long    .L57   # x = 2
  .long    .L58   # x = 3
  .long    .L61   # x = 4
  .long    .L60   # x = 5
  .long    .L60   # x = 6
```

- Jumping

  **Direct:** **`jmp .L61`**
  - Jump target is denoted by label **`.L61`**

  **Indirect:** **`jmp *.L62(,%edx,4)`**
  - Start of jump table: **`.L62`**
  - Must scale by factor of 4 (labels have 32-bit = 4 Bytes on IA32)
  - Fetch target from effective Address **`.L61 + edx*4`**
    - Only for $0 \le \mathbf{x} \le 6$

# Jump Table

Jump table

```
.section .rodata
   .align 4
.L62:
  .long    .L61   # x = 0
  .long    .L56   # x = 1
  .long    .L57   # x = 2
  .long    .L58   # x = 3
  .long    .L61   # x = 4
  .long    .L60   # x = 5
  .long    .L60   # x = 6
```

```
switch(x) {
case 1:        // .L56
    w = y*z;
    break;
case 2:        // .L57
    w = y/z;
    /* Fall Through */
case 3:        // .L58
    w += z;
    break;
case 5:
case 6:        // .L60
    w -= z;
    break;
default:       // .L61
    w = 2;
}
```

# Code Blocks (Partial)

```
switch(x) {
  . . .
case 2:       // .L57
    w = y/z;
    /* Fall Through */
case 3:       // .L58
    w += z;
    break;
  . . .
default:      // .L61
    w = 2;
}
```

```
.L61:  // Default case
   movl  $2, %ebx     # w = 2
   movl  %ebx, %eax   # Return w
   popl  %ebx
   leave
   ret
.L57:  // Case 2:
   movl  12(%ebp), %eax  # y
   cltd                  # Div prep
   idivl %ecx            # y/z
   movl  %eax, %ebx # w = y/z
# Fall through
.L58:  // Case 3:
   addl  %ecx, %ebx # w+= z
   movl  %ebx, %eax # Return w
   popl  %ebx
   leave
   ret
```

# Code Blocks (Rest)

```
switch(x) {
case 1:       // .L56
    w = y*z;
    break;
  . . .
case 5:
case 6:       // .L60
    w -= z;
    break;
  . . .
}
```

```
.L60: // Cases 5&6:
  subl  %ecx, %ebx  # w -= z
  movl  %ebx, %eax  # Return w
  popl  %ebx
  leave
  ret
.L56: // Case 1:
  movl  12(%ebp), %ebx # w = y
  imull %ecx, %ebx     # w*= z
  movl  %ebx, %eax  # Return w
  popl  %ebx
  leave
  ret
```

# IA32 Object Code

- Setup
  - Label **`.L61`** becomes address **`0x8048630`**
  - Label **`.L62`** becomes address **`0x80488dc`**

## Assembly Code

```
switch_eg:
  . . .
   ja     .L61            # if > goto default
   jmp    *.L62(,%edx,4)  # goto JTab[x]
```

## Disassembled Object Code

```
08048610 <switch_eg>:
 . . .
 8048622:  77 0c                    ja     8048630
 8048624:  ff 24 95 dc 88 04 08     jmp    *0x80488dc(,%edx,4)
```

# IA32 Object Code (cont.)

- Jump Table
  - Doesn't show up in disassembled code
  - Can inspect using GDB

```
(gdb) x/7xw 0x80488dc
```

  - E<u>x</u>amine <u>7</u> he<u>x</u>adecimal format "<u>*w*</u>ords" (4-bytes each)
  - Use command "**`help x`**" to get format documentation

```
0x80488dc:
   0x08048630
   0x08048650
   0x0804863a
   0x08048642
   0x08048630
   0x08048649
   0x08048649
```

# Disassembled Targets

```
8048630:        bb 02 00 00 00          mov    $0x2,%ebx
8048635:        89 d8                   mov    %ebx,%eax
8048637:        5b                      pop    %ebx
8048638:        c9                      leave
8048639:        c3                      ret
804863a:        8b 45 0c                mov    0xc(%ebp),%eax
804863d:        99                      cltd
804863e:        f7 f9                   idiv   %ecx
8048640:        89 c3                   mov    %eax,%ebx
8048642:        01 cb                   add    %ecx,%ebx
8048644:        89 d8                   mov    %ebx,%eax
8048646:        5b                      pop    %ebx
8048647:        c9                      leave
8048648:        c3                      ret
8048649:        29 cb                   sub    %ecx,%ebx
804864b:        89 d8                   mov    %ebx,%eax
804864d:        5b                      pop    %ebx
804864e:        c9                      leave
804864f:        c3                      ret
8048650:        8b 5d 0c                mov    0xc(%ebp),%ebx
8048653:        0f af d9                imul   %ecx,%ebx
8048656:        89 d8                   mov    %ebx,%eax
8048658:        5b                      pop    %ebx
8048659:        c9                      leave
804865a:        c3                      ret
```

# Matching Disassembled Targets

0x08048630
0x08048650
0x0804863a
0x08048642
0x08048630
0x08048649
0x08048649

```
8048630:        bb 02 00 00 00          mov
8048635:        89 d8                   mov
8048637:        5b                      pop
8048638:        c9                      leave
8048639:        c3                      ret
804863a:        8b 45 0c                mov
804863d:        99                      cltd
804863e:        f7 f9                   idiv
8048640:        89 c3                   mov
8048642:        01 cb                   add
8048644:        89 d8                   mov
8048646:        5b                      pop
8048647:        c9                      leave
8048648:        c3                      ret
8048649:        29 cb                   sub
804864b:        89 d8                   mov
804864d:        5b                      pop
804864e:        c9                      leave
804864f:        c3                      ret
8048650:        8b 5d 0c                mov
8048653:        0f af d9                imul
8048656:        89 d8                   mov
8048658:        5b                      pop
8048659:        c9                      leave
804865a:        c3                      ret
```


Carol never wore her safety goggles.
She said, YOLO instead...
STOP YOLO-ing
SCHOOL.FAILBLOG.ORG


WE DISAPPROVE OF YOUR POST
AND YOUR LYNX
IHASAHOTDOG.COM BY

# Procedure Control Flow

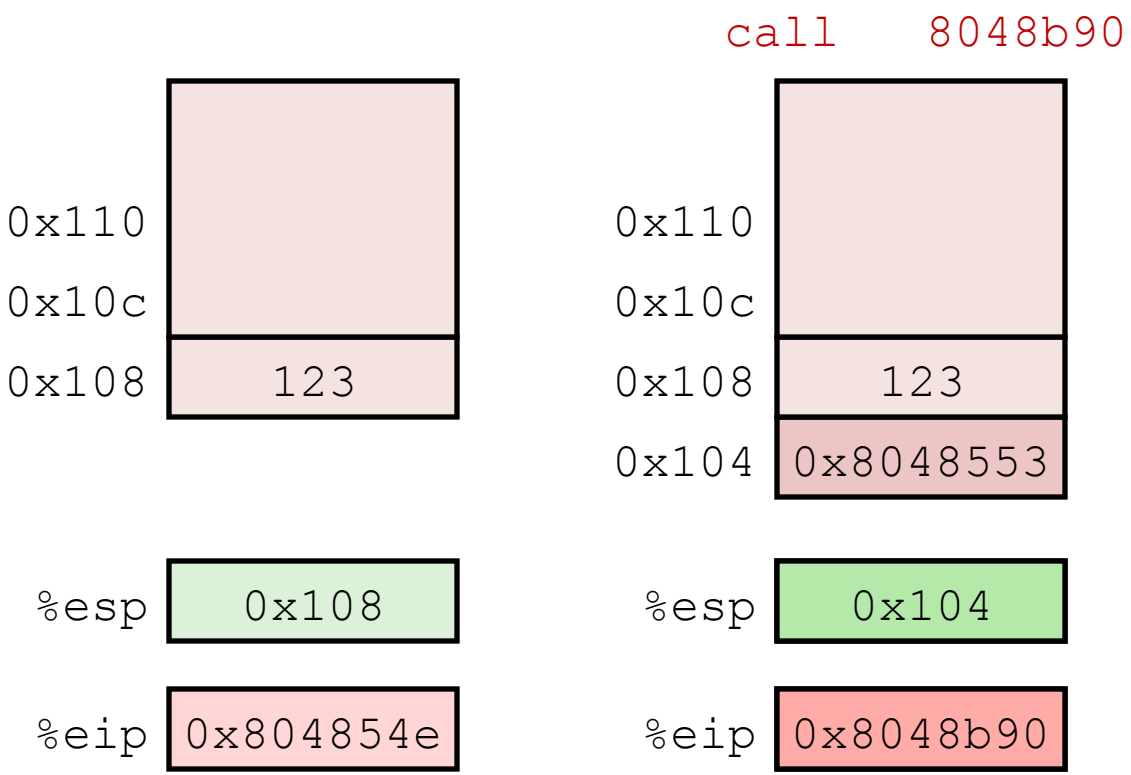
- Use stack to support procedure call and return
- Procedure call: `call label`
  - Push return address on stack
  - Jump to ***label***
- Return address:
  - Address of instruction beyond **`call`**
  - Example from disassembly

```
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

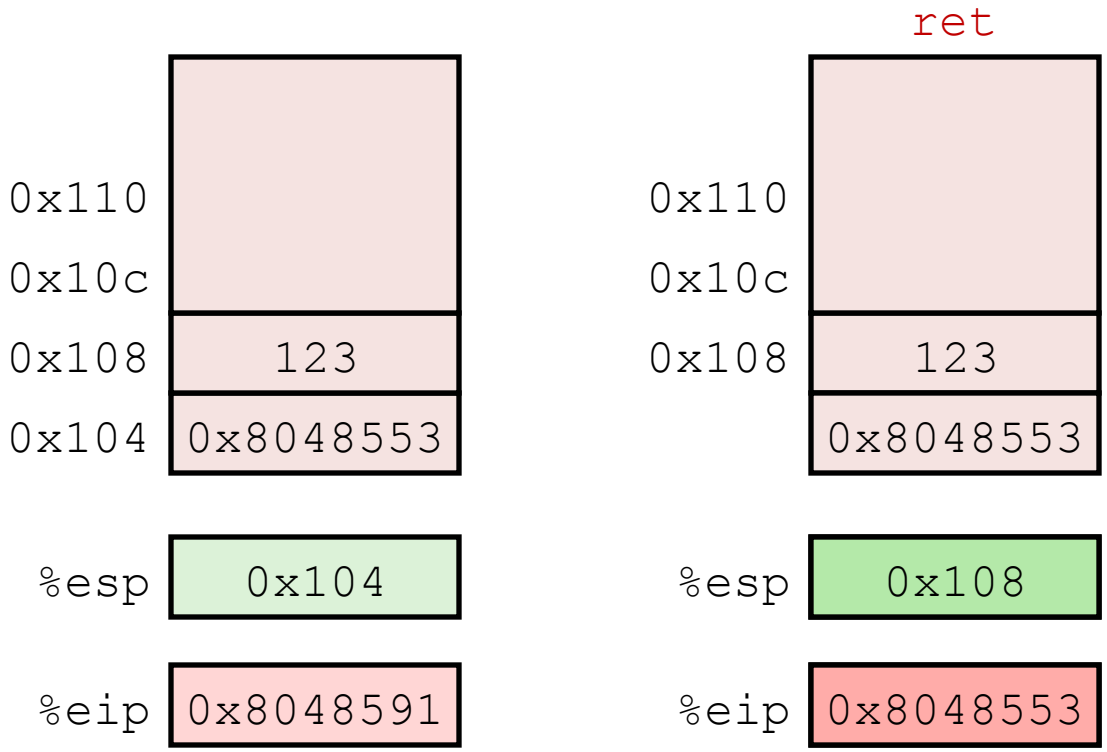
```
804854e: e8 3d 06 00 00   call   8048b90 <main>
8048553: 50               pushl  %eax
```




`%eip:` *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

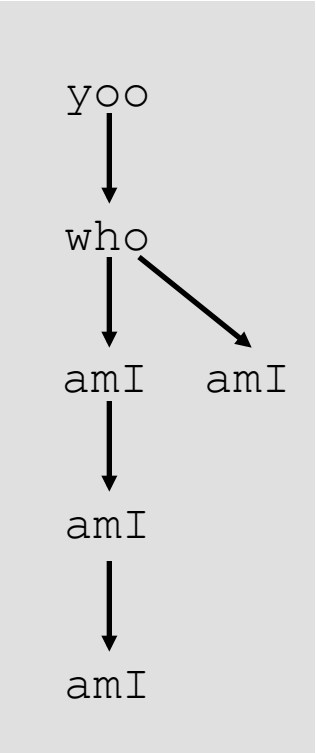
```
yoo(…)
{
    •
    •
    who();
    •
    •
}
```

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

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

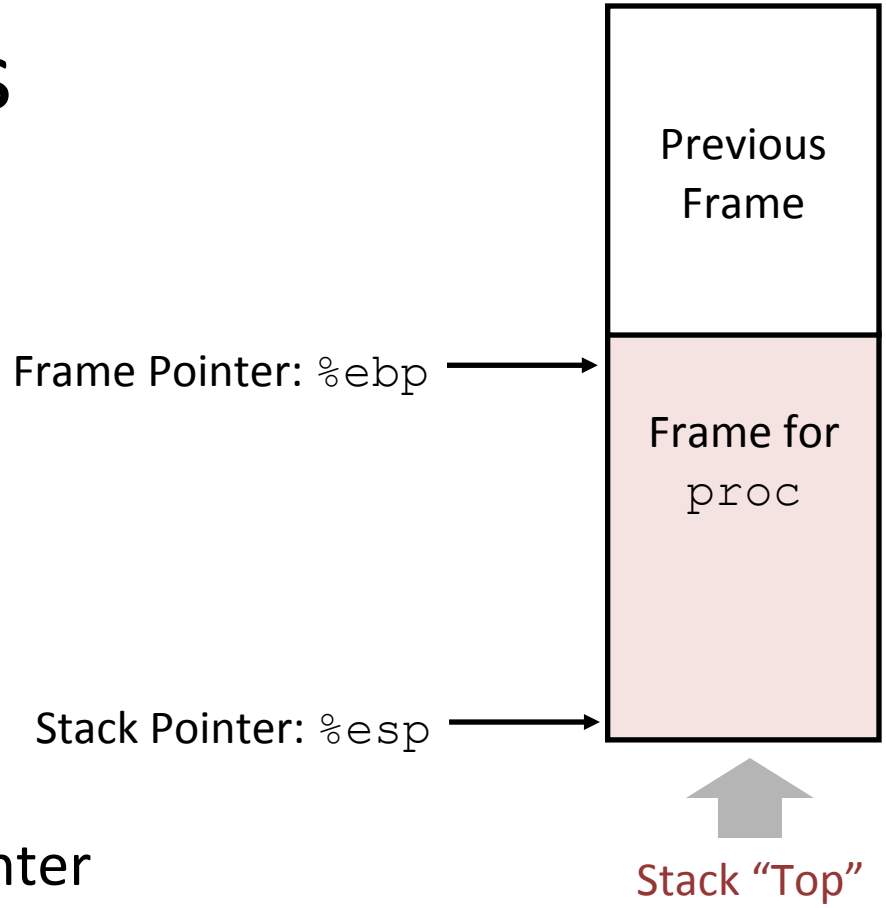
Procedure `amI` is recursive

Example
Call Chain



# Stack Frames

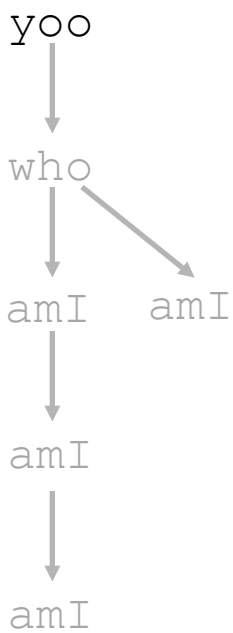
- Contents
  - Local variables
  - Return information
  - Temporary space

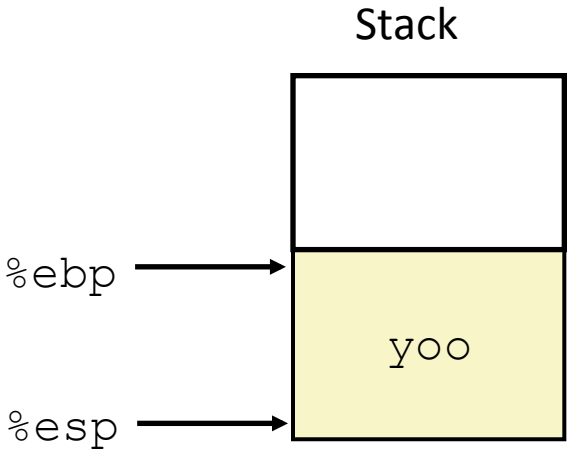



- Management
  - Space allocated when enter procedure
    - "Set-up" code
  - Deallocated when return
    - "Finish" code

# Example

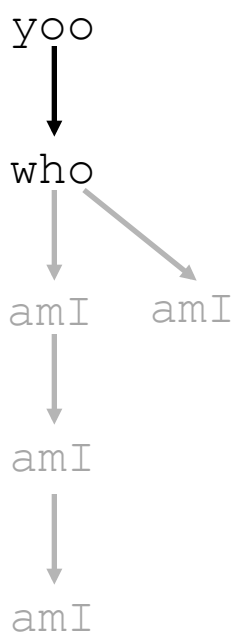
```
yoo(...)
{
    •
    •
    who();
    •
    •
}
```

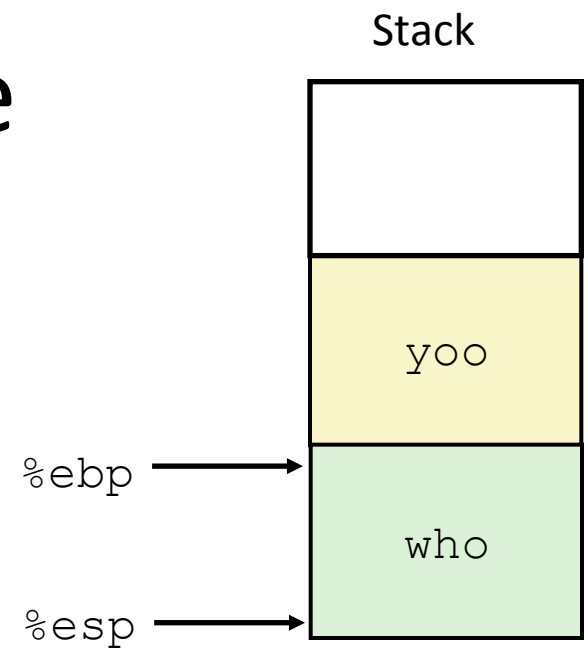


# Example

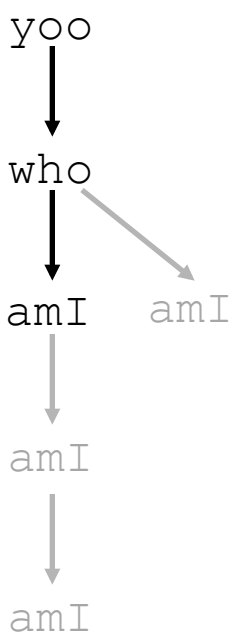
```
who(...)
{
    . . .
    amI();
    . . .
    amI();
    . . .
}
```

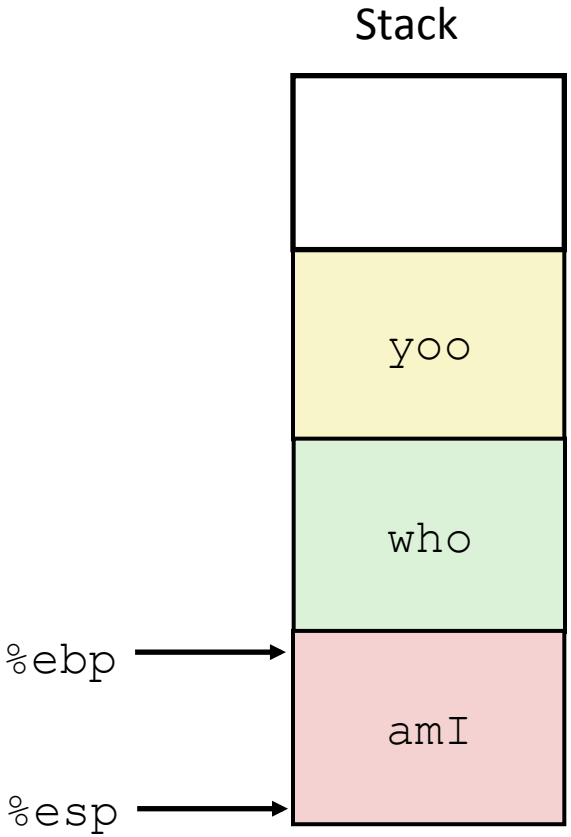


# Example

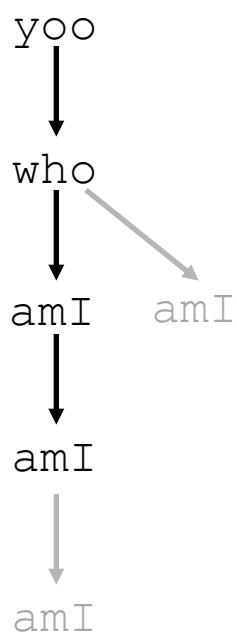
```
amI(...)
{
    •
    •
    amI();
    •
    •
}
```

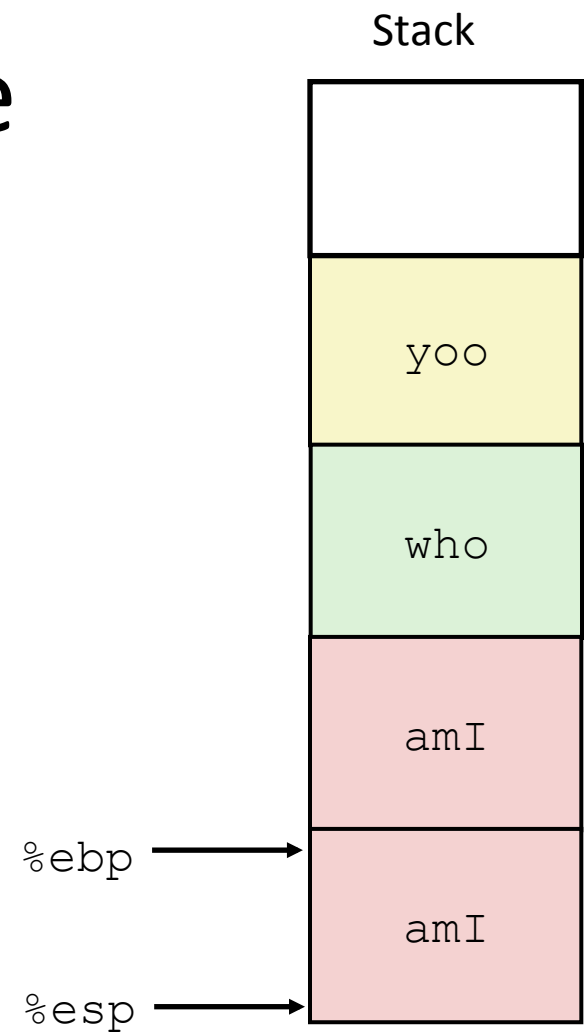


# Example

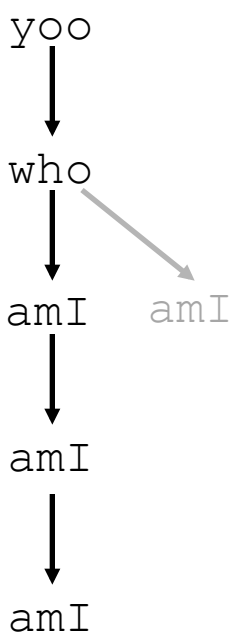
```
amI(...)
{
    •
    •
    amI();
    •
    •
}
```

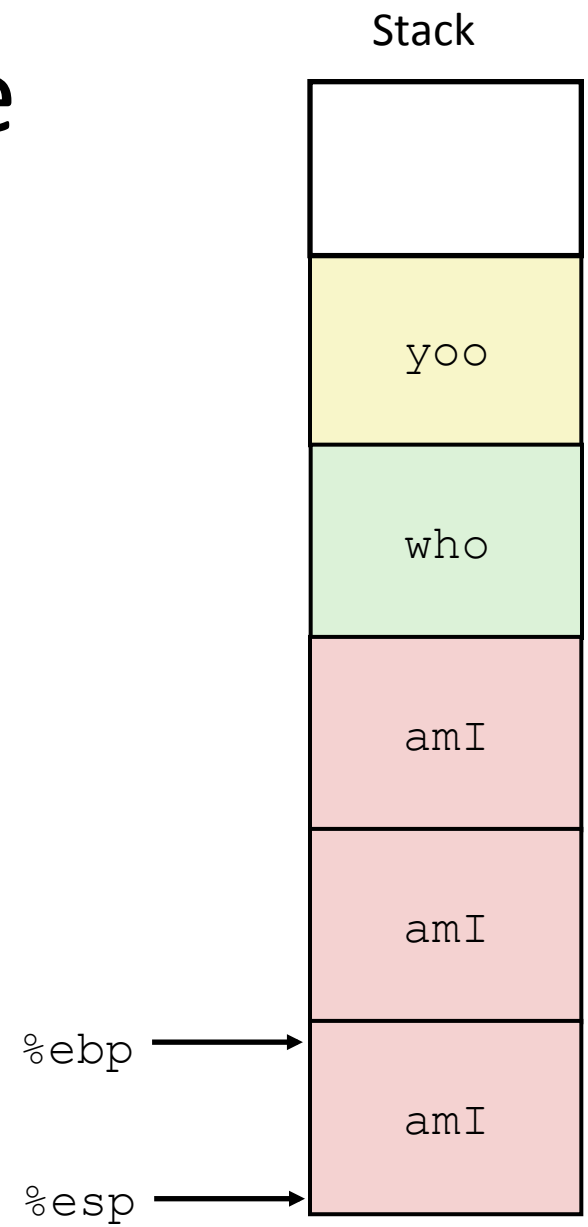


# Example

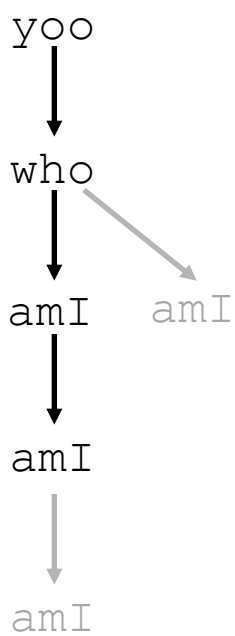
```
amI(...)
{
    •
    •
    amI();
    •
    •
}
```

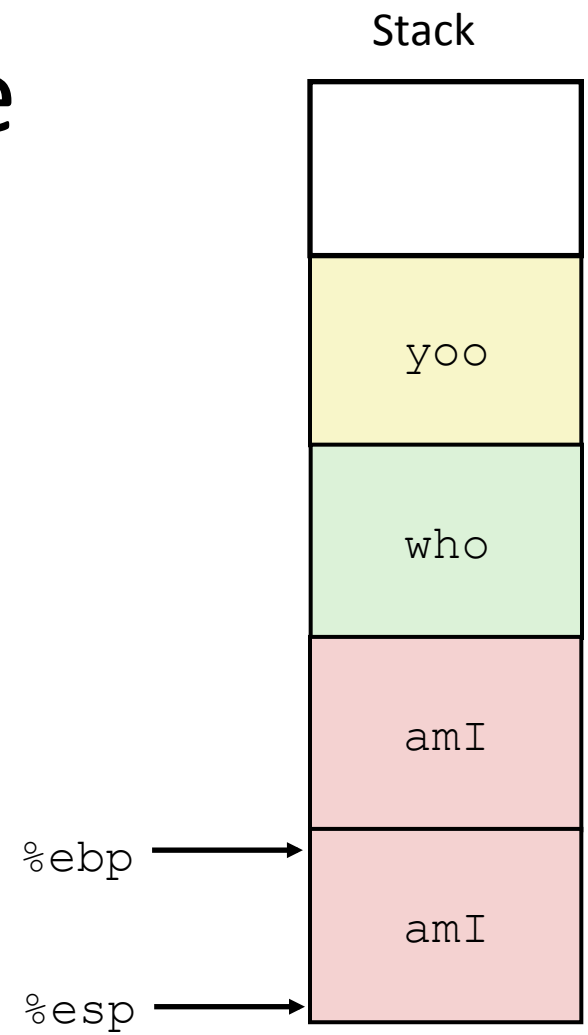


# Example

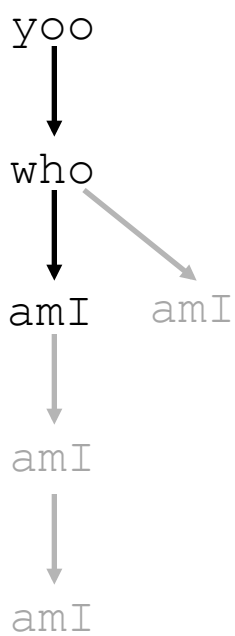
```
amI(...)
{
    •
    •
    amI();
    •
    •
}
```

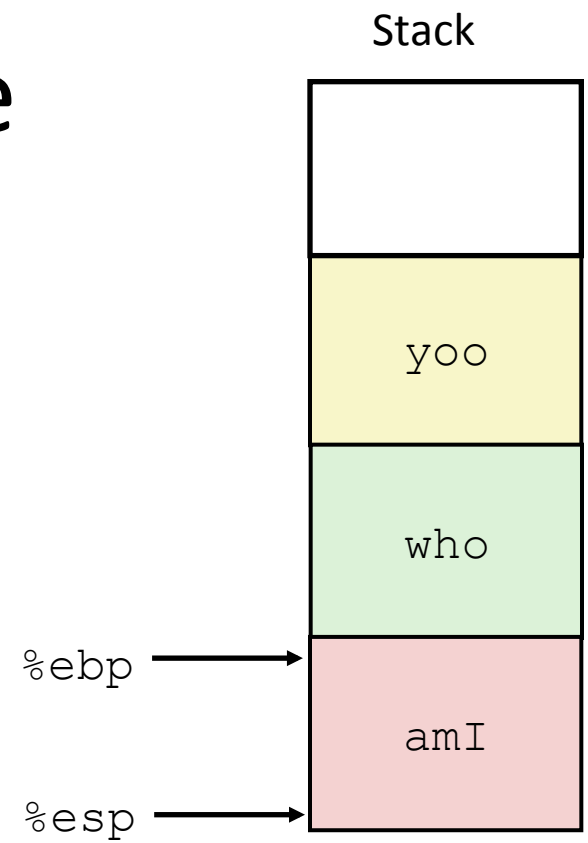


# Example

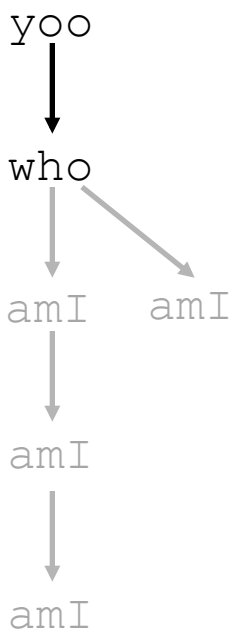
```
amI(...)
{
    •
    •
    amI();
    •
    •
}
```

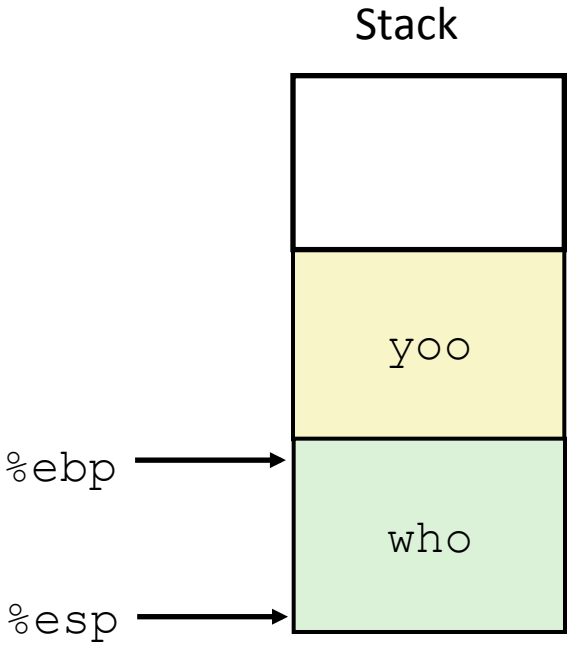


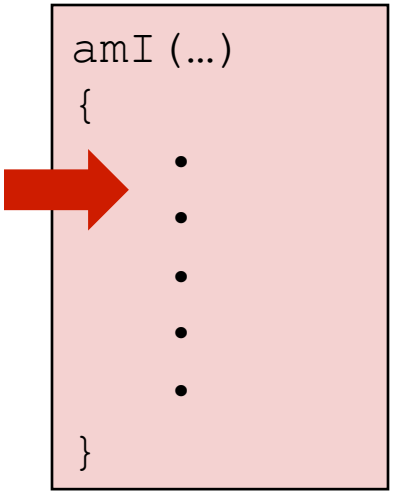
# Example

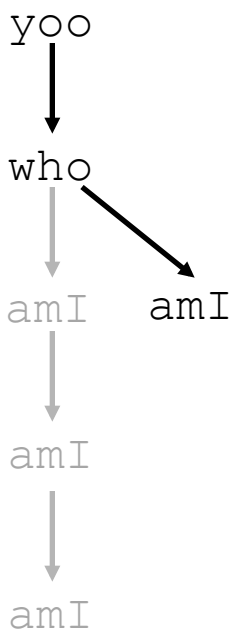
```
who(...)
{
    . . .
    amI();
    . . .
    amI();
    . . .
}
```

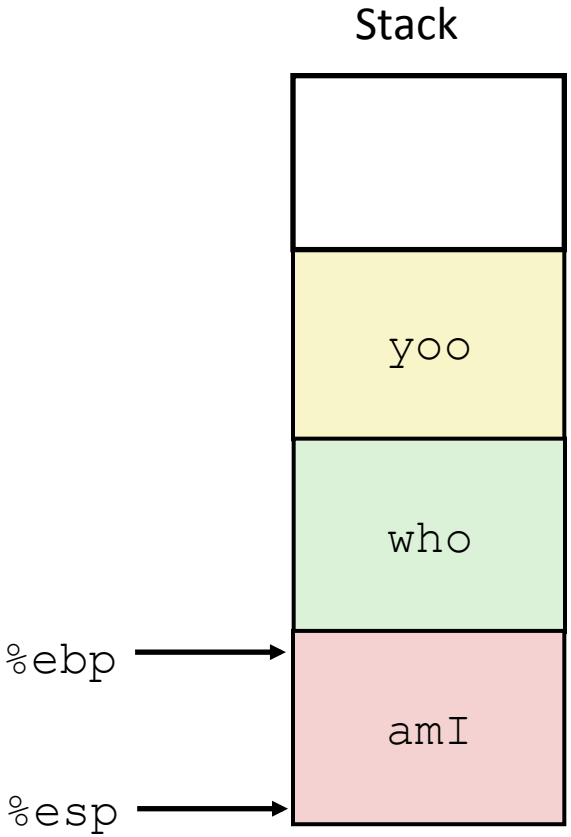


# Example

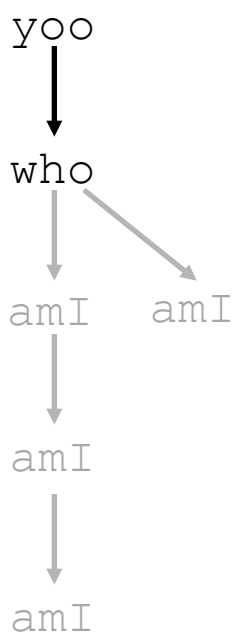


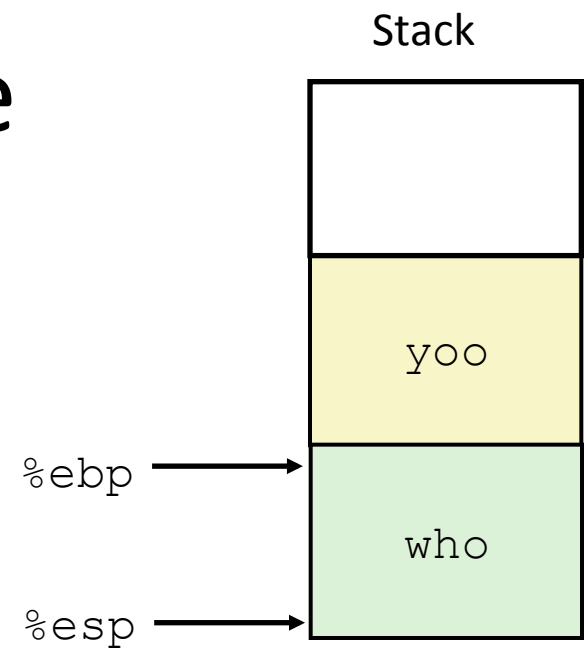



# Example

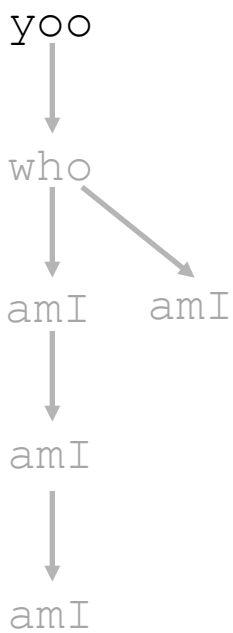
```
who(...)
{
    . . .
    amI();
    . . .
    amI();
    . . .
}
```

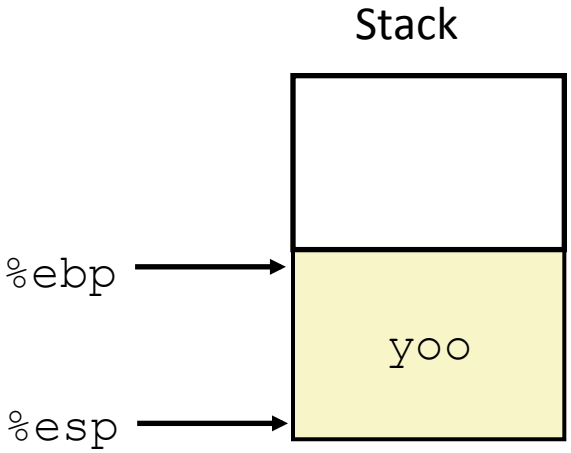


# Example

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



- 3.32
- 3.33