“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

<table>
<thead>
<tr>
<th>%edx</th>
<th>0xf000</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>0x100</td>
</tr>
</tbody>
</table>

### Expression Example

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xc(%ecx, 0xdb, 2)</td>
<td>100 + db*2 + c</td>
<td>0x2C2</td>
</tr>
</tbody>
</table>

\[ D(Rb, Ri, S) \quad \text{Mem[Reg[Rb] + S*Reg[Ri] + D]} \]

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In the following variant of the function of Figure 3.8(a), the expressions have been replaced by blanks:

```c
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
1    movl  12(%ebp), %eax
2    xorl  8(%ebp), %eax
3    sarl  $3, %eax
4    notl  %eax
5    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In the following variant of the function of Figure 3.8(a), the expressions have been replaced by blanks:

```c
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
1    movl 12(%ebp), %eax
2    xorl 8(%ebp), %eax
3    sarl $3, %eax
4    notl %eax
5    subl 16(%ebp), %eax
```

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

1. \( t_1 = x^y \)
2. \( t_2 = t_1 \gg 3 \)
3. \( t_3 = \sim t_2 \)
4. \( t_4 = t_3 - z \)

Return value from function stored in %eax
Integer Registers (IA32)

%dh = CD
%eax = 98765432

movb %dh, %al
%eax = 987654CD set byte

movsbl %dh, %eax
%eax = FFFFFFFCD 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**
  
  `cmp1  Src2,Src1`

  `cmp1  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
  \texttt{testl~Src2,Src1}
  \texttt{testl~b,a} like computing \texttt{a\&b} without setting destination

  – \textbf{CF set} if carry out from most significant bit (used for unsigned comparisons)
  – \textbf{ZF set} if \texttt{a == b}
  – \textbf{SF set} if \texttt{(a-b) < 0} (as signed)
  – \textbf{OF set} if two’s complement (signed) overflow
    \texttt{(a>0 && b<0 && (a-b)<0) || (a<0 && b>0
    && (a-b)>0)}
int absdiff(int x, int y) {
  if (x <= y)
    return y-x;
  return x-y;
}"
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 absdiff(int x, int y) {
    if (x <= y)
        return y-x;
    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;
}

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;
}

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

\[
\text{val} = \text{Test} \ ? \ \text{Then-Expr} \ : \ \text{Else-Expr};
\]

Conditional Move Version

\[
\text{val1} = \text{Then-Expr}; \\
\text{val2} = \text{Else-Expr}; \\
\text{val1} = \text{val2 if } !\text{Test};
\]

• 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 too 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 Table

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

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

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

```c
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;
}
```
Jump Table Structure

Switch Form

```java
switch(x) {
    case val_0:  
        Block 0
        break;
    case val_1:  
        Block 1
        break;
    // ... 
    case val_n-1: 
        Block n-1
        break;
}
```

Jump Table

- `jtab:`
  - `Targ0`
  - `Targ1`
  - `Targ2`
  - `Targn-1`

Jump Targets

- `Targ0:` Code Block 0
- `Targ1:` Code Block 1
- `Targ2:` Code Block 2
- `Targn-1:` Code Block n-1

Approximate Translation

```java
target = JTab[x];
goto *target;
```
Switch Statement Example (IA32)

```c
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
    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;
}

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]

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
]
Assembly Setup Explanation

• Table Structure
  – Each target requires 4 bytes
  – Base address at .L62

• 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 ≤ x ≤ 6

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
```
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;
}
switch(x) {
  
  default:      // .L61
    w = 2;

  case 2:       // .L57
    w = y/z;
    /* Fall Through */
  case 3:       // .L58
    w += z;
    break;

  . . .
}

.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
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/7yw 0x80488dc
```

- Examine 7 hexadecimal format “words” (4-bytes each)
- Use command “help x” to get format documentation

```
0x80488dc:
  0x08048630
  0x08048650
  0x0804863a
  0x08048642
  0x08048630
  0x08048649
  0x08048649
```
## Disassembled Targets

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048630:</td>
<td>bb 02 00 00 00 00</td>
<td>mov $0x2,%ebx</td>
</tr>
<tr>
<td>8048635:</td>
<td>89 d8</td>
<td>mov %ebx,%eax</td>
</tr>
<tr>
<td>8048637:</td>
<td>5b</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>8048638:</td>
<td>c9</td>
<td>leave</td>
</tr>
<tr>
<td>8048639:</td>
<td>c3</td>
<td>ret</td>
</tr>
<tr>
<td>804863a:</td>
<td>8b 45 0c</td>
<td>mov 0xc(%ebp),%eax</td>
</tr>
<tr>
<td>804863d:</td>
<td>99</td>
<td>cltd</td>
</tr>
<tr>
<td>804863e:</td>
<td>f7 f9</td>
<td>idiv %ecx</td>
</tr>
<tr>
<td>8048640:</td>
<td>89 c3</td>
<td>mov %eax,%ebx</td>
</tr>
<tr>
<td>8048642:</td>
<td>01 cb</td>
<td>add %ecx,%ebx</td>
</tr>
<tr>
<td>8048644:</td>
<td>89 d8</td>
<td>mov %ebx,%eax</td>
</tr>
<tr>
<td>8048646:</td>
<td>5b</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>8048647:</td>
<td>c9</td>
<td>leave</td>
</tr>
<tr>
<td>8048648:</td>
<td>c3</td>
<td>ret</td>
</tr>
<tr>
<td>8048649:</td>
<td>29 cb</td>
<td>sub %ecx,%ebx</td>
</tr>
<tr>
<td>804864b:</td>
<td>89 d8</td>
<td>mov %ebx,%eax</td>
</tr>
<tr>
<td>804864d:</td>
<td>5b</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>804864e:</td>
<td>c9</td>
<td>leave</td>
</tr>
<tr>
<td>804864f:</td>
<td>c3</td>
<td>ret</td>
</tr>
<tr>
<td>8048650:</td>
<td>8b 5d 0c</td>
<td>mov 0xc(%ebp),%ebx</td>
</tr>
<tr>
<td>8048653:</td>
<td>0f af d9</td>
<td>imul %ecx,%ebx</td>
</tr>
<tr>
<td>8048656:</td>
<td>89 d8</td>
<td>mov %ebx,%eax</td>
</tr>
<tr>
<td>8048658:</td>
<td>5b</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>8048659:</td>
<td>c9</td>
<td>leave</td>
</tr>
<tr>
<td>804865a:</td>
<td>c3</td>
<td>ret</td>
</tr>
</tbody>
</table>
## Matching Disassembled Targets

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly Instructions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08048630</td>
<td>bb 02 00 00 00</td>
<td>mov</td>
</tr>
<tr>
<td>0x08048635</td>
<td>89 d8</td>
<td>mov</td>
</tr>
<tr>
<td>0x08048637</td>
<td>5b</td>
<td>pop</td>
</tr>
<tr>
<td>0x08048638</td>
<td>c9</td>
<td>leave</td>
</tr>
<tr>
<td>0x08048639</td>
<td>c3</td>
<td>ret</td>
</tr>
<tr>
<td>0x0804863a</td>
<td>8b 45 0c</td>
<td>mov</td>
</tr>
<tr>
<td>0x0804863d</td>
<td>99</td>
<td>cltd</td>
</tr>
<tr>
<td>0x0804863e</td>
<td>f7 f9</td>
<td>idiv</td>
</tr>
<tr>
<td>0x08048640</td>
<td>89 c3</td>
<td>mov</td>
</tr>
<tr>
<td>0x08048642</td>
<td>01 cb</td>
<td>add</td>
</tr>
<tr>
<td>0x08048644</td>
<td>89 d8</td>
<td>mov</td>
</tr>
<tr>
<td>0x08048646</td>
<td>5b</td>
<td>pop</td>
</tr>
<tr>
<td>0x08048647</td>
<td>c9</td>
<td>leave</td>
</tr>
<tr>
<td>0x08048648</td>
<td>c3</td>
<td>ret</td>
</tr>
<tr>
<td>0x08048649</td>
<td>29 cb</td>
<td>sub</td>
</tr>
<tr>
<td>0x0804864b</td>
<td>89 d8</td>
<td>mov</td>
</tr>
<tr>
<td>0x0804864d</td>
<td>5b</td>
<td>pop</td>
</tr>
<tr>
<td>0x0804864e</td>
<td>c9</td>
<td>leave</td>
</tr>
<tr>
<td>0x0804864f</td>
<td>c3</td>
<td>ret</td>
</tr>
<tr>
<td>0x08048650</td>
<td>8b 5d 0c</td>
<td>mov</td>
</tr>
<tr>
<td>0x08048653</td>
<td>0f af d9</td>
<td>imul</td>
</tr>
<tr>
<td>0x08048656</td>
<td>89 d8</td>
<td>mov</td>
</tr>
<tr>
<td>0x08048658</td>
<td>5b</td>
<td>pop</td>
</tr>
<tr>
<td>0x08048659</td>
<td>c9</td>
<td>leave</td>
</tr>
<tr>
<td>0x0804865a</td>
<td>c3</td>
<td>ret</td>
</tr>
</tbody>
</table>
Carol never wore her safety goggles.

She said, YOLO instead...

STOP YOLO-ing

WE DISAPPROVE OF YOUR POST

AND YOUR LYNX
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 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

%esp: 0x108
%eip: 0x804854e

0x804854e:
  e8 3d 06 00 00  call 8048b90 <main>
  0x108

0x8048553:
  50 pushl %eax
  0x104

0x108:
  123

0x110:

0x10c:

0x110:
  123

%esp: 0x108
%eip: 0x804854e

call 8048b90

0x110:
  123

%esp: 0x104
%eip: 0x8048b90

%esp: program counter
Procedure Return Example

8048591: c3  

ret

%esp  0x104  
%eip  0x8048591

0x110  
0x10c  
0x108  123  
0x104  0x8048553

ret

%esp  0x108  
%eip  0x8048553

0x110  
0x10c  
0x108  123  
0x104  0x8048553

%esp  0x104  
%eip  0x8048591

%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

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

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

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

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

```
void (...) {
    who();
}
```
Example

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

Stack

%ebp  
%esp  

who

yoo

amI

amI

amI
Example

```
ami(...) {
    ·
    ·
    ami();
    ·
}
```

Stack

```
%ebp
%esp
ami
who
yoo
```
Example

```
ami(...) {
  ...
  ami();
  ...
}
```

Stack

- `yoo`
- `who`
- `ami`
- `ami`
- `ami`
Example

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

Stack

- yoo
- who
- amI
- amI
- amI
- %ebp
- %esp
Example

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

Example

```
yoo
who
amI
amI
```

Stack

```
yoo
who
%ebp
%esp
amI
```
Example

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

```
amI(...) {
  ...
  ...
  ...
  ...
}
```

-stack

```
yoo
who
amI
amI
amI
```

%ebp
%esp

-stack

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

Example
Example

```c
yoo (...) {
    who();
}
```
• 3.32
• 3.33