Lecture 9: Machine-Level Programming III: IA32 Procedures

Some slides adapted (and slightly modified) from:
• Clark Barrett
• Jinyang Li
• Randy Bryant
• Dave O’Hallaron
Switch uses jump tables

**Switch statement**

```java
switch(x) {
    case 0:
        Block 0
    case 1:
        Block 1
    ...
    case n-1:
        Block n-1
}
```

**Jump Table**

- Targ0
- Targ1
- Targn-1

**Jump Targets**

- Targ0: Code Block 0
- Targ1: Code Block 1
- Targn-1: Code Block n-1

**Approximate Translation**

```c
jtab: [Targ0, Targ1, ..., Targn-1];
goto *target;
```
Switch Statement Example (IA32)

```c
long switch_eg(long x, long y, long z)
{
    switch(x) {
        case 1:
            ... 
            default: ... 
    }
}
```

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]
...```
**Switch Statement Example (IA32)**

```
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]
```

*Indirect jump to* effective address

```
.L7 + eax*4
```

<table>
<thead>
<tr>
<th>Section</th>
<th>.rodata</th>
</tr>
</thead>
<tbody>
<tr>
<td>.align 4</td>
<td></td>
</tr>
<tr>
<td>.L7:</td>
<td></td>
</tr>
<tr>
<td>.long</td>
<td>.L2 # x = 0</td>
</tr>
<tr>
<td>.long</td>
<td>.L3 # x = 1</td>
</tr>
<tr>
<td>.long</td>
<td>.L4 # x = 2</td>
</tr>
<tr>
<td>.long</td>
<td>.L5 # x = 3</td>
</tr>
<tr>
<td>.long</td>
<td>.L2 # x = 4</td>
</tr>
<tr>
<td>.long</td>
<td>.L6 # x = 5</td>
</tr>
<tr>
<td>.long</td>
<td>.L6 # x = 6</td>
</tr>
</tbody>
</table>

**Jump table**

- Base address
- Target addr 4 bytes

**Read Only**
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
```

```
switch(x) {
    case 1:      // .L3
        w = y*z;
        break;
    case 2:      // .L4
        w = y/z;
        /* Fall Through */
    case 3:      // .L5
        w += z;
        break;
    case 5:
    case 6:      // .L6
        w -= z;
        break;
    default:     // .L2
        w = 2;
}
```
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>signed-ness</th>
<th>Flags</th>
<th>Short Jump Opcodes</th>
<th>Near Jump Opcodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>J0</td>
<td>Jump if overflow</td>
<td></td>
<td>OF = 1</td>
<td>70</td>
<td>OF 80</td>
</tr>
<tr>
<td>JNO</td>
<td>Jump if not overflow</td>
<td></td>
<td>OF = 0</td>
<td>71</td>
<td>OF 81</td>
</tr>
<tr>
<td>JS</td>
<td>Jump if sign</td>
<td>SF = 1</td>
<td></td>
<td>78</td>
<td>OF 88</td>
</tr>
<tr>
<td>JNS</td>
<td>Jump if not sign</td>
<td>SF = 0</td>
<td></td>
<td>79</td>
<td>OF 89</td>
</tr>
<tr>
<td>JE</td>
<td>Jump if equal</td>
<td></td>
<td>ZF = 1</td>
<td>74</td>
<td>OF 84</td>
</tr>
<tr>
<td>JE</td>
<td>Jump if zero</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNE</td>
<td>Jump if not equal</td>
<td></td>
<td>ZF = 0</td>
<td>75</td>
<td>OF 85</td>
</tr>
<tr>
<td>JNZ</td>
<td>Jump if not zero</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JB</td>
<td>Jump if below</td>
<td></td>
<td>CF = 1</td>
<td>72</td>
<td>OF 82</td>
</tr>
<tr>
<td>JNBE</td>
<td>Jump if not above or equal</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JC</td>
<td>Jump if carry</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNB</td>
<td>Jump if not below</td>
<td></td>
<td>CF = 0</td>
<td>73</td>
<td>OF 83</td>
</tr>
<tr>
<td>JAE</td>
<td>Jump if above or equal</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNC</td>
<td>Jump if not carry</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JBE</td>
<td>Jump if below or equal</td>
<td></td>
<td>CF = 1 or ZF = 1</td>
<td>76</td>
<td>OF 86</td>
</tr>
<tr>
<td>JNA</td>
<td>Jump if not below or equal</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JA</td>
<td>Jump if above</td>
<td></td>
<td>CF = 0 and ZF = 0</td>
<td>77</td>
<td>OF 87</td>
</tr>
<tr>
<td>JNBE</td>
<td>Jump if not below or equal</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JL</td>
<td>Jump if less</td>
<td></td>
<td>SF &lt;&gt; OF</td>
<td>7C</td>
<td>OF 8C</td>
</tr>
<tr>
<td>JNGE</td>
<td>Jump if not greater or equal</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JGE</td>
<td>Jump if greater or equal</td>
<td></td>
<td>SF = OF</td>
<td>7D</td>
<td>OF 8D</td>
</tr>
<tr>
<td>JNL</td>
<td>Jump if not greater or equal</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JLE</td>
<td>Jump if less or equal</td>
<td></td>
<td>ZF = 1 or SF &lt;&gt; OF</td>
<td>7E</td>
<td>OF 8E</td>
</tr>
<tr>
<td>JNL</td>
<td>Jump if not greater or less</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JG</td>
<td>Jump if greater or equal</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNG</td>
<td>Jump if not equal or less</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JF</td>
<td>Jump if parity</td>
<td></td>
<td>PF = 1</td>
<td>7A</td>
<td>OF 8A</td>
</tr>
<tr>
<td>JFP</td>
<td>Jump if parity even</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNP</td>
<td>Jump if not parity</td>
<td></td>
<td>PF = 0</td>
<td>7B</td>
<td>OF 8B</td>
</tr>
<tr>
<td>JPO</td>
<td>Jump if parity odd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JCE</td>
<td>Jump if %ECX register is 0</td>
<td></td>
<td>%CX = 0</td>
<td></td>
<td>E3</td>
</tr>
<tr>
<td>JECXZ</td>
<td>Jump if %ECX register is 0</td>
<td></td>
<td>%ECX = 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
IA32 Object Code

Assembly Code

```
switch_eg:
    ...
    ja .L2  # If unsigned > goto default
    jmp *0x8048660(,%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, inspect using GDB
    - (gdb) \texttt{x/7w 0x8048660}
  - Examine \texttt{7 hexadecimal format “words” (4-bytes each)}

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>\texttt{x}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80486600</td>
<td>0x8048422</td>
<td>0</td>
</tr>
<tr>
<td>0x80486640</td>
<td>0x8048432</td>
<td>1</td>
</tr>
<tr>
<td>0x80486680</td>
<td>0x804843b</td>
<td>2</td>
</tr>
<tr>
<td>0x804866c0</td>
<td>0x8048429</td>
<td>3</td>
</tr>
<tr>
<td>0x80486700</td>
<td>0x8048422</td>
<td>4</td>
</tr>
<tr>
<td>0x80486740</td>
<td>0x804844b</td>
<td>5</td>
</tr>
<tr>
<td>0x80486780</td>
<td>0x804844b</td>
<td>6</td>
</tr>
</tbody>
</table>
## Matching Disassembled Targets

<table>
<thead>
<tr>
<th>Value</th>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8048422</td>
<td><code>mov $0x2, %eax</code></td>
<td>8048422</td>
</tr>
<tr>
<td>0x8048432</td>
<td><code>jmp 8048453 &lt;switch_eg+0x43&gt;</code></td>
<td>8048427</td>
</tr>
<tr>
<td>0x804843b</td>
<td><code>mov $0x1, %eax</code></td>
<td>8048429</td>
</tr>
<tr>
<td>0x8048429</td>
<td><code>xchg %ax, %ax</code></td>
<td>804842e</td>
</tr>
<tr>
<td>0x8048432</td>
<td><code>jmp 8048446 &lt;switch_eg+0x36&gt;</code></td>
<td>8048430</td>
</tr>
<tr>
<td>0x804843b</td>
<td><code>mov 0x10(%ebp), %eax</code></td>
<td>8048435</td>
</tr>
<tr>
<td>0x8048439</td>
<td><code>imul 0xc(%ebp), %eax</code></td>
<td>8048439</td>
</tr>
<tr>
<td>0x804844b</td>
<td><code>jmp 8048453 &lt;switch_eg+0x43&gt;</code></td>
<td>8048443</td>
</tr>
<tr>
<td>0x804843b</td>
<td><code>mov 0xc(%ebp), %edx</code></td>
<td>804843b</td>
</tr>
<tr>
<td>0x804843e</td>
<td><code>mov %edx, %eax</code></td>
<td>804843e</td>
</tr>
<tr>
<td>0x8048440</td>
<td><code>sarl $0x1f, %edx</code></td>
<td>8048440</td>
</tr>
<tr>
<td>0x8048443</td>
<td><code>idivl 0x10(%ebp)</code></td>
<td>8048443</td>
</tr>
<tr>
<td>0x8048446</td>
<td><code>add 0x10(%ebp), %eax</code></td>
<td>8048446</td>
</tr>
<tr>
<td>0x8048449</td>
<td><code>jmp 8048453 &lt;switch_eg+0x43&gt;</code></td>
<td>8048449</td>
</tr>
<tr>
<td>0x804844b</td>
<td><code>mov $0x1, %eax</code></td>
<td>804844b</td>
</tr>
<tr>
<td>0x8048450</td>
<td><code>sub 0x10(%ebp), %eax</code></td>
<td>8048450</td>
</tr>
<tr>
<td>0x8048453</td>
<td><code>pop %ebp</code></td>
<td>8048453</td>
</tr>
<tr>
<td>0x8048454</td>
<td><code>ret</code></td>
<td>8048454</td>
</tr>
</tbody>
</table>
IA32 Stack

• Region of memory managed with stack discipline
• Register %esp contains address of “top” element

Stack Pointer: %esp

Stack Grows Down

increasing Addresses

Bottom

Top
IA32 Stack: Push

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

Stack Pointer: `%esp`
**IA32 Stack: Pop**

- `popl dst`
  - Write operand at address given by `%esp` to `dst`
  - Increment `%esp` by 4

**Stack Pointer:** `%esp` +4

**Bottom**

Stack Grows Down

Increasing Addresses
Procedure Control Flow

- Use stack to support procedure call and return
  - call label_of_procedure
    - Push return address on stack
    - Jump to label
  - ret
    - Pop address from stack
    - Jump to address

- Return address:
  - Address of the next instruction right after call
  - Example:
    804854e: e8 3d 06 00 00 call 8048b90 <main>
    8048553: 50 pushl %eax
  - Return address = 0x8048553
Procedure Call Example

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

call 8048b90

0x108 0x10c 0x110
0x108 0x10c 0x110
0x108 123 0x104 0x8048553
%esp 0x108 %esp 0x104
%eip 0x804854e %eip 0x8048b90

%eip: program counter
Procedure Return Example

8048591: c3
ret

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x110</td>
<td>0x10c</td>
<td>0x108</td>
<td>0x104</td>
<td></td>
</tr>
<tr>
<td>123</td>
<td>0x8048553</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td>0x104</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%eip</td>
<td>0x8048591</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x110</td>
<td>0x10c</td>
<td>0x108</td>
</tr>
<tr>
<td></td>
<td></td>
<td>123</td>
<td>0x8048553</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>%esp</td>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>%eip</td>
<td>0x8048553</td>
<td></td>
</tr>
</tbody>
</table>

%eip: program counter
Using stack for procedure invocation

- In addition to return address, stack also stores:
  - Arguments
  - Local variables
  - Scratch space
- Stack allocated in Frames:
  - State for single procedure instantiation
- Stack is well suited for procedure invocation:
  - State for given procedure needed during procedure execution (from call to return).
  - Callee returns before caller does (last-in-first-out)
Call Chain Example

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

![Diagram of stack frames with frame pointer and stack pointer](image-url)
Example

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

Stack

- %ebp
- %esp

Yoo
```c
yoo(...) {
    who(...) {
        ...
        amI();
        ...
        amI();
        ...
    }
}
```

**Example**

```
%ebp
%esp
```
Example

```c
yoo(...) {
  who(...) {
    amI(...) {
      do_something();
      ...
    }
  }
}
```

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

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

Stack

- yoo
- who

%ebp → %esp → Stack
Example

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

```
Stack

yoo

who

amI

%ebp

%esp
```
Example

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

Stack

```
%ebp

yoo

%esp

who
```
Example

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

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

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

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

Calling swap from call_swap

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

Resulting Stack

```
Rtn adr
&course1
&course2

%esp
subl

%esp
%esp
```
Revisiting swap

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

```
swap:
    pushl %ebp
    movl %esp, %ebp
    pushl %ebx
    movl 8(%ebp), %edx
    movl 12(%ebp), %ecx
    movl (%edx), %ebx
    movl (%ecx), %eax
    movl %eax, (%edx)
    movl %ebx, (%ecx)
    popl %ebx
    popl %ebp
    ret
```
swap Setup #1

swap:

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

Entering Stack

- `%ebp` ←
- `%esp` ←

&course2
&course1
Rtn adr

Resulting Stack

- `%ebp` ←

YP
xp
Rtn adr
Old %ebp

%esp ←
swap Setup #2

Entering Stack

\[
\begin{align*}
\text{\&course2} \\
\text{\&course1} \\
\text{Rtn adr}
\end{align*}
\]

%esp

Resulting Stack

\[
\begin{align*}
\text{YP} \\
\text{xp} \\
\text{Rtn adr} \\
\text{Old \%ebp}
\end{align*}
\]

%ebp %esp

swap:

\[
\begin{align*}
\text{pushl \%ebp} \\
\text{movl \%esp,\%ebp} \\
\text{pushl \%ebx}
\end{align*}
\]
swap Setup #3

Entering Stack

\[ \text{\&course2} \]
\[ \text{\&course1} \]
\[ \text{Rtn adr} \]

\[ \text{\%ebp} \leftarrow \text{\%esp} \]

\[ \text{\%ebp} \]

\[ \text{\%esp} \]

Resulting Stack

\[ \text{YP} \]
\[ \text{xp} \]
\[ \text{Rtn adr} \]
\[ \text{Old \%ebp} \]
\[ \text{Old \%ebx} \]

\[ \text{\%ebp} \leftarrow \text{\%esp} \]

\[ \text{\%esp} \]

\[ \text{swap:} \]
\[ \text{pushl \%ebp} \]
\[ \text{movl \%esp,\%ebp} \]
\[ \text{pushl \%ebx} \]
swap Body

Entering Stack

Resulting Stack

Offset relative to %ebp

&course2
&course1
Rtn adr

%ebp

12
8
4

%esp

yp
xp
Rtn adr
Old %ebp
Old %ebx

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

...
swap Finish

Stack Before Finish

- yp
- xp
- Rtn adr
- Old %ebp
- Old %ebx

Saved and restored register %ebx, but not %eax, %ecx, %edx

Resulting Stack

- yp
- xp
- Rtn adr

%ebp
%esp
popl %ebx
popl %ebp
Register Saving Conventions

- When procedure yoo calls who:
  - yoo is the *caller*
  - who is the *callee*

- Callee might use registers for temporary storage?

<table>
<thead>
<tr>
<th>yoo:</th>
<th>who:</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>movl $15213, %edx</td>
<td>movl 8(%ebp), %edx</td>
</tr>
<tr>
<td>call who</td>
<td>addl $18243, %edx</td>
</tr>
<tr>
<td>addl %edx, %eax</td>
<td>⚖️</td>
</tr>
<tr>
<td>ret</td>
<td>ret</td>
</tr>
</tbody>
</table>

- Contents of register %edx overwritten by who
- This could be trouble ➔ something should be done!
  - Need some coordination
Register Saving Conventions

• Register saving Conventions
  – “Caller Save”
    • Caller saves temporary values in its stack frame before the call
  – “Callee Save”
    • Callee saves temporary values in its stack 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
Recursive Function

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
lea (edx,%eax),%eax
.L3:
addl $4,%esp
popl %ebx
popl %ebp
ret

/* 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

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

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

```
pcount_r:
pushl %ebp
movl %esp,%ebp
pushl %ebx
subl $4,%esp
movl 8(%ebp),%ebx
```
Recursive Call #2

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

• Actions
  – If x == 0 (%eax),
    return

%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);
}

• Actions
  – Store x >> 1 on stack
  – Make recursive call
• Effect after calling pcount_r
  – %eax set to function result
  – %ebx still has value of x

movl %ebx,%eax
shrl %eax
movl %eax,(%esp)
call pcount_r
Recursive Call #4

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

• After calling `pcount_r`
  – `%eax` holds value from recursive call
  – `%ebx` holds `x`
• Actions
  – Compute `(x & 1) + computed value`
• Effect
  – `%eax` set to function result

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

/* Recursive popcount */

```c
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return (x & 1) + pcount_r(x >> 1);
}
```

**Actions**
- Restore %ebx, %ebp
- Restore %esp

L3:
```assembly
addl $4, %esp
popl %eax
popl %ebp
ret
```
/** Compute x + 3 */
int add3(int x) {
    int localx = x;
    incrk(&localx, 3);
    return localx;
}

/** 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

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

- Variable localx must be stored on stack
- Compute pointer as -4(%ebp)

First part of add3

```assembly
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

- Use leal instruction to compute address of localx

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

Middle part of add3

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

```
```
Retrieving local variable

- Retrieve localx from stack as return value

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

Final part of add3

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

• **Important Points**
  – Stack is the data structure for procedure invocation
  • If P calls Q, then Q returns before P

• **Recursion handled by normal calling conventions**

• **Pointers are addresses of values**
  – On stack or elsewhere (global, heap)