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)

- General purpose registers
  - %eax
  - %ecx
  - %edx
  - %ebx
  - %esi
  - %edi

- Current stack top
  - %esp

- Current stack frame
  - %ebp

- Instruction pointer
  - %eip

- Condition codes
  - 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.,
  
  \[
  \begin{align*}
  &\text{cmpl } \%\text{eax,}\%\text{edx} \\
  &\text{setl } \%\text{al} \\
  &\text{movzbl } \%\text{al,}\%\text{eax}
  \end{align*}
  \]

<table>
<thead>
<tr>
<th>SetX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>setne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>sets</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>setns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg</td>
<td>~(SF^OF) &amp;~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setge</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>setle</td>
<td>(SF^OF)</td>
<td>ZF</td>
</tr>
<tr>
<td>seta</td>
<td>~CF&amp;~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>setb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
Reading Condition Codes (Cont.)

\[ \text{addl/addq } \text{Src, Dest} \leftrightarrow t = a + b \]

- **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
  
  \[(a > 0 \land b > 0 \land t < 0) \lor (a < 0 \land b < 0 \land t \geq 0)\]

<table>
<thead>
<tr>
<th>OF</th>
<th>SF</th>
<th>SF^OF</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>No overflow, sign bit is correct.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>No overflow, sign bit is correct.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Overflow, sign bit is reversed.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Overflow, sign bit is reversed.</td>
</tr>
</tbody>
</table>
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

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

Body

```asm
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
```
Reading Condition Codes: x86-64

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

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

```c
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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp Label</td>
<td>1</td>
<td>Unconditional (Direct)</td>
</tr>
<tr>
<td>jmp *Operand</td>
<td>1</td>
<td>Unconditional (Indirect)</td>
</tr>
<tr>
<td>jne Label</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>js Label</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns Label</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg Label</td>
<td>~ (SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge Label</td>
<td>~ (SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl Label</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle Label</td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>ja Label</td>
<td>~CF &amp; ~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb Label</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>

jmp .L1
jmp *%eax
Conditional Branch Example

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

```
absdiff:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %edx
    movl 12(%ebp), %eax
    cmpl %eax, %edx
    jle .L6
    subl %eax, %edx
    jmp .L7
.L6:
    subl %edx, %eax
.L7:
    popl %ebp
    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;
}

- C allows “goto” as means of transferring control
  - Closer to machine-level programming style

- Generally considered bad coding style

absdiff:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %edx
    movl 12(%ebp), %eax
    cmpl %eax, %edx
    jle .L6
    subl %eax, %edx
    movl %edx, %eax
    jmp .L7
.L6:
    subl %edx, %eax
.L7:
    popl %ebp
    ret

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

Body1
Body2a
Body2b
Finish
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;
}
```c
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
    movl  %esp, %ebp
    movl  8(%ebp), %edx
    movl  12(%ebp), %eax
    cmpl  %eax, %edx
    jle   .L6
    subl  %eax, %edx
    movl  %edx, %eax
    jmp   .L7
.L6:
    subl  %edx, %eax
.L7:
    popl  %ebp
    ret
```
General Conditional Expression Translation

C Code

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

\[
\text{val} = x > y \ ? \ x - y : y - x;
\]

Goto Version

\[
\text{nt} = \!\! \text{Test};
\text{if} (\text{nt}) \ \text{goto} \ \text{Else};
\text{val} = \text{Then}\_\text{Expr};
\text{goto} \ \text{Done};
\text{Else}:
\phantom{nt = !\! \text{Test};}
\phantom{if (nt) goto Else;}
\phantom{val = Then\_Expr;}
\phantom{goto Done;}
\text{val} = \text{Else}\_\text{Expr};
\text{Done}:
\phantom{nt = !\! \text{Test};}
\phantom{if (nt) goto Else;}
\phantom{val = Then\_Expr;}
\phantom{goto Done;}
\phantom{Else:}
\phantom{val = Else\_Expr;}
\phantom{Done:}
\phantom{...}
\]

- 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:
  ```markdown
  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**
```c
val = Test ? Then_Expr : Else_Expr;
```

**Goto Version**
```c
result = Else_Expr;
t = Test;
if (t) result = tval;
return result;
```
Conditional Move Example: x86-64

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

absdiff:

- `x` in `%edi`
- `y` in `%esi`

```
movl %edi, %edx  # tval = x-y
subl %esi, %edx
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

\[
\text{val} = \text{Test}(x) \ ? \ \text{Hard1}(x) : \ \text{Hard2}(x);
\]

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

Risky Computations

\[
\text{val} = p \ ? \ *p : 0;
\]

- Both values get computed
- May have undesirable effects

Computations with side effects

\[
\text{val} = x > 0 \ ? \ x*=7 : x+=3;
\]

- 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

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

Goto Version

```c
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

```c
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
        # loop:
.L2:
    movl    %edx, %eax # t = x & 1
    andl    $1, %eax   # result += t
    addl    %eax, %ecx # x >>= 1
    shrl    %edx
    jne     .L2        # If !0, goto loop
```
General “Do-While” Translation

C Code
```c
do
    Body
while (Test);
```

- **Body:**
  ```c
  { 
  Statement\textsubscript{1};
  Statement\textsubscript{2};
  ...
  Statement\textsubscript{n};
  }
  ```

Goto Version
```c
loop:
    Body
    if (Test)
    goto loop
```

- **Test returns integer**
  - = 0 interpreted as false
  - ≠ 0 interpreted as true
“While” Loop Example

**C Code**

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

**Goto Version**

```c
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

```c
#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

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

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

Init
- `i = 0`

Test
- `i < WSIZE`

Update
- `i++`

Body
```c
{
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
}
```
"For" Loop $\rightarrow$ While Loop

For Version

\[
\text{for (Init; Test; Update )}
\]

\[
\text{Body}
\]

While Version

\[
\text{Init ;}
\]

\[
\text{while (Test )} \\ \\
\text{Body}
\]

\[
\text{Update ;}
\]
“For” Loop $\rightarrow$ ... $\rightarrow$ Goto

For Version

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

While Version

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

```
Init;
    if (!Test)
        goto done;
loop:
    Body
    Update
    if (Test)
        goto loop;
done:
```
"For" Loop Conversion Example

C Code

```c
#include 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

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

- Control: Condition codes
- Conditional branches
- Loops
- Switch statements
- IA 32 Procedures
  - Stack Structure
  - Calling Conventions
  - Illustrations of Recursion & Pointers
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
    case val_1:
        Block 1
        ...
    case val_n-1:
        Block n-1
}
```

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

```
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;
}

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]

What range of values takes default?

Note that w not initialized here
Switch Statement Example (IA32)

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

Setup:

```assembly
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]
```

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
```
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 $0 \leq x \leq 6$

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

.switch .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;
}

Handling Fall-Through

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

```c
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
```
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

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

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
```

```
.L3:
    movq  %rdx, %rax
    imulq  %rsi, %rax
    ret
```
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   0x0804844b
```
### Deciphering Jump Table

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8048660</td>
<td>0x08048422</td>
<td>0</td>
</tr>
<tr>
<td>0x8048664</td>
<td>0x08048432</td>
<td>1</td>
</tr>
<tr>
<td>0x8048668</td>
<td>0x0804843b</td>
<td>2</td>
</tr>
<tr>
<td>0x804866c</td>
<td>0x08048429</td>
<td>3</td>
</tr>
<tr>
<td>0x8048670</td>
<td>0x08048422</td>
<td>4</td>
</tr>
<tr>
<td>0x8048674</td>
<td>0x0804844b</td>
<td>5</td>
</tr>
<tr>
<td>0x8048678</td>
<td>0x0804844b</td>
<td>6</td>
</tr>
</tbody>
</table>
**Disassembled Targets**

<table>
<thead>
<tr>
<th>Address</th>
<th>Machine Code</th>
<th>Assembly Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048422</td>
<td>b8 02 00 00 00</td>
<td>mov $0x2,%eax</td>
<td></td>
</tr>
<tr>
<td>8048427</td>
<td>eb 2a</td>
<td>jmp 8048453 &lt;switch_eg+0x43&gt;</td>
<td></td>
</tr>
<tr>
<td>8048429</td>
<td>b8 01 00 00 00</td>
<td>mov $0x1,%eax</td>
<td></td>
</tr>
<tr>
<td>804842e</td>
<td>66 90</td>
<td>xchg %ax,%ax # noop</td>
<td></td>
</tr>
<tr>
<td>8048430</td>
<td>eb 14</td>
<td>jmp 8048446 &lt;switch_eg+0x36&gt;</td>
<td></td>
</tr>
<tr>
<td>8048432</td>
<td>8b 45 10</td>
<td>mov 0x10(%ebp),%eax</td>
<td></td>
</tr>
<tr>
<td>8048435</td>
<td>0f af 45 0c</td>
<td>imul 0xc(%ebp),%eax</td>
<td></td>
</tr>
<tr>
<td>8048439</td>
<td>eb 18</td>
<td>jmp 8048453 &lt;switch_eg+0x43&gt;</td>
<td></td>
</tr>
<tr>
<td>804843b</td>
<td>8b 55 0c</td>
<td>mov 0xc(%ebp),%edx</td>
<td></td>
</tr>
<tr>
<td>804843e</td>
<td>89 d0</td>
<td>mov %edx,%eax</td>
<td></td>
</tr>
<tr>
<td>8048440</td>
<td>c1 fa 1f</td>
<td>sar $0x1f,%edx</td>
<td></td>
</tr>
<tr>
<td>8048443</td>
<td>f7 7d 10</td>
<td>idivl 0x10(%ebp)</td>
<td></td>
</tr>
<tr>
<td>8048446</td>
<td>03 45 10</td>
<td>add 0x10(%ebp),%eax</td>
<td></td>
</tr>
<tr>
<td>8048449</td>
<td>eb 08</td>
<td>jmp 8048453 &lt;switch_eg+0x43&gt;</td>
<td></td>
</tr>
<tr>
<td>804844b</td>
<td>b8 01 00 00 00</td>
<td>mov $0x1,%eax</td>
<td></td>
</tr>
<tr>
<td>8048450</td>
<td>2b 45 10</td>
<td>sub 0x10(%ebp),%eax</td>
<td></td>
</tr>
<tr>
<td>8048453</td>
<td>5d</td>
<td>pop %ebp</td>
<td></td>
</tr>
<tr>
<td>8048454</td>
<td>c3</td>
<td>ret</td>
<td></td>
</tr>
</tbody>
</table>
Matching Disassembled Targets

<table>
<thead>
<tr>
<th>Value</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8048422</td>
<td>mov $0x2,%eax</td>
</tr>
<tr>
<td>0x8048427</td>
<td>jmp 8048453 &lt;switch_eg+0x43&gt;</td>
</tr>
<tr>
<td>0x8048429</td>
<td>mov $0x1,%eax</td>
</tr>
<tr>
<td>0x804842e</td>
<td>xchg %ax,%ax</td>
</tr>
<tr>
<td>0x8048430</td>
<td>jmp 804846 &lt;switch_eg+0x36&gt;</td>
</tr>
<tr>
<td>0x8048432</td>
<td>mov 0x10(%ebp),%eax</td>
</tr>
<tr>
<td>0x8048435</td>
<td>imul 0xc(%ebp),%eax</td>
</tr>
<tr>
<td>0x8048439</td>
<td>jmp 8048453 &lt;switch_eg+0x43&gt;</td>
</tr>
<tr>
<td>0x804843b</td>
<td>mov 0xc(%ebp),%edx</td>
</tr>
<tr>
<td>0x804843e</td>
<td>mov %edx,%eax</td>
</tr>
<tr>
<td>0x8048440</td>
<td>sar $0x1f,%edx</td>
</tr>
<tr>
<td>0x8048443</td>
<td>idivl 0x10(%ebp)</td>
</tr>
<tr>
<td>0x8048446</td>
<td>add 0x10(%ebp),%eax</td>
</tr>
<tr>
<td>0x8048449</td>
<td>jmp 8048453 &lt;switch_eg+0x43&gt;</td>
</tr>
<tr>
<td>0x804844b</td>
<td>mov $0x1,%eax</td>
</tr>
<tr>
<td>0x8048450</td>
<td>sub 0x10(%ebp),%eax</td>
</tr>
<tr>
<td>0x8048453</td>
<td>pop %ebp</td>
</tr>
<tr>
<td>0x8048454</td>
<td>ret</td>
</tr>
</tbody>
</table>

Value

<table>
<thead>
<tr>
<th>0x8048422</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8048432</td>
</tr>
<tr>
<td>0x804843b</td>
</tr>
<tr>
<td>0x8048429</td>
</tr>
<tr>
<td>0x8048422</td>
</tr>
<tr>
<td>0x804844b</td>
</tr>
<tr>
<td>0x804844b</td>
</tr>
</tbody>
</table>
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

Stack Pointer: `%esp`

Stack “Top”

Stack “Bottom”

Increasing Addresses

Stack Grows Down
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**

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Call Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>804854e:</td>
<td>e8 3d 06 00 00 call 8048b90 &lt;main&gt;</td>
</tr>
<tr>
<td>8048553:</td>
<td>50 pushl %eax</td>
</tr>
</tbody>
</table>

```
804854e:
call 8048b90
8048553:
pushl %eax
```

- **%esp**:
  - 0x108 (initial)
  - Pointer to stored data

- **%eip**:
  - 0x804854e (initial)
  - Program counter

- **0x110**: Value stored by call

- **0x10c**: Value stored by call

- **0x108**: Value stored by call

- **0x104**: Value stored by call

- **0x8048553**: Address of pushl instruction
Procedure Return Example

8048591:  c3  ret

%esp: 0x104
%eip: 0x8048591
%esp: 0x104
%eip: 0x8048591
ret
0x108 123
0x10c 0x8048553
0x110 0x8048553
0x110 0x8048553
0x10c 123
%esp: 0x104
%esp: 0x108
%eip: 0x8048591
%eip: 0x8048553
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

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

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

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

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

Stack

```
yoo

%ebp
%esp
```

```
yoo
```

```
who
```

```
amI
```

```
amI
```

```
amI
```

```
amI
```
Example

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

Stack

- `%ebp`
- `%esp`
Example

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

Stack

%ebp
%esp

yoo
who
amI
```
Example

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

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

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

who() {
  • amI();
  • amI();
}
```

Stack

```
%ebp
%
%esp
```
Example

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

who(); 
amI(); 
```

Stack

```
<table>
<thead>
<tr>
<th>yoo</th>
</tr>
</thead>
<tbody>
<tr>
<td>who</td>
</tr>
<tr>
<td>amI</td>
</tr>
<tr>
<td>%ebp</td>
</tr>
<tr>
<td>%esp</td>
</tr>
</tbody>
</table>
```
Example

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

Stack

```
%ebp
%
estp
```
Example

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

Stack

```
hoo

%ebp

%esp

who
```
Example

```c
void who(...) {
    if (amI(...)) {
        ...%ebp%esp%
    } else {
        ...%ebp%esp%
    }
}
```
Example

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

Stack

```
yoo

%ebp
%esp

who
```

Diagram:

- yoo
- who
- amI

Stack:
- yoo
- who

Arrow directions:
- %ebp
- %esp
Example

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

Stack

```
%ebp
%esp
```

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

<table>
<thead>
<tr>
<th>Caller Frame</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Return Addr</td>
</tr>
<tr>
<td></td>
<td>Old %ebp</td>
</tr>
<tr>
<td>Frame pointer</td>
<td>%ebp</td>
</tr>
<tr>
<td>Saved Registers + Local Variables</td>
<td></td>
</tr>
<tr>
<td>Argument Build</td>
<td></td>
</tr>
</tbody>
</table>

Stack pointer
%esp
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
%esp
%esp
```

subl call
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
```

Set Up

Body

Finish
swap Setup #1

Entering Stack

\[
\begin{align*}
\text{\%ebp} & \quad \text{\%esp} \\
& \text{\&course2} \\
& \text{\&course1} \\
& \text{Rtn adr}
\end{align*}
\]

Resulting Stack

\[
\begin{align*}
\text{\%ebp} & \quad \text{\%esp} \\
& \text{yp} \\
& \text{xp} \\
& \text{Rtn adr} \\
& \text{Old \%ebp}
\end{align*}
\]

swap:

\[
\begin{align*}
\text{pushl \%ebp} \\
\text{movl \%esp,\%ebp} \\
\text{pushl \%ebx}
\end{align*}
\]
swap Setup #2

Entering Stack

\[ \text{\underline{\%ebp}} \]

\[ \text{\underline{\%esp}} \]

Resulting Stack

\[ \text{\underline{\%ebp}} \]

\[ \text{\underline{\%esp}} \]

\[ \text{\underline{\text{Rtn adr}}} \]

\[ \text{\underline{\text{\&course1}}} \]

\[ \text{\underline{\text{\&course2}}} \]

\[ \text{\underline{\text{Rtn adr}}} \]

\[ \text{yp} \]

\[ \text{xp} \]

\[ \text{\underline{\text{Old \%ebp}}} \]

\[ \text{\underline{\text{\%ebp}}} \]

\[ \text{\underline{\%ebp}} \]

\[ \text{\underline{\%esp}} \]

\[ \text{\underline{\%esp}} \]

\text{\underline{\%ebp}}

\[ \text{\underline{\%esp}} \]

\[ \text{\underline{\text{\&course1}}} \]

\[ \text{\underline{\text{\&course2}}} \]

\[ \text{\underline{\text{Rtn adr}}} \]

\[ \text{\underline{\text{old \%ebp}}} \]

\[ \text{\underline{\%ebp}} \]

\[ \text{\underline{\%esp}} \]

\text{\underline{\%ebp}}

\[ \text{\underline{\%esp}} \]

\[ \text{\underline{\text{\&course1}}} \]

\[ \text{\underline{\text{\&course2}}} \]

\[ \text{\underline{\text{Rtn adr}}} \]

\[ \text{\underline{\text{old \%ebp}}} \]

\[ \text{\underline{\%ebp}} \]

\[ \text{\underline{\%esp}} \]

\[ \text{\underline{\text{old \%ebp}}} \]

\[ \text{\underline{\%ebp}} \]

\[ \text{\underline{\%esp}} \]

\[ \text{\underline{\text{old \%ebp}}} \]

\[ \text{\underline{\%ebp}} \]

\[ \text{\underline{\%esp}} \]

\[ \text{\underline{\text{old \%ebp}}} \]

\[ \text{\underline{\%ebp}} \]

\[ \text{\underline{\%esp}} \]

\[ \text{\underline{\text{old \%ebp}}} \]

\[ \text{\underline{\%ebp}} \]

\[ \text{\underline{\%esp}} \]

\[ \text{\underline{\text{old \%ebp}}} \]

\[ \text{\underline{\%ebp}} \]

\[ \text{\underline{\%esp}} \]

\[ \text{\underline{\text{old \%ebp}}} \]
swap Setup #3

Entering Stack

•
•

&course2
&course1
Rtn adr

Resulting Stack

•
•

yp
xp
Rtn adr
Old %ebp
Old %ebx

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

Entering Stack

Resulting Stack

\[
\text{movl } 8(\%ebp),\%edx \quad \# \text{ get } xp \\
\text{movl } 12(\%ebp),\%ecx \quad \# \text{ get } yp \\
\ldots
\]
swap Finish

Stack Before Finish

Resulting Stack

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

08048384 <swap>:

```
PUSH %EBP
MOV %ESP,%EBP
PUSH %EBX
MOV 0x8(%EBP),%EDX
MOV 0xc(%EBP),%ECX
MOV (%EDX),%EBX
MOV (%ECX),%EAX
MOV %EAX,(%EDX)
MOV %EBX,(%ECX)
POP %EBX
POP %EBP
RET
```

Calling Code

```
MOVL $0x8049658,0x4(%ESP)  # Copy &course2
MOVL $0x8049654,(%ESP)   # Copy &course1
CALL 08048384 <swap>     # Call swap
LEAVE                     # Prepare to return
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 *you* calls *who*:
  - *you* is the *caller*
  - *who* is the *callee*

- Can register be used for temporary storage?

```
<table>
<thead>
<tr>
<th>you:</th>
<th>who:</th>
</tr>
</thead>
<tbody>
<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>• • •</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

- When procedure you calls who:
  - you 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 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);
}

■ 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
    • • •

%esp ➔ %ebp ➔ Old %ebx ➔ Old %ebp ➔ Rtn adr ➔ x ➔ %ebx
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}

Actions
- If x == 0, return
  - with %eax set to 0
/* 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
- %eax set to function result
- %ebx still has value of x

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

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

Rtn adr
Old %ebp
Old %ebx
x >> 1
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}

■ Assume
  ▪ %eax holds value from recursive call
  ▪ %ebx holds x

■ Actions
  ▪ Compute (x & 1) + computed value

■ Effect
  ▪ %eax set to function result
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}

### Actions
- Restore values of `%ebx` and `%ebp`
- Restore `%esp`

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

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

### Referencing Pointer

```c
/* 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

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

- Use leal instruction to compute address of localx

Middle part of `add3`

```assembly
movl $3, 4(%esp)     # 2\text{nd} arg = 3
leal -4(%ebp), %eax  # &localx
movl %eax, (%esp)    # 1\text{st} arg = &localx
call incrk
```
Retrieving local variable

```c
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