Introduction to Computer Systems
15-213/18-243, spring 2009
5th Lecture, Sep. 8th

Instructors:
Roger B. Dannenberg and Greg Ganger

Last Time: Machine Programming, Basics
- History of Intel processors and architectures
- C, assembly, machine code
- Assembly (IA32):
  - Registers
  - Operands
  - Move (what's the l in movl?)

```plaintext
movl $0x4,%eax
movl %eax,%edx
movl (%eax),%edx
```

Complete Memory Addressing Modes
- Most General Form
  \[ D(Rb,Ri,S) \rightarrow \text{Mem}[\text{Reg}[Rb]+S\times\text{Reg}[Ri]+D] \]
  - D: Constant "displacement" 1, 2, or 4 bytes
  - Rb: Base register: Any of 8 integer registers
  - Ri: Index register: Any, except for %esp
  - S: Scale: 1, 2, 4, or 8 (why these numbers?)

Special Cases
- (Rb,Ri) \rightarrow \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]]
- D(Rb,Ri) \rightarrow \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]+D]
- (Rb,Ri,S) \rightarrow \text{Mem}[\text{Reg}[Rb]+S\times\text{Reg}[Ri]]

Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx 0x8</td>
<td>0x8000 + 0x8</td>
<td>0x008</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x100</td>
<td></td>
</tr>
<tr>
<td>0x8(%edx)</td>
<td>0x8000 + 0x8</td>
<td>0x008</td>
</tr>
<tr>
<td>(%edx,%ecx)</td>
<td>0x8000 + 0x100</td>
<td>0x100</td>
</tr>
<tr>
<td>(%edx,%ecx,4)</td>
<td>0x8000 + 4*0x100</td>
<td>0x400</td>
</tr>
<tr>
<td>0x80(,%edx,2)</td>
<td>2*0x8000 + 0x80</td>
<td>0x1a080</td>
</tr>
</tbody>
</table>

will disappear blackboard?
Address Computation Instruction

- **`leal`** `Src, Dest`
  - `Src` is address mode expression
  - Set `Dest` to address denoted by expression

**Uses**
- Computing addresses without a memory reference
  - E.g., translation of `p = &x[1];`
- Computing arithmetic expressions of the form `x + k*y`
  - `k = 1, 2, 4, or 8`

**Example**

---

**Some Arithmetic Operations**

- **Two Operand Instructions:**
  - **Format**
  - **Computation**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addl</code> <code>Src, Dest</code></td>
<td><code>Dest = Dest + Src</code></td>
</tr>
<tr>
<td><code>subl</code> <code>Src, Dest</code></td>
<td><code>Dest = Dest - Src</code></td>
</tr>
<tr>
<td><code>imull</code> <code>Src, Dest</code></td>
<td><code>Dest = Dest * Src</code></td>
</tr>
<tr>
<td><code>sal</code> <code>Src, Dest</code></td>
<td><code>Dest = Dest &lt;&lt; Src</code> Also called <code>shl</code></td>
</tr>
<tr>
<td><code>shr</code> <code>Src, Dest</code></td>
<td><code>Dest = Dest &gt;&gt; Src</code> Logical</td>
</tr>
<tr>
<td><code>andl</code> <code>Src, Dest</code></td>
<td><code>Dest = Dest &amp; Src</code></td>
</tr>
<tr>
<td><code>orl</code> <code>Src, Dest</code></td>
<td>`Dest = Dest</td>
</tr>
</tbody>
</table>

**One Operand Instructions**

- **Format**
- **Computation**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>incl</code> <code>Dest</code></td>
<td><code>Dest = Dest + 1</code></td>
</tr>
<tr>
<td><code>decl</code> <code>Dest</code></td>
<td><code>Dest = Dest - 1</code></td>
</tr>
<tr>
<td><code>negl</code> <code>Dest</code></td>
<td><code>Dest = -Dest</code></td>
</tr>
<tr>
<td><code>notl</code> <code>Dest</code></td>
<td><code>Dest = -Dest</code></td>
</tr>
</tbody>
</table>

**Some Arithmetic Operations**

**Today**

- Complete addressing mode, address computation (`leal`)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches
- While loops

---

**Using `leal` for Arithmetic Expressions**

```c
int arith (int x, int y, int z) {
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

---

**Understanding `arith`**

```c
int arith (int x, int y, int z) {
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

```
<table>
<thead>
<tr>
<th>Offset</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>x</td>
</tr>
<tr>
<td>12</td>
<td>y</td>
</tr>
<tr>
<td>8</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td>Old %ebp</td>
</tr>
<tr>
<td>0</td>
<td>%ebp</td>
</tr>
</tbody>
</table>
```

will disappear blackboard?
Another Example

```c
int logical(int x, int y) {
    int t1 = x^y;
    int t2 = t1 >> 17;
    int t3 = t2 + t1;
    int t4 = t3 + t2;
    int t5 = t4 + t3;
    int t6 = t5 + t4;
    int t7 = t6 + t5;
    int t8 = t7 + t6;
    int t9 = t8 + t7;
    int t10 = t9 + t8;
    int t11 = t10 + t9;
    int t12 = t11 + t10;
    int t13 = t12 + t11;
    int t14 = t13 + t12;
    int t15 = t14 + t13;
    int t16 = t15 + t14;
    int t17 = t16 + t15;
    int t18 = t17 + t16;
    return t18;
}
```

Another Example

```c
int logical(int x, int y) {
    int t1 = x^y;
    int t2 = t1 >> 17;
    int t3 = t2 + t1;
    int t4 = t3 + t2;
    int t5 = t4 + t3;
    int t6 = t5 + t4;
    int t7 = t6 + t5;
    int t8 = t7 + t6;
    int t9 = t8 + t7;
    int t10 = t9 + t8;
    int t11 = t10 + t9;
    int t12 = t11 + t10;
    int t13 = t12 + t11;
    int t14 = t13 + t12;
    int t15 = t14 + t13;
    int t16 = t15 + t14;
    int t17 = t16 + t15;
    int t18 = t17 + t16;
    return t18;
}
```
Another Example

```c
int logical(int x, int y)
{
    int t1 = x ^ y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

Example:
```
movl 8(%ebp),%eax
movl 12(%ebp),%eax
sarl $17,%eax
andl $8185,%eax
movl %ebp,%esp
popl %ebp
ret
```

Today

- Complete addressing mode, address computation (lea)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches
- While loops

Data Representations: IA32 + x86-64

- Sizes of C Objects (in Bytes)
  - C Data Type: Typical 32-bit, Intel IA32, x86-64
    ```
    | C Data Type | Typical 32-bit | Intel IA32 | x86-64 |
    |-------------|----------------|------------|--------|
    | unsigned    | 4              | 4          | 4      |
    | int         | 4              | 4          | 4      |
    | long int    | 4              | 4          | 8      |
    | char        | 1              | 1          | 1      |
    | short       | 2              | 2          | 2      |
    | float       | 4              | 4          | 4      |
    | double      | 8              | 8          | 8      |
    | long double | 8              | 10/12      | 16     |
    | char *      | 4              | 4          | 8      |
    | Or any other pointer | |
    ```

x86-64 Integer Registers

- ```
  x86-64:
<table>
<thead>
<tr>
<th>x86</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>rax</td>
<td>%rax</td>
</tr>
<tr>
<td>rbx</td>
<td>%rbx</td>
</tr>
<tr>
<td>rcx</td>
<td>%rcx</td>
</tr>
<tr>
<td>rdx</td>
<td>%rdx</td>
</tr>
<tr>
<td>rsi</td>
<td>%rsi</td>
</tr>
<tr>
<td>rdi</td>
<td>%rdi</td>
</tr>
<tr>
<td>r8</td>
<td>%r8</td>
</tr>
<tr>
<td>r9</td>
<td>%r9</td>
</tr>
</tbody>
</table>
  ```

Instructions

- Long word l (4 Bytes) ↔ Quad word q (8 Bytes)
  - New instructions:
    ```
    movl → movq
    addl → addq
    sall → salq
    etc.
    ```

- 32-bit instructions that generate 32-bit results
  ```
  Set higher order bits of destination register to 0
  Example: addl
  ```
Swap in 32-bit Mode

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

**Setup**
- pushl %ebp
- movl %esp,%ebp
- pushl %ebx

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

**Finish**
- movl -4(%ebp),%ebx
- movl %ebp,%esp
- popl %ebp
- ret

Swap in 64-bit Mode

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

**Setup**
- pushl %ebp
- movl %esp,%ebp
- pushl %ebx

**Body**
- movl (%rdi), %edx
- movl (%rsi), %eax
- movl %eax, (%rdi)
- movl %edx, (%rsi)
- retq

Swap Long Ints in 64-bit Mode

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

**Setup**
- movq (%rdi), %rdx
- movq (%rsi), %rax

**Body**
- movq %rax, (%rdi)
- movq %rdx, (%rsi)
- retq

Today
- Complete addressing mode, address computation (lea)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches
- While loops

Processor State (IA32, Partial)

- Information about currently executing program
  - Temporary data (%eax,...)
  - Location of runtime stack (%ebp, %esp)
  - Location of current code control point (%ebp,...)
  - Status of recent tests (%cf, %zf, %sf, %of)

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

Current stack top
- %esp

Current stack frame
- %ebp

Instruction pointer

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/adcq Src, Dest = t = arw
    - 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 also on course website
Condition Codes (Explicit Setting: Compare)

- **Explicit Setting by Compare Instruction**
  - `cmpl` or `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` or `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 to 1 or 0 based on combinations of condition codes

<table>
<thead>
<tr>
<th>SetX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete</td>
<td>Equal</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</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setle</td>
<td>~SF^OF</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl</td>
<td>~CF^OF</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>setle</td>
<td>~CF^OF</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>setb</td>
<td>CF</td>
<td>Below (unsigned)</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**

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

  **Note**: Inverted ordering!

Reading Condition Codes: x86-64

- **SetX Instructions**: Set single byte based on combination of condition codes
- **One of 8 addressable byte registers**
  - Does not alter remaining 3 bytes

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

  **Body (same for both)**

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

  **Will disappear Blackboard?**
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;
}

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

**Body (same for both)**

```c
xorl %eax, %eax
# eax = 0

cmpq %rsi, %rdi
# Compare x and y

setg %al
# al = x > y
```

- Is %rax zero?
  - Yes: 32-bit instructions set high order 32 bits to 0!

Jumping

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

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

Conditional Branch Example

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

```c
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;
}
```

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

Conditional Branch Example (Cont.)

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

Conditional Branch Example (Cont.)

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

Today

- Complete addressing mode, address computation (leal)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches
- While loops
### Conditional Branches (Cont.)

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

```assembly
absdiff:
    pushl %ebp
    movl %esp, %ebp
    movl %eax, %esi
    movl %edx, %edi
    cmpl %esi, %edi
    jle .L7
    subl %esi, %edi
    movl %esi, %edi
    .L8:
    leave
    ret
    .L7:
    subl %edi, %esi
    jmp .L8
```

### Conditional Branches (Cont.)

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

```assembly
absdiff:
    pushl %ebp
    movl %esp, %ebp
    movl %eax, %esi
    movl %edx, %edi
    cmpl %esi, %edi
    jle .L7
    subl %esi, %edi
    movl %esi, %edi
    .L8:
    leave
    ret
    .L7:
    subl %edi, %esi
    jmp .L8
```

### General Conditional Expression Translation

**C Code**

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

**Goto Version**

```c
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

### Conditionals: x86-64

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

```assembly
absdiff:
    # x in %edi, y in %esi
    movl %edi, %esi
    movl %esi, %edi
    subl %esi, %edi
    cmpl %esi, %edi
    cmovl %esi, %edi
    ret
```

- Conditional move instruction
  - cmovC src, dest
  - Move value from src to dest if condition C holds
  - More efficient than conditional branching (simple control flow)
  - But overhead: both branches are evaluated
General Form with Conditional Move

C Code

val1 = Test ? Then-Expr : Else-Expr;
val2 = Else-Expr;
val1 = val2 if !Test;

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 too expensive

“Do-While” Loop Example

C Code

int fact_do(int x) { 
  int result = 1;
  do { 
    result *= x;
    x = x-1;
  } while (x > 1);
  return result;
}

Goto Version

int fact_goto(int x) { 
  int result = 1;
  loop: 
    result *= x;
    x = x-1;
    if (x > 1) 
      goto loop;
  return result;
}

- Use backward branch to continue looping
- Only take branch when “while” condition holds

“Do-While” Loop Compilation

Goto Version

int fact_goto(int x) { 
  int result = 1;
  loop: 
    result *= x;
    x = x-1;
    if (x > 1) 
      goto loop;
  return result;
}

Assembly

pushl %ebp  # Setup
movl %esp,%ebp # Setup
movl $1,%eax  # eax = 1
movl %ebp,%edx # edx = x
.L11: imull %edx,%eax  
  # result *= x
  x = x-1;
  deccl %edx  
  x--;
  cmpl $1,%edx  
  # Compare x = 1
  jg .L11  # if > goto loop
movl %ebp,%esp  # Finish
popl %ebp  # Finish
ret

Will disappear Blackboard?

General “Do-While” Translation

C Code

do Body 
  while (Test); 

Goto Version

Loop: 
  Body 
  if (Test) goto loop

- Body: 
  Statement;
  Statement;
  ... 
  Statement;

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

C Code
```c
int fact_while(int x) {
    int result = 1;
    while (x > 1) {
        result *= x;
        x = x-1;
    }
    return result;
}
```

Goto Version #1
```c
int fact_while_goto(int x) {
    int result = 1;
    loop:
    if (!(x > 1))
        goto done;
    result *= x;
    x = x-1;
    goto loop;
    done:
    return result;
}
```

- Is this code equivalent to the do-while version?
- Must jump out of loop if test fails

Alternative “While” Loop Translation

C Code
```c
int fact_while(int x) {
    int result = 1;
    while (x > 1) {
        result *= x;
        x = x-1;
    }
    return result;
}
```

Goto Version #2
```c
int fact_while_goto2(int x) {
    int result = 1;
    if (!(x > 1))
        goto done;
    loop:
    result *= x;
    x = x-1;
    if (x > 1)
        goto loop;
    done:
    return result;
}
```

- Historically used by GCC
- Uses same inner loop as do-while version
- Guards loop entry with extra test

General “While” Translation

While version
```
while (Test)
    Body
```

Do-While Version
```
if (!Test)
    goto done;
Body
while (Test);
done:
```

Goto Version
```
if (!Test)
    goto loop;
Body
if (Test)
    goto loop;
done:
```

New Style “While” Loop Translation

C Code
```c
int fact_while(int x) {
    int result = 1;
    while (x > 1) {
        result *= x;
        x = x-1;
    }
    return result;
}
```

Goto Version
```c
int fact_while_goto3(int x) {
    int result = 1;
    goto middle;
    loop:
    result *= x;
    x = x-1;
    middle:
    if (x > 1)
        goto loop;
    return result;
}
```

- Recent technique for GCC
  - Both IA32 & x86-64
  - First iteration jumps over body computation within loop

Jump-to-Middle While Translation

C Code
```
while (Test)
    Body
```

Goto Version
```
goto middle;
loop:
    Body
middle:
    if (Test)
        goto loop;
done:
```

- Avoids duplicating test code
- Unconditional goto incurs no performance penalty
- For loops compiled in similar fashion

Jump-to-Middle Example
```
int fact_while(int x) {
    int result = 1;
    while (x > 1) {
        result *= x;
        x--;
    }
    return result;
}
```

```
# x in %edx, result in %eax
jmp .L34       # goto Middle
.L35:          # Loop:
imul %edx, %eax # result *= x
dec %edx         # x--
.L34:          # Middle:
cmp $1, %edx # x:1
jg .L35        # if >, goto Loop
```

- # x in %edx, result in %eax
- Jump to Middle
- Loop:
- Imul %edx, %eax # result *= x
- Dec %edx # x--
- Middle:
- Cmpl $1, %edx # x:1
- Jg .L35 # if >, goto Loop
Implementing Loops

- **IA32**
  - All loops translated into form based on "do-while"

- **x86-64**
  - Also make use of "jump to middle"

**Why the difference**
- IA32 compiler developed for machine where all operations costly
- x86-64 compiler developed for machine where unconditional branches incur (almost) no overhead

Next Time...

- for Loops
- switch Statements
- Procedures