Lecture 5: More machine operations

Computer Architecture and Systems Programming
(252-0061-00)

Timothy Roscoe
Herbstsemester 2012
Last time:
Machine programming, basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly (IA32):
  - Registers
  - Operands
  - Move (what’s the 1 in movl?)
    
    \[
    \begin{align*}
    \text{movl } & \$0x4,\%eax \\
    \text{movl } & \%eax,\%edx \\
    \text{movl } & (\%eax),\%edx
    \end{align*}
    \]
Today

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

• Most General Form:

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

- D: Constant “displacement” 1, 2, or 4 bytes
- Rb: Base register: Any of 8 integer registers
- Ri: Index register: Any, except for \%esp
  - Unlikely you’d use \%ebp, either
- S: Scale: 1, 2, 4, or 8 (why these numbers?)

• Special Cases

\begin{align*}
\text{(Rb,Ri)} & \quad \text{Mem[Reg[Rb]+Reg[Ri]]} \\
\text{D(Rb,Ri)} & \quad \text{Mem[Reg[Rb]+Reg[Ri]+D]} \\
\text{(Rb,Ri,S)} & \quad \text{Mem[Reg[Rb]+S*Reg[Ri]]}
\end{align*}
Address computation examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0x8(%edx)$</td>
<td>$0xf000 + 0x8$</td>
<td>$0xf008$</td>
</tr>
<tr>
<td>$(%edx, %ecx)$</td>
<td>$0xf000 + 0x100$</td>
<td>$0xf100$</td>
</tr>
<tr>
<td>$(%edx, %ecx, 4)$</td>
<td>$0xf000 + 4*0x100$</td>
<td>$0xf400$</td>
</tr>
<tr>
<td>$0x80(, %edx, 2)$</td>
<td>$2*0xf000 + 0x80$</td>
<td>$0x1e080$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$%edx$</th>
<th>$0xf000$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$%ecx$</td>
<td>$0x100$</td>
</tr>
</tbody>
</table>
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[i]`;
  - Computing arithmetic expressions of the form `x + k*y`
    *k = 1, 2, 4, or 8*
Today

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

• Two operand instructions:

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>addl</td>
<td>Dest ← Dest + Src</td>
</tr>
<tr>
<td>subl</td>
<td>Dest ← Dest - Src</td>
</tr>
<tr>
<td>imull</td>
<td>Dest ← Dest * Src</td>
</tr>
<tr>
<td>sall</td>
<td>Dest ← Dest &lt;&lt; Src</td>
</tr>
<tr>
<td>sarl</td>
<td>Dest ← Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>shr1</td>
<td>Dest ← Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>xorl</td>
<td>Dest ← Dest ^ Src</td>
</tr>
<tr>
<td>andl</td>
<td>Dest ← Dest &amp; Src</td>
</tr>
<tr>
<td>orl</td>
<td>Dest ← Dest</td>
</tr>
</tbody>
</table>

Also called shll
Arithmetic
Logical

• No distinction between signed and unsigned int (why?)
Some arithmetic operations

• One operand instructions

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>incl</td>
<td>Dest ← Dest + 1</td>
</tr>
<tr>
<td>decl</td>
<td>Dest ← Dest - 1</td>
</tr>
<tr>
<td>negl</td>
<td>Dest ← -Dest</td>
</tr>
<tr>
<td>notl</td>
<td>Dest ← ~Dest</td>
</tr>
</tbody>
</table>

• See book for more instructions
Using \texttt{leal} for arithmetic expressions

\begin{verbatim}
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;
}
\end{verbatim}

\begin{verbatim}
pushl %ebp
movl %esp,%ebp

movl 8(%ebp),%eax
movl 12(%ebp),%edx
leal (%edx,%eax),%ecx
leal (%edx,%edx,2),%edx
sall $4,%edx
addl 16(%ebp),%ecx
leal 4(%edx,%eax),%eax
imull %ecx,%eax

movl %ebp,%esp
popl %ebp
ret
\end{verbatim}
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;
}

movl 8(%ebp),%eax  # eax = x
movl 12(%ebp),%edx  # edx = y
leal (%edx,%eax),%ecx  # ecx = x+y (t1)
leal (%edx,%edx,2),%edx  # edx = 3*y
sall $4,%edx  # edx = 48*y (t4)
addl 16(%ebp),%ecx  # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax  # eax = 4+t4+x (t5)
imull %ecx,%eax  # eax = t5*t2 (rval)
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;
}
```

2^13 = 8192, 2^13 – 7 = 8185

---

```
movl 8(%ebp),%eax        # eax = x
xorl 12(%ebp),%eax       # eax = x^y (t1)
sarl $17,%eax            # eax = t1>>17 (t2)
andl  $8185,%eax         # eax = t2 & 8185
```

---

logical:
pushl %ebp
movl %esp,%ebp
movl 8(%ebp),%eax
xorl 12(%ebp),%eax
sarl $17,%eax
andl $8185,%eax
movl %ebp,%esp
popl %ebp
ret
Today

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

**ia32 and x86-64**

- Sizes of C objects (in bytes)

<table>
<thead>
<tr>
<th>C data type</th>
<th>Typical 32-bit</th>
<th>ia32</th>
<th>Intel x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>long long</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>8</td>
<td>10/12</td>
<td>10/16</td>
</tr>
<tr>
<td>char * (or any other pointer)</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>
### x86-64 Integer Registers

<table>
<thead>
<tr>
<th>x86-64 Register</th>
<th>x86 Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%eax</td>
</tr>
<tr>
<td>%rbx</td>
<td>%ebx</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
</tr>
<tr>
<td>%r8</td>
<td>%r8d</td>
</tr>
<tr>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%r15</td>
<td>%r15d</td>
</tr>
</tbody>
</table>

- Extend existing registers. Add 8 new ones.
- Make %ebp/%rbp general purpose
Instructions

• Long word \( l \) (4 Bytes) ↔ Quad word \( q \) (8 Bytes)

• New instructions:
  – movl \( \rightarrow \) movq
  – addl \( \rightarrow \) addq
  – sall \( \rightarrow \) 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;
}
```

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```
Swap in 64-bit Mode

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

• Operands passed in registers (why useful?)
  – First (xp) in %rdi, second (yp) in %rsi
  – 64-bit pointers
• No stack operations required
• 32-bit data
  – Data held in registers %eax and %edx
  – movl operation

swap:
    movl (%rdi), %edx
    movl (%rsi), %eax
    movl %eax, (%rdi)
    movl %edx, (%rsi)
    retq
Swap Long Ints in 64-bit Mode

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

swap_l:
   movq (%rdi), %rdx
   movq (%rsi), %rax
   movq %rax, (%rdi)
   movq %rdx, (%rsi)
   retq

• 64-bit data
  – Data held in registers %rax and %rdx
  – movq operation
  – “q” stands for quad-word
Today

• Complete addressing mode, address computation (leal)
• 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 (%eip, ...)
  - Status of recent tests (CF, ZF, SF, OF)

General purpose registers

Current stack top
Current stack frame
Instruction pointer
Condition codes
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: \texttt{addl/addq\ 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
    \[(a>0 \&\& b>0 \&\& t<0) \mid\mid (a<0 \&\& b<0 \&\& t>=0)\]

  - Not set by \texttt{lea} instruction
  - \textbf{Full documentation} (ia32), link also on course website
Condition Codes
(Explicit Setting: Compare)

• Explicit Setting by Compare Instruction

\texttt{cмр1/cмрq} \ Source2, Source1

\texttt{cмр1 b, a} like computing \( a - b \) without setting destination

\texttt{CF} set if carry out from most significant bit
\hspace{1cm} (used for unsigned comparisons)
\texttt{ZF} set if \( a == b \)
\texttt{SF} set if \( (a - b) < 0 \) (as signed)
\texttt{OF} set if two’s complement (signed) overflow
\hspace{1cm} \( (a>0 \ \&\& \ b<0 \ \&\& \ (a-b)<0) \)
\hspace{1cm} \| \ (a<0 \ \&\& \ b>0 \ \&\& \ (a-b)>0) \)
Condition Codes
(Explicit Setting: Test)

• Explicit Setting by Test instruction

\texttt{testl@testq\ Src2,Src1}

\texttt{testl b,a} like computing \texttt{a\&b} w/o setting destination

– Sets condition codes based on value of \texttt{Src1 \& Src2}
– Useful to have one of the operands be a mask

\texttt{ZF} set when \texttt{a\&b == 0}
\texttt{SF} set when \texttt{a\&b < 0}
Reading Condition Codes

• SetX Instructions
  – Set single byte 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>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>settl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>settle</td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</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.)

• `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;
}
```

```
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 lggt (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

- \textbf{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>1</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>Less or Equal (Signed)</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>
Today

- Complete addressing mode, address computation (lea1)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches
- While loops
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 .L7
    subl %eax, %edx
    movl %edx, %eax
    jmp .L8
    .L8:
    leave
    ret
    .L7:
    subl %edx, %eax
    jmp .L8

Body1:

Set up:

Finish:

Body2:
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

absdiff:

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

.L8:
leave
ret

.L7:
subl %edx, %eax
jmp .L8
General Conditional Expression Translation

C Code

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

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

- 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

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

```assembly
absdiff: # x in %edi, y in %esi
    movl %edi, %eax  # eax = x
    movl %esi, %edx  # edx = y
    subl %esi, %eax  # eax = x-y
    subl %edi, %edx  # edx = y-x
    cmpl %esi, %edi  # x:y
    cmovle %edx, %eax  # eax=edx if <=
    ret
```

```c
int absdiff(int x, int y) {
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```
General Form with Conditional Move

C Code

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

Conditional Move Version

```c
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 expressions have side effects
  - Then and else expressions are too expensive
Today

- Complete addressing mode, address computation (`lea`)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches
- While loops
“Do-While” Loop Example

C Code

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

Goto Version

```c
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

```c
int fact_goto(int x)
{
    int result = 1;

    loop:
        result *= x;
        x = x-1;
        if (x > 1)
            goto loop;

    return result;
}
```

Assembly

```assembly
fact_goto:
    pushl %ebp  # Setup
    movl %esp,%ebp  # Setup
    movl $1,%eax  # eax = 1
    movl 8(%ebp),%edx  # edx = x

.L11:
    imull %edx,%eax  # result *= x
    decl %edx  # x--
    cmpl $1,%edx  # Compare x : 1
    jg .L11  # if > goto loop

    movl %ebp,%esp  # Finish
    popl %ebp  # Finish
    ret  # Finish
```

Registers:

- `%edx` : `x`
- `%eax` : `result`

37
General “Do-While” Translation

C Code

```
do
    Body
while (Test);
```

• Body:  
  
  ```
  { 
  Statement_1;  
  Statement_2;  
  ...  
  Statement_n;  
  }
  ```

• Test returns integer
  = 0 interpreted as false
  ≠ 0 interpreted as true

Goto Version

```
loop:
    Body
    if (Test)
        goto loop
```
“While” Loop Example

C Code

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

Goto Version #1

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

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

Do-While Version

\[
\text{if} \ (!\text{Test}) \\
\quad \text{goto} \ \text{done}; \\
\quad \text{do} \\
\quad \quad \text{Body} \\
\quad \quad \text{while} \ (\text{Test}); \\
\text{done:} \\
\]

Goto Version

\[
\text{if} \ (!\text{Test}) \\
\quad \text{goto} \ \text{done}; \\
\text{loop:} \\
\quad \text{Body} \\
\quad \text{if} \ (\text{Test}) \\
\quad \quad \text{goto} \ \text{loop}; \\
\text{done:} \\
\]
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

```c
while (Test)
    Body
```

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

Goto Version

```c
goto middle;
loop:
    Body
middle:
    if (Test)
        goto loop;
```

Goto (Previous) Version

```c
if (!Test)
    goto done;
loop:
    Body
if (Test)
    goto loop;
done:
```
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:
imull %edx, %eax # result *= x
decl %edx  # x--
.L34:    # 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