This week

- Lab 1 due 11pm today
  - Frustrating? Awesome? Both? Neither?

- Lab 2 out later today, due 11pm Wednesday July 17
  - Disassembly, reverse engineering machine code

- For those fascinated by floating point:
  - Lecture 1pm Wednesday July 10 in CSE 305 by William Kahan, main designer of the IEEE floating point standard and Turing Award winner.

- HW 2 due 11pm Thursday July 11

32-bit vs. 64-bit operands

- Long word \( l \) (4 Bytes) \( \leftrightarrow \) Quad word \( q \) (8 Bytes)

- New instruction forms:
  - \texttt{movl} \( \rightarrow \) \texttt{movq}
  - \texttt{addl} \( \rightarrow \) \texttt{addq}
  - \texttt{sall} \( \rightarrow \) \texttt{salq}
  - etc.

- x86-64 can still use 32-bit instructions that generate 32-bit results
  - Higher-order bits of destination register are just set to 0
  - Example: \texttt{addl}

Swap Ints in 32-bit Mode

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

Swap Ints in 64-bit Mode

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

- Arguments passed in registers (why useful?)
  - First (xp) in \%rdi, second (yp) in \%rsi
  - 64-bit pointers

- No stack operations required: faster

- 32-bit data
  - Data held in registers \%eax and \%edx
  - \texttt{movl} operation (the 1 refers to data width, not address width)
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;
}
```

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

Complete Memory Addressing Modes

- Remember, the addresses used for accessing memory in `mov` (and other) instructions can be computed in several different ways

  - **Most General Form:**
    - `D(Rb,Ri,S)`
    - `Mem[Reg[Rb] + S*Reg[Ri] + D]`
    - **D:** Constant “displacement” value represented in 1, 2, or 4 bytes
    - **Rb:** Base register: Any of the 8/16 integer registers
    - **Ri:** Index register: Any, except for `%esp` or `%rsp`; `%ebp` unlikely
    - **S:** Scale: 1, 2, 4, or 8 (`why these numbers?`)

  - **Special Cases:** can use any combination of `D`, `Rb`, `Ri` and `S`

    - `(Rb,Ri)`
    - `Mem[Reg[Rb]+Reg[Ri]]`
    - `(S=1, D=0)`
    - `D(Rb,Ri)`
    - `Mem[Reg[Rb]+Reg[Ri]+D]`
    - `(S=1)`
    - `(Rb,Ri,S)`
    - `Mem[Reg[Rb]+S*Reg[Ri]]`
    - `(D=0)`
    - `...`

Address Computation Examples

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

Load Effective Address

- **lea Src, Dest**
- **leaq Src, Dest**
  - `Src` is address mode expression
  - Set `Dest` to address computed by expression
  - 32-bit Example: `lea (%edx,%ecx,4), %eax`
  - 64-bit Example: `leaq (%rdx,%rcx,4), %rax`

Uses

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

- Two Operand (Binary) 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>shll</td>
<td>Dest = Dest &lt;&lt; Src Also called sall</td>
</tr>
<tr>
<td>sarl</td>
<td>Dest = Dest &gt;&gt; Src Arithmetic</td>
</tr>
<tr>
<td>shrl</td>
<td>Dest = Dest &gt;&gt; Src Logical</td>
</tr>
<tr>
<td>xorl</td>
<td>Dest = Dest &amp; Src</td>
</tr>
<tr>
<td>orl</td>
<td>Dest = Dest</td>
</tr>
</tbody>
</table>

- Watch out for argument order! (especially subl)
- Few distinctions between signed and unsigned int (why?)
  - except sarl vs. shrl, see CS:APP 3.5.5 about extra case for imull

Using leal for Arithmetic Expressions (IA32)

```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 (IA32)

```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;
}
```

```c
movl 8(%ebp),%eax # eax = x
movl 12(%ebp),%edx # edx = y
leal (%edx,%eax),%ecx # ecx = x+y (t1)
leal (%edx,%eax),%ecx # edx = y + 2*y = 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)
```
Understanding arith (IA32)

```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;
}
```

### Stack

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

### Code

```assembly
movl 8(%ebp),%eax  # eax = x
movl 12(%ebp),%edx  # edx = y
leal (%edx,%eax),%ecx  # ecx = x+y (t1)
leal (%edx,%eax,2),%edx  # edx = y + 2*y = 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)
```

### Observations about arith

- Instructions in different order from C code
- Some expressions require multiple instructions
- Some instructions cover multiple expressions
- Get exact same code when compile:

```
(x+y+z) * (x+4+48*y)
```
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;
}

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;
}

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

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:             { Set Up }
pushl %ebp
movl %esp,%ebp

Logical:             { Body }
movl 8(%ebp),%eax    movl 8(%ebp),%eax
xorl 12(%ebp),%eax   xorl 12(%ebp),%eax
sarl $17,%eax        sarl $17,%eax
andl $8185,%eax      andl $8185,%eax

Logical:             { Finish }
movl %ebp,%esp
popl %ebp
ret
Topics: control flow

- Condition codes
- Conditional and unconditional branches
- Loops

Conditionals and Control Flow

- A conditional branch is sufficient to implement most control flow constructs offered in higher level languages
  - if (condition) then {...} else {...}
  - while (condition) {...}
  - do {...} while (condition)
  - for (initialization; condition; iterative) {...}

- Unconditional branches implement some related control flow constructs
  - break, continue

- In x86, we’ll refer to branches as “jumps” (either conditional or unconditional)

Jumping

- jX Instructions
  - Jump to different part of code depending on condition codes
  - Takes address as argument

<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>~ZF</td>
</tr>
<tr>
<td>ja</td>
<td><del>CF</del>ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th>General purpose registers</th>
<th>Current stack top</th>
<th>Current stack frame</th>
<th>Instruction pointer</th>
<th>Condition codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td></td>
<td></td>
<td>%esp</td>
<td>%eip</td>
</tr>
<tr>
<td>%ecx</td>
<td></td>
<td></td>
<td>%ebp</td>
<td></td>
</tr>
<tr>
<td>%edx</td>
<td></td>
<td></td>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%ebx</td>
<td></td>
<td></td>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
<td></td>
<td>%ebp</td>
<td>%eip</td>
</tr>
<tr>
<td>%ebp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%eip</td>
<td></td>
<td></td>
<td>CF, ZF, SF, OF</td>
<td></td>
</tr>
</tbody>
</table>
### Condition Codes (Implicit Setting)

- **Single-bit registers**
  - CF Carry Flag (for unsigned)
  - ZF Zero Flag
  - SF Sign Flag (for signed)
  - OF Overflow Flag (for signed)

- **Implicitly set (think of it as side effect) by arithmetic operations**
  - **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
    \[(a>0 \&\& b>0 \&\& t<0) \mid (a<0 \&\& b<0 \&\& t>=0)\]

- **Not set by lea instruction (beware!)**

- **Full documentation (IA32):** http://www.jegerlehner.ch/intel/IntelCodeTable.pdf

### Condition Codes (Explicit Setting: Compare)

- **Single-bit registers**
  - CF Carry Flag (for unsigned)
  - SF Sign Flag (for signed)
  - ZF Zero Flag
  - OF Overflow Flag (for signed)

- **Explicit Setting by Compare Instruction**
  - `cmpl/cmpeq 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
      \[(a>0 \&\& b<0 \&\& (a-b)<0) \mid (a<0 \&\& b>0 \&\& (a-b)>0)\]

### Condition Codes (Explicit Setting: Test)

- **Single-bit registers**
  - CF Carry Flag (for unsigned)
  - ZF Zero Flag
  - SF Sign Flag (for signed)
  - OF Overflow Flag (for signed)

- **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** if a\&b == 0
  - **SF set** if a\&b < 0

- **testl %eax, %eax**
  - Sets SF and ZF, check if eax is +,0,-

### Reading Condition Codes

- **SetX Instructions**
  - Set a single byte to 0 or 1 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>setsn</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg</td>
<td>~ (SF&amp;OF)&amp;~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setge</td>
<td>~ (SF&amp;OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl</td>
<td>(SF&amp;OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>setle</td>
<td>(SF&amp;OF)&amp;ZF</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 to 0 or 1 based on combination of condition codes
- **One of 8 addressable byte registers**
  - Does not alter remaining 3 bytes
  - Typically use `movzbl` to finish job

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

**Body: y at 12(%ebp), x at 8(%ebp)**

```
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
```

Reading Condition Codes (Cont.)

- **SetX Instructions:**
  - Set single byte to 0 or 1 based on combination of condition codes
- **One of 8 addressable byte registers**
  - Does not alter remaining 3 bytes
  - Typically use `movzbl` to finish job

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

**Body: y at 12(%ebp), x at 8(%ebp)**

```
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
```

**Conditional Branch Example**

```
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 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
.
.L8:
    leave
    ret
    .L7:
    subl %eax, %edx
    movl %edx, %eax
    jmp .L8
```

Jumping

- **jX Instructions**
  - Jump to different part of code depending on condition codes
  - Takes address as argument

<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><del>(SF</del>OF)</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge</td>
<td><del>(SF</del>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><del>CF</del>ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>

```
Conditional Branch Example (Cont.)

```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.)

```
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
```

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

```
int x       %edx
int y       %eax
```

```
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
```

```
int x       %edx
int y       %eax
```

```
http://xkcd.com/292/
```
### Conditional Branch Example (Cont.)

```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;
}
```

```assembly
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
```

### Conditional Branch Example (Cont.)

```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;
}
```

```assembly
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
```

### Conditional Branch Example (Cont.)

```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;
}
```

```assembly
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;
```

**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
- How might you make this more efficient?
Conditionals: x86-64

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

- Conditional move instruction
  - `cmovC` src, dest
  - Move value from src to dest if condition C holds
  - *Why is this good?*

PC Relative Addressing

```
0x100  cmp  r2, r3  0x1000
0x102  je   0x70   0x1002
0x104  ...  0x1004
...    ...  ...
0x172  add  r3, r4  0x1072
```

- PC relative branches are **relocatable**
- Absolute branches are not

Compiling Loops

**C/Java code:**
```
while ( sum != 0 ) {
    <loop body>
}
```

**Machine code:**
```
loopTop:  cmpl $0, %eax
         je   loopDone
         <loop body code>
         jmp   loopTop
loopDone:
```

- *How to compile other loops should be straightforward*
  - The only slightly tricky part is to be sure where the conditional branch occurs: top or bottom of the loop
- *How would for(i=0; i<100; i++) be implemented?*
“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

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

General “Do-While” Translation

C Code

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

Goto Version

```c
do
    Body
if (Test) goto loop
```

- **Body**: 
  - `Statement;`
  - `Statement;`
  - `...`
  - `Statement;`
- **Test** returns integer
  - 0 interpreted as false
  - # interpreted as true
“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_goto(int x) {
    int result = 1;
    while (x > 1) {
        result *= x;
        x = x-1;
    } goto middle;
    if (x > 1) 
        goto loop;
    return result;
}
```

- Used by GCC for both IA32 & x86-64
- First iteration jumps over body computation within loop straight to test

“While” Loop Example

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

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

“For” Loop Example: Square-and-Multiply

```c
/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned int p) {
    int result;
    for (result = 1; p != 0; p = p>>1) {
        if (p & 0x1)
            result *= x;
        x = x*x;
    }
    return result;
}
```

Algorithm
- Exploit bit representation: \( p = p_0 + 2p_1 + 2^2p_2 + \ldots + 2^{n-1}p_{n-1} \)
- Gives: \( x^p = z_0 \cdot z_1^2 \cdot (z_2^2)^2 \cdot \ldots \cdot (z_{n-1}^2)^2 \)
- Complexity \( O(\log p) \)

Example
\[
3^{10} = 3^8 \cdot 3^2 = 3^2 \cdot (3^2)^2^2
\]

ipwr Computation

```c
/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned int p) {
    int result;
    for (result = 1; p != 0; p = p>>1) {
        if (p & 0x1)
            result *= x;
        x = x*x;
    }
    return result;
}
```

<table>
<thead>
<tr>
<th>before iteration</th>
<th>result</th>
<th>x=3</th>
<th>p=10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>10=1010</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>9</td>
<td>5=101</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>81</td>
<td>2=10</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>6561</td>
<td>1=1</td>
</tr>
<tr>
<td>5</td>
<td>59049</td>
<td>43046721</td>
<td>0=</td>
</tr>
</tbody>
</table>
"For" Loop Example

```c
int result;
for (result = 1; p ! = 0; p = p >> 1)
{
    if (p & 0x1)
        result *= x;
    x = x * x;
}
```

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

<table>
<thead>
<tr>
<th>Init</th>
<th>Test</th>
<th>Update</th>
<th>Body</th>
</tr>
</thead>
<tbody>
<tr>
<td>result = 1</td>
<td>p ! = 0</td>
<td>p = p &gt;&gt; 1</td>
<td></td>
</tr>
</tbody>
</table>

"For"→"While"

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

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

While Version
```
Init;
while (Test) {
  Body
  Update;
}
```

Quick Review

- Complete memory addressing mode
  - (%eax), 17(%eax), 2(%ebx, %ecx, 8), ...

- Arithmetic operations that do set condition codes
  - subl %eax, %ecx         # ecx = eax + edx
  - sall $4,%edx           # edx = edx << 4
  - addl 16(%ebp),%ecx     # ecx = eax + Mem[16+ebp]
  - imull %ecx,%eax        # eax = eax * ecx

- Arithmetic operations that do NOT set condition codes
  - leal 4(%edx,%eax),%eax  # eax = 4 + edx + eax
Quick Review

- **x86-64 vs. IA32**
  - Integer registers: 16 x 64-bit vs. 8 x 32-bit
  - `movq, addq, ...` vs. `movl, addl, ...`
  - `movq` -> “move quad word” or 4*16-bits
  - x86-64: better support for passing function arguments in registers

- **Control**
  - Condition code registers
  - Set as side effect or by `cmp`, `test`
  - Used:
    - Read out by `setx` instructions (`setg, setle, ...`)
    - Or by conditional jumps (`jle .L4, je .L10, ...`)
    - Or by conditional moves (`cmovle %edx, %eax`)

---

Quick Review

- **Do-While loop**
  - C Code
    ```
    do
    Body
    while (!Test);
    ```
  - Goto Version
    ```
    loop:
    Body
    if (!Test) goto loop
    ```

- **While-Do loop**
  - While version
    ```
    while (Test)
    Body
    ```
  - Do-While Version
    ```
    do
    Body
    while (Test);
    ```
  - Goto Version
    ```
    if (!Test) goto done;
    loop:
    Body
    if (!Test) goto loop;
    done:
    goto middle;
    ```

---

Summarizing

- **C Control**
  - if-then-else
  - do-while
  - while, for
  - switch

- **Assembler Control**
  - Conditional jump
  - Conditional move
  - Indirect jump
  - Compiler
  - Must generate assembly code
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 (see text)

- **Conditions in CISC**
  - CISC machines generally have condition code registers