Machine-Level Programming II: Arithmetic & Control

15-213 / 18-213: Introduction to Computer Systems
6th Lecture, Sep. 13, 2012

Instructors:
David O’Hallaron, Greg Ganger, and Greg Kesden
Today

- Complete addressing mode, address computation (leal)
- Arithmetic operations
- Control: Condition codes
- Conditional branches
- While loops
Complete Memory Addressing Modes

- **Most General Form**
  - **D(Rb,Ri,S)**  
    - **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**
  - **(Rb,Ri)**  
    - Mem[Reg[Rb]+Reg[Ri]]
  - **D(Rb,Ri)**  
    - Mem[Reg[Rb]+Reg[Ri]+D]
  - **(Rb,Ri,S)**  
    - Mem[Reg[Rb]+S*Reg[Ri]]
## Address Computation Examples

<table>
<thead>
<tr>
<th>Address</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>0xf000</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x0100</td>
</tr>
</tbody>
</table>

### Expression | Address Computation | Address
---|---|---
0x8(%edx) | | |
(%edx,%ecx) | | |
(%edx,%ecx,4) | | |
0x80(,%edx,2) | | |
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`

- **Example**

```c
int mul12(int x)
{
    return x*12;
}
```

Converted to ASM by compiler:

```
leal (%eax,%eax,2), %eax ; t <- x+x*2
sall $2, %eax ; return t<<2
```
Today

- Complete addressing mode, address computation (leal)
- Arithmetic operations
- 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>shrl</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>

Watch out for argument order!

No distinction between signed and unsigned int (why?)
Some Arithmetic Operations

- **One Operand Instructions**

  - `incl` Dest Dest = Dest + 1
  - `decl` Dest Dest = Dest - 1
  - `negl` Dest Dest = - Dest
  - `notl` Dest Dest = ~Dest

- **See book for more instructions**
Arithmetic Expression Example

```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;
}
```

```assembly
arith:
    pushl  %ebp
    movl   %esp, %ebp

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

    popl   %ebp
    ret
```

Set Up

Body

Finish
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;
}
```

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

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

- Instructions in different order from C code
- Some expressions require multiple instructions
- Some instructions cover multiple expressions
- Get exact same code when compile:
  - \((x+y+z)\times(x+4+48y)\)

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

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

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

logical:
- **Set Up**:
  - `pushl %ebp`
  - `movl %esp,%ebp`

- **Body**:
  - `movl 12(%ebp),%eax`
  - `xorl 8(%ebp),%eax`
  - `sarl $17,%eax`
  - `andl $8185,%eax`

- **Finish**:
  - `popl %ebp`
  - `ret`

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

**logical:**
```
pushl %ebp
movl %esp,%ebp

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

popl %ebp
ret
```
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;
}
```

**logical:**

- **Set Up**
  - `pushl %ebp`
  - `movl %esp,%ebp`

- **Body**
  - `movl 12(%ebp),%eax`
  - `xorl 8(%ebp),%eax`  
  - `sarl $17,%eax`
  - `andl $8185,%eax`

- **Finish**
  - `popl %ebp`
  - `ret`

\[2^{13} = 8192, 2^{13} - 7 = 8185\]
Today

- Complete addressing mode, address computation (leal)
- Arithmetic operations
- Control: Condition codes
- Conditional branches
- 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)

- %eax
- %ecx
- %edx
- %ebx
- %esi
- %edi
- %esp
- %ebp

- %eip

- General purpose registers
  - Current stack top
  - Current stack frame

- 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: 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 leave instruction

- Full documentation (IA32), link on course website
Condition Codes (Explicit Setting: Compare)

- **Explicit Setting by Compare Instruction**
  - `cmp1/cmpq` Src2, Src1
  - `cmp1 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 low-order byte to 0 or 1 based on combinations of condition codes
  - Does not alter remaining 3 bytes

<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>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;
}
```

**Body**

```
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

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

```
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

- Complete addressing mode, address computation (leal)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches & Moves
- Loops
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>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>
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.)

```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;
}
```

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

- Generally considered bad coding style

```asm
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
```

Setup
Body1
Body2a
Body2b
Finish
Conditional Branch Example (Cont.)

```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;
}
```

```assembly
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
```

- **Setup**
- **Body1**
- **Body2a**
- **Body2b**
- **Finish**
Conditional Branch Example (Cont.)

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

Conditional Branch Example (Cont.)

<table>
<thead>
<tr>
<th>Body1</th>
<th>Body2a</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup</td>
<td>Finish</td>
</tr>
</tbody>
</table>
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;
goto Done;
Else:
   val = Else_Expr;
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
Using Conditional Moves

### Conditional Move Instructions

- Instruction supports:
  
  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
tval = Then_Expr;
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
    subl %esi, %edx  # tval = x-y
    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

- Complete addressing mode, address computation (lea)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches and moves
- Loops
“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;
}
```

```assembly
movl  $0, %ecx  # result = 0
.L2:
    movl  %edx, %eax  # loop:
    andl $1, %eax  # t = x & 1
    addl %eax, %ecx  # result += t
    shrl %edx  # x >>= 1
    jne .L2  # If !0, goto loop
```

- Registers:
  - %edx: x
  - %ecx: result
General “Do-While” Translation

**C Code**

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

**Goto Version**

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

- **Body:**
  ```c
  {  
  Statement_1;  
  Statement_2;  
  ...  
  Statement_n;  
  }
  ```

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

C Code

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

Goto Version

```c
int pcound_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

for (Init; Test; Update)

Body

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

Init

i = 0

Test

i < WSIZE

Update

i++

Body

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

For Version

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

Body
```

While Version

```c
Init;

while (Test ) {

Body

Update;

}
```
"For" Loop $\rightarrow$ ... $\rightarrow$ Goto

For Version

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

Body
```

While Version

```c
Init;
while (Test ) {

Body

Update;
}
```

```c
Init;
if (! Test)
    goto done;
loop:
    Body
    Update
    if (Test)
        goto loop;
done:
```
“For” Loop Conversion 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;
}
```

- 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;
}
```
Summary

■ Today
  ▪ Complete addressing mode, address computation (leal)
  ▪ Arithmetic operations
  ▪ Control: Condition codes
  ▪ Conditional branches & conditional moves
  ▪ Loops

■ Next Time
  ▪ Switch statements
  ▪ Stack
  ▪ Call / return
  ▪ Procedure call discipline