3.5 Conditional Codes

Spring, 2014
Euiseong Seo
(euiseong@skku.edu)
Today

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

- Most General Form
  - \( D(R_b, R_i, S) \)
    - \( \text{Mem}[ \text{Reg}[R_b] + S \times \text{Reg}[R_i] + D ] \)
  - \( D \) (constant): displacement 1, 2, or 4 bytes
  - \( R_b \) (base register): Any of 8 integer registers
  - \( R_i \) (index register): Any, except for \( \%\text{esp} \)
    - Unlikely you’d use \( \%\text{ebp} \), either
  - \( S \) (scale): 1, 2, 4, or 8

- Special Cases
  - \( (R_b, R_i) \Rightarrow \text{Mem}[ \text{Reg}[R_b] + \text{Reg}[R_i] ] \)
  - \( D(R_b, R_i) \Rightarrow \text{Mem}[ \text{Reg}[R_b] + \text{Reg}[R_i] + D ] \)
  - \( (R_b, R_i, S) \Rightarrow \text{Mem}[ \text{Reg}[R_b] + S \times \text{Reg}[R_i] ] \)
# 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`

**Example**

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

**Converted to ASM by compiler:**

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

- Complete addressing mode, address computation (lea)
- Arithmetic operations
- Control: Condition codes
- Conditional branches
- While loops
<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addl</code> Src, Dest</td>
<td>Dest = Dest + Src</td>
</tr>
<tr>
<td><code>subl</code> Src, Dest</td>
<td>Dest = Dest - Src</td>
</tr>
<tr>
<td><code>imull</code> Src, Dest</td>
<td>Dest = Dest * Src</td>
</tr>
<tr>
<td><code>sall</code> Src, Dest</td>
<td>Dest = Dest &lt;&lt;&lt; Src</td>
</tr>
<tr>
<td><code>sarl</code> Src, Dest</td>
<td>Dest = Dest &gt;&gt;&gt; Src</td>
</tr>
<tr>
<td><code>xorl</code> Src, Dest</td>
<td>Dest = Dest ^ Src</td>
</tr>
<tr>
<td><code>andl</code> Src, Dest</td>
<td>Dest = Dest &amp; Src</td>
</tr>
<tr>
<td><code>orl</code> Src, Dest</td>
<td>Dest = Dest</td>
</tr>
<tr>
<td><code>incl</code> Src, Dest</td>
<td>Dest = Dest + 1</td>
</tr>
<tr>
<td><code>decl</code> Src, Dest</td>
<td>Dest = Dest - 1</td>
</tr>
<tr>
<td><code>negl</code> Src, Dest</td>
<td>Dest = - Dest</td>
</tr>
<tr>
<td><code>notl</code> Src, Dest</td>
<td>Dest = Dest ~ Src</td>
</tr>
</tbody>
</table>
Some Arithmetic Operations

- One Operand Instructions

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

- See book for more instructions
Using leal for Arithmetic Expressions

```
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;
}
```

```
arith:
    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
```
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;
}
```

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

Stack

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

`Sungkyunkwan University`
### 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;
}
```

### Assembly Code

```assembly
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)
```
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;
}

call logical

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

movl %ebp,%esp

popl %ebp

ret
Complete addressing mode, address computation (leal)
Arithmetic operations
Control: Condition codes
Conditional branches
While loops
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)
### 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
  - `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**
**Condition Codes: **

- **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: 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

<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;
}
```

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source Registers</th>
<th>Destination Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl</td>
<td>12(%ebp), %eax</td>
<td>%eax = y</td>
</tr>
<tr>
<td>cmpl</td>
<td>%eax, 8(%ebp)</td>
<td># Compare x : y</td>
</tr>
<tr>
<td>setg</td>
<td>%al</td>
<td># al = x &gt; y</td>
</tr>
<tr>
<td>movzbl</td>
<td>%al, %eax</td>
<td># Zero rest of %eax</td>
</tr>
</tbody>
</table>

Note inverted ordering!
SetX Instructions:
- Set single byte based on combination of condition codes
- Does not alter remaining 3 bytes

ASM is the same for both (gt and lgt)

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

```c
long lgt (long x, long y)
{
    return 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>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>~CF &amp; ~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
### Encodings for Jumps

- Jump targets are written using symbolic labels
  - PC-relative
    - Difference between address of target instruction and address of the instruction immediately following jump
    - This offset can be encoded using one, two or four bytes
  - Absolute
    - Four bytes to directly specify the target
Example

```
.jle .L4
.p2align 4,,7
.L5:
    movl    %edx, %eax
    sarl    $1, %eax
    subl    %eax, %edx
    testl   %edx, %edx
    jg      .L5
.L4:
    movl    %edx, %eax
```

<table>
<thead>
<tr>
<th>8:</th>
<th>7e 11</th>
<th>jle</th>
<th>1b &lt;silly+0x1b&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>a:</td>
<td>8d b6 00 00 00 00</td>
<td>lea</td>
<td>0x0(%esi),%esi</td>
</tr>
<tr>
<td>10:</td>
<td>89 d0</td>
<td>mov</td>
<td>%edx,%eax</td>
</tr>
<tr>
<td>12:</td>
<td>c1 f8 01</td>
<td>sar</td>
<td>$0x1,%eax</td>
</tr>
<tr>
<td>15:</td>
<td>29 c2</td>
<td>sub</td>
<td>%eax, %edx</td>
</tr>
<tr>
<td>17:</td>
<td>85 d2</td>
<td>test</td>
<td>%edx,%edx</td>
</tr>
<tr>
<td>19:</td>
<td>7f f5</td>
<td>jg</td>
<td>10 &lt;silly+0x10&gt;</td>
</tr>
<tr>
<td>1b:</td>
<td>89 d0</td>
<td>mov</td>
<td>%edx,%eax</td>
</tr>
</tbody>
</table>
A. What is the target of the jbe instruction below?

8048d1c: 76 da jbe XXXXXX
8048dle: eb 24 jmp 8048d44

SOL>

value in PC is 0x848dle.
0xda + 0x8048dle