Lab4

• Strategy?
• lea: how does adding work again?
• What’s going on w/ data and control transfer?
• Look at a jump table example
Simple leal example

```c
int t1 = x+y;
t1: %ecx
x: %edx
y: %eax

leal (%edx,%eax),%ecx
```
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: `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
      \[(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)\]

- *Not* set by `lea` instruction
Condition Codes (Explicit Setting: Compare)

• Explicit Setting by Compare Instruction
  \texttt{cmpl/cmpq \ Src2,Src1}
  \texttt{cmpl b,a} like computing \texttt{a-b} without setting destination

  – \textbf{CF set} if carry out from most significant bit (used for unsigned comparisons)
  – \textbf{ZF set} if \texttt{a == b}
  – \textbf{SF set} if \texttt{(a-b) < 0} (as signed)
  – \textbf{OF set} if two’s complement (signed) overflow
    \texttt{(a>0 && b<0 && (a-b)<0) || (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} without setting destination

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

  – ZF set when \texttt{a\&b} == 0
  – 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>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**

```assembly
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
```

<table>
<thead>
<tr>
<th>%eax</th>
<th>%ah</th>
<th>%al</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>%ch</td>
<td>%cl</td>
</tr>
<tr>
<td>%edx</td>
<td>%dh</td>
<td>%dl</td>
</tr>
<tr>
<td>%ebx</td>
<td>%bh</td>
<td>%bl</td>
</tr>
<tr>
<td>%esi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td></td>
<td></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!
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
    jmp .L8
.L7:
    subl %edx, %eax
.L8:
    leave
    ret
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;
}

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

• C allows “goto” as means of transferring control
  – Closer to machine-level programming style
• Generally considered bad coding style
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 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;
}

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;
}
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
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 expression have side effects
  - Then and else expression are to expensive