Some slides adapted (and slightly modified) from:
• Clark Barrett
• Jinyang Li
• Randy Bryant
• Dave O’Hallaron
<table>
<thead>
<tr>
<th>Type</th>
<th>Form</th>
<th>Operand value</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>$Imm$</td>
<td>$Imm$</td>
<td>Immediate</td>
</tr>
<tr>
<td>Register</td>
<td>$E_a$</td>
<td>$R[E_a]$</td>
<td>Register</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm$</td>
<td>$M[Imm]$</td>
<td>Absolute</td>
</tr>
<tr>
<td>Memory</td>
<td>$(E_a)$</td>
<td>$M[R[E_a]]$</td>
<td>Indirect</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(E_b)$</td>
<td>$M[Imm + R[E_b]]$</td>
<td>Base + displacement</td>
</tr>
<tr>
<td>Memory</td>
<td>$(E_b,E_i)$</td>
<td>$M[R[E_b] + R[E_i]]$</td>
<td>Indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(E_b,E_i)$</td>
<td>$M[Imm + R[E_b] + R[E_i]]$</td>
<td>Indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$(E_i,s)$</td>
<td>$M[R[E_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(E_i,s)$</td>
<td>$M[Imm + R[E_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$(E_b,E_i,s)$</td>
<td>$M[R[E_b] + R[E_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(E_b,E_i,s)$</td>
<td>$M[Imm + R[E_b] + R[E_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
</tbody>
</table>

Copyright © 2011, 2003 by Randal E. Bryant and David R. O’Hallaron
# 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>0x8+%edx</code></td>
<td><code>0xf008</code></td>
</tr>
<tr>
<td><code>(%edx,%ecx)</code></td>
<td><code>%edx+%ecx</code></td>
<td><code>0xf100</code></td>
</tr>
<tr>
<td><code>(%edx,%ecx,4)</code></td>
<td><code>%edx+4*%ecx</code></td>
<td><code>0xf400</code></td>
</tr>
<tr>
<td><code>0x80 (%edx,2)</code></td>
<td><code>2*%edx+0x80</code></td>
<td><code>0x1e080</code></td>
</tr>
</tbody>
</table>
**lea instruction**

- **lea** *Src, Dest*
  - *Src* is address mode expression
  - Set *Dest* to a register

- 2 common uses
  - Computing address
    - E.g., translation of `p = &x[i];`
  - Computing arithmetic expressions `x + k*y` for `k = 1, 2, 4, or 8`
    - Example

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

```assembly
lea (%eax,%eax,2), %eax ;t ← x+x*2
sall $2, %eax ;return t<<2
```
### Arithmetic Operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>TODO: Define Pointer Type</th>
<th>TODO: Define Pointer Type</th>
<th>TODO: Define Pointer Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>addl</td>
<td>Src, Dest</td>
<td>Dest = Dest + Src</td>
<td></td>
</tr>
<tr>
<td>subl</td>
<td>Src, Dest</td>
<td>Dest = Dest − Src</td>
<td></td>
</tr>
<tr>
<td>imull</td>
<td>Src, Dest</td>
<td>Dest = Dest * Src</td>
<td></td>
</tr>
<tr>
<td>incl</td>
<td>Dest</td>
<td>Dest = Dest + 1</td>
<td></td>
</tr>
<tr>
<td>decl</td>
<td>Dest</td>
<td>Dest = Dest − 1</td>
<td></td>
</tr>
<tr>
<td>negl</td>
<td>Dest</td>
<td>Dest = − Dest</td>
<td></td>
</tr>
<tr>
<td>notl</td>
<td>Dest</td>
<td>Dest = ¬Dest</td>
<td></td>
</tr>
<tr>
<td>sall</td>
<td>Src, Dest</td>
<td>Dest = Dest ≪ Src</td>
<td></td>
</tr>
<tr>
<td>sarl</td>
<td>Src, Dest</td>
<td>Dest = Dest ≫ Src</td>
<td></td>
</tr>
<tr>
<td>xorl</td>
<td>Src, Dest</td>
<td>Dest = Dest ^ Src</td>
<td></td>
</tr>
<tr>
<td>andl</td>
<td>Src, Dest</td>
<td>Dest = Dest &amp; Src</td>
<td></td>
</tr>
<tr>
<td>orl</td>
<td>Src, Dest</td>
<td>Dest = Dest</td>
<td>Src</td>
</tr>
</tbody>
</table>
Arithmetic Expression Example

```c
int foo(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;
}
```

Note: x, y, and z are stored at offsets 8, 12, and 16 from %ebp
int foo(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

- Instructions in different order from C code
- Some expressions require multiple instructions
- Some instructions cover multiple expressions

```c
int foo(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
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 bar(int x, int y)
{
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

```assembly
bar:
    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
```

$2^{13} = 8192, 2^{13} - 7 = 8185$
Processor State (IA32)

- Processor state = information kept in CPU on currently executing program

- **General purpose registers**
  - `%eax`
  - `%ecx`
  - `%edx`
  - `%ebx`
  - `%esi`
  - `%edi`

- **Stack top**
  - `%esp`

- **Stack frame**
  - `%ebp`

- **Instruction pointer**
  - `%eip`

- **Condition codes**
  - `CF`
  - `ZF`
  - `SF`
  - `OF`
Setting Condition Codes

• Can be implicitly set by arithmetic operations

Example: \texttt{addl \ Src,Dest \ (t = a+b)}

\textbf{CF (Carry flag) set} if carry out from most significant (31-st) bit \textit{(unsigned overflow)}, \texttt{inc} and \texttt{dec} do not affect CF

\textbf{ZF (Zero flag) set} if \texttt{t} == 0

\textbf{SF (Sign flag) set} if \texttt{t} < 0 (as signed)

\textbf{OF (Overflow flag) set} if \textit{signed} overflow, i.e. carry out from 30-th bit

\texttt{(a>0 \&\& b>0 \&\& t<0) \texttt{|| (a<0 \&\& b<0 \&\& t>=0)}}

• Not set by \texttt{lea} instruction
Setting Condition Codes

- Can also be explicitly set

  `cmpl b, a` set condition codes based on `(a-b)`

  **CF set** if `(a-b)` results in unsigned overflow
  **ZF set** if `a == b`
  **SF set** if `(a-b) < 0` (as signed)
  **OF set** if `(a-b)` results in signed overflow

  `testl b, a` set condition codes based on value of `(a & b)`
Setting Condition Codes

Important

The processor does not know if you are using signed or unsigned integers. OF and CF are set for every arithmetic operation.
**Reading Condition Codes**

- *SetX dest* instruction: Set the **lower byte** of some register 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>ZF</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.)

- Example

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

- Move Zero-Extended Byte to Long

```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
```
Reading Condition Codes: x86-64

long lgt (long x, long y) {
    return x > y;
}

cmpq %rsi, %rdi
setg %al
movzbl %al, %eax

Is %rax zero?
Yes: 32-bit instructions set high order 32 bits to 0!
Jumping

- **jX label** Instruction: 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>
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
    movl  %edx, %eax
    jmp   .L7

.L6:
    subl  %edx, %eax

.L7:
    popl  %ebp
    ret
Conditional Branch Example

```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:**
```assembly
pushl  %ebp
movl  %esp, %ebp
movl  8(%ebp), %edx
movl  12(%ebp), %eax
cmpl  %eax, %edx
jle   .L6
subl  %eax, %edx
movl  %edx, %eax
jmp   .L7

.L6:
subl  %edx, %eax

.L7:
popl  %ebp
ret
```

### Setup
- Push `%ebp`
- Move `%esp` to `%ebp`
- Move 8 bytes from `%ebp` to `%edx`
- Move 12 bytes from `%ebp` to `%eax`
- Compare `%eax` and `%edx`
- Jump if Less or Equal to `.L6`
- Subtract `%eax` from `%edx`
- Move `%edx` to `%eax`
- Jump to `.L7`

### Body1
- Subtract `%edx` from `%eax`

### Body2a
- Subtract `%eax` from `%edx`

### Body2b
- Pop `%ebp`
- Return

### Finish
Conditional Branch Example

```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:
```assembly
pushl %ebp
movl %esp, %ebp
movl 8(%ebp), %edx
movl 12(%ebp), %eax
cmpl %eax, %edx
jle .L6
subl %eax, %edx
movl %edx, %eax
jmp .L7
.L6:
    subl %edx, %eax
.L7:
    popl %ebp
    ret
```
Conditional Branch Example

```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:
```plaintext
pushl %ebp
movl %esp, %ebp
movl 8(%ebp), %edx
movl 12(%ebp), %eax
cmpl %eax, %edx
jle .L6
subl %eax, %edx
movl %edx, %eax
jmp .L7
.L6:
    subl %edx, %eax
.L7:
    popl %ebp
    ret
```
Conditional Branch Example

```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;
}
```

```asm
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
    jmp .L7
.L6:
    subl %edx, %eax
.L7:
    popl %ebp
    ret
```

Setup

Body1

Body2a

Body2b

Finish
"Do-While" Loop Example

- Count number of 1's in argument \( x \)

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

```c
int pcount_do(unsigned x)
{
    int result = 0;
    loop:
    result += x & 0x1;
    x >>= 1;
    if (x)
        goto loop;
    return result;
}
```
"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;
}
```

- Registers:
  - %edx  x
  - %ecx  result

```
movl  $0, %ecx     #  result = 0
.L2:                #  loop:
    movl  %edx, %eax
    andl  $1, %eax    #    t = x & 1
    addl  %eax, %ecx  #    result += t
    shrl  %edx        #    x >>= 1
    jne   .L2         #    If !0, goto loop
```
General “Do-While” and “While” Translation

**Goto Version**

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

**Goto Version**

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

**Goto Version**

```plaintext
while (Test)
    Body
```

**Goto Version**

```plaintext
if (!Test)
    goto done;
loop:
    Body
if (Test)
    goto loop;
done:
```
“For” Loop Translation

```
for (Init; Test; Update) {
    Body
}
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;
}
```

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;
}
```
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;
}
```

x in %edi, y in %esi

```assembly
absdiff:
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
Conclusions

• We have covered
  – Complete addressing mode, address computation (leal)
  – Arithmetic operations
  – Control: Condition codes
  – Conditional branches & conditional moves
  – Loops