Today:

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

Complete Memory Addressing Modes

- Most General Form
- \( D(R_b, R_i, S) \) \( \text{Mem}[R_b[R_i] + S*\text{Reg}[R_i] + D] \)
  - \( D \): Constant "displacement" 1, 2, or 4 bytes
  - \( R_b \): Base register: Any of 8 integer registers
  - \( R_i \): Index register: Any, except for \( %\text{esp} \)
  - \( S \): Scale: 1, 2, 4, or 8 (why these numbers?)

- Special Cases
  - \((R_b, R_i)\) \( \text{Mem}[R_b[R_i]] \)
  - \( (R_b, R_i, 4) \) \( \text{Mem}[R_b[R_i] + 4*\text{Reg}[R_i]] \)
  - \( (R_b, S, R_i) \) \( \text{Mem}[R_b[R_i] + S*\text{Reg}[R_i]] \)

Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx 0xf000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ecx 0x0100</td>
<td></td>
<td></td>
</tr>
<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, %ecx)</td>
<td>2*0xf000 + 0x800</td>
<td>0x1e080</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; }
  ```

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>shrll</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>xorl</td>
<td>Dest = Dest ^ Src</td>
</tr>
<tr>
<td>orl</td>
<td>Dest = Dest</td>
</tr>
</tbody>
</table>

  - Also called shll
  - Arithmetic
  - Logical

- One Operand Instructions

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

  - See book for more instructions
  - Watch out for argument order!
  - No distinction between signed and unsigned int (why?)

Arithmetic Expression Example

```c
int arith(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = x*y;
    int t3 = x*z;
    int t4 = y+48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

Understanding `arith`

```
Offset  | Old %ebp | %ebp |
--------|----------|------|
16      |          | x    |
12      |          | y    |
8       |          | x    |
4       | Rtn Addr |
0       |          | %ebp |
```

Note: `x`, `y`, and `z` are stored at offsets 8, 12, and 16 from `%ebp`
Understanding `arith`

```c
int arith(int x, int y, int z)
{
    int t1 = x*y;
    int t2 = x*t1;
    int t3 = x*y;
    int t4 = y + z;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return 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 arith(int x, int y, int z)
{
    int t1 = x*y;
    int t2 = x*t1;
    int t3 = x*y;
    int t4 = y + z;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return 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;
}
```

```
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;
}
```
Today

- Complete addressing mode, address computation (lea)
- 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)

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)
  - 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)
  - SF set if two’s-complement (signed) overflow
  - OF set if two’s-complement (signed) overflow

- Not set by lea instruction

Condition Codes (Explicit Setting: Compare)

- Explicit Setting by Compare Instruction
  - cmpb/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)
  - SF set if (a-b) < 0 (as signed)
  - OF set if two’s-complement (signed) overflow

- ZF set when a&b == 0
- SF set when 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 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>setls</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 movzb1 to finish job

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

**Body**

```c
movl 12(%ebp), %eax # eax = y
cmpl %eax, 8(%ebp) # Compare x : y
setg %al # al = x > y
movzb1 %al, %eax # Zero rest of %eax
```

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></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>jz</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jnz</td>
<td>-SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg</td>
<td>(-SF^OF)+ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>(-SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>jl</td>
<td>-SF^OF</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>-CF+ZF</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>jg</td>
<td>CF ^ SF</td>
<td>Greater (Signed)</td>
</tr>
</tbody>
</table>

Conditional Branch Example

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

Conditional Branch Example (Cont.)

```c
int goto_ad(int x, int y) {
    int result;
    if (x <= y) goto Else;
    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

Today

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

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

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:
pushl %ebp
movl $0, %ecx  # result = 0
.L2:  # loop:
movl %edx, %eax
andl $1, %eax  # t = x & l
addl %eax, %eax  # result += t
shrl %edx  # x >>= 1
jne .L2  # if 0, goto loop
ret
```

```
result;
```

```
Setup
Body1
Body2a
Body2b
Body1
Body2b
```

```
result = 0;
```

```
Finish
```

“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;
}
```

**C Code**

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

**Goto Version**

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

- Body:
  - Statement;
  - Statement;
  - ...
  - Statement;
- Test returns integer
  - 0 interpreted as false
  - ≠ 0 interpreted as true
“While” Loop Example

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

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

• Is this code equivalent to the do-while version?
  • Must jump out of loop if test fails

“For” Loop Example

C Code
```c
#define WSIZE (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;
}
```

“For” Loop Form

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

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

“For” Loop → While Loop

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

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

“For” Loop → … → Goto

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

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

“For” Loop Example

C Code
```c
#define WSIZE 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;
}
```

“For” Loop Form

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

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

“For” Loop → While Loop

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

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

“For” Loop → … → Goto

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

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

“For” Loop Example

C Code
```c
#define WSIZE 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;
}
```

“For” Loop Form

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

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

“For” Loop → While Loop

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

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

“For” Loop → … → Goto

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

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

“For” Loop Example

C Code
```c
#define WSIZE 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;
}
```

“For” Loop Form

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

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

“For” Loop → While Loop

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

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

“For” Loop → … → Goto

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

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

“For” Loop Example

C Code
```c
#define WSIZE 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;
}
```

“For” Loop Form

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

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

“For” Loop → While Loop

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

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

“For” Loop → … → Goto

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

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

“For” Loop Example

C Code
```c
#define WSIZE 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;
}
```

“For” Loop Form

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

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

“For” Loop → While Loop

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

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

“For” Loop → … → Goto

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

While Version
```
Init;
while (Test) {
    Body
    Update;
}
```
"For" Loop Conversion Example

<table>
<thead>
<tr>
<th>C Code</th>
<th>Goto Version</th>
</tr>
</thead>
</table>
| #define WSIZE \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;  
} | int pcount_for_gt(unsigned x) {  
    int i;  
    int result = 0;  
    i = 0;  
    if (!i < WSIZE) {Test  
        goto done;  
    }  
    loop:  
        unsigned mask = 1 << i;  
        result += (x & mask) != 0;  
        i++;  
        if (i < WSIZE) Test  
            goto loop;  
    done:  
        return result;  
} |

Summary

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

DNHI

Write the assembly code that implements 3 versions of pcount function discussed in these slides. Then use GCC to generate the assembly code for the three functions and see how they differ from yours.