Machine Programming II: Instructions

- Move instructions, registers, and operands
- Complete addressing mode, address computation (lea)
- Arithmetic operations (including some x86-64 instructions)
- Condition codes
- Control, unconditional and conditional branches
- While loops

Integer Registers (IA32)

<table>
<thead>
<tr>
<th>General Purpose</th>
<th>General Purpose</th>
<th>General Purpose</th>
<th>General Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>%eax</td>
<td>%eax</td>
<td>%eax</td>
</tr>
<tr>
<td>%ecx</td>
<td>%ecx</td>
<td>%ecx</td>
<td>%ecx</td>
</tr>
<tr>
<td>%edx</td>
<td>%edx</td>
<td>%edx</td>
<td>%edx</td>
</tr>
<tr>
<td>%ebx</td>
<td>%ebx</td>
<td>%ebx</td>
<td>%ebx</td>
</tr>
<tr>
<td>%esi</td>
<td>%esi</td>
<td>%esi</td>
<td>%esi</td>
</tr>
<tr>
<td>%edi</td>
<td>%edi</td>
<td>%edi</td>
<td>%edi</td>
</tr>
<tr>
<td>%esp</td>
<td>%esp</td>
<td>%esp</td>
<td>%esp</td>
</tr>
<tr>
<td>%ebp</td>
<td>%ebp</td>
<td>%ebp</td>
<td>%ebp</td>
</tr>
</tbody>
</table>

16-bit virtual registers (backwards compatibility)
Moving Data: IA32

- Moving Data
  - \texttt{movx Source, Dest}
  - \texttt{x} is one of \{b, w, l\}
  - \texttt{movl Source, Dest:}
    Move 4-byte “long word”
  - \texttt{movw Source, Dest:}
    Move 2-byte “word”
  - \texttt{movb Source, Dest:}
    Move 1-byte “byte”

- Lots of these in typical code

Moving Data: IA32

- Moving Data
  - \texttt{movl Source, Dest:}

- Operand Types
  - \texttt{Immediate:} Constant integer data
    - Example: $0x400$, $-533$
    - Like C constant, but prefixed with `\$`
    - Encoded with 1, 2, or 4 bytes
  - \texttt{Register:} One of 8 integer registers
    - Example: \%eax, \%edx
    - But \%esp and \%ebp reserved for special use
    - Others have special uses for particular instructions
  - \texttt{Memory:} 4 consecutive bytes of memory at address given by register
    - Simplest example: (\%eax)
    - Various other “address modes”
movl Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src, Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reg</td>
<td>Reg</td>
<td>movl $0x4, %eax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl $-147, (%eax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl (%eax), %edx</td>
<td>temp = *p;</td>
</tr>
</tbody>
</table>

Cannot do memory-memory transfer with a single instruction

Simple Memory Addressing Modes

- Normal (R) Mem[Reg[R]]
  - Register R specifies memory address

  movl (%ecx), %eax

- Displacement D(R) Mem[Reg[R]+D]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

  movl 8(%ebp), %edx
Using Simple Addressing Modes

```c
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```assembly
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}

movl 12(%ebp),%ecx       # ecx = yp
movl 8(%ebp),%edx        # edx = xp
movl (%ecx),%eax         # eax = *yp (t1)
movl (%edx),%ebx         # ebx = *xp (t0)
movl %eax,%edx         # *xp = eax
movl %ebx,%ecx         # *yp = ebx

movl 12(%ebp),%ecx       # ecx = yp
movl 8(%ebp),%edx        # edx = xp
movl (%ecx),%eax         # eax = *yp (t1)
movl (%edx),%ebx         # ebx = *xp (t0)
movl %eax,%edx         # *xp = eax
movl %ebx,%ecx         # *yp = ebx
**Understanding Swap**

<table>
<thead>
<tr>
<th>%eax</th>
<th>0x120</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td>0x104</td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
<th>YP</th>
<th>XP</th>
<th>%ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td>12</td>
<td>0x120</td>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x120</td>
<td>8</td>
<td>0x124</td>
<td>0x10c</td>
<td></td>
</tr>
<tr>
<td>0x11c</td>
<td>4</td>
<td>0x108</td>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x118</td>
<td></td>
<td></td>
<td>0x104</td>
<td></td>
</tr>
<tr>
<td>0x114</td>
<td>-4</td>
<td>0x100</td>
<td>0x100</td>
<td></td>
</tr>
</tbody>
</table>

- **movl** 12(%ebp),%ecx  # ecx = yp
- **movl** 8(%ebp),%edx  # edx = xp
- **movl** (%ecx),%eax  # eax = *yp (t1)
- **movl** (%edx),%ebx  # ebx = *xp (t0)
- **movl** %eax,%edx    # *xp = eax
- **movl** %ebx,%ecx    # *yp = ebx
Understanding Swap

| %eax  | 456 |
| %edx  | 0x124 |
| %ecx  | 0x120 |
| %ebx  | 123 |
| %esi  |       |
| %edi  |       |
| %esp  |       |
| %ebp  | 0x104 |

movl 12(%ebp),%ecx  # ecx = yp
movl 8(%ebp),%edx  # edx = xp
movl (%ecx),%eax  # eax = *yp (t1)
movl (%edx),%ebx  # ebx = *xp (t0)
movl %eax,%edx  # *xp = eax
movl %ebx,%ecx  # *yp = ebx
Understanding Swap

 movl 12(%ebp),%ecx    # ecx = yp 
 movl 8(%ebp),%edx      # edx = xp 
 movl (%ecx),%eax       # eax = *yp (t1) 
 movl (%edx),%ebx       # ebx = *xp (t0) 
 movl %eax,(%edx)       # *xp = eax 
 movl %ebx,(%ecx)       # *yp = ebx 

---

Understanding Swap

 movl 12(%ebp),%ecx    # ecx = yp 
 movl 8(%ebp),%edx      # edx = xp 
 movl (%ecx),%eax       # eax = *yp (t1) 
 movl (%edx),%ebx       # ebx = *xp (t0) 
 movl %eax,(%edx)       # *xp = eax 
 movl %ebx,(%ecx)       # *yp = ebx
Complete Memory Addressing Modes

■ Most General Form

\[ D(Rb,Ri,S) \quad \text{Mem}[\text{Reg}[Rb]+S\times\text{Reg}[Ri]+D] \]

- \( D \): Constant “displacement” 1, 2, or 4 bytes
- \( Rb \): Base register: Any of 8 integer registers
- \( Ri \): Index register: Any, except for \( \text{esp} \)
  - Unlikely you’d use \( \text{ebp} \), either
- \( S \): Scale: 1, 2, 4, or 8 (\textit{why these numbers?})

■ Special Cases

\[
\begin{align*}
(Rb,Ri) & \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri]] \\
D(Rb,Ri) & \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] + D] \\
(Rb,Ri,S) & \quad \text{Mem}[\text{Reg}[Rb]+S\times\text{Reg}[Ri]]
\end{align*}
\]

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></td>
<td></td>
</tr>
<tr>
<td>(%edx,%ecx)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%edx,%ecx,4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x80 (%edx,2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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>

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*i`
    - `k = 1, 2, 4, or 8`
Some Arithmetic Operations

- **Two Operand Instructions:**

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>addl</strong></td>
<td>Dest = Dest + Src</td>
</tr>
<tr>
<td><strong>subl</strong></td>
<td>Dest = Dest - Src</td>
</tr>
<tr>
<td><strong>imull</strong></td>
<td>Dest = Dest * Src</td>
</tr>
<tr>
<td><strong>sall</strong></td>
<td>Dest = Dest &lt;&lt; Src</td>
</tr>
<tr>
<td><strong>sarl</strong></td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td><strong>shrl</strong></td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td><strong>xorl</strong></td>
<td>Dest = Dest ^ Src</td>
</tr>
<tr>
<td><strong>andl</strong></td>
<td>Dest = Dest &amp; Src</td>
</tr>
<tr>
<td><strong>orl</strong></td>
<td>Dest = Dest</td>
</tr>
</tbody>
</table>

- No distinction between signed and unsigned int (why?)

- **One Operand Instructions**

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

- See book for more instructions
Using `leal` for Arithmetic Expressions

```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;
}
```

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

```
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),%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
```

What does each of these instructions mean?

Stack

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

%ebp
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),%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)
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),%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)
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;
}
```

```
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
```

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;
}
```

```
movl 8(%ebp),%eax       # eax = x
xorl 12(%ebp),%eax      # eax = x^y (t1)
sarl $17,%eax           # eax = t1>>17 (t2)
andl $8185,%eax         # eax = t2 & 8185
```
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:**
```assembly
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

Set Up

Body

Finish

Example in Assembly:
```assembly
movl 8(%ebp),%eax
xorl 12(%ebp),%eax
sarl $17,%eax
andl $8185,%eax
```

```
eax = x
eax = x^y (t1)
eax = t1>>17 (t2)
eax = t2 & 8185
```

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

Set Up

Body

Finish

Example in Assembly:
```assembly
movl 8(%ebp),%eax
xorl 12(%ebp),%eax
sarl $17,%eax
andl $8185,%eax
```

```
eax = x
eax = x^y (t1)
eax = t1>>17 (t2)
eax = t2 & 8185
```

2^{13} = 8192, 2^{13} - 7 = 8185