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>Integer Register</th>
<th>General Purpose</th>
<th>Origin (mostly obsolete)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>%ax</td>
<td>%ah %al</td>
</tr>
<tr>
<td>%ecx</td>
<td>%cx</td>
<td>%ch %cl</td>
</tr>
<tr>
<td>%edx</td>
<td>%dx</td>
<td>%dh %dl</td>
</tr>
<tr>
<td>%ebx</td>
<td>%bx</td>
<td>%bh %bl</td>
</tr>
<tr>
<td>%esi</td>
<td>%si</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td>%di</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td>%sp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>%bp</td>
<td></td>
</tr>
</tbody>
</table>

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

- Moving Data
  - `movx Source, Dest`
  - `x` is one of {`b`, `w`, `l`}

  - `movl Source, Dest`
    - Move 4-byte “long word”
  - `movw Source, Dest`
    - Move 2-byte “word”
  - `movb Source, Dest`
    - Move 1-byte “byte”

- Lots of these in typical code

Operand Types

- **Immediate**: Constant integer data
  - Example: `$0x400, $-533`
  - Like C constant, but prefixed with `$`
  - Encoded with 1, 2, or 4 bytes

- **Register**: One of 8 integer registers
  - Example: `%eax, %edx`
  - But `%esp` and `%ebp` reserved for special use
  - Others have special uses for particular instructions

- **Memory**: 4 consecutive bytes of memory at address given by register
  - Simplest example: (`%eax`)
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><em>Imm</em></td>
<td>Reg</td>
<td>movl $0x4,%eax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>movl $-147,(%eax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td></td>
<td>Reg</td>
<td>movl %eax,%edx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>movl %eax,(%edx)</td>
<td>*p = temp;</td>
</tr>
<tr>
<td></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;
}
```

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

```
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
```
### Understanding Swap

#### Function Definition

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

#### Stack Layout

<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>yp</td>
</tr>
<tr>
<td>8</td>
<td>xp</td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>0</td>
<td>Old %ebp</td>
</tr>
<tr>
<td>-4</td>
<td>Old %ebx</td>
</tr>
</tbody>
</table>

#### Register Values

- `%ecx` = yp
- `%edx` = xp
- `%eax` = t1
- `%ebx` = t0

#### Machine Code

- `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

#### Memory Addressing

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

#### Machine Code

- `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
%edx
%ecx 0x120
%ebx
%esi
%edi
%esp
%ebp 0x104
```

<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>Rtn adr</td>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x118</td>
<td>0</td>
<td>0x104</td>
<td>0x100</td>
<td></td>
</tr>
<tr>
<td>0x114</td>
<td>-4</td>
<td>0x114</td>
<td>0x114</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 |  |

#### Address

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x124</td>
</tr>
<tr>
<td></td>
<td>0x120</td>
</tr>
<tr>
<td></td>
<td>0x11c</td>
</tr>
<tr>
<td></td>
<td>0x118</td>
</tr>
<tr>
<td></td>
<td>0x114</td>
</tr>
<tr>
<td></td>
<td>0x110</td>
</tr>
<tr>
<td></td>
<td>0x124</td>
</tr>
<tr>
<td></td>
<td>0x10c</td>
</tr>
<tr>
<td></td>
<td>0x108</td>
</tr>
<tr>
<td></td>
<td>0x104</td>
</tr>
<tr>
<td></td>
<td>0x100</td>
</tr>
</tbody>
</table>

#### Offset

<table>
<thead>
<tr>
<th></th>
<th>yp</th>
<th>xp</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0x120</td>
<td>0x124</td>
</tr>
<tr>
<td>8</td>
<td>0x110</td>
<td>0x10c</td>
</tr>
<tr>
<td>4</td>
<td>0x108</td>
<td>0x104</td>
</tr>
</tbody>
</table>

#### Addresses

- \texttt{movl} 12(%ebp),%ecx \quad \# ecx = yp
- \texttt{movl} 8(%ebp),%edx \quad \# edx = xp
- \texttt{movl} (%ecx),%eax \quad \# eax = *yp (t1)
- \texttt{movl} (%edx),%ebx \quad \# ebx = *xp (t0)
- \texttt{movl} %eax,(%edx) \quad \# *xp = eax
- \texttt{movl} %ebx,(%ecx) \quad \# *yp = ebx
Complete Memory Addressing Modes

■ Most General Form

\[ D(R_b, R_i, S) \quad \text{Mem}[R_b+S\times R_i+D] \]

- \( D \): 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 (*why these numbers?*)

■ Special Cases

- \((R_b, R_i)\) \quad \text{Mem}[R_b+R_i]
- \( D(R_b, R_i) \quad \text{Mem}[R_b+R_i+D] \)
- \((R_b, R_i, S)\) \quad \text{Mem}[R_b+S\times R_i]\n
Address Computation Examples

\[
\begin{array}{|c|c|}
\hline
\%edx & 0xf000 \\
\%ecx & 0x100 \\
\hline
\end{array}
\]

<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

- **lea**l <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>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 Also called shll</td>
</tr>
<tr>
<td>sarl</td>
<td>Dest = Dest &gt;&gt; Src Arithmetic</td>
</tr>
<tr>
<td>shrl</td>
<td>Dest = Dest &gt;&gt; Src Logical</td>
</tr>
<tr>
<td>xorl</td>
<td>Dest = Dest ^ Src</td>
</tr>
<tr>
<td>andl</td>
<td>Dest = Dest &amp; Src</td>
</tr>
<tr>
<td>orl</td>
<td>Dest = Dest</td>
</tr>
</tbody>
</table>

- **One Operand Instructions**

  | incl    | Dest = Dest + 1 |
  | decl    | Dest = Dest - 1 |
  | negl    | Dest = -Dest |
  | notl    | Dest = ~Dest |

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

```assembly
    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
```

#### Set Up

1. `pushl %ebp`
2. `movl %esp,%ebp`
3. `movl 8(%ebp),%eax`
4. `movl 12(%ebp),%edx`
5. `leal (%edx,%eax),%ecx`
6. `leal (%edx,%edx,2),%edx`
7. `sall $4,%edx`
8. `addl 16(%ebp),%ecx`
9. `leal 4(%edx,%eax),%eax`
10. `imull %ecx,%eax`

#### Body

11. `movl %ebp,%esp`
12. `popl %ebp`
13. `ret`

#### Finish

### 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;
}
```

**Stack**

- `z`
- `y`
- `x`
- `Rtn adr`
- `Old %ebp`

#### What does each of these instructions mean?

- `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`:
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;
}
```

Stack

Offset  Stack
16       z
12       y
8        x
4        Rtn adr
0  Old %ebp

movl 8(%ebp),%eax  # eax = x
movl 12(%ebp),%edx  # edx = y
leal (%edx,%eax),%ecx  # ecx = x+y (t1)
leal (%edx,%eax),%ecx  # ecx = x+y (t1)
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;
}
```

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

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:
  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

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
```

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

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
```