Introduction to Computer Systems
15-213, fall 2009
5th Lecture, Sep. 7th

Instructors:
Majd Sakr and Khaled Harras
Last Time: Machine Programming, Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly (IA32):
  - Registers
  - Operands
  - Move (what’s the l in movl?)

```
movl $0x4,%eax
movl %eax,%edx
movl (%eax),%edx
```
Today

- Addressing modes, address computation (leal)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches
- While loops
Simple Memory Addressing Modes

- **Normal (R) Mem[Reg[R]]**
  - Register R specifies memory address
  
  \[
  \text{movl} \ (%\text{ecx}),\%\text{eax}
  \]

- **Displacement D(R) Mem[Reg[R]+D]**
  - Register R specifies start of memory region
  - Constant displacement D specifies offset
  
  \[
  \text{movl} \ 8(%\text{ebp}),\%\text{edx}
  \]
Using Simple Addressing Modes

_{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\_\_

Set Up

Body

Finish
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
```

- **Set Up**
- **Body**
- **Finish**
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
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

%eax
%edx
%ecx
%ebx
%esi
%edi
%esp
%ebp 0x104
### Understanding Swap

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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp</td>
<td>12</td>
</tr>
<tr>
<td>xp</td>
<td>8</td>
</tr>
<tr>
<td>%ebp</td>
<td>0</td>
</tr>
<tr>
<td>Rtn adr</td>
<td>0</td>
</tr>
</tbody>
</table>

<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>0</td>
</tr>
<tr>
<td>0x10c</td>
<td>0</td>
</tr>
<tr>
<td>0x108</td>
<td>0</td>
</tr>
<tr>
<td>0x104</td>
<td>0</td>
</tr>
<tr>
<td>0x100</td>
<td>0</td>
</tr>
</tbody>
</table>
Understanding Swap

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x120</td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x118</td>
</tr>
<tr>
<td>%ebx</td>
<td>0x11c</td>
</tr>
<tr>
<td>%esi</td>
<td>0x114</td>
</tr>
<tr>
<td>%edi</td>
<td>0x110</td>
</tr>
<tr>
<td>%esp</td>
<td>0x108</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0x120</td>
</tr>
<tr>
<td>8</td>
<td>0x124</td>
</tr>
<tr>
<td>4</td>
<td>0x10c</td>
</tr>
<tr>
<td>0</td>
<td>0x104</td>
</tr>
<tr>
<td>-4</td>
<td>0x100</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

```
  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>Variable</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>456</td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td>123</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>

\[
\text{movl } 12(\%ebp),\%ecx # \text{ecx} = \text{yp} \\
\text{movl } 8(\%ebp),\%edx # \text{edx} = \text{xp} \\
\text{movl } (%ecx),\%eax # \text{eax} = *\text{yp} \ (t1) \\
\text{movl } (%edx),\%ebx # \text{ebx} = *\text{xp} \ (t0) \\
\text{movl } \%eax,(%edx) # *\text{xp} = \text{eax} \\
\text{movl } \%ebx,(%ecx) # *\text{yp} = \text{ebx} \\
\]

Address Offset

<table>
<thead>
<tr>
<th>Variable</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp</td>
<td>12</td>
</tr>
<tr>
<td>xp</td>
<td>8</td>
</tr>
<tr>
<td>Rtn adr</td>
<td>0</td>
</tr>
<tr>
<td>%ebp</td>
<td>0</td>
</tr>
</tbody>
</table>

Offset

<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>0</td>
</tr>
<tr>
<td>0x10c</td>
<td>-4</td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x104</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td></td>
</tr>
</tbody>
</table>
Understanding Swap

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

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>456</td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td>123</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>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp</td>
<td>12</td>
</tr>
<tr>
<td>xp</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td>%ebp</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>-4</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
Complete Memory Addressing Modes

■ Most General Form

\[ D(Rb,Ri,S) \rightarrow Mem[Reg[Rb]+S\times Reg[Ri]+D] \]

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

■ Special Cases

- \((Rb,Ri)\rightarrow Mem[Reg[Rb]+Reg[Ri]]\)
- \(D(Rb,Ri)\rightarrow Mem[Reg[Rb]+Reg[Ri]+D]\)
- \((Rb,Ri,S)\rightarrow Mem[Reg[Rb]+S\times Reg[Ri]]\)
# 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>will disappear blackboard?</td>
<td></td>
</tr>
</tbody>
</table>

- dx = 0xf000
- ecx = 0x100
# Address Computation Examples

<table>
<thead>
<tr>
<th>%edx</th>
<th>0xf000</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>0x100</td>
</tr>
</tbody>
</table>

<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*y`
    - `k = 1, 2, 4, or 8`

- **Example**
Today

- Complete addressing mode, address computation (lea1)
- Arithmetic operations
- x86-64
- 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>sarl</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>shr1</td>
<td>Dest = Dest &gt;&gt; Src</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>

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

- One Operand Instructions

  incl  Dest  \[ Dest = Dest + 1 \]
  decl Dest  \[ Dest = Dest - 1 \]
  negl Dest  \[ Dest = \neg Dest \]
  notl Dest  \[ Dest = \neg Dest \]

- 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;
}
```

```assembly
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

```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 (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)
```

Stack

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

will disappear
blackboard?
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;
}
```

```assembly
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
sal $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)
```

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)
```
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  # 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;
}
```

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

**Movl 8(%ebp),%eax** # eax = x
**Xorl 12(%ebp),%eax** # eax = x^y
**Sarlx $17,%eax** # eax = t1>>17
**Andl $8185,%eax** # eax = t2 & 8185
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:**

```
pushl %ebp
movl %esp,%ebp

movl 8(%ebp),%eax         \( \text{eax} = x \)
xorl 12(%ebp),%eax        \( \text{eax} = x^y \) (t1)
sarl $17,%eax             \( \text{eax} = t1>>17 \) (t2)
andl $8185,%eax           \( \text{eax} = t2 \& 8185 \)
```

```
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:
- **Set Up**
  - `pushl %ebp`
  - `movl %esp,%ebp`

- **Body**
  - `movl 8(%ebp),%eax`
  - `xorl 12(%ebp),%eax`
  - `sarl $17,%eax`
  - `andl $8185,%eax`

- **Finish**
  - `movl %ebp,%esp`
  - `popl %ebp`
  - `ret`

2\(^{13}\) = 8192, 2\(^{13}\) – 7 = 8185

movl 8(%ebp),%eax  
eaax = x
xorl 12(%ebp),%eax  
eaax = x^y \quad (t1)
sarl $17,%eax  
eaax = t1>>17 \quad (t2)
andl $8185,%eax  
eaax = t2 & 8185
Today

- Complete addressing mode, address computation (lea)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches
- While loops
## Data Representations: IA32 + x86-64

### Sizes of C Objects (in Bytes)

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Typical 32-bit</th>
<th>Intel IA32</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>8</td>
<td>10/12</td>
<td>16</td>
</tr>
<tr>
<td>char *</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

Or any other pointer
## x86-64 Integer Registers

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
<th>%r8</th>
<th>%r8d</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
<td>%r15</td>
<td>%r15d</td>
</tr>
</tbody>
</table>

- Extend existing registers. Add 8 new ones.
- Make %ebp/%rbp general purpose
Instructions

- Long word \( 1 \) (4 Bytes) ↔ Quad word \( q \) (8 Bytes)

- New instructions:
  - \texttt{movl} → \texttt{movq}
  - \texttt{addl} → \texttt{addq}
  - \texttt{sall} → \texttt{salq}
  - etc.

- 32-bit instructions that generate 32-bit results
  - Set higher order bits of destination register to 0
  - Example: \texttt{addl}
Today

- Complete addressing mode, address computation (leal)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches
- While 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
  - **SF** Sign Flag (for signed)
  - **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 overflow)
    - **ZF set** if \( t = 0 \)
    - **SF set** if \( t < 0 \) (as signed)
    - **OF set** if two’s complement (signed) overflow
      \( (a>0 \land \land b>0 \land \land t<0) \lor (a<0 \land \land b<0 \land \land t>=0) \)

- Not set by `lea` instruction
- **Full documentation** (IA32), link also on course website
Condition Codes (Explicit Setting: Compare)

- **Explicit Setting by Compare Instruction**
  
  \[ \text{cmp1/cmpq } \text{Src2,Src1} \]
  
  \[ \text{cmp1 } b,a \text{ like computing } a-b \text{ without setting destination} \]

  - **CF set** if carry out from most significant bit (used for unsigned comparisons)
  - **ZF set** if \( a == b \)
  - **SF set** if \( (a-b) < 0 \) (as signed)
  - **OF set** if two’s complement (signed) overflow \( (a>0 \land b<0 \land (a-b)<0) \lor (a<0 \land b>0 \land (a-b)>0) \)
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>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

```
movl 12(%ebp),%eax
cmpl %eax,8(%ebp)
setg %al
movzbl %al,%eax
```

Will disappear Blackboard?
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

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

Body
```
movl 12(%ebp),%eax         # eax = y
cmp1 %eax,8(%ebp)          # Compare x and y
setg %al                   # al = x > y
movzbl %al,%eax            # Zero rest of %eax
```

Note inverted ordering!
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>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>Less or Equal (Signed)</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>
Today

- Complete addressing mode, address computation (lea1)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches
- While loops
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
    movl %edx, %eax
.L8:
    leave
    ret
.L7:
    subl %edx, %eax
    jmp .L8

Body1
Setup
Finish
Body2
Conditional Branch Example (Cont.)

```c
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;
}
```

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

```assembly
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
```
Conditional Branch Example (Cont.)

```c
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;
}
```

```assembly
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
.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;
}

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
Conditional Branch Example (Cont.)

```c
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
    jmp .L8
.L7:
    subl %edx, %eax
.L8:
    leave
    ret
```

General Conditional Expression Translation

C Code

\[
\text{val} = \text{Test} \ ? \ \text{Then-Expr} : \ \text{Else-Expr};
\]

\[
\text{val} = \text{x>y} \ ? \ \text{x-y} : \ \text{y-x};
\]

Goto Version

\[
\text{nt} = \neg \text{Test};
\text{if (nt) goto Else;}
\text{val} = \text{Then-Expr;}
\text{Done:}
\ldots
\text{Else:}
\text{val} = \text{Else-Expr;}
\text{goto Done;}
\]

- \text{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

```
val = Test ? Then-Expr : Else-Expr;
```

Conditional Move Version

```
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 too expensive
Today

- Complete addressing mode, address computation \( \text{lea}1 \)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches
- While loops
“Do-While” Loop Example

**C Code**

```c
int fact_do(int x)
{
    int result = 1;
    do {
        result *= x;
        x = x-1;
    } while (x > 1);
    return result;
}
```

**Goto Version**

```c
int fact_goto(int x)
{
    int result = 1;
    loop:
        result *= x;
        x = x-1;
    if (x > 1)
        goto loop;
    return result;
}
```

- Use backward branch to continue looping
- Only take branch when “while” condition holds
“Do-While” Loop Compilation

Goto Version

```c
int fact_goto(int x)
{
    int result = 1;

    loop:
    result *= x;
    x = x-1;
    if (x > 1)
        goto loop;
    return result;
}
```

Assembly

```assembly
fact_goto:
    pushl %ebp
    movl %esp,%ebp
    movl $1,%eax
    movl 8(%ebp),%edx

.L11:
    imull %edx,%eax
    decl %edx
    cmpl $1,%edx
    jg .L11
    movl %ebp,%esp
    popl %ebp
    ret
```

Registers:
- `%edx`  `x`
- `%eax`  `result`

Will disappear Blackboard?
“Do-While” Loop Compilation

Goto Version

```c
int fact_goto(int x)
{
    int result = 1;

    loop:
        result *= x;
        x = x-1;
        if (x > 1)
            goto loop;

    return result;
}
```

Assembly

```
fact_goto:
    pushl %ebp
    movl %esp,%ebp
    movl $1,%eax
    movl 8(%ebp),%edx
.L11:
    imull %edx,%eax
    decl %edx
    cmpl $1,%edx
    jg .L11
    movl %ebp,%esp
    popl %ebp
    ret
```

Registers:

- %edx : x
- %eax : result

```
```
General “Do-While” Translation

C Code

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

Goto Version

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

- **Body:** 
  
  ```c
  { 
  Statement_1; 
  Statement_2; 
  ... 
  Statement_n; 
  }
  ```

- **Test** returns integer
  
  - = 0 interpreted as false
  - ≠ 0 interpreted as true
“While” Loop Example

C Code

```c
int fact_while(int x)
{
    int result = 1;
    while (x > 1) {
        result *= x;
        x = x-1;
    }
    return result;
}
```

Goto Version #1

```c
int fact_while_goto(int x)
{
    int result = 1;
    loop:
    if (!(x > 1))
        goto done;
    result *= x;
    x = x-1;
    goto loop;
    done:
    return result;
}
```

- Is this code equivalent to the do-while version?
- Must jump out of loop if test fails
Alternative “While” Loop Translation

C Code

```c
int fact_while(int x)
{
    int result = 1;
    while (x > 1) {
        result *= x;
        x = x-1;
    }
    return result;
}
```

Goto Version #2

```c
int fact_while_goto2(int x)
{
    int result = 1;
    if (!(x > 1))
        goto done;
    loop:
        result *= x;
        x = x-1;
        if (x > 1)
            goto loop;
    done:
        return result;
}
```

- Historically used by GCC
- Uses same inner loop as do-while version
- Guards loop entry with extra test
General “While” Translation

While version

```c
while (Test)
    Body
```

Do-While Version

```c
if (!Test)
    goto done;
do
    Body
    while (Test);
done:
```

Goto Version

```c
if (!Test)
    goto done;
loop:
    Body
    if (Test)
        goto loop;
done:
```
New Style “While” Loop Translation

C Code

```c
int fact_while(int x)
{
    int result = 1;
    while (x > 1) {
        result *= x;
        x = x-1;
    }
    return result;
}
```

Goto Version

```c
int fact_while_goto3(int x)
{
    int result = 1;
    goto middle;
    loop:
        result *= x;
        x = x-1;
    middle:
        if (x > 1)
            goto loop;
    return result;
}
```

- Recent technique for GCC
  - Both IA32 & x86-64
- First iteration jumps over body computation within loop
Jump-to-Middle While Translation

C Code

```
while (Test)
  Body
```

- Avoids duplicating test code
- Unconditional goto incurs no performance penalty
- for loops compiled in similar fashion

Goto Version

```
goto middle;
loop:
  Body
middle:
  if (Test)
    goto loop;
```

Goto (Previous) Version

```
if (!Test)
  goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```
Jump-to-Middle Example

```c
int fact_while(int x)
{
    int result = 1;
    while (x > 1) {
        result *= x;
        x--;
    }
    return result;
}
```

```assembly
# x in %edx, result in %eax
    jmp .L34            # goto Middle
.L35:                  # Loop:
    imull %edx, %eax    # result *= x
    decl %edx           # x--
.L34:                  # Middle:
    cmpl $1, %edx       # x:1
    jg .L35            # if >, goto Loop
```
Implementing Loops

- IA32
  - All loops translated into form based on “do-while”
  - IA32 compiler developed for machine where all operations costly