X86 Assembly, and C-to-assembly

- 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
Three Kinds of Instructions

- **Perform arithmetic function on register or memory data**
  - \( c = a + b; \)

- **Transfer data between memory and register**
  - Load data from memory into register
    - \( %\text{reg} = \text{Mem}[\text{address}] \)
  - Store register data into memory
    - \( \text{Mem}[\text{address}] = %\text{reg} \)

- **Transfer control (control flow)**
  - Unconditional jumps to/from procedures
  - Conditional branches
Example

C Code
- Add two signed integers

Assembly
- Add 2 4-byte integers
  - “Long” words in GCC speak
  - Same instruction whether signed or unsigned
- Operands:
  - \( x \): Register \( %eax \)
  - \( y \): Memory \( M[ebp+8] \)
  - \( t \): Register \( %eax \)
    - Return function value in \( %eax \)

Object Code
- 3-byte instruction
- Stored at address \( 0x401046 \)

```c
int t = x+y;
```

```assembly
addl 8(%ebp),%eax
```

Similar to expression:
\( x += y \)

More precisely:
```
int eax;
int *ebp;
eax += ebp[2]
```

```
0x401046: 03 45 08
```
Integer Registers (IA32)

%eax
%ecx
%edx
%ebx
%esi
%edi
%esp
%ebp
# Integer Registers (IA32)

<table>
<thead>
<tr>
<th>Register</th>
<th>General Purpose</th>
<th>Origin (mostly obsolete)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>%ax, %ah, %al</td>
<td>accumulate</td>
</tr>
<tr>
<td>%ecx</td>
<td>%cx, %ch, %cl</td>
<td>counter</td>
</tr>
<tr>
<td>%edx</td>
<td>%dx, %dh, %dl</td>
<td>data</td>
</tr>
<tr>
<td>%ebx</td>
<td>%bx, %bh, %bl</td>
<td>base</td>
</tr>
<tr>
<td>%esi</td>
<td>%si</td>
<td>source index</td>
</tr>
<tr>
<td>%edi</td>
<td>%di</td>
<td>index</td>
</tr>
<tr>
<td>%esp</td>
<td>%sp</td>
<td>destination index</td>
</tr>
<tr>
<td>%ebp</td>
<td>%bp</td>
<td>stack pointer</td>
</tr>
</tbody>
</table>

- **General Purpose:**
  - %eax: General purpose register.
  - %ecx: General purpose register.
  - %edx: General purpose register.
  - %ebx: General purpose register.
  - %esi: General purpose register.
  - %edi: General purpose register.
  - %esp: Stack pointer.
  - %ebp: Base pointer.

- **Origin (mostly obsolete):**
  - Accumulate
  - Counter
  - Data
  - Base
  - Source index
  - Destination index
  - Stack pointer
  - Base pointer

**16-bit virtual registers (backwards compatibility)**
### 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>

- Twice the number of registers
- Accessible as 8, 16, 32, 64 bits
### x86-64 Integer Registers: Usage Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
<td>%r8</td>
<td>Argument #5</td>
</tr>
<tr>
<td>%rbx</td>
<td>Callee saved</td>
<td>%r9</td>
<td>Argument #6</td>
</tr>
<tr>
<td>%rcx</td>
<td>Argument #4</td>
<td>%r10</td>
<td>Caller saved</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument #3</td>
<td>%r11</td>
<td>Caller saved</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument #2</td>
<td>%r12</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%rdi</td>
<td>Argument #1</td>
<td>%r13</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%rsp</td>
<td>Stack pointer</td>
<td>%r14</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%rbp</td>
<td>Callee saved</td>
<td>%r15</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>
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
  `movl Source, Dest`:

- 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 )`
    - 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>Imm</td>
<td>Reg</td>
<td>movl $0x4,%eax</td>
<td></td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl $-147,(%eax)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Reg</td>
<td>movl %eax,%edx</td>
<td></td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl %eax,(%edx)</td>
<td></td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl (%eax),%edx</td>
<td></td>
</tr>
</tbody>
</table>

*Cannot do memory-memory transfer with a single instruction.*

How do you copy from a memory location to another then?
### 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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Simple Memory Addressing Modes

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

  \[
  \text{movl } (\%ecx),\%eax
  \]

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

  \[
  \text{movl } 8(\%ebp),\%edx
  \]
Using Simple Addressing Modes

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

```asm
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
# Understanding Swap

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>yp</td>
</tr>
<tr>
<td>%edx</td>
<td>xp</td>
</tr>
<tr>
<td>%eax</td>
<td>t1</td>
</tr>
<tr>
<td>%ebx</td>
<td>t0</td>
</tr>
</tbody>
</table>

### Stack (in memory)

<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>

```assembly
    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>%edx</th>
<th>%ecx</th>
<th>%ebx</th>
<th>%esi</th>
<th>%edi</th>
<th>%esp</th>
<th>%ebp</th>
<th>0x104</th>
</tr>
</thead>
</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

Offset

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

Address

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
</tr>
<tr>
<td>456</td>
</tr>
<tr>
<td>0x120</td>
</tr>
<tr>
<td>0x11c</td>
</tr>
<tr>
<td>0x118</td>
</tr>
<tr>
<td>0x114</td>
</tr>
<tr>
<td>0x120</td>
</tr>
<tr>
<td>0x110</td>
</tr>
<tr>
<td>0x124</td>
</tr>
<tr>
<td>0x10c</td>
</tr>
<tr>
<td>Rtn adr</td>
</tr>
<tr>
<td>0x108</td>
</tr>
<tr>
<td>0x104</td>
</tr>
<tr>
<td>0x100</td>
</tr>
</tbody>
</table>
Understanding Swap

%eax
%edx
%ecx 0x120
%ebx
%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

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
<th>Yp</th>
<th>Xp</th>
<th>Rtn adr</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>0x104</td>
<td></td>
</tr>
<tr>
<td>0x118</td>
<td></td>
<td>0x104</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x114</td>
<td></td>
<td>0x100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%eax</th>
<th>456</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></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>

```
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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x100</td>
</tr>
<tr>
<td>4</td>
<td>0x108</td>
</tr>
<tr>
<td>8</td>
<td>0x10c</td>
</tr>
<tr>
<td>12</td>
<td>0x110</td>
</tr>
<tr>
<td></td>
<td>0x114</td>
</tr>
<tr>
<td></td>
<td>0x118</td>
</tr>
<tr>
<td></td>
<td>0x120</td>
</tr>
<tr>
<td></td>
<td>0x124</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>

#### Address 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>Rtn adr</td>
</tr>
<tr>
<td>0</td>
<td>0x104</td>
</tr>
<tr>
<td>-4</td>
<td>0x100</td>
</tr>
</tbody>
</table>

#### Code Snippet

```plaintext
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

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

- 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

\( (R_b, R_i) \quad \text{Mem}[\text{Reg}[R_b] + \text{Reg}[R_i]] \)
\( D(R_b, R_i) \quad \text{Mem}[\text{Reg}[R_b] + \text{Reg}[R_i] + D] \)
\( (R_b, R_i, S) \quad \text{Mem}[\text{Reg}[R_b] + S^*\text{Reg}[R_i]] \)
Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8 (%edx)</td>
<td>(Rb,Ri) Mem[Reg[Rb]+Reg[Ri]]</td>
<td></td>
</tr>
<tr>
<td>(%edx,%ecx)</td>
<td>(Rb,Ri) Mem[Reg[Rb]+Reg[Ri]+D]</td>
<td></td>
</tr>
<tr>
<td>(%edx,%ecx,4)</td>
<td>(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]]</td>
<td></td>
</tr>
<tr>
<td>0x80 (%edx,2)</td>
<td>(Rb,Ri) Mem[Reg[Rb]+Reg[Ri]]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%edx</th>
<th>0xf000</th>
<th>(Rb,Ri) Mem[Reg[Rb]+Reg[Ri]]</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>0x100</td>
<td>(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]]</td>
</tr>
</tbody>
</table>
Address Computation Examples

%edx  0xf000
%ecx  0x100

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

### Two Operand Instructions:

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

- *Also called shll*
- *Arithmetic*
- *Logical*

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

■ One Operand Instructions

 incl $Dest$ \hspace{1cm} Dest = Dest + 1
 decl $Dest$ \hspace{1cm} Dest = Dest - 1
 negl $Dest$ \hspace{1cm} Dest = \neg Dest
 notl $Dest$ \hspace{1cm} Dest = \neg\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;
}
```

```
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

Body

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;
}
```

```
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)
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)
```

What does each of these instructions mean?
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)
```
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;
}
```

```
logical:
pushl %ebp
movl %esp,%ebp

movl 8(%ebp),%eax
xorl 12(%ebp),%eax
sar1 $17,%eax
andl $8185,%eax

movl %ebp,%esp
popl %ebp
ret
```

- Set Up
- Body
- Finish

```
movl 8(%ebp),%eax # eax = x
xorl 12(%ebp),%eax # eax = x^y
sar1 $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;
}
```

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
```

```assembly
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;
}
```

```
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;
}
```
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;
}
```

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

```
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)
sarlo $17,%eax     eax = t1>>17  (t2)
andl $8185,%eax   eax = t2 & 8185
```
Control-Flow/Conditionals

- Unconditional

```java
while (true) {
    do_something;
}
...
```

- Conditional

```java
int absdiff(int x, int y) {
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```
Conditionals and Control Flow

- A test / conditional branch is sufficient to implement most control flow constructs offered in higher level languages
  - if (condition) then {...} else {...}
  - while(condition) {...}
  - do {...} while (condition)
  - for (initialization; condition; ) {...}
- (Unconditional branches implemented some related control flow constructs
  - break, continue)
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>
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
  - \%eax
  - \%ecx
  - \%edx
  - \%ebx
  - \%esi
  - \%edi
  - \%esp
  - \%ebp

- Current stack top
- Current stack frame
- Instruction pointer
- Condition codes
Condition Codes (Implicit Setting)

- Single bit registers
  - CF: Carry Flag (for unsigned)
  - SF: Sign Flag (for signed)
  - 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 overflow)
  - ZF set if `t == 0`
  - SF set if `t < 0` (as signed)
  - OF set if two’s complement (signed) overflow
    - `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`

- Not set by `lea` instruction (beware!)

- Full documentation (IA32)
Condition Codes (Explicit Setting: Compare)

- **Explicit Setting by Compare Instruction**
  
  `cmpl/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)
- **ZF set** if `a == b`
- **SF set** if `(a-b) < 0` (as signed)
- **OF set** if two’s complement (signed) overflow
  
  `(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (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`

- `testl %eax, %eax`
  - Sets SF and ZF, check if eax is +,0,-
Reading Condition Codes

- **SetX Instructions**
  - Set a 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>settle</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

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

**Body**

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

```assembly
movl 12(%ebp),%eax  # eax = y
cmpl %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>
Conditional Branch Example

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

```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;
}
```

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

```asm
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
    jmp .L8
```

51
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
    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:`
```assembly
    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
    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;
}
```

```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
    jmp .L8
    .L8:
    leave
    ret
```

General Conditional Expression Translation

C Code

```c
val = Test ? Then-Expr : Else-Expr;
val = x>y ? x-y : y-x;
```

Goto Version

```c
nt = !Test;
if (nt) goto Else;
val = Then-Expr;
Done:
    ...;
Else:
    val = Else-Expr;
    goto Done;
```

- *Test* is expression returning integer
  - = 0 interpreted as false
  - ≠0 interpreted as true
- Create separate code regions for then & else expressions
- Execute appropriate one
- How would you make this efficient?
Conditionals: x86-64

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

```assembly
absdiff: # x in %edi, y in %esi
    movl   %edi, %eax  # eax = x
    movl   %esi, %edx  # edx = y
    subl   %esi, %eax  # eax = x-y
    subl   %edi, %edx  # edx = y-x
    cmpl   %esi, %edi  # x:y
    cmovle %edx, %eax  # eax=edx if <=
    ret
```

- **Conditional move instruction**
  - `cmovC src, dest`
  - Move value from src to dest if condition C holds
  - More efficient than conditional branching (simple control flow)
  - But overhead: both branches are evaluated
PC Relative Addressing

0x100     cmp  r2, r3     0x1000
0x102     je   0x70       0x1002
0x104     …               0x1004
…         …               …
0x172     add  r3, r4     0x1072

- PC relative branches are **relocatable**
- Absolute branches are not
Compiling Loops

C/Java code

```c
while ( sum != 0 ) {
    <loop body>
}
```

Machine code

```c
loopTop:   cmp   r3, $0
            be   loopDone
            <loop body code>
            jmp   loopTop
loopDone:
```

- How to compile other loops should be clear to you
  - The only slightly tricky part is to be sure where the conditional branch occurs: top or bottom of the loop

- Q: How is \[\text{for}(i=0; \ i<100; \ i++)\] implemented?
- Q: How are break and continue implemented?
Machine Programming II: Instructions (cont’d)

- 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
- For loops
- Switch statements
“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

```
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
    poipl %ebp
    ret
```
```
int fact_goto(int x)
{
    int result = 1;

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

    return result;
}
```

Registers:

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

```
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
```

# Setup
# Setup
# eax = 1
# edx = x

# result *= x
# x--
# Compare x : 1
# if > goto loop

# Finish
# Finish
# Finish
General “Do-While” Translation

C Code

do
  
  Body

  while (Test);

Goto Version

loop:
  
  Body

  if (Test)

  goto loop

- **Body:**
  
  \{ 

  \textit{Statement}_1;

  \textit{Statement}_2;

  ... 

  \textit{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

while (Test)
  Body

Do-While Version

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

Goto Version

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

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

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

Goto Version

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

Goto (Previous) Version

```c
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;
}
```

```
# 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
```
Quick Review

■ Complete memory addressing mode
  ▪ (%eax), 17(%eax), 2(%ebx, %ecx, 8), ...

■ Arithmetic operations that do set condition codes
  ▪ subl %eax, %ecx     # ecx = ecx + eax
  ▪ sall $4,%edx       # edx = edx << 4
  ▪ addl 16(%ebp),%ecx  # ecx = ecx + Mem[16+ebp]
  ▪ imull %ecx,%eax    # eax = eax * ecx

■ Arithmetic operations that do NOT set condition codes
  ▪ leal 4(%edx,%eax),%eax  # eax = 4 + edx + eax
Quick Review

- **x86-64 vs. IA32**
  - Integer registers: *16 x 64-bit vs. 8 x 32-bit*
  - `movq, addq, ... vs. movl, addl, ...`
  - Better support for passing function arguments in registers

- **Control**
  - Condition code registers
  - Set as side effect or by `cmp, test`
  - Used:
    - Read out by setx instructions (`setg, setle, ...`)
    - Or by conditional jumps (`jle .L4, je .L10, ...`)
Quick Review

- **Do-While loop**

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

  **Goto Version**
  
  ```c
  loop:
    Body
    if (Test)
    goto loop
  ```

- **While-Do loop**

  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:
  goto middle;
  ```

  *or*

  ```c
  goto middle;
  loop:
    Body
    middle:
    if (Test)
    goto loop;
  ```
“For” Loop Example: Square-and-Multiply

/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned p)
{
    int result;
    for (result = 1; p != 0; p = p>>1) {
        if (p & 0x1)
            result *= x;
        x = x*x;
    }
    return result;
}

- **Algorithm**
  - Exploit bit representation: \( p = p_0 + 2p_1 + 2^2p_2 + \ldots + 2^{n-1}p_{n-1} \)
  - Gives: 
    \[
    x^p = z_0 \cdot z_1^2 \cdot (z_2^2)^2 \cdot \ldots \cdot \ldots((z_{n-1}^2)^2)\ldots
    \]
    
    - \( z_i = 1 \) when \( p_i = 0 \)
    - \( z_i = x \) when \( p_i = 1 \)
  - Complexity \( \mathcal{O}(\log p) \)

**Example**

\[
3^{10} = 3^2 \cdot 3^8 = 3^2 \cdot ((3^2)^2)^2
\]
ipwr Computation

/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned p)
{
    int result;
    for (result = 1; p != 0; p = p>>1) {
        if (p & 0x1)
            result *= x;
        x = x*x;
    }
    return result;
}

<table>
<thead>
<tr>
<th>before iteration</th>
<th>result</th>
<th>x=3</th>
<th>p=10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>10=1010₂</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>9</td>
<td>5=101₂</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>81</td>
<td>2=10₂</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>6561</td>
<td>1=1₂</td>
</tr>
<tr>
<td>5</td>
<td>59049</td>
<td>43046721</td>
<td>0₂</td>
</tr>
</tbody>
</table>
“For” Loop Example

```c
int result;
for (result = 1; p != 0; p = p>>1)
{
    if (p & 0x1)
        result *= x;
    x = x*x;
}
```

**General Form**

```c
for (Init; Test; Update)
    Body
```

<table>
<thead>
<tr>
<th>Test</th>
<th>Init</th>
<th>Update</th>
<th>Body</th>
</tr>
</thead>
<tbody>
<tr>
<td>p != 0</td>
<td>result = 1</td>
<td>p = p &gt;&gt; 1</td>
<td>{</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>if (p &amp; 0x1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>result *= x;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>x = x*x;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>
“For” → “While” → “Do-While”

**For Version**

```c
for (Init; Test; Update )
    Body
```

**While Version**

```c
Init;
while (Test ) {
    Body
    Update ;
}
```

**Goto Version**

```c
Init;
    if (!Test)
        goto done;
loop:
    Body
    Update ;
    if (Test)
        goto loop;
done:
```

**Do-While Version**

```c
Init;
    if (!Test)
        goto done;
    do {
        Body
        Update ;
    } while (Test)
done:
```
For-Loop: Compilation #1

For Version

```c
for (Init; Test; Update )
    Body
```

Goto Version

```c
Init;
    if (!Test)
        goto done;

loop:
    Body
    Update ;
    if (Test)
        goto loop;

done:
```

```c
result = 1;
if (p != 0; p = p>>1)
{
    if (p & 0x1)
        result *= x;
    x = x*x;
}
```

```c
result = 1;
if (p == 0)
    goto done;

loop:
    if (p & 0x1)
        result *= x;
    x = x*x;
    p = p >> 1;
    if (p != 0)
        goto loop;

done:
```
“For” → “While” (Jump-to-Middle)

For Version

\[
\text{for} \ (Init; \ Test; \ Update) \\
\quad \text{Body}
\]

While Version

\[
Init; \\
\text{while} \ (Test) \ { \\
\quad \text{Body} \\
\quad Update; \\
}\}
\]

Goto Version

\[
Init; \\
goto \ middle; \\
\text{loop:} \\
\quad \text{Body} \\
\quad Update; \\
middle: \\
\quad \text{if} \ (Test) \\
\quad \quad \text{goto} \ loop; \\
\text{done:}
\]
For-Loop: Compilation #2

For Version

```c
for (Init; Test; Update )
    Body
```

Goto Version

```c
Init;
goto middle;
loop:
    Body
    Update ;
middle:
    if (Test)
        goto loop;
done:
```

```c
for (result = 1; p != 0; p = p>>1)
{
    if (p & 0x1)
        result *= x;
    x = x*x;
}
```

```c
result = 1;
goto middle;
loop:
    if (p & 0x1)
        result *= x;
    x = x*x;
p = p >> 1;
middle:
    if (p != 0)
        goto loop;
done:
```
Switch Statement Example

- **Multiple case labels**
  - Here: 5, 6

- **Fall through cases**
  - Here: 2

- **Missing cases**
  - Here: 4

```c
long switch_eg (long x, long y, long z) {
    long w = 1;
    switch(x) {
        case 1:
            w = y*z;
            break;
        case 2:
            w = y/z;
            /* Fall Through */
        case 3:
            w += z;
            break;
        case 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;
    }
    return w;
}
```
Jump Table Structure

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
  • • •
  case val_{n-1}:
    Block n-1
}
```

**Jump Table**

- `jtab: Targ0`
- `Targ1`
- `Targ2`
- `Targ_{n-1}`

**Jump Targets**

- `Targ0: Code Block 0`
- `Targ1: Code Block 1`
- `Targ2: Code Block 2`
- `Targ_{n-1}: Code Block n-1`

**Approximate Translation**

```
target = JTab[x];
goto *target;
```
Switch Statement Example (IA32)

```c
long switch_eg(long x, long y, long z) {
  long w = 1;
  switch (x) {
    ... 
  }
  return w;
}
```

Setup: switch_eg:
```
pushl %ebp  # Setup
  movl %esp, %ebp  # Setup
  pushl %ebx  # Setup
  movl $1, %ebx
  movl 8(%ebp), %edx
  movl 16(%ebp), %ecx
  cmpl $6, %edx
  ja .L61
  jmp *.L62(,%edx,4)
  ....
```

Translation?
Switch Statement Example (IA32)

```c
long switch_eg(long x, long y, long z) {
    long w = 1;
    switch (x) {
        ...
    }
    return w;
}
```

**Setup:** switch_eg:

- `pushl %ebp`  # Setup
- `movl %esp, %ebp`  # Setup
- `pushl %ebx`  # Setup
- `movl $1, %ebx`  # w = 1
- `movl 8(%ebp), %edx`  # edx = x
- `movl 16(%ebp), %ecx`  # ecx = z
- `cmpl $6, %edx`  # x:6
- `ja .L61`  # if > goto default
- `jmp *.L62(,%edx,4)`  # goto JTab[x]

**Jump table**

```assembly
.section .rodata
.align 4
.L62:
.long .L61 # x = 0
.long .L56 # x = 1
.long .L57 # x = 2
.long .L58 # x = 3
.long .L61 # x = 4
.long .L60 # x = 5
.long .L60 # x = 6
```

**Indirect jump**
Assembly Setup Explanation

- **Table Structure**
  - Each target requires 4 bytes
  - Base address at .L62

- **Jumping**
  - **Direct:** `jmp .L61`
  - Jump target is denoted by label .L61

  - **Indirect:** `jmp * .L62(, %edx, 4)`
    - Start of jump table: .L62
    - Must scale by factor of 4 (labels have 32-bit = 4 Bytes on IA32)
    - Fetch target from effective Address .L62 + edx*4
      - Only for $0 \leq x \leq 6$

```
.L62:
.long  .L61  # x = 0
.long  .L56  # x = 1
.long  .L57  # x = 2
.long  .L58  # x = 3
.long  .L61  # x = 4
.long  .L60  # x = 5
.long  .L60  # x = 6
```
Jump Table

Jump table

```
.section .rodata
.align 4
.L62:
.long .L61  # x = 0
.long .L56  # x = 1
.long .L57  # x = 2
.long .L58  # x = 3
.long .L61  # x = 4
.long .L60  # x = 5
.long .L60  # x = 6
```

```
switch(x) {
    case 1:     // .L56
        w = y*z;
        break;
    case 2:     // .L57
        w = y/z;
        /* Fall Through */
    case 3:     // .L58
        w += z;
        break;
    case 5:
    case 6:     // .L60
        w -= z;
        break;
    default:    // .L61
        w = 2;
}
```
switch(x) {
  . . .
  case 2:    // .L57
    w = y/z;
    /* Fall Through */
  case 3:    // .L58
    w += z;
    break;
  . . .
  default:   // .L61
    w = 2;
}

.L61:    // Default case
    movl  $2, %ebx    # w = 2
    movl  %ebx, %eax  # Return w
    popl  %ebx
    leave
    ret

.L57:    // Case 2:
    movl  12(%ebp), %eax  # y
    cltd             # Div prep
    idivl %ecx       # y/z
    movl  %eax, %ebx # w = y/z
    # Fall through
.L58:    // Case 3:
    addl  %ecx, %ebx # w+= z
    movl  %ebx, %eax # Return w
    popl  %ebx
    leave
    ret
Code Blocks (Rest)

```c
switch(x) {
    case 1:     // .L56
        w = y*z;
        break;
    . . .
    case 5:
    case 6:    // .L60
        w -= z;
        break;
    . . .
}
```

```assembly
.L60: // Cases 5&6:
    subl %ecx, %ebx  # w -= z
    movl %ebx, %eax  # Return w
    popl %ebx
    leave
    ret

.L56: // Case 1:
    movl 12(%ebp), %ebx  # w = y
    imull %ecx, %ebx     # w*= z
    movl %ebx, %eax     # Return w
    popl %ebx
    leave
    ret
```
IA32 Object Code

Setup

- Label `.L61` becomes address 0x08048630
- Label `.L62` becomes address 0x080488dc

Assembly Code

```
switch_eg:
    .
    ja   .L61              # if > goto default
    jmp  *.*L62(,%edx,4)  # goto JTab[x]
```

Disassembled Object Code

```
08048610 <switch_eg>:
    .
08048622:  77 0c      ja  8048630
08048624: ff 24 95 dc 88 04 08 jmp *0x80488dc(,%edx,4)
```
IA32 Object Code (cont.)

- Jump Table
  - Doesn’t show up in disassembled code
  - Can inspect using GDB

```
gdb asm-cntl
(gdb) x/7xw 0x080488dc
```
  - Examine 7 hexadecimal format “words” (4-bytes each)
  - Use command “help x” to get format documentation

```
0x080488dc:
  0x08048630
  0x08048650
  0x0804863a
  0x08048642
  0x08048630
  0x08048649
  0x08048649
```
## Disassembled Targets

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048630</td>
<td>bb 02 00 00 00</td>
<td>mov $0x2,%ebx</td>
</tr>
<tr>
<td>8048635</td>
<td>89 d8</td>
<td>mov %ebx,%eax</td>
</tr>
<tr>
<td>8048637</td>
<td>5b</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>8048638</td>
<td>c9</td>
<td>leave</td>
</tr>
<tr>
<td>8048639</td>
<td>c3</td>
<td>ret</td>
</tr>
<tr>
<td>804863a</td>
<td>8b 45 0c</td>
<td>mov 0xc(%ebp),%eax</td>
</tr>
<tr>
<td>804863d</td>
<td>99</td>
<td>cltd</td>
</tr>
<tr>
<td>804863e</td>
<td>f7 f9</td>
<td>idiv %ecx</td>
</tr>
<tr>
<td>8048640</td>
<td>89 c3</td>
<td>mov %eax,%ebx</td>
</tr>
<tr>
<td>8048642</td>
<td>01 cb</td>
<td>add %ecx,%ebx</td>
</tr>
<tr>
<td>8048644</td>
<td>89 d8</td>
<td>mov %ebx,%eax</td>
</tr>
<tr>
<td>8048646</td>
<td>5b</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>8048647</td>
<td>c9</td>
<td>leave</td>
</tr>
<tr>
<td>8048648</td>
<td>c3</td>
<td>ret</td>
</tr>
<tr>
<td>8048649</td>
<td>29 cb</td>
<td>sub %ecx,%ebx</td>
</tr>
<tr>
<td>804864b</td>
<td>89 d8</td>
<td>mov %ebx,%eax</td>
</tr>
<tr>
<td>804864d</td>
<td>5b</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>804864e</td>
<td>c9</td>
<td>leave</td>
</tr>
<tr>
<td>804864f</td>
<td>c3</td>
<td>ret</td>
</tr>
<tr>
<td>8048650</td>
<td>8b 5d 0c</td>
<td>mov 0xc(%ebp),%ebx</td>
</tr>
<tr>
<td>8048653</td>
<td>0f af d9</td>
<td>imul %ecx,%ebx</td>
</tr>
<tr>
<td>8048656</td>
<td>89 d8</td>
<td>mov %ebx,%eax</td>
</tr>
<tr>
<td>8048658</td>
<td>5b</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>8048659</td>
<td>c9</td>
<td>leave</td>
</tr>
<tr>
<td>804865a</td>
<td>c3</td>
<td>ret</td>
</tr>
</tbody>
</table>
Matching Disassembled Targets

8048630: bb 02 00 00 00 mov
8048635: 89 d8 mov
8048637: 5b pop
8048638: c9 leave
8048639: c3 ret
804863a: 8b 45 0c mov
804863d: 99 cltd
804863e: f7 f9 div
8048640: 89 c3 mov
8048642: 01 cb add
8048644: 89 d8 mov
8048646: 5b pop
8048647: c9 leave
8048648: c3 ret
8048649: 29 cb sub
804864b: 89 d8 mov
804864d: 5b pop
804864e: c9 leave
804864f: c3 ret
8048650: 8b 5d 0c mov
8048653: 0f af d9 imul
8048656: 89 d8 mov
8048658: 5b pop
8048659: c9 leave
804865a: c3 ret
Summarizing

- **C Control**
  - if-then-else
  - do-while
  - while, for
  - switch

- **Assembler Control**
  - Conditional jump
  - Conditional move
  - Indirect jump
  - Compiler
  - Must generate assembly code to implement more complex control

- **Standard Techniques**
  - Loops converted to do-while form
  - Large switch statements use jump tables
  - Sparse switch statements may use decision trees (see text)

- **Conditions in CISC**
  - CISC machines generally have condition code registers