Machine Programming II: 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
    - %reg = Mem[address]
  - Store register data into memory
    - Mem[address] = %reg

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

---

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`)
### 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>temp = 0x4;</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>movl $-147,(%eax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td>Reg</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>Mem</td>
<td>Reg</td>
<td>movl (%eax),%edx</td>
<td>temp = *p;</td>
</tr>
</tbody>
</table>

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

How do you copy from a memory location to another then?
Memory vs. registers

- Why both?
- Performance?
- Usage difference?

Simple Memory Addressing Modes

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

- Displacement D(R) \text{ Mem[Reg[R]+D]}
  - Register R specifies start of memory region
  - Constant displacement D specifies offset
  
  \texttt{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;
}
```

Set Up

```assembly
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)
```

Body

```assembly
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Finish

Understanding Swap

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

Stack (in memory)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>%ebp</td>
</tr>
<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>

<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>

```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

Address

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>123 0x124</td>
</tr>
<tr>
<td></td>
<td>456 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>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>

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>4</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
Understanding Swap

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

Address

| 123 | 0x124 |
| 456 | 0x120 |
| 0x11c | |
| 0x118 | |
| 0x114 | |
| 0x110 | |
| 0x10c | |
| 0x108 | |
| 0x104 | |
| 0x100 | |

Offset

| yp | 12 |
| xp | 8 |
| %ebp | 0 |
| Rtn adr | |

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,(%eax)  # *yp = ebx
Understanding Swap

<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>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>

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 |

Address

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>456</td>
<td>0x124</td>
</tr>
<tr>
<td>123</td>
<td>0x120</td>
</tr>
<tr>
<td>0x11c</td>
<td>4</td>
</tr>
<tr>
<td>0x118</td>
<td>8</td>
</tr>
<tr>
<td>0x114</td>
<td>12</td>
</tr>
<tr>
<td>0x110</td>
<td>0x124</td>
</tr>
<tr>
<td>0x10c</td>
<td>0x114</td>
</tr>
<tr>
<td>0x108</td>
<td>0x120</td>
</tr>
<tr>
<td>0x104</td>
<td>0x108</td>
</tr>
<tr>
<td>0x100</td>
<td>0</td>
</tr>
</tbody>
</table>

Rtn adr

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

x86-64 Integer Registers

| %rax | %eax |
| %rbx | %ebx |
| %rcx | %ecx |
| %rdx | %edx |
| %rsi | %esi |
| %rdi | %edi |
| %rsp | %esp |
| %rbp | %ebp |
| %r8  | %r8d |
| %r9  | %r9d |
| %r10 | %r10d|
| %r11 | %r11d|
| %r12 | %r12d|
| %r13 | %r13d|
| %r14 | %r14d|
| %r15 | %r15d|

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

- Long word $l$ (4 Bytes) $\leftrightarrow$ Quad word $q$ (8 Bytes)

- New instructions:
  - `movl` $\rightarrow$ `movq`
  - `addl` $\rightarrow$ `addq`
  - `sall` $\rightarrow$ `salq`
  - etc.

- 32-bit instructions generate 32-bit results,
  - What about the other 32 bits in the register?
  - Set higher order bits of destination register to 0
  - Example: `addl`

Swap in 32-bit Mode

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

swap:
```assembly
    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
```

Swap:
- Setup
- Body
- Finish
Swap in 64-bit Mode

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

- **Operands passed in registers (why useful?)**
  - First (xp) in %rdi, second (yp) in %rsi
  - 64-bit pointers
- **No stack operations required**
- **32-bit data**
  - Data held in registers %eax and %edx
  - `movl` operation

Swap Long Ints in 64-bit Mode

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

- **64-bit data**
  - Data held in registers %rax and %rdx
  - `movq` operation
  - “q” stands for quad-word
Complete Memory Addressing Modes

**Most General Form**

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

- **D**: Constant “displacement” 1, 2, or 4 bytes
- **R_b**: Base register: Any of 8 integer registers
- **R_i**: Index register: Any, except for %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}[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 \cdot 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>Mem[Reg[Rb]] + 0x80</td>
<td>0x80(,%edx,2)</td>
</tr>
<tr>
<td>(%edx,%ecx)</td>
<td>Mem[Reg[Rb]] + Reg[Ri]]</td>
<td>(Rb,Ri)</td>
</tr>
<tr>
<td>(%edx,%ecx,4)</td>
<td>Mem[Reg[Rb]] + 0x80</td>
<td>0x80(,%edx,4)</td>
</tr>
<tr>
<td>0x80(,%edx,2)</td>
<td>Mem[Reg[Rb]] + 0x80</td>
<td>0x80(,%edx,2)</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>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> \ (Also called <code>shll</code>)</td>
</tr>
<tr>
<td><code>sarl</code></td>
<td><code>Dest = Dest &gt;&gt; Src</code> \ (Arithmetic)</td>
</tr>
<tr>
<td><code>shrl</code></td>
<td><code>Dest = Dest &gt;&gt; Src</code> \ (Logical)</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>

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 \( Dest = -Dest \)
- 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
lea1 %ebp,%esp
movl %esp,%ebp

set up
pushl %ebp
movl %esp,%ebp
movl 8(%ebp),%eax
movl 12(%ebp),%edx
lea1 (%edx,%eax),%ecx
lea1 (%edx,%edx,2),%edx
sall $4,%edx
addl 16(%ebp),%ecx
lea1 4(%edx,%eax),%eax
imull %ecx,%eax
movl %ebp,%esp
popl %ebp
ret

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;
}
```

The function `arith` takes three integers `x`, `y`, and `z` as input and returns a new integer calculated based on these inputs. The calculation involves addition and multiplication operations.

### Instructions Interpretation

The machine code for the function is as follows:

```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
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?

1. `movl 8(%ebp),%eax`: Move the value at the second 8 bytes from the base pointer to `eax`.
2. `movl 12(%ebp),%edx`: Move the value at the second 12 bytes from the base pointer to `edx`.
3. `leal (%edx,%eax),%ecx`: Load a long expression defined by the displacement from `ecx`.
4. `leal (%edx,%edx,2),%edx`: Load a long expression defined by the displacement from `edx`.
5. `sall $4,%edx`: Shift left by 4.
6. `addl 16(%ebp),%ecx`: Add 16 bytes from the base pointer to `ecx`.
7. `leal 4(%edx,%eax),%eax`: Load a long expression defined by the displacement from `eax`.
8. `imull %ecx,%eax`: Multiply `eax` by `ecx`.

The final result is stored in `eax`, which is returned by the function.

This code snippet demonstrates how to perform arithmetic operations using registers and memory locations in assembly language, and how these operations are translated into machine instructions for execution.
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;
}
```

### 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;
}
```
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`  
  ```
  Set Up
  ```
- `movl 8(%ebp),%eax`  
  - `xorl 12(%ebp),%eax`  
  - `sarl $17,%eax`  
  - `andl $8185,%eax`  
  ```
  Body
  ```
- `movl %ebp,%esp`  
  ```
  Finish
  ```
- `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;
}
```

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

```assembly
logical:
    pushl %ebp
    movl %esp,%ebp
    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
    movl %ebp,%esp
    popl %ebp
    ret
```

Control-Flow/Conditionals

- **Unconditional**

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

- **Conditional**

  ```c
  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; iterative) {...}

- 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

- Current stack top
  - %esp

- Current stack frame
  - %ebp

- Instruction pointer
  - %eip

- Condition codes
  - CF
  - ZF
  - SF
  - OF

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
    \[(a>0 \&\& b>0 \&\& t<0) \text{ or } (a<0 \&\& b<0 \&\& t>=0)\]

- Not set by lea instruction (beware!)

- Full documentation (IA32) http://www.jegerlehner.ch/intel/IntelCodeTable.pdf
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
  
  \[
  (a>0 \&\& b<0 \&\& (a-b)<0) \text{ } || \text{ } (a<0 \&\& b>0 \&\& (a-b)>0)
  \]

Condition Codes (Explicit Setting: Test)

- **Explicit Setting by Test instruction**
  
  \[
  \text{testl/testq } \text{Src2,Src1} \\
  \text{testl } b,a \text{ like computing } a\&b \text{ without setting destination}
  \]

- Sets condition codes based on value of \( \text{Src1} \& \text{Src2} \)
- 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>setsi</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)&amp;ZF</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>

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
```

What does each of these instructions do?
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 movzb to finish job

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

**Body**

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

**Jumping**

- **jX Instructions**
  - Jump to different part of code depending on condition codes

<table>
<thead>
<tr>
<th>jX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>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 0(%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

```assembly
absdiff:
    pushl %ebp
    movl %esp, %ebp
    movl 0(%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
```
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
    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
    movl %edx, %eax
.L8:
    leave
    ret
.L7:
    subl %edx, %eax
    jmp .L8
```
General Conditional Expression Translation

C Code

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

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

- \text{Test} is expression returning integer
  - = 0 interpreted as false
  - \neq 0 interpreted as true
- Create separate code regions for then & else expressions
- Execute appropriate one
- How would you make this efficient?

Goto Version

\[
\text{nt} = \neg \text{Test};
\]

\[
\text{if} (\text{nt}) \text{ goto Else;}
\]

\[
\text{val} = \text{Then-Expr};
\]

\[
\text{Done:}
\]

\[
\ldots
\]

\[
\text{Else:}
\]

\[
\text{val} = \text{Else-Expr};
\]

\[
\text{goto Done;}
\]

Conditionals: x86-64

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

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

- PC relative branches are relocatable
- Absolute branches are not

Machine code:

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 straightforward
  - The only slightly tricky part is to be sure where the conditional branch occurs: top or bottom of the loop
- Q: How is for(i=0; i<100; i++) 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

```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:

<table>
<thead>
<tr>
<th>Register</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>x</td>
</tr>
<tr>
<td>%eax</td>
<td>result</td>
</tr>
</tbody>
</table>

Translation?
General “Do-While” Translation

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

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

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

- **Test** returns integer
- 0 interpreted as false
- ≠0 interpreted as true

---

“While” Loop Translation

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

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

- Used by GCC for both IA32 & x86-64
- First iteration jumps over body computation within loop straight to test
“While” Loop Example

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

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, ...
  - movq -> “move quad word” or 4*16-bits
- 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);
  ```
- Goto Version
  ```c
  loop:
  if (!Test)
    goto done;
  loop:
  Body
  if (Test)
    goto loop;
  done:
```
**“For” Loop Example: Square-and-Multiply**

```c
/* 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: \( x^p = z_0 \cdot z_1^2 \cdot (z_2^2)^2 \cdot \ldots \cdot ((z_{n-1}^2)^2)^2 \)
  - \( z_i = 1 \) when \( p_i = 0 \)
  - \( z_i = x \) when \( p_i = 1 \)
- Complexity \( O(\log p) \)

**Example**
- \( 3^{10} = 3^2 \cdot 3^8 \)
  - \( = 3^2 \cdot ((3^2)^2)^2 \)

---

**ipwr Computation**

```c
/* 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=1010_2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>9</td>
<td>5= 101_2</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>81</td>
<td>2= 10_2</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>6561</td>
<td>1= 1_2</td>
</tr>
<tr>
<td>5</td>
<td>59049</td>
<td>43046721</td>
<td>0_2</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>
</table>
| p != 0    | result = 1 | p = p >> 1 | {
  if (p & 0x1)
      result *= x;
  x = x*x;
} |

“For”→“While”

For Version

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

While Version

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

Goto Version

```c
Init;
goto middle;
loop:
    Body
    Update ;
middle:
    if (Test)
        goto loop;
done:
```
For-Loop: Compilation

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:
```
Jump Table Structure

Switch Form

```
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
    ...
    Targn-1
```

Jump Targets

```
Targ0: Code Block 0
Targ1: Code Block 1
Targ2: Code Block 2
    ...
Targn-1: Code Block n-1
```

Approximate Translation

```
target = JTab[x];
goto *target;
```

Jump Table Structure

C code:

```
switch(x) {
    case 0: <some code>
        break;
    case 1: <some code>
        break;
    case 6: <some code>
        break;
    default: <some code>
        break;
}
```

We can use the jump table when x <= 6:

```
if (x <= 6)
    target = JTab[x];
    goto *target;
else
    goto default;
```
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:      // .L60
    case 6:      // .L60
        w -= z;
        break;
    default:     // .L61
        w = 2;
}
```

Switch Statement Example (IA32)

```
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;
}
```

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)
```
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]`

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 are 32-bits = 4 bytes on IA32)
  - Fetch target from effective Address `.L62 + edx*4`
    - Only for `0 ≤ x ≤ 6`
Jump Table

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
```

```assembly
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;
}
```

Code Blocks (Partial)

```assembly
switch(x) {
  ...
  .case 2:      // .L57
    w = y/z;
    /* Fall Through */
  .case 3:      // .L58
    w += z;
    break;
  ...
  .default:     // .L61
    w = 2;
}
```

```assembly
.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;
    . . .
}
```

```asm
.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

```
8048630:   bb 02 00 00 00          mov   $0x2,%ebx
8048633:   89 d8                   mov   %ebx,%eax
8048637:   5b                      pop   %ebx
8048638:   c9                      leave
8048639:   c3                      ret
804863a:   8b 45 0c                mov   0xc(%ebp),%eax
804863d:   99                      cltd
804863e:   f7 f9                   idiv   %ecx
8048640:   89 c3                   mov   %eax,%ebx
8048642:   01 cb                   add   %ecx,%ebx
8048644:   89 d8                   mov   %ebx,%eax
8048646:   5b                      pop   %ebx
8048647:   c9                      leave
8048648:   c3                      ret
8048649:   29 cb                   sub   %ecx,%ebx
804864b:   89 d8                   mov   %ebx,%eax
804864d:   5b                      pop   %ebx
804864e:   c9                      leave
804864f:   c3                      ret
8048650:   8b 5d 0c                mov   0xc(%ebp),%ebx
8048653:   0f 48 f9               imul   %ecx,%ebx
8048656:   89 d8                   mov   %ebx,%eax
8048658:   5b                      pop   %ebx
8048659:   c9                      leave
804865a:   c3                      ret
```
Matching Disassembled Targets

0x08048630
0x08048650
0x0804863a
0x08048642
0x08048630
0x08048649
0x08048649

Question

Would you implement this with a jump table?

```
switch(x) {
    case 0: <some code> break;
    case 10: <some code> break;
    case 52000: <some code> break;
    default: <some code> break;
}
```

Probably not:
- Don’t want a jump table with 52000 entries (too big)
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