Computer Systems
CSE 410 Autumn 2013
6 – x86 Assembly Programming
Roadmap

C:
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);

Java:
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
c.getMPG();

Assembly language:
get_mpg:
pushq %rbp
movq %rsp, %rbp
...
popq %rbp
ret

Machine code:
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111

Computer system:

Memory & data
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C
x86 Assembly Programming

- Move instructions, registers, and operands
- Memory addressing modes
- `swap` example: 32-bit vs. 64-bit
- Arithmetic operations
- Condition codes
- Conditional and unconditional branches
- Loops
- Switch statements
Three Basic Kinds of Instructions

- **Transfer data between memory and register**
  - *Load* data from memory into register
    - %reg = Mem[address]
  - *Store* register data into memory
    - Mem[address] = %reg

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

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

Remember: memory is indexed just like an array[]!
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)`
    - 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><strong>Imm</strong></td>
<td>Reg</td>
<td>movl $0x4,%eax</td>
<td>( \text{var}_a = 0x4; )</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>movl $-147,(%eax)</td>
<td>( *p_a = -147; )</td>
</tr>
<tr>
<td><strong>Reg</strong></td>
<td>Reg</td>
<td>movl %eax,%edx</td>
<td>( \text{var}_d = \text{var}_a; )</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>movl %eax,(%edx)</td>
<td>( *p_d = \text{var}_a; )</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl (%eax),%edx</td>
<td>( \text{var}_d = *p_a; )</td>
</tr>
</tbody>
</table>

*Cannot do memory-memory transfer with a single instruction.*
Memory Addressing Modes: Basic

- **Indirect**  (R)  \(\text{Mem}[\text{Reg}[R]]\)
  - Register R specifies the memory address

  \texttt{movl \texttt{(\texttt{\%ecx}),\texttt{\%eax}}}  \\

- **Displacement**  D(R)  \(\text{Mem}[\text{Reg}[R]+D]\)
  - Register R specifies a memory address
    - (e.g. the start of some memory region)
  - Constant displacement D specifies the offset from that address

  \texttt{movl \texttt{8(\texttt{\%ebp}),\texttt{\%edx}}}
Using Basic Addressing Modes

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

swap:

    pushl %ebp
    movl %esp,%ebp
    pushl %ebx

    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)

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

Set Up

Body

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

### Register Value

<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 Code

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

x86
Understanding Swap

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

<table>
<thead>
<tr>
<th>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>0x110</td>
</tr>
<tr>
<td>12</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

{%eax %edx %ecx %ebx %esi %edi %esp %ebp

<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>0x120</td>
<td>0x120</td>
<td>0x120</td>
<td>0x120</td>
<td>0x120</td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
\text{movl} &\ 12(\%ebp),\%ecx & \quad \# \ \text{ecx} = \text{yp} \\
\text{movl} &\ 8(\%ebp),\%edx & \quad \# \ \text{edx} = \text{xp} \\
\text{movl} &\ (%ecx),\%eax & \quad \# \ \text{eax} = *\text{yp} \ (t1) \\
\text{movl} &\ (%edx),\%ebx & \quad \# \ \text{ebx} = *\text{xp} \ (t0) \\
\text{movl} &\ %eax,(%edx) & \quad \# \ *\text{xp} = \text{eax} \\
\text{movl} &\ %ebx,(%ecx) & \quad \# \ *\text{yp} = \text{ebx}
\end{align*}
\]
# Understanding Swap

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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp</td>
<td>12</td>
</tr>
<tr>
<td>xp</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td>%ebp</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>-4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>0x124</td>
<td></td>
<td></td>
</tr>
<tr>
<td>456</td>
<td>0x120</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x11c</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x118</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x114</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x110</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x124</td>
<td></td>
<td></td>
</tr>
<tr>
<td>456</td>
<td>0x10c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rtn adr</td>
<td>0x108</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x104</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x100</td>
<td></td>
<td></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>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>456</td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td>123</td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

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

- `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

### x86 Architecture

<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>

### Memory Locations

<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></td>
</tr>
<tr>
<td>-4</td>
<td></td>
</tr>
</tbody>
</table>

### Assembly Code

```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>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>

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

<table>
<thead>
<tr>
<th><code>%rax</code></th>
<th><code>%eax</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rbx</code></td>
<td><code>%ebx</code></td>
</tr>
<tr>
<td><code>%rcx</code></td>
<td><code>%ecx</code></td>
</tr>
<tr>
<td><code>%rdx</code></td>
<td><code>%edx</code></td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td><code>%esi</code></td>
</tr>
<tr>
<td><code>%rdi</code></td>
<td><code>%edi</code></td>
</tr>
<tr>
<td><code>%rsp</code></td>
<td><code>%esp</code></td>
</tr>
<tr>
<td><code>%rbp</code></td>
<td><code>%ebp</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>%r8</code></th>
<th><code>%r8d</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%r9</code></td>
<td><code>%r9d</code></td>
</tr>
<tr>
<td><code>%r10</code></td>
<td><code>%r10d</code></td>
</tr>
<tr>
<td><code>%r11</code></td>
<td><code>%r11d</code></td>
</tr>
<tr>
<td><code>%r12</code></td>
<td><code>%r12d</code></td>
</tr>
<tr>
<td><code>%r13</code></td>
<td><code>%r13d</code></td>
</tr>
<tr>
<td><code>%r14</code></td>
<td><code>%r14d</code></td>
</tr>
<tr>
<td><code>%r15</code></td>
<td><code>%r15d</code></td>
</tr>
</tbody>
</table>

- Extend existing registers, and add 8 new ones; *all* accessible as 8, 16, 32, 64 bits.
32-bit vs. 64-bit operands

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

- New instruction forms:
  - $\text{movl} \rightarrow \text{movq}$
  - $\text{addl} \rightarrow \text{addq}$
  - $\text{sall} \rightarrow \text{salq}$
  - etc.

- x86-64 can still use 32-bit instructions that generate 32-bit results
  - Higher-order bits of destination register are just set to 0
  - Example: $\text{addl}$
Swap Ints 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 Ints in 64-bit Mode

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

Arguments 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 (the 1 refers to data width, not address width)
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;
}
```

![Code snippet for swap_l function in x86 assembly](image)

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

- Remember, the addresses used for accessing memory in `mov` (and other) instructions can be computed in several different ways.

**Most General Form:**

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

- **D:** Constant “displacement” 1, 2, or 4 bytes
- **R_b:** Base register: Any of the 8/16 integer registers
- **R_i:** Index register: Any, except for `%esp` or `%rsp`
  - Unlikely you’d use `%ebp`, either
- **S:** Scale: 1, 2, 4, or 8 (*why these numbers?*)

**Special Cases:** can use any combination of **D**, **R_b**, **R_i** and **S**

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

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8 (%edx)</td>
<td>0xf000 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>(%edx,%ecx)</td>
<td>0xf000 + 0x100</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%edx,%ecx,4)</td>
<td>0xf000 + 4*0x100</td>
<td>0xf400</td>
</tr>
<tr>
<td>0x80(,%edx,2)</td>
<td>2*0xf000 + 0x80</td>
<td>0x1e080</td>
</tr>
</tbody>
</table>
Address Computation Instruction

- **leal Src, Dest**
  - *Src* is address mode expression
  - Set *Dest* to address computed by expression
    - (lea stands for *load effective address*)
  - Example: `leal (%edx,%ecx,4), %eax`

- **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 (Binary) Instructions:

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>addl</td>
<td>Dest = Dest + Src</td>
</tr>
<tr>
<td>subl</td>
<td>Dest = Dest - Src</td>
</tr>
<tr>
<td>imull</td>
<td>Dest = Dest * Src</td>
</tr>
<tr>
<td>sall</td>
<td>Dest = Dest &lt;&lt; Src</td>
</tr>
<tr>
<td>sarl</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>shr1</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>xorl</td>
<td>Dest = Dest ^ Src</td>
</tr>
<tr>
<td>andl</td>
<td>Dest = Dest &amp; Src</td>
</tr>
<tr>
<td>orl</td>
<td>Dest = Dest</td>
</tr>
</tbody>
</table>

- Also called shll
- Arithmetic
- Logical

- Watch out for argument order! (especially subl)
- No distinction between signed and unsigned int (why?)
Some Arithmetic Operations

- **One Operand (Unary) Instructions**
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>incl</td>
<td>Dest</td>
<td>Dest = Dest + 1</td>
</tr>
<tr>
<td>decl</td>
<td>Dest</td>
<td>Dest = Dest - 1</td>
</tr>
<tr>
<td>negl</td>
<td>Dest</td>
<td>Dest = ~Dest</td>
</tr>
<tr>
<td>notl</td>
<td>Dest</td>
<td>Dest = ~Dest</td>
</tr>
</tbody>
</table>

- See textbook section 3.5.5 for more instructions: mul, cltd, idivl, divl
Using `leal` for Arithmetic Expressions

```c
int arith(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

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

    movl 8(%ebp),%eax
    movl 12(%ebp),%edx
    leal (%edx,%eax),%ecx
    leal (%edx,%edx,2),%edx
    sall $4,%edx
    addl 16(%ebp),%ecx
    leal 4(%edx,%eax),%eax
    imull %ecx,%eax

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

```c
int arith
  (int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

```assembly
movl 8(%ebp),%eax          # eax = x
movl 12(%ebp),%edx         # edx = y
leal (%edx,%eax),%ecx      # ecx = x+y  (t1)
leal (%edx,%edx,2),%edx     # edx = y + 2*y = 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;
}
```

The corresponding assembly code is:

```
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 = y + 2*y = 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 = y + 2*y = 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 = y + 2*y = 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)
```
Observations about \texttt{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;
}
```

- Instructions in different order from C code
- Some expressions require multiple instructions
- Some instructions cover multiple expressions
- Get exact same code if we compile \((x+y+z) * (x+4+48*y)\)

x86 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 = y + 2*y = 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:****

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

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

**Set Up**

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

**Body**

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

**Finish**

```
    movl 8(%ebp),%eax    ; eax = x
    xorl 12(%ebp),%eax   ; eax = x^y (t1)
    sarl $17,%eax        ; eax = t1>>17 (t2)
    andl $8185,%eax      ; eax = t2 & 8185
```
Another Example

```c
int logical(int x, int y) {
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

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

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

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

- **Finish**

**x86**

- `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;
}
```

\[ 2^{13} = 8192, \quad 2^{13} - 7 = 8185 \]

\[ \ldots 0010000000000000, \ldots 0001111111111001 \]

**logical:**

```
    pushl %ebp
    movl %esp,%ebp
    ... Set Up ...
    movl 8(%ebp),%eax  \( eax = x \)
    xorl 12(%ebp),%eax  \( eax = x^y \)  (t1)
    sarl $17,%eax  \( eax = t1\gg\!17 \)  (t2)
    andl $8185,%eax  \( eax = t2 \& \ 8185 \)
    ... Body ...
    popl %ebp
    ret          ...
    ... Finish ...
```
Conditionals and Control Flow

- A 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 implement some related control flow constructs
  - break, continue

- In x86, we’ll refer to branches as “jumps” (either conditional or unconditional)
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>ZF</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 (%esp)
- Current stack frame (%ebp)
- Instruction pointer (%eip)
- Condition codes

x86
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
      \((a>0 \&\& b>0 \&\& t<0) \lor (a<0 \&\& b<0 \&\& t>=0)\)

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

- **Full documentation (IA32):** [http://www.jegerlehner.ch/intel/IntelCodeTable.pdf](http://www.jegerlehner.ch/intel/IntelCodeTable.pdf)
Condition Codes (Explicit Setting: Compare)

■ Single-bit registers
  - CF Carry Flag (for unsigned)
  - SF Sign Flag (for signed)
  - ZF Zero Flag
  - OF Overflow Flag (for signed)

■ Explicit Setting by Compare Instruction

  - `cmp l/cmp q Src2,Src1`
  - `cmp l 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
    
    \[(a>0 \land \land b<0 \land \land (a-b)<0) \lor (a<0 \land \land b>0 \land \land (a-b)>0)\]
Condition Codes (Explicit Setting: Test)

- Single-bit registers
  - CF  Carry Flag (for unsigned)
  -ZF  Zero Flag
  -SF  Sign Flag (for signed)
  -OF  Overflow Flag (for signed)

- 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** if `a&b == 0`
      - **SF set** if `a&b < 0`
  - `testl %eax, %eax`
    - Sets SF and ZF, check if eax is +,0,-
Reading Condition Codes

- **SetX Instructions**
  - Set a single byte to 0 or 1 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 to 0 or 1 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:** y at 12(%ebp), x at 8(%ebp)
```
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 to 0 or 1 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:** y at 12(%ebp), x at 8(%ebp)

```
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
```

(x – y)
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
```

---

{x86}
Conditional Branch Example (Cont.)

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

```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

---

C allows “goto” as means of transferring control

- Closer to machine-level programming style
- Generally considered bad coding style
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
    movl    %edx, %eax
    .L8:
    leave
    ret
    .L7:
    subl    %edx, %eax
    jmp     .L8
```

int x  %edx
int y  %eax
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;
}
```

```x86
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
.J8:
    leave
    ret
```

```
int x %edx
int y %eax
```
Conditional Branch Example (Cont.)

```
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
```

```
int x %edx
int y %eax
```
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;
}
```

```assembler
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:
    leave
    ret
.L8:
    leave
    ret
```

```plaintext
int x %edx
int y %eax
```
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
     .L8:
    leave
    ret
    .L7:
    subl %edx, %eax
    jmp .L8
```

int x %edx
int y %eax
General Conditional Expression Translation

C Code

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

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

Goto Version

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

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

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

\[
\text{Done:}
\]

\[
\ldots
\]

\[
\text{Else:}
\]

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

\[
\text{goto Done;}
\]

- \text{Test} is expression returning integer
  - = 0 interpreted as false
  - \neq 0 interpreted as true
- Create separate code regions for then & else expressions
- Execute appropriate one
- How might you make this more efficient?
Conditionals: x86-64

### 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

```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
```
PC Relative Addressing

- **PC relative branches are** **relocatable**
  (same code works no matter where code is stored in memory)

- **Absolute branches are not**
  (actual branch address encoded in instruction)
Compiling Loops

C/Java code:
```c
while ( sum != 0 ) {
  <loop body>
}
```

Machine code:
```assembly
loopTop:   cmpl  $0, %eax
            je    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

- **How would for(i=0; i<100; i++) be implemented?**
“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  # eax = 1
    movl 8(%%ebp),%edx  # edx = x

.L11:
    imull %edx,%eax  # result *= x
    decl %edx  # x--
    cmpl $1,%edx  # Compare x : 1
    jg .L11      # if > goto loop
    movl %ebp,%esp  # Finish
    popl %ebp
    ret
```

Registers:

- `%edx` - x
- `%eax` - result
General “Do-While” Translation

**C Code**

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

**Goto Version**

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

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

- **Test** returns integer
  - = 0 interpreted as false
  - ≠ 0 interpreted as true
“While” Loop 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_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;
}
```

```x86
# x in %edi, result in %eax
jmph .L34 # goto Middle
.L35:  # Loop:
imull %edi, %eax  # result *= x
dcl %edi  # x--
.L34: # Middle:
cmp $1, %edi  # x:1
jgh .L35 # if >, goto Loop
```

"While" Loop Example
“For” Loop Example: Square-and-Multiply

```c
/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned int 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: \( x^p = z_0 \cdot z_1^2 \cdot (z_2^2)^2 \cdot \ldots \cdot (\ldots((z_{n-1}^2)^2)^2) \ldots )^2 \)
  - \( z_i = 1 \) 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

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

| before iteration | result | x=3 | p=10
|------------------|--------|-----|-----
| 1                | 1      | 3   | 10=1010₂
| 2                | 1      | 9   | 5=101₂
| 3                | 9      | 81  | 2=10₂
| 4                | 9      | 6561| 1=1₂
| 5                | 59049  | 43046721 | 0₂
"For" Loop Example

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

General Form

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

<table>
<thead>
<tr>
<th>Init</th>
<th>Test</th>
<th>Update</th>
<th>Body</th>
</tr>
</thead>
<tbody>
<tr>
<td>result = 1</td>
<td>p != 0</td>
<td>p = p &gt;&gt; 1</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"

**For Version**

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

**While Version**

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

**Goto Version**

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

x86
Switch Statement Example

- Multiple case labels
  - Here: 5, 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4

- Lots to manage, we need a *jump table*
Jump Table Structure

**Switch Form**

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

**Jump Table**

```
<table>
<thead>
<tr>
<th>JTab:</th>
<th>Targ0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Targ1</td>
</tr>
<tr>
<td></td>
<td>Targ2</td>
</tr>
<tr>
<td></td>
<td>Targn-1</td>
</tr>
</tbody>
</table>
```

**Jump Targets**

```
<table>
<thead>
<tr>
<th>Targ0:</th>
<th>Code Block 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Targ1:</td>
<td>Code Block 1</td>
</tr>
<tr>
<td>Targ2:</td>
<td>Code Block 2</td>
</tr>
<tr>
<td>Targn-1:</td>
<td>Code Block n-1</td>
</tr>
</tbody>
</table>
```

**Approximate Translation**

```
target = JTab[x];
goto *target;
```
Jump Table Structure

C code:

```c
switch(x) {
    case 1: <some code>
        break;
    case 2: <some code>
    case 3: <some code>
        break;
    case 5:
    case 6: <some code>
        break;
    default: <some code>
}
```

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

```c
if (x <= 6)
    target = JTab[x];
    goto *target;
else
    goto default;
```
Jump Table

Jump table

```
.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;
}
```

.x86
Switch Statement Example (IA32)

```c
long switch_eg(unsigned 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`
- `ja .L61`
- `jmp *.L62(,%edx,4)`

Jump table
```
.sect .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
```

Translation?
Switch Statement Example (IA32)

long switch_eg(unsigned 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
  cmp $6, %edx  # x:6
  ja  .L61  # if > goto default
  jmp  *.L62(%edx,4)  # goto JTab[x]

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
Assembly Setup Explanation

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

Jumping: different address modes for target
- 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
  - target = JTab[x]; goto *target; (only for 0 ≤ x ≤ 6)
Code Blocks (Partial)

```c
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
```

x86
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
```

x86
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
    
    \[ \text{gdb \ asm-cntl} \]

    \[
    \text{(gdb)} \ x/7xw \ 0x080488dc
    \]
    - Examine 7 hexadecimal format “words” (4-bytes each)
    - Use command “\text{help x}” to get format documentation

\[ \text{0x080488dc:} \]

\[
\begin{array}{l}
  0x08048630 \\
  0x08048650 \\
  0x0804863a \\
  0x08048642 \\
  0x08048630 \\
  0x08048649 \\
  0x08048649 \\
\end{array}
\]
## Disassembled Targets

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comment</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 %ebp,%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 %ebp,%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

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08048630</td>
<td>bb 02 00 00 00</td>
<td>mov</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08048635</td>
<td>89 d8</td>
<td>mov</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08048637</td>
<td>5b</td>
<td>pop</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08048638</td>
<td>c9</td>
<td>leave</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08048639</td>
<td>c3</td>
<td>ret</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0804863a</td>
<td>8b 45 0c</td>
<td>mov</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0804863d</td>
<td>99</td>
<td>cltd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0804863e</td>
<td>f7 f9</td>
<td>idiv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08048640</td>
<td>89 c3</td>
<td>mov</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08048642</td>
<td>01 cb</td>
<td>add</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08048644</td>
<td>89 d8</td>
<td>mov</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08048646</td>
<td>5b</td>
<td>pop</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08048647</td>
<td>c9</td>
<td>leave</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08048648</td>
<td>c3</td>
<td>ret</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08048649</td>
<td>29 cb</td>
<td>sub</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08048649</td>
<td>29 cb</td>
<td>sub</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x080486a6</td>
<td>8b 5d 0c</td>
<td>mov</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08048653</td>
<td>0f af d9</td>
<td>imul</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08048656</td>
<td>89 d8</td>
<td>mov</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08048658</td>
<td>5b</td>
<td>pop</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08048659</td>
<td>c9</td>
<td>leave</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0804865a</td>
<td>c3</td>
<td>ret</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**x86**
Question

Would you implement this with a jump table?

```c
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 52001 entries (too big)