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

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

Computer system:

Machine code:
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111

OS:
Windows 8
Mac

x86 Programming
Next x86 topics

- Move instructions, registers, and operands
- Memory addressing modes
- `swap` example: 32-bit vs. 64-bit
- Arithmetic operations
What Is A Register (again)?

- A location in the CPU that stores a small amount of data, which can be accessed very quickly (once every clock cycle)

- Registers have names, not addresses.

- Registers are at the heart of assembly programming
  - They are a precious commodity in all architectures, but especially x86
Integer Registers (IA32)

- `%eax`
- `%ecx`
- `%edx`
- `%ebx`
- `%esi`
- `%edi`
- `%esp`
- `%ebp`

General purpose registers are 32-bits wide.

Origin (mostly obsolete):
- accumulate
- counter
- data
- base
- source
- index
- destination
- index
- stack
- pointer
- base
- pointer
## Integer Registers (IA32)

<table>
<thead>
<tr>
<th>%eax</th>
<th>%ax</th>
<th>%ah</th>
<th>%al</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>%cx</td>
<td>%ch</td>
<td>%cl</td>
</tr>
<tr>
<td>%edx</td>
<td>%dx</td>
<td>%dh</td>
<td>%dl</td>
</tr>
<tr>
<td>%ebx</td>
<td>%bx</td>
<td>%bh</td>
<td>%bl</td>
</tr>
<tr>
<td>%esi</td>
<td>%si</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td>%di</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td>%sp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>%bp</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **%ah**, **%ch**, **%dh**, **%bh**, **%bl**
- **%al**, **%cl**, **%dl**, **%bl**

### General Purpose Registers
- **%eax**: Accumulate
- **%ecx**: Counter
- **%edx**: Data
- **%ebx**: Base
- **%esi**: Source Index
- **%edi**: Destination Index
- **%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

| %rax | %eax | 64-bits wide |
| %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, and add 8 new ones; all accessible as 8, 16, 32, 64 bits.
Assembly Data Types

- “Integer” data of 1, 2, 4 (IA32), or 8 (just in x86-64) bytes
  - Data values
  - Addresses (untyped pointers)

- Floating point data of 4, 8, or 10 bytes

- What about “aggregate” types such as arrays?
  - Just contiguous memory locations
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;  
  - z = x << y;  
  - i = h & g;

- **Transfer control: what instruction to execute next**
  - Unconditional jumps to/from procedures
  - Conditional branches

Remember: memory is indexed just like an array[] of bytes!
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

confusing historical terms...
not the current machine word size
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></td>
<td>Reg</td>
<td>movl $0x4,%eax</td>
<td>var_a = 0x4;</td>
</tr>
<tr>
<td></td>
<td>Imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>movl $-147,(%eax)</td>
<td>*p_a = -147;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reg</td>
<td>Reg</td>
<td>movl %eax,%edx</td>
<td>var_d = var_a;</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>movl %eax,(%edx)</td>
<td>*p_d = var_a;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl (%eax),%edx</td>
<td>var_d = *p_a;</td>
</tr>
</tbody>
</table>

Cannot do memory-memory transfer with a single instruction.
How would you do it?
Memory vs. registers

- What is the main difference?
- Addresses vs. Names
- Big vs. Small
Memory Addressing Modes: Basic

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

- **Displacement**  
  \( \text{Mem}[\text{Reg}[R]+D] \)
  
  - Register R specifies a memory address
    - (e.g. the start of some memory region)
  
  - Constant displacement D specifies the offset from that address
  
  \[
  \text{movl } 8(\%ebp),\%edx
  \]
Using Basic Addressing Modes

```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
```

- **Set Up**
- **Body**
- **Finish**
void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}

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

Register   Value
%ecx    yp
%edx    xp
%eax    t1
%ebx    t0

Stack (in memory)
Offset     xp
12         yp
8
4
0
-4
Old %ebp
Old %ebx

Register <- variable mapping
Understanding Swap

%eax
%edx
%ecx
%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

Address
0x124
0x120
0x11c
0x118
0x114
0x110
0x124
0x10c
0x108
0x104
0x100

Offset
12
8
4
0
-4

yp
xp
%ebp
Rtn adr
 lowers
higher addresses
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>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x124</td>
</tr>
<tr>
<td>%edx</td>
<td>0x120</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

<table>
<thead>
<tr>
<th>Variable</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></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>12</td>
<td>0x120</td>
</tr>
<tr>
<td>8</td>
<td>0x124</td>
</tr>
<tr>
<td>4</td>
<td>0x11c</td>
</tr>
<tr>
<td>0</td>
<td>0x118</td>
</tr>
<tr>
<td>-4</td>
<td>0x114</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>

```
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
```
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
```
x86-64 Integer Registers

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

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

- New instruction forms:
  - movl → movq
  - addl → addq
  - sall → 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: 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
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
```

**Setup**

**Body**

**Finish**

---

<table>
<thead>
<tr>
<th>Offset</th>
<th>Variable</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>

Spring 2014
Swap Ints in 64-bit Mode

```c
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: faster
- 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;
}
```

```assembly
swap_l:              
    movq (%rdi), %rdx
    movq (%rsi), %rax
    movq %rax, (%rdi)
    movq %rdx, (%rsi)
    retq
```

**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:**

\[
D(Rb, Ri, S) \quad \text{Mem}[\text{Reg}[Rb] + S*\text{Reg}[Ri] + D]
\]

- **D:** Constant “displacement” value represented in 1, 2, or 4 bytes
- **Rb:** Base register: Any of the 8/16 integer registers
- **Ri:** Index register: Any, except for `%esp` or `%rsp`; `%ebp` unlikely
- **S:** Scale: 1, 2, 4, or 8 (*why these numbers?*)

**Special Cases:** can use any combination of D, Rb, Ri and S

- \((Rb, Ri)\) \quad \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]] \quad (S=1, D=0)
- \(D(Rb, Ri)\) \quad \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]+D] \quad (S=1)
- \((Rb, Ri, S)\) \quad \text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]] \quad (D=0)
## 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>D(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><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>shll</code> <code>Src,Dest</code></td>
<td><code>Dest = Dest &lt;&lt; Src</code> <strong>Also called sall</strong></td>
</tr>
<tr>
<td><code>sarl</code> <code>Src,Dest</code></td>
<td><code>Dest = Dest &gt;&gt; Src</code> <strong>Arithmetic</strong></td>
</tr>
<tr>
<td><code>shrl</code> <code>Src,Dest</code></td>
<td><code>Dest = Dest &gt;&gt; Src</code> <strong>Logical</strong></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>

- **Watch out for argument order! (especially `subl`)**
- **No distinction between signed and unsigned int (why?)**
  - except arithmetic vs. logical shift right
Some Arithmetic Operations

- **One Operand (Unary) Instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>incl Dest</td>
<td>Dest = Dest + 1 increment</td>
</tr>
<tr>
<td>decl Dest</td>
<td>Dest = Dest - 1 decrement</td>
</tr>
<tr>
<td>negl Dest</td>
<td>Dest = -Dest negate</td>
</tr>
<tr>
<td>notl Dest</td>
<td>Dest = ~Dest bitwise complement</td>
</tr>
</tbody>
</table>

- See textbook section 3.5.5 for more instructions: mull, cltd, idivl, divl
Using `leal` for Arithmetic Expressions (IA32)

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

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

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

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

Set Up

Body

Finish
Understanding arith (IA32)

```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 (IA32)

```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 (IA32)

```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 (IA32)

```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 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 when compile:
  - \((x+y+z) \times (x+4+48\times y)\)
Another Example (IA32)

```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 function:
- **Set Up**:
  - `pushl %ebp`
  - `movl %esp,%ebp`
  - `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`
  - `xorl 12(%ebp),%eax`
  - `sarl $17,%eax`
  - `andl $8185,%eax`

Stack:
- Offset
  - 0: Old %ebp
  - 4: Rtn adr
  - 8: x
  - 12: y
Another Example (IA32)

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

```assembly
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 (IA32)

```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
```

Set Up

Body

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`
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:
\begin{align*}
\text{pushl} & \quad \%ebp \\
\text{movl} & \quad \%esp,\%ebp \\
\text{movl} & \quad 8(\%ebp),\%eax \\
\text{xorl} & \quad 12(\%ebp),\%eax \\
\text{sarl} & \quad \$17,\%eax \\
\text{andl} & \quad \$8185,\%eax \\
\text{movl} & \quad \%ebp,\%esp \\
\text{popl} & \quad \%ebp \\
\text{ret} & \quad \end{align*}

compiler optimization

\begin{align*}
\text{movl} & \quad 8(\%ebp),\%eax \\
\text{xorl} & \quad 12(\%ebp),\%eax \\
\text{sarl} & \quad \$17,\%eax \\
\text{andl} & \quad \$8185,\%eax \\
\end{align*}

eax = x
\begin{align*}
\text{movl} & \quad 8(\%ebp),\%eax \\
\text{xorl} & \quad 12(\%ebp),\%eax \\
\text{sarl} & \quad \$17,\%eax \\
\text{andl} & \quad \$8185,\%eax \\
\end{align*}

eax = x^y \quad (t1)
eax = t1>>17 \quad (t2)
eax = t2 \& 8185

---

Set Up
Body
Finish
Topics: control flow

- Condition codes
- Conditional and unconditional branches
- Loops
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
  - Takes address as argument

<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)^~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)^ZF</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>ja</td>
<td>~CF^~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

Current stack top

Current stack frame

Instruction pointer

Condition codes
Condition Codes (Implicit Setting)

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

■ Implicitly set (think of it as side effect) by arithmetic operations

  Example:  \( \text{addl/addq } \text{Src,Dest} \leftrightarrow t = a+b \)

  - **CF set** if carry out from most significant bit (unsigned overflow)
  - **ZF set** if \( t == 0 \)
  - **SF set** if \( t < 0 \) (as signed)
  - **OF set** if two’s complement (signed) overflow
    
    \[ (a>0 \land b>0 \land t<0) \lor (a<0 \land b<0 \land t>=0) \]

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

■ **Full documentation (IA32):**  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**
  - `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)

- **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)

```assembly
movl 12(%ebp),%eax  # eax = y
cmpl %eax,8(%ebp)   # Compare x : y
setg %al
movzbl %al,%eax    # Zero rest of %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)

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

- **jX Instructions**
  - Jump to different part of code depending on condition codes
  - Takes address as argument

<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
```

Setup

Body1

Finish

Body2
Conditional Branch Example (Cont.)

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

```
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
Conditional Branch Example (Cont.)

```c
int goto_ad(int x, int y)
{
    int result;
    if (x <= y) goto Else;
    result = x-y;
    Exit:
    return result;
Else:
    result = y-x;
    goto Exit;
}
```

```
absdiff:
pushl  %ebp
movl  %esp, %ebp
movl  8(%ebp), %edx
movl  12(%ebp), %eax
cmpeq  %eax, %edx
jle  .L7
subl  %eax, %edx
movl  %edx, %eax
.L8:
leave
ret
.L7:
subl  %edx, %eax
jmp  .L8
```

```c
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;
}
```

```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:
    leave
    ret
.L8:
    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;
}
```

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 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:
```
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
```

```c
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:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %edx
    movl 12(%ebp), %eax
    cmpl %eax, %edx
    jle .L7
    subl %eax, %edx
    jmp .L8
.L8:
    leave
    ret
.L7:
    subl %edx, %eax
    jmp .L8
```

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

C Code

```c
val = Test ? Then-Expr : Else-Expr;
result = 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 might you make this more 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;
}
```

```
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
  - *Why is this good?*
Conditionals: x86-64

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

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

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
“Do-While” Loop Example

C Code

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

Goto Version

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

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

Goto Version

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

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

    return result;
}
```

Assembly

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

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

Registers:

- %edx: x
- %eax: result

Translation?
“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
    cmpl $1,%edx
    jg .L11

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

Registers:

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

C Code

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

- **Body:**
  \[
  \begin{align*}
  &\text{Statement}_1; \\
  &\text{Statement}_2; \\
  &\quad \ldots \\
  &\text{Statement}_n;
  \end{align*}
  \]

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

Goto Version

\[
\text{loop:} \\
\quad \text{Body} \\
\quad \text{if} \ (\text{Test}) \\
\quad \quad \text{goto loop}
\]
“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;
}
```

```
# x in %edx, result in %eax
jmp .L34    # goto Middle
.L35:    # Loop:
imull %edx, %eax # result *= x
decl %edx # x--
.L34: # Middle:
compl $1, %edx # x:1
jg .L35 # if >, goto Loop
```
“For” Loop Example: Square-and-Multiply

Algorithm

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

```
/* 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;
}
```

Example

$3^{10} = 3^2 * 3^8$

$= 3^2 * ((3^2)^2)^2$

$3^1 = 1$

$12^{31} * \ldots * 16 * x^8 * x^4 * 1^2 * x^1 = x^{13}$

$x^m * x^n = x^{m+n}$
/* 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;
}

<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

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

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

For Version

```c
for (Initialize; Test; Update) {
    Body
}
```

While Version

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

Goto Version

```c
Initialize;
goto middle;

loop:
    Body
    Update;

middle:
    if (Test)
        goto loop;

done:
```
For-Loop: Compilation

For Version

```c
for (Initialize; Test; Update)
{
  Body
}
```

Goto Version

```c
result = 1;
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

- Lots to manage, we need a *jump table*

```c
long switch_eg (unsigned 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 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:

```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 (IA32)

```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:      // .L60
        w -= z;
        break;
    case 6:      // .L60
        w -= z;
        break;
    default:     // .L61
        w = 2;
}
```

- declaring data, not instructions
- Jump table
- 4-byte memory alignment
- "long" as in movl: 4 bytes would be .quad in x86-64
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
```
Switch Statement Example (IA32)

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

**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
```

**Setup:**
```
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 > 6 goto default
jmp  *.L62(%edx,4) # goto JTab[x]
```
Assembly Setup Explanation (IA32)

■ 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;
}
return w;
```

```assembly
.L61:    // Default case
  movl $2, %ebx    # w = 2
  jmp .L63
.L57:    // Case 2:
  movl 12(%ebp), %eax # y
  cltd             # Div prep
  idivl %ecx       # y/z
  movl %eax, %ebx  # w = y/z
  # Fall through - no jmp
.L58:    // Case 3:
  addl %ecx, %ebx  # w += z
  jmp .L63
...
.L63
  movl %ebx, %eax  # return w
  popl %ebx
  leave
  ret
```
Code Blocks (Rest)

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

```
.L56:  // Case 1:
    movl 12(%ebp), %ebx  # w = y
    imull %ecx, %ebx     # w*= z
    jmp .L63

.L60:  // Cases 5&6:
    subl %ecx, %ebx      # w -= z
    jmp .L63

.L63
    movl %ebx, %eax      # return w
    popl %ebx
    leave
    ret
```
switch(x) {
    . . .
    case 2:      // .L57
        w = y/z;
        /* Fall Through */
    case 3:      // .L58
        w += z;
        break;
    . . .
    default:     // .L61
        w = 2;
}

The compiler might choose to pull the return statement in to each relevant case rather than jumping out to it.
The compiler might choose to pull the return statement in to each relevant case rather than jumping out to it.
IA32 Object Code

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

**Assembly Code**

```assembly
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
    ```

```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x080488dc</td>
<td>0x08048630</td>
</tr>
<tr>
<td>0x08048650</td>
<td></td>
</tr>
<tr>
<td>0x0804863a</td>
<td></td>
</tr>
<tr>
<td>0x08048642</td>
<td></td>
</tr>
<tr>
<td>0x08048630</td>
<td></td>
</tr>
<tr>
<td>0x08048649</td>
<td></td>
</tr>
<tr>
<td>0x08048649</td>
<td></td>
</tr>
</tbody>
</table>

---
## Disassembled Targets

<table>
<thead>
<tr>
<th>Address</th>
<th>Machine Code</th>
<th>Assembly Code</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

### Code Snippet

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08048630</td>
<td>bb 02 00 00 00</td>
<td>mov</td>
</tr>
<tr>
<td>0x08048635</td>
<td>89 d8</td>
<td>mov</td>
</tr>
<tr>
<td>0x08048637</td>
<td>5b</td>
<td>pop</td>
</tr>
<tr>
<td>0x08048638</td>
<td>c9</td>
<td>leave</td>
</tr>
<tr>
<td>0x08048639</td>
<td>c3</td>
<td>ret</td>
</tr>
<tr>
<td>0x0804863a</td>
<td>8b 45 0c</td>
<td>mov</td>
</tr>
<tr>
<td>0x0804863d</td>
<td>99</td>
<td>cltd</td>
</tr>
<tr>
<td>0x0804863e</td>
<td>f7 f9</td>
<td>idiv</td>
</tr>
<tr>
<td>0x08048640</td>
<td>89 c3</td>
<td>mov</td>
</tr>
<tr>
<td>0x08048642</td>
<td>01 cb</td>
<td>add</td>
</tr>
<tr>
<td>0x08048644</td>
<td>89 d8</td>
<td>mov</td>
</tr>
<tr>
<td>0x08048646</td>
<td>5b</td>
<td>pop</td>
</tr>
<tr>
<td>0x08048647</td>
<td>c9</td>
<td>leave</td>
</tr>
<tr>
<td>0x08048648</td>
<td>c3</td>
<td>ret</td>
</tr>
<tr>
<td>0x08048649</td>
<td>29 cb</td>
<td>sub</td>
</tr>
<tr>
<td>0x0804864b</td>
<td>89 d8</td>
<td>mov</td>
</tr>
<tr>
<td>0x0804864d</td>
<td>5b</td>
<td>pop</td>
</tr>
<tr>
<td>0x0804864e</td>
<td>c9</td>
<td>leave</td>
</tr>
<tr>
<td>0x0804864f</td>
<td>c3</td>
<td>ret</td>
</tr>
<tr>
<td>0x08048650</td>
<td>8b 5d 0c</td>
<td>mov</td>
</tr>
<tr>
<td>0x08048653</td>
<td>0f af d9</td>
<td>imul</td>
</tr>
<tr>
<td>0x08048656</td>
<td>89 d8</td>
<td>mov</td>
</tr>
<tr>
<td>0x08048658</td>
<td>5b</td>
<td>pop</td>
</tr>
<tr>
<td>0x08048659</td>
<td>c9</td>
<td>leave</td>
</tr>
<tr>
<td>0x0804865a</td>
<td>c3</td>
<td>ret</td>
</tr>
</tbody>
</table>

### References

- University of Washington
  - Spring 2014
  - x86 Programming
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 for only 4 cases (too big)
- about 200KB = 200,000 bytes
- text of this switch statement = about 200 bytes
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
  - x86-64: better support for passing function arguments in registers

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

- Immediate (constant), Register, and Memory Operands
  - 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
Quick Review

- **Control**
  - 1-bit condition code registers
  - Set as side effect by arithmetic instructions or by `cmp, test`
  - Used:
    - Read out by setx instructions (`setg, setle, ...`)
    - Or by conditional jumps (`jle .L4, je .L10, ...`)
    - Or by conditional moves (`cmovle %edx, %eax`)

- **Arithmetic operations also set condition codes**
  - `subl, addl, imull, shrl, etc.`

- **Load Effective Address does NOT set condition codes**
  - `lea 4(%edx,%eax),%eax  # eax = 4 + edx + eax`
Quick Review

- **Do-While loop**

  ![C Code for Do-While loop]

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

  ![Goto Version for Do-While loop]

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

- **While-Do loop**

  ![While version]

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

  ![Do-While Version for While-Do loop]

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

  ![Goto Version for While-Do loop]

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

  ![or]

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

- **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 textbook)

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