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

OS:

Windows 8
Mac

Memory & data
Integers & floats
Machine code & C

x86 assembly

Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C
Next x86 topics

- x86 basics: registers
- Move instructions, registers, and operands
- Memory addressing modes
- swap example
- 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
### x86-64 Integer Registers – 64 bits wide

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

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)
Some History: IA32 Registers – 32 bits wide

<table>
<thead>
<tr>
<th>Register</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>accumulate</td>
</tr>
<tr>
<td>%ecx</td>
<td>counter</td>
</tr>
<tr>
<td>%edx</td>
<td>data</td>
</tr>
<tr>
<td>%ebx</td>
<td>base</td>
</tr>
<tr>
<td>%esi</td>
<td>source index</td>
</tr>
<tr>
<td>%edi</td>
<td>destination index</td>
</tr>
<tr>
<td>%esp</td>
<td>stack pointer</td>
</tr>
<tr>
<td>%ebp</td>
<td>base pointer</td>
</tr>
</tbody>
</table>

Name Origin (mostly obsolete)

16-bit virtual registers (backwards compatibility)
Assembly Data Types

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

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

- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory
Three Basic Kinds of Instructions

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

- **Perform arithmetic function on register or memory data**
  - \( c = a + b; \quad z = x \ll y; \quad i = h \& g; \)

- **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

- **Moving Data**
  - `movq 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 16 integer registers
    - Example: `%rax, %r13`
    - But `%rsp` reserved for special use
    - Others have special uses for particular instructions
  - **Memory**: 8 consecutive bytes of memory at address given by register
    - Simplest example: `( %rax )`
    - Various other “address modes”
**movq, movl, movw, movb**

**Moving Data**
- **movx Source, Dest**: 
  Move 8-byte “quad word”
- **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
movq 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>movq $0x4, %rax</td>
<td>var_a = 0x4;</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>movq $-147, (%rax)</td>
<td>*p_a = -147;</td>
</tr>
<tr>
<td></td>
<td>Reg</td>
<td>movq %rax, %rdx</td>
<td>var_d = var_a;</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>movq %rax, (%rdx)</td>
<td>*p_d = var_a;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq (%rax), %rdx</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 \((R)\) \(\text{Mem}[\text{Reg}[R]]\)
  - Register R specifies the memory address
  - Aha! Pointer dereferencing in C

\[
\text{movq} \ (\%rcx),\%rax
\]

- 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

\[
\text{movq} \ 8(\%rbp),\%rdx
\]
Example of Basic Addressing Modes

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

swap:

```
    movq (%rdi), %rax
    movq (%rsi), %rdx
    movq %rdx, (%rdi)
    movq %rax, (%rsi)
    ret
```
Understanding Swap()

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

**Memory**

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>xp</td>
</tr>
<tr>
<td>%rsi</td>
<td>yp</td>
</tr>
<tr>
<td>%rax</td>
<td>t0</td>
</tr>
<tr>
<td>%rdx</td>
<td>t1</td>
</tr>
</tbody>
</table>

**Assembly**

```assembly
swap:
    movq (%rdi), %rax  # t0 = *xp
    movq (%rsi), %rdx  # t1 = *yp
    movq %rdx, (%rdi)  # *xp = t1
    movq %rax, (%rsi)  # *yp = t0
    ret
```
Understanding Swap()\

Registers

<table>
<thead>
<tr>
<th>%rdi</th>
<th>0x120</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td></td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>123</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x120</td>
</tr>
<tr>
<td></td>
<td>0x118</td>
</tr>
<tr>
<td></td>
<td>0x110</td>
</tr>
<tr>
<td></td>
<td>0x108</td>
</tr>
<tr>
<td></td>
<td>0x100</td>
</tr>
<tr>
<td></td>
<td>456</td>
</tr>
</tbody>
</table>

swap:

- `movq (%rdi), %rax` # t0 = *xp
- `movq (%rsi), %rdx` # t1 = *yp
- `movq %rdx, (%rdi)` # *xp = t1
- `movq %rax, (%rsi)` # *yp = t0
- `ret`
Understanding `Swap()`

**Registers**

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rdi</code></td>
<td>0x120</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>0x100</td>
</tr>
<tr>
<td><code>%rax</code></td>
<td>123</td>
</tr>
<tr>
<td><code>%rdx</code></td>
<td></td>
</tr>
</tbody>
</table>

**Memory**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>123</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td>456</td>
</tr>
</tbody>
</table>

swap:

```
movq (%rdi), %rax # t0 = *xp
movq (%rsi), %rdx # t1 = *yp
movq %rdx, (%rdi) # *xp = t1
movq %rax, (%rsi) # *yp = t0
ret
```
Understanding Swap()

Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td>123</td>
</tr>
<tr>
<td>%rdx</td>
<td>456</td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>123</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td>456</td>
</tr>
</tbody>
</table>

swap:

```
movq    (%rdi), %rax  # t0 = *xp
movq    (%rsi), %rdx  # t1 = *yp
movq    %rdx, (%rdi)  # *xp = t1
movq    %rax, (%rsi)  # *yp = t0
ret
```
Understanding Swap()

Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td>123</td>
</tr>
<tr>
<td>%rdx</td>
<td>456</td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>456</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td>456</td>
</tr>
</tbody>
</table>

swap:

```
movq (%rdi), %rax  # t0 = *xp
movq (%rsi), %rdx  # t1 = *yp
movq %rdx, (%rdi)  # *xp = t1
movq %rax, (%rsi)  # *yp = t0
ret
```
# Understanding Swap()

## Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td>123</td>
</tr>
<tr>
<td>%rdx</td>
<td>456</td>
</tr>
</tbody>
</table>

## Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>456</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td></td>
</tr>
</tbody>
</table>

## swap:

```
movq    (%rdi), %rax  # t0 = *xp
movq    (%rsi), %rdx  # t1 = *yp
movq    %rdx, (%rdi)  # *xp = t1
movq    %rax, (%rsi)  # *yp = t0
ret     
```
Memory Addressing Modes: Basic

- **Indirect (R)**  \(\text{Mem}[\text{Reg}[R]]\)
  - Register R specifies the memory address
  - Aha! Pointer dereferencing in C

  \[
  \text{movq } (%rcx),%rax
  \]

- **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

  \[
  \text{movq } 8(%rbp),%rdx
  \]
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 16 integer registers
- **Ri**: Index register: Any, except for `%rsp`
- **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>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%rdx)</td>
<td>Mem[Reg[Rb]+Reg[Ri]]</td>
<td></td>
</tr>
<tr>
<td>(%rdx,%rcx)</td>
<td>Mem[S*Reg[Ri]+D]</td>
<td></td>
</tr>
<tr>
<td>(%rdx,%rcx,4)</td>
<td>Mem[Reg[Rb]+S*Reg[Ri]]</td>
<td></td>
</tr>
<tr>
<td>0x80(,%rdx,2)</td>
<td>Mem[Reg[Rb]+D]</td>
<td></td>
</tr>
</tbody>
</table>
# Address Computation Examples

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

**leaq** *Src, Dest*

- *Src* is address expression (Any of the formats we just discussed!)
- *Dest* is a register
- Set *Dest* to address computed by expression
  - (lea stands for *load effective address*)
- Example: `leaq (%rdx, %rcx, 4), %rax`

**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`
leaq vs. movq example

Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td></td>
</tr>
<tr>
<td>%rbx</td>
<td></td>
</tr>
<tr>
<td>%rcx</td>
<td>0x4</td>
</tr>
<tr>
<td>%rdx</td>
<td>0x100</td>
</tr>
<tr>
<td>%rdi</td>
<td></td>
</tr>
<tr>
<td>%rsi</td>
<td></td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400</td>
<td></td>
</tr>
<tr>
<td>0xf</td>
<td></td>
</tr>
<tr>
<td>0x8</td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td></td>
</tr>
<tr>
<td>0x1</td>
<td></td>
</tr>
<tr>
<td>0x120</td>
<td></td>
</tr>
<tr>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td></td>
</tr>
</tbody>
</table>

leaq (%rdx, %rcx, 4), %rax
movq (%rdx, %rcx, 4), %rbx
leaq (%rdx), %rdi
movq (%rdx), %rsi
leaq vs. movq example (solution)

Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x110</td>
</tr>
<tr>
<td>%rbx</td>
<td>0x8</td>
</tr>
<tr>
<td>%rcx</td>
<td>0x4</td>
</tr>
<tr>
<td>%rdx</td>
<td>0x100</td>
</tr>
<tr>
<td>%rdi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x1</td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400</td>
<td></td>
</tr>
<tr>
<td>0xf</td>
<td></td>
</tr>
<tr>
<td>0x8</td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td></td>
</tr>
<tr>
<td>0x1</td>
<td></td>
</tr>
</tbody>
</table>

leaq (%rdx,%rcx,4), %rax
movq (%rdx,%rcx,4), %rbx
leaq (%rdx), %rdi
movq (%rdx), %rsi
# Some Arithmetic Operations

## Two Operand (Binary) Instructions:

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq</td>
<td>Dest = Dest + Src</td>
</tr>
<tr>
<td>subq</td>
<td>Dest = Dest - Src</td>
</tr>
<tr>
<td>imulq</td>
<td>Dest = Dest * Src</td>
</tr>
<tr>
<td>shlq</td>
<td>Dest = Dest &lt;&lt; Src</td>
</tr>
<tr>
<td>sarq</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>shrq</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>xorq</td>
<td>Dest = Dest ^ Src</td>
</tr>
<tr>
<td>andq</td>
<td>Dest = Dest &amp; Src</td>
</tr>
<tr>
<td>orq</td>
<td>Dest = Dest</td>
</tr>
</tbody>
</table>

- **Also called salq**
- **Arithmetic**
- **Logical**

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

- One Operand (Unary) Instructions

  - `incq Dest`: \( \text{Dest} = \text{Dest} + 1 \)  
    increment
  
  - `decq Dest`: \( \text{Dest} = \text{Dest} - 1 \)  
    decrement
  
  - `negq Dest`: \( \text{Dest} = -\text{Dest} \)  
    negate
  
  - `notq Dest`: \( \text{Dest} = \sim\text{Dest} \)  
    bitwise complement

- See textbook section 3.5.5 for more instructions: `mulq`, `cqto`, `idivq`, `divq`
The `leaq` Instruction

- “lea” stands for *load effective address*
- Example: `leaq (%rdx, %rcx, 4), %rax`

Does the `leaq` instruction go to memory?

**NO**

“`leaq` – it just does math”
Using `leaq` for Arithmetic Expressions

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

**Interesting Instructions**

- **leaq**: address computation
- **salq**: shift
- **imulq**: multiplication
  - But, only used once instead of twice
Understanding arith

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

arith:
leaq (%rdi,%rsi), %rax  # t1
addq %rdx, %rax         # t2
leaq (%rsi,%rsi,2), %rdx
salq $4, %rdx           # t4
leaq 4(%rdi,%rdx), %rcx # t5
imulq %rcx, %rax        # rval
ret

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>t1, t2, rval</td>
</tr>
<tr>
<td>%rdx</td>
<td>t4</td>
</tr>
<tr>
<td>%rcx</td>
<td>t5</td>
</tr>
</tbody>
</table>
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) &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 (x86-64, Partial)

- Information about currently executing program
  - Temporary data (\%rax, ... )
  - Location of runtime stack (\%rsp )
  - Location of current code control point (\%rip, ... )
  - Status of recent tests (CF, ZF, SF, OF )

<table>
<thead>
<tr>
<th>Registers</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%r8</td>
</tr>
<tr>
<td>%rbx</td>
<td>%r9</td>
</tr>
<tr>
<td>%rcx</td>
<td>%r10</td>
</tr>
<tr>
<td>%rdx</td>
<td>%r11</td>
</tr>
<tr>
<td>%rsi</td>
<td>%r12</td>
</tr>
<tr>
<td>%rdi</td>
<td>%r13</td>
</tr>
<tr>
<td>%rsp</td>
<td>%r14</td>
</tr>
<tr>
<td>%rbp</td>
<td>%r15</td>
</tr>
</tbody>
</table>

Current stack top

Instruction pointer

<table>
<thead>
<tr>
<th>Condition codes</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CF</td>
<td></td>
</tr>
<tr>
<td>ZF</td>
<td></td>
</tr>
<tr>
<td>SF</td>
<td></td>
</tr>
<tr>
<td>OF</td>
<td></td>
</tr>
</tbody>
</table>
Condition Codes (Implicit Setting)

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

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

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

- **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
  \[ \text{if } (a>0 \land b>0 \land t<0) \text{ or } (a<0 \land b<0 \land t>=0) \]

**Not set by leaq instruction (beware!)**
Condition Codes (Explicit Setting: Compare)

- **Explicit Setting by Compare Instruction**

  \[ \text{cmpq } \text{Src2}, \text{Src1} \]

  \[ \text{cmpq b,a} \text{ like computing } a-b \text{ without setting destination} \]

  Single bit registers

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

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

\[ (a>0 \land b<0 \land (a-b)<0) \lor (a<0 \land b>0 \land (a-b)>0) \]
Condition Codes (Explicit Setting: Test)

Explicit Setting by Test instruction

\texttt{testq \ Src2,Src1}
\texttt{testq b,a} like computing \( a \land b \) without setting destination
  - Sets condition codes based on value of \( Src1 \land Src2 \)
  - Useful to have one of the operands be a mask

Single bit registers

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

- ZF set if \( a \land b = 0 \)
- SF set if \( a \land b < 0 \)

- \texttt{testq \%rax, \%rax}
  - Sets SF and ZF, check if rax is +,0,-
# Reading Condition Codes

- **SetX Instructions**
  - Set a low-order byte to 0 or 1 based on combinations of condition codes
  - Does not alter remaining 7 bytes

<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>
## x86-64 Integer Registers

<table>
<thead>
<tr>
<th>%rax</th>
<th>%al</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%bl</td>
</tr>
<tr>
<td>%rcx</td>
<td>%cl</td>
</tr>
<tr>
<td>%rdx</td>
<td>%dl</td>
</tr>
<tr>
<td>%rsi</td>
<td>%sil</td>
</tr>
<tr>
<td>%rdi</td>
<td>%dil</td>
</tr>
<tr>
<td>%rsp</td>
<td>%spl</td>
</tr>
<tr>
<td>%rbp</td>
<td>%bpl</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%r8</th>
<th>%r8b</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r9</td>
<td>%r9b</td>
</tr>
<tr>
<td>%r10</td>
<td>%r10b</td>
</tr>
<tr>
<td>%r11</td>
<td>%r11b</td>
</tr>
<tr>
<td>%r12</td>
<td>%r12b</td>
</tr>
<tr>
<td>%r13</td>
<td>%r13b</td>
</tr>
<tr>
<td>%r14</td>
<td>%r14b</td>
</tr>
<tr>
<td>%r15</td>
<td>%r15b</td>
</tr>
</tbody>
</table>

- Can reference low-order byte
Reading Condition Codes (Cont.)

- **SetX Instructions:**
  - Set single byte to 0 or 1 based on combination of condition codes
  - Operand is one of the byte registers (eg. al, dl) or a byte in memory

- **Set instruction does not alter remaining bytes in register**
  - Typically use movzbl to finish job - Sets upper 32 bits to zero
    - Aside: In x86-64, any instruction that generates a 32-bit value for a register also sets the high-order portion of the register to 0.

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

```c
cmpq    %rsi, %rdi
setg    %al
movzbl  %al, %eax
ret
```

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
  - Operand is one of the byte registers (eg. `al, dl`) or a byte in memory

- **Set instruction does not alter remaining bytes in register**
  - Typically use `movzbl` to finish job - Sets upper 32 bits to zero
    - Aside: In x86-64, any instruction that generates a 32-bit value for a register also sets the high-order portion of the register to 0.

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument <code>x</code></td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument <code>y</code></td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

```asm
cmpq %rsi, %rdi       # Compare x:y
setg %al             # al = x > y
movzbl %al, %eax     # Zero rest of %rax
ret                   
```
Aside: **movz** and **movs** examples

**movzbl** `Src, RegisterDest`  
*Move with zero extension*

**movsbl** `Src, RegisterDest`  
*Move with sign extension*

- For use when copying a smaller source value to a larger destination
- Source can be memory or register; Destination must be a register

**movz** `SD`  
- fills out remaining bytes of the destination with zeroes

**movs** `SD`  
- fills out remaining bytes of the destination by sign extension, replicating the most significant bit of the source

- `S` can be `b=byte, w=16-bit word`
- `D` can be `w=16-bit word, l=32-bit long word, q=64-bit quad word`

**Note:** In x86-64, **any instruction** that generates a 32-bit (long word) value for a register also sets the high-order portion of the register to 0.

Good example in the “Aside” on p. 184 in 3e CS-APP (our text)
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>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>
Conditional Branch Example (Old Style)

Generation

gcc -Og -S -fno-if-conversion control.c

long absdiff
    (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
Expressing with Goto Code

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

```c
long absdiff_j
    (long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
Else:
    result = y-x;
Done:
    return result;
}
```

- C allows “goto” as means of transferring control
  - Closer to machine-level programming style
- Generally considered bad coding style
General Conditional Expression Translation (Using Branches)

C Code

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

Example:

```c
result = x>y ? x-y : y-x;
```

Goto Version

```c
ntest = !Test;
if (ntest) goto Else;
val = Then_Expr;
goto Done;
Else:
  val = Else_Expr;
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
Using Conditional Moves

Conditional Move Instructions

- `cmovC src, dest`
- Move value from src to dest if condition \( C \) holds
- Instruction supports:
  - if \( (\text{Test}) \) Dest \( \leftarrow \) Src
- Supported in post-1995 x86 processors
- GCC tries to use them
  - But, only when known to be \textit{safe}

Why is this useful?

- Branches are very disruptive to instruction flow through pipelines
- Conditional moves do not require control transfer

C Code

```c
val = Test
? Then_Expr
: Else_Expr;
```

Goto Version

```c
result = Then_Expr;
else_val = Else_Expr;
nt = !Test;
if (nt) result = else_val;
return result;
```
Conditional Move Example

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

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

absdiff:
  movq %rdi, %rax # x
  subq %rsi, %rax # result = x-y
  movq %rsi, %rdx
  subq %rdi, %rdx # else_val = y-x
  cmpq %rsi, %rdi # x:y
  cmovle %rdx, %rax # if <=, result = else_val
ret
Bad Cases for Conditional Move

Expensive Computations

\[ \text{val} = \text{Test}(x) \ ? \text{Hard1}(x) \ : \text{Hard2}(x); \]

- Both values get computed
- Only makes sense when computations are very simple

Risky Computations

\[ \text{val} = p \ ? \ast p : 0; \]

- Both values get computed
- May have undesirable effects

Computations with side effects

\[ \text{val} = x > 0 \ ? x*=7 : x+=3; \]

- Both values get computed
- Must be side-effect free
Compiling Loops

C/Java code:

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

Machine code:

```assembly
loopTop:  cmpl $0, %eax
          je   loopDone
          ; <loop body code>
loopDone: jmp   loopTop
```

- 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
long pcount_do(unsigned long x) {
    long result = 0;
    do {
        result += x & 0x1;
        x >>= 1;
    } while (x);
    return result;
}
```

Goto Version

```c
long pcount_goto(unsigned long x) {
    long result = 0;
    loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
    return result;
}
```

- Count number of 1’s in argument x (“popcount”)
- Use backward branch to continue looping
- Only take branch when “while” condition holds
Do-While Loop Compilation

Goto Version

```c
long pcount_goto(unsigned long x)
{
    long result = 0;

    loop:
        result += x & 0x1;
        x >>= 1;
        if(x) goto loop;
    return result;
}
```

Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rax</td>
<td>result</td>
</tr>
</tbody>
</table>

```
movl $0, %eax    # result = 0
    .L2:            # loop:
    movq %rdi, %rdx
    movq %rdi, %rdx
    andl $1, %edx   # t = x & 0x1
    addq %rdx, %rax # result += t
    shrq %rdi       # x >>= 1
    jne .L2         # if (x) goto loop
rep; ret
```
General Do-While Loop 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
General **While Loop - Translation #1**

- “Jump-to-middle” translation
- Used with `-Og`

While version

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

Goto Version

```c
goto test;
loop:
    Body
test:
    if (Test)
        goto loop;
done:
```
While Loop Example – Translation #1

C Code

```c
long pcount_while
    (unsigned long x) {
    long result = 0;
    while (x) {
        result += x & 0x1;
        x >>= 1;
    }
    return result;
}
```

Jump to Middle

```c
long pcount_goto_jtm
    (unsigned long x) {
    long result = 0;
    goto test;
    loop:
    result += x & 0x1;
    x >>= 1;
    test:
    if(x) goto loop;
    return result;
}
```

- Used with -Og
- Compare to do-while version of function
- Initial goto starts loop at test
General While Loop - Translation #2

While version

while (Test)
  Body

Do-While Version

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

Goto Version

if (!Test)
  goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
While Loop Example – Translation #2

C Code

```c
long pcount_while
    (unsigned long x) {
    long result = 0;
    while (x) {
        result += x & 0x1;
        x >>= 1;
    }
    return result;
}
```

Do-While Version

```c
long pcount_goto_dw
    (unsigned long x) {
    long result = 0;
    if (!x) goto done;
    loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
    done:
    return result;
}
```

- Used with -O1
- Compare to do-while version of function
- Initial conditional guards entrance to loop
For Loop Form

General Form

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

\[ \text{Body} \]

#define WSIZE 8*sizeof(int)
long pcount_for
  (unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
For Loop $\rightarrow$ While Loop

For Version

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

While Version

```c
Init;
while (Test) {
    Body
    Update;
}
```
For Loop-While Conversion

Init

\( i = 0 \)

Test

\( i < \text{WSIZE} \)

Update

\( i++ \)

Body

\[
\begin{align*}
\text{long pcount_for_while} \\
\quad (\text{unsigned long x}) \\
\quad \{ \\
\quad \quad \text{size_t i;} \\
\quad \quad \text{long result} = 0; \\
\quad \quad i = 0; \\
\quad \quad \text{while} (i < \text{WSIZE}) \\
\quad \quad \quad \{ \\
\quad \quad \quad \quad \text{unsigned bit} = \\
\quad \quad \quad \quad (x >> i) \& 0x1; \\
\quad \quad \quad \quad \text{result} += \text{bit;} \\
\quad \quad \quad \quad i++; \\
\quad \quad \quad \} \\
\quad \}\text{return result;}
\end{align*}
\]
For Loop Do-While Conversion

C Code

```c
long pcount_for_for_loop_dw
(unsigned long x)
{
    size_t i;
    long result = 0;
    for (i = 0; i < WSIZE; i++)
    {
        unsigned bit =
            (x >> i) & 0x1;
        result += bit;
    }
    return result;
}
```

- Initial test can be optimized away

Goto Version

```c
long pcount_for_goto_dw
(unsigned long x) {
    size_t i;
    long result = 0;
    i = 0;
    if (!(i < WSIZE))
        goto done;
    loop:
    {
        unsigned bit =
            (x >> i) & 0x1;
        result += bit;
    }
    i++;
    if (i < WSIZE)
        goto loop;
    done:
    return result;
}
```

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

Switch Form

```plaintext
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;
```
Switch Statement Example

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

Setup:

```
switch_eg:
    movq   %rdx, %rcx
    cmpq   $6, %rdi    # x:6
    ja     .L8
    jmp    *.L4(%rdi,8)
```

What range of values takes default?

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

Note that w not initialized here
Switch Statement Example

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

Setup:

```
jump above
(like jg, but unsigned)
```

```
switch_eg:
    movq  %rdx, %rcx
    cmpq  $6, %rdi       # x:6
    ja    .L8           # Use default
    jmp   *.L4(,%rdi,8)  # goto *JTab[x]
```

Jump table

```
.section .rodata
.align 8
.L4:
    .quad  .L8  # x = 0
    .quad  .L3  # x = 1
    .quad  .L5  # x = 2
    .quad  .L9  # x = 3
    .quad  .L8  # x = 4
    .quad  .L7  # x = 5
    .quad  .L7  # x = 6
```
Assembly Setup Explanation

- **Table Structure**
  - Each target requires 8 bytes
  - Base address at .L4

- **Jumping**
  - **Direct:** `jmp .L8`
  - Jump target is denoted by label `.L8`

  - **Indirect:** `jmp *.`L4`(.r8di, 8)`
  - Start of jump table: `.L4`
  - Must scale by factor of 8 (addresses are 8 bytes)
  - Fetch target from effective Address `.L4 + x*8`
    - Only for $0 \leq x \leq 6$
Jump Table

```
switch(x) {
    case 1:      // .L3
        w = y*z;
        break;
    case 2:      // .L5
        w = y/z;
        /* Fall Through */
    case 3:      // .L9
        w += z;
        break;
    case 5:       
    case 6:      // .L7
        w -= z;
        break;
    default:     // .L8
        w = 2;
}
```
switch(x) {
  case 1:    // .L3
      w = y*z;
      break;
  . . .
}

.L3:
  movq  %rsi, %rax  # y
  imulq  %rdx, %rax  # y*z
  ret

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
long w = 1;
    . . .
switch(x) {
    . . .
case 2:    // .L5
    w = y/z;
    /* Fall Through */
case 3:    // .L9
    w += z;
    break;
    . . .
}
case 2:
    w = y/z;
    goto merge;
case 3:
    w = 1;
merge:
    w += z;
Code Blocks (x == 2, x == 3)

long w = 1;

. . .
switch(x) {
  . . .
  case 2:   // .L5
    w = y/z;
    /* Fall Through */
  case 3:   // .L9
    w += z;
    break;
  . . .
}

.L5:
  movq %rsi, %rax # y in rax
  cqto
  idivq %rcx # y/z
  jmp .L6 # goto merge

.L9:
  movl $1, %eax # w = 1

.L6:
  addq %rcx, %rax # w += z
  ret

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Code Blocks (x == 5, x == 6, default)

```
switch(x) {
    ...  
    case 5:  // .L7
        w -= z;
        break;
    case 6:  // .L7
        w -= z;
        break;
    default: // .L8
        w = 2;
}
```

```
.L7:               # Case 5,6
    movl $1, %eax   # w = 1
    subq %rdx, %rax # w -= z
    ret

.L8:               # Default:
    movl $2, %eax   # 2
    ret
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
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