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:
0111010000000011000
1000110100000000000
1000100111000010
110000000000001111

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
Next x86 topics

- x86 basics: registers
- Move instructions, registers, and operands
- Arithmetic operations
- Memory addressing modes
- swap example
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*
  - In assembly, they start with % (e.g. %rsi)

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

#### General Purpose Registers

- **eax**: 32-bit general purpose register
- **ecx**: 32-bit general purpose register
- **edx**: 32-bit general purpose register
- **ebx**: 32-bit general purpose register
- **esi**: 32-bit general purpose register
- **edi**: 32-bit general purpose register
- **esp**: 32-bit stack pointer
- **ebp**: 32-bit base pointer

#### 16-bit Virtual Registers (backwards compatibility)

- **%ax**: 16-bit virtual register
- **%cx**: 16-bit virtual register
- **%dx**: 16-bit virtual register
- **%bx**: 16-bit virtual register
- **%si**: 16-bit virtual register
- **%di**: 16-bit virtual register
- **%sp**: 16-bit stack pointer
- **%bp**: 16-bit base pointer

#### Name Origin

- **%eax**: Accumulate
- **%ecx**: Counter
- **%edx**: Data
- **%ebx**: Base
- **%esi**: Source Index
- **%edi**: Destination Index
- **%esp**: Stack Pointer
- **%ebp**: Base Pointer
Assembly Data Types

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

■ Floating point data of 4, 8, 10 or 2x8 or 4x4 or 8x2
  ▪ Different registers for those (e.g. `%xmm1, %ymm2)
  ▪ Come from extensions to x86 (SSE, AVX, ...)
  ▪ Probably won’t have time to get into these 😞

■ No aggregate types such as arrays or structures
  ▪ Just contiguously allocated bytes in memory

Two common syntaxes

■ “AT&T”: used by our course, slides, textbook, gnu tools, ...
■ “Intel”: used by Intel documentation, Intel tools, ...
■ Must know which you’re reading
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**
  - \( c = a + b; \quad z = x \ll y; \quad i = h \& g; \)

- **Control flow: what instruction to execute next**
  - Unconditional jumps to/from procedures
  - Conditional branches

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

- **Immediate**: Constant integer data
  - Example: $0x400, $-533
  - Like C literal, but prefixed with ‘$’
  - Encoded with 1, 2, 4, or 8 bytes depending on the instruction

- **Register**: One of 16 integer registers
  - Example: %rax, %r13
  - But %rsp reserved for special use
  - Others have special uses for particular instructions

- **Memory**: Consecutive bytes of memory at a given/computed address
  - Simplest example: (%rax)
  - Various other “address modes”
Moving Data

- **mov** _Source, Dest_
  - _ is one of {b, w, l, q}
    - Determines size of operands!

  - **movq** _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”

  *confusing historical terms... not the current machine word size*

- **Lots of these in typical code**
### 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>$0x4, %rax</td>
<td>Mem</td>
<td>movq $0x4, %rax</td>
<td>var_a = 0x4;</td>
</tr>
<tr>
<td>$-147, (%rax)</td>
<td>Mem</td>
<td>movq $-147, (%rax)</td>
<td>*p_a = -147;</td>
</tr>
<tr>
<td>%rax, %rdx</td>
<td>Reg</td>
<td>movq %rax, %rdx</td>
<td>var_d = var_a;</td>
</tr>
<tr>
<td>%rax, (%rdx)</td>
<td>Mem</td>
<td>movq %rax, (%rdx)</td>
<td>*p_d = var_a;</td>
</tr>
<tr>
<td>(%rax), %rdx</td>
<td>Mem, 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?**
### Some Arithmetic Operations

**Binary (two-operand) Instructions:**

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

*Maximum of one memory operand.*

**Notes:**

- **Watch out for argument order!** (especially `subq`)
- **No distinction between signed and unsigned int (why?)**
  - except arithmetic vs. logical shift right
- **How do you implement, “r3 = r1 + r2”**?
Some Arithmetic Operations

- **One Operand (Unary) Instructions**
  - `incq  Dest`  \( Dest = Dest + 1 \) \( \text{increment} \)
  - `decq  Dest`  \( Dest = Dest - 1 \) \( \text{decrement} \)
  - `negq  Dest`  \( Dest = -Dest \) \( \text{negate} \)
  - `notq  Dest`  \( Dest = \sim Dest \) \( \text{bitwise complement} \)

- **See textbook section 3.5.5 for more instructions:** `mulq, cqto, idivq, divq`
Arithmetic Example

```c
long simple_arith(long x, long y)
{
    long t1 = x + y;
    long t2 = t1 * 3;
    return t2;
}
```

<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>

simple_arith:

```
addq %rdi, %rsi
imulq $3, %rsi
movq %rdi, %rax
ret
```
Memory vs. registers

What is the main difference?
- Addresses vs. Names
- Big vs. Small
- Slow vs. Fast
- Dynamic vs. Static
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**

---

**Registers**

```
<table>
<thead>
<tr>
<th>Register</th>
<th>Variable</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>
```

**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>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
</tr>
<tr>
<td>0x118</td>
</tr>
<tr>
<td>0x110</td>
</tr>
<tr>
<td>0x108</td>
</tr>
<tr>
<td>0x100</td>
</tr>
</tbody>
</table>

Address 17

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></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>

The code snippet to swap the values:

```
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>123</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>456</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<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>
</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>%rdi</th>
<th>0x120</th>
</tr>
</thead>
<tbody>
<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>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
</tr>
<tr>
<td>0x118</td>
</tr>
<tr>
<td>0x110</td>
</tr>
<tr>
<td>0x108</td>
</tr>
<tr>
<td>0x100</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>123</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**  \( (R) \)  \( \text{Mem}[R] \)
  - Register \( R \) specifies the memory address
  - Aha! Pointer dereferencing in C

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

- **Displacement**  \( D(R) \)  \( \text{Mem}[R+D] \)
  - 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:**

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

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

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

  - \( (R_b, R_i) \quad \text{Mem}[R_b + R_i] \) \quad (S=1, \ D=0)
  - \( D(R_b, R_i) \quad \text{Mem}[R_b + R_i + D] \) \quad (S=1)
  - \( (R_b, R_i, S) \quad \text{Mem}[R_b + R_i \times S] \) \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>%rdx + 0xf000</td>
<td>0xf000</td>
</tr>
<tr>
<td>(%rdx,%rcx)</td>
<td>%rdx + 0x0100</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%rdx,%rcx,4)</td>
<td>%rdx + 0x0100</td>
<td>0xf100</td>
</tr>
<tr>
<td>0x80(,%rdx,2)</td>
<td>%rdx + 0x80</td>
<td>0xe80</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><code>0x8(%rdx)</code></td>
<td><code>0xf000 + 0x8</code></td>
<td><code>0xf008</code></td>
</tr>
<tr>
<td><code>(rdx,rcx)</code></td>
<td><code>0xf000 + 0x100</code></td>
<td><code>0xf100</code></td>
</tr>
<tr>
<td><code>(rdx,rcx,4)</code></td>
<td><code>0xf000 + 0x100*4</code></td>
<td><code>0xf400</code></td>
</tr>
<tr>
<td><code>0x80(,%rdx,2)</code></td>
<td><code>0xf000*2 + 0x80</code></td>
<td><code>0x1e080</code></td>
</tr>
</tbody>
</table>

#### Address Computation Examples Table

<table>
<thead>
<tr>
<th>%rdx</th>
<th>0xf000</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rcx</td>
<td>0x0100</td>
</tr>
</tbody>
</table>

- `(Rb,Ri)`: `Mem[Rb + Ri]`
- `D(Ri,S)`: `Mem[Ri*S + D]`
- `(Rb,Ri,S)`: `Mem[Rb + Ri*S]`
- `D(Rb)`: `Mem[Rb + D]`
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
  - 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`
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”**
leaq vs. movq example

Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x116</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>0x120</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>0x0</td>
<td>0x400</td>
</tr>
<tr>
<td>0x0</td>
<td>0xf</td>
</tr>
<tr>
<td>0x0</td>
<td>0x8</td>
</tr>
<tr>
<td>0x0</td>
<td>0x110</td>
</tr>
<tr>
<td>0x0</td>
<td>0x10</td>
</tr>
<tr>
<td>0x0</td>
<td>0x108</td>
</tr>
<tr>
<td>0x0</td>
<td>0x100</td>
</tr>
</tbody>
</table>

Address 0x110 = 0x100 + 0x10

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>leaq (%rdx, %rcx, 4), %rax</td>
<td>%rdx, %rcx, 4, %rax</td>
</tr>
<tr>
<td>movq (%rdx, %rcx, 4), %rbx</td>
<td>%rdx, %rcx, 4, %rbx</td>
</tr>
<tr>
<td>leaq (%rdx), %rdi</td>
<td>%rdx, %rdi</td>
</tr>
<tr>
<td>movq (%rdx), %rsi</td>
<td>%rdx, %rsi</td>
</tr>
</tbody>
</table>
leaq vs. movq example (solution)

Registers

| %rax | 0x110 |
| %rbx | 0x8  |
| %rcx | 0x4  |
| %rdx | 0x100|
| %rdi | 0x100|
| %rsi | 0x1  |

Memory

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

Address

| 0x120   |
| 0x118   |
| 0x110   |
| 0x108   |
| 0x100   |

leaq (%rdx,%rcx,4), %rax
movq (%rdx,%rcx,4), %rbx
leaq (%rdx), %rdi
movq (%rdx), %rsi
Arithmetic Example

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
Arithmetic Example

```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`  
- `addq %rdx, %rax`  
- `leaq (%rsi,%rsi,2), %rdx`  
- `salq $4, %rdx`  
- `leaq 4(%rdi,%rdx), %rcx`  
- `imulq %rcx, %rax`  
- `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
- Switches
Control Flow

```
long max(long x, long y) {
    long max;
    if (x > y) {
        max = x;
    } else {
        max = y;
    }
    return max;
}
```

```
max:
    ???
    movq %rdi, %rax
    ???
    ???
    movq %rsi, %rax
    ???
    ret
```
Control Flow

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

Conditionals and Control Flow

- **Conditional branch/jump**
  - Jump to somewhere else if some *condition* is true, otherwise execute next instruction

- **Unconditional branch/jump**
  - *Always* jump when you get to this instruction

- **Together, they can implement most control flow constructs in high-level languages:**
  - `if (condition) then {...} else {...}`
  - `while (condition) {...}`
  - `do {...} while (condition)`
  - `for (initialization; condition; iterative) {...}`
  - `switch {...}`
Jumping

- _j_ Instructions
  - Only takes _jump target_ as argument (actually just an address)
  - Conditional jump relies on special _condition code registers_
    - Splits conditional branches into 2 (or more) instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp Target</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je Target</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne Target</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>js Target</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns Target</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg Target</td>
<td>~(SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge Target</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl Target</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle Target</td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>ja Target</td>
<td>~CF &amp; ~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb Target</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)

Registers

| %rax | %8 |
| %rbx | %9 |
| %rcx | %r10 |
| %rdx | %r11 |
| %rsi | %r12 |
| %rdi | %r13 |
| %rsp | %r14 |
| %rbp | %r15 |

Instruction pointer

%rip

Current stack top

CF ZF SF OF

Condition codes
Condition Codes (Implicit Setting)

- **Condition codes:** single bit registers
  - **CF** (Carry Flag)
  - **ZF** (Zero Flag)
  - **SF** (Sign Flag)
  - **OF** (Overflow Flag)

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

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

- \( CF = 1 \) if carry out from most significant bit (unsigned overflow)
- \( ZF = 1 \) if \( t \) == 0
- \( SF = 1 \) if \( t < 0 \) (assuming signed, actually just if MSB is 1)
- \( OF = 1 \) if twos-complement (signed) overflow
  \( (a>0 \land \land b>0 \land \land t<0) \lor \lor (a<0 \land \land b<0 \land \land t>=0) \)

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

- **Explicit Setting by Compare Instruction**

  \[ \text{cmpq } Src2, Src1 \]
  \[ \text{cmpq } b, a \text{ like computing } a - b \text{ without setting destination} \]

- **CF = 1** if carry out from most significant bit (used for unsigned comparisons)
- **ZF = 1** if \(a == b\)
- **SF = 1** if \((a - b) < 0\) (as signed)
- **OF = 1** if twos complement (signed) overflow
  \((a > 0 && b < 0 && (a - b) < 0) || (a < 0 && b > 0 && (a - b) > 0)\)
Condition Codes (Explicit Setting: Test)

- **Explicit Setting by Test instruction**

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

  \[\begin{align*}
  \text{ZF} &= 1 \quad \text{if } a \& b = 0 \\
  \text{SF} &= 1 \quad \text{if } a \& b < 0
  \end{align*}\]

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

- **set_ 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>set_</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&amp;ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setge</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>setle</td>
<td>(SF^OF)</td>
<td>ZF</td>
</tr>
<tr>
<td>seta</td>
<td>~CF&amp;~ZF</td>
<td>Above (unsigned “&gt;”)</td>
</tr>
<tr>
<td>setb</td>
<td>CF</td>
<td>Below (unsigned “&lt;”)</td>
</tr>
</tbody>
</table>

CF: Carry Flag  ZF: Zero Flag  SF: Sign Flag  OF: Overflow Flag
x86-64 Integer Registers

- %rax
- %rbx
- %rcx
- %rdx
- %rsi
- %rdi
- %rsp
- %rbp
- %al
- %bl
- %cl
- %dl
- %sil
- %dil
- %spl
- %bpl
- %r8
- %r9
- %r10
- %r11
- %r12
- %r13
- %r14
- %r15
- %r8b
- %r9b
- %r10b
- %r11b
- %r12b
- %r13b
- %r14b
- %r15b

- Can reference low-order byte
Aside: movz and movs examples

\texttt{movz\_\_ \textit{Src, RegisterDest}} \quad \textit{Move with zero extension}
\texttt{movs\_\_ \textit{Src, RegisterDest}} \quad \textit{Move with sign extension}

- Use when copying a \textit{smaller} source value to a \textit{larger} destination
- Source can be memory or register; Destination must be a register
- Fills remaining bytes of dest with zeroes (\texttt{movz}) or by sign extension (\texttt{movs})

\texttt{movz\textsubscript{SD} / movs\textsubscript{SD}}:
\begin{itemize}
  \item \textit{S} – size of source (b = 1 byte, w = 2 bytes)
  \item \textit{D} – size of dest (w = 2 bytes, l = 4, q = 8)
\end{itemize}

Examples:
\texttt{movz}\underline{bq} \%al\,\,\,\%rbx
\begin{tabular}{cccccc}
\texttt{\%al} & \texttt{\%rbx} & \texttt{\%rax} & \texttt{\%r0} & \texttt{\%r0} & \texttt{\%r0} \\
\texttt{0x00} & \texttt{0x00} & \texttt{0x00} & \texttt{0x00} & \texttt{0x00} & \texttt{0x00} & \texttt{0xFF} & \texttt{0xFF} \\
\end{tabular}

\texttt{movsbl} \underline{(\%rax)}, \%rbx
\begin{itemize}
  \item Copy 1 byte from memory into 8-byte register \& sign extend it
\end{itemize}

\textbf{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 on p. 184 in the textbook.
April 18

https://xkcd.com/409/
Reading Condition Codes (Cont.)

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

- **set_ instructions do not alter remaining bytes in register**
  - Typically use movzbl to finish job
    - ”zero-extended” move, sets upper bytes 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>

What does each of these instructions do?
Reading Condition Codes (Cont.)

- **set_ 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_ instructions do not alter remaining bytes in register**
  - Typically use movzbl to finish job
    - ”zero-extended” move, sets upper bytes 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>

```assembly
cmpq %rsi, %rdi  # Compare x:y
setg %al         # Set when >
movzbl %al, %eax # Zero rest of %rax
ret
```
## Choosing instructions for conditionals

<table>
<thead>
<tr>
<th></th>
<th>cmp b,a</th>
<th>test a,b</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>je</strong></td>
<td>“Equal”</td>
<td>a == b</td>
</tr>
<tr>
<td><strong>jne</strong></td>
<td>“Not equal”</td>
<td>a != b</td>
</tr>
<tr>
<td><strong>js</strong></td>
<td>“Sign” (negative)</td>
<td>a &amp; b &lt; 0</td>
</tr>
<tr>
<td><strong>jns</strong></td>
<td>(non-negative)</td>
<td>a &amp; b &gt;= 0</td>
</tr>
<tr>
<td><strong>jg</strong></td>
<td>“Greater”</td>
<td>a &gt; b</td>
</tr>
<tr>
<td><strong>jge</strong></td>
<td>“Greater or equal”</td>
<td>a &gt;= b</td>
</tr>
<tr>
<td><strong>jl</strong></td>
<td>“Less”</td>
<td>a &lt; b</td>
</tr>
<tr>
<td><strong>jle</strong></td>
<td>“Less or equal”</td>
<td>a &lt;= b</td>
</tr>
<tr>
<td><strong>ja</strong></td>
<td>“Above” (unsigned &gt;)</td>
<td>a &gt; b</td>
</tr>
<tr>
<td><strong>jb</strong></td>
<td>“Below” (unsigned &lt;)</td>
<td>a &lt; b</td>
</tr>
</tbody>
</table>

### test a,a

<table>
<thead>
<tr>
<th></th>
<th>cmp 5,(p)</th>
<th>test a,0x1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>je</strong></td>
<td>*p == 5</td>
<td>a_{LSB} == 0</td>
</tr>
<tr>
<td><strong>jne</strong></td>
<td>*p != 5</td>
<td>a_{LSB} == 1</td>
</tr>
<tr>
<td><strong>jg</strong></td>
<td>*p &gt; 5</td>
<td></td>
</tr>
<tr>
<td><strong>jl</strong></td>
<td>*p &lt; 5</td>
<td></td>
</tr>
</tbody>
</table>
### Choosing instructions for conditionals

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>cmp b,a</th>
<th>test a,b</th>
</tr>
</thead>
<tbody>
<tr>
<td>je</td>
<td>“Equal”</td>
<td>a == b</td>
<td>a &amp; b == 0</td>
</tr>
<tr>
<td>jne</td>
<td>“Not equal”</td>
<td>a != b</td>
<td>a &amp; b != 0</td>
</tr>
<tr>
<td>js</td>
<td>“Sign” (negative)</td>
<td>a &amp; b &lt; 0</td>
<td></td>
</tr>
<tr>
<td>jns</td>
<td>(non-negative)</td>
<td>a &amp; b &gt;= 0</td>
<td></td>
</tr>
<tr>
<td>jg</td>
<td>“Greater”</td>
<td>a &gt; b</td>
<td>a &amp; b &gt; 0</td>
</tr>
<tr>
<td>jge</td>
<td>“Greater or equal”</td>
<td>a &gt;= b</td>
<td>a &amp; b &gt;= 0</td>
</tr>
<tr>
<td>jl</td>
<td>“Less”</td>
<td>a &lt; b</td>
<td>a &amp; b &lt; 0</td>
</tr>
<tr>
<td>jle</td>
<td>“Less or equal”</td>
<td>a &lt;= b</td>
<td>a &amp; b &lt;= 0</td>
</tr>
<tr>
<td>ja</td>
<td>“Above” (unsigned &gt;)</td>
<td>a &gt; b</td>
<td></td>
</tr>
<tr>
<td>jb</td>
<td>“Below” (unsigned &lt;)</td>
<td>a &lt; b</td>
<td></td>
</tr>
</tbody>
</table>

```c
if (x < 3) {
    return 1;
}
return 2;
```

```c
ifq $3, %rdi
    jge T2
T1: # x < 3:
    movq $1, %rax
    ret
T2: # !(x < 3):
    movq $2, %rax
    ret
```

### 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>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Choosing instructions for conditionals

<table>
<thead>
<tr>
<th>Condition</th>
<th>cmp b,a</th>
<th>test a,b</th>
</tr>
</thead>
<tbody>
<tr>
<td>je “Equal”</td>
<td>a == b</td>
<td>a &amp; b == 0</td>
</tr>
<tr>
<td>jne “Not equal”</td>
<td>a != b</td>
<td>a &amp; b != 0</td>
</tr>
<tr>
<td>js “Sign” (negative)</td>
<td>a &amp; b &lt; 0</td>
<td></td>
</tr>
<tr>
<td>jns (non-negative)</td>
<td>a &amp; b &gt;= 0</td>
<td></td>
</tr>
<tr>
<td>jg “Greater”</td>
<td>a &gt; b</td>
<td>a &amp; b &gt; 0</td>
</tr>
<tr>
<td>jge “Greater or equal”</td>
<td>a &gt;= b</td>
<td>a &amp; b &gt;= 0</td>
</tr>
<tr>
<td>jl “Less”</td>
<td>a &lt; b</td>
<td>a &amp; b &lt; 0</td>
</tr>
<tr>
<td>jle “Less or equal”</td>
<td>a &lt;= b</td>
<td>a &amp; b &lt;= 0</td>
</tr>
<tr>
<td>ja “Above” (unsigned &gt;)</td>
<td>a &gt; b</td>
<td></td>
</tr>
<tr>
<td>jb “Below” (unsigned &lt;)</td>
<td>a &lt; b</td>
<td></td>
</tr>
</tbody>
</table>

```c
if (x < 3 && x == y) {
    return 1;
} else {
    return 2;
}
```

```assembly
T1: # x < 3 && x == y:
    movq $1, %rax
    ret

T2: # else
    movq $2, %rax
    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>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Conditional Branch Example (with Jumps)

### Generation

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

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

### absdiff:

```
# x > y:
    movq %rdi, %rax
    subq %rsi, %rax
    ret

.L4:
# x <= y:
    movq %rsi, %rax
    subq %rdi, %rax
    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>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Conditional Branch Example (with Jumps)

Generation

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

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

<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>

absdiff:

```asm
cmpq   %rsi, %rdi    # x:y
jle    .L4           # x > y:
    movq   %rdi, %rax
    subq   %rsi, %rax
    ret
.L4:                           # x <= y:
    movq   %rsi, %rax
    subq   %rdi, %rax
    ret
```
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
Conditional Move

Conditional Move Instructions
- `cmovC src, dest`
- Move value from src to dest if condition `C` holds
- `if (Test) Dest ← Src`
- GCC tries to use them (but, only when known to be safe)

Why is this useful?
- Branches are very disruptive to instruction flow through pipelines
- Conditional moves do not require control transfer

```c
long absdiff(long x, long y)
{
    return x > y ? x - y : y - x;
}
```

```c
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
```
Compiling Loops

How to compile other loops basically similar

- Will show variations and complications in coming slides, but likely to skip all the details in class...

Most important to consider:

- When should conditionals be evaluated? (*while* vs. *do-while*)
- How much jumping is involved?
Compiling Loops

**While loop**

C/Java code:

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

Machine code:

```
loopTop:    testq %rax, %rax
            je    loopDone
            <loop body code>
            jmp   loopTop

loopDone:
```

**Do-while loop**

C/Java code:

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

Machine code:

```
loopTop:      <loop body code>
              testq %rax, %rax
              jne   loopTop

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

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

C Code

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

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

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

Goto Version

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

Detailed Walkthrough

*(skipping in class)*
General **While Loop - Translation #1**

- "Jump-to-middle" translation
- Used with \-0g

### While version
```
while (Test)
  Body
```

### Goto Version
```
goto test;
loop:
  Body
test:
  if (Test)
    goto loop;
done:
```

---

**Detailed Walkthrough** *(skipping in class)*
**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

**Detailed Walkthrough (skipping in class)**
General **While Loop - Translation #2**

**While version**

```plaintext
while (Test) 
  Body 
```

**Do-While Version**

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

- **“Do-while” conversion**
- **Used with –01**

**Goto Version**

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

**Detailed Walkthrough** *(skipping in class)*
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 `-01`
- Compare to do-while version of function (one less jump?)
- Initial conditional guards entrance to loop

Detailed Walkthrough (skipping in class)
**For Loop Form**

**General Form**

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

**Body**

```c
#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;
}
```

**Init**

```
i = 0
```

**Test**

```
i < WSIZE
```

**Update**

```
i++
```

**Detailed Walkthrough**

(kipping in class)
**For Loop → While Loop**

**For Version**

```plaintext
for (Init; Test; Update) {
    Body
}
```

**While Version**

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

**Caveat:**

- *C and Java have break and continue*
- *Conversion works fine for break*
- *But not continue: would skip doing Update, which it should do with for-loops*
- *Need goto to fix this*
- *Slides ignore this detail; textbook gets it right*
For Loop-While Conversion

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

Init

\[
i = 0
\]

Test

\[
i < \text{WSIZE}
\]

Update

\[
i++
\]

Body

\[
\{
    \text{unsigned} \text{ bit} = (x >> i) & 0x1;
    \text{result} += \text{bit};
\}
```
For Loop Do-While Conversion

Goto Version

C Code

```c
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;
}
```

- Initial test can be optimized away
Switch Statement Example

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

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

- **Missing cases**
  - Here: 4

- **Implemented with:**
  - *Jump table*
  - *Indirect jump instruction*
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>
<th>Targ1</th>
<th>Targ2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Targ0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Targ1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Targ2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Targn-1</td>
</tr>
</tbody>
</table>
```

Jump Targets

```
| Targ0: | Code Block 0 |
|------------------------------------------|
| Targ1: | Code Block 1 |
|------------------------------------------|
| Targ2: | Code Block 2 |
|------------------------------------------|
|       | • • • |
|------------------------------------------|
| Targn-1: | Code Block n-1 |
```
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 $x \leq 6$:

```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;
}
```

**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 compiler chose to not initialize `w` here.
Switch Statement Example

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

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
```

**Jump above** (like jg, but unsigned)
```
switch_eg:
    movq  %rdx, %rcx
    cmpq  $6, %rdi  # x:6
    ja    .L8
    jmp  *.L4(%rdi,8)
```
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(%rdi,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

Jump table

declaring data, not instructions

8-byte memory alignment

```
.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;
}
```
Code Blocks (x == 1)

```c
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>
Handling Fall-Through

```c
long w = 1;
  . .
switch (x) {
  . .
case 2: // .L5
  w = y/z;
  /* Fall Through */
case 3:  // .L9
  w += z;
  break;
  . .
}
```

More complicated choice than “just fall-through” forced by “migration” of

w = 1;

- Example compilation trade-off

```c
case 2:
  w = y/z;
  goto merge;
merge:
  w += z;
```
## Code Blocks (x == 2, x == 3)

```c
long w = 1;
... 
switch (x) {
  ... 
  case 2: // .L5
    w = y/z;
    /* Fall Through */
  case 3: // .L9
    w += z;
    break;
  ... 
}
```

### 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>%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)

```c
switch (x) {
    . . .
    case 5: // .L7
    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?

```
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
Conditional Operator with Jumps

C Code

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

Example:

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

Goto Version

```
n-test = !Test;
if (n-test) goto Else;
val = Then_Expr;
goto Done;
Else:
   val = Else_Expr;
Done:    ... 
```

- Ternary operator ?:
- *Test* is expression returning integer
  - = 0 interpreted as false
  - ≠ 0 interpreted as true
- Create separate code regions for then & else expressions
- Execute appropriate one

Bonus Content
(nonessential)
Conditional Operator with Jumps

C Code

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

Example:

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

Goto Version

ntest = !Test;
if (ntest) goto Else;
val = Then_Expr;
goto Done;
Else:
val = Else_Expr;
Done:
. . .

- Ternary operator ?:
- 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 (Test) Dest ← Src
- Supported in post-1995 x86 processors
- GCC tries to use them
  - But, only when known to be safe

Why is this useful?

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

C Code

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

“Goto” Version

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

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

<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>

```
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

val = Test(x) ? Hard1(x) : Hard2(x);

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

Risky Computations

val = p ? *p : 0;

- Both values get computed
- May have undesirable effects

Computations with side effects

val = x > 0 ? x*=7 : x+=3;

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