Machine-Level Programming: Basics

CMPU 224 – Computer Organization
Jason Waterman
From Last Time... Rounding

- Rounding Modes (illustrate with rounding to the nearest dollar)

<table>
<thead>
<tr>
<th></th>
<th>$1.40</th>
<th>$1.60</th>
<th>$1.50</th>
<th>$2.50</th>
<th>−$1.50</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nearest Even (default)</td>
<td>$1</td>
<td>$2</td>
<td>$2</td>
<td>$2</td>
<td>−$2</td>
</tr>
<tr>
<td>Towards zero</td>
<td>$1</td>
<td>$1</td>
<td>$1</td>
<td>$2</td>
<td>−$1</td>
</tr>
<tr>
<td>Round down (−∞)</td>
<td>$1</td>
<td>$1</td>
<td>$1</td>
<td>$2</td>
<td>−$2</td>
</tr>
<tr>
<td>Round up (+∞)</td>
<td>$2</td>
<td>$2</td>
<td>$2</td>
<td>$3</td>
<td>−$1</td>
</tr>
</tbody>
</table>
Closer Look at Round-To-Even

- Default Rounding Mode
  - Hard to get any other kind without dropping into assembly
  - All others are statistically biased
    - Sum of set of positive numbers will consistently be over- or under- estimated

- Applying to Other Decimal Places / Bit Positions
  - When exactly halfway between two possible values
    - Round so that least significant digit is even
  - E.g., round to nearest hundredth
    - 7.8949999   7.89 (Less than half way)
    - 7.8950001   7.90 (Greater than half way)
    - 7.8950000   7.90 (Half way—round up)
    - 7.8850000   7.88 (Half way—round down)
Rounding Binary Numbers

• Binary Fractional Numbers
  • “Even” when least significant bit is 0
  • “Half way” when bits to right of rounding position = 100…₂

• Examples
  • Round to nearest 1/4 (2 bits right of binary point)

<table>
<thead>
<tr>
<th>Value</th>
<th>Binary</th>
<th>Rounded</th>
<th>Action</th>
<th>Rounded Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 3/32</td>
<td>10.000₁₁₂</td>
<td>10.00₂</td>
<td>(&lt;1/2—down)</td>
<td>2</td>
</tr>
<tr>
<td>2 3/16</td>
<td>10.00₁₁₀₂</td>
<td>10.01₂</td>
<td>(&gt;1/2—up)</td>
<td>2 1/4</td>
</tr>
<tr>
<td>2 7/8</td>
<td>10.₁₁₁₀₀₂</td>
<td>11.₀₀₂</td>
<td>( 1/2—up)</td>
<td>3</td>
</tr>
<tr>
<td>2 5/8</td>
<td>10.₁₀₁₀₀₂</td>
<td>10.₁₀₂</td>
<td>( 1/2—down)</td>
<td>2 1/2</td>
</tr>
</tbody>
</table>
Floating Point in C

• C Guarantees Two Levels
  • float  single precision
  • double  double precision

• Conversions/Casting
  • Casting between int, float, and double changes bit representation
  • double/float → int
    • Truncates fractional part
    • Like rounding toward zero
    • Not defined when out of range or NaN: Generally sets to TMin
  • int → double
    • Exact conversion, as long as int has ≤ 53 bit word size
  • int → float
    • Will round according to rounding mode
Summary

• IEEE Floating Point has clear mathematical properties
• Represents numbers of form $M \times 2^E$
• One can reason about operations independent of implementation
  • As if computed with perfect precision and then rounded
• Not the same as real arithmetic
  • Violates associativity/distributivity in some corner cases
    • Overflow and inexactness of rounding
    • $(3.14+1e10)-1e10 = 0$, $3.14+(1e10-1e10) = 3.14$
  • Makes life difficult for compilers & serious numerical applications programmers
Intel x86 Processors

• Dominate laptop/desktop/server market

• Evolutionary design
  • Backwards compatible up until 8086, introduced in 1978
  • Added more features as time goes on

• Complex instruction set computer (CISC)
  • Many different instructions with many different formats
    • But, only small subset encountered with Linux programs
  • Hard to match performance of Reduced Instruction Set Computers (RISC)
  • But, Intel has done just that!
    • In terms of speed at least, less so for low power
Intel x86 Processors

- **Machine Evolution**
  - **Name** | **Date** | **Transistors** | **MHz**
  - 8086 | 1979 | 29k | 5-10
  - 386 | 1985 | 0.3M | 16-33
  - Pentium | 1993 | 3.1M | 60-300
  - Pentium 4 | 2000 | 45M | 1400-1500
  - Core 2 Duo | 2006 | 291M | 1860-2670
  - Core i7 | 2008 | 731M | 1700-3900
  - Core i7 Skylake | 2015 | 1.75B | 2800-4000

- **Added Features**
  - Instructions to support multimedia operations
  - Instructions to enable more efficient conditional operations
  - Transition from 32 bits to 64 bits
  - More cores
  - Built-in Graphics Processor

9/19/2019
CMPU 224 -- Computer Organization
Our Coverage

• IA32
  • The traditional x86

• x86-64
  • The current standard

• Presentation
  • Book covers x86-64
  • Web aside on IA32
  • We will only cover x86-64
Definitions

• **Architecture:** (also ISA: instruction set architecture) The parts of a processor design that one needs to understand or write assembly/machine code
  - Examples: instruction set specification, registers

• **Microarchitecture:** Implementation of the architecture
  - Can have many microarchitectures implement the same ISA e.g., different cache sizes and core frequencies

• **Code Forms:**
  - **Machine Code:** The byte-level programs that a processor executes
  - **Assembly Code:** A text representation of machine code

• **Example ISAs:**
  - Intel: IA32, Itanium, x86-64
  - ARM: ARMv6, ARMv7E, ARMv8
  - RISC-V: RV32I, RV64I, RV64G
Assembly/Machine Code View

Programmer-Visible State

- **PC**: Program counter
  - Address of next instruction
  - Called “RIP” (Instruction Pointer Register) in X86-64

- **Register file**
  - Heavily used program data

- **Condition codes**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

- **Memory**
  - Byte addressable array
  - Code and user data
Turning C into Object Code

- Code in files `p1.c` `p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`
  - Use basic optimizations (`-Og`) [New to recent versions of GCC]
  - Put resulting binary in file `p`

```
C program (p1.c p2.c)
```

```
Compiler (gcc -Og -S)
```

```
Asm program (p1.s p2.s)
```

```
Assembler (gcc -Og -c or as)
```

```
Object program (p1.o p2.o)
```

```
Linker (gcc or ld)
```

```
Executable program (p)
```

Static libraries (\.a)
Compiling Into Assembly

C Code (sum.c)

```c
long plus(long x, long y);
void sumstore(long x, long y, long *dest){
    long t = plus(x, y);
    *dest = t;
}
```

Generated x86-64 Assembly

```asm
sumstore:
    pushq  %rbx
    movq  %rdx, %rbx
    call   plus
    movq  %rax, (%rbx)
    popq  %rbx
    ret
```

Obtain (on a lab machine) with command

```
gcc -Og -S sum.c
```

Produces file `sum.s`

**Warning:** You will get very different results on other machines (e.g., MacOS) due to different versions of gcc and different compiler settings
Assembly Characteristics: Data Types

• “Integer” data of 1 (char), 2 (short), 4 (int), or 8 (long) bytes
  • Data values
  • Addresses (untyped pointers)

• Floating point data of 4 (float) or 8 (double) bytes
  • Stored in a different set of registers

• Code: Byte sequences encoding series of instructions

• No aggregate types such as arrays or structures
  • Just contiguously allocated bytes in memory
Assembly Characteristics: Operations

• Perform arithmetic function on registers or memory data
  • Math and logic operations

• Transfer data between memory and register
  • Load data from memory into register
  • Store register data into memory

• Transfer control
  • Unconditional jumps to/from procedures
  • Conditional branches
Object Code

Code for sumstore

0x0400595:
0x53
0x48
0x89
0xd3
0xe8
0xf2
0xff
0xff
0xff
0x48
0x89
0x03
0x5b
0xc3

- Total of 14 bytes
- Each instruction 1, 3, or 5 bytes
- Starts at address 0x0400595

- Assembler: gcc -Og -c sum.s
  - Translates .s into .o
  - Binary encoding of each instruction
  - Nearly-complete image of executable code
  - Missing linkages between code in different files

- Linker
  - Resolves references between files
  - Combines with static run-time libraries
    - E.g., code for malloc(), printf()
  - Some libraries are dynamically linked
    - Linking occurs when program begins execution

```assembly
pushq %rbx
movq %rdx, %rbx
call plus
movq %rax, (%rbx)
popq %rbx
ret
```
Machine Instruction Example

*dest = t;

movq %rax, (%rbx)

- **C Code**
  - Store value \( t \) where designated by \( \text{dest} \)

- **Assembly**
  - Move 8-byte value to memory
    - Quad words in x86-64 parlance
  - Operands:
    - \( t \): Register \( %rax \)
    - \( \text{dest} \): Register \( %rbx \)
    - \( *\text{dest} \): Memory \( M[\%rbx] \)

- **Object Code**
  - 3-byte instruction
  - Stored at address \( 0x40059e \)
Disassembling Object Code

• Disassembler
  
  `objdump -d sum`

  • Useful tool for examining object code
  • Analyzes bit pattern of series of instructions
  • Produces approximate rendition of assembly code
  • Can be run on either executable binary program or .o file

• Disassembled

```
0000000000400595 <sumstore>:
  400595:  53            push   %rbx
  400596:  48 89 d3     mov     %rdx,%rbx
  400599:  e8 f2 ff ff ff callq  400590 <plus>
  40059e:  48 89 03     mov     %rax,(%rbx)
  4005a1:  5b            pop     %rbx
  4005a2:  c3            retq
```
# x86-64 Integer Registers

<table>
<thead>
<tr>
<th>%rax</th>
<th>%r8</th>
</tr>
</thead>
<tbody>
<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>
### x86-64 Integer Registers

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

- Can reference low-order 4 bytes
### Integer Registers

<table>
<thead>
<tr>
<th>64-bit register</th>
<th>Lower 32 bits</th>
<th>Lower 16 bits</th>
<th>Lower 8 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>rax</td>
<td>eax</td>
<td>ax</td>
<td>al</td>
</tr>
<tr>
<td>rbx</td>
<td>ebx</td>
<td>bx</td>
<td>bl</td>
</tr>
<tr>
<td>rcx</td>
<td>ecx</td>
<td>cx</td>
<td>cl</td>
</tr>
<tr>
<td>rdx</td>
<td>edx</td>
<td>dx</td>
<td>dl</td>
</tr>
<tr>
<td>rsi</td>
<td>esi</td>
<td>si</td>
<td>sil</td>
</tr>
<tr>
<td>rdi</td>
<td>edi</td>
<td>di</td>
<td>dil</td>
</tr>
<tr>
<td>rbp</td>
<td>ebp</td>
<td>bp</td>
<td>bpl</td>
</tr>
<tr>
<td>rsp</td>
<td>esp</td>
<td>sp</td>
<td>spl</td>
</tr>
<tr>
<td>r8</td>
<td>r8d</td>
<td>r8w</td>
<td>r8b</td>
</tr>
<tr>
<td>r9</td>
<td>r9d</td>
<td>r9w</td>
<td>r9b</td>
</tr>
<tr>
<td>r10</td>
<td>r10d</td>
<td>r10w</td>
<td>r10b</td>
</tr>
<tr>
<td>r11</td>
<td>r11d</td>
<td>r11w</td>
<td>r11b</td>
</tr>
<tr>
<td>r12</td>
<td>r12d</td>
<td>r12w</td>
<td>r12b</td>
</tr>
<tr>
<td>r13</td>
<td>r13d</td>
<td>r13w</td>
<td>r13b</td>
</tr>
<tr>
<td>r14</td>
<td>r14d</td>
<td>r14w</td>
<td>r14b</td>
</tr>
<tr>
<td>r15</td>
<td>r15d</td>
<td>r15w</td>
<td>r15b</td>
</tr>
</tbody>
</table>
Assembly instructions

• Instruction Format:

\[ \text{ins } \text{Source, Dest} \]

• 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
    • Some registers have special uses for particular instructions
  
  • **Memory**: Consecutive bytes of memory at a given address
    • Simplest example: (%rax)
    • Various other “address modes”
    • Note: It can also be a constant without dollar sign ($)
Our first instruction: Move (mov)

• `movq Source, Dest`
  • Moves (copies) the source operand to the destination operand
  • Has many purposes
    • Load a immediate value (number) into a register
    • Copy a register value from one register into another register
    • Read a value from a memory address
    • Write a value from a memory address
    • Copy a value from one register to another

• In other hardware architectures, these operations are done with several different instructions
### movq Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src,Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imm</td>
<td>Reg</td>
<td>movq $0x4,%rax</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq $-147,(%rax)</td>
</tr>
<tr>
<td>Reg</td>
<td>Reg</td>
<td>movq %rax,%rdx</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq %rax,(%rdx)</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq (%rax),%rdx</td>
</tr>
</tbody>
</table>

**Cannot do memory-memory transfer with a single instruction**
Instruction suffixes

• Most assembly instructions take a suffix:
  • b (byte: 1 byte)
  • w (word: 2 bytes)
  • l (long word: 4 bytes)
  • q (quad word: 8 bytes)

• Often used with the low-order registers (e.g., %eax, %ax, %al)
  • movb $-17, %al
  • movl $0x4050, %eax
  • movw %bp, %sp

• In general, only the specific register bytes or memory locations are modified
  • Exception: “l” instructions that have a register as a destination will set the upper order bits to 0
Normal Memory Addressing Modes

• Normal (R) Mem[Reg[R]]
  • Register R specifies memory address
  • Pointer dereferencing in C

  \texttt{movq (\%rcx),\%rax}
Simple Memory Addressing Modes

• Normal (R) Mem[Reg[R]]
  • Register R specifies memory address
  • Pointer dereferencing in C

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

• Displacement D(R) Mem[Reg[R]+D]
  • Register R specifies start of memory region
  • Constant displacement D specifies offset

\[\text{movq } 8(%rbp), %rdx\]
Indexed Memory Addressing Modes

- Indexed \((R_b, R_i)\) \(\text{Mem}[\text{Reg}[R_b] + \text{Reg}[R_i]]\)
  - Register \(R_b\) often specifies base memory address
  - Register \(R_i\) often acts as an index
  - Often used in accessing arrays
    - \texttt{movq} (\%rcx, \%rdx), \%rax

- Scaled Indexed \((R_b, R_i, s)\) \(\text{Mem}[\text{Reg}[R_b] + \text{Reg}[R_i]*s]\)
  - \(s\) is called the scaling factor
  - Must be 1, 2, 4, 8 \(\text{(why these numbers?)}\)
Complete Memory Addressing Modes

• Most General Form

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

• D: Constant “displacement” 1, 2, or 4 bytes
• R_b: Base register: Any of 16 integer registers
• R_i: Index register: Any, except for \%rsp
• S: Scale: 1, 2, 4, or 8
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></td>
<td></td>
</tr>
<tr>
<td>(%rdx,%rcx)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%rdx,%rcx,4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x80 (,%rdx,2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table:**

<table>
<thead>
<tr>
<th>%rdx</th>
<th>0xf000</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rcx</td>
<td>0x0100</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></td>
<td></td>
</tr>
<tr>
<td>(%rdx,%rcx,4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x80(,%rdx,2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| \%rdx     | 0xf000  |
| \%rcx     | 0x0100  |
## Address Computation Examples

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

<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></td>
<td></td>
</tr>
<tr>
<td>0x80(,%rdx,2)</td>
<td></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></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><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 + 4*0x100</code></td>
<td><code>0xf400</code></td>
</tr>
<tr>
<td><code>0x80(,%rdx,2)</code></td>
<td><code>2*0xf000 + 0x80</code></td>
<td><code>0x1e080</code></td>
</tr>
</tbody>
</table>
### Address Computation Examples

<table>
<thead>
<tr>
<th>Operand</th>
<th>Address</th>
<th>Value at Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x104</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%rax)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4(%rax)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9(%rax, %rdx)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>260(%rcx, %rdx)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xFC(, %rcx, 4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%rax, %rdx, 4)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x100</td>
</tr>
<tr>
<td>%rcx</td>
<td>0x1</td>
</tr>
<tr>
<td>%rdx</td>
<td>0x3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0xFF</td>
</tr>
<tr>
<td>0x104</td>
<td>0xAB</td>
</tr>
<tr>
<td>0x108</td>
<td>0x13</td>
</tr>
<tr>
<td>0x10C</td>
<td>0x11</td>
</tr>
</tbody>
</table>
Pointers

```c
int main() {

    int x; // integer declaration of variable x
    int *xp; // int pointer declaration of variable xp

    x = 32;
    xp = &x; // Address of x
    printf("Assigning x = 32, xp = &x\n");
    printf("value of x: %d, address of x: %p\n", x, &x);
    printf("value of xp: %p, address of xp %p, dereference of xp: %d\n", xp, &xp, *xp);

    x = 64;
    printf("Assigning x = 64\n");
    printf("value of x: %d, address of x: %p\n", x, &x);
    printf("value of xp: %p, address of xp %p, dereference of xp: %d\n", xp, &xp, *xp);

    *xp = 96;
    printf("Assigning *xp = 96\n");
    printf("value of x: %d, address of x: %p\n", x, &x);
    printf("value of xp: %p, address of xp %p, dereference of xp: %d\n", xp, &xp, *xp);
}
```
Running the pointer code

Assigning x = 32, xp = &x
value of x: 32, address of x: 0x7ffdca42773c
value of xp: 0x7ffdca42773c, address of xp 0x7ffdca427740, deref of xp: 32

Assigning x = 64
value of x: 64, address of x: 0x7ffdca42773c
value of xp: 0x7ffdca42773c, address of xp 0x7ffdca427740, deref of xp: 64

Assigning *xp = 96
value of x: 96, address of x: 0x7ffdca42773c
value of xp: 0x7ffdca42773c, address of xp 0x7ffdca427740, deref of xp: 96
Address Computation Instruction

• **leaq** *Src, Dst*
  • *Src* is address mode expression
  • Set *Dst* to address denoted by expression

• Uses
  • Computing addresses without a memory reference
    • E.g., translation of `p = &x[i];`
  • Computing arithmetic expressions of the form `x + k*y`
    • `k = 1, 2, 4, or 8`

• Example

```c
long m12(long x)
{
    return x*12;
}
```

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax  # t <- x+x*2
salq $2, %rax               # return t<<2
```
Some Arithmetic Operations

- **Two Operand Instructions:**

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq</td>
<td><code>Src, Dest</code></td>
</tr>
<tr>
<td>subq</td>
<td><code>Src, Dest</code></td>
</tr>
<tr>
<td>imulq</td>
<td><code>Src, Dest</code></td>
</tr>
<tr>
<td>salq</td>
<td><code>Src, Dest</code></td>
</tr>
<tr>
<td>sarq</td>
<td><code>Src, Dest</code></td>
</tr>
<tr>
<td>shrq</td>
<td><code>Src, Dest</code></td>
</tr>
<tr>
<td>xorq</td>
<td><code>Src, Dest</code></td>
</tr>
<tr>
<td>andq</td>
<td><code>Src, Dest</code></td>
</tr>
<tr>
<td>orq</td>
<td><code>Src, Dest</code></td>
</tr>
</tbody>
</table>

  - *Also called shlq*
  - *Arithmetic shift*
  - *Logical shift*

- **Watch out for argument order, `subq` in particular**
- **No distinction between signed and unsigned int (why?)**
Some Arithmetic Operations

• One Operand Instructions

  incq  Dest  Dest = Dest + 1
  decq  Dest  Dest = Dest − 1
  negq  Dest  Dest = − Dest
  notq  Dest  Dest = ~Dest

• See book for more instructions
Machine Programming: Summary

• History of Intel processors and architectures
  • Evolutionary design leads to many quirks and artifacts

• C, assembly, machine code
  • New forms of visible state: program counter, registers, ...
  • Compiler must transform statements, expressions, procedures into low-level instruction sequences

• Assembly Basics: Registers, operands, move
  • The x86-64 move instructions cover wide range of data movement forms

• Arithmetic
  • C compiler will figure out different instruction combinations to carry out computation