x86 Data Access and Operations
Machine-Level Representations

Prior lectures
- Data representation

This lecture
- Program representation
- Encoding is architecture dependent
  - We will focus on the Intel x86-64 or x64 architecture
  - Prior edition used IA32
Intel x86

Evolutionary design starting in 1978 with 8086
- i386 in 1986: First 32-bit Intel CPU (IA32)
- Pentium4E in 2004: First 64-bit Intel CPU (x86-64)
  - Adopted from AMD Opteron (2003)
- Core 2 in 2006: First multi-core Intel CPU
- Core 7 in 2008: Current generation
- New features and instructions added over time
  - Vector operations for multimedia
  - Memory protection for security
  - Conditional data movement instructions for performance
  - Expanded address space for scaling
- Many obsolete features

Complex Instruction Set Computer (CISC)
- Many different instructions with many different formats
- But we’ll only look at a small subset
2015
Core i7 Broadwell
How do you program it?

Initially, no compilers or assemblers

Machine code generated by hand!

- Error-prone
- Time-consuming
- Hard to read and write
- Hard to debug
Assemblers

Assign mnemonics to machine code

- Assembly language for specifying machine instructions
- Names for the machine instructions and registers
  - `movq %rax, %rcx`
- There is no standard for x86 assemblers
  - Intel assembly language
  - AT&T Unix assembler
  - Microsoft assembler
  - GNU uses Unix style with its assembler `gas`

Even with the advent of compilers, assembly still used

- Early compilers made big, slow code
- Operating Systems were written mostly in assembly, into the 1980s
- Accessing new hardware features before compiler has a chance to incorporate them
Compiling Into Assembly

C Code (sum.c)

```c
long plus(long x, long y);

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

Compiled using basic optimizations (-Og)

gcc -Og -S sum.c

Generated x86-64 assembly

```assembly
sumstore:
    pushq  %rbx
    movq  %rdx, %rbx
    call  plus
    movq  %rax, (%rbx)
    popq  %rbx
    ret
```
Assembly Programmer’s View

Programmer-Visible State

- **RIP**
  - Instruction Pointer or Program Counter
  - Address of next instruction

- **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, user data, OS data
- Includes stack used to support procedures
64-bit memory map

48-bit canonical addresses to make page-tables smaller

Kernel addresses have high-bit set

user stack (created at runtime)

%esp (stack pointer)

memory mapped region for shared libraries

run-time heap (managed by malloc)

brk

read/write segment (.data, .bss)

read-only segment (.init, .text, .rodata)

unused

reserved for kernel (code, data, heap, stack)

memory invisible to user code

loaded from the executable file

cat /proc/self/maps
Registers

Special memory not part of main memory

- Located on CPU
- Used to store temporary values
- Typically, data is loaded into registers, manipulated or used, and then written back to memory
### 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>

- Accessible as 8, 16, 32, 64 bits

Format different since registers added with x86-64
64-bit registers

Multiple access sizes \%rax, \%rbx, \%rcx, \%rdx

\%ah, \%al : low order bytes (8 bits)
\%ax : low word (16 bits)
\%eax : low “double word” (32 bits)
\%rax : quad word (64 bits)

Similar access for \%rdi, \%rsi, \%rbp, \%rsp
64-bit registers

**Multiple access sizes %r8, %r9, ... , %r15**

- %r8b : low order byte (8 bits)
- %r8w : low word (16 bits)
- %r8d : low "double word" (32 bits)
- %r8  : quad word (64 bits)
Register evolution

The x86 architecture initially “register poor”

- Few general purpose registers (8 in IA32)
  - Initially, driven by the fact that transistors were expensive
  - Then, driven by the need for backwards compatibility for certain instructions pusha (push all) and popa (pop all) from 80186

- Other reasons
  - Makes context-switching amongst processes easy (less register-state to store)
  - Add fast caches instead of more registers (L1, L2, L3 etc.)
Instruction types

A typical instruction acts on 2 or more operands of a particular width

- `addq %rcx, %rdx` adds the contents of `rcx` to `rdx`
- “`addq`” stands for add “quad word”
- Size of the operand denoted in instruction
- Why “quad word” for 64-bit registers?
  - Baggage from 16-bit processors

Now we have these crazy terms

- 8 bits = byte = `addb`
- 16 bits = word = `addw`
- 32 bits = double or long word = `addl`
- 64 bits = quad word = `addq`
# C types and x86-64 instructions

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Intel x86-64 type</th>
<th>GAS suffix</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>byte</td>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>word</td>
<td>w</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>double word</td>
<td>l</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>quad word</td>
<td>q</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>single precision</td>
<td>s</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>double precision</td>
<td>l</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>extended precision</td>
<td>t</td>
<td>10/16</td>
</tr>
<tr>
<td>pointer</td>
<td>quad word</td>
<td>q</td>
<td>8</td>
</tr>
</tbody>
</table>
Instruction operands

Example instruction

\texttt{movq } \textit{Source}, \textit{Dest}

Three operand types

\begin{itemize}
  \item Immediate
    \begin{itemize}
      \item Constant integer data
      \item Like C constant, but preceded by $\$
      \item e.g., $\$0x400$, $\$-533$
      \item Encoded directly into instructions
    \end{itemize}
  \item Register: One of 16 integer registers
    \begin{itemize}
      \item Example: $\%rax$, $\%r13$
      \item Note $\%rsp$ reserved for special use
    \end{itemize}
  \item Memory: a memory address
    \begin{itemize}
      \item There are many modes for addressing memory
      \item Simplest example: ($\%rax$)
    \end{itemize}
\end{itemize}
Operand examples using \texttt{mov}

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{Imm}</td>
<td>\textit{Reg}</td>
<td>\texttt{movq $0x4,%rax}</td>
</tr>
<tr>
<td></td>
<td>\texttt{Mem}</td>
<td>\texttt{movq $-147, (%rax)}</td>
</tr>
<tr>
<td>\textit{Reg}</td>
<td>\textit{Reg}</td>
<td>\texttt{movq %rax, %rdx}</td>
</tr>
<tr>
<td></td>
<td>\textit{Mem}</td>
<td>\texttt{movq %rax, (%rdx)}</td>
</tr>
<tr>
<td>\textit{Mem}</td>
<td>\textit{Reg}</td>
<td>\texttt{movq (%rax), %rdx}</td>
</tr>
</tbody>
</table>

- Memory-memory transfers cannot be done with single instruction
Immediate mode

Immediate has only one mode

- Form: $Imm$
- Operand value: Imm

- movq $0x8000,%rax
- movq $array,%rax

» int array[30]; /* array = global variable stored at 0x8000 */
Register mode

Register has only one mode

- Form: $E_a$
- Operand value: $R[E_a]$
  - movq $%rcx, %rax$

```
%rax
%rcx 0x0030
%rdx
```

Main memory

```
0x8000
```
Memory modes

Memory has multiple modes

- **Absolute**
  - specify the address of the data

- **Indirect**
  - use register to calculate address

- **Base + displacement**
  - use register plus absolute address to calculate address

- **Indexed**
  - Indexed
    - Add contents of an index register
  - Scaled index
    - Add contents of an index register scaled by a constant
Memory modes

Memory mode: Absolute

- Form: Imm
- Operand value: M[Imm]
  - movq 0x8000, %rax
  - movq array, %rax

```
long array[30]; /* global variable at 0x8000 */
```
Memory modes

Memory mode: Indirect

- Form: \((E_a)\)
- Operand value: \(M[R[E_a]]\)
  - Register \(E_a\) specifies the memory address
  - `movq (%rcx), %rax`

Main memory

\[
\begin{array}{c}
\%rax \\
\%rcx \quad 0x8000 \\
\%rax \\
0x8000 \\
\end{array}
\]
Memory modes

Memory mode: Base + Displacement

- **Form:** $\text{Imm}(E_b)$
- **Operand value:** $M[\text{Imm}+R[E_b]]$
  - Register $E_b$ specifies start of memory region
  - $\text{Imm}$ specifies the offset/displacement
- $\text{movq } 16(\%rcx), \%rax$

![Diagram of memory layout with registers and memory addresses](image)
Memory modes

Memory mode: Scaled indexed

- Most general format
- Used for accessing structures and arrays in memory
- Form: $\text{Imm}(E_b, E_i, S)$
- Operand value: $M[\text{Imm} + R[E_b] + S*R[E_i]]$
  - $E_b$ specifies start of memory region
  - $E_i$ holds index
  - $S$ is integer scale (1, 2, 4, 8)
  - `movq 8(%rdx,%rcx,8),%rax`

Main memory

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000</td>
</tr>
<tr>
<td>0x8008</td>
</tr>
<tr>
<td>0x8020</td>
</tr>
<tr>
<td>0x8028</td>
</tr>
<tr>
<td>0x8010</td>
</tr>
<tr>
<td>0x8018</td>
</tr>
<tr>
<td>0x8000</td>
</tr>
<tr>
<td>0x8008</td>
</tr>
<tr>
<td>0x8010</td>
</tr>
<tr>
<td>0x8018</td>
</tr>
<tr>
<td>0x8020</td>
</tr>
<tr>
<td>0x8028</td>
</tr>
<tr>
<td>0x8030</td>
</tr>
<tr>
<td>0x8038</td>
</tr>
</tbody>
</table>

- %rax
- %rcx 0x03
- %rdx 0x8000
Addressing Mode Examples

```
addl 12(%rbp),%ecx  
Add the double word at address rbp + 12 to ecx

movb (%rax,%rcx),%dl  
Load the byte at address rax + rcx into dl

subq %rdx,(%rcx,%rax,8)  
Subtract rdx from the quad word at address rcx+(8*rax)

incw 0xA,(%rcx,8)  
Increment the word at address 0xA+(8*rcx)
```

Also note: We do not put ‘$’ in front of constants when they are addressing indexes, only when they are literals.
## 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>
Practice Problem 3.1

<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>0x108</td>
<td>0xAB</td>
</tr>
<tr>
<td>0x110</td>
<td>0x13</td>
</tr>
<tr>
<td>0x118</td>
<td>0x11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operand</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x100</td>
</tr>
<tr>
<td>0x108</td>
<td>0x108</td>
</tr>
<tr>
<td>$0x108</td>
<td>0x108</td>
</tr>
<tr>
<td>(%rax)</td>
<td>0xFF</td>
</tr>
<tr>
<td>8(%rax)</td>
<td>0xAB</td>
</tr>
<tr>
<td>13(%rax, %rdx)</td>
<td>0x13</td>
</tr>
<tr>
<td>260(%rcx, %rdx)</td>
<td>0xAB</td>
</tr>
<tr>
<td>0xF8(%rcx, 8)</td>
<td>0xFF</td>
</tr>
<tr>
<td>(%rax, %rdx, 8)</td>
<td>0x11</td>
</tr>
</tbody>
</table>
Example: swap()

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

**Memory**

**Registers**

- %rdi
- %rsi
- %rax
- %rdx

**Register** | **Value**
--- | ---
%rdi | xp
%rsi | yp
%rax | t0
%rdx | t1

**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></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>

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></td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>123</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>456</td>
</tr>
</tbody>
</table>

Address

<table>
<thead>
<tr>
<th>0x120</th>
<th>0x118</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x110</td>
<td>0x108</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
# Understanding Swap()

The swap function can be implemented using the following assembly code:

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

### 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>0x120</th>
<th>0x118</th>
<th>0x110</th>
<th>0x108</th>
<th>0x100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>123</td>
<td></td>
<td></td>
<td></td>
<td>456</td>
</tr>
</tbody>
</table>
Understanding Swap()

Registers

| %rdi  | 0x120 |
| %rsi  | 0x100 |
| %rax  | 123   |
| %rdx  | 456   |

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

<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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>456</td>
</tr>
<tr>
<td>123</td>
</tr>
</tbody>
</table>

swap:

```plaintext
movq  (%rdi), %rax  # t0 = *xp
movq  (%rsi), %rdx  # t1 = *yp
movq  %rdx, (%rdi)  # *xp = t1
movq  %rax, (%rsi)  # *yp = t0
ret
```
Practice Problem 3.5

A function has this prototype:

```c
void decode(long *xp, long *yp, long *zp);
```

Here is the body of the code in assembly language:

```assembly
/* xp in %rdi, yp in %rsi, zp in %rdx */
1 movq (%rdi), %r8
2 movq (%rsi), %rcx
3 movq (%rdx), %rax
4 movq %r8,(%rsi)
5 movq %rcx,(%rdx)
6 movq %rax,(%rdi)
```

Write C code for this function

```c
void decode(long *xp, long *yp, long *zp) {
    long x = *xp; /* Line 1 */
    long y = *yp; /* Line 2 */
    long z = *zp; /* Line 3 */
    *yp = x;      /* Line 6 */
    *zp = y;      /* Line 8 */
    *xp = z;      /* Line 7 */
    return z;
}
```
Practice Problem

Suppose an array in C is declared as a global variable:

```c
long array[34];
```

Write some assembly code that:
- sets rsi to the address of array
- sets rbx to the constant 9

Use scaled index memory mode
Practice Problem

Suppose an array in C is declared as a global variable:

```c
long array[34];
```

Write some assembly code that:
- sets `rsi` to the address of `array`
- sets `rbx` to the constant 9

Use scaled index memory mode

```assembly
movl $array, %rsi
movl $0x9, %rbx
movl (%rsi, %rbx, 8), %rax
```
Arithmetic and Logical Operations
Load address

**Load Effective Address (Quad)**

\[
\text{leaq } S, D \implies D \leftarrow &S
\]

- Loads the *address* of \( S \) in \( D \), not the *contents*
  - \( \text{leaq} \ (\%\text{rax}), \%\text{rdx} \)
  - Equivalent to \( \text{movq} \ \%\text{rax}, \%\text{rdx} \)

- Destination must be a register

- Used to compute addresses without a memory reference
  - e.g., translation of \( p = &x[i] \);
Load address

`leaq S, D ⇒ D ← &S`

- Commonly used by compiler to do simple arithmetic
  - If `%rdx = x`,
    - `leaq 7(%rdx, %rdx, 4), %rdx ⇒ 5x + 7`
    - Multiply and add all in one instruction

- Example

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

Converted to ASM by compiler:

```asm
leaq (%rdi,%rdi,2), %rax # t ← x+x*2
salq $2, %rax # return t<<2
```
Practice Problem 3.6

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result in %rdx</th>
</tr>
</thead>
<tbody>
<tr>
<td>leaq 6(%rax), %rdx</td>
<td>x+6</td>
</tr>
<tr>
<td>leaq (%rax, %rcx), %rdx</td>
<td>x+y</td>
</tr>
<tr>
<td>leaq (%rax, %rcx, 4), %rdx</td>
<td>x+4y</td>
</tr>
<tr>
<td>leaq 7(%rax, %rax, 8), %rdx</td>
<td>9x+7</td>
</tr>
<tr>
<td>leaq 0xA(, %rcx, 4), %rdx</td>
<td>4y+10</td>
</tr>
<tr>
<td>leaq 9(%rax, %rcx, 2), %rdx</td>
<td>x+2y+9</td>
</tr>
</tbody>
</table>
Two Operand Arithmetic Operations

A little bit tricky

- Second operand is both a source and destination
- A bit like C operators ‘+=’, ‘-=', etc.
- Max shift is 64 bits, so k is either an immediate byte, or register (e.g. %cl where %cl is byte 0 of register %rcx)
- No distinction between signed and unsigned int (why?)

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq</td>
<td>S, D</td>
</tr>
<tr>
<td></td>
<td>D = D + S</td>
</tr>
<tr>
<td>subq</td>
<td>S, D</td>
</tr>
<tr>
<td></td>
<td>D = D - S</td>
</tr>
<tr>
<td>imulq</td>
<td>S, D</td>
</tr>
<tr>
<td></td>
<td>D = D * S</td>
</tr>
<tr>
<td>salq</td>
<td>S, D</td>
</tr>
</tbody>
</table>
|        | D = D << S           | Also called shlq
| sarq   | S, D                 |
|        | D = D >> S           | Arithmetic shift right (sign extend)
| shrq   | S, D                 |
|        | D = D >> S           | Logical shift right (zero fill)
| xorq   | S, D                 |
|        | D = D ^ S            |
| andq   | S, D                 |
|        | D = D & S            |
| orq    | S, D                 |
|        | D = D | S              |
## One Operand Arithmetic Operations

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>incq</td>
<td>D = D + 1</td>
</tr>
<tr>
<td>decq</td>
<td>D = D - 1</td>
</tr>
<tr>
<td>negq</td>
<td>D = - D</td>
</tr>
<tr>
<td>notq</td>
<td>D = ~D</td>
</tr>
</tbody>
</table>

See book for more instructions
Practice Problem 3.9

```c
long shift_left4_rightn(long x, long n)
{
    x <<= 4;
    x >>= n;
    return x;
}

_shift_left4_rightn:
    movq %rdi, %rax ; get x
    salq $4, %rax ; x <<= 4;
    movq %esi, %rcx ; get n
    shrq %cl, %rax ; x >>= n;
    ret
```
Practice Problem 3.8

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0xFF</td>
</tr>
<tr>
<td>0x108</td>
<td>0xAB</td>
</tr>
<tr>
<td>0x110</td>
<td>0x13</td>
</tr>
<tr>
<td>0x118</td>
<td>0x11</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>Instruction</th>
<th>Destination address</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq %rcx, (%rax)</td>
<td>0x100</td>
<td>0x100</td>
</tr>
<tr>
<td>subq %rdx, 8(%rax)</td>
<td>0x108</td>
<td>0xA8</td>
</tr>
<tr>
<td>imulq $16, (%rax, %rdx, 8)</td>
<td>0x118</td>
<td>0x110</td>
</tr>
<tr>
<td>incq 16(%rax)</td>
<td>0x110</td>
<td>0x14</td>
</tr>
<tr>
<td>decq %rcx</td>
<td>%rcx</td>
<td>0x0</td>
</tr>
<tr>
<td>subq %rdx, %rax</td>
<td>%rax</td>
<td>0xFD</td>
</tr>
</tbody>
</table>
Arithmetic Expression 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;
}
```

Compiler trick to generate efficient code

<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>
Practice Problem 3.10

What does this instruction do?

```
xorq %rdx, %rdx
```

Zeros out register

How might it be different than this instruction?

```
movq $0, %rdx
```

3-byte instruction versus 7-byte
Null bytes encoded in instruction
# Exam practice

## Chapter 3 Problems (Part 1)

<table>
<thead>
<tr>
<th>Problem</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>x86 operands</td>
</tr>
<tr>
<td>3.2,3.3</td>
<td>instruction operand sizes</td>
</tr>
<tr>
<td>3.4</td>
<td>instruction construction</td>
</tr>
<tr>
<td>3.5</td>
<td>disassemble to C</td>
</tr>
<tr>
<td>3.6</td>
<td>leaq</td>
</tr>
<tr>
<td>3.7</td>
<td>leaq disassembly</td>
</tr>
<tr>
<td>3.8</td>
<td>operations in x86</td>
</tr>
<tr>
<td>3.9</td>
<td>fill in x86 from C</td>
</tr>
<tr>
<td>3.10</td>
<td>fill in C from x86</td>
</tr>
<tr>
<td>3.11</td>
<td>xorq</td>
</tr>
</tbody>
</table>
Extra slides
Definitions

Architecture or instruction set architecture (ISA)
- Instruction specification, registers
- Examples: x86 IA32, x86-64, ARM

Microarchitecture
- Implementation of the architecture
- Examples: cache sizes and core frequency

Machine code (or object code)
- Byte-level programs that a processor executes

Assembly code
- A text representation of machine code
Disassembling Object Code

Disassembled

```
00000000000400595 <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
```

Disassembler

`objdump -d sumstore`

Useful tool for examining object code
Analyze bit pattern of series of instructions
Produces approximate rendition of assembly code
Can be run on either `a.out` (complete executable) or `.o` file
Alternate Disassembly

Disassembled

Dump of assembler code for function sumstore:
0x0000000000400595 <+0>: push %rbx
0x0000000000400596 <+1>: mov %rdx,%rbx
0x0000000000400599 <+4>: callq 0x400590 <plus>
0x000000000040059e <+9>: mov %rax,(%rbx)
0x00000000004005a1 <+12>: pop %rbx
0x00000000004005a2 <+13>: retq

Within gdb Debugger

gdb sum
disassemble sumstore
Disassemble procedure
x/14xb sumstore
Examine the 14 bytes starting at sumstore

http://thefengs.com/wuchang/courses/cs201/class/05/math_examples.c
Object Code

**Code for sumstore**

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

0x0400595:

0x53
0x48
0x89
0xd3
0xe8
0xf2
0xff
0xff
0xff
0x48
0x89
0x03
0x5b
0xc3
### Some History: IA32 Registers

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

- **%eax, %ax** (accumulator)
- **%ecx, %cx** (counter)
- **%edx, %dx** (data)
- **%ebx, %bx** (base)
- **%esi, %si** (source)
- **%edi, %di** (index)
- **%esp, %sp** (destination)
- **%ebp, %bp** (base pointer)

The virtual registers (%ah, %al) are backwards compatible with 16-bit systems.
Memory modes

Memory mode: Scaled indexed

- Absolute, indirect, base+displacement, indexed are simply special cases of Scaled indexed
- More special cases
  - $(E_b, E_i, S) M[R[E_b] + R[E_i]*S]$
  - $(E_b, E_i) M[R[E_b] + R[E_i]]$
  - $(, E_i, S) M[R[E_i]*S]$
  - Imm$(, E_i, S) M[Imm + R[E_i]*S]$
Alternate mov instructions

Not all move instructions are equivalent

- There are three byte move instructions and each produces a different result

  - movb only changes specific byte
  - movsbl does sign extension
  - movzbl sets other bytes to zero

Assumptions: %dh = 0x8D, %rax = 0x98765432

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source</th>
<th>Destination</th>
<th>%rax</th>
</tr>
</thead>
<tbody>
<tr>
<td>movb %dh, %al</td>
<td>%dh</td>
<td>%al</td>
<td>0x9876548D</td>
</tr>
<tr>
<td>movsbl %dh, %rax</td>
<td>%dh</td>
<td>%rax</td>
<td>FFFFFFFF8D</td>
</tr>
<tr>
<td>movzbl %dh, %rax</td>
<td>%dh</td>
<td>%rax</td>
<td>0000008D</td>
</tr>
</tbody>
</table>
## Data Movement Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl</td>
<td>D ← S</td>
<td>Move double word</td>
</tr>
<tr>
<td>movw</td>
<td>D ← S</td>
<td>Move word</td>
</tr>
<tr>
<td>movb</td>
<td>D ← S</td>
<td>Move byte</td>
</tr>
<tr>
<td>movsbl</td>
<td>D ← SignExtend(S)</td>
<td>Move sign-extended byte</td>
</tr>
<tr>
<td>movzbl</td>
<td>D ← ZeroExtend(S)</td>
<td>Move zero-extended byte</td>
</tr>
</tbody>
</table>