ASSMEBLY BASICS

Spring, 2017
Euiseong Seo
(euiseong@skku.edu)

This Powerpoint slides are modified from its original version available at http://www.cs.cmu.edu/afs/cs/academic/class/15213-s09/www/lectures/ppt-sources/
- Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
Intel x86 Processors

- Totally dominate computer 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!
Intel x86 Evolution: Milestones

- **Name**  |  **Date**  |  **Transistors**  |  **MHz**  
- 8086     | 1978      | 29K             | 5-10     
  - First 16-bit Intel processor. Basis for IBM PC & DOS
  - 1MB address space
- 386      | 1985      | 275K            | 16-33    
  - First 32 bit Intel processor, referred to as IA32
  - Added “flat addressing”, capable of running Unix
- Pentium 4E | 2004    | 125M            | 2800-3800  
  - First 64-bit Intel x86 processor, referred to as x86-64
- Core 2   | 2006     | 291M            | 1060-3500 
  - First multi-core Intel processor
- Core i7   | 2008     | 731M            | 1700-3900 
  - Four cores
### Intel x86 Processors: Overview

<table>
<thead>
<tr>
<th>Architectures</th>
<th>Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>X86-16</td>
<td>8086</td>
</tr>
<tr>
<td></td>
<td>286</td>
</tr>
<tr>
<td>X86-32/IA32</td>
<td>386</td>
</tr>
<tr>
<td></td>
<td>486</td>
</tr>
<tr>
<td></td>
<td>Pentium</td>
</tr>
<tr>
<td></td>
<td>Pentium MMX</td>
</tr>
<tr>
<td>MMX</td>
<td></td>
</tr>
<tr>
<td>SSE</td>
<td>Pentium III</td>
</tr>
<tr>
<td>SSE2</td>
<td>Pentium 4</td>
</tr>
<tr>
<td>SSE3</td>
<td>Pentium 4E</td>
</tr>
<tr>
<td>X86-64 / EM64t</td>
<td>Pentium 4F</td>
</tr>
<tr>
<td>SSE4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Core 2 Duo</td>
</tr>
<tr>
<td></td>
<td>Core i7</td>
</tr>
</tbody>
</table>

IA: often redefined as latest Intel architecture
**Intel x86 Processors: Overview**

**Machine Evolution**
- 386 1985 0.3M
- Pentium 1993 3.1M
- Pentium/MMX 1997 4.5M
- PentiumPro 1995 6.5M
- Pentium III 1999 8.2M
- Pentium 4 2001 42M
- Core 2 Duo 2006 291M
- Core i7 2008 731M

**Added Features**
- Instructions to support multimedia operations
- Instructions to enable more efficient conditional operations
- Transition from 32 bits to 64 bits
- More cores
**x86 Clones: Advanced Micro Devices**

- **Historically**
  - AMD has followed just behind Intel
  - A little bit slower, a lot cheaper

- **Then**
  - Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
  - Built Opteron: tough competitor to Pentium 4
  - Developed x86-64, their own extension to 64 bits

- **Recent Years**
  - Intel got its act together
    - Leads the world in semiconductor technology
  - AMD has fallen behind
    - Relies on external semiconductor manufacturer
Intel’s 64-Bit History

- **2001: Intel Attempts Radical Shift from IA32 to IA64**
  - Totally different architecture (Itanium)
  - Executes IA32 code only as legacy
  - Performance disappointing

- **2003: AMD Steps in with Evolutionary Solution**
  - x86-64 (now called “AMD64”)

- **Intel Felt Obligated to Focus on IA64**
  - Hard to admit mistake or that AMD is better

- **2004: Intel Announces EM64T extension to IA32**
  - Extended Memory 64-bit Technology
  - Almost identical to x86-64!

- **All but low-end x86 processors support x86-64**
  - But, lots of code still runs in 32-bit mode
Our Coverage

- IA32
  - The traditional x86
  - RIP, Spring 2015

- x86-64
  - The standard
  - `shark> gcc hello.c`
  - `shark> gcc -m64 hello.c`

- Presentation
  - Book covers x86-64
  - We will only cover x86-64
Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
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
  - Examples: cache sizes and core frequency

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

- **Example ISAs**:
  - Intel: x86, IA32, Itanium, x86-64
  - ARM: Used in almost all mobile phones
Programmer-Visible State

- **PC**: Program counter
  - Address of next instruction
  - Called “RIP” (x86-64)
- **Register file**
  - Heavily used program data
- **Condition codes**
  - Store status information about most recent arithmetic operation
  - Used for conditional branching

**Memory**

- Byte addressable array
- Code, user data, (some) OS data
- Includes stack used to support procedures
- Code in files p1.c p2.c
- Compile with command: gcc –Og p1.c p2.c -o p
  - Use basic optimizations (-Og)
  - Put resulting binary in file p

**Turning C into Object Code**

- **text**
  - C program (p1.c p2.c)
    - Compiler (gcc –Og –S)
- **text**
  - Asm program (p1.s p2.s)
    - Assembler (gcc or as)
- **binary**
  - Object program (p1.o p2.o)
    - Linker (gcc or ld)
- **binary**
  - Executable program (p)

Static libraries (.a)
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

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

Obtain with command

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

Produces file `sum.s`
Assembly Characteristics: Data Types

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

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

- 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 register or memory data
- 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
- 0x48
- 0x89
- 0x03
- 0x5b
- 0xc3

**Assembler**
- 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

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

**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

\[
*\text{dest} = t;
\]

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

\[
0x40059e: \quad 48 \quad 89 \quad 03
\]
Disassembling Object Code

- **Disassembler**
  - `objdump -d p`
  - Useful tool for examining object code
  - Analyzes bit pattern of series of instructions
  - Produces approximate rendition of assembly code
  - Can be run on either `a.out` (complete executable) or `.o` file

```
00401040 <_sum>:
  0:  55  push  %ebp
  1:  89 e5  mov  %esp,%ebp
  3:  8b 45 0c  mov  0xc(%ebp),%eax
  6:  03 45 08  add  0x8(%ebp),%eax
  9:  89 ec  mov  %ebp,%esp
 b:  5d  pop  %ebp
 c:  c3  ret
 d:  8d 76 00  lea  0x0(%esi),%esi
```
Within **gdb** Debugger

- **gdb p**
- disassemble **sum**
- Disassemble procedure
- **x/13b sum**
- Examine the 13 bytes starting at **sum**

### Disassembled

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>0x401040</th>
<th>0x401041</th>
<th>0x401043</th>
<th>0x401046</th>
<th>0x401049</th>
<th>0x40104b</th>
<th>0x40104c</th>
<th>0x40104d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040</td>
<td>push %ebp</td>
<td>0x00000005</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0x401041</td>
<td>mov %esp, %ebp</td>
<td>0x00000089</td>
<td>0x0000000e</td>
<td>0x0000000c</td>
<td>0x00000003</td>
<td>0x00000008</td>
<td>0x000000ec</td>
<td>0x0000000c</td>
<td>0x0000005d</td>
</tr>
<tr>
<td>0x401043</td>
<td>mov 0xc(%ebp), %eax</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0x401046</td>
<td>add 0x8(%ebp), %eax</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0x401049</td>
<td>mov %ebp, %esp</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0x40104b</td>
<td>pop %ebp</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0x40104c</td>
<td>ret</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0x40104d</td>
<td>lea 0x0(%esi), %esi</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>

### Object

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040</td>
<td>0x55</td>
</tr>
<tr>
<td>0x401041</td>
<td>0x89</td>
</tr>
<tr>
<td>0x401043</td>
<td>0xe5</td>
</tr>
<tr>
<td>0x401046</td>
<td>0x8b</td>
</tr>
<tr>
<td>0x401049</td>
<td>0x45</td>
</tr>
<tr>
<td>0x40104b</td>
<td>0x0c</td>
</tr>
<tr>
<td>0x40104c</td>
<td>0x03</td>
</tr>
<tr>
<td>0x40104d</td>
<td>0x45</td>
</tr>
<tr>
<td>0x40104f</td>
<td>0x08</td>
</tr>
<tr>
<td>0x401051</td>
<td>0x89</td>
</tr>
<tr>
<td>0x401053</td>
<td>0xec</td>
</tr>
<tr>
<td>0x401055</td>
<td>0x5d</td>
</tr>
<tr>
<td>0x401057</td>
<td>0xc3</td>
</tr>
</tbody>
</table>
Anything that can be interpreted as executable code

Disassembler examines bytes and reconstructs assembly source

```bash
% objdump -d WINWORD.EXE

WINWORD.EXE: file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000: 55  push  %ebp
30001001: 8b ec  mov  %esp,%ebp
30001003: 6a ff  push  $0xffffffff
30001005: 68 90 10 00 30  push  $0x30001090
3000100a: 68 91 dc 4c 30  push  $0x304cdc91
```
Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
Data Formats

Intel uses “word” to refer to a 16-bit data type
- 32-bit quantities as double words and 64-bit quantities as quad words

<table>
<thead>
<tr>
<th>C declaration</th>
<th>Intel data type</th>
<th>GAS suffix</th>
<th>Size (B)</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>unsigned</td>
<td>Double word</td>
<td>l</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>Quad word</td>
<td>q</td>
<td>8</td>
</tr>
<tr>
<td>unsigned long</td>
<td>Quad word</td>
<td>q</td>
<td>8</td>
</tr>
<tr>
<td>char *</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>10/12</td>
</tr>
</tbody>
</table>

mov: movb, movw, movq, movl
### x86-64 Integer Registers

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%ebx</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
</tr>
<tr>
<td>%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

**General Purpose:**
- `%eax`  
- `%ecx`  
- `%edx`  
- `%ebx`  
- `%esi`  
- `%edi`  
- `%esp`  
- `%ebp`

**Origin (mostly obsolete):**
- `%ax`  
- `%cx`  
- `%dx`  
- `%bx`  
- `%si`  
- `%di`  
- `%sp`  
- `%bp`

**Virtual Registers:**
- `%ah`, `%al`
- `%ch`, `%cl`
- `%dh`, `%dl`
- `%bh`, `%bl`

**16-bit virtual registers (backwards compatibility):**
- `%ah`, `%al`
- `%ch`, `%cl`
- `%dh`, `%dl`
- `%bh`, `%bl`
**Integer Register Anatomy**

<table>
<thead>
<tr>
<th>eax</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ax</th>
</tr>
</thead>
<tbody>
<tr>
<td>ah</td>
</tr>
</tbody>
</table>

- eax
- ax
- ah
- al
Moving Data

\texttt{movq Source, Dest:}

**Operand Types**

- **Immediate:** Constant integer data
  - Example: $0x400, -533$
  - Like C constant, but prefixed with `$`.
  - Encoded with 1, 2, or 4 bytes.

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

- **Memory:** 8 consecutive bytes of memory at address given by register.
  - Simplest example: (\%\texttt{rax})
  - Various other “address modes”
## MOVQ Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>Example</th>
<th>C analogy</th>
</tr>
</thead>
<tbody>
<tr>
<td>immediate (Imm)</td>
<td>register</td>
<td>movq $0x4,%rax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td></td>
<td>memory</td>
<td>movq $-147,(%rax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td>register</td>
<td>register</td>
<td>movq %rax,%rdx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td></td>
<td>memory</td>
<td>movq %rax,(%rdx)</td>
<td>*p = temp;</td>
</tr>
<tr>
<td>memory</td>
<td>register</td>
<td>movq (%rax),%rdx</td>
<td>temp = *p;</td>
</tr>
</tbody>
</table>

- Cannot do memory-memory transfer with a single instruction
**Simple Memory Addressing Modes**

- **Normal** (R) \[\text{Mem}[	ext{Reg}[R]]\]
  - Register R specifies memory address
  - Aha! Pointer dereferencing in C
    
    \[
    \text{movq} \ (\%\text{rcx}),\%\text{rax}
    \]

- **Displacement** D(R) \[\text{Mem}[	ext{Reg}[R]+D]\]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset
    
    \[
    \text{movq} \ 8(\%\text{rbp}),\%\text{rdx}
    \]
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
void swap
    (long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}

Registers

Memory

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

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

### swap:

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

**Registers**

- `%rdi` 0x120
- `%rsi` 0x100
- `%rax` 123
- `%rdx`  

**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>Value</td>
<td>123</td>
<td></td>
<td></td>
<td>456</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()

### Registers

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

### Memory

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

swap:

```asm
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>
</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:**

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

Registers

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

Swap:

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

- **Normal (R) Mem[Reg[R]]**
  - Register R specifies memory address
  - Aha! 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

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

► Most General Form

\[ D(Rb,Ri,S) \rightarrow Mem[Reg[Rb]+S*Reg[Ri]+ D] \]

- D: Constant “displacement” 1, 2, or 4 bytes
- Rb: Base register: Any of 16 integer registers
- Ri: Index register: Any, except for %esp
- S: Scale: 1, 2, 4, or 8 (why these numbers?)

► Special Cases

(Rb,Ri) \rightarrow Mem[Reg[Rb]+Reg[Ri]]
D(Rb,Ri) \rightarrow Mem[Reg[Rb]+Reg[Ri]+D]
(Rb,Ri,S) \rightarrow Mem[Reg[Rb]+S*Reg[Ri]]
# 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>

%rdx = 0xf000
%rcx = 0x0100
### Data Movements Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl S,D</td>
<td>D ← S</td>
<td>Move double word</td>
</tr>
<tr>
<td>movw S,D</td>
<td>D ← S</td>
<td>Move word</td>
</tr>
<tr>
<td>movb S,D</td>
<td>D ← S</td>
<td>Move byte</td>
</tr>
<tr>
<td>movsbl S,D</td>
<td>D ← SignExtend(S)</td>
<td>Move sign-extended byte</td>
</tr>
<tr>
<td>movzbq S,D</td>
<td>D ← ZeroExtend(S)</td>
<td>Move zero-extended byte</td>
</tr>
<tr>
<td>pushl S</td>
<td>R[%esp] ← R[%esp] – 4; M[R[%esp]] ← S</td>
<td>Push</td>
</tr>
<tr>
<td>popl D</td>
<td>D ← M[R[%esp]]; R[%esp] ← R[%esp] + 4</td>
<td>Pop</td>
</tr>
</tbody>
</table>

#### Exercise

<table>
<thead>
<tr>
<th>%dh = 0x8d, %eax = 98765432</th>
</tr>
</thead>
<tbody>
<tr>
<td>movb %dh, %eax</td>
</tr>
<tr>
<td>movsbl %dh, %eax</td>
</tr>
<tr>
<td>mobzbl %dh, %eax</td>
</tr>
</tbody>
</table>
Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
**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:

  ```asm
  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>Dest = Dest + Src</td>
</tr>
<tr>
<td>subq</td>
<td>Dest = Dest – Src</td>
</tr>
<tr>
<td>imulq</td>
<td>Dest = Dest * Src</td>
</tr>
<tr>
<td>salq</td>
<td>Dest = Dest &lt;&lt; Src <strong>Also called shlq</strong></td>
</tr>
<tr>
<td>sarq</td>
<td>Dest = Dest &gt;&gt; Src <strong>Arithmetic</strong></td>
</tr>
<tr>
<td>shrq</td>
<td>Dest = Dest &gt;&gt; Src <strong>Logical</strong></td>
</tr>
<tr>
<td>xorq</td>
<td>Dest = Dest ^ Src</td>
</tr>
<tr>
<td>andq</td>
<td>Dest = Dest &amp; Src</td>
</tr>
<tr>
<td>orq</td>
<td>Dest = Dest</td>
</tr>
</tbody>
</table>

- Watch out for argument order!
- No distinction between signed and unsigned int (why?)
Some Arithmetic Operations

- One Operand Instructions
  
  incq \[ DestDest = Dest + 1 \]
  
  decq \[ DestDest = Dest - 1 \]
  
  negq \[ DestDest = - Dest \]
  
  notq \[ DestDest = \sim Dest \]

- See book for more instructions
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
long arith
(long x, long y, long z)
{
    long t1 = x + y;
    long t2 = z + t1;
    long t3 = x + 4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}

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

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