Machine-Level Programming: Basics

Computer Systems Organization (Spring 2015)
CSCI-UA 201, Section 3

Instructor: Joanna Klukowska
Teaching Assistants: Paige Connelly & Carlos Guzman

Slides adapted from
Andrew Case, Jinyang Li, Mohamed Zahran, Randy Bryant and Dave O'Hallaron
Today: Machine Programming: Basics

• History of Intel processors and architectures (brief)

• C, assembly, machine code

• Assembly Basics: Registers, operands, move

• Intro to x86-64
Intel x86 Processors

• Totally 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. Less so for low power.
## Intel x86 Evolution: Milestones

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
<th>MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>1978</td>
<td>29K</td>
<td>5-10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>First 16-bit processor. Basis for IBM PC &amp; DOS</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1MB address space</td>
<td></td>
</tr>
<tr>
<td>386</td>
<td>1985</td>
<td>275K</td>
<td>16-33</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>First 32 bit processor, referred to as IA32</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added “flat addressing”</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Capable of running Unix</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>32-bit Linux/gcc uses no instructions introduced in later models</td>
<td></td>
</tr>
<tr>
<td>Pentium 4F</td>
<td>2004</td>
<td>125M</td>
<td>2800-3800</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>First 64-bit processor, referred to as x86-64</td>
<td></td>
</tr>
<tr>
<td>Core i7</td>
<td>2008</td>
<td>731M</td>
<td>2667-3333</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Our shark machines</td>
<td></td>
</tr>
</tbody>
</table>
Intel x86 Processors: Overview

Architectures

- X86-16
- X86-32/IA32
  - MMX
  - SSE
  - SSE2
  - SSE3
- X86-64 / EM64t
  - SSE4

Processors

- 8086
- 286
- 386
- 486
- Pentium
- Pentium MMX
- Pentium III
- Pentium 4
- Pentium 4E
- Pentium 4F
- Core 2 Duo
- Core i7

IA: often redefined as latest Intel architecture
More Information

• Intel processors ([Wikipedia](https://en.wikipedia.org/wiki/Intel_processor))
• Intel [microarchitectures](https://en.wikipedia.org/wiki/Microarchitecture)
Our Coverage

• IA32
  • The traditional x86

• x86-64/EM64T
  • The emerging standard

• Presentation
  • Book presents IA32 in Sections 3.1—3.12
  • Covers x86-64 in 3.13
  • We will cover IA32 and look briefly at x86-64 simultaneously
Today: Machine Programming: Basics

• History of Intel processors and architectures
• C, assembly, machine code
• Assembly Basics: Registers, operands, move
• Intro to x86-64
Definitions

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

• **Microarchitecture:** Implementation of the architecture.
  • Examples: cache sizes and core frequency.
Assembly Programmer’s View

- **Programmer-Visible State**
  - **PC: Program counter**
    - Address of next instruction
    - Called “EIP” (IA32) or “RIP” (x86-64)
  - **Registers**
    - 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
Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc -O1 p1.c p2.c -o p`
  - Use basic optimizations (`-O1`)
  - Put resulting binary in file `p`
Compiling Into Assembly

C Code

```c
int sum(int x, int y) {
    int t = x+y;
    return t;
}
```

Generated IA32 Assembly

```
sum:
pushl %ebp
movl %esp,%ebp
movl 12(%ebp),%eax
addl 8(%ebp),%eax
popl %ebp
ret
```

Some compilers use instruction “leave”

Obtain with command

```
/usr/local/bin/gcc -O1 -S code.c
```

Produces file `code.s`

**Note:** due to gcc version differences, the code generated on our course machine is different (although equivalent).
Assembly Characteristics: Data Types

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

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

• No aggregate types such as arrays or structures
  • Just contiguously allocated bytes in memory
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**
Machine Instruction Example

• **C Code**
  - Add two signed integers

• **Assembly**
  - Add 2 4-byte integers
    - “Long” words in GCC parlance
    - Same instruction whether signed or unsigned

• **Operands:**
  - x: Register %eax
  - y: Memory M[%ebp+8]
  - t: Register %eax
  - Return function value in %eax

• **Object Code**
  - 3-byte instruction
  - Stored at address 0x80483ca
Disassembling Object Code

Disassembled

\[
\begin{array}{llll}
080483c4 & \text{<sum>}: \\
80483c4 & \text{55 push } \%ebp \\
80483c5 & \text{89 e5 mov } \%esp,\%ebp \\
80483c7 & \text{8b 45 0c mov 0xc(\%ebp),\%eax} \\
80483ca & \text{03 45 08 add 0x8(\%ebp),\%eax} \\
80483cd & \text{5d pop \%ebp} \\
80483ce & \text{c3 ret}
\end{array}
\]

• Disassembler

\texttt{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 \texttt{a.out} (complete executable) or \texttt{.o} file
Alternate Disassembly

Object

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040</td>
<td>0x55</td>
</tr>
<tr>
<td>0x401040</td>
<td>0x89</td>
</tr>
<tr>
<td>0x401040</td>
<td>0xe5</td>
</tr>
<tr>
<td>0x401040</td>
<td>0x8b</td>
</tr>
<tr>
<td>0x401040</td>
<td>0x45</td>
</tr>
<tr>
<td>0x401040</td>
<td>0xc0</td>
</tr>
<tr>
<td>0x401040</td>
<td>0x03</td>
</tr>
<tr>
<td>0x401040</td>
<td>0x45</td>
</tr>
<tr>
<td>0x401040</td>
<td>0x08</td>
</tr>
<tr>
<td>0x401040</td>
<td>0x5d</td>
</tr>
<tr>
<td>0x401040</td>
<td>0xc3</td>
</tr>
</tbody>
</table>

Disassembled

Dump of assembler code for function sum:

```
0x080483c4 <sum+0>:     push   %ebp
0x080483c5 <sum+1>:     mov    %esp,%ebp
0x080483c7 <sum+3>:     mov    0xc(%ebp),%eax
0x080483ca <sum+6>:     add    0x8(%ebp),%eax
0x080483cd <sum+9>:     pop     %ebp
0x080483ce <sum+10>:    ret
```

- Within gdb Debugger
  
gdb p
disassemble sum
- Disassemble procedure
  
x/11xb sum
- Examine the 11 bytes starting at sum
Today: Machine Programming I: Basics

• History of Intel processors and architectures
• C, assembly, machine code
• Assembly Basics: Registers, operands, move
• Intro to x86-64
Integer Registers (IA32)

<table>
<thead>
<tr>
<th>Register</th>
<th>General Purpose</th>
<th>Origin (mostly obsolete)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>%ax</td>
<td>accumulate</td>
</tr>
<tr>
<td>%ecx</td>
<td>%cx</td>
<td>counter</td>
</tr>
<tr>
<td>%edx</td>
<td>%dx</td>
<td>data</td>
</tr>
<tr>
<td>%ebx</td>
<td>%bx</td>
<td>base</td>
</tr>
<tr>
<td>%esi</td>
<td>%si</td>
<td>source index</td>
</tr>
<tr>
<td>%edi</td>
<td>%di</td>
<td>destination index</td>
</tr>
<tr>
<td>%esp</td>
<td>%sp</td>
<td>stack pointer</td>
</tr>
<tr>
<td>%ebp</td>
<td>%bp</td>
<td>base pointer</td>
</tr>
</tbody>
</table>

16-bit virtual registers (backwards compatibility)
Moving Data: IA32

• Moving Data
  \texttt{movl} Source, Dest

• Operand Types

  • \textbf{Immediate}: Constant integer data
    • Example: $0\times400, -533$
    • Like C constant, but prefixed with ‘$’
    • Encoded with 1, 2, or 4 bytes

  • \textbf{Register}: One of 8 integer registers
    • Example: %eax, %edx
    • But %esp and %ebp reserved for special use
    • Others have special uses for particular instructions

  • \textbf{Memory}: 4 consecutive bytes of memory at address given by register
    • Simplest example: (%eax)
    • Various other “address modes” – we will see them soon
## `movl` Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src, Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imm</td>
<td>Reg</td>
<td><code>movl $0x4,%eax</code></td>
<td><code>temp = 0x4;</code></td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td><code>movl $-147,(%eax)</code></td>
<td><code>*p = -147;</code></td>
</tr>
<tr>
<td>Reg</td>
<td>Reg</td>
<td><code>movl %eax,%edx</code></td>
<td><code>temp2 = temp1;</code></td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td><code>movl %eax,(%edx)</code></td>
<td><code>*p = temp;</code></td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td><code>movl (%eax),%edx</code></td>
<td><code>temp = *p;</code></td>
</tr>
</tbody>
</table>

**No memory to memory transfer using single instruction!**
Simple Memory Addressing Modes

• Normal (R) \( \text{Mem}[	ext{Reg}[R]] \)
  - Register R specifies memory address

  \textit{movl} (\%ecx),\%eax

• Displacement D(R) \( \text{Mem}[	ext{Reg}[R]+D] \)
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

  \textit{movl} 8(\%ebp),\%edx

Note: the normal mode is a special case of displacement mode in which \( D = 0 \)
Using Simple Addressing Modes

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

```
swap:
  pushl %ebp
  movl %esp,%ebp
  pushl %ebx

  movl 8(%ebp), %edx
  movl 12(%ebp), %ecx
  movl (%edx), %ebx
  movl (%ecx), %eax
  movl %eax, (%edx)
  movl %ebx, (%ecx)

  popl %ebx
  popl %ebp
  ret
```
Using Simple Addressing Modes

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

swap:
```
pushl %ebp
movl %esp,%ebp
pushl %ebx

movl 8(%ebp), %edx
movl 12(%ebp), %ecx
movl (%edx), %ebx
movl (%ecx), %eax
movl %eax, (%edx)
movl %ebx, (%ecx)

popl %ebx
popl %ebp
ret
```
Understanding Swap

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

```
Register   Value
%edx       xp
%ecx       yp
%ebx       t0
%eax       t1
```

```
movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx # ecx = yp
movl (%edx), %ebx   # ebx = *xp (t0)
movl (%ecx), %eax   # eax = *yp (t1)
movl %eax, (%edx)   # *xp = t1
movl %ebx, (%ecx)   # *yp = t0
```
Understanding Swap

<table>
<thead>
<tr>
<th>%eax</th>
<th>%edx</th>
<th>%ecx</th>
<th>%ebx</th>
<th>%esi</th>
<th>%edi</th>
<th>%esp</th>
<th>%ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x104</td>
<td>0x104</td>
</tr>
</tbody>
</table>

```
movl 8(%ebp), %edx    # edx = xp
movl 12(%ebp), %ecx   # ecx = yp
movl (%edx), %ebx     # ebx = *xp (t0)
movl (%ecx), %eax     # eax = *yp (t1)
movl %eax, (%edx)     # *xp = t1
movl %ebx, (%ecx)     # *yp = t0
```
Understanding Swap

<table>
<thead>
<tr>
<th>%eax</th>
<th>%edx</th>
<th>%ecx</th>
<th>%ebx</th>
<th>%esi</th>
<th>%edi</th>
<th>%esp</th>
<th>%ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x104</td>
</tr>
</tbody>
</table>

movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx  # ecx = yp
movl (%edx), %ebx  # ebx = *xp (t0)
movl (%ecx), %eax  # eax = *yp (t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td>12</td>
</tr>
<tr>
<td>0x120</td>
<td>8</td>
</tr>
<tr>
<td>0x11c</td>
<td>4</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td>0x114</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x10c</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x104</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td></td>
</tr>
</tbody>
</table>
Understanding Swap

```plaintext
movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx # ecx = yp
movl (%edx), %ebx   # ebx = *xp (t0)
movl (%ecx), %eax   # eax = *yp (t1)
movl %eax, (%edx)   # *xp = t1
movl %ebx, (%ecx)   # *yp = t0
```
Understanding Swap

%eax | 0x124
%edx | 0x120
%ecx | 123
%ebx | 0x11c
%esi |
%edi |
%esp |
%ebp | 0x104

Address | Offset
123 | 0x124
456 | 0x120
4 | 0x11c
8 | 0x118
12 | 0x114
Rtn adr |
0x120 | 0x110
0x124 | 0x10c
0x104 | 0x108
0x100 |

movl 8(%ebp), %edx # edx = xp
movl 12(%ebp), %ecx # ecx = yp
movl (%edx), %ebx # ebx = *xp (t0)
movl (%ecx), %eax # eax = *yp (t1)
movl %eax, (%edx) # *xp = t1
movl %ebx, (%ecx) # *yp = t0
# Understanding Swap

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>456</td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td>123</td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

## Addresses

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0x120</td>
</tr>
<tr>
<td>8</td>
<td>0x124</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x104</td>
</tr>
<tr>
<td>-4</td>
<td>0x100</td>
</tr>
</tbody>
</table>

## Code Snippet

```assembly
movl 8(%%ebp), %edx  # edx = xp
movl 12(%%ebp), %ecx  # ecx = yp
movl (%edx), %ebx    # ebx = *xp (t0)
movl (%ecx), %eax    # eax = *yp (t1)
movl %eax, (%edx)    # *xp = t1
movl %ebx, (%ecx)    # *yp = t0
```
Understanding Swap

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp</td>
<td>12</td>
</tr>
<tr>
<td>xp</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-4</td>
<td>0</td>
</tr>
</tbody>
</table>

movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx  # ecx = yp
movl (%edx), %ebx  # ebx = *xp (t0)
movl (%ecx), %eax  # eax = *yp (t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0
Understanding Swap

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>456</td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td>123</td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp</td>
<td>12 0x120 0x110</td>
</tr>
<tr>
<td>xp</td>
<td>8 0x124 0x10c</td>
</tr>
<tr>
<td>%ebp</td>
<td>0 0x104 0x100</td>
</tr>
</tbody>
</table>

movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx  # ecx = yp
movl (%edx), %ebx  # ebx = *xp (t0)
movl (%ecx), %eax  # eax = *yp (t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0
Complete Memory Addressing Modes

• Most General Form
  \[ D(Rb,Ri,S) \rightarrow \text{Mem}[\text{Reg}[Rb] + S \times \text{Reg}[Ri] + D] \]

  • D: Constant “displacement” / “offset”
  • Rb: Base register: Any of 8 integer registers
  • Ri: Index register: Any, except for %esp
    - Unlikely you’d use %ebp, either
  • S: Scale: 1, 2, 4, or 8

• Special Cases
  
  \[
  \begin{align*}
  (Rb,Ri) & \rightarrow \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri]] \\
  D(Rb,Ri) & \rightarrow \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] + D] \\
  (Rb,Ri,S) & \rightarrow \text{Mem}[\text{Reg}[Rb] + S \times \text{Reg}[Ri]]
  \end{align*}
  \]
Today: Machine Programming: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Intro to x86-64
Data Representations: IA32 + x86-64

• Sizes of C Objects (in Bytes)

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Generic 32-bit</th>
<th>Intel IA32</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>8</td>
<td>10/12</td>
<td>16</td>
</tr>
<tr>
<td>char *</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

*Or any other pointer*
## 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>

- Extend existing registers. Add 8 new ones.
- Make %ebp/%rbp general purpose
Instructions

• Long word 1 (4 Bytes) ↔ Quad word q (8 Bytes)

• New instructions:
  • movl → movq
  • addl → addq
  • sall → salq
  • etc.

• 32-bit instructions that generate 32-bit results
  • Set higher order bits of destination register to 0
  • Example: addl
### 32-bit code for swap

#### C Code

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

#### Assembly Code

```assembly
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 8(%ebp), %edx
    movl 12(%ebp), %ecx
    movl (%edx), %ebx
    movl (%ecx), %eax
    movl %eax, (%edx)
    movl %ebx, (%ecx)
    popl %ebx
    popl %ebp
    ret
```

- **Set Up**
- **Body**
- **Finish**
64-bit code for swap

void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}

swap:

    movl (%rdi), %edx
    movl (%rsi), %eax
    movl %eax, (%rdi)
    movl %edx, (%rsi)

    ret

• Operands passed in registers (why useful?)
  • First (xp) in %rdi, second (yp) in %rsi
  • 64-bit pointers

• No stack operations required

• 32-bit data
  • Data held in registers %eax and %edx
  • movl operation
Machine Programming I: Summary

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

• C, assembly, machine code
  • Compiler must transform statements, expressions, procedures into low-level instruction sequences

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

• Intro to x86-64
  • A major departure from the style of code seen in IA32