Lecture 8: Machine-Level Programming I: Basics

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

Some slides adapted (and slightly modified) from:
- Clark Barrett
- Jinyang Li
- Randy Bryant
- Dave O’Hallaron
Intel x86 Processors

• Evolutionary design
  – Backwards compatible up until 8086, introduced in 1978

• Complex instruction set computer (CISC)
  – Many instructions, many formats
  – By contrast, ARM architecture (in most cell phones) is RISC
# Intel x86 Evolution: Milestones

<table>
<thead>
<tr>
<th>Name</th>
<th>Transistors</th>
<th>MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086 (1978)</td>
<td>29K</td>
<td>5-10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>– First 16-bit processor. Basis for IBM PC &amp; DOS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>– 1MB address space</td>
<td></td>
<td></td>
</tr>
<tr>
<td>386 (1985)</td>
<td>275K</td>
<td>16-33</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>– First 32 bit processor, referred to as <strong>IA32</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>– Capable of running Unix</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium 4F (2004)</td>
<td>125M</td>
<td>2800-3800</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>– First 64-bit processor, referred to as <strong>x86-64</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core i7 (2008)</td>
<td>731M</td>
<td>2667-3333</td>
</tr>
<tr>
<td>Xeon E7 (2011)</td>
<td>2.2B</td>
<td>~2400</td>
</tr>
</tbody>
</table>

We cover both IA32 and x86-64. Labs are done in IA32.
Assembly Programmer’s View

- **Execution context**
  - **PC**: Program counter
    - Address of next instruction
    - Called “EIP” (IA32) or “RIP” (x86-64)
  - **Registers**
    - Heavily used program data
  - **Condition codes**
    - Info of recent arithmetic operation
    - Used for conditional branching
Assembly Data Types

• "Integer" data of 1, 2, or 4 bytes
  – Represent either data value
  – or address (untyped pointer)

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

• No arrays or structures
3 Kind of Assembly Operations

• Perform arithmetic on register or memory data
  – Add, subtract, multiplication...

• 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
Turning C into Object Code

- Code in files `p1.c` `p2.c`
- Compile with command: `gcc -O1 p1.c p2.c -o p`

Text

- `C program (p1.c p2.c)`
  - Compiler (gcc -S)
  - `Asm program (p1.s p2.s)`
    - Assembler (gcc -c)
      - `Object program (p1.o p2.o)`
        - Linker
          - `Executable program (p)`

Output file is `p`
Compiling Into Assembly

**sum.c**

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

**sum.s**

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

**gcc –S sum.c**

```assembly
80483c4: 55 89 e5 8b 45 0c 03 45 08 5d c3
```

**gcc –c sum.c**

**gcc –c sum.s**

**objcump –d sum.o**

Note: If your platform is 64-bit, you may want to force it to generate 32-bit assembly by `gcc –m32 –S sum.c` to get the above output.
Compiling Into Assembly

sum.c

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

sum.s

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

Refer to register %eax
Refer to memory at address %ebp+8
## Integer Registers (IA32)

### General Purpose Registers
- `%eax` (%ax, %ah, %al)
- `%ecx` (%cx, %ch, %cl)
- `%edx` (%dx, %dh, %dl)
- `%ebx` (%bx, %bh, %bl)
- `%esi` (%si)
- `%edi` (%di)
- `%esp` (%sp)
- `%ebp` (%bp)

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

### Register Functions
- Accumulate
- Counter
- Data
- Base
- Source
- Index
- Destination
- Stack
- Pointer

### Origin
(mostly obsolete)
Moving Data: IA32

- `movl Source, Dest`

- **Operand Types**
  - *Immediate*: Integer constant
    - e.g. `$0x400`
  - *Register*: One of 8 integer registers
    - e.g. `%eax`
  - *Memory*: 4 consecutive bytes of memory at address given by register
    - Simplest example (%eax)
### 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>Mem</td>
<td>Mem</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>
</tbody>
</table>

**No memory-to-memory instruction**
Memory Addressing Modes

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

  \[
  \text{movl } (\%ecx),\%eax
  \]

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

  \[
  \text{movl } 8(\%ebp),\%edx
  \]
Using Simple Addressing Modes

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

```
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

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x104</td>
</tr>
<tr>
<td>%edx</td>
<td></td>
</tr>
<tr>
<td>%ecx</td>
<td></td>
</tr>
<tr>
<td>%ebx</td>
<td></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>

### Assembly Code

```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>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td></td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td></td>
</tr>
<tr>
<td>%ebx</td>
<td></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></td>
<td>0x124</td>
</tr>
<tr>
<td>8</td>
<td>0x124</td>
</tr>
<tr>
<td>12</td>
<td>0x120</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>-4</td>
<td></td>
</tr>
</tbody>
</table>

### Instructions

- `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>0x120</td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td></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</td>
</tr>
<tr>
<td>xp</td>
<td>8</td>
</tr>
<tr>
<td>%ebp</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

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

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

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

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>Offset</th>
<th>Address</th>
<th>123</th>
<th>456</th>
<th>0x120</th>
<th>0x110</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp</td>
<td>12</td>
<td></td>
<td></td>
<td>0x120</td>
<td>0x110</td>
</tr>
<tr>
<td>xp</td>
<td>8</td>
<td>0x124</td>
<td></td>
<td>0x10c</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>0</td>
<td>0x108</td>
<td></td>
<td>0x104</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x100</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
Address
0x124
0x120
0x11c
0x118
0x114
0x10c
0x108
0x104
0x100
```
# Understanding Swap

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp</td>
<td>0x120</td>
</tr>
<tr>
<td>xp</td>
<td>0x124</td>
</tr>
<tr>
<td></td>
<td>0x110</td>
</tr>
<tr>
<td></td>
<td>0x11c</td>
</tr>
<tr>
<td></td>
<td>0x10c</td>
</tr>
<tr>
<td></td>
<td>0x108</td>
</tr>
<tr>
<td></td>
<td>0x104</td>
</tr>
<tr>
<td></td>
<td>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
```
# Understanding Swap

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td>12</td>
<td>yp</td>
</tr>
<tr>
<td>0x120</td>
<td>8</td>
<td>xp</td>
</tr>
<tr>
<td>0x11c</td>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x114</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x104</td>
<td>0</td>
<td>%ebp</td>
</tr>
<tr>
<td>0x100</td>
<td>-4</td>
<td></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>

#### 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
```
General Memory Addressing Modes

• Most General Form

\[ D(\text{Rb}, \text{Ri}, S) \]

- Base register
- Index register (no %esp)
- Scale (1,2,4,8)
- Constant displacement

\[ \text{Mem}[\text{Reg}[\text{Rb}]+S*\text{Reg}[\text{Ri}]+D] \]

• Special Cases

- \((\text{Rb}, \text{Ri})\)
  \[ \text{Mem}[\text{Reg}[\text{Rb}]+\text{Reg}[\text{Ri}]] \]

- \(D(\text{Rb}, \text{Ri})\)
  \[ \text{Mem}[\text{Reg}[\text{Rb}]+\text{Reg}[\text{Ri}]+D] \]

- \((\text{Rb}, \text{Ri}, S)\)
  \[ \text{Mem}[\text{Reg}[\text{Rb}]+S*\text{Reg}[\text{Ri}]] \]
Size of C objects on IA32 and x86-64

<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>• 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>8-bit Registers</th>
<th>32-bit Registers</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>
</tbody>
</table>

#### Extend existing registers

- %rax, %rbx, %rcx, %rdx, %rsi, %rdi, %rsp, %rbp

#### Add 8 new ones

- %r8, %r9, %r10, %r11, %r12, %r13, %r14, %r15
Instructions

• New instructions for 8-byte types:
  - movl $\rightarrow$ movq
  - addl $\rightarrow$ addq
  - sall $\rightarrow$ salq
  - etc.
64-bit code for swap

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

• Arguments passed in registers
  – First (xp) in %rdi, second (yp) in %rsi
  – Why hold data in %eax and %edx instead of %rax %rdx?
  – Why movl operation instead of movq?
void swap(long *xp, long *yp) {
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}

moveq (%rdi), %rdx
moveq (%rsi), %rax
moveq %rax, (%rdi)
moveq %rdx, (%rsi)
ret
Conclusions

• 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