Machine-Level Programming I: Basics

Computer Systems Organization
Andrew Case

Slides adapted from Jinyang Li, Randy Bryant and Dave O’Hallaron
x86 Processors

- Totally dominate laptop/desktop/server market

- Evolutionary design
  - Backwards compatible all the way to 8086, introduced in 1978
  - Added more features as time goes on

- Complex instruction set computer (CISC)
  - Many different instructions with many different formats
  - By contrast, ARM architecture (in cell phones) is RISC (Reduced Instruction Set Computers)
    - Generally better for low power devices
# 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>386 (1985)</td>
<td>275K</td>
<td>16-33</td>
</tr>
<tr>
<td>Pentium 4F (2004)</td>
<td>125M</td>
<td>2800-3800</td>
</tr>
<tr>
<td>Core i7 (2008)</td>
<td>731M</td>
<td>2667-3333</td>
</tr>
</tbody>
</table>

- **8086 (1978)**
  - 29K transistors
  - 5-10 MHz
  - First 16-bit processor. Basis for IBM PC & DOS
  - 1MB address space

- **386 (1985)**
  - 275K transistors
  - 16-33 MHz
  - First 32-bit processor, referred to as IA32
  - Capable of running Unix

- **Pentium 4F (2004)**
  - 125M transistors
  - 2800-3800 MHz
  - First Intel 64-bit processor, referred to as x86-64

- **Core i7 (2008)**
  - 731M transistors
  - 2667-3333 MHz

---

We cover both IA32 and x86-64. Labs are done in IA32.
# 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>MMX</td>
<td>Pentium</td>
</tr>
<tr>
<td></td>
<td>Pentium MMX</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>Core 2 Duo</td>
</tr>
<tr>
<td></td>
<td>Core i7</td>
</tr>
</tbody>
</table>

IA: often redefined as latest Intel architecture
Ia64 - Itanium

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Itanium</td>
<td>2001</td>
<td>10M</td>
</tr>
</tbody>
</table>

- First shot at 64-bit architecture: first called IA64
- Not 32-bit compatible (emulation)
  - Basic reason for failure
x86 Clones: Advanced Micro Devices (AMD)

- Developed x86-64, their own extension to 64 bits
  - AMD implementation name: AMD64
  - Intel implementation name: Intel64/EM64T
Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Intro to x86-64
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)
  
  Asm program (p1.s p2.s)
  
  Object program (p1.o p2.o)

binary
  Executable program (p)
```

- Compiler (gcc -S)
- Assembler (gcc -c)
- Linker
- Static libraries (.a)
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 –c sum.c

gcc –S sum.c

gcc –c sum.s

objdump –d sum.o

(gdb) disassemble sum

sum.o

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

sum.o

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

Refer to register %eax

Refer to memory at address %ebp+8
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`

```c
int t = x+y;
```

```assembly
addl 8(%ebp),%eax
```

Similar to expression:

```c
x += y
```

More precisely:

```c
int eax;
int *ebp;
eax += ebp[2]
```

`0x80483ca: 03 45 08`
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></td>
<td>accumulate</td>
</tr>
<tr>
<td>%ecx</td>
<td></td>
<td>counter</td>
</tr>
<tr>
<td>%edx</td>
<td></td>
<td>data</td>
</tr>
<tr>
<td>%ebx</td>
<td></td>
<td>base</td>
</tr>
<tr>
<td>%esi</td>
<td></td>
<td>source</td>
</tr>
<tr>
<td>%edi</td>
<td></td>
<td>index</td>
</tr>
<tr>
<td>%esp</td>
<td>%sp</td>
<td>destination</td>
</tr>
<tr>
<td>%ebp</td>
<td>%bp</td>
<td>stack</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pointer</td>
</tr>
</tbody>
</table>

The registers are divided into:
- General purpose: %eax, %ecx, %edx, %ebx, %esi, %edi
- 16-bit virtual registers: %esp, %ebp

The 16-bit virtual registers are primarily used for backwards compatibility with older systems.
Moving Data: IA32

- **movl Source, Dest:**

- **Operand Types**
  - **Immediate:** Constant integer data
    - Example: $0x400, $-533
    - Like C constant, but prefixed with `$`
  - **Register:** One of 8 integer registers
    - Example: `%eax, %edx`
    - But `%esp` and `%ebp` reserved for special use
    - Others have special uses for particular instructions
  - **Memory:** 4 consecutive bytes of memory at address given by register
    - 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>movl $0x4,%eax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl $-147,(%eax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl %eax,%edx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl %eax,(%edx)</td>
<td>*p = temp;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl (%eax),%edx</td>
<td>temp = *p;</td>
</tr>
</tbody>
</table>

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

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

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

- **Displacement D(R)**  
  Mem[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;
}
```

```
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

<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></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
%ebx
%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

offset
0
4
8
12
-4

Address
0x124
0x120
0x11c
0x118
0x114
0x110
0x124
0x10c
0x108
0x104
0x100
Understanding Swap

\[
\begin{align*}
\text{%eax} & \quad 0x124 \\
\text{%edx} & \quad 0x120 \\
\text{%ecx} & \quad 0x120 \\
\text{%ebx} & \quad 0x120 \\
\text{%esi} & \\
\text{%edi} & \\
\text{%esp} & \quad 0x120 \\
\text{%ebp} & \quad 0x120 \\
\end{align*}
\]
Understanding Swap

%eax
%edx 0x124
%ecx 0x120
%ebx 123
%esi
%edi
%esp 0x104
%ebp 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>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>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
<tr>
<td>%ebx</td>
<td>0x100</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x108</td>
</tr>
<tr>
<td>%edx</td>
<td>0x110</td>
</tr>
<tr>
<td>yp</td>
<td>0x120</td>
</tr>
<tr>
<td>xp</td>
<td>0x124</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>

Address Table:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x100</td>
</tr>
<tr>
<td>4</td>
<td>0x104</td>
</tr>
<tr>
<td>8</td>
<td>0x110</td>
</tr>
<tr>
<td>12</td>
<td>0x120</td>
</tr>
</tbody>
</table>

Code:

```
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</td>
</tr>
<tr>
<td>xp</td>
<td>8</td>
</tr>
<tr>
<td>%ebp</td>
<td>0</td>
</tr>
<tr>
<td>-4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
</tr>
<tr>
<td>0x104</td>
</tr>
<tr>
<td>0x108</td>
</tr>
<tr>
<td>0x10c</td>
</tr>
<tr>
<td>0x110</td>
</tr>
<tr>
<td>0x114</td>
</tr>
<tr>
<td>0x118</td>
</tr>
<tr>
<td>0x120</td>
</tr>
<tr>
<td>0x124</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
```
General Memory Addressing Modes

- **Most General Form**

  \[ D (Rb, Ri, S) \]

  \[ Mem[Reg[Rb]+S*Reg[Ri]+ D] \]

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

- **Special Cases**

  | (Rb,Ri) | Mem[Reg[Rb]+Reg[Ri]] |
  | D(Rb,Ri) | Mem[Reg[Rb]+Reg[Ri]+D] |
  | (Rb,Ri,S) | Mem[Reg[Rb]+S*Reg[Ri]] |
Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Intro to x86-64
## 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>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>– Or any other pointer</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 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>%rsp</td>
<td>%esp</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</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>

- **Extend existing registers**
- **Add new ones**
Instructions

- **New instructions for 8-byte types:**
  - `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
void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```
64-bit code for swap

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

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

- **32-bit data (int)**
  - Data held in %eax and %edx (instead of %rax and %rdx)
  - `movl` operation instead of `movq`

**swap:**

```assembly
... 
movl (%rdi), %edx
movl (%rsi), %eax
movl %eax, (%rdi)
movl %edx, (%rsi)
... 
ret
```

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

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

64-bit data
- Data held in registers `%rax` and `%rdx`
- `movq` operation
  - “q” stands for quad-word
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