Introduction to Computer Systems
15-213/18-243, spring 2010
5th Lecture, Jan. 26th

Instructors:
Gregory Kesden Hunter Pitelka and Bill Naeem Hunter Pitelka
Administriva

Datalab: Due this Thursday
Bomblab: Released this Thursday

Due: Thursday, February 11th.

Questions:
15-213-staff@cs.cmu.edu
##213 on irc.freenode.net
Office Hours: Sunday – Thursday 6-9pm  Wean 5207
Last Time: Floating Point

Fractional binary numbers
IEEE floating point standard: Definition
Example and properties
Rounding, addition, multiplication
Floating point in C
Summary
Machine Programming I: Basics

History of Intel processors and architectures
C, assembly, machine code
Assembly Basics: Registers, operands, move
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

<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>First 16-bit processor. Basis for IBM PC &amp; DOS 1MB address space</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>386</td>
<td>1985</td>
<td>275K</td>
<td>16-33</td>
</tr>
<tr>
<td>First 32 bit processor, referred to as IA32 Added “flat addressing” Capable of running Unix 32-bit Linux/gcc uses no instructions introduced in later models</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium 4F</td>
<td>2005</td>
<td>230M</td>
<td>2800-3800</td>
</tr>
<tr>
<td>First 64-bit processor Meanwhile, Pentium 4s (Netburst arch.) phased out in favor of “Core” line</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# 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>X86-32/IA32</td>
<td>286, 386, 486, Pentium, Pentium MMX</td>
</tr>
<tr>
<td>MMX</td>
<td>Pentium III</td>
</tr>
<tr>
<td>SSE</td>
<td>Pentium 4</td>
</tr>
<tr>
<td>SSE2</td>
<td>Pentium 4E</td>
</tr>
<tr>
<td>SSE3</td>
<td>Pentium 4F, Core 2 Duo, Core i7</td>
</tr>
<tr>
<td>X86-64 / EM64t</td>
<td></td>
</tr>
</tbody>
</table>

IA: often redefined as latest Intel architecture
Intel x86 Processors, contd.

Machine Evolution

<table>
<thead>
<tr>
<th>Processor</th>
<th>Year</th>
<th>Total Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>486</td>
<td>1989</td>
<td>1.9M</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>3.1M</td>
</tr>
<tr>
<td>Pentium/MMX</td>
<td>1997</td>
<td>4.5M</td>
</tr>
<tr>
<td>PentiumPro</td>
<td>1995</td>
<td>6.5M</td>
</tr>
<tr>
<td>Pentium III</td>
<td>1999</td>
<td>8.2M</td>
</tr>
<tr>
<td>Pentium 4</td>
<td>2001</td>
<td>42M</td>
</tr>
<tr>
<td>Core 2 Duo</td>
<td>2006</td>
<td>291M</td>
</tr>
</tbody>
</table>

Added Features

- Instructions to support multimedia operations
- Parallel operations on 1, 2, and 4-byte data, both integer & FP
- Instructions to enable more efficient conditional operations
More Information

Intel processors (Wikipedia)
Intel microarchitectures
New Species: ia64, then IPF, then 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>
<tr>
<td>Itanium 2</td>
<td>2002</td>
<td>221M</td>
</tr>
<tr>
<td>Itanium 2 Dual-Core</td>
<td>2006</td>
<td>1.7B</td>
</tr>
</tbody>
</table>

First shot at 64-bit architecture: first called IA64
Radically new instruction set designed for high performance
Can run existing IA32 programs
On-board “x86 engine”
Joint project with Hewlett-Packard

Itanium has not taken off in marketplace
Lack of backward compatibility, no good compiler support,
x86 Clones: Advanced Micro Devices (AMD)

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

Recently
Intel much quicker with dual core design
Intel currently far ahead in performance
em64t backwards compatible to x86-64
Intel’s 64-Bit

Intel Attempted Radical Shift from IA32 to IA64
Totally different architecture (Itanium)
Executes IA32 code only as legacy
Performance disappointing

AMD Stepped 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!
Our Saltwater fish machines

Meanwhile: EM64t well introduced, however, still often not used by OS, programs
Our Coverage

**IA32**
The traditional x86

**x86-64/EM64T**
The emerging standard

**Presentation**
Book has IA32
Handout has x86-64
Lecture will cover both
Machine Programming I: Basics

History of Intel processors and architectures
C, assembly, machine code
Assembly Basics: Registers, operands, move
Definitions

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

Microarchitecture: Implementation of the architecture.

Architecture examples: instruction set specification, registers.
Microarchitecture examples: cache sizes and core frequency.

Example ISAs (Intel): x86, IA, IPF
Assembly Programmer’s View

**Programmer-Visible State**

PC: Program counter
Address of next instruction
Called “EIP” (IA32) or “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
Turning C into Object Code

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

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

Obtain with command

gcc -O -S code.c

Produces file code.s

Some compilers use single instruction “leave”
Assembly: Data Types

“Integer” data of 1, 2, 4, or 8 bytes
   1 → byte
   2 → word
   4 → long
   8 → quad

Floating point data of 4, 8, or 10 bytes
No aggregate types such as arrays or structures
Just contiguously allocated bytes in memory
Assembly: 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 sum

Object Code

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 13 bytes

Each instruction 1, 2, or 3 bytes

Starts at address 0x401040

0x401040 <sum>: 0x55
0x89
0xe5
0x8b
0x45
0x0c
0x03
0x45
0x08
0x89
0xec
0xc3
**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

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

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

Similar to expression:

```c
x += y
```

More precisely:

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

```
0x401046: 03 45 08
```
Disassembling Object Code

Disassembled

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00401040</td>
<td><code>push %ebp</code></td>
<td>Push %ebp onto the stack</td>
</tr>
<tr>
<td>1</td>
<td><code>mov %esp,%ebp</code></td>
<td>Move %esp to %ebp</td>
</tr>
<tr>
<td>3</td>
<td><code>mov 0xc(%ebp),%eax</code></td>
<td>Move 0xc(%ebp) to %eax</td>
</tr>
<tr>
<td>6</td>
<td><code>add 0x8(%ebp),%eax</code></td>
<td>Add 0x8(%ebp) to %eax</td>
</tr>
<tr>
<td>9</td>
<td><code>mov %ebp,%esp</code></td>
<td>Move %ebp to %esp</td>
</tr>
<tr>
<td>b</td>
<td><code>pop %ebp</code></td>
<td>Pop %ebp from the stack</td>
</tr>
<tr>
<td>c</td>
<td><code>ret</code></td>
<td>Return from function</td>
</tr>
<tr>
<td>d</td>
<td><code>lea 0x0(%esi),%esi</code></td>
<td>Load effective address</td>
</tr>
</tbody>
</table>

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
Alternate Disassembly

Within gdb Debugger

```
gdb p
disassemble sum
x/13b sum
```

Disassemble procedure

Examine the 13 bytes starting at sum
What Can be Disassembled?

Anything that can be interpreted as executable code
Disassembler examines bytes and reconstructs assembly source
Machine Programming I: Basics

History of Intel processors and architectures
C, assembly, machine code
Assembly Basics: Registers, operands, move
## Integer Registers (IA32)

### Origin
- Mostly obsolete
- Accumulate
- Counter
- Data
- Base
- Source
- Index
- Destination
- Stack
- Pointer
- Base
- Pointer

### 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

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

Moving Data

\[
\text{movx Source, Dest}
\]
\[
x \in \{b, w, l\}
\]

\[
\text{movl Source, Dest:}
\]
Move 4-byte “long word”

\[
\text{movw Source, Dest:}
\]
Move 2-byte “word”

\[
\text{movb Source, Dest:}
\]
Move 1-byte “byte”

Lots of these in typical code
Moving Data: IA32

Moving Data

`movl 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 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
  - Simplest example: `(%eax)`
  - Various other “address modes”
## 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><strong>Imm</strong></td>
<td>Reg</td>
<td>movl $0x4,%eax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>movl $-147,(%eax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td><strong>Reg</strong></td>
<td>Reg</td>
<td>movl %eax,%edx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td></td>
<td>Mem</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>

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

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

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

Displacement D(R) \text{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;
}
```

```assembly
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Set Up

Body

Finish
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 12(%ebp),%ecx
movl 8(%ebp),%edx
movl (%ecx),%eax
movl (%edx),%ebx
movl %eax,(%edx)
movl %ebx,(%ecx)
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```
Understanding Swap

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

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

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

Offset  | Address  
---------|---------
 yp 12   | 0x120   
 xp 8    | 0x124   
 4       | Rtn adr 
 %ebp 0  | 0x104   
 -4      | 0x100   

Understanding Swap

\[
\begin{align*}
\text{movl } &\ 12(%ebp),%ecx & \# \text{ ecx } = \text{ yp} \\
\text{movl } &\ 8(%ebp),%edx & \# \text{ edx } = \text{ xp} \\
\text{movl } &\ (%ecx),%eax & \# \text{ eax } = *\text{ yp} \ (t1) \\
\text{movl } &\ (%edx),%ebx & \# \text{ ebx } = *\text{ xp} \ (t0) \\
\text{movl } &\ %eax,(%edx) & \# \ *\text{ xp} = \text{ eax} \\
\text{movl } &\ %ebx,(%ecx) & \# *\text{ yp} = \text{ ebx}
\end{align*}
\]
Understanding Swap

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

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

<table>
<thead>
<tr>
<th>Reg</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></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>

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
<th>Rtn adr</th>
<th>yp</th>
<th>xp</th>
<th>%ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td>12</td>
<td>0x120</td>
<td>0x110</td>
<td>0x114</td>
<td>0x108</td>
</tr>
<tr>
<td>0x120</td>
<td>8</td>
<td>0x124</td>
<td>0x10c</td>
<td>0x118</td>
<td>0x104</td>
</tr>
<tr>
<td>0x11c</td>
<td>4</td>
<td>Rtn adr</td>
<td>0x118</td>
<td>0x114</td>
<td>0x100</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
<td></td>
<td>0x114</td>
<td>0x100</td>
<td></td>
</tr>
<tr>
<td>0x114</td>
<td></td>
<td></td>
<td>0x100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>%eax</th>
<th>456</th>
</tr>
</thead>
<tbody>
<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</td>
</tr>
<tr>
<td>xp</td>
<td>8 0x124</td>
</tr>
<tr>
<td>Rtn adr</td>
<td>4 0x104</td>
</tr>
</tbody>
</table>

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

<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>

<table>
<thead>
<tr>
<th>Variable</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>%esp</td>
<td>0x104</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

#### Assembly Code

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

Most General Form

\[ D(Rb,Ri,S) \quad \text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]+D] \]

D: Constant “displacement” 1, 2, or 4 bytes

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

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

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

\( (Rb,Ri,S) \quad \text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]] \)