Machine-Level Programming I: Introduction
Sept. 10, 2002

Topics
- Assembly Programmer’s Execution Model
- Accessing Information
  - Registers
  - Memory
- Arithmetic operations

IA32 Processors

Totally Dominate Computer Market

Evolutionary Design
- Starting in 1978 with 8086
- Added more features as time goes on
- Still support old features, although obsolete

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!

X86 Evolution: Programmer’s View

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>1978</td>
<td>29K</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16-bit processor. Basis for IBM PC &amp; DOS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Limited to 1MB address space. DOS only gives you 640K</td>
</tr>
<tr>
<td>80286</td>
<td>1982</td>
<td>134K</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added elaborate, but not very useful, addressing scheme</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Basis for IBM PC-AT and Windows</td>
</tr>
<tr>
<td>386</td>
<td>1985</td>
<td>275K</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Extended to 32 bits. Added “flat addressing”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Capable of running Unix</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Linux/gcc uses no instructions introduced in later models</td>
</tr>
<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></td>
<td></td>
<td>Added special collection of instructions for operating on 64-bit vectors of 1, 2, or 4 byte integer data</td>
</tr>
<tr>
<td>PentiumPro</td>
<td>1995</td>
<td>6.5M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added conditional move instructions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Big change in underlying microarchitecture</td>
</tr>
</tbody>
</table>
### X86 Evolution: Programmer’s View

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium III</td>
<td>1999</td>
<td>8.2M</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| *Added “streaming SIMD” instructions for operating on 128-bit vectors of 1, 2, or 4 byte integer or floating point data*  
| *Our fish machines* |
| Pentium 4   | 2001  | 42M         |
|            |       |             |
| *Added 8-byte formats and 144 new instructions for streaming SIMD mode* |

### X86 Evolution: Clones

#### Advanced Micro Devices (AMD)

- **Historically**
  - AMD has followed just behind Intel
  - A little bit slower, a lot cheaper
- **Recently**
  - Recruited top circuit designers from Digital Equipment Corp.
  - Exploited fact that Intel distracted by IA64
  - Now are close competitors to Intel
- **Developing own extension to 64-bits**

#### Transmeta

- **Recent start-up**
  - Employer of Linus Torvalds
- **Radically different approach to implementation**
  - Translates x86 code into “Very Long Instruction Word” (VLIW) code
  - High degree of parallelism
- **Shooting for low-power market**

### New Species: IA64

<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></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| *Extends to IA64, a 64-bit architecture*  
| *Radically new instruction set designed for high performance*  
| *Will be able to run existing IA32 programs*  
| *On-board “x86 engine”*  
| *Joint project with Hewlett-Packard* |
| Itanium 2   | 2002  | 221M        |
|            |       |             |
| *Big performance boost* |
**Assembly Programmer’s View**

- **EIP** Program Counter
  - Address of next instruction
- **Registers**
  - Heavily used program data
- **Condition Codes**
  - Store status information about most recent arithmetic operation
  - Used for conditional branching

**Programmer-Visible State**

- EIP
- Program Counter
  - Address of next instruction
- Registers
  - Heavily used program data
- Condition Codes
  - Store status information about most recent arithmetic operation
  - Used for conditional branching

**Memory**

- Object Code
- Program Data
- OS Data

**Stack**

**Compiling Into Assembly**

**C Code**

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

**Generated 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`

**Turning C into Object Code**

- **Code in files** `p1.c` `p2.c`
- **Compile with command:** `gcc -O p1.c p2.c -o p`
  - Use optimizations (`-O`)
  - Put resulting binary in file `p`

**Assembly Characteristics**

**Minimal 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

**Primitive 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**

0x401040 <sum>:
- 0x55: push %ebp
- 0x89 e5: mov %esp, %ebp
- 0x08 45 0c: mov 0xc(%ebp),%eax
- 0x8b 45 08: add 0x8(%ebp),%eax
- 0x89 ec: mov %ebp,%esp
- 0x5d: pop %ebp
- 0xc3: ret

**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

**Machine Instruction Example**

**C Code**

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

**Assembly**

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

Similar to expression

```c
x += y
```

**Disassembled Object Code**

Disassembled

```
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
```

**Disassembler**
- 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**

```
0x401040:
  0x55
  0x89 e5
  0x08 45 0c
  0x8b 45 08
  0x89 ec
 b: 5d
 c: c3
 d: 8d 76 00
```

**Within gdb Debugger**
- gdb p disassemble sum
- Disassemble procedure
- x/13b sum
- 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

% 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

Moving Data

Moving Data

imm

Move 4-byte ("long") word
Lots of these in typical code

Operand Types

Immediate: Constant integer data
- Like C constant, but prefixed with '$_'
- E.g., $_0x400, $_-533
- Encoded with 1, 2, or 4 bytes

Register: One of 8 integer registers
- But $_esp and $_ebp reserved for special use
- Others have special uses for particular instructions

Memory: 4 consecutive bytes of memory
- Various "address modes"

Operand Combinations

movl Source, Dest:

Source Destination C Analog

movl $0x4,%eax temp = 0x4;

movl $-147,(%eax) *p = -147;

movl (%eax),%edx temp2 = temp1;

movl %eax,%edx temp = *p;

Simple Addressing Modes

Normal (R) Mem[Reg[R]]

- Register R specifies memory address
- movl (%ecx),%eax

Displacement D(R) Mem[Reg[R]+D]

- Register R specifies start of memory region
- Constant displacement D specifies offset
- movl 8(%ebp),%edx

- Cannot do memory-memory transfers with single instruction
Using Simple Addressing Modes

**void swap(int *xp, int *yp)**

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

**Set Up**

- `pushl %ebp`
- `movl %esp,%ebp`
- `pushl %ebx`
- `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

**Body**

- `movl -4(%ebp),%ebx` # ebx = t0
- `movl %ebp,%esp` # bp is for return address
- `popl %ebp`
- `ret`

**Finish**

- `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**

### Register Variables

- `%eax` yp
- `%edx` xp
- `%ecx` t1
- `%ebx` t0

### Address

- **Offset**
  - `%ebp` 0
  - `%esp` 0x104
  - `%edi` 0x104
  - `%esi` 0x108
  - `%ebx` 0x10c
  - `%edx` 0x110
  - `%ecx` 0x114
  - `%ebp` 0x118
  - `%esp` 0x120
  - `%edi` 0x124
  - `%esi` 0x128
  - `%ebx` 0x130
  - `%edx` 0x135
  - `%ecx` 0x140
  - `%ebp` 0x145
  - `%esp` 0x150

### Instruction

- `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

### Stack Offset

- yp 12
- xp 8
- Rtn adr 4
- %ebp 0
- %ebp -4

### Address Offset

- 123
- 456
- 0x124
- 0x120
- 0x11c
- 0x118
- 0x114
- 0x110
- 0x10c
- 0x108
- 0x104
- 0x100
- 0x104

---

15-213, F'02
**Understanding Swap**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x100</td>
<td>push %ebp</td>
</tr>
<tr>
<td>4</td>
<td>0x104</td>
<td>movl %ebp, (%ecx) # ecx = yp</td>
</tr>
<tr>
<td>8</td>
<td>0x10c</td>
<td>movl 8(%ebp), %edx # edx = xp</td>
</tr>
<tr>
<td>12</td>
<td>0x114</td>
<td>movl (%ecx), %eax # eax = *yp (t1)</td>
</tr>
<tr>
<td>4</td>
<td>0x108</td>
<td>movl %eax, (%edx) # *xp = eax</td>
</tr>
<tr>
<td>-4</td>
<td>0x110</td>
<td>movl %ebx, (%ecx) # *yp = ebx</td>
</tr>
<tr>
<td>12</td>
<td>0x114</td>
<td>movl 12(%ebp), %ecx # ecx = yp</td>
</tr>
<tr>
<td>8</td>
<td>0x110</td>
<td>movl 8(%ebp), %edx # edx = xp</td>
</tr>
<tr>
<td>4</td>
<td>0x10c</td>
<td>movl (%ecx), %eax # eax = *yp (t1)</td>
</tr>
<tr>
<td>-4</td>
<td>0x110</td>
<td>movl %eax, (%edx) # *xp = eax</td>
</tr>
<tr>
<td>12</td>
<td>0x114</td>
<td>movl 12(%ebp), %ecx # ecx = yp</td>
</tr>
<tr>
<td>8</td>
<td>0x110</td>
<td>movl 8(%ebp), %edx # edx = xp</td>
</tr>
<tr>
<td>4</td>
<td>0x10c</td>
<td>movl (%ecx), %eax # eax = *yp (t1)</td>
</tr>
<tr>
<td>-4</td>
<td>0x110</td>
<td>movl %eax, (%edx) # *xp = eax</td>
</tr>
<tr>
<td>12</td>
<td>0x114</td>
<td>movl 12(%ebp), %ecx # ecx = yp</td>
</tr>
<tr>
<td>8</td>
<td>0x110</td>
<td>movl 8(%ebp), %edx # edx = xp</td>
</tr>
<tr>
<td>4</td>
<td>0x10c</td>
<td>movl (%ecx), %eax # eax = *yp (t1)</td>
</tr>
<tr>
<td>-4</td>
<td>0x110</td>
<td>movl %eax, (%edx) # *xp = eax</td>
</tr>
<tr>
<td>12</td>
<td>0x114</td>
<td>movl 12(%ebp), %ecx # ecx = yp</td>
</tr>
<tr>
<td>8</td>
<td>0x110</td>
<td>movl 8(%ebp), %edx # edx = xp</td>
</tr>
<tr>
<td>4</td>
<td>0x10c</td>
<td>movl (%ecx), %eax # eax = *yp (t1)</td>
</tr>
<tr>
<td>-4</td>
<td>0x110</td>
<td>movl %eax, (%edx) # *xp = eax</td>
</tr>
<tr>
<td>12</td>
<td>0x114</td>
<td>movl 12(%ebp), %ecx # ecx = yp</td>
</tr>
<tr>
<td>8</td>
<td>0x110</td>
<td>movl 8(%ebp), %edx # edx = xp</td>
</tr>
<tr>
<td>4</td>
<td>0x10c</td>
<td>movl (%ecx), %eax # eax = *yp (t1)</td>
</tr>
<tr>
<td>-4</td>
<td>0x110</td>
<td>movl %eax, (%edx) # *xp = eax</td>
</tr>
<tr>
<td>12</td>
<td>0x114</td>
<td>movl 12(%ebp), %ecx # ecx = yp</td>
</tr>
<tr>
<td>8</td>
<td>0x110</td>
<td>movl 8(%ebp), %edx # edx = xp</td>
</tr>
<tr>
<td>4</td>
<td>0x10c</td>
<td>movl (%ecx), %eax # eax = *yp (t1)</td>
</tr>
<tr>
<td>-4</td>
<td>0x110</td>
<td>movl %eax, (%edx) # *xp = eax</td>
</tr>
<tr>
<td>12</td>
<td>0x114</td>
<td>movl 12(%ebp), %ecx # ecx = yp</td>
</tr>
<tr>
<td>8</td>
<td>0x110</td>
<td>movl 8(%ebp), %edx # edx = xp</td>
</tr>
<tr>
<td>4</td>
<td>0x10c</td>
<td>movl (%ecx), %eax # eax = *yp (t1)</td>
</tr>
<tr>
<td>-4</td>
<td>0x110</td>
<td>movl %eax, (%edx) # *xp = eax</td>
</tr>
<tr>
<td>12</td>
<td>0x114</td>
<td>movl 12(%ebp), %ecx # ecx = yp</td>
</tr>
<tr>
<td>8</td>
<td>0x110</td>
<td>movl 8(%ebp), %edx # edx = xp</td>
</tr>
<tr>
<td>4</td>
<td>0x10c</td>
<td>movl (%ecx), %eax # eax = *yp (t1)</td>
</tr>
<tr>
<td>-4</td>
<td>0x110</td>
<td>movl %eax, (%edx) # *xp = eax</td>
</tr>
<tr>
<td>12</td>
<td>0x114</td>
<td>movl 12(%ebp), %ecx # ecx = yp</td>
</tr>
<tr>
<td>8</td>
<td>0x110</td>
<td>movl 8(%ebp), %edx # edx = xp</td>
</tr>
<tr>
<td>4</td>
<td>0x10c</td>
<td>movl (%ecx), %eax # eax = *yp (t1)</td>
</tr>
<tr>
<td>-4</td>
<td>0x110</td>
<td>movl %eax, (%edx) # *xp = eax</td>
</tr>
<tr>
<td>12</td>
<td>0x114</td>
<td>movl 12(%ebp), %ecx # ecx = yp</td>
</tr>
<tr>
<td>8</td>
<td>0x110</td>
<td>movl 8(%ebp), %edx # edx = xp</td>
</tr>
<tr>
<td>4</td>
<td>0x10c</td>
<td>movl (%ecx), %eax # eax = *yp (t1)</td>
</tr>
<tr>
<td>-4</td>
<td>0x110</td>
<td>movl %eax, (%edx) # *xp = eax</td>
</tr>
<tr>
<td>12</td>
<td>0x114</td>
<td>movl 12(%ebp), %ecx # ecx = yp</td>
</tr>
<tr>
<td>8</td>
<td>0x110</td>
<td>movl 8(%ebp), %edx # edx = xp</td>
</tr>
<tr>
<td>4</td>
<td>0x10c</td>
<td>movl (%ecx), %eax # eax = *yp (t1)</td>
</tr>
<tr>
<td>-4</td>
<td>0x110</td>
<td>movl %eax, (%edx) # *xp = eax</td>
</tr>
<tr>
<td>12</td>
<td>0x114</td>
<td>movl 12(%ebp), %ecx # ecx = yp</td>
</tr>
<tr>
<td>8</td>
<td>0x110</td>
<td>movl 8(%ebp), %edx # edx = xp</td>
</tr>
<tr>
<td>4</td>
<td>0x10c</td>
<td>movl (%ecx), %eax # eax = *yp (t1)</td>
</tr>
<tr>
<td>-4</td>
<td>0x110</td>
<td>movl %eax, (%edx) # *xp = eax</td>
</tr>
<tr>
<td>12</td>
<td>0x114</td>
<td>movl 12(%ebp), %ecx # ecx = yp</td>
</tr>
<tr>
<td>8</td>
<td>0x110</td>
<td>movl 8(%ebp), %edx # edx = xp</td>
</tr>
<tr>
<td>4</td>
<td>0x10c</td>
<td>movl (%ecx), %eax # eax = *yp (t1)</td>
</tr>
<tr>
<td>-4</td>
<td>0x110</td>
<td>movl %eax, (%edx) # *xp = eax</td>
</tr>
<tr>
<td>12</td>
<td>0x114</td>
<td>movl 12(%ebp), %ecx # ecx = yp</td>
</tr>
<tr>
<td>8</td>
<td>0x110</td>
<td>movl 8(%ebp), %edx # edx = xp</td>
</tr>
<tr>
<td>4</td>
<td>0x10c</td>
<td>movl (%ecx), %eax # eax = *yp (t1)</td>
</tr>
<tr>
<td>-4</td>
<td>0x110</td>
<td>movl %eax, (%edx) # *xp = eax</td>
</tr>
<tr>
<td>12</td>
<td>0x114</td>
<td>movl 12(%ebp), %ecx # ecx = yp</td>
</tr>
<tr>
<td>8</td>
<td>0x110</td>
<td>movl 8(%ebp), %edx # edx = xp</td>
</tr>
<tr>
<td>4</td>
<td>0x10c</td>
<td>movl (%ecx), %eax # eax = *yp (t1)</td>
</tr>
<tr>
<td>-4</td>
<td>0x110</td>
<td>movl %eax, (%edx) # *xp = eax</td>
</tr>
<tr>
<td>12</td>
<td>0x114</td>
<td>movl 12(%ebp), %ecx # ecx = yp</td>
</tr>
<tr>
<td>8</td>
<td>0x110</td>
<td>movl 8(%ebp), %edx # edx = xp</td>
</tr>
<tr>
<td>4</td>
<td>0x10c</td>
<td>movl (%ecx), %eax # eax = *yp (t1)</td>
</tr>
<tr>
<td>-4</td>
<td>0x110</td>
<td>movl %eax, (%edx) # *xp = eax</td>
</tr>
<tr>
<td>12</td>
<td>0x114</td>
<td>movl 12(%ebp), %ecx # ecx = yp</td>
</tr>
<tr>
<td>8</td>
<td>0x110</td>
<td>movl 8(%ebp), %edx # edx = xp</td>
</tr>
<tr>
<td>4</td>
<td>0x10c</td>
<td>movl (%ecx), %eax # eax = *yp (t1)</td>
</tr>
<tr>
<td>-4</td>
<td>0x110</td>
<td>movl %eax, (%edx) # *xp = eax</td>
</tr>
<tr>
<td>12</td>
<td>0x114</td>
<td>movl 12(%ebp), %ecx # ecx = yp</td>
</tr>
<tr>
<td>8</td>
<td>0x110</td>
<td>movl 8(%ebp), %edx # edx = xp</td>
</tr>
<tr>
<td>4</td>
<td>0x10c</td>
<td>movl (%ecx), %eax # eax = *yp (t1)</td>
</tr>
<tr>
<td>-4</td>
<td>0x110</td>
<td>movl %eax, (%edx) # *xp = eax</td>
</tr>
<tr>
<td>12</td>
<td>0x114</td>
<td>movl 12(%ebp), %ecx # ecx = yp</td>
</tr>
<tr>
<td>8</td>
<td>0x110</td>
<td>movl 8(%ebp), %edx # edx = xp</td>
</tr>
<tr>
<td>4</td>
<td>0x10c</td>
<td>movl (%ecx), %eax # eax = *yp (t1)</td>
</tr>
<tr>
<td>-4</td>
<td>0x110</td>
<td>movl %eax, (%edx) # *xp = eax</td>
</tr>
<tr>
<td>12</td>
<td>0x114</td>
<td>movl 12(%ebp), %ecx # ecx = yp</td>
</tr>
<tr>
<td>8</td>
<td>0x110</td>
<td>movl 8(%ebp), %edx # edx = xp</td>
</tr>
<tr>
<td>4</td>
<td>0x10c</td>
<td>movl (%ecx), %eax # eax = *yp (t1)</td>
</tr>
<tr>
<td>-4</td>
<td>0x110</td>
<td>movl %eax, (%edx) # *xp = eax</td>
</tr>
<tr>
<td>12</td>
<td>0x114</td>
<td>movl 12(%ebp), %ecx # ecx = yp</td>
</tr>
<tr>
<td>8</td>
<td>0x110</td>
<td>movl 8(%ebp), %edx # edx = xp</td>
</tr>
<tr>
<td>4</td>
<td>0x10c</td>
<td>movl (%ecx), %eax # eax = *yp (t1)</td>
</tr>
<tr>
<td>-4</td>
<td>0x110</td>
<td>movl %eax, (%edx) # *xp = eax</td>
</tr>
<tr>
<td>12</td>
<td>0x114</td>
<td>movl 12(%ebp), %ecx # ecx = yp</td>
</tr>
<tr>
<td>8</td>
<td>0x110</td>
<td>movl 8(%ebp), %edx # edx = xp</td>
</tr>
<tr>
<td>4</td>
<td>0x10c</td>
<td>movl (%ecx), %eax # eax = *yp (t1)</td>
</tr>
<tr>
<td>-4</td>
<td>0x110</td>
<td>movl %eax, (%edx) # *xp = eax</td>
</tr>
</tbody>
</table>
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>%ebp</td>
<td>0x100</td>
</tr>
<tr>
<td>Rtn adr</td>
<td>0x108</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset</th>
<th>YP</th>
<th>Xp</th>
<th>EBP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x110</td>
<td>12</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>0x114</td>
<td>0x120</td>
<td>0x104</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0x11c</td>
<td>0x118</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

Indexed Addressing Modes

Most General Form

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

Address Computation Instruction

leal Src, Dest

- Src is address mode expression
- Set Dest to address denoted by expression

Uses

- Computing address without doing memory reference
  - E.g., translation of p = &x[i];
- Computing arithmetic expressions of the form x + k*y
  - k = 1, 2, 4, or 8.

Some Arithmetic Operations

Format                     Computation
Two Operand Instructions

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>addl</td>
<td>Dest, Src</td>
<td>Dest = Dest + Src</td>
</tr>
<tr>
<td>subl</td>
<td>Dest, Src</td>
<td>Dest = Dest − Src</td>
</tr>
<tr>
<td>imull</td>
<td>Dest, Src</td>
<td>Dest = Dest * Src</td>
</tr>
<tr>
<td>sal</td>
<td>Dest, Src</td>
<td>Dest = Dest &lt;&lt; Src</td>
</tr>
<tr>
<td>sar</td>
<td>Dest, Src</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>xorl</td>
<td>Dest, Src</td>
<td>Dest = Dest ^ Src</td>
</tr>
<tr>
<td>andl</td>
<td>Dest, Src</td>
<td>Dest = Dest &amp; Src</td>
</tr>
<tr>
<td>orl</td>
<td>Dest, Src</td>
<td>Dest = Dest</td>
</tr>
</tbody>
</table>
Some Arithmetic Operations

### Format Computation

#### One Operand Instructions

- **incl** \( Dest \)  
  \[ Dest = Dest + 1 \]

- **decl** \( Dest \)  
  \[ Dest = Dest - 1 \]

- **negl** \( Dest \)  
  \[ Dest = - Dest \]

- **notl** \( Dest \)  
  \[ Dest = \neg Dest \]

### Using leal for Arithmetic Expressions

#### int arith

\[
\text{arith}(\text{x, y, z}) = \left( (\text{x} + \text{y}) + (\text{z} + (\text{x} + 4))) \right) \times (\text{x} + 48)
\]

```c
int arith(int x, int y, int z) {
  int t1 = x + y;
  int t2 = z + t1;
  int t3 = x + 4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```

#### Assembly Code

```assembly
pushl %ebp
movl %esp,%ebp
movl 8(%ebp),%eax    # eax = x
movl 12(%ebp),%edx  # edx = y
leal (%edx,%eax),%ecx # ecx = x+y (t1)
leal (%edx,%edx,2),%edx # edx = 3*y
sall $4,%edx         # edx = 48*y (t4)
addl 16(%ebp),%ecx   # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax # eax = 4+t4+x (t5)
imull %ecx,%eax      # eax = t5*t2 (rval)
movl %ebp,%esp
popl %ebp
ret
```

### Understanding arith

- **Offset**  
  - 0: Old %ebp
  - 4: Rtn adr
  - 8: x
  - 12: y
  - 16: z

- **Stack**  
  - %ebp
  - *
  - *
  - *

- **Body**  
  - Set Up
  - Body
  - Finish
int logical(int x, int y) {
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}

2^{13} = 8192, 2^{13} - 7 = 8185

movl 8(%ebp),%eax  eax = x
xorl 12(%ebp),%eax  eax = x^y  (t1)
sarl $17,%eax   eax = t1>>17 (t2)
andl $8185,%eax  eax = t2 & 8185

Summary: Abstract Machines

Machine Models

<table>
<thead>
<tr>
<th>Data</th>
<th>Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) char</td>
<td>1) loops</td>
</tr>
<tr>
<td>2) int, float</td>
<td>2) conditionals</td>
</tr>
<tr>
<td>3) double</td>
<td>3) goto</td>
</tr>
<tr>
<td>4) struct, array</td>
<td>4) Proc. call</td>
</tr>
<tr>
<td>5) pointer</td>
<td>5) Proc. return</td>
</tr>
</tbody>
</table>

Assembly

<table>
<thead>
<tr>
<th>mem</th>
<th>proc</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem</td>
<td>Stack</td>
</tr>
<tr>
<td>Cond. Codes</td>
<td>alu</td>
</tr>
<tr>
<td>regs</td>
<td></td>
</tr>
</tbody>
</table>

Pentium Pro (P6)

History

- Announced in Feb. '95
- Basis for Pentium II, Pentium III, and Celeron processors
- Pentium 4 similar idea, but different details

Features

- Dynamically translates instructions to more regular format
  - Very wide, but simple instructions
- Executes operations in parallel
  - Up to 5 at once
- Very deep pipeline
  - 12–18 cycle latency
PentiumPro Block Diagram

PentiumPro Operation

Translates instructions dynamically into “Uops”
- 118 bits wide
- Holds operation, two sources, and destination

Executes Uops with “Out of Order” engine
- Uop executed when
  - Operands available
  - Functional unit available
- Execution controlled by “Reservation Stations”
  - Keeps track of data dependencies between uops
  - Allocates resources

Consequences
- Indirect relationship between IA32 code & what actually gets executed
- Tricky to predict / optimize performance at assembly level

Whose Assembler?

Intel/Microsoft Format

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lea</code> eax,[ecx+ecx*2]</td>
<td></td>
</tr>
<tr>
<td><code>sub</code> esp,8</td>
<td></td>
</tr>
<tr>
<td><code>cmp</code> dword ptr [ebp-8],0</td>
<td></td>
</tr>
<tr>
<td><code>mov</code> eax,dword ptr [eax*4+100h]</td>
<td></td>
</tr>
</tbody>
</table>

GAS/Gnu Format

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>leal</code> (%ecx,%ecx,2),%eax</td>
<td></td>
</tr>
<tr>
<td><code>subl</code> $8,%esp</td>
<td></td>
</tr>
<tr>
<td><code>cmpl</code> $0,-8(%ebp)</td>
<td></td>
</tr>
<tr>
<td><code>movl</code> $0x100(%eax,4),%eax</td>
<td></td>
</tr>
</tbody>
</table>

Intel/Microsoft Diffe...