Machine-Level Programming I: Introduction

Dr. Steve Goddard
goddard@cse.unl.edu

http://cse.unl.edu/~goddard/Courses/CSCE230J

Giving credit where credit is due

- Most of slides for this lecture are based on slides created by Drs. Bryant and O’Hallaron, Carnegie Mellon University.
- I have modified them and added new slides.
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!
<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>80286</td>
<td>1982</td>
<td>134K</td>
</tr>
<tr>
<td>386</td>
<td>1985</td>
<td>275K</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>PentiumPro</td>
<td>1995</td>
<td>6.5M</td>
</tr>
</tbody>
</table>

- 16-bit processor. Basis for IBM PC & DOS
- Limited to 1MB address space. DOS only gives you 640K
- Added elaborate, but not very useful, addressing scheme
- Basis for IBM PC-AT and Windows
- Extended to 32 bits. Added “flat addressing”
- Capable of running Unix
- Linux/gcc uses no instructions introduced in later models
- Added special collection of instructions for operating on 64-bit vectors of 1, 2, or 4 byte integer data
- Added conditional move instructions
- Big change in underlying microarchitecture
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>
<tr>
<td></td>
<td></td>
<td>▪ Added “streaming SIMD” instructions for operating on 128-bit vectors of 1, 2, or 4 byte integer or floating point data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▪ Our fish machines</td>
</tr>
<tr>
<td>Pentium 4</td>
<td>2001</td>
<td>42M</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>▪ Added 8-byte formats and 144 new instructions for streaming SIMD mode</td>
</tr>
</tbody>
</table>

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
X86 Evolution: Clones

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

Programmer-Visible State
- EIP  Program Counter
  - Address of next instruction
- 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 -O p1.c p2.c -o p
  - Use optimizations (-O)
  - Put resulting binary in file p

11
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

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`

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

Object Code

<table>
<thead>
<tr>
<th>Code for <code>sum</code></th>
<th>Assembler</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040 &lt;sum&gt;: 0x55</td>
<td>• Translates .s into .o</td>
</tr>
<tr>
<td></td>
<td>• Binary encoding of each instruction</td>
</tr>
<tr>
<td>0x89 0xe5</td>
<td>• Nearly-complete image of executable code</td>
</tr>
<tr>
<td>0x8b 0x45 0x0c 0x03 0x45</td>
<td>• Missing linkages between code in different files</td>
</tr>
<tr>
<td>0x08 0x89 0xec 0x5d 0xc3</td>
<td></td>
</tr>
</tbody>
</table>

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 0x401046

<table>
<thead>
<tr>
<th>Machine Instruction Example</th>
<th>C Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int t = x+y;</code></td>
<td></td>
</tr>
<tr>
<td><code>addl 8(%ebp),%eax</code></td>
<td></td>
</tr>
<tr>
<td>Similar to expression</td>
<td></td>
</tr>
<tr>
<td><code>x += y</code></td>
<td></td>
</tr>
</tbody>
</table>

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

Disassembled

00401040 <_sum>:
0:  55  push  %ebp
1:  89 e5  mov  %esp,%ebp
2:  8b 45 0c  mov  0xc(%ebp),%eax
3:  03 45 08  add  0x8(%ebp),%eax
4:  89 ec  mov  %ebp,%esp
5:  5d  pop  %ebp
6:  c3  ret

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

Object

Disassembled

0x401040:  push  %ebp
0x401041:  mov  %esp,%ebp
0x401043:  mov  0xc(%ebp),%eax
0x401046:  add  0x8(%ebp),%eax
0x401049:  mov  %ebp,%esp
0x40104b:  pop  %ebp
0x40104c:  ret
0x40104d:  lea  0x0(%esi),%esi

Within gdb Debugger

gdb p
disassemble sum
- Disassemble procedure
x/13b sum
- Examine the 13 bytes starting at sum
What Can be Disassembled?

% 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

- Anything that can be interpreted as executable code
- Disassembler examines bytes and reconstructs assembly source

Moving Data

Moving Data

movl Source, Dest:

- 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"
**movl Operand Combinations**

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

- Cannot do memory-memory transfers with single instruction

---

**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
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 # 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

    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;
}
```
Understanding Swap

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

| %eax | 456 |
| %edx | 0x124 |
| %ecx | 0x120 |
| %ebx | |
| %esi | |
| %edi | |
| %esp | |
| %ebp | 0x104 |

Address

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

Offset

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

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

- `%eax` 456
- `%edx` 0x124
- `%ecx` 0x120
- `%ebx` 123
- `%esi` 
- `%edi` 
- `%esp` 
- `%ebp` 0x104

### Understanding Swap

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
<th>YP</th>
<th>XP</th>
<th>Rtn adr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td>12</td>
<td>0x120</td>
<td>4</td>
<td>0x108</td>
</tr>
<tr>
<td>0x120</td>
<td></td>
<td>0x124</td>
<td>0</td>
<td>0x104</td>
</tr>
<tr>
<td>0x11c</td>
<td></td>
<td>0x110</td>
<td>0</td>
<td>0x100</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
<td>0x114</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0x114</td>
<td></td>
<td>0x110</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0x10c</td>
<td></td>
<td>0x108</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
<td>0x104</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td></td>
<td>0x100</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

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

- `%eax` 456
- `%edx` 0x124
- `%ecx` 0x120
- `%ebx` 123
- `%esi` 
- `%edi` 
- `%esp` 
- `%ebp` 0x104
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>

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

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

- **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

- \((Rb,Ri)\rightarrow \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]]\)
- \(D(Rb,Ri)\rightarrow \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]+D]\)
- \((Rb,Ri,S)\rightarrow \text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]]\)
**Address Computation Examples**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%edx)</td>
<td>0xf000 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>(%edx,%ecx)</td>
<td>0xf000 + 0x100</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%edx,%ecx,4)</td>
<td>0xf000 + 4*0x100</td>
<td>0xf400</td>
</tr>
<tr>
<td>0x80,(%edx,2)</td>
<td>2*0xf000 + 0x80</td>
<td>0x1e080</td>
</tr>
</tbody>
</table>

**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
- **addl** \( Src, Dest \)  
  \( Dest = Dest + Src \)
- **subl** \( Src, Dest \)  
  \( Dest = Dest - Src \)
- **imull** \( Src, Dest \)  
  \( Dest = Dest * Src \)
- **sall** \( Src, Dest \)  
  \( Dest = Dest \ll Src \) Also called **shll**
- **sarl** \( Src, Dest \)  
  \( Dest = Dest \gg Src \) Arithmetic
- **shrl** \( Src, Dest \)  
  \( Dest = Dest \gg Src \) Logical
- **xorl** \( Src, Dest \)  
  \( Dest = Dest \wedge Src \)
- **andl** \( Src, Dest \)  
  \( Dest = Dest \& Src \)
- **orl** \( Src, Dest \)  
  \( Dest = Dest \mid Src \)

### One Operand Instructions

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
</table>
- **incl** \( Dest \)  
  \( Dest = Dest + 1 \)
- **decl** \( Dest \)  
  \( Dest = Dest - 1 \)
- **negl** \( Dest \)  
  \( Dest = -Dest \)
- **notl** \( Dest \)  
  \( Dest = \sim Dest \)
Using `lea` for Arithmetic Expressions

```
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;
    }
```

```
arith:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl 12(%ebp),%edx
    leal (%edx,%eax),%ecx
    leal (%edx,%edx,2),%edx
    sall $4,%edx
    addl 16(%ebp),%ecx
    leal 4(%edx,%eax),%eax
    imull %ecx,%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

Understanding `arith`

```
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;
    }
```

```
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)
```
Understanding arith

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

Another Example

```c
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;
}
```

```assembly
movl 8(%ebp), %eax
xorl 12(%ebp), %eax
sarl $17, %eax
andl $8185, %eax
movl %ebp, %esp
popl %ebp
ret
```

```assembly
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
```

\[2^{13} = 8192, 2^{13} - 7 = 8185\]
CISC Properties

Instruction can reference different operand types
- Immediate, register, memory

Arithmetic operations can read/write memory

Memory reference can involve complex computation
- $R_b + S^*R_i + D$
- Useful for arithmetic expressions, too

Instructions can have varying lengths
- IA32 instructions can range from 1 to 15 bytes

Summary: Abstract Machines

Machine Models

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

Assembly

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

| 1) byte | 3) branch/jump |
| 2) 2-byte word | 4) call |
| 3) 4-byte long word | 5) ret |
| 4) contiguous byte allocation |
| 5) address of initial byte |
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 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?

<table>
<thead>
<tr>
<th>Intel/Microsoft Format</th>
<th>GAS/Gnu Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>lea eax, [ecx+ecx*2]</td>
<td>leal (%ecx,%ecx,2),%eax</td>
</tr>
<tr>
<td>sub esp, 8</td>
<td>subl $8,%esp</td>
</tr>
<tr>
<td>cmp dword ptr [ebp-8], 0</td>
<td>cmpl $0,-8(%ebp)</td>
</tr>
<tr>
<td>mov eax,dword ptr [eax*4+100h]</td>
<td>movl $0x100(%eax,4),%eax</td>
</tr>
</tbody>
</table>

Intel/Microsoft Differs from GAS
- Operands listed in opposite order
  mov Dest, Src               movl Src, Dest
- Constants not preceded by `$`, Denote hex with ‘h’ at end
  100h $0x100
- Operand size indicated by operands rather than operator suffix
  sub subl
- Addressing format shows effective address computation
  [eax*4+100h] $0x100 (%eax,4)