15-213
“The course that gives CMU its Zip!”

Machine-Level Programming I:
Introduction
Sept. 11, 2001

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

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>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>
</tbody>
</table>

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>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>
</tbody>
</table>

Pentium/MMX
• Added special collection of instructions for operating on 64-bit vectors of 1, 2, or 4 byte integer data

PentiumPro
• Added conditional move instructions
• Big change in underlying microarchitecture

Pentium III
• 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
• 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

Name     Date     Transistors

Itanium     2001    10M
- 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

Assembly Programmer’s View

CPU
- Registers
- Condition Codes

Memory
- Object Code
- Program Data
- OS Data

Stack

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

C program (p1.c p2.c)
Compiler (gcc -S)

Asm program (p1.s p2.s)
Assembler (gcc or as)

Object program (p1.o p2.o)
Linker (gcc or ld)

Executable program (p)
Static libraries (.a)
### 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`

<table>
<thead>
<tr>
<th>Address</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040</td>
<td><code>&lt;sum&gt;</code>:</td>
</tr>
<tr>
<td>0x401040</td>
<td>0x55</td>
</tr>
<tr>
<td>0x401041</td>
<td>0x89</td>
</tr>
<tr>
<td>0x401042</td>
<td>0x05</td>
</tr>
<tr>
<td>0x401043</td>
<td>0x8b</td>
</tr>
<tr>
<td>0x401044</td>
<td>0x54</td>
</tr>
<tr>
<td>0x401045</td>
<td>0x0c</td>
</tr>
<tr>
<td>0x401046</td>
<td>0x03</td>
</tr>
<tr>
<td>0x401047</td>
<td>0x45</td>
</tr>
<tr>
<td>0x401048</td>
<td>0x08</td>
</tr>
<tr>
<td>0x401049</td>
<td>0x89</td>
</tr>
<tr>
<td>0x40104A</td>
<td>0x0C</td>
</tr>
<tr>
<td>0x40104B</td>
<td>0x5D</td>
</tr>
<tr>
<td>0x40104C</td>
<td>0xC3</td>
</tr>
</tbody>
</table>

**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

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

#### Assembly

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

Similar to expression

```
x += y
```

### Machine Instruction Example

#### C Code

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

#### Assembly

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

Similar to expression

```
x += y
```

#### Object Code

- 3-byte instruction
  - Stored at address 0x401046
Disassembling Object Code

Disassembled

<table>
<thead>
<tr>
<th>00401040</th>
<th>&lt;sum&gt;:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>55</td>
</tr>
<tr>
<td>1:</td>
<td>89 e5</td>
</tr>
<tr>
<td>3:</td>
<td>8b 45 0c</td>
</tr>
<tr>
<td>6:</td>
<td>03 45 08</td>
</tr>
<tr>
<td>9:</td>
<td>89 ec</td>
</tr>
<tr>
<td>b:</td>
<td>5d</td>
</tr>
<tr>
<td>c:</td>
<td>c3</td>
</tr>
<tr>
<td>d:</td>
<td>8d 76 00</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

- Disassemble procedure
- 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 $0x304cd91

- 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

- Accounts for 31% of all instructions in sample

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>Imm</td>
<td>movl $0x4,%eax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td>Reg</td>
<td>movl $-147,%eax</td>
<td>*p = -147;</td>
</tr>
<tr>
<td>Mem</td>
<td>movl %eax,%edx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td>Reg</td>
<td>movl %eax,(%edx)</td>
<td>*p = temp;</td>
</tr>
<tr>
<td>Mem</td>
<td>movl (%eax),%edx</td>
<td>temp = *p;</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;
}
```

**Understanding Swap**

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

**Stack**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>yp</td>
</tr>
<tr>
<td>8</td>
<td>xp</td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>0</td>
<td>%ebp</td>
</tr>
<tr>
<td>-4</td>
<td>Old %ebp</td>
</tr>
</tbody>
</table>

**Register Variable**

```c
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**

\[ D(R_b,R_i,S) \rightarrow \text{Mem}[\text{Reg}[R_b]+S \times \text{Reg}[R_i]+D] \]

- **D**: Constant “displacement” 1, 2, or 4 bytes
- **R_b**: Base register: Any of 8 integer registers
- **R_i**: Index register: Any, except for %esp
  - Unlikely you’d use %ebp, either
- **S**: Scale: 1, 2, 4, or 8

**Special Cases**

- \((R_b,R_i) \rightarrow \text{Mem}[\text{Reg}[R_b]+\text{Reg}[R_i]]\)
- \(D(R_b,R_i) \rightarrow \text{Mem}[\text{Reg}[R_b]+\text{Reg}[R_i]+D]\)
- \((R_b,R_i,S) \rightarrow \text{Mem}[\text{Reg}[R_b]+S \times \text{Reg}[R_i]]\)

---

## 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 \( x + k \times y \)
  - \( k = 1, 2, 4, \) or 8.

---

## Some Arithmetic Operations

### Format Computation

#### Two Operand Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>addl</strong> <em>Src, Dest</em></td>
<td>Dest = Dest + <em>Src</em></td>
</tr>
<tr>
<td><strong>subl</strong> <em>Src, Dest</em></td>
<td>Dest = Dest - <em>Src</em></td>
</tr>
<tr>
<td><strong>imull</strong> <em>Src, Dest</em></td>
<td>Dest = Dest * <em>Src</em></td>
</tr>
<tr>
<td><strong>sall</strong> <em>Src, Dest</em></td>
<td>Dest = Dest &lt;&lt; <em>Src</em>  Also called <strong>shll</strong></td>
</tr>
<tr>
<td><strong>sarl</strong> <em>Src, Dest</em></td>
<td>Dest = Dest &gt;&gt; <em>Src</em>  Arithmetic</td>
</tr>
<tr>
<td><strong>xorl</strong> <em>Src, Dest</em></td>
<td>Dest = Dest ^ <em>Src</em>  Logical</td>
</tr>
<tr>
<td><strong>andl</strong> <em>Src, Dest</em></td>
<td>Dest = Dest &amp; <em>Src</em></td>
</tr>
<tr>
<td><strong>orl</strong> <em>Src, Dest</em></td>
<td>Dest = Dest</td>
</tr>
</tbody>
</table>

#### One Operand Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>incl</strong> <em>Dest</em></td>
<td>Dest = Dest + 1</td>
</tr>
<tr>
<td><strong>decl</strong> <em>Dest</em></td>
<td>Dest = Dest - 1</td>
</tr>
<tr>
<td><strong>negl</strong> <em>Dest</em></td>
<td>Dest = - Dest</td>
</tr>
<tr>
<td><strong>notl</strong> <em>Dest</em></td>
<td>Dest = ~ Dest</td>
</tr>
</tbody>
</table>

---

## Using leal for Arithmetic Expressions

**int arith**

```c
(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, %eax, 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

```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;
}
```

```asm
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)
imul %ecx,%eax         # eax = t5*t2 (rval)
```

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;
}
```

```asm
logical:
    pushl %ebp
    movl %esp ,%ebp
    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
    movl %ebp ,%esp
    popl %ebp
    ret
```

CISC Properties

- 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>mem</td>
<td>proc</td>
<td></td>
</tr>
<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) goto</td>
</tr>
<tr>
<td></td>
<td>4) struct, array</td>
<td>4) goto</td>
</tr>
<tr>
<td></td>
<td>5) pointer</td>
<td>5) Proc. call</td>
</tr>
</tbody>
</table>
```

Assembly

```
mem   regs   alu
Stack Cond. Codes processor
```

- 1) byte
- 2) 4-byte long word
- 3) 8-byte quad word
- 4) contiguous byte allocation
- 5) address of initial byte
Pentium Pro (P6)

History
- Announced in Feb. ‘95
- Basis for Pentium II, Pentium III, Pentium 4, and Celeron processors

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
- Difficult to predict / optimize performance at assembly level

Whose Assembler?

Intel/Microsoft Format

<table>
<thead>
<tr>
<th>Instruction</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>cmp $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
- Constants not preceded by ‘$’, Denote hexadecimal with ‘h’ at end
  - 100h
  - $0x100
- Operand size indicated by operands rather than operator suffix
- Addressing format shows effective address computation
  - [eax*4+100h]
  - $0x100(,%eax,4)