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
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>• 16-bit processor. Basis for IBM PC &amp; DOS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Limited to 1MB address space. DOS only gives you 640K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80286</td>
<td>1982</td>
<td>134K</td>
</tr>
<tr>
<td>• Added elaborate, but not very useful, addressing scheme</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Basis for IBM PC-AT and Windows</td>
<td></td>
<td></td>
</tr>
<tr>
<td>386</td>
<td>1985</td>
<td>275K</td>
</tr>
<tr>
<td>• Extended to 32 bits. Added “flat addressing”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Capable of running Unix</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Linux/gcc uses no instructions introduced in later models</td>
<td></td>
<td></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>
# 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></td>
<td></td>
<td></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>
<tr>
<td>Pentium III</td>
<td>1999</td>
<td>8.2M</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>• 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

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>
</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
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`

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

```bash
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`

0x401040 <sum>:

- Total of 13 bytes
- Each instruction 1, 2, or 3 bytes
- Starts at address 0x401040

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

- 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
Disassembling Object Code

Disassembled

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

Disassembled

Object

0x401040:
0x55
0x89
0xe5
0x8b
0x45
0x03
0x45
0x08
0x89
0xec
0xc3

Disassembled:

0x401040 <sum>: push %ebp
0x401041 <sum+1>: mov %esp,%ebp
0x401043 <sum+3>: mov 0xc(%ebp),%eax
0x401046 <sum+6>: add 0x8(%ebp),%eax
0x401049 <sum+9>: mov %ebp,%esp
0x40104b <sum+11>: pop %ebp
0x40104c <sum+12>: ret
0x40104d <sum+13>: 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?

- 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   $0xfffffff f
30001005: 68 90 10 00 30 push   $0x3000109 0
3000100a: 68 91 dc 4c 30 push   $0x304cdc9 1
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>Reg</td>
<td>movl $0x4,%eax</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl $-147,(%eax)</td>
</tr>
<tr>
<td>Reg</td>
<td>Reg</td>
<td>movl %eax,%edx</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl %eax,(%edx)</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl (%eax),%edx</td>
</tr>
</tbody>
</table>

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

Normal (R) \text{ Mem}[\text{Reg}[R]]
- Register R specifies memory address
  \text{movl} (\%ecx),\%eax

Displacement D(R) \text{ Mem}[\text{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
Understanding Swap

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
Indexed Addressing Modes

Most General Form

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

- **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

\begin{align*}
(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 \times \text{Reg}[Ri]]
\end{align*}
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

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Two Operand Instructions</strong></td>
<td></td>
</tr>
<tr>
<td><code>addl</code> <code>Src, Dest</code></td>
<td><code>Dest = Dest + Src</code></td>
</tr>
<tr>
<td><code>subl</code> <code>Src, Dest</code></td>
<td><code>Dest = Dest - Src</code></td>
</tr>
<tr>
<td><code>imull</code> <code>Src, Dest</code></td>
<td><code>Dest = Dest * Src</code></td>
</tr>
<tr>
<td><code>sall</code> <code>Src, Dest</code></td>
<td><code>Dest = Dest &lt;&lt; Src</code></td>
</tr>
<tr>
<td><code>srll</code> <code>Src, Dest</code></td>
<td><code>Dest = Dest &gt;&gt; Src</code></td>
</tr>
<tr>
<td><code>xorl</code> <code>Src, Dest</code></td>
<td><code>Dest = Dest ^ Src</code></td>
</tr>
<tr>
<td><code>andl</code> <code>Src, Dest</code></td>
<td><code>Dest = Dest &amp; Src</code></td>
</tr>
<tr>
<td><code>orl</code> <code>Src, Dest</code></td>
<td>`Dest = Dest</td>
</tr>
<tr>
<td><strong>One Operand Instructions</strong></td>
<td></td>
</tr>
<tr>
<td><code>incl</code> <code>Dest</code></td>
<td><code>Dest = Dest + 1</code></td>
</tr>
<tr>
<td><code>decl</code> <code>Dest</code></td>
<td><code>Dest = Dest - 1</code></td>
</tr>
<tr>
<td><code>negl</code> <code>Dest</code></td>
<td><code>Dest = - Dest</code></td>
</tr>
<tr>
<td><code>notl</code> <code>Dest</code></td>
<td><code>Dest = ~ Dest</code></td>
</tr>
</tbody>
</table>

Also called `shll` Arithmetic Logical
Using `lea`l for Arithmetic Expressions

```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;
}
```

```
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
```

### Set Up

### Body

### Finish
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)
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;
}
```

```
set up
pushl %ebp
movl %esp,%ebp

body
movl 8(%ebp),%eax
xorl 12(%ebp),%eax
sarl $17,%eax
andl $8185,%eax
movl %ebp,%esp

finish
popl %ebp
ret
```

```plaintext
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`
CISC Properties

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

Arithmetic operations can read/write memory

Memory reference can involve complex computation
  • \( R_b + S \cdot R_i + D \)
  • Useful for arithmetic expressions, too

Instructions can have varying lengths
  • IA32 instructions can range from 1 to 15 bytes
## Summary: Abstract Machines

### Machine Models

- **Machine Models**
  - **C**
    - mem
    - proc

### Data

1. char
2. int, float
3. double
4. struct, array
5. pointer

### Control

1. loops
2. conditionals
3. goto
4. Proc. call
5. Proc. return

### Assembly

- **mem**
- **Stack**
  - regs
  - Cond. Codes
  - alu
  - processor

1. byte
2. 4-byte long word
3. branch/jump
4. call
5. ret
6. contiguous byte allocation
7. 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>lea</th>
<th>eax, [ecx+ecx*2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub</td>
<td>esp, 8</td>
</tr>
<tr>
<td>cmp</td>
<td>dword ptr [ebp-8], 0</td>
</tr>
<tr>
<td>mov</td>
<td>eax, dword ptr [eax*4+100h]</td>
</tr>
</tbody>
</table>

GAS/Gnu Format

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

Intel/Microsoft Differs from GAS

- Operands listed in opposite order
  
<table>
<thead>
<tr>
<th>mov</th>
<th>Dest, Src</th>
<th>movl</th>
<th>Src, Dest</th>
</tr>
</thead>
</table>

- Constants not preceded by ‘$’, Denote hexadecimal with ‘h’ at end
  
  | 100h | $0x100 |

- Operand size indicated by operands rather than operator suffix

<table>
<thead>
<tr>
<th>sub</th>
<th>subl</th>
</tr>
</thead>
</table>

- Addressing format shows effective address computation

  | [eax*4+100h] | $0x100(,%eax,4) |