Machine-Level Programming I: Introduction
Sept. 9, 2003

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>
</tbody>
</table>

X86 Evolution: Programmer’s View

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</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></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>
<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

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

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

Itanium 2

- 2002
- 221M
- Big performance boost
Assembly Programmer’s View

Programmer-Visible State
- EIP: Program Counter
- Address of next instruction
- Register File
- Memory
  - Byte addressable array
  - Code, user data, (some) OS data
  - Includes stack used to support procedures
- Condition Codes
  - Store status information about most recent arithmetic operation
  - Used for conditional branching

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

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>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040</td>
<td>&lt;_sum&gt;:</td>
</tr>
<tr>
<td>0x55</td>
<td>push %ebp</td>
</tr>
<tr>
<td>0x89</td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td>0xe5</td>
<td>mov 0xc(%ebp),%eax</td>
</tr>
<tr>
<td>0x45</td>
<td>mov %ebp,%esp</td>
</tr>
<tr>
<td>0x8c</td>
<td>pop %ebp</td>
</tr>
<tr>
<td>0x8d</td>
<td>ret</td>
</tr>
<tr>
<td>0x0d</td>
<td>mov 0x0(%esi),%esi</td>
</tr>
</tbody>
</table>

- 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

**Disassembling Object Code**

<table>
<thead>
<tr>
<th>Address</th>
<th>Disassembled</th>
</tr>
</thead>
<tbody>
<tr>
<td>00401040</td>
<td>&lt;_sum&gt;:</td>
</tr>
<tr>
<td>0: 0x55</td>
<td>push %ebp</td>
</tr>
<tr>
<td>1: 0x89 0xe5</td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td>3: 0x8b 0x45 0xc</td>
<td>mov 0xc(%ebp),%eax</td>
</tr>
<tr>
<td>6: 0x03 0x45 0x08</td>
<td>add 0x8(%ebp),%eax</td>
</tr>
<tr>
<td>9: 0x89 0xec</td>
<td>mov %ebp,%esp</td>
</tr>
<tr>
<td>b: 0x5d</td>
<td>pop %ebp</td>
</tr>
<tr>
<td>c: 0xc3</td>
<td>ret</td>
</tr>
<tr>
<td>d: 0x8d 0x76 0x00</td>
<td>lea 0x0(%esi),%esi</td>
</tr>
</tbody>
</table>

**C Code**

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

**Assembly**

- Add two signed integers
- 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

**Alternate Disassembly**

<table>
<thead>
<tr>
<th>Address</th>
<th>Disassembled</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040</td>
<td>&lt;_sum&gt;:</td>
</tr>
<tr>
<td>0x55</td>
<td>push %ebp</td>
</tr>
<tr>
<td>0x89</td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td>0xe5</td>
<td>mov 0xc(%ebp),%eax</td>
</tr>
<tr>
<td>0x8b</td>
<td>mov %ebp,%esp</td>
</tr>
<tr>
<td>0x45</td>
<td>add 0x8(%ebp),%eax</td>
</tr>
<tr>
<td>0x8c</td>
<td>mov %ebp,%esp</td>
</tr>
<tr>
<td>0x0c</td>
<td>pop %ebp</td>
</tr>
<tr>
<td>0x03</td>
<td>ret</td>
</tr>
<tr>
<td>0x45</td>
<td>mov 0x0(%esi),%esi</td>
</tr>
</tbody>
</table>

**Machine Instruction Example**

**Disassembled**

0x401046: 03 45 08

**Within gdb Debugger**

```
gdb p disassemble 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

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

movl Source Destination C Analog

Imm

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

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

Reg

movl %eax, %edx temp2 = temp1;

Mem

movl %eax, (%edx) *p = temp;

Reg

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
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;
}
```

```
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 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  

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

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 \( %spp \)
  - 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 Examples

Address Computation Instruction

\textbf{leal} \ Src, \ Dest

- \( Src \) is address mode expression
- Set \( Dest \) to address denoted by expression

\textbf{Uses}

- Computing address without doing memory reference
  - E.g., translation of \( p = s[x[1]] \):
- Computing arithmetic expressions of the form \( x + k\times y \)
  - \( k = 1, 2, 4, \text{ or } 8 \).
Some Arithmetic Operations

Format | Computation

Two Operand Instructions
- \texttt{addl \ Src,\Dest} \quad \text{Dest} = \Dest + \Src
- \texttt{subl \ Src,\Dest} \quad \text{Dest} = \Dest - \Src
- \texttt{imull \ Src,\Dest} \quad \text{Dest} = \Dest \ast \Src
- \texttt{sall \ Src,\Dest} \quad \text{Dest} = \Dest \ll \Src \text{ Also called \texttt{shll}}
- \texttt{sarl \ Src,\Dest} \quad \text{Dest} = \Dest \gg \Src \text{ Logical}
- \texttt{xorl \ Src,\Dest} \quad \text{Dest} = \Dest \oplus \Src
- \texttt{andl \ Src,\Dest} \quad \text{Dest} = \Dest \land \Src
- \texttt{orl \ Src,\Dest} \quad \text{Dest} = \Dest \lor \Src

One Operand Instructions
- \texttt{incl \ Dest} \quad \text{Dest} = \Dest + 1
- \texttt{decl \ Dest} \quad \text{Dest} = \Dest - 1
- \texttt{negl \ Dest} \quad \text{Dest} = -\Dest
- \texttt{notl \ Dest} \quad \text{Dest} = -\Dest

Using \texttt{leal} for Arithmetic Expressions

\texttt{int arith (int x, int y, int z)}
\{\texttt{int t1 = x+y;}}
\{\texttt{int t2 = z+t1;}}
\{\texttt{int t3 = x+4;}}
\{\texttt{int t4 = y \ast 48;}}
\{\texttt{int t5 = t3 + t4;}}
\{\texttt{int rval = t2 \ast t5;}}
\{\texttt{return rval;}}

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;
}
```

Stack

<table>
<thead>
<tr>
<th>Offset</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>z</td>
</tr>
<tr>
<td>12</td>
<td>y</td>
</tr>
<tr>
<td>8</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td>Rtn addr</td>
</tr>
<tr>
<td>0</td>
<td>Old %ebp</td>
</tr>
</tbody>
</table>
### 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 = t2 * t5;
    int rval = t2 * t5;
    return 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;
}
```

CISC Properties

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

- Arithmetic operations can read/write memory

- Memory reference can involve complex computation
  - Rb + S^RI + 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**

- C
  - mem ─ proc

**Data**

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

**Control**

- 1) loops
- 2) conditionals
- 3) switch
- 4) Proc. call
- 5) Proc. return

**Assembly**

- 1) byte
- 2) 2-byte word
- 3) 4-byte long word
- 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 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>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 Diffs from GAS
- Operands listed in opposite order
- Operand size indicated by operands rather than operator suffix
- Addressing format shows effective address computation

In summary, the Pentium Pro (P6) is an advanced processor designed to provide dynamic instruction translation, parallel execution, and deep pipeline capabilities. Its features include very wide, yet simple instructions, the ability to execute up to 5 operations simultaneously, and a very deep pipeline that allows for 12–18 cycle latency. The block diagram provides a visual representation of the processor's architecture, while the operation section details how instructions are translated and executed. The comparison between Intel/Microsoft and GAS/Gnu formats highlights the differences in how operands are listed, size is indicated, and addressing is handled, offering insights into the assembly language syntax used with these processors.