Machine-Level Programming I: Introduction
Feb. 1, 2000

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>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>Pentium II</td>
<td>1997</td>
<td>7M</td>
</tr>
<tr>
<td>Pentium III</td>
<td>1999</td>
<td>8.2M</td>
</tr>
<tr>
<td>Merced</td>
<td>2000</td>
<td>10M</td>
</tr>
</tbody>
</table>

- 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
- Added “streaming SIMD” instructions for operating on 128-bit vectors of 1, 2, or 4 byte integer or floating point data
- 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”
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

CPU

Memory
Object Code
Program Data
OS Data

Stack

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

Compiler (gcc -s)

Asm program (p1.s p2.s)

Assembler (gcc or as)

Linker (gcc or ld)

Object program (p1.o p2.o)

Static libraries (.a)

Executable program (p)

Compiling Into Assembly

C Code

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

Generated Assembly

```
_sum:
pushl %ebp
movl %esp,%ebp
movl 12(%ebp),%eax
addl 8(%ebp),%eax
movl %eax,%ebp
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 or 8 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
### Machine Instruction Example

**C Code**
- Add two signed integers
  ```c
  int t = x+y;
  ```

**Assembly**
- Add 2 4-byte integers
  ```assembly
  addl 8(%ebp),%eax
  ```
- Similar to expression
  ```c
  x += y
  ```

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

### Disassembling Object Code

**Disassembled**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040</td>
<td>push %ebp</td>
</tr>
<tr>
<td>0x401041</td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td>0x401043</td>
<td>mov 0xc(%ebp),%eax</td>
</tr>
<tr>
<td>0x401046</td>
<td>add 0x8(%ebp),%eax</td>
</tr>
<tr>
<td>0x401049</td>
<td>mov %ebp,%esp</td>
</tr>
<tr>
<td>0x40104b</td>
<td>lea 0x0(%esi),%esi</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**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040</td>
<td>push %ebp</td>
</tr>
<tr>
<td>0x401041</td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td>0x401043</td>
<td>mov 0xc(%ebp),%eax</td>
</tr>
<tr>
<td>0x401046</td>
<td>add 0x8(%ebp),%eax</td>
</tr>
<tr>
<td>0x401049</td>
<td>mov %ebp,%esp</td>
</tr>
<tr>
<td>0x40104b</td>
<td>lea 0x0(%esi),%esi</td>
</tr>
</tbody>
</table>

**Within gdb Debugger**
- `gdb p`
  - Disassemble procedure
  - `x/13b sum`
  - Examine the 13 bytes starting at sum

---

**Object Code**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040</td>
<td>push %ebp</td>
</tr>
<tr>
<td>0x401041</td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td>0x401043</td>
<td>mov 0xc(%ebp),%eax</td>
</tr>
<tr>
<td>0x401046</td>
<td>add 0x8(%ebp),%eax</td>
</tr>
<tr>
<td>0x401049</td>
<td>mov %ebp,%esp</td>
</tr>
<tr>
<td>0x40104b</td>
<td>lea 0x0(%esi),%esi</td>
</tr>
</tbody>
</table>
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
  • Accounts for 31% of all instructions in sample

Operand Types

• Immediate: Constant integer data
  • E.g., $0x400, $-533
  • Encoded with 1, 2, or 4 bytes
• Register: One of 8 integer registers
  • Others have special uses for particular instructions
• Memory: 4 consecutive bytes of memory
  • Various "address modes"
```

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

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

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

Understanding Swap

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

Indexed Addressing Modes

**Most General Form**
\[
D(Rb, Ri, S) \rightarrow Mem[Reg[Rb]+S*Reg[Ri]+D]
\]

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

**Special Cases**

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

Address Computation Instruction

\textbf{leal} \textit{Src, Dest}

- \textit{Src} is address mode expression
- \textit{Set Dest} to address denoted by expression

**Uses**

- Computing address without doing memory reference
  - E.g., translation of \(p = k * i\)
- Computing arithmetic expressions of the form \(x + k \cdot y\)
  - \(k = 1, 2, 4,\) or 8.
### Some Arithmetic Operations

#### Format

**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 << Src  Also called `shll`
- `sarl Src, Dest`: Dest = Dest >> Src  Arithmetic
- `xorl Src, Dest`: Dest = Dest ^ Src
- `andl Src, Dest`: Dest = Dest & Src
- `orl Src, Dest`: Dest = Dest | Src

**One Operand Instructions**
- `incl Dest`: Dest = Dest + 1
- `decl Dest`: Dest = Dest - 1
- `negl Dest`: Dest = - Dest
- `notl Dest`: Dest = ~ Dest

### Using `leal` 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;
}
```

```
```

### 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;
}
```

### 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;
}
```

```c
```
**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**

<table>
<thead>
<tr>
<th>Machine Models</th>
<th>Data</th>
<th>Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Assembly

```
mem proc
```

```
1) byte
2) 4-byte long word
3) branch/jump
4) call
5) ret
```

**Pentium Pro (P6)**

**History**
- Announced in Feb. '95
- Basis for Pentium II & Pentium III

**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

**Pentium Pro Block Diagram**

Microprocessor Report 2/16/95
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 Differ from GAS
- Operands listed in opposite order
- Constants not preceded by ‘$’. Denote hexadecimal with ‘h’ at end
- Operand size indicated by operands rather than operator suffix
- Addressing format shows effective address computation

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

leal (%ecx,%ecx,2), %eax
movl $0x100(,%eax,4), %eax