Systems I

Machine-Level Programming I:
Introduction

Topics

- Assembly Programmer’s Execution Model
- Accessing Information
  - Registers
IA32 Processors

Totally Dominate General Purpose CPU 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>
<th>Details</th>
</tr>
</thead>
</table>
| 8086   | 1978 | 29K         | - 16-bit processor. Basis for IBM PC & DOS  
- Limited to 1MB address space. DOS only gives you 640K |
| 80286  | 1982 | 134K        | - Added elaborate, but not very useful, addressing scheme  
- Basis for IBM PC-AT and Windows |
| 386    | 1985 | 275K        | - Extended to 32 bits. Added “flat addressing”  
- Capable of running Unix  
- Linux/gcc uses no instructions introduced in later models |
## X86 Evolution: Programmer’s View

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>486</td>
<td>1989</td>
<td>1.9M</td>
<td>- Added on-chip floating-point unit</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>3.1M</td>
<td></td>
</tr>
<tr>
<td>Pentium/MMX</td>
<td>1997</td>
<td>4.5M</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>
<td>- Added conditional move instructions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Hardware can execute instructions out of order</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>■ 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>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>
<tr>
<td></td>
<td></td>
<td>■ “Superpipelined” with very fast clocks</td>
</tr>
<tr>
<td>“Nehalem”</td>
<td>2009</td>
<td>700M+</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ 4 Cores on the same chip</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ 8MB+ of on-chip memory</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
  - Drove 64-bit extensions to IA32 architecture
  - Acquired ATI (graphics chip company)
  - Increasing core counts too
  - 6-core Opteron (Istanbul) 2009

Variety of x86 chips in different markets

- Embedded/low power (Atom, Neo)
- Desktop/laptop
- Server
- Supercomputer
Abstract and Concrete Machine Models

Machine Models

C

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

mem ─── proc

mem ─── regs ─── alu
Stack

Cond. Codes

processor

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

3) call
4) ret
5) address of initial byte
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**

- **Object Code**
- **Program Data**
- **OS Data**

- **Stack**
  - Byte addressable array
  - Code, user data, (some) OS data
  - Includes stack used to support procedures
By the *architecture* of a system, I mean the complete and detailed specification of the user interface. ... As Blaauw has said, “Where architecture tells *what* happens, implementation tells *how* it is made to happen.”

*The Mythical Man-Month*, Brooks, pg 45
Instruction Set Architecture

Principles

Contract between programmer and the hardware

- Defines visible state of the system
- Defines how state changes in response to instructions

Programmer: ISA is model of how a program will execute

Hardware Designer: ISA is formal definition of the correct way to execute a program

- With a stable ISA, SW doesn’t care what the HW looks like under the covers
  - Hardware implementations can change (drastically)
  - As long as the HW implements the same ISA, all prior SW will still run
- Example: x86 ISA has spanned many chips
  - Instructions have been added but the SW of prior chips still runs on later chips

ISA specification

- The binary encodings of the instruction set
Instruction Set Architecture

Contract between programmer and the hardware
- Defines visible state of the system
- Defines how state changes in response to instructions

Programmer: ISA is model of how a program will execute

Hardware Designer: ISA is formal definition of the correct way to execute a program

ISA specification
- The binary encodings of the instruction set
ISA Basics

Instruction formats
Instruction types
Addressing modes

Before State

Mem
Regs

instruction
Op Mode Ra Rb

Data types
Operations
Interrupts/Events

After State

Mem
Regs

Machine state
Memory organization
Register organization
Architecture vs. Implementation

Architecture: defines what a computer system does in response to a program and a set of data

- Programmer visible elements of computer system

Implementation: defines how a computer does it

- Sequence of steps to complete operations
- Time to execute each operation
- Hidden “bookkeeping” functions
Examples

Architecture or Implementation?

- Number of GP registers
- Width of memory bus
- Binary representation of the instruction
  \texttt{sub \ r4, r2, \#27}
- Number of cycles to execute FP instruction
- How condition code bits are set on a move instruction
- Size of the instruction cache
- Type of FP format
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>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040</td>
<td><code>sum</code></td>
</tr>
<tr>
<td>0x55</td>
<td></td>
</tr>
<tr>
<td>0x89</td>
<td></td>
</tr>
<tr>
<td>0xe5</td>
<td></td>
</tr>
<tr>
<td>0x8b</td>
<td></td>
</tr>
<tr>
<td>0x45</td>
<td></td>
</tr>
<tr>
<td>0x0c</td>
<td></td>
</tr>
<tr>
<td>0x03</td>
<td></td>
</tr>
<tr>
<td>0x45</td>
<td></td>
</tr>
<tr>
<td>0x08</td>
<td></td>
</tr>
<tr>
<td>0x89</td>
<td></td>
</tr>
<tr>
<td>0xec</td>
<td></td>
</tr>
<tr>
<td>0x5d</td>
<td></td>
</tr>
<tr>
<td>0xc3</td>
<td></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
Machine Instruction Example

C Code

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

- 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`
## Disassembling Object Code

### Disassembled

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>00401040</td>
<td><code>&lt;_sum&gt;</code>:</td>
<td></td>
</tr>
<tr>
<td>0:</td>
<td>55</td>
<td>push %ebp</td>
</tr>
<tr>
<td>1:</td>
<td>89 e5</td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td>3:</td>
<td>8b 45 0c</td>
<td>mov 0xc(%ebp),%eax</td>
</tr>
<tr>
<td>6:</td>
<td>03 45 08</td>
<td>add 0x8(%ebp),%eax</td>
</tr>
<tr>
<td>9:</td>
<td>89 ec</td>
<td>mov %ebp,%esp</td>
</tr>
<tr>
<td>b:</td>
<td>5d</td>
<td>pop %ebp</td>
</tr>
<tr>
<td>c:</td>
<td>c3</td>
<td>ret</td>
</tr>
<tr>
<td>d:</td>
<td>8d 76 00</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

Object

Disassembled

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040</td>
<td>push %ebp</td>
<td></td>
</tr>
<tr>
<td>0x401041</td>
<td>mov %esp,%ebp</td>
<td></td>
</tr>
<tr>
<td>0x401043</td>
<td>mov 0xc(%ebp),%eax</td>
<td></td>
</tr>
<tr>
<td>0x401046</td>
<td>add 0x8(%ebp),%eax</td>
<td></td>
</tr>
<tr>
<td>0x401049</td>
<td>mov %ebp,%esp</td>
<td></td>
</tr>
<tr>
<td>0x40104b</td>
<td>pop %ebp</td>
<td></td>
</tr>
<tr>
<td>0x40104c</td>
<td>ret</td>
<td></td>
</tr>
<tr>
<td>0x40104d</td>
<td>lea 0x0(%esi),%esi</td>
<td></td>
</tr>
</tbody>
</table>

Within gdb Debugger

gdb p
disable 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

```bash
% 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
```

# Whose Assembler?

## Intel/Microsoft Format

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lea</code></td>
<td><code>eax,[ecx+ecx*2]</code></td>
</tr>
<tr>
<td><code>sub</code></td>
<td><code>esp,8</code></td>
</tr>
<tr>
<td><code>cmp</code></td>
<td><code>dword ptr [ebp-8],0</code></td>
</tr>
<tr>
<td><code>mov</code></td>
<td><code>eax,dword ptr [eax*4+100h]</code></td>
</tr>
</tbody>
</table>

## GAS/Gnu Format

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

## Intel/Microsoft Differs from GAS

- **Operands listed in opposite order**
  - Intel/Microsoft: `mov Dest, Src`
  - GAS/Gnu: `movl Src, Dest`

- **Constants not preceded by ‘$’, Denote hex with ‘h’ at end**
  - Intel/Microsoft: `100h`
  - GAS/Gnu: `$0x100`

- **Operand size indicated by operands rather than operator suffix**
  - Intel/Microsoft: `sub` and `mov`
  - GAS/Gnu: `subl` and `movl`

- **Addressing format shows effective address computation**
  - Intel/Microsoft: `leal (%ecx,%ecx,2),%eax`
  - GAS/Gnu: `$0x100(%eax,4),%eax`
Moving Data

Moving Data

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

Today
- ISA/processor evolution (for x86)
- Programmer machine models
- Introduction to ISA and usage

Next time
- Memory access
- Arithmetic operations
- C pointers and Addresses