Lecture 4: Machine Basics

Computer Architecture and Systems Programming
(252-0061-00)

Timothy Roscoe
Herbstsemester 2012

Last Time: Floating Point

- Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

Today: Basic machine language programming

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move

What is an “Architecture”?

- Machine viewed by software (machine code)
- Assembly Instructions, registers, trap handling, etc.

Intel x86 Processors

- The x86 Architecture dominates the computer market

- Evolutionary design
  - Backwards compatible up until 8086, introduced in 1978
  - Added more features as time goes on

- 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!

Intel x86 Evolution: Milestones

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
<th>MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>1978</td>
<td>29K</td>
<td>5-10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80386</td>
<td>1985</td>
<td>275K</td>
<td>16-33</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80486</td>
<td>2005</td>
<td>230M</td>
<td>2800-3800</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Intel x86 Processors: Overview

Architectures
- X86-16
- X86-32/IA32
- MMX
- SSE
- SSE2
- SSE3
- SSE4
- X86-64 / EM64t

Processors
- 8086
- 286
- 386
- 486
- Pentium
- Pentium MMX
- Pentium III
- Pentium 4
- Pentium 4E
- Pentium 4F
- Core 2 Duo
- Core i7

IA: often redefined as latest Intel architecture
Intel x86 Processors, contd.

- **Machine Evolution**
  - 486 1989 1.9M
  - Pentium 1993 3.1M
  - Pentium/MMX '97 74.5M
  - PentiumPro 1995 6.5M
  - Pentium III 1999 8.2M
  - Pentium 4 2001 42M
  - Core 2 Duo 2006 291M
- **Added Features**
  - Instructions to support multimedia operations
    - Parallel operations on 1, 2, and 4-byte data, both integer & FP
  - Instructions to enable more efficient conditional operations

---

x86 Clones: Advanced Micro Devices (AMD)

- **Historically**
  - AMD has followed just behind Intel
  - A little bit slower, a lot cheaper
- **Then**
  - Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
  - Built Opteron: tough competitor to Pentium 4
  - Developed x86-64, their own extension to 64 bits
- **Recently**
  - Intel much quicker with dual core design
  - Intel currently far ahead in performance
  - em64t backwards compatible to x86-64

---

Intel’s 64-Bit (partially true…)

- **Intel Attempted Radical Shift from IA32 to IA64**
  - Totally different architecture (Itanium)
  - Executes IA32 code only as legacy
  - Performance disappointing
- **AMD Stepped in with Evolutionary Solution**
  - x86-64 (now called “AMD64”)
- **Intel Felt Obligated to Focus on IA64**
  - Hard to admit mistake or that AMD is better
- **2004: Intel Announces EM64T extension to IA32**
  - Extended Memory 64-bit Technology
  - Almost identical to x86-64!

---

Intel Nehalem-EX

- **Current leader (for the next few weeks)**
  - 2.3 billion transistors/die
  - 8 or 10 cores per die
  - 2 threads per core
  - Up to 8 packages (= 128 contexts!)
  - 4 memory channels per package
  - Virtualization support
  - etc.
- **Good illustration of why it is hard to teach state-of-the-art processor design!**

---

Intel Single-Chip Cloud Computer - 2010

- **Experimental processor**
  - (only a few 100 made)
    - Designed for research
    - Working version in our Lab
- **48 old-style Pentium cores**
- **Very fast interconnection network**
  - Hardware support for messaging between cores
  - Variable speed of network
- **Non-cache coherent**
  - Sharing memory between cores won’t work with a conventional OS!

---

There are many architectures...

- **You’ve already seen MIPS 2000 → MIPS 3000 → ...**
- Workstations, minicomputers, now mostly embedded networking
- **IBM S/360 → S/370 → ... → zSeries**
  - First to separate architecture from (many) implementations
- **ARM (several variants)**
  - Very common in embedded systems, basis for Advanced OS course at ETHZ
- **IBM POWER → PowerPC (→ Cell, sort of)**
  - Basis for all 3 current games console systems
- **DEC Alpha**
  - Personal favorite; killed by Compaq, team left for Intel to work on...
- **Intel Itanium**
  - First 64-bit Intel product; very fast (esp. FP), hot, and expensive
  - Mostly overtaken by 64-bit x86 designs
- **etc.**
Our Coverage

- ia32
  - The traditional x86
- x86-64/EM64T
  - The emerging standard
- Presentation
  - Book has ia32
  - Handout has x86-64
  - Lecture will cover both

Comparison with MIPS

(remember Digital Design?)

- MIPS is RISC: Reduced Instruction Set
  - Motivation: simpler is faster
    - Fewer gates ⇒ higher frequency
    - Fewer gates ⇒ more transistors left for cache
  - Seemed like a really good idea
- x86 is CISC: Complex Instruction Set
  - More complex instructions, addressing modes
    - Intel turned out to be way too good at manufacturing
    - Difference in gate count became too small to make a difference
    - x86 inside is mostly RISC anyway, decode logic is small
  - ⇒ Argument is mostly irrelevant these days

Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move

Definitions

- Architecture: (also instruction set architecture: ISA) The parts of a processor design that one needs to understand to write assembly code.
- Microarchitecture: Implementation of the architecture.
- Architecture examples: instruction set specification, registers.
- Microarchitecture examples: cache sizes and core frequency.
- Example ISAs: x86, MIPS, ia64, VAX, Alpha, ARM, etc.

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

Assembly Programmer’s View

- CPU
  - PC
  - Registers
  - Condition Codes
- Memory
  - Addresses
  - Data
  - Instructions
  - Object Code
  - Program Data
  - OS Data
  - Stack
- Programmer-Visible State
  - PC: Program counter
    - Address of next instruction
      - Called "EIP" (ia32) or "RIP" (x86-64)
  - Register file
    - Heavily used program data
  - Condition codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching
- Static libraries (.a)

- C program (p1.c p2.c)
  - Compiler (gcc -S)
  - Assembler (gcc or as)
  - Object program (p1.o p2.o)
  - Linker (gcc or ld)
  - Executable program (p)
Compiling Into Assembly

C code

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

Generated IA32 Assembly

```assembly
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`

Some compilers use single instruction "leave"

Assembly Characteristics: 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

Assembly Characteristics: 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`

```assembly
0x401040 <sum>:
0x55 pushl %ebp
0x89 movl %esp,%ebp
0x45 movl 12(%ebp),%eax
0x45 addl 8(%ebp),%eax
0x89 movl %ebp,%esp
0x45 popl %ebp
0xc3 ret
```

Machine Instruction Example

```
int t = x+y;
addl 8(%ebp),%eax
```

Similar to expression:

```
x += y
```

More precisely:

```assembly
int eax;
int *ebp;
eax += ebp[2]
```

Disassembling Object Code

```
Disassembled
```

```assembly
00401040 <_sum>:
0: 55 pushl %ebp
1: 89 e5 movl %esp,%ebp
3: 8b 45 0c movl 0xc(%ebp),%eax
6: 03 45 08 addl 0x8(%ebp),%eax
9: 89 ec movl %ebp,%esp
b: 5d popl %ebp
c: c3 ret
```

```assembly
0401046: 03 45 08
```

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 `.out` (complete executable) or `.o` file
Alternate Disassembly

Object

Disassembled

Within gdb Debugger

- `gdb p` – disassemble `sum`
- Disassemble procedure
- `x/13b sum` – Examine the 13 bytes starting at `sum`

What Can be Disassembled?

```bash
% objdump -d WINWORD.EXE
WINWORD.EXE:  file format pe-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

Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move

Integer Registers (IA32)

<table>
<thead>
<tr>
<th>General Purpose</th>
<th>Accumulate</th>
<th>Counter</th>
<th>Data</th>
<th>Base</th>
<th>Source Index</th>
<th>Destination Index</th>
<th>Stack Pointer</th>
<th>Base Pointer</th>
<th>Origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>%ax</td>
<td>%ah</td>
<td>%al</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ecx</td>
<td>%cx</td>
<td>%ch</td>
<td>%cl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%edx</td>
<td>%dx</td>
<td>%dh</td>
<td>%dl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ebx</td>
<td>%bx</td>
<td>%bh</td>
<td>%bl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%esi</td>
<td>%si</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td>%di</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td>%sp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>%bp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 16-bit virtual registers (backwards compatibility)
- Mostly obsolete

Moving Data: IA32

- Moving Data
  - `movx Source, Dest` – `x` in `{b, w, l}`
  - `movl Source, Dest`: Move 4-byte “long word”
  - `movw Source, Dest`: Move 2-byte “word”
  - `movb Source, Dest`: Move 1-byte “byte”
- Lots of these in typical code

Moving Data: IA32

- Moving Data
  - `movl Source, Dest`:
  - Immediate: Constant integer data
    - Example: `$0x400, $-533`
    - Like C constant, but prefixed with `$`
    - Encoded with 1, 2, or 4 bytes
  - Register: One of 8 integer registers
    - Example: `%eax, %edx`
    - But `%esp` and `%ebp` reserved for special use
    - Others have special uses for particular instructions
  - Memory: 4 consecutive bytes of memory at address given by register
    - Simplest example: (`%eax`)
**movl** Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src, Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>Reg</td>
<td>mem-memory transfer cannot be done with a single instruction</td>
</tr>
<tr>
<td>Imm</td>
<td>Reg</td>
<td>Reg</td>
<td>mem-memory transfer cannot be done with a single instruction</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>Reg</td>
<td>mem-memory transfer cannot be done with a single instruction</td>
</tr>
</tbody>
</table>

---

**Simple Memory Addressing Modes**

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

- **Displacement** (D)  \( \text{Mem[Reg[R]+D]} \)
  - Register R specifies start of memory region
  - Constant displacement D specifies offset
  
  ```
  movl 8(%ebp),%edx
  ```

---

**Using Simple Addressing Modes**

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

```
swap:
pushl %ebp
movl %esp,%ebp
pushl %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
ret
```

---

**Understanding Swap**

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

Stack (in memory)

```
Offset  YP  xp  Rtn adr  Old %ebp  Old %ebx
12      yp  xp  
8       
4       0    0     0
0       0    0     0
```

Register file

```
%eax  123  0x124
%edx  0x120
%ecx  0x12c
%esi  0x11c
%edi  0x118
%ebp  0x114
%esp  0x10c
```

Stack (in memory)

```
Offset  YP  xp  Rtn adr  Old %ebp  Old %ebx
12      yp  xp  
8       
4       0    0     0
0       0    0     0
```

Register file

```
%eax  123  0x124
%edx  0x120
%ecx  0x12c
%esi  0x11c
%edi  0x118
%ebp  0x114
%esp  0x10c
```
Understanding Swap

Register file

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>456</td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td>0x120</td>
</tr>
<tr>
<td>%esi</td>
<td>0x11c</td>
</tr>
<tr>
<td>%edi</td>
<td>0x11c</td>
</tr>
<tr>
<td>%esp</td>
<td>0x118</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x11c</td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0x120</td>
</tr>
<tr>
<td>8</td>
<td>0x124</td>
</tr>
<tr>
<td>4</td>
<td>0x108</td>
</tr>
<tr>
<td>0</td>
<td>0x104</td>
</tr>
</tbody>
</table>

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

Complete Memory Addressing Modes

- Most General Form
  D(Rb,Ri,S) 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 %esp
    - Unlikely you’d use %ebp, either
  - S: Scale: 1, 2, 4, or 8 (why these numbers?)

- Special Cases
  (Rb,Ri) Mem[Reg[Rb]+Reg[Ri]]
  D(Rb,Ri) Mem[Reg[Rb]+Reg[Ri]+D]
  (Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]]

Next time

- Complete addressing mode, address computation (leal)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches
- While loops