Computer Systems
CSE 410 Spring 2012
5 - Machine Architecture & Instruction Sets
Roadmap

C:
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);

Java:
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
c.getMPG();

Assembly language:
get_mpg:
pushq %rbp
movq %rsp, %rbp
...
popq %rbp
ret

Machine code:
0111010000011000
100011010000010000000010
1000100111000010
11000001111110101000011111

OS:
Windows 8
Mac

Computer system:

Memory & data
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C
Basics of architecture, machine programming

- What is an ISA (Instruction Set Architecture)?
- A brief history of Intel processors and architectures
- C, assembly, machine code
- x86 basics: registers

Reading: Bryant/O’Hallaron sec. 3.1-3.2
What makes programs run fast?

Translation

Code Time

User program in C

Compile Time

C compiler

Assembler

Run Time

Hardware

.c file

.exe file

Introduction to ISA
Translation Impacts Performance

- The time required to execute a program depends on:
  - *The program* (as written in C, for instance)
  - *The compiler*: what set of assembler instructions it translates the C program into
  - *The instruction set architecture* (ISA): what set of instructions it makes available to the compiler
  - *The hardware implementation*: how much time it takes to execute an instruction

- Extremely complex interaction between these
Instruction Set Architectures

- The ISA defines:
  - The system’s state (e.g. registers, memory, program counter)
  - The instructions the CPU can execute
  - The effect that each of these instructions will have on the system state
General ISA Design Decisions

- **Instructions**
  - What instructions are available? What do they do?
  - How are they encoded?

- **Registers**
  - How many registers are there?
  - How wide are they?

- **Memory**
  - How do you specify a memory location?
x86

- Processors that implement the x86 ISA completely dominate the server, desktop and laptop markets

- 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
  - (as opposed to Reduced Instruction Set Computers (RISC), which use simpler instructions)
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>First 16-bit processor. Basis for IBM PC &amp; DOS</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1MB address space</td>
<td></td>
</tr>
<tr>
<td>386</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>First 32 bit processor, referred to as IA32</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added “flat addressing”</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Capable of running Unix</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>32-bit Linux/gcc targets i386 by default</td>
<td></td>
</tr>
<tr>
<td>Pentium 4F</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>First 64-bit Intel x86 processor, referred to as x86-64</td>
<td></td>
</tr>
</tbody>
</table>
Intel x86 Processors

Machine Evolution

- 486 1989 1.9M
- Pentium 1993 3.1M
- Pentium/MMX 1997 4.5M
- PentiumPro 1995 6.5M
- Pentium III 1999 8.2M
- Pentium 4 2001 42M
- Core 2 Duo 2006 291M
- Core i7 2008 731M

Added Features

- Instructions to support multimedia operations
  - Parallel operations on 1, 2, and 4-byte data
- Instructions to enable more efficient conditional operations
- More cores!
More information

- References for Intel processor specifications:
  - Intel’s “automated relational knowledgebase”:
    - http://ark.intel.com/
  - Wikipedia:
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 and other downward trending companies
  - Built Opteron: tough competitor to Pentium 4
  - Developed x86-64, their own extension of x86 to 64 bits
Our Coverage

- IA32
  - The traditional x86

- x86-64
  - The emerging standard – all lab assignments use x86-64!
Definitions

- **Architecture**: (also instruction set architecture or ISA)
  The parts of a processor design that one needs to understand to write assembly code
  - “What is directly visible to software”

- **Microarchitecture**: Implementation of the architecture

- Is cache size “architecture”?  
- How about core frequency?  
- And number of registers?
Assembly Programmer’s View

- **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

- **Memory**
  - Byte addressable array
  - Code, user data, (some) OS data
  - Includes stack used to support procedures (we’ll come back to that)
Turning C into Object Code

- Code in files `p1.c  p2.c`
- Compile with command: `gcc -O1 p1.c p2.c -o p`
  - Use basic optimizations (`-O1`)
  - Put resulting binary in file `p`

1. **Text**
   - C program (`p1.c  p2.c`)

2. **Text**
   - Asm program (`p1.s  p2.s`)

3. **Binary**
   - Object program (`p1.o  p2.o`)

4. **Binary**
   - Executable program (`p`)

5. **Static libraries** (`.a`)

Instruction Set Architecture
Compiling Into Assembly

C Code

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

Generated IA32 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 -O1 -S code.c
```

Produces file `code.s`
Three Basic Kinds of Instructions

- 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
Assembly Characteristics: Data Types

- “Integer” data of 1, 2, 4 (IA32), or 8 (just in x86-64) bytes
  - Data values
  - Addresses (untyped pointers)

- Floating point data of 4, 8, or 10 bytes

- What about “aggregate” types such as arrays or structs?
  - No aggregate types, just contiguously allocated bytes in memory
Object Code

Code for `sum`

```
0x401040  <sum>:
  0x55
  0x89
  0xe5
  0x8b
  0x45
  0x0c
  0x03
  0x45
  0x08
  0x89
  0xec
  0x5d
  0xc3
```

- Total of 13 bytes
- Each instruction 1, 2, or 3 bytes
- Starts at address 0x401040
- Not at all obvious where each instruction starts and ends

**Assembler**
- Translates `.s` into `.o`
- Binary encoding of each instruction
- Nearly-complete image of executable code
- Missing links between code in different files

**Linker**
- Resolves references between object files and (re)locates their data
- Combines with static run-time libraries
  - E.g., code for `malloc`, `printf`
- Some libraries are *dynamically linked*
  - Linking occurs when program begins execution
C Code: add two signed integers

Assembly

- Add two 4-byte integers
  - “Long” words in GCC speak
  - 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

```
00401040 <__sum>:  push  %ebp
  0:  55
  1:  89 e5  mov  %esp,%ebp
  3:  8b 45 0c  mov  0xc(%ebp),%eax
  6:  03 45 08  add  0x8(%ebp),%eax
  9:  89 ec  mov  %ebp,%esp
 b:  5d  pop  %ebp
 c:  c3  ret
```

**Disassembler**

`objdump -d p`

- Useful tool for examining object code (*man 1 objdump*)
- Analyzes bit pattern of series of instructions (delineates instructions)
- Produces near-exact rendition of assembly code
- Can be run on either `p` (complete executable) or `p1.o / p2.o` file
Alternate Disassembly

Object

Disassembled

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040:</td>
<td>push %ebp</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>

Within gdb debugger

gdb p
disassemble sum
(disassemble function)
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 $0xffffffff
30001005: 68 90 10 00 30 push $0x30001090
3000100a: 68 91 dc 4c 30 push $0x304cdc91
```

Instruction Set Architecture
What Is A Register?

- A location in the CPU that stores a small amount of data, which can be accessed very quickly (once every clock cycle)

- Registers are at the heart of assembly programming
  - They are a precious commodity in all architectures, but especially x86
Integer Registers (IA32)

- %eax
- %ecx
- %edx
- %ebx
- %esi
- %edi
- %esp
- %ebp

- Origin (mostly obsolete)
  - accumulate
  - counter
  - data
  - base
  - source
  - index
  - destination
  - index
  - stack
  - pointer
  - base
  - pointer

- 32-bits wide
### Integer Registers (IA32)

<table>
<thead>
<tr>
<th>Register</th>
<th>Purpose</th>
<th>Origin</th>
<th>16-bit virtual registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>General purpose</td>
<td>accumulate</td>
<td>backwards compatibility</td>
</tr>
<tr>
<td>%ecx</td>
<td></td>
<td>counter</td>
<td></td>
</tr>
<tr>
<td>%edx</td>
<td></td>
<td>data</td>
<td></td>
</tr>
<tr>
<td>%ebx</td>
<td></td>
<td>base</td>
<td></td>
</tr>
<tr>
<td>%esi</td>
<td></td>
<td>source</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>index</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
<td>destination</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
<td>stack</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td></td>
<td>base pointer</td>
<td></td>
</tr>
</tbody>
</table>

**16-bit virtual registers (backwards compatibility)**

**General purpose registers:**
- %eax
- %ecx
- %edx
- %ebx
- %esi
- %edi
- %esp
- %ebp

**Origin (mostly obsolete):**
- %ah
- %al
- %ch
- %cl
- %dh
- %dl
- %bh
- %bl

**Instruction Set Architecture**
x86-64 Integer Registers

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
<th>%r8</th>
<th>%r8d</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
<td>%r15</td>
<td>%r15d</td>
</tr>
</tbody>
</table>

- Extend existing registers, and add 8 new ones; all accessible as 8, 16, 32, 64 bits.
Summary: Machine Programming

- **What is an ISA (Instruction Set Architecture)?**
  - Defines the system’s state and instructions that are available to the software

- **History of Intel processors and architectures**
  - Evolutionary design leads to many quirks and artifacts

- **C, assembly, machine code**
  - Compiler must transform statements, expressions, procedures into low-level instruction sequences

- **x86 registers**
  - Very limited number
  - Not all general-purpose