Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Intro to x86-64

Intel x86 Processors

- Dominant in laptop/desktop/server market
  - Second place: compatible designs from AMD
- Evolutionary design
  - Backwards compatible through 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 a subset encountered with Linux/GCC programs
  - Alternative Reduced Instruction Set Computer (RISC) designs have theoretical advantages
  - Intel borrows ideas from RISC while keeping CISC compatibility
  - RISC-style ARM dominates lower-power (e.g. phone) market

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>80386</td>
<td>1985</td>
<td>275K</td>
<td>16-33</td>
</tr>
<tr>
<td>Pentium 4F</td>
<td>2004</td>
<td>125M</td>
<td>2800-3800</td>
</tr>
<tr>
<td>Core i7</td>
<td>2008</td>
<td>731M</td>
<td>2667-3333</td>
</tr>
</tbody>
</table>

Intel x86 Processors: Overview

- IA: often redefined as latest Intel architecture

Intel x86 Processors, contd.

- Machine Evolution
  - 386 1985 0.3M
  - 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, both integer & FP
  - Instructions to enable more efficient conditional operations
- Linux/GCC Evolution
  - Two major steps: 1) support 32-bit 386. 2) support 64-bit x86-64
New Species: IA64 aka IPF aka Itanium,...

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Itanium</td>
<td>2001</td>
<td>25M + cache?</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Itanium 2</td>
<td>2002</td>
<td>221M</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Itanium 2 Dual-Core</td>
<td>2006</td>
<td>1.7B</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Itanium has not taken off in marketplace</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Intel’s 64-Bit

- 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 also 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 (now “Intel 64”)
  - Almost identical to x86-64!

- All but low-end x86 processors support x86-64
  - But, lots of code still runs in 32-bit mode

Our Coverage

- x86-32/IA32
  - The traditional x86

- x86-64/EM64T/AMD64/Intel 64/x64
  - The emerging standard

- Presentation
  - Book presents IA32 in Sections 3.1—3.12
  - Covers x86-64 in 3.13
  - We will cover both interleaved
  - Labs will be mostly based on IA32

Today: Machine Programming I: Basics

- History of Intel processors and architectures

- C, assembly, machine code

- Assembly Basics: Registers, operands, move

- Intro to x86-64

Definitions

- Architecture: (also instruction set architecture: ISA) The parts of a processor design that one needs to understand to write assembly code.
  - Examples: instruction set specification, registers.

- Microarchitecture: Implementation of the architecture.
  - Examples: cache sizes and core frequency.

- Example ISAs (Intel): x86, Itanium
Assembly Programmer’s View

- CPU
  - 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

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

- Assembler (gcc or as)
  - Compiles .s files
  - Produces .o files

- Linker (gcc or ld)
  - 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

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
             addl 8(%ebp),%eax
             popl %ebp
             ret
```

Some compilers use instruction “leave”

Obtain with command
```bash
/usr/bin/gcc -O1 -S code.c
```

Produces file code.s

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

C program (p1.c p2.c)
Compiler (gcc -S) (includes preproc.)

Asm program (p1.s p2.s)
Assembler (gcc or as)

Object program (p1.o p2.o)
Linker (gcc or ld)

Executable program (p)

Static libraries (.a)

Turning C into Object Code

Code for sum
```
0x401040 <sum>:  0xe5
  0x55
  0x89
  0xe5
  0xe8
  0xe5
  0x0c
  0x03
  0xe5
  0xc8
  0xe8
  0xc3
```

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**
  - 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 0x80483ca

```c
int t = x+y;

addl (%ebp), %eax
```

Disassembling Object Code

- **Disassembled**
  - 0x80483c4 <sum):
    - push %ebp
  - 0x80483c5:
    - mov %esp,%ebp
  - 0x80483c7:
    - 8b 4c 05 mov 0x8(%ebp),%eax
  - 0x80483ca:
    - addl 8(%ebp),%eax
  - 0x80483ce:
    - ret

- **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**
  - Dump of assembler code for function sum:
    - 0x80483c4 <sum+0>:
      - push %ebp
    - 0x80483c5 <sum+1>:
      - mov %esp,%ebp
    - 0x80483c7 <sum+3>:
      - mov 0xc(%ebp),%eax
    - 0x80483ca <sum+6>:
      - add 0x8(%ebp),%eax
    - 0x80483ce <sum+9>:
      - ret

- **Within gdb Debugger**
  - gdb p
disassemble sum
  - Disassemble procedure
  - x/11xb sum
  - Examine the 11 bytes starting at sum

Aside: x86 Assembly Formats

- **This class uses “AT&T” format, which is standard for Unix/Linux x86 systems**
  - Similar to historic Unix all the way back to PDP-11
- **Intel’s own documentation, and Windows, use a different “Intel” syntax**
  - Many arbitrary differences, but more internally consistent

<table>
<thead>
<tr>
<th>AT&amp;T syntax</th>
<th>Intel syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destination &amp; last operand</td>
<td>Destination &amp; first operand</td>
</tr>
<tr>
<td>Size suffix like “l” in mov</td>
<td>Size on memory operands (“DWORD PTR”)</td>
</tr>
<tr>
<td>% on register names</td>
<td>Just letters in register names</td>
</tr>
<tr>
<td>’t’ on immediate values</td>
<td>Just digits in immediates</td>
</tr>
<tr>
<td>Addressing modes with (.)</td>
<td>Addressing modes with (+)</td>
</tr>
</tbody>
</table>

Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Intro to x86-64
Integer Registers (IA32)

<table>
<thead>
<tr>
<th>Register</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>Accumulator</td>
</tr>
<tr>
<td>%edx</td>
<td>Counter</td>
</tr>
<tr>
<td>%ecx</td>
<td>Data</td>
</tr>
<tr>
<td>%ebx</td>
<td>Base</td>
</tr>
<tr>
<td>%esi</td>
<td>Source index</td>
</tr>
<tr>
<td>%edi</td>
<td>Destination index</td>
</tr>
<tr>
<td>%esp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>%ebp</td>
<td>Base pointer</td>
</tr>
<tr>
<td>%ax</td>
<td>8/16-bit virtual registers (backwards compatibility)</td>
</tr>
<tr>
<td>%cx</td>
<td></td>
</tr>
<tr>
<td>%dx</td>
<td></td>
</tr>
<tr>
<td>%bx</td>
<td></td>
</tr>
<tr>
<td>%si</td>
<td></td>
</tr>
<tr>
<td>%di</td>
<td></td>
</tr>
<tr>
<td>%sp</td>
<td></td>
</tr>
<tr>
<td>%bp</td>
<td></td>
</tr>
</tbody>
</table>

8/16-bit virtual registers (backwards compatibility)

Moving Data: IA32

- **Moving Data**
  - `movl` `Source`, `Dest`

- **Operand Types**
  - **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)`
    - Various other "address modes"

Operand Combinations

- `movl` `Imm`, `Reg`
- `movl` `Reg`, `Mem`
- `movl` `Reg`, `Mem`
- `movl` `Reg`, `Mem`
- `movl` `Reg`, `Mem`

Cannot do memory-memory transfer with a single instruction

Simple Memory 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;
}
```

**Set Up**
- `pushl %ebp`
- `movl %esp,%ebp`
- `pushl %ebx`
- `movl 8(%ebp),%edx`
- `movl 12(%ebp),%ecx`
- `movl (%edx),%ebx`
- `movl (%ecx),%eax`
- `movl %eax,(%edx)`
- `movl %edx,(%ecx)`

**Body**
- `popl %ebx`
- `popl %ebp`

**Finish**
- `ret`

Using Simple Addressing Modes

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

**Set Up**
- `pushl %ebp`
- `movl %esp,%ebp`
- `pushl %ebx`
- `movl 8(%ebp),%edx`
- `movl 12(%ebp),%ecx`
- `movl (%edx),%ebx`
- `movl (%ecx),%eax`
- `movl %eax,(%edx)`
- `movl %edx,(%ecx)`

**Body**
- `popl %ebx`
- `popl %ebp`

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

Register Value

movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx  # ecx = yp
movl (%edx), %ebx  # ebx = *xp (t0)
movl (%ecx), %eax  # eax = *yp (t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0

movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx  # ecx = yp
movl (%edx), %ebx  # ebx = *xp (t0)
movl (%ecx), %eax  # eax = *yp (t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0

movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx  # ecx = yp
movl (%edx), %ebx  # ebx = *xp (t0)
movl (%ecx), %eax  # eax = *yp (t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0

movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx  # ecx = yp
movl (%edx), %ebx  # ebx = *xp (t0)
movl (%ecx), %eax  # eax = *yp (t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0

movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx  # ecx = yp
movl (%edx), %ebx  # ebx = *xp (t0)
movl (%ecx), %eax  # eax = *yp (t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0

movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx  # ecx = yp
movl (%edx), %ebx  # ebx = *xp (t0)
movl (%ecx), %eax  # eax = *yp (t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0

movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx  # ecx = yp
movl (%edx), %ebx  # ebx = *xp (t0)
movl (%ecx), %eax  # eax = *yp (t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0

movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx  # ecx = yp
movl (%edx), %ebx  # ebx = *xp (t0)
movl (%ecx), %eax  # eax = *yp (t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0

movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx  # ecx = yp
movl (%edx), %ebx  # ebx = *xp (t0)
movl (%ecx), %eax  # eax = *yp (t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0

movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx  # ecx = yp
movl (%edx), %ebx  # ebx = *xp (t0)
movl (%ecx), %eax  # eax = *yp (t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0
Understanding Swap

Complete Memory Addressing Modes

Today: Machine Programming I: Basics

Data Representations: IA32 + x86-64

---

**Understanding Swap**

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x124</td>
</tr>
<tr>
<td>%edx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x11c</td>
</tr>
<tr>
<td>%ebx</td>
<td>0x118</td>
</tr>
<tr>
<td>%esi</td>
<td>0x114</td>
</tr>
<tr>
<td>%edi</td>
<td>0x110</td>
</tr>
<tr>
<td>%esp</td>
<td>0x10c</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x108</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x100</td>
</tr>
</tbody>
</table>

**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
- **S**: Scale: 1, 2, 4, or 8 (why these numbers?)

**Special Cases**

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

---

**Today: Machine Programming I: Basics**

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Intro to x86-64

---

**Data Representations: IA32 + x86-64**

**Sizes of C Objects (in Bytes)**

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Generic 32-bit</th>
<th>Intel IA32</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>8</td>
<td>10/12</td>
<td>16</td>
</tr>
<tr>
<td>char *</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

---

**Logistics Break: Turning In**

- **Lab 1 is due tonight by 11:55pm**
  - (A few of you have already submitted it, congrats)
  - Reminder: make sure driver.pl works!
- **Homework assignment 1 is due at the beginning of class (3:35pm) Monday**
  - Only option for full credit is turning in on paper at beginning of class
  - Late paper submissions accepted through end of lecture
  - All other late submissions must be online on the Moodle
- **Other homework notes:**
  - Only problems 3 and 4 need be submitted for grading
  - A computer printout is strongly recommended/requested
x86-64 Integer Registers

<table>
<thead>
<tr>
<th>rax</th>
<th>rdx</th>
</tr>
</thead>
<tbody>
<tr>
<td>r8x</td>
<td>r16x</td>
</tr>
<tr>
<td>r9x</td>
<td>r17x</td>
</tr>
<tr>
<td>r10x</td>
<td>r18x</td>
</tr>
<tr>
<td>r11x</td>
<td>r19x</td>
</tr>
<tr>
<td>r12x</td>
<td>r20x</td>
</tr>
<tr>
<td>r13x</td>
<td>r21x</td>
</tr>
<tr>
<td>r14x</td>
<td>r22x</td>
</tr>
<tr>
<td>r15x</td>
<td>r23x</td>
</tr>
</tbody>
</table>

- Extend existing registers. Add 8 new ones.
- Make %ebp/%rbp general purpose

Instructions

- Long word l (4 Bytes) ↔ Quad word q (8 Bytes)
- New instructions:
  - movl ↔ movq
  - addl ↔ adds
  - sall ↔ salq
  - etc.
- 32-bit instructions that generate 32-bit results
  - Set higher order bits of destination register to 0
  - Example: addi

64-bit code for swap

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

- Operands passed in registers (why useful?)
  - First (xp) in rdi, second (yp) in rsi
  - 64-bit pointers
  - No stack operations required
  - 32-bit data
    - Data held in registers %eax and %edx
    - movl operation

Machine Programming I: Summary

- 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
- Assembly Basics: Registers, operands, move
  - The x86 move instructions cover wide range of data movement forms
- Intro to x86-64
  - A major departure from the style of code seen in IA32