Today: Machine Programming: Basics

• History of Intel processors and architectures (brief)

• C, assembly, machine code

• Assembly Basics: Registers, operands, move

• Intro to x86-64

Intel x86 Processors

• Totally dominate laptop/desktop/server 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!
    • In terms of speed. Less so for low power.

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>386</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

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>X86-16</td>
<td>8086, 286</td>
</tr>
<tr>
<td>X86-32/IA32</td>
<td>386, 486</td>
</tr>
<tr>
<td>MMX</td>
<td>Pentium, Pentium MMX</td>
</tr>
<tr>
<td>SSE</td>
<td>Pentium III</td>
</tr>
<tr>
<td>SSE2</td>
<td>Pentium 4</td>
</tr>
<tr>
<td>SSE3</td>
<td>Pentium 4E</td>
</tr>
<tr>
<td>X86-64 / EM64t</td>
<td>Pentium 4F</td>
</tr>
<tr>
<td>SSE4</td>
<td>Core 2 Duo, Core i7</td>
</tr>
</tbody>
</table>

IA: often redefined as latest Intel architecture

More Information

• Intel processors (Wikipedia)
• Intel microarchitectures
**Our Coverage**

- IA32
  - The traditional x86
- x86-64/EM64T
  - The emerging standard

**Presentation**

- Book presents IA32 in Sections 3.1—3.12
- Covers x86-64 in 3.13
- We will cover IA32 and look briefly at x86-64 simultaneously

---

**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.

---

**Today: Machine Programming: Basics**

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

---

**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`

---

**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 %ebp, %esp
    addl $12(%ebp), %eax
    addl %ebx, %eax
    popl %ebp
    ret
```

**Note:** Due to gcc version differences, the code generated on our course machine is different (although equivalent).
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

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

- Object Code
  - 3-byte instruction
  - Stored at address 0x80483ca

Disassembling Object Code

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

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

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)

- \( \%eax \)
- \( \%ecx \)
- \( \%edx \)
- \( \%ebx \)
- \( \%esi \)
- \( \%edi \)
- \( \%esp \)
- \( \%ebp \)
- \( \%ax \)
- \( \%cx \)
- \( \%dx \)
- \( \%bx \)
- \( \%si \)
- \( \%di \)
- \( \%sp \)
- \( \%bp \)
- \( \%ah \)
- \( \%ch \)
- \( \%dh \)
- \( \%bh \)
- \( \%al \)
- \( \%cl \)
- \( \%dl \)
- \( \%bl \)

16-bit virtual registers (backwards compatibility)

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

16-bit virtual registers

moving Data: IA32

- Moving Data
  - `movl Source, Dest`

- Operand Types
  - Immediate: Constant integer data
    - Example: $0x400, 0x533
    - Like C constant, but prefixed with `$`
    - 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" – we will see them soon

Operand Combinations

- \( \text{movl imm \{Reg, Mem\}} \)
  - \( \text{movl imm, \%eax} \) \text{temp} = 0x4;
  - \( \text{movl imm, (\%eax)} \) \text{p} = -\text{147};
- \( \text{movl reg \{Reg, Mem\}} \)
  - \( \text{movl \%eax, \%edx} \) \text{temp} = \text{temp1};
  - \( \text{movl \%eax, (\%edx)} \) \text{p} = \text{temp};
- \( \text{movl mem \{Reg, Mem\}} \)
  - \( \text{movl (\%eax), \%edx} \) \text{temp} = \text{p};

No memory to memory transfer using single instruction!

Simple Memory Addressing Modes

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

Note: the normal mode is a special case of displacement mode in which \( D = 0 \)

Using Simple Addressing Modes

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

Set Up
- push \%ebp
- movl \%esp, \%ebp
- movl \%esp, \%ebx
- movl 8(\%ebp), \%edx
- movl 12(\%ebp), \%ecx
- movl (\%edx), \%bx
- movl (\%ecx), \%ax
- movl \%eax, \%edx
- movl \%ebx, (\%ecx)
- popl \%ebx
- popl \%ebp
- ret

Body
- push \%ebp
- movl \%esp, \%ebp
- movl \%esp, \%ebx
- movl 8(\%ebp), \%edx
- movl 12(\%ebp), \%ecx
- movl (\%edx), \%bx
- movl (\%ecx), \%ax
- movl \%eax, (\%edx)
- movl \%ebx, (\%ecx)
- popl \%ebx
- popl \%ebp
- ret

Finish

Using Simple Addressing Modes
Understanding Swap

void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *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 %ebx, %eax # eax = *xp (t0)
movl %ecx, %edx # edx = xp
movl 8(%ebp), %edx # edx = xp
movl 12(%ebp), %ecx # ecx = yp
movl (%edx), %ebx # ebx = *xp (t0)
movl %ebx, %ecx # ecx = yp (t1)
movl %ecx, %edx # edx = xp
movl 8(%ebp), %ecx # ecx = yp
movl 12(%ebp), %edx # edx = xp
movl (%ecx), %ebx # ebx = *yp (t1)
movl %ebx, %edx # edx = xp
movl 8(%ebp), %edx # edx = xp
movl 12(%ebp), %ecx # ecx = yp
movl (%edx), %ebx # ebx = *xp (t0)
movl %ebx, %ecx # ecx = yp (t1)
movl %ecx, %edx # edx = xp
movl 8(%ebp), %ecx # ecx = yp
movl 12(%ebp), %edx # edx = xp
movl (%ecx), %ebx # ebx = *yp (t1)
movl %ebx, %ecx # ecx = xp (t1)
movl %ecx, %edx # edx = xp
movl 8(%ebp), %ecx # ecx = xp
movl 12(%ebp), %edx # edx = xp
movl (%ecx), %ebx # ebx = *xp (t0)
movl %ebx, %ecx # ecx = xp (t1)
movl %ecx, %edx # edx = xp
movl 8(%ebp), %ecx # ecx = xp
movl 12(%ebp), %edx # edx = xp
movl (%ecx), %ebx # ebx = *xp (t0)
movl %ebx, %ecx # ecx = xp (t1)
movl %ecx, %edx # edx = xp
movl 8(%ebp), %edx # edx = xp
movl 12(%ebp), %ecx # ecx = yp
movl (%edx), %ebx # ebx = *xp (t0)
movl %ebx, %ecx # ecx = yp (t1)
movl %ecx, %edx # edx = xp
movl 8(%ebp), %ecx # ecx = yp
movl 12(%ebp), %edx # edx = xp
movl (%ecx), %ebx # ebx = *yp (t1)
movl %ebx, %ecx # ecx = xp (t1)
movl %ecx, %edx # edx = xp
movl 8(%ebp), %edx # edx = yp
movl 12(%ebp), %ecx # ecx = xp
movl (%edx), %ebx # ebx = *yp (t1)
movl %ebx, %ecx # ecx = xp (t1)
movl %ecx, %edx # edx = yp
movl 8(%ebp), %ecx # ecx = yp
movl 12(%ebp), %edx # edx = xp
movl (%ecx), %ebx # ebx = *yp (t1)
movl %ebx, %ecx # ecx = xp (t1)
movl %ecx, %edx # edx = yp
movl 8(%ebp), %edx # edx = yp
movl 12(%ebp), %ecx # ecx = xp
movl (%edx), %ebx # ebx = *yp (t1)
movl %ebx, %ecx # ecx = xp (t1)
movl %ecx, %edx # edx = yp
movl 8(%ebp), %ecx # ecx = xp
movl 12(%ebp), %edx # edx = yp
movl (%ecx), %ebx # ebx = *xp (t0)
movl %ebx, %ecx # ecx = yp (t1)
movl %ecx, %edx # edx = xp
movl 8(%ebp), %ecx # ecx = xp
movl 12(%ebp), %edx # edx = yp
movl (%ecx), %ebx # ebx = *xp (t0)
movl %ebx, %ecx # ecx = yp (t1)
movl %ecx, %edx # edx = xp
movl 8(%ebp), %ecx # ecx = xp
movl 12(%ebp), %edx # edx = yp
movl (%ecx), %ebx # ebx = *xp (t0)
Complete Memory Addressing Modes

- Most General Form
  \[ D(Rb,Ri,S) \quad \text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]+D] \]

  - D: Constant “displacement”/“offset”
  - 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

- Special Cases
  \[
  \begin{align*}
  (Rb,Ri) & \quad \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]] \\
  D(Rb,Ri) & \quad \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]+D] \\
  (Rb,Ri,S) & \quad \text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]]
  \end{align*}
  \]

Data Representations: IA32 + x86-64

- Sizes of C Objects (in Bytes)

  \[
  \begin{array}{llll}
  \text{C Data Type} & \text{Generic 32-bit} & \text{Intel IA32} & \text{x86-64} \\
  \hline
  \text{unsigned} & 4 & 4 & 4 \\
  \text{int} & 4 & 4 & 4 \\
  \text{long int} & 4 & 4 & 8 \\
  \text{char} & 1 & 1 & 1 \\
  \text{short} & 2 & 2 & 2 \\
  \text{float} & 4 & 4 & 4 \\
  \text{double} & 8 & 8 & 8 \\
  \text{long double} & 8 & 10/12 & 16 \\
  \text{char *} & 4 & 4 & 8 \\
  \end{array}
  \]

  - Or any other pointer

x86-64 Integer Registers

- 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` ↔ `addq`
  * `sall` ↔ `salq`
  * etc.

• 32-bit instructions that generate 32-bit results
  * Set higher order bits of destination register to 0
  * Example: `addl`

32-bit code for swap

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

64-bit code for swap

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

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