Today: Machine Programming I: Basics

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

Intel x86 Processors

- 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 a subset encountered with Linux programs
  - Matches performance of more modern Reduced Instruction Set Computers (RISC)
  - In terms of speed. Less so for low power consumption.

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 4E</td>
<td>2004</td>
<td>125M</td>
<td>2800-3800</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1995</td>
<td>6.5M</td>
<td>2.4-2.55 GHz</td>
</tr>
<tr>
<td>Pentium III</td>
<td>1999</td>
<td>8.2M</td>
<td>3.3-3.6 GHz</td>
</tr>
<tr>
<td>Pentium 4</td>
<td>2001</td>
<td>42M</td>
<td>4-4.5 GHz</td>
</tr>
<tr>
<td>Core 2 Duo</td>
<td>2006</td>
<td>291M</td>
<td>2.0-3.2 GHz</td>
</tr>
<tr>
<td>Core i7</td>
<td>2008</td>
<td>731M</td>
<td>2.6-2.9 GHz</td>
</tr>
</tbody>
</table>

Intel x86 Processors, cont.

- Added Features
  - Instructions to support multimedia operations
  - Instructions to enable more efficient conditional operations
  - Transition from 32 bits to 64 bits
  - More cores

2015 State of the Art

- Core i7 Broadwell 2015

Desktop Model

- 4 cores
- Integrated graphics
- 3.3-3.8 GHz
- 65W

Server Model

- 8 cores
- Integrated I/O
- 2.2-2.6 GHz
- 45W
**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
- **Recent Years**
  - Intel got its act together
  - Leads the world in semiconductor technology
  - AMD has fallen behind
  - Spun off its semiconductor factories

**Intel’s 64-Bit History**

- **2001: Intel Attempts Radical Shift from IA32 to IA64**
  - Totally different architecture (Itanium)
  - Executes IA32 code only as legacy
  - Performance disappointing
- **2003: AMD Steps 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 (now called “Intel 64”)
  - Almost identical to x86-64!
  - All but lowest-end x86 processors support x86-64
  - But, lots of code still runs in 32-bit mode

**Our Coverage**

- **IA32**
  - The traditional x86
  - For 2021: RIP, Summer 2015
- **x86-64**
  - The standard
  - `csealabs > gcc hello.c`
  - `csealabs > gcc -m64 hello.c`
- **Presentation**
  - Book covers x86-64
  - Web aside on IA32
  - We will only cover x86-64

**Today: Machine Programming I: Basics**

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

**Definitions**

- **Architecture:** (also ISA: instruction set architecture) The parts of a processor design that one needs to understand or write assembly/machine code.
  - Examples: instruction set specification, registers.
- **Microarchitecture:** Implementation of the architecture.
  - Examples: cache sizes and core frequency.
- **Code Forms:**
  - Machine Code: The byte-level programs that a processor executes
  - Assembly Code: A text representation of machine code
- **Example ISAs:**
  - Intel: x86, IA32, Itanium, x86-64
  - ARM: Used in almost all smartphones

**Assembly/Machine Code View**

- **Programmer-Visible State**
  - PC: Program counter
  - Address of next instruction
  - On x86-64, called “RIP”
  - Register file
  - Heavily used program data
  - Condition codes
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching
- **Memory**
  - Byte addressable array
  - Code and user data
  - Stack to support procedures
Turning C into Object Code

- Code in files p1.c p2.c
- Compile with command: gcc -Og p1.c p2.c -o p
- Use basic optimizations (-Og) [New since GCC 4.8]
- Put resulting binary in file p

<table>
<thead>
<tr>
<th>text</th>
<th>Compiler (gcc -Og -S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asm</td>
<td>program (p1.s p2.s)</td>
</tr>
<tr>
<td>binary</td>
<td>Assembler (gcc or a)</td>
</tr>
<tr>
<td>binary</td>
<td>Object program (p1.o p2.o)</td>
</tr>
<tr>
<td>binary</td>
<td>Executable program (p)</td>
</tr>
</tbody>
</table>

Obtain (on Ubuntu 14.04 machine) with command

gcc -Og -S sum.c

Produces file sum.s

Note: You may get different results on different machines (older Linux, Mac OS X, ...) due to different versions of gcc and different compiler settings.

Compiling Into Assembly

C Code (sum.c) Generated x86-64 Assembly

long plus (long x, long y):
void sumstore (long x, long y, long *dest) {
   long t = plus (x, y);
   *dest = t;
}

sumstore: 
   pushq %rbx
   movq %rdx, %rbx
   call plus
   movq %rax, (%rbx)
   popq %rbx
   ret

Assembly Characteristics: Data Types

- "Integer" data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses (untyped pointers)

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

- Code: Byte sequences encoding series of instructions

- 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 sumstore

| 0xa400595: | 0xa53 |
| 0xa48:    | 0xa99 |
| 0xa3:     | 0x88  |
| 0xe2:     | 0xe1  |
| 0xff:     | 0xff  |
| 0xef:     | 0xef  |
| 0xe8:     | 0xe8  |
| 0xe3:     | 0xe3  |
| 0x48:     | 0x48  |
| 0x99:     | 0x9a  |
| 0xda:     | 0x5b  |
| 0xe3:     | 0x3   |

- Assembler
  - Translates .c 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

| 0xa40059e: | 48 89 03 |

- C Code
  - Store value t where designated by dest

- Assembly
  - Move 8-byte value to memory
  - Quad words in Intel parlance
  - Operands:
    - t: Register %rax
    - dest: Register %rbx
  - *dest: Memory M[%rbx]

- Object Code
  - 3-byte instruction
  - Stored at address 0xa40059e
Disassembled Object Code

Disassembler

<table>
<thead>
<tr>
<th>Assembly Instruction</th>
<th>Disassembled Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>push %rbx</td>
<td>0x53</td>
</tr>
<tr>
<td>mov %rax,%rbx</td>
<td>0x89 0d 0b</td>
</tr>
<tr>
<td>retq</td>
<td>0xc3</td>
</tr>
</tbody>
</table>

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

Legal note: reverse engineering of commercial software is often forbidden by license agreements, and its status under statute varies by jurisdiction

objdump -d sum

What Can Be Disassembled?

Anything that can be interpreted as executable code

Disassembler examines bytes and reconstructs assembly source

Legal note: reverse engineering of commercial software is often forbidden by license agreements, and its status under statute varies by jurisdiction

Alternate Disassembly

Disassembled

% objdump -d WINWORD.EXE

WINWORD.EXE: .file format pei -i386

No symbols in:

Disassembly of:

30001000 <.text>:

30001000: 55 push %ebp

30001001: 8b ec mov %esp,%ebp

30001003: 6a 08 push 0x8

30001005: 68 90 10 00 30 push 0x30900100

3000100a: 68 91 dc 4c 30 push 0x304c4d91

Within gdb Debugger

% gdb sum

(gdb) disassemble sumstore

Disassemble procedure

(gdb) x/x14xb sumstore

Examine the 14 bytes starting at sumstore

Today: Machine Programming I: Basics

History of Intel processors and architectures

C, assembly, machine code

Assembly Basics: Registers, operands, move

Arithmetic & logical operations

x86-64 Integer Registers

Can reference low-order 4 bytes (also low-order 1 & 2 bytes)
Some History: IA32 Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>Accumulate</td>
</tr>
<tr>
<td>%ecx</td>
<td>Counter</td>
</tr>
<tr>
<td>%edx</td>
<td>Data</td>
</tr>
<tr>
<td>%ebx</td>
<td>Base</td>
</tr>
<tr>
<td>%esi</td>
<td>Source</td>
</tr>
<tr>
<td>%edi</td>
<td>Index</td>
</tr>
<tr>
<td>%esp</td>
<td>Destination</td>
</tr>
<tr>
<td>%ebp</td>
<td>Stack pointer</td>
</tr>
</tbody>
</table>

Origin (mostly obsolete)

16-bit virtual registers (backward compatibility)

Moving Data

- **Moving Data**
  - `movq 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 16 integer registers
    - Example: `%rax, %rdx`
    - But `%rsp` reserved for special use
    - Some others have special uses for particular instructions
  - **Memory**: 8 consecutive bytes of memory at address given by register
    - Simplest example: `( %rax )`
    - Various other "address modes"

Operand Combinations

```
movq Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src,Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cannot do memory-memory transfer with a single instruction
```

Simple Memory Addressing Modes

- **Normal** `(R)` Mem[Reg[R]]
  - Register R specifies memory address
  - Like pointer dereferencing in C
  - Movq `%rcx`, `%rax`

- **Displacement** `(R)` Mem[Reg[R]+D]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset
  - Movq 8(`%rbp`), `%rdx`

Example of Simple Addressing Modes

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

Understanding Swap()

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

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td></td>
</tr>
<tr>
<td>%rsi</td>
<td></td>
</tr>
<tr>
<td>%rax</td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>xp</td>
</tr>
<tr>
<td>%rsi</td>
<td>yp</td>
</tr>
<tr>
<td>%rax</td>
<td>t0</td>
</tr>
<tr>
<td>%rdx</td>
<td>t1</td>
</tr>
</tbody>
</table>

Movq `( %rdi ), %rax` # t0 = *xp
Movq `( %rsi ), %rdx` # t1 = *yp
Movq `( %rdx ), %rdi` # *xp = t0
Movq `( %rsi ), %rax` # *yp = t0
Ret
Understanding Swap()

Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td>0x123</td>
</tr>
<tr>
<td>%rdx</td>
<td>456</td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>123</td>
</tr>
<tr>
<td>0x118</td>
<td>0x110</td>
</tr>
<tr>
<td>0x108</td>
<td>123</td>
</tr>
<tr>
<td>456</td>
<td>0x100</td>
</tr>
</tbody>
</table>

swap:

- `movq (rdi),rax` # t0 = *xp
- `movq (rsi),rdx` # t1 = *yp
- `movq rdx,(rdi)` # *xp = t1
- `movq rax,(rsi)` # *yp = t0
- `ret`

Complete Memory Addressing Modes

**Most General Form**

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

- **D**: Constant "displacement" 1, 2, or 4 bytes
- **Rb**: Base register: Any of 16 integer registers
- **Ri**: Index register: Any, except for %rsp
- **S**: Scale: 1, 2, 4, or 8 (why these numbers?)

**Special Cases**

- `(Rb,Ri)`
- `D(Rb,Ri)`
- `(Rb,Ri,S)`

These cases include:

- `Mem[Reg[Rb]+D]`
- `Mem[Reg[Rb]+Reg[Ri]]`
- `Mem[Reg[Rb]+Reg[Ri]+D]`
- `Mem[Reg[Rb]+S*Reg[Ri]]`
### Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%rdx)</td>
<td>0xf000 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>(%rdx, %rcx)</td>
<td>0xf000 + 0x100</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%rdx, %rcx, 4)</td>
<td>0xf000 + 4*0x100</td>
<td>0xf400</td>
</tr>
<tr>
<td>0x80(%rdx, 2)</td>
<td>2*0xf000 + 0x80</td>
<td>0x1e080</td>
</tr>
</tbody>
</table>

### Logistics announcements

- **Exercise set #1 is out now**
  - Due on paper at the beginning of Monday's lecture
- **HA2 on data operations coming soon**
  - Continuation of today’s lab
  - To be due Friday, October 5th

### Today: Machine Programming I: Basics

- **History of Intel processors and architectures**
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- ** Arithmetic & logical operations**

### Address Computation Instruction

- **leaq Src, Dest**
  - Src is address mode expression
  - Set Dest to address denoted by expression

- **Uses**
  - Computing addresses without a memory reference
  - E.g., translation of `p = &x[1];`
  - Computing arithmetic expressions of the form `x + k*y`
    - `k = 1, 2, 4, or 8`

- **Example**

```
long m12(long x) {
    return x*12;
}
```

Converted to ASM by compiler:

```
leaq (%rdi, %rdi, 2), %rax # t <- x*12
salq $2, %rax # return t<<2
```

### Some Arithmetic Operations

- **Two Operands Instructions:**

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq</td>
<td>Dest = Dest + Src</td>
</tr>
<tr>
<td>subq</td>
<td>Dest = Dest – Src</td>
</tr>
<tr>
<td>imulq</td>
<td>Dest = Dest * Src</td>
</tr>
<tr>
<td>shrq</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>xorq</td>
<td>Dest = Dest ^ Src</td>
</tr>
<tr>
<td>andq</td>
<td>Dest = Dest &amp; Src</td>
</tr>
<tr>
<td>orq</td>
<td>Dest = Dest</td>
</tr>
</tbody>
</table>

*Also called salq*

### Watch out for argument order!
- No distinction between signed and unsigned int (why?)
Some Arithmetic Operations

- **One Operand Instructions**
  - `incq Dest`: Dest = Dest + 1
  - `decq Dest`: Dest = Dest - 1
  - `negq Dest`: Dest = -Dest
  - `notq Dest`: Dest = ~Dest

- **See book for more instructions**

Arithmetic Expression Example

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = t1+z;
  long t3 = x*4;
  long t4 = y*48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>t1, t2, rval</td>
</tr>
<tr>
<td>%r12</td>
<td>t4</td>
</tr>
<tr>
<td>%r13</td>
<td>t5</td>
</tr>
</tbody>
</table>

Interesting Instructions
- `leaq`: address computation
- `salq`: shift
- `imulq`: multiplication
  - But, only used once

Machine Programming I: Summary

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

- **C, assembly, machine code**
  - New forms of visible state: program counter, registers, ...
  - Compiler must transform statements, expressions, procedures into low-level instruction sequences

- **Assembly Basics: Registers, operands, move**
  - The x86-64 move instructions cover wide range of data movement forms

- **Arithmetic**
  - C compiler will figure out different instruction combinations to carry out computation