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>Core 2</td>
<td>2006</td>
<td>291M</td>
<td>1060-3500</td>
</tr>
<tr>
<td>Core i7</td>
<td>2008</td>
<td>731M</td>
<td>1700-3900</td>
</tr>
</tbody>
</table>

Intel x86 Processors, cont.

- Machine Evolution
  - 386 1985 0.3M
  - Pentium 1993 3.1M
  - Pentium/MMX 1997 4.5M
  - Pentium Pro 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
  - 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
    - 2-2.6 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
  - `cslabs> gcc hello.c`
  - `cslabs> 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**

- CPU: Registers, Condition Codes
- Addresses: Data Instructions
- Memory: Code Data Stack

- 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
Compiling Into Assembly

**C Code (sum.c)**

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

**Generated x86-64 Assembly**

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

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.

---

**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

---

**Object Code**

**Code for sumstore**

```
0xa400595:  0x53 0x48 0x99 0xe8 0xe2 0x2f 0x0f 0xe8 0x48 0x99 0xe3 0x9b 0x5b 0xe3
```

**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**

```
*dest = t;
```

**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`
### Disassembling Object Code

**Disassembled**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000000000400595 &lt;sumstore&gt;:</td>
<td>push %rbx</td>
</tr>
<tr>
<td>0x0000000000400596:</td>
<td>mov %rdx,%rbx</td>
</tr>
<tr>
<td>0x0000000000400599:</td>
<td>callq 400590 &lt;plus&gt;</td>
</tr>
<tr>
<td>0x000000000040059d:</td>
<td>mov %rax,(%rbx)</td>
</tr>
<tr>
<td>0x00000000004005a1:</td>
<td>pop %rbx</td>
</tr>
<tr>
<td>0x00000000004005a2:</td>
<td>retq</td>
</tr>
</tbody>
</table>

- **Disassembler**
  - objdump -d sum
    - 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**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0400595:</td>
<td>push %rbx</td>
</tr>
<tr>
<td>0x0400596:</td>
<td>mov %rdx,%rbx</td>
</tr>
<tr>
<td>0x0400599:</td>
<td>callq 0x400590 &lt;plus&gt;</td>
</tr>
<tr>
<td>0x040059d:</td>
<td>mov %rax,(%rbx)</td>
</tr>
<tr>
<td>0x04005a1:</td>
<td>pop %rbx</td>
</tr>
<tr>
<td>0x04005a2:</td>
<td>retq</td>
</tr>
</tbody>
</table>

- **Within gdb Debugger**
  - % gdb sum (gdb) disassemble sumstore
  - Disassemble procedure
  - (gdb) x/14xb sumstore
  - Examine the 14 bytes starting at sumstore

### What Can Be Disassembled?

```
% objdump -d WINWORD.EXE

WINWORD.EXE: File format pei
No symbols in
Disassembly of:
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

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

- Anything that can be interpreted as executable code
- Disassembler examines bytes and reconstructs assembly source

### Aside: x86 Assembly Formats

- This class uses “AT&T” format, which is standard for Unix/Linux x86(-64) 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 is last operand</td>
<td>Destination is first operand</td>
</tr>
<tr>
<td>Size suffixes like <code>l</code> in mov</td>
<td>Size on memory operands (&quot;DWORD PTR&quot;)</td>
</tr>
<tr>
<td><code>%</code> on register names</td>
<td>Just letters in register names</td>
</tr>
<tr>
<td><code>&quot;</code> on immediate values</td>
<td>Just digits in immediates</td>
</tr>
<tr>
<td>Addressing modes with (``)</td>
<td>Addressing modes with [<code>*</code>]</td>
</tr>
</tbody>
</table>

### 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

<table>
<thead>
<tr>
<th>64-bit registers</th>
<th>32-bit sub-sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%r8</td>
</tr>
<tr>
<td>%rbx</td>
<td>%r9</td>
</tr>
<tr>
<td>%rcx</td>
<td>%r10</td>
</tr>
<tr>
<td>%rdx</td>
<td>%r11</td>
</tr>
<tr>
<td>%rsi</td>
<td>%r12</td>
</tr>
<tr>
<td>%rdi</td>
<td>%r13</td>
</tr>
<tr>
<td>%rESP</td>
<td>%r14</td>
</tr>
<tr>
<td>%rbp</td>
<td>%r15</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>General purpose</th>
<th>Origin (mostly obsolete)</th>
<th>Some History: IA32 Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r eax$</td>
<td>$ax$</td>
<td>$%eax$逃避</td>
</tr>
<tr>
<td>$rcx$</td>
<td>$cx$</td>
<td>$%ecx$逃避</td>
</tr>
<tr>
<td>$rdx$</td>
<td>$dx$</td>
<td>$%edx$逃避</td>
</tr>
<tr>
<td>$rbx$</td>
<td>$bx$</td>
<td>$%ebx$逃避</td>
</tr>
<tr>
<td>$rsi$</td>
<td>$si$</td>
<td>$%esi$逃避</td>
</tr>
<tr>
<td>$rdi$</td>
<td>$di$</td>
<td>$%edi$逃避</td>
</tr>
<tr>
<td>$rsp$</td>
<td>$sp$</td>
<td>$%esp$逃避</td>
</tr>
<tr>
<td>$rbp$</td>
<td>$bp$</td>
<td>$%ebp$逃避</td>
</tr>
<tr>
<td>$rN$</td>
<td></td>
<td>$%esp$逃避</td>
</tr>
</tbody>
</table>

16-bit virtual registers (backwards 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`, `$%r13`
    - But `$%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”

### Simple Memory Addressing Modes

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

- **Displacement** (`D(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

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

### Understanding Swap()

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

#### Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</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>123</td>
</tr>
<tr>
<td>%rdx</td>
<td>456</td>
</tr>
</tbody>
</table>

#### Memory

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
</tr>
<tr>
<td>0x118</td>
</tr>
<tr>
<td>0x110</td>
</tr>
<tr>
<td>0x108</td>
</tr>
<tr>
<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

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

- **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

- \((Rb, Ri)\)  
  \(\text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]]\)
- \(\text{D}(Rb, Ri)\)  
  \(\text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]+D]\)
- \((Rb, Ri, S)\)  
  \(\text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]]\)
### 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></td>
<td></td>
</tr>
<tr>
<td>(%rdx,%rcx)</td>
<td>0x000 + 0x8</td>
<td>0x000</td>
</tr>
<tr>
<td>(%rdx,%rcx,4)</td>
<td>0x0000 + 4*0x100</td>
<td>0x0f000</td>
</tr>
<tr>
<td>0x80(%rdx,2)</td>
<td>2*0x0000 + 0x80</td>
<td>0x1e080</td>
</tr>
</tbody>
</table>

### 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[i];`
  - Computing arithmetic expressions of the form `x + k*y`
    - `k = 1, 2, 4, or 8`

- **Example**

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

  Converted to ASM by compiler:

  ```asm
  leaq (%rdi,%rdi,2), %rax
  salq $2, %rax
  ```

### Some Arithmetic Operations

- **Two Operand Instructions:**
  - **Format**
    - `addq Src, Dest` Dest = Dest + Src
    - `subq Src, Dest` Dest = Dest − Src
    - `imulq Src, Dest` Dest = Dest * Src
    - `shlq Src, Dest` Dest = Dest << Src
    - `srrq Src, Dest` Dest = Dest >> Src
    - `xorq Src, Dest` Dest = Dest ^ Src
    - `andq Src, Dest` Dest = Dest & Src
    - `orq Src, Dest` Dest = Dest | Src

- **Watch out for argument order!**
- **No distinction between signed and unsigned int (why?)**

- **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 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

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

Understanding Arithmetic Expression Example

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

Register Use(s)
- `%rdi`: Argument `x`
- `%rsi`: Argument `y`
- `%rdx`: Argument `z`
- `%rax`: `t1, t2, rval`
- `%r8`: `t4`
- `%rcx`: `t5`

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