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>
<th>Washington 2015 State of the Art</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>1978</td>
<td>29K</td>
<td>5-10</td>
<td>Core i7 Broadwell 2015</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Desktop Model</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8 cores</td>
</tr>
<tr>
<td>386</td>
<td>1985</td>
<td>4.5M</td>
<td>2800-3800</td>
<td>Integrated I/O</td>
</tr>
<tr>
<td>386</td>
<td>1985</td>
<td>275K</td>
<td>16-33</td>
<td>3.3-3.8 GHz</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>3.1M</td>
<td>1060-3500</td>
<td>65W</td>
</tr>
<tr>
<td>Pentium/MMX</td>
<td>1997</td>
<td>4.5M</td>
<td></td>
<td>Server Model</td>
</tr>
<tr>
<td>PentiumPro</td>
<td>1995</td>
<td>6.5M</td>
<td></td>
<td>8 cores</td>
</tr>
<tr>
<td>Pentium III</td>
<td>1999</td>
<td>8.2M</td>
<td></td>
<td>Integrated I/O</td>
</tr>
<tr>
<td>Pentium 4</td>
<td>2001</td>
<td>42M</td>
<td></td>
<td>2.2-6 GHz</td>
</tr>
<tr>
<td>Core 2 Duo</td>
<td>2006</td>
<td>291M</td>
<td></td>
<td>45W</td>
</tr>
<tr>
<td>Core i7</td>
<td>2008</td>
<td>731M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core 2 Duo</td>
<td>2006</td>
<td>291M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core i7</td>
<td>2008</td>
<td>731M</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Intel x86 Processors, cont.

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

- Desktop Model
  - 8 cores
  - Integrated I/O
  - 2.2-6 GHz
  - 45W

- Server Model
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
  - Comprehensive coverage of x86-64
  - cselabs> gcc hello.c
  - cselabs> 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

- 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`

### 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

### 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**
```
0x0400595:
  0x53
  0x48
  0x89
  0x33
  0x88
  0x02
  0xff
  0xff
  0xff
  0x48
  0x48
  0x89
  0x03
  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
- **C Code**
  ```c
  *dest = t;
  ```
- **Assembler**
  ```assembly
  movq %rax, (%rbx)
  ```

- **Object Code**
  ```assembly
  0x040059e: 48 89 03
  ```

- **C Code**
  ```c
  Store value t where designated by dest
  ```
- **Assembly**
  Move 8-byte value to memory
  ```assembly
  movq %rax, (%rbx)
  ```
  - Quad words in Intel parlance
  - Operands:
    - `t`: Register `%rax`
    - `dest`: Register `%rbx`
    - `Memory[M%rbx]`

- **Object Code**
  ```assembly
  0x040059e: 48 89 03
  ```
  - 3-byte instruction
  - Stored at address `0x040059e`
Disassembling Object Code

Disassembled

```
0x0000000000400595 <sumstore>:
  0400595:  53               push %rbx
  0400596:  48 89 d3         mov %rcx,%rbx
  0400599:  w8 f2 ff ff ff   callq 0400590 <plus>
  040059e:  48 89 03         mov %rax,(%rbx)
  04005a1:  5b               pop %rbx
  04005a2:  c3               retq
```

- 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

```
0x0000000000400595: Dump of assembler code for function sumstore:
  0x0000000000400595 <+0>: push %rbx
  0x0000000000400596 <+1>: mov %rdx,%rbx
  0x0000000000400599 <+4>: callq 0x400590 <plus>
  0x000000000040059e <+9>: mov %rax,(%rbx)
  0x00000000004005a1 <+12>: pop %rbx
  0x00000000004005a2 <+13>: retq
```

- Within `gdb` Debugger
  - `% gdb sum (gdb) disassemble sumstore`
  - Disassemble procedure
  - Examine the 14 bytes starting at `sumstore`

What Can Be Disassembled?

```
% objdump -d WINWORD.EXE
WINWORD.EXE:      file format pei-32bit

No symbols in Disassembly of WINWORD.EXE

30001000 <.text>:
  30001000:  90 00 30 10 push 0x30001090
  30001004:  41 4c 91 dc 00 push $0x00414c91
```

- 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

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 “l” in <code>movl</code></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>“$” 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>

x86-64 Integer Registers

```
%rax %eax
%rbx %ebx
%rcx %ecx
%rdx %edx
%rsi %esi
%rdi %edi
%r8 %r8d
%r9 %r9d
%r10 %r10d
%r11 %r11d
%r12 %r12d
%r13 %r13d
%r14 %r14d
%r15 %r15d
```

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)

Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
Some History: IA32 Registers

- General purpose registers
  - %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)
  - accumulator
  - counter
  - data
  - base
  - source
  - index
  - destination
  - stack
  - pointer
  - base

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"

Operand Combinations

- `movq` Source, Dest:
  - `movq $0x4, %rax`  \( \text{temp} = 0x4; \)
  - `movq $-147, (%rax)` \( \text{*p} = -147; \)

Simple Memory Addressing Modes

- Normal \((R)\)  \(\text{Mem[Reg[R]]}\)
  - Register R specifies memory address
  - Like pointer dereferencing in C
  - Example:
    - `movq (%rcx), %rax`

- Displacement \((D)\)  \(\text{Mem[Reg[R]+D]}\)
  - Register R specifies start of memory region
  - Constant displacement D specifies offset
  - Example:
    - `movq 8(%rhp), %rhp`

Example of Simple Addressing Modes

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

Understanding Swap()

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

- Registers
  - `%rdi`, `%rsi`
  - `%rax`, `%rdx`

- Memory
  - `%rax`, `%rdx`
  - `%rsi`, `%rdi`
Understanding Swap()

**Registers**
- %rdi 0x120
- %rsi 0x100
- %rax 123
- %rdx 456

**Memory**
- Address 0x120
  - 0x118
  - 0x110
  - 0x108
  - 0x100

**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**
  - \( D(Rb,Ri) \) → Mem[Reg[Rb]+Reg[Ri]]
  - \( D(Rb,Ri) \) → Mem[Reg[Rb]+Reg[Ri]+D]
  - \( (Rb,Ri,S) \) → Mem[Reg[Rb]+S*Reg[Ri]]
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></td>
<td></td>
</tr>
<tr>
<td>(%rdx,%rcx,4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x80(%rdx,2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Address Computation Instruction

- **leaq Src, Dst**
  - Src is address mode expression
  - Set Dst 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**

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

  Converted to ASM by compiler:

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

Some Arithmetic Operations

- **Two Operand 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>salq</td>
<td>Dest = Dest &lt;&lt; Src</td>
</tr>
<tr>
<td>sarq</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>shrq</td>
<td>Dest = Dest &gt;&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>

- **One Operand Instructions**

  | Incq | Dest = Dest + 1 |
  | Decq | Dest = Dest - 1|
  | Neq  | Dest = ~Dest   |

- **See book for more instructions**

Arithmetic Expression Example

```c
long arith
(long x, long y, long z)
{
  long t1 = x*y;
  long t2 = z*t1;
  long t3 = x*z;
  long t4 = y * 48;  // 48 = 2 * 2 * 2 * 3
  long t5 = t3 + t4;
  long rval = t2 + t5;
  return rval;
}
```
Understanding Arithmetic Expression Example

```c
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;
}
```

<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>%rdx</td>
<td>t4</td>
</tr>
<tr>
<td>%rcx</td>
<td>t5</td>
</tr>
</tbody>
</table>

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