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</td>
<td>1993</td>
<td>3.1M</td>
<td></td>
</tr>
<tr>
<td>Pentium/MMX</td>
<td>1997</td>
<td>4.5M</td>
<td></td>
</tr>
<tr>
<td>PentiumPro</td>
<td>1995</td>
<td>6.5M</td>
<td></td>
</tr>
<tr>
<td>Pentium III</td>
<td>1999</td>
<td>8.2M</td>
<td></td>
</tr>
<tr>
<td>Pentium 4</td>
<td>2001</td>
<td>42M</td>
<td></td>
</tr>
<tr>
<td>Core 2 Duo</td>
<td>2006</td>
<td>291M</td>
<td></td>
</tr>
<tr>
<td>Core i7</td>
<td>2008</td>
<td>731M</td>
<td></td>
</tr>
<tr>
<td>Pentium 4E</td>
<td>2004</td>
<td>125M</td>
<td>2800-3800</td>
</tr>
<tr>
<td>Pentium 4</td>
<td>2006</td>
<td>291M</td>
<td>1060-3500</td>
</tr>
<tr>
<td>Core 2</td>
<td>2008</td>
<td>731M</td>
<td>1700-3900</td>
</tr>
<tr>
<td>Core i7</td>
<td>2008</td>
<td>731M</td>
<td>1700-3900</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
  - 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

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
  - `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

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`

```text
C program (p1.c p2.c)
```

Compiler (`gcc`)

```text
Assembler (gcc or `as`)
```

Linker (`gcc` or `ld`)

C program (`p1.c p2.c`)

Asm program (`p1.s p2.s`)

Object program (`p1.o p2.o`)

Executable program (`p`)

Static libraries (`a`)

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

C Code for `sumstore`

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

Machine Instruction Example

- C Code
  - Store value `t` where designated by `dest`

```
*dest = t;
```

Assembly

- Move 8-byte value to memory
  - Quad words in Intel parlance
- Operands:
  - `t`: Register `r8`
  - `dest`: Register `rbx`

```
    movq rax, (%rbx)
```

Object Code

- 3-byte instruction
  - Stored at address `0x40059e`

```
    0x40059e:  48 89 03
```

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.
Disassembling Object Code

Disassembled

0000000000400595 <sumstore>:
400595: 53 push %rbx
400596: 48 89 d3 mov %rax,%rbx
400599: 48 89 d3 mov %rax,%rbx
4005a1: 5b pop %rbx
4005a2: 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

What Can Be Disassembled?

- 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

AT&T syntax | Intel syntax
--- | ---
Destination is last operand | Destination is first operand
Size suffixes like “l” in `movl` | Size on memory operands (“DWORD PTR”)
“%” on register names | Just letters in register names
“$” on immediate values | Just digits in immediates
Addressing modes with (,) | Addressing modes with [+]
x86-64 Integer Registers

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

Some History: IA32 Registers

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

16-bit virtual registers
(backwards compatibility)

Moving Data

- Moving Data
  movq Source, Dest:

Operand Types

- Immediate: Constant integer data
  - Example: $0x100, $-533
  - Like C constant, but prefixed with `$`
  - Encoded with 1, 2, or 4 bytes
- Register: One of 16 integer registers
  - Example: %rax, %r13
  - But %rbp reserved for special use
- Memory: 8 consecutive bytes of memory address given by register
  - Simplest example: (%rax)
  - Various other "address modes"

movq Operand Combinations

Source    Dest    Src,Dest    C Analog

- imm
  movq $0x4,%rax
  temp = 0x4;

- reg
  movq %rax,%rdx
  *p = %rax;

- mem
  movq 8(%rbp),%rdx
  temp = 8[%rbp];

Example of Simple 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

void swap
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}

swap:
    movq (%rdi, %rax)
    movq (%rdi, %rax)
    movq 0,%rax
    movq (%rdi, %rax)
    movq (%rdi, %rax)
    movq (%rdi, %rax)
    movq (%rdi, %rax)
    ret
void swap (long *xp, long *yp) {
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}

Understanding Swap()

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

swap:
  movq (%rdi), %rax # t0 = *xp
  movq (%rsi), %rdx # t1 = *yp
  movq %rdx, (%rdi) # *xp = t1
  movq %rax, (%rsi) # *yp = t0
  ret

Understanding Swap()

Registers | Memory | Address
---|---|---
%rdi | 0x120 | 0x120
%rsi | 0x100 | 0x118
%rax | 123 | 0x110
%rdx | 456 | 0x108

swap:
  movq (%rdi), %rax # t0 = *xp
  movq (%rsi), %rdx # t1 = *yp
  movq %rdx, (%rdi) # *xp = t1
  movq %rax, (%rsi) # *yp = t0
  ret

Understanding Swap()

Registers | Memory | Address
---|---|---
%rdi | 0x120 | 0x120
%rsi | 0x100 | 0x118
%rax | 123 | 0x110
%rdx | 456 | 0x108

swap:
  movq (%rdi), %rax # t0 = *xp
  movq (%rsi), %rdx # t1 = *yp
  movq %rdx, (%rdi) # *xp = t1
  movq %rax, (%rsi) # *yp = t0
  ret

Understanding Swap()

Registers | Memory | Address
---|---|---
%rdi | 0x120 | 0x120
%rsi | 0x100 | 0x118
%rax | 123 | 0x110
%rdx | 456 | 0x108

swap:
  movq (%rdi), %rax # t0 = *xp
  movq (%rsi), %rdx # t1 = *yp
  movq %rdx, (%rdi) # *xp = t1
  movq %rax, (%rsi) # *yp = t0
  ret

Understanding Swap()

Registers | Memory | Address
---|---|---
%rdi | 0x120 | 0x120
%rsi | 0x100 | 0x118
%rax | 123 | 0x110
%rdx | 456 | 0x108

swap:
  movq (%rdi), %rax # t0 = *xp
  movq (%rsi), %rdx # t1 = *yp
  movq %rdx, (%rdi) # *xp = t1
  movq %rax, (%rsi) # *yp = t0
  ret

Understanding Swap()

Registers | Memory | Address
---|---|---
%rdi | 0x120 | 0x120
%rsi | 0x100 | 0x118
%rax | 123 | 0x110
%rdx | 456 | 0x108

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

Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdx 0x000</td>
<td>0x000 + 0x8</td>
<td>0x0008</td>
</tr>
<tr>
<td>%rcx 0x100</td>
<td>0x000 + 0x100</td>
<td>0x0100</td>
</tr>
<tr>
<td>(%rdx,%rcx)</td>
<td>0x000 + 4*0x100</td>
<td>0x0400</td>
</tr>
<tr>
<td>(%rdx,%rcx,4)</td>
<td>0x000 + 4*0x100</td>
<td>0x0400</td>
</tr>
<tr>
<td>0x80(,%rdx,2)</td>
<td>2*0x0000 + 0x80</td>
<td>0x1e080</td>
</tr>
</tbody>
</table>

Address Computation Instruction

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

- **Uses**
  
  - Computing addresses without a memory reference
  - Example: translation of \( p = 4x[1] \)
  - 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, %rdi, 2), %rax # t <- x*12
  salq $2, %rax # return t<<2
  ```

Today: Machine Programming I: Basics

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

Some Arithmetic Operations

- **Two Operand Instructions:**
  
<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq</td>
<td>Src, Dest</td>
</tr>
<tr>
<td>subq</td>
<td>Src, Dest</td>
</tr>
<tr>
<td>imulq</td>
<td>Src, Dest</td>
</tr>
<tr>
<td>salq</td>
<td>Src, Dest</td>
</tr>
<tr>
<td>sarq</td>
<td>Src, Dest</td>
</tr>
<tr>
<td>xorq</td>
<td>Src, Dest</td>
</tr>
<tr>
<td>andq</td>
<td>Src, Dest</td>
</tr>
<tr>
<td>orq</td>
<td>Src, Dest</td>
</tr>
</tbody>
</table>
  
  Also called shl
  Also called shr
  Arithmetic
  Logical

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

- **One Operand Instructions**
  - `incq` Dest = Dest + 1
  - `decq` Dest = Dest - 1
  - `negq` Dest = -Dest
  - `notq` 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 + 4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
leaq (%rdi, %rsi), %rax  # t1
addq %rdx, %rax  # t2
leaq (%rax, %rax, 2), %rdx  # t3
salq $4, %rdx  # t4
leaq 4(%rdi, %rdx), %rcx  # t5
imulq %rcx, %rax  # rval
ret
```

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

Understanding Arithmetic Expression Example

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

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