Chapter 3
Machine-Level Programming I: Basics

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 4</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 Transistors (Moore’s Law)**
  - 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

Recent Architecture
- 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
    - Relies on external semiconductor manufacturer
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 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
  - Almost identical to x86-64!
- All but low-end x86 processors support x86-64
  - But, lots of code still runs in 32-bit mode

Our Coverage
- IA32
  - The traditional x86
- x86-64
  - The standard
    - gcc hello.c
    - gcc -m64 hello.c
- Presentation
  - Book covers x86-64
  - Web aside on IA32
  - We will only cover x86-64

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 mobile phones

Assembly/Machine Code View

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 to recent versions of GCC]
  - Put resulting binary in file p

  text
  - C program (p1.c p2.c)
  - Compiler (gcc -Og -S)
  - Assembler (gcc or as)
  - Linker (gcc or 1d)

  binary
  - Object program (p1.o p2.o)
  - Static libraries (-s)
  - Executable program (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

```
sumstore:            pushq  %rbx
            movq  %rdx, %rbx
            callq  plus
            movq  %rax, (%rbx)
popq  %rbx
retq
```

Obtain with command

```
gcc -Og -S sum.c
```

Produces file sum.s

Warning: May get very different results on other types of machines (Andrew 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

```
0x0400595:  53    pushq  %rbx
0x0400596:  48 89 d3  movq  %rdx, %rbx
0x0400599:  e8 f2 ff ff ff  callq  400590 <plus>
0x040059e:  48 89 03  movq  %rax, (%rbx)
0x04005a1:  5b    popq  %rbx
0x04005a2:  c3    retq
```

Disassembling Object Code

Disassembled

```
000000000400595 <sumstore>:  pushq  %rbx
400595:  48 89 d3  movq  %rdx, %rbx
400599:  e8 f2 ff ff ff  callq  400590 <plus>
40059a:  48 89 03  movq  %rax, (%rbx)
4005aa:  5b    popq  %rbx
4005ab:  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

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

Machine Instruction Example

```
*dest = t;
```

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

```
0x40059a:  48 89 03
```

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

```
0x0400595:  53    pushq  %rbx
0x0400596:  48 89 d3  movq  %rdx, %rbx
0x0400599:  e8 f2 ff ff ff  callq  400590 <plus>
0x040059e:  48 89 03  movq  %rax, (%rbx)
0x04005a1:  5b    popq  %rbx
0x04005a2:  c3    retq
```

Disassembling Object Code

Disassembled

```
000000000400595 <sumstore>:  pushq  %rbx
400595:  48 89 d3  movq  %rdx, %rbx
400599:  e8 f2 ff ff ff  callq  400590 <plus>
40059a:  48 89 03  movq  %rax, (%rbx)
4005aa:  5b    popq  %rbx
4005ab:  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
Disassembled Dump of assembler code for function sumstore:

```assembly
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` disassemble `sumstore`
- Disassemble procedure `x/14xb sumstore`
- Examine the 14 bytes starting at `sumstore`

**x86-64 Integer Registers**

<table>
<thead>
<tr>
<th>rax</th>
<th>rdx</th>
<th>rbx</th>
<th>rsi</th>
<th>rdi</th>
</tr>
</thead>
<tbody>
<tr>
<td>tr8</td>
<td>tr9</td>
<td>tr10</td>
<td>tr11</td>
<td>tr12</td>
</tr>
<tr>
<td>tr13</td>
<td>tr14</td>
<td>tr15</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

**Moving Data**

```assembly
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, r12`
  - But `%esp reserved for special use`
  - 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"

**Some History: IA32 Registers**

<table>
<thead>
<tr>
<th>eax</th>
<th>ecx</th>
<th>edx</th>
</tr>
</thead>
<tbody>
<tr>
<td>trax</td>
<td>trbx</td>
<td>trcx</td>
</tr>
</tbody>
</table>

- `%esp` reserved for special use
- 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"
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>Imm</td>
<td>Reg</td>
<td>movq $0x4, %rax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq $-147, (%rax)</td>
<td>*p = -147;</td>
</tr>
</tbody>
</table>

movq

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;
}
```

Simple Memory Addressing Modes

- **Normal** (R) **Mem[Reg[R]]**
  - Register R specifies memory address
  - Aha! Pointer dereferencing in C
    ```c
    movq (%rdx), %rax
    ```

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

Understanding Swap()

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

```c
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 | 0x00 | 0x100
%rsi 0x100 | 0x110 | 0x108
%rax 123 | 0x180 | 0x108
%rdx 456 | 0x120 | 0x100

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

Simple Memory Addressing Modes

- **Normal (R) Mem[Reg[R]]**
  - Register R specifies memory address
  - Ahh! 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

Complete Memory Addressing Modes

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

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>0x0000 + 0x8</td>
<td>0x0008</td>
</tr>
<tr>
<td>(%rdx,%rcx)</td>
<td>0x0000 + 0x100</td>
<td>0x0100</td>
</tr>
<tr>
<td>(%rdx,%rcx,4)</td>
<td>0x0000 + 4*0x100</td>
<td>0x0400</td>
</tr>
<tr>
<td>0x80,(%rdx,2)</td>
<td>2*0x0000 + 0x80</td>
<td>0x1a080</td>
</tr>
</tbody>
</table>
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, Dst**
  - load effective address quad
  - Src is address mode expression
  - Set Dst to address denoted by expression

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

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

  Converted to ASM by compiler:
  ```asm
  leaq (%rdi,%rsi,2), %rax
  addq $2, %rax
  ```

Some Arithmetic Operations

- **Two-Operand Instructions**
  - Format
  - Computation
  - **addq** Src, Dest
    - Dest = Dest + Src
  - **subq** Src, Dest
    - Dest = Dest - Src
  - **imulq** Src, Dest
    - Dest = Dest * Src
  - **salq** Src, Dest
    - Dest = Dest << Src
  - **xorq** Src, Dest
    - Dest = Dest ^ Src
  - **andq** Src, Dest
    - Dest = Dest & Src
  - **orq** Src, Dest
    - Dest = Dest | Src

  **Also called shlq**
  **Arithmetic**

  **Logical**

  **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

  **See book for more instructions**

Arithmetic Expression Example

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

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

Understanding Arithmetic Expression Example

```c
long arith (long x, long y, long z)
{
    long t1 = x + y;
    long t2 = y * 4;
    long t3 = t2 + 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>%rdi</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