Machine-Level Representation: Basic
CSCI 2021: Machine Architecture and Organization
Pen-Chung Yew
Department Computer Science and Engineering
University of Minnesota
http://www.cs.umn.edu/~yew

With Slides from Randy Bryant, David O’Hallaron, and Antonia Zhai

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 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 small subset encountered with Linux programs
  • Hard to match performance of Reduced Instruction Set Computers (RISC)
  • But, Intel has done just that!
  • In terms of speed. Less so for low power.

Intel x86 Processors: Evolution
• Machine Evolution
  • 8086 1978 29K
  • 386 1985 200K
  • Pentium 1993 31M
  • Pentium/MMX 1997 45M
  • PentiumPro 1995 63M
  • Pentium III 1999 82M
  • Pentium 4 2001 422M
  • 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

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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>386</td>
<td>1985</td>
<td>275K</td>
<td>16-33</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium</td>
<td>2004</td>
<td>125M</td>
<td>2800-3800</td>
</tr>
<tr>
<td>Pentium</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>

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
    - 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 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
    - Almost identical to x86-64!
  - All but low-end x86 processors support x86-64

Our Coverage

- IA32
  - The traditional x86
  - For 15/18-213: RIP, Summer 2015
- x86-64
  - The standard
  - `shark> gcc hello.c`
  - `shark> 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

Programmer-Visible State
- PC: Program counter
  - Address of next instruction
  - Called “RIP” (x86-64)
- 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 to recent versions of GCC]
  - Put resulting binary in file `p`

**Text**

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

**Assembler (gcc or as)**

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

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

**Linker (gcc or 1d)**

**Static libraries (.a)**

**Executable program (p)**

Compiling Into Assembly

**C Code (sum.c)**

```
long plus(long x, long y):
    long t = plus(x, y);
    *dest = t;
```

**Generated x86-64 Assembly**

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

- Obtain (on shark machine) with command
  - `gcc -Og -S sum.c`
- Produces file `sum.s`
- Warning: Will get very different results on non-Shark 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

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**

```
0x040059e:  48 89 03
```

- Assembler
  - Translates .S 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**

```
*dest = t;
```

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

**Assembly**

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

- Move 8-byte value to memory
  - Quad words in x86-64 parlance
- Operands:
  - `t`: Register `%rax`
  - `dest`: Register `%rbx`
  - `%dest`: Memory `%rbx`

**Object Code**

- 3-byte instruction
- Stored at address `0x040059e`
Disassembled

Disassembled 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

0000000000400595 <sumstore>:
400595:  53     push   %rbx
400596:  48 89 d3 mov %rdx,%rbx
400599:  e8 f2 ff ff ff callq 0x400590 <plus>
40059e:  48 89 03 mov %rax,(%rbx)
4005a1:  5b     pop    %rbx
4005a2:  c3     retq

• Disassembler
  objdump --d sum
  • Useful tool for examining object code
  • Examines bit pattern of series of instructions
  • Produces approximate rendition of assembly code
  • Can be run on either a.out (complete executable) or .o file

• Within gdb Debugger
  gdb sum
  disassemble sumstore
  • Disassemble procedure
  x/14xb sumstore
  • Examine the 14 bytes starting at sumstore

What Can be Disassembled?

Reverse engineering forbidden by Microsoft End User License Agreement

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

x86-64 Integer Registers

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

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

Origin (mostly obsolete)
accumulate counter
base data
data
destination

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
    - Others have special uses for particular instructions
  - **Memory:** 8 consecutive bytes of memory at address given by register
    - Simplest example: `(%rax)`

- **Operand Combination**
  - Cannot do memory-memory transfer with a single instruction

```
movq $0x400,%rax
movq $-147,(%rax)
movq %rax,%rdx
movq (%%rax),%rdx
```

Simple Memory Addressing Modes

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

- **Displacement D(R) Mem[Reg[R]+D]**
  - Register R specifies start of memory region
  - Constant displacement D specifies offset
  - Example: `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;
}
```

movq Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imm</td>
<td>Reg</td>
<td>movq $0x4,%rax temp = 0x4;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq 8-147,(%rax) *p = -147;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq (%%rax),%rdx *p = temp;</td>
</tr>
</tbody>
</table>

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

Registers

<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>

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></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td>456</td>
</tr>
</tbody>
</table>

Example of Swap

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

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

Registers
- rdi: 0x120
- rsi: 0x100
- rax: 0x100
- rdx: 0x100

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

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

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

---

Simple Memory Addressing Modes

- **Normal (R)**:
  - Mem[Reg[R]]
  - Register R specifies memory address
  - Aha! 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)  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)  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>%rdx 0x000</td>
<td>0xf000 + 0x8</td>
<td>0x800</td>
</tr>
<tr>
<td>%rcx 0x0100</td>
<td>0x000 + 0x100</td>
<td>0x100</td>
</tr>
<tr>
<td>0x8(%rdx)</td>
<td>0xf000 + 0x8</td>
<td>0x800</td>
</tr>
<tr>
<td>(%rdx,%rcx)</td>
<td>0xf000 + 4*0x100</td>
<td>0xf400</td>
</tr>
<tr>
<td>0x80(%rdx,%rcx,4)</td>
<td>2*0xf000 + 0x80</td>
<td>0x1e080</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**
  - Src is address mode expression
  - Set Dst to address denoted by expression
- **Uses**
  - Computing addresses without a memory reference
    - E.g., translation of \( p = \text{ax}[]; \)
  - Computing arithmetic expressions of the form \( x + k' y \)
    - \( k \in 1, 2, 4, \text{or} 8 \)
- **Example**

```
long m12(long x)
{
    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;
}
```

Converted to ASM by compiler:

```
leaq (%rdi,%rsi,4), %rax
salq $4, %rdx
leaq 4(%rdi,%,rdx), %rcx
imulq %rcx, %rax
ret
```

### Some Arithmetic Operations

- **Two Operand Instructions:**
  - **addq** Src, Dest
  - **subq** Src, Dest
  - **imulq** Src, Dest
  - **salq** Src, Dest
  - **shrq** Src, Dest
  - **xorq** Src, Dest
  - **andq** Src, Dest
  - **orq** Src, Dest

  Also called **shlq**

  Arithmetic

  Logical

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

### Some 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;
}
```

Interesting Instructions

- **leaq**: address computation
- **salq**: shift
- **imulq**: multiplication
  - But, only used once
Understanding Arith 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;
}
```

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