Machine Level Programming: Basics

Computer Systems Organization (Spring 2016)
CSCI-UA 201, Section 2

Instructor: Joanna Klukowska

Slides adapted from
Randal E. Bryant and David R. O'Hallaron (CMU)
Mohamed Zahran (NYU)

Why do we look at machine code?

- understanding how the high-level programming language instructions are executed on a processor
- understanding how optimizing high-level program affects instructions executed in practice
- understanding security flaws of programs
- understanding things that are not handled at the high-level programming language

We will be working with the machine code for x86-64 processors.

Intel x86 Processors

- Totally 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 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.

We will just scratch the surface of the available instructions.

A Bit of History

Intel x86 Evolution

<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>80386</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 4F</td>
<td>2004</td>
<td>125M</td>
<td>2800-3800</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core i7</td>
<td>2008</td>
<td>731M</td>
<td>2667-3333</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notice that the speed is not increasing as much any more.

Schematic of an Intel Processor
Another schematic of an Intel Processor

- Core i7 Broadwell 2015
- Desktop Model
  - 4 cores
  - Integrated graphics
  - 3.3-3.6 GHz
  - 65W
- Server Model
  - 8 cores
  - Integrated I/O
  - 2.2-2.6 GHz
  - 45W

X86 Close's Advanced Micro Devices (AMD)

- Historically
  - AMD has followed just behind Intel
  - A little bit slower, a bit 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

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 "AM64")
- 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

C, assembly, machine code

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,
  - Target of the compiler
- 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 of a Computer

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

- Sys 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

Compilation Flow:

- C program (p1.c p2.c)
- Asm program (p1.s p2.s)
- Object program (p1.o p2.o)
- Static libraries (.a)
- Executable program (.o)

Compiling into Assembly

```c
long plus(long x, long y) {
    return x + y;
}

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

Assembly code generated using

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

Output written to `sum.s` file.

Assembly Characteristics: Data Types

- "Integers" data of 1, 2, 4, or 8 bytes
  - Data values (it does not matter if it is signed or not at the level of assembly)
  - Addresses (untyped pointers)
- Floating point data of 4, 8, or 10 bytes
  - We will not really go into floating point numbers at the level of assembly
- Code: Byte sequences encoding series of instructions
- No aggregate types such as arrays or structures, just contiguously allocated bytes in memory

Assembly Operations

- Perform arithmetic function on register or memory data
- Transfer data between memory and register
- Load data from memory into register
- Store register's data into memory
- Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches

Very limited in what can be done in one instruction - does only one thing: move data, single simple arithmetic operation, memory dereference.

Object Code

```
0x4005a2:
  0x53
  0x48
  0x89
  0x88
  0xff
  0xff
  0xff
  0xff
  0xff
  0x48
  0x89
  0x83
  0x5b
  0xc3
```

Total of 14 bytes. Each instruction can use a different number of bytes. Small at location 0x4005a2.

- Assembler
  - Translates a into a
  - 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 Instructions - Example

- C Code
  - Store value `t` where designated by dest
    ```c
    *dest = t;
    ```
- Assembly
  - Move 8-byte value to memory
    - Quad words in x86-64 format
    ```
    mov rax, (rbx)
    ```
- Operands:
  - `t`: Register `rax`
  - `dest`: Register `rbx`
  - `*dest`: Memory `[rbx]`
- Object Code
  - 3-byte instruction
  - Stored at address 0x40059e
    ```
    0x40059a: 48 89 03
    ```
Disassembling Object Code

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 file

```
0600000000000000x <sumstore>
46005a2: 53
46005a3: 48 89 d1
46005a4: e8 ff ff ff FF
46005a5: 48 89 03
46005a6: 5b
46005a7: c3
```

Alternate Disassembly

Within gdb Debugger

```
gdb sum
```

- `disassemble sumstore`
- Disassemble procedure
- `x/16 db sumstore`
- Examine the 14 bytes starting at sumstore

```
0x46005a2 <sumstore>
0x46005a5 <sumstore+4>
```

What Can Be Disassembled?

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

BUT:

The end user license agreement for some software forbids reverse engineering of code.

Assembly Basics: Registers, Operands, Move

x86-64 Integer Registers

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%ebx</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
</tr>
</tbody>
</table>

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

IA32 Registers (History)

<table>
<thead>
<tr>
<th>teax</th>
<th>tax</th>
<th>t1h</th>
<th>t1l</th>
</tr>
</thead>
<tbody>
<tr>
<td>scex</td>
<td>ecx</td>
<td>tch</td>
<td>tcl</td>
</tr>
<tr>
<td>sedx</td>
<td>edx</td>
<td>tdh</td>
<td>tdl</td>
</tr>
<tr>
<td>sees</td>
<td>esi</td>
<td>tsh</td>
<td>tdl</td>
</tr>
<tr>
<td>sebp</td>
<td>ebp</td>
<td>tsp</td>
<td>tsh</td>
</tr>
</tbody>
</table>

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: 0x000000, 0x533
    - 32-bit constant, but prefixed with 5
  - Encoded with 1, 2, or 4 bytes
  - Register: One of 16 integer registers
    - Example: `rax, %rdi`
    - Bit %rip reserved for special use
  - 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>Mem</td>
<td>movq %rax,%rax</td>
<td>*p = 147;</td>
</tr>
<tr>
<td>Reg</td>
<td>Reg</td>
<td>movq %rax,%rdx</td>
<td>temp2 = temp;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq %rax,%rdx</td>
<td>*p = temp;</td>
</tr>
</tbody>
</table>

Cannot do memory-memory transfer with a single instruction

Simple Memory Addressing Modes

- Normal (R) Mem[Reg[Reg]]
  - Register R specifies memory address
  - Example: `movq (%rdx), %rax`

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

Note: the normal mode is a special case of displacement mode in which D = 0

Simple Addressing Modes - `swap()` Examples

```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;
}
```

Registers

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</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></td>
</tr>
<tr>
<td>%rdx</td>
<td></td>
</tr>
</tbody>
</table>

Register Value

<table>
<thead>
<tr>
<th>%rdi</th>
<th>%rsi</th>
<th>%rax</th>
<th>%rdx</th>
</tr>
</thead>
<tbody>
<tr>
<td>xp</td>
<td>yp</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

swap:

```c
movq (%rdi), %rax
movq (%rdi), %rdx
movq (%rsi), %rax
movq (%rsi), %rdx
ret
```

Understanding `swap()`

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

Registers

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</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>456</td>
</tr>
<tr>
<td>%rdx</td>
<td></td>
</tr>
</tbody>
</table>

Address

<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:

```c
movq (%rdi), %rax
movq (%rsi), %rax
movq (%rdi), %rdx
movq (%rsi), %rdx
ret
```
Understanding `swap()`

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

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

Simple Memory Addressing Modes

- **Normal (R)** `Mem[Reg[RI]]`
  - Register R specifies memory address
  - Pointer dereferencing in C

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

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

  ```
  movq %rbp, %rax
  ```

Note: the normal mode is a special case of displacement mode in which D = 0

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 %esp
  - `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>0x8(%rdx)</td>
<td>0xf000 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>(%rdx,%rcx)</td>
<td>0xf000 + 0x100</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%rdx,%rcx,4)</td>
<td>0xf000 + 4*0x100</td>
<td>0xf400</td>
</tr>
<tr>
<td>0x80,(%rdx,2)</td>
<td>2^0x0f000 + 0x80</td>
<td>0xe080</td>
</tr>
</tbody>
</table>

Logical and Arithmetic Operations

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>

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

Example: arithmetic expression

```c
long arith (long x, long y, long z) 
{ 
    long t1 = x+y; 
    long t2 = x+1; 
    long t3 = x+y; 
    long t4 = y - t3; 
    long t5 = t3 + t4; 
    long rval = t5 * t5; 
    return rval; 
}
```

Arithmetic Operations

- One Operand Instructions:

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>incq</td>
<td>Dest = Dest + 1</td>
</tr>
<tr>
<td>decq</td>
<td>Dest = Dest - 1</td>
</tr>
<tr>
<td>negq</td>
<td>Dest = -Dest</td>
</tr>
<tr>
<td>notq</td>
<td>Dest = -Dest</td>
</tr>
</tbody>
</table>

- Watch out for argument order!
- No distinction between signed and unsigned int
Example: arithmetic expression

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

```
arith:
  long (rdi, rax, rax) $t1$
  add(tbx, rax) $t2$
  long (rdi, rax, rax) rdx $t4$
  loadq(rdi, rax) $t4$
  loadq (rdi, rdx) rax $tval$
  ret

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>rax</td>
<td>Argument y</td>
</tr>
<tr>
<td>rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>t1, t2</td>
<td>tval</td>
</tr>
<tr>
<td>t4</td>
<td>t4</td>
</tr>
<tr>
<td>t5</td>
<td>t5</td>
</tr>
</tbody>
</table>
```