CSCI-UA.0201

Computer Systems Organization

Machine-Level Programming I

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

Some slides adapted (and slightly modified) from:
• Clark Barrett
• Jinyang Li
• Randy Bryant
• Dave O’Hallaron
Intel x86 Processors

• Evolutionary design
  – Backwards compatible up until 8086, introduced in 1978

• Complex instruction set computer (CISC)
  – Many instructions, many formats
  – By contrast, ARM architecture (in most cell phones and tablets) is RISC
# Intel x86 Evolution: Milestones

<table>
<thead>
<tr>
<th>Name</th>
<th>Transistors</th>
<th>MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086 (1978)</td>
<td>29K</td>
<td>5-10</td>
</tr>
<tr>
<td>• First 16-bit processor. Basis for IBM PC &amp; DOS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• 1MB address space</td>
<td></td>
<td></td>
</tr>
<tr>
<td>386 (1985)</td>
<td>275K</td>
<td>16-33</td>
</tr>
<tr>
<td>• First 32-bit processor, referred to as IA32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Capable of running Unix</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium 4F (2004)</td>
<td>125M</td>
<td>2800-3800</td>
</tr>
<tr>
<td>• First 64-bit processor, referred to as x86-64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core i7 (2008)</td>
<td>731M</td>
<td>2667-3333</td>
</tr>
<tr>
<td>Xeon E7 (2011)</td>
<td>2.2B</td>
<td>~2400</td>
</tr>
</tbody>
</table>

We will cover x86-64.
Example from 2015

- Intel Skylake

- 4-8 cores
- Integrated graph
- 2.4-4.0 GHz
- Integrated I/O
- ~35W-95W
Execution context
- **PC**: Program counter
  - Address of next instruction
  - Called “RIP” (x86-64)
- **Registers**
  - Heavily used program data
- **Condition code registers**
  - 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
Assembly Data Types

• “Integer” data of 1, 2, or 4 bytes
  – Represent either data value
  – or address

• Floating point data of 4, 8, or 10 bytes

• Code: Byte sequences encoding series of instructions

• No arrays or structures
3 Kind of Assembly Operations

• Perform arithmetic on register or memory data
  – Add, subtract, multiplication...

• 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
Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`

```
Optimizer level
Output file is 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 with command

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

Produces file `sum.s`

*Warning*: Will get very different results on different machines due to different versions of gcc and different compiler settings.
Object Code

Code for *sumstore*

0x0400595:
0x53
0x48
0x89
0xd3
0xe8
0xf2
0xff
0xff
0xff
0x48
0x89
0x03
0x5b
0xc3

- Total of 14 bytes
- Each instruction 1, 3, or 5 bytes
- Starts at address 0x0400595

- **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** (look at sum.c 2 slides back)
  – Store value \( t \) where designated by \( \text{dest} \)

\[
*\text{dest} = t;
\]

• **Assembly**
  – Move 8-byte value to memory
    • Quad words in x86-64 parlance
  – Operands:
    \( t: \) Register \( \%rax \)
    \( \text{dest}: \) Register \( \%rbx \)
    \( *\text{dest}: \) Memory \( M[\%rbx] \)

\[
\text{movq } \%rax, (\%rbx)
\]

• **Object Code**
  – 3-byte instruction
  – Stored at address \( 0x40059e \)
Disassembling Object Code

Disassembled

```
0000000000400595 <sumstore>:
  400595:  53       push %rbx
  400596:  48 89 d3 mov %rdx,%rbx
  400599:  e8 f2 ff ff ff callq 400590 <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
  - 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

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
disassemble sumstore
  – Disassemble procedure

  x/14xb sumstore
  – Examine the 14 bytes starting at sumstore

Object

0x0400595:
0x53
0x48
0x89
0xd3
0xe8
0xf2
0xff
0xff
0x48
0x89
0x03
0x5b
0xc3
### x86-64 Integer Registers

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
<th>%r8</th>
<th>%r8d</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
<td>%r15</td>
<td>%r15d</td>
</tr>
</tbody>
</table>

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)
Some History:
Integer Registers (IA32)

- %eax, %ax, %ah, %al
- %ecx, %cx, %ch, %cl
- %edx, %dx, %dh, %dl
- %ebx, %bx, %bh, %bl
- %esi, %si
- %edi, %di
- %esp, %sp
- %ebp, %bp

16-bit virtual registers
(backwards compatibility)

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

```c
movq Source, Dest
```

### Operand Types

- **Immediate**: Constant integer data
  - Example: $0x400, $-533
  - Like C constant, but prefixed with `$`

- **Register**: One of 16 integer registers
  - Example: `%rax, %r13`
  - But `%rsp` reserved for special use
  - Others have special uses for particular instructions (later on that)

- **Memory**: 8 consecutive bytes of memory at address given by register
  - Simplest example: `%rax`
  - We will see various other “address modes” later.
## 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>Imm</td>
<td>Mem</td>
<td>movq $-147, (%rax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td>Reg</td>
<td>Reg</td>
<td>movq %rax, %rdx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td>Reg</td>
<td>Mem</td>
<td>movq %rax, (%rdx)</td>
<td>*p = temp;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq (%rax), %rdx</td>
<td>temp = *p;</td>
</tr>
</tbody>
</table>

No memory-to-memory instruction
<table>
<thead>
<tr>
<th>C Declaration</th>
<th>Intel Data Type</th>
<th>Assembly code suffix</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Char</td>
<td>Byte</td>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>Short</td>
<td>Word</td>
<td>w</td>
<td>2</td>
</tr>
<tr>
<td>Int</td>
<td>Double Word</td>
<td>l</td>
<td>4</td>
</tr>
<tr>
<td>Long</td>
<td>Quad Word</td>
<td>q</td>
<td>8</td>
</tr>
<tr>
<td>Pointer</td>
<td>Quad Word</td>
<td>q</td>
<td>8</td>
</tr>
</tbody>
</table>
Special Type of mov

- **movz** $S, R \rightarrow R = \text{ZeroExtend}(S)$
  - movzbow
  - movzbbl
  - movzbq
  - movzbq
  - movzwll
  - movzwq

- **movs** $S, R \rightarrow R = \text{SignExtend}(S)$
  - movsbbw
  - movsbbll
  - movsbq
  - movsbq
  - movswll
  - movswq
  - movswq
  - movswq

- **S**: memory or register   **R**: register
Yet Another Special Case

• You can have movq $num, %register
  – num is an immediate number (signed)
  – register is any 64-bit register
  – num cannot exceed 32 bits
  – The rest is sign-extended

• movabsq $num, %register
  – Means move num absolute to register
  – num is 64-bit
Simple Memory Addressing Modes

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

  ```
  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

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

swap:
... some setup code
movq (%rdi), %rax
movq (%rsi), %rdx
movq %rdx, (%rdi)
movq %rax, (%rsi)
... wrap-up code
ret
Understanding `swap()`

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

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

Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</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>

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>

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rdi</code></td>
<td>0x120</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>0x100</td>
</tr>
<tr>
<td><code>%rax</code></td>
<td>123</td>
</tr>
<tr>
<td><code>%rdx</code></td>
<td></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>

### Swap:

```
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

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</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>123</td>
</tr>
<tr>
<td>%rdx</td>
<td>456</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>

### swap:

```
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

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</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>123</td>
</tr>
<tr>
<td>%rdx</td>
<td>456</td>
</tr>
</tbody>
</table>

### Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>456</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></td>
</tr>
</tbody>
</table>

### Swap:

```
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

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</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>123</td>
</tr>
<tr>
<td>%rdx</td>
<td>456</td>
</tr>
</tbody>
</table>

### Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>456</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>123</td>
</tr>
</tbody>
</table>

### Code

```assembly
swap:
    movq (%rdi), %rax  # t0 = *xp
    movq (%rsi), %rdx  # t1 = *yp
    movq %rdx, (%rdi)  # *xp = t1
    movq %rax, (%rsi)  # *yp = t0
    ret
```
General Memory Addressing Modes

• Most General Form

\[ D (Rb, Ri, S) \]

- Constant displacement (cannot be bigger than 4 bytes)
- Base register (any of the 16 registers)
- Index register (any if the 16 registers except %rsp)
- Scale (1, 2, 4, 8)

\[ \text{Mem}[\text{Reg}[Rb] + S \times \text{Reg}[Ri] + D] \]

• Special Cases

- \((Rb, Ri)\) \hspace{1cm} \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri]]
- \(D(Rb, Ri)\) \hspace{1cm} \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] + D]
- \((Rb, Ri, S)\) \hspace{1cm} \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>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*0xf000 + 0x80</td>
<td>0x1e080</td>
</tr>
</tbody>
</table>
Address Computation Instruction

• **leaq** *Src, Dst*
  – *Src* is address mode expression
  – Set *Dst* to address calculated for *src*

• **Uses**
  – Computing addresses **without a memory access**
    • E.g., translation of \( p = &x[i] \);
  – 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:

```assembly
leaq (%rdi,%rdi,2), %rax  # t <- x+x*2
salq $2, %rax             # return t<<2
```
Conclusions

• History of Intel processors and architectures
  – Evolutionary design leads to many quirks and artifacts

• C, assembly, machine code
  – Compiler must transform statements, expressions, procedures into low-level instruction sequences

• Assembly Basics: Registers, operands, move
  – The x86 move instructions cover wide range of data movement forms