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) 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></td>
<td>First 16-bit processor. Basis for IBM PC &amp; DOS</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>386 (1985)</td>
<td>275K</td>
<td>16-33</td>
</tr>
<tr>
<td></td>
<td>First 32 bit processor, referred to as IA32</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium 4F (2004)</td>
<td>125M</td>
<td>2800-3800</td>
</tr>
<tr>
<td></td>
<td>First 64-bit processor, referred to as x86-64</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core i7 (2008)</td>
<td>731M</td>
<td>2667-3333</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Xeon E7 (2011)</td>
<td>2.2B</td>
<td>~2400</td>
</tr>
</tbody>
</table>

We will cover x86-64.
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
Assembly Programmer's View

- **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 (untyped pointer)

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

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

- **Object Code**
  - 3-byte instruction
  - Stored at address `0x40059e`

C Code:

```c
*dest = t;
```

Assembly:

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

Object Code:

```
0x40059e:  48 89 03
```
Disassembling Object Code

Disassembled

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>400595:</td>
<td>53</td>
<td>push %rbx</td>
</tr>
<tr>
<td>400596:</td>
<td>48 89 d3</td>
<td>mov %rdx,%rbx</td>
</tr>
<tr>
<td>400599:</td>
<td>e8 f2 ff ff ff</td>
<td>callq 400590 &lt;plus&gt;</td>
</tr>
<tr>
<td>40059e:</td>
<td>48 89 03</td>
<td>mov %rax,(%rbx)</td>
</tr>
<tr>
<td>4005a1:</td>
<td>5b</td>
<td>pop %rbx</td>
</tr>
<tr>
<td>4005a2:</td>
<td>c3</td>
<td>retq</td>
</tr>
</tbody>
</table>

- **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:

Disassembled:

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

<table>
<thead>
<tr>
<th>0x0400595:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x53</td>
</tr>
<tr>
<td>0x48</td>
</tr>
<tr>
<td>0x89</td>
</tr>
<tr>
<td>0xdd</td>
</tr>
<tr>
<td>0xe8</td>
</tr>
<tr>
<td>0xf2</td>
</tr>
<tr>
<td>0xff</td>
</tr>
<tr>
<td>0xff</td>
</tr>
<tr>
<td>0xff</td>
</tr>
<tr>
<td>0x48</td>
</tr>
<tr>
<td>0x89</td>
</tr>
<tr>
<td>0x03</td>
</tr>
<tr>
<td>0x5b</td>
</tr>
<tr>
<td>0xc3</td>
</tr>
</tbody>
</table>
### x86-64 Integer Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%eax</td>
</tr>
<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>%r8</td>
<td>%r8d</td>
</tr>
<tr>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<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
- %ecx
- %edx
- %ebx
- %esi
- %edi
- %esp
- %ebp

General purpose registers:
- %ax %ch %cl
- %cx %dh %dl
- %dx %bh %bl
- %bx
- %si
- %di
- %sp
- %bp

16-bit virtual registers (backwards compatibility):
- %ah %al
- %ch %cl
- %dh %dl
- %bh %bl

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

• Moving Data
  `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)`
    • Various other “address modes”
movq Operand Combinations

movl

<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>
<tr>
<td>Reg</td>
<td>Reg</td>
<td>movq %rax, %rdx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</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

• \text{movz} \ S, \ R \rightarrow \ R = \text{ZeroExtend}(S)
  - \text{movz}bw
  - \text{movz}bl
  - \text{movz}bq
  - \text{movz}wl
  - \text{movz}wq

• \text{movs} \ S, \ R \rightarrow \ R = \text{SignExtend}(S)
  - \text{movs}bw
  - \text{movs}bl
  - \text{movs}bq
  - \text{movs}wl
  - \text{movs}wq
  - \text{movs}lq

• $S$: memory or register    $R$: register
• Note that \text{movz}wq does not exist!
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` can be 64-bit
Simple Memory Addressing Modes

• Normal \((R) \rightarrow \text{Mem}[\text{Reg}[R]]\)
  – Register \(R\) specifies memory address

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

• Displacement \(D(R) \rightarrow \text{Mem}[\text{Reg}[R]+D]\)
  – Register \(R\) specifies start of memory region
  – Constant displacement \(D\) specifies offset

\[
\text{movq} \ 8(\%rbp), \%rdx
\]
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;
}
```

**Memory**

**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>

**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>Address</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:

```assembly
movq    (%rdi), %rax  # t0 = *xp
movq    (%rsi), %rdx  # t1 = *yp
movq    %rdx, (%rdi)  # *xp = t1
movq    %rax, (%rsi)  # *yp = t0
ret
```
Understand *Swap*()
Understanding \texttt{Swap()}

\begin{center}
\begin{tabular}{|c|c|}
\hline
\textbf{Registers} & \textbf{Memory} \\
\hline
%rdi & 0x120 & 123 & 0x120 \\
%rsi & 0x100 & & 0x118 \\
%rax & 123 & & 0x110 \\
%rdx & 456 & & 0x108 \\
\hline
\end{tabular}
\end{center}

\texttt{swap:}
\begin{verbatim}
movq   (%rdi), %rax  # t0 = *xp
movq  (%rsi), %rdx  # t1 = *yp
movq   %rdx, (%rdi)  # *xp = t1
movq   %rax, (%rsi)  # *yp = t0
ret
\end{verbatim}
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:

```
    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>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>

### swap:

```
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 of the 16 registers except %rsp)
  - **Scale** (1, 2, 4, 8)

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

- **Special Cases**

  - \( (Rb, Ri) \)
  - \( D(Rb, Ri) \)
  - \( (Rb, Ri, S) \)

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

- **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:
```
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