CSCI-UA.0201

Computer Systems Organization

Machine Level – Assembly (x86-64) basics

Thomas Wies
wies@cs.nyu.edu
https://cs.nyu.edu/wies
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 a reduced instruction set computer (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 <strong>IA32</strong></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 <strong>x86-64</strong></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 graphics
• 2.4-4.0 GHz
• Integrated I/O
• ~35W-95W
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 values
  – or addresses

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

• Code: Byte sequences encoding series of instructions

• No arrays or structures
3 Kinds of Assembly Operations

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

• 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`

```
C program (p1.c p2.c)
Compiler (gcc -S)
```

```
Asm program (p1.s p2.s)
Assembler (gcc -c)
```

```
Object program (p1.o p2.o)
Linker (ld)
```

```
Executable program (p)
Static libraries (.a)
```
Compiling Into Assembly

C Code (sumstore.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 with command

```
gcc -Og -S sumstore.c
```

Produces file `sumstore.s`

**Warning**: Will get very different results on different machines due to different versions of gcc, different compiler settings, and different hardware architecture.
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
  – 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`
  
  ```c
  *dest = t;
  ```

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

Disassembled

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000400595</td>
<td>&lt;sumstore&gt;:</td>
</tr>
<tr>
<td>400595:</td>
<td>53</td>
</tr>
<tr>
<td>400596:</td>
<td>48 89 d3</td>
</tr>
<tr>
<td>400599:</td>
<td>e8 f2 ff ff ff</td>
</tr>
<tr>
<td>40059e:</td>
<td>48 89 03</td>
</tr>
<tr>
<td>4005a1:</td>
<td>5b</td>
</tr>
<tr>
<td>4005a2:</td>
<td>c3</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

Object

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
x86-64 Integer Registers

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

<table>
<thead>
<tr>
<th>Register</th>
<th>General Purpose</th>
<th>Origin (mostly obsolete)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>%ax %ah %al</td>
<td>accumulate</td>
</tr>
<tr>
<td>%ecx</td>
<td>%cx %ch %cl</td>
<td>counter</td>
</tr>
<tr>
<td>%edx</td>
<td>%dx %dh %dl</td>
<td>data</td>
</tr>
<tr>
<td>%ebx</td>
<td>%bx %bh %bl</td>
<td>base</td>
</tr>
<tr>
<td>%esi</td>
<td>%si</td>
<td>source index</td>
</tr>
<tr>
<td>%edi</td>
<td>%di</td>
<td>dest. index</td>
</tr>
<tr>
<td>%esp</td>
<td>%sp</td>
<td>stack pointer</td>
</tr>
<tr>
<td>%ebp</td>
<td>%bp</td>
<td>base pointer</td>
</tr>
</tbody>
</table>

16-bit virtual registers (backwards compatibility)
Moving Data
### Moving Data

- **Moving Data**

  \[ \text{movq \ Source, \ Dest} \]

### 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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reg</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>Mem</td>
<td>Mem</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></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**No memory-to-memory instruction**
### movq

<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 → R = ZeroExtend(S)**
  
  – movzbw (zero extend byte to word)
  – movzbl (zero extend byte to double word)
  – movzbx (zero extend byte to quad word)
  – movzw (zero extend word to double word)
  – movzwq (zero extend word to quad word)

• **movs S,R → R = SignExtend(S)**

  – movsbw (sign extend byte to word)
  – movsbl (sign extend byte to double word)
  – movsbq (sign extend byte to quad word)
  – movs (sign extend word to double word)
  – movswq (sign extend word to quad word)
  – movslq (sign extend double word to quad word)

• **S**: memory or register  **R**: register