Chapter 3
Machine-Level Programming I: Basics
Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
## 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 4E</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 2</td>
<td>2006</td>
<td>291M</td>
<td>1060-3500</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core i7</td>
<td>2008</td>
<td>731M</td>
<td>1700-3900</td>
</tr>
</tbody>
</table>

- First 16-bit Intel processor. Basis for IBM PC & DOS
- First 32 bit Intel processor, referred to as IA32
- Added “flat addressing”, capable of running Unix
- First 64-bit Intel x86 processor, referred to as x86-64
- First multi-core Intel processor
- Four cores
Intel x86 Processors, cont.

■ Machine Transistors (Moore’s Law)

- 386  1985  0.3M
- Pentium  1993  3.1M
- Pentium/MMX  1997  4.5M
- PentiumPro  1995  6.5M
- Pentium III  1999  8.2M
- Pentium 4  2001  42M
- 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
Recent Architecture

- 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
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 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**
  - But, lots of code still runs in 32-bit mode
Our Coverage

- **IA32**
  - The traditional x86

- **x86-64**
  - The standard
  - `gcc hello.c`
  - `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)`
- `Compiler (gcc -Og -S)`

Text

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

Binary

- `Object program (p1.o p2.o)`
- `Linker (gcc or 1d)`

Binary

- `Executable program (p)`
- `Static libraries (.a)`
### 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:** May get very different results on other types of 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

0x0400595:
0x53
0x48
0x89
0xd3
0xe8
0xff
0xff
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`
Disassembling Object Code

Disassembled

```
000000000400595 <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

Object

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

Disassembled

Dump of assembler code for function sumstore:

0x00000000000400595 <+0>: push %rbx
0x00000000000400596 <+1>: mov %rdx,%rbx
0x00000000000400599 <+4>: callq 0x400590 <plus>
0x0000000000040059e <+9>: mov %rax,(%rbx)
0x000000000004005a1 <+12>: pop %rbx
0x000000000004005a2 <+13>: retq

Within gdb Debugger

- `gdb sum`
- `disassemble sumstore`
  - Disassemble procedure
- `x/14xb sumstore`
  - Examine the 14 bytes starting at sumstore
What Can be Disassembled?

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

% objdump -d WINWORD.EXE

WINWORD.EXE:  file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000: 55  push %ebp
30001001: 8b ec  mov  %esp,%ebp
30001003: 6a ff  push $0xffffffff
30001005: 68 90 10 00 30  push $0x30001090
3000100a: 68 91 dc 4c 30  push $0x304cdc91

Reverse engineering forbidden by Microsoft End User License Agreement
Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
# 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>

<table>
<thead>
<tr>
<th>%r8</th>
<th>%r8d</th>
</tr>
</thead>
<tbody>
<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: IA32 Registers

- **%eax** to **%ehl**: general purpose registers
- **%ax**, **%ah**, **%al**: 16-bit virtual registers (backwards compatibility)
- **%ecx** to **%cl**: general purpose
- **%edx** to **%dl**: general purpose
- **%ebx** to **%bl**: general purpose
- **%esi** to **%si**: general purpose
- **%edi** to **%di**: general purpose
- **%esp** to **%sp**: stack pointer
- **%ebp** to **%bp**: base pointer

**Origin (mostly obsolete):**
- Accumulate
- Counter
- Data
- Base
- Source
- Index
- Destination
- Index

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition
Moving Data

Moving Data

\texttt{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)
  - 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><strong>Imm</strong></td>
<td><em>Reg</em></td>
<td>movq $0x4,%rax</td>
<td><code>temp = 0x4;</code></td>
</tr>
<tr>
<td></td>
<td><em>Mem</em></td>
<td>movq $-147,(%rax)</td>
<td><code>*p = -147;</code></td>
</tr>
<tr>
<td><strong>Reg</strong></td>
<td><em>Reg</em></td>
<td>movq %rax,%rdx</td>
<td><code>temp2 = temp1;</code></td>
</tr>
<tr>
<td></td>
<td><em>Mem</em></td>
<td>movq %rax,(%rdx)</td>
<td><code>*p = temp;</code></td>
</tr>
<tr>
<td><em>Mem</em></td>
<td><em>Reg</em></td>
<td>movq (%rax),%rdx</td>
<td><code>temp = *p;</code></td>
</tr>
</tbody>
</table>

*Cannot do memory-memory transfer with a single instruction*
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
  ```
Example of Simple Addressing Modes

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

```assembly
swap:
    movq (%rdi), %rax
    movq (%rsi), %rdx
    movq %rdx, (%rdi)
    movq %rax, (%rsi)
    ret
```
Understanding Swap()

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

**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**:

- `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>456</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>%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></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>%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:

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

![Registers and Memory Diagram]

**Registers**

<table>
<thead>
<tr>
<th>%rdi</th>
<th>0x120</th>
</tr>
</thead>
<tbody>
<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**

- Address 0x120: 456
- Address 0x118: 
- Address 0x110: 
- Address 0x108: 
- Address 0x100: 456

**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>

**Swap:**

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

- **Normal**  \((R)\)  \(\text{Mem[Reg[R]]}\)
  - Register R specifies memory address
  - Aha! Pointer dereferencing in C

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

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

  \[
  \text{movq} 8(\%rbp),\%rdx
  \]
Complete Memory Addressing Modes

- **Most General Form**
  \[ D(Rb,Ri,S) \rightarrow \text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]+D] \]
  - **D:** Constant “displacement” 1, 2, or 4 bytes
  - **Rb:** Base register: Any of 16 integer registers
  - **Ri:** Index register: Any, except for \%\texttt{rsp}
  - **S:** Scale: 1, 2, 4, or 8 (why these numbers?)

- **Special Cases**
  \[
  \begin{align*}
  (Rb,Ri) & \rightarrow \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]] \\
  D(Rb,Ri) & \rightarrow \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]+D] \\
  (Rb,Ri,S) & \rightarrow \text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]]
  \end{align*}
  \]
## 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>
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
  - load effective address quad
  - Src is address mode expression
  - Set Dst to address denoted by expression

- **Uses**
  - Computing addresses without a memory reference
    - 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
```
Some Arithmetic Operations

- **Two-Operand Instructions**

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addq</code></td>
<td><code>Dest = Dest + Src</code></td>
</tr>
<tr>
<td><code>subq</code></td>
<td><code>Dest = Dest − Src</code></td>
</tr>
<tr>
<td><code>imulq</code></td>
<td><code>Dest = Dest * Src</code></td>
</tr>
<tr>
<td><code>salq</code></td>
<td><code>Dest = Dest &lt;&lt; Src</code></td>
</tr>
<tr>
<td><code>sarq</code></td>
<td><code>Dest = Dest &gt;&gt; Src</code></td>
</tr>
<tr>
<td><code>shrq</code></td>
<td><code>Dest = Dest &gt;&gt; Src</code></td>
</tr>
<tr>
<td><code>xorq</code></td>
<td><code>Dest = Dest ^ Src</code></td>
</tr>
<tr>
<td><code>andq</code></td>
<td><code>Dest = Dest &amp; Src</code></td>
</tr>
<tr>
<td><code>orq</code></td>
<td>`Dest = Dest</td>
</tr>
</tbody>
</table>

- *Also called shlq*
- *Arithmetic*
- *Logical*

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

- **One Operand Instructions**
  
  - `incq Dest`  
    \[ Dest = Dest + 1 \]
  
  - `decq Dest`  
    \[ Dest = Dest - 1 \]
  
  - `negq Dest`  
    \[ Dest = -Dest \]
  
  - `notq Dest`  
    \[ Dest = \overline{Dest} \]

- **See book for more instructions**
Arithmetic 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;
}
```

**arith:**
- `leaq (%rdi,%rsi), %rax`
- `addq %rdx, %rax`
- `leaq (%rsi,%rsi,2), %rdx`
- `salq $4, %rdx`
- `leaq 4(%rdi,%rdx), %rcx`
- `imulq %rcx, %rax`
- `ret`

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

```
arith:
    leaq  (%rdi,%rsi), %rax  # t1
    addq  %rdx, %rax         # t2
    leaq  (%rsi,%rsi,2), %rdx
    salq  $4, %rdx           # t4
    leaq  4(%rdi,%rdx), %rcx # t5
    imulq %rcx, %rax         # rval
    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>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>t1, t2, rval</td>
</tr>
<tr>
<td>%rdx</td>
<td>t4</td>
</tr>
<tr>
<td>%rcx</td>
<td>t5</td>
</tr>
</tbody>
</table>
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