Machine-Level Programming I: Basics

Introduction to Computer Systems
5th Lecture, Sep. 15, 2015

Instructors:
Younghoon Kim

Original slides from Randal E. Bryant and David R. O’Hallaron are adopted and modified by Younghoon Kim
Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
Intel x86 Processors

- 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.
# 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></td>
<td></td>
<td></td>
<td></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></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></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></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>

- **8086**
  - First 16-bit Intel processor. Basis for IBM PC & DOS
  - 1MB address space

- **386**
  - First 32 bit Intel processor, referred to as IA32
  - Added “flat addressing”, capable of running Unix

- **Pentium 4E**
  - First 64-bit Intel x86 processor, referred to as x86-64

- **Core 2**
  - First multi-core Intel processor

- **Core i7**
  - Four cores
Intel x86 Processors, cont.

**Machine Evolution**

- 386 1985 0.3M
- Pentium 1993 3.1M
- Pentium/MMX 1997 4.5M
- Pentium Pro 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
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
x86 Clones: Advanced Micro Devices (AMD)

**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 Evolutionary 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
Today: 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 -O p1.c p2.c -o p`
  - Use basic optimizations (`-O`)
  - Put resulting binary in file `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 with command

```
gcc -O -S sum.c
```

Produces file `sum.s`

*Warning*: Will get very different results 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
  - Procedure calls and returns
Object Code

Code for sumstore

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

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

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

```
00000000000400595 <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 `a.out` (complete executable) or `.o` file
Alternate Disassembly

Disassembled

Object

Disassembled Dump of assembler code for function sumstore:

0x000000000000400595 <+0>: push %rbx
0x000000000000400596 <+1>: mov %rdx,%rbx
0x000000000000400599 <+4>: callq 0x400590 <plus>
0x00000000000040059e <+9>: mov %rax,(%rbx)
0x0000000000004005a1 <+12>: pop %rbx
0x0000000000004005a2 <+13>: retq

Within gdb Debugger

gdb sum
disable 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
Today: 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

| %rax | %eax |
| %rbx | %ebx |
| %rcx | %ecx |
| %rdx | %edx |
| %rsi | %esi |
| %rdi | %edi |
| %rsp | %esp |
| %rbp | %ebp |

| %r8  | %r8d |
| %r9  | %r9d |
| %r10 | %r10d |
| %r11 | %r11d |
| %r12 | %r12d |
| %r13 | %r13d |
| %r14 | %r14d |
| %r15 | %r15d |

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Type</th>
<th>Origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>%ax</td>
<td>accumulate</td>
</tr>
<tr>
<td>%ecx</td>
<td>%cx</td>
<td>counter</td>
</tr>
<tr>
<td>%edx</td>
<td>%dx</td>
<td>data</td>
</tr>
<tr>
<td>%ebx</td>
<td>%bx</td>
<td>base</td>
</tr>
<tr>
<td>%esi</td>
<td>%si</td>
<td>source</td>
</tr>
<tr>
<td>%edi</td>
<td>%di</td>
<td>index</td>
</tr>
<tr>
<td>%esp</td>
<td>%sp</td>
<td>destination</td>
</tr>
<tr>
<td>%ebp</td>
<td>%bp</td>
<td>base</td>
</tr>
</tbody>
</table>

16-bit virtual registers (backwards compatibility)
Moving Data

Moving Data

moving 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>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>Mem</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>
</tbody>
</table>

**Cannot do memory-memory transfer with a single instruction**
Simple Memory Addressing Modes

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

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

- **Displacement** \( D(R) \) \( \text{Mem}[\text{Reg}[R]+D] \)
  - 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;
}
```

```assembly
swap:
  movq (%rdi), %rax
  movq (%rsi), %rdx
  movq %rdx, (%rdi)
  movq %rax, (%rsi)
  ret
```
void swap 
  (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}

Memory

Registers

%rdi
%rsi
%rax
%rdx

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

<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>
Understanding `Swap()`

### Registers

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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></td>
</tr>
<tr>
<td><code>%rdx</code></td>
<td></td>
</tr>
</tbody>
</table>

### Memory

<table>
<thead>
<tr>
<th>Address (Hex)</th>
<th>Value (Dec)</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>

### Assembly Code

```
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>0x123</td>
</tr>
<tr>
<td>%rdx</td>
<td></td>
</tr>
</tbody>
</table>

Memory

<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:

```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>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>

## Code Snippet

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

---

The code snippet above demonstrates how to swap two memory locations using the `movq` instruction. The registers `%rdi` and `%rsi` are used to access the memory locations, and `%rax` and `%rdx` are used to store temporary values. The `movq` instructions are used to store the temporary values back into the original memory locations.
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>123</td>
</tr>
<tr>
<td>%rdx</td>
<td>456</td>
</tr>
</tbody>
</table>

Memory

<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>

Address

456

swap:

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

\begin{align*}
\text{Registers} & \\
\%rdi & 0x120 \\
\%rsi & 0x100 \\
\%rax & 123 \\
\%rdx & 456 \\
\end{align*}

\begin{align*}
\text{Memory} & \\
& 456 \\
& 0x120 \\
& 0x118 \\
& 0x110 \\
& 0x108 \\
& 0x100 \\
\end{align*}

\textbf{swap:}

\begin{align*}
\text{movq} & \quad (%rdi), \quad %rax \quad \# \quad t0 = *xp \\
\text{movq} & \quad (%rsi), \quad %rdx \quad \# \quad t1 = *yp \\
\text{movq} & \quad %rdx, \quad (%rdi) \quad \# \quad *xp = t1 \\
\text{movq} & \quad %rax, \quad (%rsi) \quad \# \quad *yp = t0 \\
\text{ret} & \\
\end{align*}
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
  ```
Complete Memory Addressing Modes

**Most General Form**

\[ D(Rb,Ri,S) \rightarrow Mem[Reg[Rb]+S*Reg[Ri]+D]\]

- **D**: Constant “displacement” 1, 2, or 4 bytes
- **Rb**: Base register: Any of 16 integer registers
- **Ri**: Index register: Any, except for \( \%r_{sp} \)
- **S**: Scale: 1, 2, 4, or 8 (*why these numbers?*)

**Special Cases**

- \((Rb, Ri)\rightarrow Mem[Reg[Rb]+Reg[Ri]]\)
- \(D(Rb, Ri)\rightarrow Mem[Reg[Rb]+Reg[Ri]+D]\)
- \((Rb, Ri, S)\rightarrow Mem[Reg[Rb]+S*Reg[Ri]]\)
# 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*0xf000 + 0x80</td>
<td>0x1e080</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%rdx</th>
<th>0xf000</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rcx</td>
<td>0x0100</td>
</tr>
</tbody>
</table>
Today: 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*
  - *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:**

```asm
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;&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 = ~Dest |
Arithmetic Expression Example

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