Machine-Level Programming I: Basics

15-213/18-213: Introduction to Computer Systems
5th Lecture, Sep. 15, 2015

Instructors:
Randal E. Bryant and David R. O’Hallaron
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>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>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>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Intel x86 Processors, cont.

- **Machine Evolution**
  - 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
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
Our Coverage

- **IA32**
  - The traditional x86
  - For 15/18-213: RIP, Summer 2015

- **x86-64**
  - The standard
  - `shark> gcc hello.c`
  - `shark> gcc -m64 hello.c`

- **Presentation**
  - Book covers x86-64
  - Web aside on IA32
  - We will only cover x86-64
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 -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)

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

  Object program (p1.o p2.o)
  Linker (gcc or ld)

  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 (on shark machine) with command

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

Produces file `sum.s`

**Warning:** Will get very different results on non-Shark 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
  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 \( \text{dest} \)

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

- **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

<table>
<thead>
<tr>
<th>Object</th>
<th>Disassembled</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0400595:</td>
<td>0x00000000000400595 &lt;+0&gt;: push %rbx</td>
</tr>
<tr>
<td>0x53</td>
<td>0x00000000000400596 &lt;+1&gt;: mov %rdx,%rbx</td>
</tr>
<tr>
<td>0x48</td>
<td>0x00000000000400599 &lt;+4&gt;: callq 0x400590 &lt;plus&gt;</td>
</tr>
<tr>
<td>0x89</td>
<td>0x000000000004005ae &lt;+9&gt;: mov %rax,(%rbx)</td>
</tr>
<tr>
<td>0xd3</td>
<td>0x000000000004005a1 &lt;+12&gt;: pop %rbx</td>
</tr>
<tr>
<td>0xe8</td>
<td>0x000000000004005a2 &lt;+13&gt;: retq</td>
</tr>
<tr>
<td>0xf2</td>
<td></td>
</tr>
<tr>
<td>0xff</td>
<td></td>
</tr>
<tr>
<td>0xff</td>
<td></td>
</tr>
<tr>
<td>0xff</td>
<td></td>
</tr>
<tr>
<td>0x48</td>
<td></td>
</tr>
<tr>
<td>0x89</td>
<td></td>
</tr>
<tr>
<td>0x03</td>
<td></td>
</tr>
<tr>
<td>0x5b</td>
<td></td>
</tr>
<tr>
<td>0xc3</td>
<td></td>
</tr>
</tbody>
</table>

- 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
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

<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: IA32 Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax, %ax, %ah, %al</td>
<td>accumulate</td>
</tr>
<tr>
<td>%ecx, %cx, %ch, %cl</td>
<td>counter</td>
</tr>
<tr>
<td>%edx, %dx, %dh, %dl</td>
<td>data</td>
</tr>
<tr>
<td>%ebx, %bx, %bh, %bl</td>
<td>base</td>
</tr>
<tr>
<td>%esi, %si,</td>
<td>source index</td>
</tr>
<tr>
<td>%edi, %di,</td>
<td>destination index</td>
</tr>
<tr>
<td>%esp, %sp,</td>
<td>stack pointer</td>
</tr>
<tr>
<td>%ebp, %bp,</td>
<td>base pointer</td>
</tr>
</tbody>
</table>

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

- **Moving Data**
  
  `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><code>Imm</code></td>
<td><code>Reg</code></td>
<td><code>movq $0x4,%rax</code></td>
<td><code>temp = 0x4;</code></td>
</tr>
<tr>
<td></td>
<td><code>Mem</code></td>
<td><code>movq $-147,(%rax)</code></td>
<td><code>*p = -147;</code></td>
</tr>
<tr>
<td><code>Reg</code></td>
<td><code>Reg</code></td>
<td><code>movq %rax,%rdx</code></td>
<td><code>temp2 = temp1;</code></td>
</tr>
<tr>
<td></td>
<td><code>Mem</code></td>
<td><code>movq %rax,(%rdx)</code></td>
<td><code>*p = temp;</code></td>
</tr>
<tr>
<td><code>Mem</code></td>
<td><code>Reg</code></td>
<td><code>movq (%rax),%rdx</code></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

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

■ Displacement  
  D(R)  
  Mem[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:
    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;
}
```

Memory

<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>

Registers

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>%rdi</th>
<th>0x120</th>
</tr>
</thead>
<tbody>
<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>123 0x120</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x118</td>
</tr>
<tr>
<td></td>
<td>0x110</td>
</tr>
<tr>
<td></td>
<td>0x108</td>
</tr>
<tr>
<td></td>
<td>0x100</td>
</tr>
</tbody>
</table>

Address

0x120

0x118

0x110

0x108

0x100

swap:

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

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

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>
</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
```
Simple Memory Addressing Modes

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

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

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

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

- **Most General Form**

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

  - **D:** Constant “displacement” 1, 2, or 4 bytes
  - **Rb:** Base register: Any of 16 integer registers
  - **Ri:** Index register: Any, except for %esp
  - **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>
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>
</table>
| addq    | Src, Dest            | Dest = Dest + Src  
| subq    | Src, Dest            | Dest = Dest - Src  
| imulq   | Src, Dest            | Dest = Dest * Src  
| salq    | Src, Dest            | Dest = Dest << Src  
| sarq    | Src, Dest            | Dest = Dest >> Src  
| shrq    | Src, Dest            | Dest = Dest >> Src  
| xorq    | Src, Dest            | Dest = Dest ^ Src  
| andq    | Src, Dest            | Dest = Dest & Src  
| orq     | Src, Dest            | Dest = Dest | Src  

- **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

- **See book for more instructions**
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

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