Machine-level Programs - Introduction

Han, Hwansoo
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>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>1978</td>
<td>29K</td>
<td>5-10</td>
<td>- First 16-bit Intel processor. Basis for IBM PC &amp; DOS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- 1MB address space</td>
</tr>
<tr>
<td>386</td>
<td>1985</td>
<td>275K</td>
<td>16-33</td>
<td>- First 32 bit Intel processor, referred to as IA32</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- Added “flat addressing”, capable of running Unix</td>
</tr>
<tr>
<td>Pentium 4E</td>
<td>2004</td>
<td>125M</td>
<td>2800-3800</td>
<td>- First 64-bit Intel x86 processor, referred to as x86-64</td>
</tr>
<tr>
<td>Core 2</td>
<td>2006</td>
<td>291M</td>
<td>1060-3500</td>
<td>- First multi-core Intel processor</td>
</tr>
<tr>
<td>Core i7</td>
<td>2008</td>
<td>731M</td>
<td>1700-3900</td>
<td>- Four cores</td>
</tr>
</tbody>
</table>
Intel x86 Processors (Cont’d)

❖ 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
Intel’s 64-bit History

- **2001**: Radical shift from IA32 to IA64
  - Totally different architecture (Itanium)
  - Executes IA32 code only in emulation mode

- **2003**: AMD introduces x86-64
  - Natural extension from IA32 (now called AMD64)

- **2004**: Intel announces EM64T extension to IA32
  - Extended memory 64-bit technology
  - Almost identical to x86-64 (also called Intel 64)

- **All processors except low-end x86 support x86-64**
  - But, lots of code still runs in 32-bit mode
Definitions

❖ **Architecture:** *(also ISA: instruction set architecture)*
  - A processor design that one needs to understand for writing 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 Programmer’s 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 operation
    - Used for conditional branching

- Memory
  - Byte addressable array
  - Includes stack used to support procedures
**Turning C into Object code**

- **gcc -Og p1.c p2.c -o p**
  - Use basic optimizations (-Og) [new to recent version of GCC]
  - Put resulting binary in file `p`

![Diagram of the process](image)

- **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

- `gcc -Og -S sum.c`

**sum.c**

```c
long plus(long x, long y);
void sumstore(long x, long y, long *dest)
{
    long t = plus(x, y);
    *dest = t;
}
```

**sum.s**

```assembly
sumstore:
    pushq %rbx
    movq %rdx, %rbx
    call plus
    movq %rax, (%rbx)
    popq %rbx
    ret
```

* Assembly code is obtained by using GCC version 5.4.0 on Ubuntu 16.04.4
* Different machines will result in a little bit different assembly code, due to different versions of GCC and compiler options
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

- **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

- **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 *.o files
  - Library linking
    - e.g., code for malloc(), printf(), etc.
  - Static linking
    - Combines with static runtime libraries
  - Dynamic linking
    - Linking occurs when program begins execution

```plaintext
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
```
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 \( \%rbx \)
  - \( \ast \text{dest} \): 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

**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>0xd3</td>
</tr>
<tr>
<td>0xe8</td>
</tr>
<tr>
<td>0xf2</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>

**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
disable sumstore
  - Disassemble procedure
x/14xb sumstore
  - Examine the 14 bytes starting at sumstore
  ```
### 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>
<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: IA32 Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Use</th>
<th>Origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>general purpose</td>
<td>accumulate</td>
</tr>
<tr>
<td>%ecx</td>
<td>( %ax )</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 index</td>
</tr>
<tr>
<td>%edi</td>
<td>( %di )</td>
<td>destination 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)

- %ah
- %cl
- %dh
- %dl
- %bh
- %bl
- %al
- %ch
Moving Data

Moving Data
\text{movq \ Source, Dest:}

Operand Types

- **Immediate:** Constant integer data
  - Example: \$0\times400, \$-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>Reg</strong></td>
<td><strong>Reg</strong></td>
<td>movq $0x4,%rax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td><strong>Reg</strong></td>
<td><strong>Mem</strong></td>
<td>movq $-147,(%rax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td><strong>Mem</strong></td>
<td><strong>Reg</strong></td>
<td>movq %rax,%rdx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td><strong>Mem</strong></td>
<td><strong>Reg</strong></td>
<td>movq (%rax),%rdx</td>
<td>*p = temp;</td>
</tr>
</tbody>
</table>

*Cannot perform memory-memory transfer with a single instruction*
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

\[
\text{movq } 8(\%rbp),%rdx
\]
void swap
   (long *xp, long *yp)
{
   long t0 = *xp;
   long t1 = *yp;
   *xp = t1;
   *yp = t0;
}

Example of Simple Addressing Modes

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

Memroy

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>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></td>
</tr>
<tr>
<td><code>%rdx</code></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>

#### Instructions

```assembly
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>456</td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td></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><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>

**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><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>
</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>

The swap function:

```asm
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>

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

❖ Most General Form

\[
D(R_b,R_i,S) \quad \text{Mem}[\text{Reg}[R_b]+S\times\text{Reg}[R_i]+D]
\]

▪ **D:** Constant “displacement” 1, 2, or 4 bytes
▪ **R_b:** Base register: Any of 16 integer registers
▪ **R_i:** Index register: Any, except for \%\text{rsp}
▪ **S:** Scale: 1, 2, 4, or 8 (*why these numbers?*)

❖ Special Cases

\[
\begin{align*}
(R_b,R_i) & \quad \text{Mem}[\text{Reg}[R_b]+\text{Reg}[R_i]] \\
D(R_b,R_i) & \quad \text{Mem}[\text{Reg}[R_b]+\text{Reg}[R_i]+D] \\
(R_b,R_i,S) & \quad \text{Mem}[\text{Reg}[R_b]+S\times\text{Reg}[R_i]]
\end{align*}
\]

* Useful to access arrays and structures
### Address Computation Example

<table>
<thead>
<tr>
<th>Expression</th>
<th>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 address without doing memory reference
    - e.g., translation of `p = &x[i];`
  - Computing arithmetic expressions of the form `x + k*y`
    - `k = 1, 2, 4, or 8`

```
long m12(long x)
{
    return x*12;
}
```

Assembly code by “gcc –S”
```
leaq (%rdi,%rdi,2), %rax  # t ← x+x*2
salq $2, %rax             # return t<<2
```
Some Arithmetic Operations

Two Operand Instructions

- **addq** Src, Dest  \( \text{Dest} = \text{Dest} + \text{Src} \)
- **subq** Src, Dest  \( \text{Dest} = \text{Dest} - \text{Src} \)
- **imulq** Src, Dest  \( \text{Dest} = \text{Dest} \times \text{Src} \)
- **salq** Src, Dest  \( \text{Dest} = \text{Dest} \ll \text{Src} \) (= **shlq**)
- **sarq** Src, Dest  \( \text{Dest} = \text{Dest} \gg \text{Src} \) (Arithmetic)
- **shrq** Src, Dest  \( \text{Dest} = \text{Dest} \gg\gg \text{Src} \) (Logical)
- **xorq** Src, Dest  \( \text{Dest} = \text{Dest} \oplus \text{Src} \)
- **andq** Src, Dest  \( \text{Dest} = \text{Dest} \& \text{Src} \)
- **orq** Src, Dest  \( \text{Dest} = \text{Dest} | \text{Src} \)

* Note the order of arguments
* No distinction between signed and unsigned int (why?)
Some Arithmetic Operations

❖ One Operand Instructions
   - incq Dest = Dest + 1
   - decq Dest = Dest - 1
   - negq Dest = - Dest (negative)
   - notq Dest = ~Dest (complement)

❖ 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;
}
```

### Interesting Instructions

- **leaq**: address computation
- **salq**: shift
- **imulq**: multiplication
  - But, only used once
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;
}

Understanding Arithmetic Expression

```
long 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