Machine-Level Representation: Basic

CSCI 2021: Machine Architecture and Organization

Pen-Chung Yew

Department Computer Science and Engineering

University of Minnesota

http://www.cs.umn.edu/~yew

With Slides from Randy Bryant, David O’Hallaron, and Antonia Zhai
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>• First 16-bit Intel processor. Basis for IBM PC &amp; DOS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• 1MB address space</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>• First 32 bit Intel processor, referred to as IA32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Added “flat addressing”, capable of running Unix</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>• First 64-bit Intel x86 processor, referred to as x86-64</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>• First multi-core Intel processor</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>• Four cores</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Our Coverage

• IA32
  • The traditional 32-bit 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**: (instruction set architecture - ISA)
  - 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 (binaries, or object code) 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` in 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`

```
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 ld)
```

```
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 x86-64 assembly with command
  - `gcc -Og -S sum.c`
- Produces file `sum.s`
- **Warning**: Will get very different results on different machines (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 (mov instructions)
  - Load data from memory into register
  - Store register data into memory
- Transfer control
  - Unconditional jumps to/from procedures (call instruction)
  - Conditional branches (jmp instructions)
Announcement 2/8/2016

• Data Lab due 11:55pm today 2/8/2019

• Homework Assignment #1 due 2/10 Wednesday before class
Review: Assembly/Machine Code View

Programmer-Visible (Accessible) Machine State

- **PC**: Program counter
  - Address of **next instruction**
  - Called `%rip` in x86-64

- **Register file**
  - Heavily used program data

- **Condition codes**
  - Store **status information** after execution of **current arithmetic** or **logical instr.**
  - Used for **conditional branching**

- **Memory**
  - Byte addressable array
  - Code and user data
  - Stack to support procedures
Review: Compiling 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` `(-o p)`

```
Review:
  Compiling
  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 (-o p)
```
Code for `sumstore`

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

### Assembler
- Translates `.s` into `.o`
- Binary encoding of each instruction
- Nearly-complete image of executable code
- Missing linkages between code in different files

### Linker (to be covered in Chapt 7)
- 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

---

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x53</td>
<td>pushq %rbx</td>
</tr>
<tr>
<td>0x48</td>
<td>movq %rdx, %rbx</td>
</tr>
<tr>
<td>0x89</td>
<td>call plus</td>
</tr>
<tr>
<td>0xd3</td>
<td>movq %rax, (%rbx)</td>
</tr>
<tr>
<td>0xe8</td>
<td>popq %rbx</td>
</tr>
<tr>
<td>0xf2</td>
<td>ret</td>
</tr>
</tbody>
</table>

- **Total of 14 bytes**
- **Each instruction 1, 3, or 5 bytes**
- **Starts at address 0x0400595**
Machine Instruction Example

• C Code
  • Store value \( t \) to the memory location designated by \( \text{dest} \)

\[ \text{*dest} = \text{t;} \]

movq \%rax, (%rbx)

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

0x40059e: 48 89 03

• Object Code
  • 3-byte instruction
  • Stored at address \( 0x40059e \)
## Disassembling Object Code

Disassembled (from Object Code to Assembly)

```
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.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
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

16-bit virtual registers (backwards compatibility)

<table>
<thead>
<tr>
<th>General Purpose</th>
<th>Origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>accumulate</td>
</tr>
<tr>
<td>%ecx</td>
<td>counter</td>
</tr>
<tr>
<td>%edx</td>
<td>data</td>
</tr>
<tr>
<td>%ebx</td>
<td>base</td>
</tr>
<tr>
<td>%esi</td>
<td>Source index</td>
</tr>
<tr>
<td>%edi</td>
<td>Dest index</td>
</tr>
<tr>
<td>%esp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>%ebp</td>
<td>Base pointer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>16-bit Virtual Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>%al</td>
</tr>
<tr>
<td>%cl</td>
</tr>
<tr>
<td>%dl</td>
</tr>
<tr>
<td>%bl</td>
</tr>
</tbody>
</table>
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><strong>movq</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Imm</strong></td>
<td><strong>Reg</strong></td>
<td>movq $0x4,%rax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td></td>
<td><strong>Mem</strong></td>
<td>movq $-147,(%rax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td><strong>Reg</strong></td>
<td><strong>Reg</strong></td>
<td>movq %rax,%rdx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td></td>
<td><strong>Mem</strong></td>
<td>movq %rax,(%rdx)</td>
<td>*p = temp;</td>
</tr>
<tr>
<td><strong>Mem</strong></td>
<td><strong>Reg</strong></td>
<td>movq (%rax),%rdx</td>
<td>temp = *p;</td>
</tr>
</tbody>
</table>

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

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

swap:

```assembly
    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;
}

Registers

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>

swap:

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

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</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>0x110</td>
</tr>
<tr>
<td>%rdx</td>
<td>0x108</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>%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></td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>Address</th>
<th></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:

```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>

#### 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

| %rdi | 0x120 |
| %rsi | 0x100 |
| %rax | 123   |
| %rdx | 456   |

Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>456</th>
</tr>
</thead>
</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()

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:

1. movq (%rdi), %rax  # t0 = *xp
2. movq (%rsi), %rdx  # t1 = *yp
3. movq %rdx, (%rdi)  # *xp = t1
4. movq %rax, (%rsi)  # *yp = t0
5. ret
Announcement 2/10/2016

• Homework Assignment #1 due before lecture today

• Bomb Lab issued today, due Friday 2/26/2016
  • Recitation session Thursday will cover Bomb Lab
  • Will cover GDB – debugger needed in Bomb Lab
**Review: 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>movq</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<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 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

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

- Most general form for an operand

\[ D(Rb,Ri,S) \quad \text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]+ D] \]

- **D**: Constant “displacement” 1, 2, or 4 bytes
- **Rb**: Base register: Any of 16 integer registers
- **Ri**: Index register: Any, except for \%rsp
- **S**: Scale: 1, 2, 4, or 8 (for different data types)

- Special cases

\[
\begin{align*}
(Rb,Ri) & \quad \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]] \\
D(Rb,Ri) & \quad \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]+D] \\
(Rb,Ri,S) & \quad \text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]]
\end{align*}
\]
Review: Word-Oriented Memory Organization

- Addresses specify byte locations
  - Address of **first byte** in a **word**
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)
## 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 (lea)
  - 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**

```
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>addq</td>
<td>Dest = Dest + Src</td>
</tr>
<tr>
<td>subq</td>
<td>Dest = Dest - Src</td>
</tr>
<tr>
<td>imulq</td>
<td>Dest = Dest * Src</td>
</tr>
<tr>
<td>shlq</td>
<td>Dest = Dest &lt;&lt; Src</td>
</tr>
<tr>
<td>sarq</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>shrq</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>xorq</td>
<td>Dest = Dest ^ Src</td>
</tr>
<tr>
<td>andq</td>
<td>Dest = Dest &amp; Src</td>
</tr>
<tr>
<td>orq</td>
<td>Dest = Dest</td>
</tr>
</tbody>
</table>

- Watch out for argument order!
- No distinction between signed and unsigned int (why?)
Review: Two’s Complement Addition

Operands: $w$ bits

\[
\begin{array}{c}
\text{u}
\end{array}
\begin{array}{c}
\cdots
\end{array}
\begin{array}{c}
\cdots
\end{array}
\begin{array}{c}
\cdots
\end{array}
\begin{array}{c}
\cdots
\end{array}
\begin{array}{c}
\cdots
\end{array}
\]

True Sum: $w+1$ bits

\[
\begin{array}{c}
\text{u + v}
\end{array}
\begin{array}{c}
\text{w bits}
\end{array}
\]

Discard Carry: $w$ bits

\[
\begin{array}{c}
\text{TAdd}\_w(u, v)
\end{array}
\begin{array}{c}
\text{w bits}
\end{array}
\]

- TAdd and UAdd have Identical Bit-Level Behavior
  - Signed vs. unsigned addition in C:
    
    ```
    int s, t, u, v;
    s = (int) ((unsigned) u + (unsigned) v);
    t = u + v
    ```
  - Will give \( s == t \)
Some Arithmetic Operations

• One Operand Instructions

  \[\text{incq} \quad \text{Dest} \quad \text{Dest} = \text{Dest} + 1\]
  \[\text{decq} \quad \text{Dest} \quad \text{Dest} = \text{Dest} - 1\]
  \[\text{negq} \quad \text{Dest} \quad \text{Dest} = -\text{Dest}\]
  \[\text{notq} \quad \text{Dest} \quad \text{Dest} = \sim\text{Dest}\]

• 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;
}
```

Instructions Used

- **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;
}

Where is $t_3$ calculated?

<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>$t_1, t_2, rval$</td>
</tr>
<tr>
<td>%rdx</td>
<td>$t_4$</td>
</tr>
<tr>
<td>%rcx</td>
<td>$t_5$</td>
</tr>
</tbody>
</table>

arith:

- leaq (%rdi,%rsi), %rax  # $t_1$
- addq %rdx, %rax          # $t_2$
- leaq (%rsi,%rsi,2), %rdx
- salq $4, %rdx            # $t_4$
- leaq 4(%rdi,%rdx), %rcx  # $t_5$
- imulq %rcx, %rax         # rval
- ret
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
Move on to Program Control