

UNIVERSITY of WASHINGTON | L08: x86-64 Programming I | CSE351, Autumn 2017

x86-64 Programming I

CSE 351 Autumn 2017

Instructor:
Justin Hsia

Teaching Assistants:
Lucas Wotton
Michael Zhang
Parker DeWilde
Ryan Wong
Sam Gehman
Sam Wolfson
Savanna Yee
Vinny Palaniappan

UNIVERSITY of WASHINGTON | L08: x86-64 Programming I | CSE351, Autumn 2017

Administrivia

- ❖ Lab 1 due tonight at 11:59pm
 - You have *late days* available
- ❖ Homework 2 due next Friday (10/20)
- ❖ Lab 2 (x86-64) released on Tuesday (10/17)
 - Due on 10/27

2

UNIVERSITY of WASHINGTON | L08: x86-64 Programming I | CSE351, Autumn 2017

Review: Operand types

- ❖ **Immediate:** Constant integer data
 - Examples: `$0x400`, `$-533`
 - Like C literal, but prefixed with `'$'`
 - Encoded with 1, 2, 4, or 8 bytes depending on the instruction
- ❖ **Register:** 1 of 16 integer registers
 - Examples: `%rax`, `%r13`
 - But `%rsp` reserved for special use
 - Others have special uses for particular instructions
- ❖ **Memory:** Consecutive bytes of memory at a computed address
 - Simplest example: `(%rax)`
 - Various other "address modes"

<code>%rax</code>
<code>%rcx</code>
<code>%rdx</code>
<code>%rbx</code>
<code>%rsi</code>
<code>%rdi</code>
<code>%rsp</code>
<code>%rbp</code>
<code>%rN</code>

3

UNIVERSITY of WASHINGTON | L08: x86-64 Programming I | CSE351, Autumn 2017

Moving Data

- ❖ General form: `mov_ source, destination`
 - Missing letter (`_`) specifies size of operands
 - Note that due to backwards-compatible support for 8086 programs (16-bit machines!), "word" means 16 bits = 2 bytes in x86 instruction names
 - Lots of these in typical code

❖ <code>movb src, dst</code>	❖ <code>movl src, dst</code>
▪ Move 1-byte "byte"	▪ Move 4-byte "long word"
❖ <code>movw src, dst</code>	❖ <code>movq src, dst</code>
▪ Move 2-byte "word"	▪ Move 8-byte "quad word"

4

UNIVERSITY of WASHINGTON | L08: x86-64 Programming I | CSE351, Autumn 2017

movq Operand Combinations

	Source	Dest	Src, Dest	C Analog
movq	Imm	Reg	<code>movq \$0x4, %rax</code>	<code>var_a = 0x4;</code>
		Mem	<code>movq \$-147, (%rax)</code>	<code>*p_a = -147;</code>
	Reg	Reg	<code>movq %rax, %rdx</code>	<code>var_d = var_a;</code>
		Mem	<code>movq %rax, (%rdx)</code>	<code>*p_d = var_a;</code>
Mem	Reg	<code>movq (%rax), %rdx</code>	<code>var_d = *p_a;</code>	

- ❖ **Cannot do memory-memory transfer with a single instruction**
 - How would you do it?

5

UNIVERSITY of WASHINGTON | L08: x86-64 Programming I | CSE351, Autumn 2017

x86-64 Introduction

- ❖ Arithmetic operations
- ❖ Memory addressing modes
 - swap example
- ❖ Address computation instruction (`leaq`)

6

UNIVERSITY of WASHINGTON | LOB: x86-64 Programming I | CSE351, Autumn 2017

Some Arithmetic Operations

- Binary (two-operand) Instructions:
 - Maximum of one memory operand**
 - Beware argument order!
 - No distinction between signed and unsigned
 - Only arithmetic vs. logical shifts
 - How do you implement "r3 = r1 + r2"?

Format	Computation	
<code>addq src, dst</code>	$dst = dst + src$	(dst += src)
<code>subq src, dst</code>	$dst = dst - src$	
<code>imulq src, dst</code>	$dst = dst * src$	signed mult
<code>sarq src, dst</code>	$dst = dst \gg src$	Arithmetic
<code>shrq src, dst</code>	$dst = dst \gg src$	Logical
<code>shlq src, dst</code>	$dst = dst \ll src$	(same as <code>salq</code>)
<code>xorq src, dst</code>	$dst = dst \wedge src$	
<code>andq src, dst</code>	$dst = dst \& src$	
<code>orq src, dst</code>	$dst = dst src$	

↑ operand size specifier

7

UNIVERSITY of WASHINGTON | LOB: x86-64 Programming I | CSE351, Autumn 2017

Some Arithmetic Operations

- Unary (one-operand) Instructions:

Format	Computation	
<code>incq dst</code>	$dst = dst + 1$	increment
<code>decq dst</code>	$dst = dst - 1$	decrement
<code>negq dst</code>	$dst = -dst$	negate
<code>notq dst</code>	$dst = \sim dst$	bitwise complement
- See CSPP Section 3.5.5 for more instructions: `mulq`, `cqto`, `idivq`, `divq`

8

UNIVERSITY of WASHINGTON | LOB: x86-64 Programming I | CSE351, Autumn 2017

Arithmetic Example

```
long simple_arith(long x, long y)
{
    long t1 = x + y;
    long t2 = t1 * 3;
    return t2;
}
```

Register	Use(s)
<code>%rdi</code>	1 st argument (x)
<code>%rsi</code>	2 nd argument (y)
<code>%rax</code>	return value

```
simple_arith:
    addq    %rdi, %rsi
    imulq  $3, %rsi
    movq   %rsi, %rax
    ret
```

```
Y += x;
Y *= 3;
long r = y;
return r;
```

9

UNIVERSITY of WASHINGTON | LOB: x86-64 Programming I | CSE351, Autumn 2017

Example of Basic 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
```

10

UNIVERSITY of WASHINGTON | LOB: x86-64 Programming I | CSE351, Autumn 2017

Understanding swap()

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

Registers	Memory
<code>%rdi</code>	123
<code>%rsi</code>	456
<code>%rax</code>	
<code>%rdx</code>	

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

Register	Variable
<code>%rdi</code>	<code>xp</code>
<code>%rsi</code>	<code>yp</code>
<code>%rax</code>	<code>t0</code>
<code>%rdx</code>	<code>t1</code>

11

UNIVERSITY of WASHINGTON | LOB: x86-64 Programming I | CSE351, Autumn 2017

Understanding swap()

Registers	Memory	Word Address
<code>%rdi</code>	123	0x120
<code>%rsi</code>	456	0x118
<code>%rax</code>		0x110
<code>%rdx</code>		0x108
		456
		0x100

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

12

UNIVERSITY of WASHINGTON | L08: x86-64 Programming I | CSE351, Autumn 2017

Understanding swap()

Registers		Memory	Word Address
%rdi	0x120	123	0x120
%rsi	0x100		0x118
%rax	123		0x110
%rdx		456	0x108

```

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

13

UNIVERSITY of WASHINGTON | L08: x86-64 Programming I | CSE351, Autumn 2017

Understanding swap()

Registers		Memory	Word Address
%rdi	0x120	123	0x120
%rsi	0x100		0x118
%rax	123		0x110
%rdx	456	456	0x108

```

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

14

UNIVERSITY of WASHINGTON | L08: x86-64 Programming I | CSE351, Autumn 2017

Understanding swap()

Registers		Memory	Word Address
%rdi	0x120	456	0x120
%rsi	0x100		0x118
%rax	123		0x110
%rdx	456	456	0x108

```

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

15

UNIVERSITY of WASHINGTON | L08: x86-64 Programming I | CSE351, Autumn 2017

Understanding swap()

Registers		Memory	Word Address
%rdi	0x120	456	0x120
%rsi	0x100		0x118
%rax	123		0x110
%rdx	456	123	0x108

```

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

16

UNIVERSITY of WASHINGTON | L08: x86-64 Programming I | CSE351, Autumn 2017

Memory Addressing Modes: Basic

- ❖ **Indirect:** $(R) \quad \text{Mem}[\text{Reg}[R]]$
 - Data in register R specifies the memory address
 - Like pointer dereference in C
 - Example:** `movq (%rcx), %rax`
- ❖ **Displacement:** $D(R) \quad \text{Mem}[\text{Reg}[R]+D]$
 - Data in register R specifies the *start* of some memory region
 - Constant displacement D specifies the offset from that address
 - Example:** `movq 8(%rbp), %rdx`

17

UNIVERSITY of WASHINGTON | L08: x86-64 Programming I | CSE351, Autumn 2017

Complete Memory Addressing Modes

- ❖ **General:**
 - $D(Rb, Ri, S) \quad \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]*S+D]$
 - Rb: Base register (any register)
 - Ri: Index register (any register except %rsp)
 - S: Scale factor (1, 2, 4, 8) – *why these numbers?*
 - D: Constant displacement value (a.k.a. immediate)
- ❖ **Special cases** (see CSPP Figure 3.3 on p.181)
 - $D(Rb, Ri) \quad \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]+D] \quad (S=1)$
 - $(Rb, Ri, S) \quad \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]*S] \quad (D=0)$
 - $(Rb, Ri) \quad \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]] \quad (S=1, D=0)$
 - $(, Ri, S) \quad \text{Mem}[\text{Reg}[Ri]*S] \quad (Rb=0, D=0)$

18

UNIVERSITY of WASHINGTON | L08: x86-64 Programming I | CSE351, Autumn 2017

Address Computation Examples

<code>%rdx</code>	<code>0xf000</code>	$D(Rb, Ri, S) \rightarrow$ $Mem[Reg[Rb]+Reg[Ri]*S+D]$
<code>%rcx</code>	<code>0x0100</code>	

Expression	Address Computation	Address
<code>0x8(%rdx)</code>		
<code>(%rdx,%rcx)</code>		
<code>(%rdx,%rcx,4)</code>		
<code>0x80(%rdx,2)</code>		

19

UNIVERSITY of WASHINGTON | L08: x86-64 Programming I | CSE351, Autumn 2017

Address Computation Instruction

- ❖ `leaq src, dst`
 - “lea” stands for *load effective address*
 - `src` is address expression (any of the formats we’ve seen)
 - `dst` is a register
 - Sets `dst` to the *address* computed by the `src` expression (*does not go to memory! – it just does math*)
 - **Example:** `leaq(%rdx,%rcx,4), %rax`
- ❖ Uses:
 - Computing addresses without a memory reference
 - e.g. translation of `p = &x[i]`;
 - Computing arithmetic expressions of the form $x+k*i+d$
 - Though `k` can only be 1, 2, 4, or 8

20

UNIVERSITY of WASHINGTON | L08: x86-64 Programming I | CSE351, Autumn 2017

Example: lea vs. mov

Registers	Memory	Word Address
<code>%rax</code>	<code>0x400</code>	0x120
<code>%rbx</code>	<code>0xF</code>	0x118
<code>%rcx</code> <code>0x4</code>	<code>0x8</code>	0x110
<code>%rdx</code> <code>0x100</code>	<code>0x10</code>	0x108
<code>%rdi</code>	<code>0x1</code>	0x100
<code>%rsi</code>		

```

leaq(%rdx,%rcx,4), %rax
movq(%rdx,%rcx,4), %rbx
leaq(%rdx), %rdi
movq(%rdx), %rsi
  
```

21

UNIVERSITY of WASHINGTON | L08: x86-64 Programming I | CSE351, Autumn 2017

Arithmetic Example

Register	Use(s)
<code>%rdi</code>	1 st argument (x)
<code>%rsi</code>	2 nd argument (y)
<code>%rdx</code>	3 rd argument (z)

```

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
    leaq4(%rdi,%rdx), %rcx
    imulq%rcx, %rax
    ret
  
```

- ❖ Interesting Instructions
 - `leaq`: “address” computation
 - `salq`: shift
 - `imulq`: multiplication
 - Only used once!

22

UNIVERSITY of WASHINGTON | L08: x86-64 Programming I | CSE351, Autumn 2017

Arithmetic Example

Register	Use(s)
<code>%rdi</code>	x
<code>%rsi</code>	y
<code>%rdx</code>	z, t4
<code>%rax</code>	t1, t2, rval
<code>%rcx</code>	t5

```

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 # rax/t1 = x + y
    addq%rdx, %rax # rax/t2 = t1 + z
    leaq(%rsi,%rsi,2), %rdx # rdx = 3 * y
    salq$4, %rdx # rdx/t4 = (3*y) * 16
    leaq4(%rdi,%rdx), %rcx # rcx/t5 = x + t4 + 4
    imulq%rcx, %rax # rax/rval = t5 * t2
    ret
  
```

23

UNIVERSITY of WASHINGTON | L08: x86-64 Programming I | CSE351, Autumn 2017

Peer Instruction Question

- ❖ Which of the following x86-64 instructions correctly calculates `%rax=9*%rdi`?
 - Vote at <http://PollEv.com/justin>

A. `leaq(,%rdi,9), %rax`
 B. `movq(,%rdi,9), %rax`
 C. `leaq(%rdi,%rdi,8), %rax`
 D. `movq(%rdi,%rdi,8), %rax`
 E. We’re lost...

24

Summary

- ❖ **Memory Addressing Modes:** The addresses used for accessing memory in `mov` (and other) instructions can be computed in several different ways
 - *Base register, index register, scale factor, and displacement* map well to pointer arithmetic operations
- ❖ `lea` is address calculation instruction
 - Does NOT actually go to memory
 - Used to compute addresses or some arithmetic expressions