

x86-64 Programming II

CSE 351 Winter 2020

Instructor:

Ruth Anderson

Teaching Assistants:

Jonathan Chen

Justin Johnson

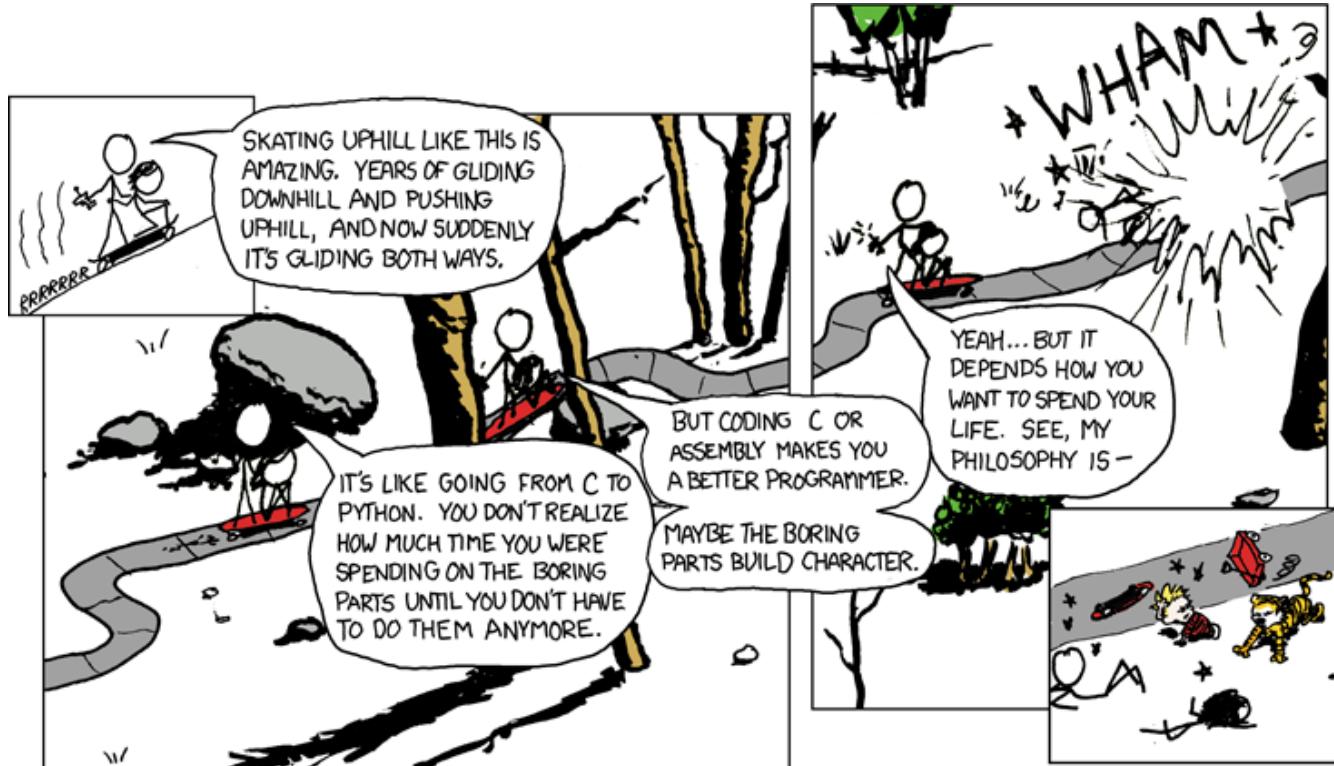
Porter Jones

Josie Lee

Jeffery Tian

Callum Walker

Eddy (Tianyi) Zhou



<http://xkcd.com/409/>

Administrivia

- ❖ Lab 1a due tonight!
 - Pay attention to Gradescope's feedback!
- ❖ Lab 2 (x86-64) coming soon
 - Learn to read x86-64 assembly and use GDB
- ❖ Submissions that fail the autograder get a **ZERO**
 - No excuses – make full use of tools & Gradescope's interface
- ❖ Midterm is in two weeks (2/10 during lecture)
 - You will be provided a fresh reference sheet
 - Study and use this NOW so you are comfortable with it when the exam comes around
 - Form study groups and look at past exams!

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

Example: lea vs. mov

Registers

%rax	
%rbx	
%rcx	0x4
%rdx	0x100
%rdi	
%rsi	

Memory

0x400
0xF
0x8
0x10
0x1

Word Address

0x120
0x118
0x110
0x108
0x100

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

lea – “It just does math”

Arithmetic 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
```

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

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

Arithmetic 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;
}
```

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

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
leaq	4(%rdi,%rdx), %rcx	# rcx/t5	= x + t4 + 4
imulq	%rcx, %rax	# rax/rval	= t5 * t2
ret			

Polling Question

- ❖ Which of the following x86-64 instructions correctly calculates $\%rax = 9 * \%rdi$?
 - Vote at <http://pollev.com/reassembly>
- 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...**

Control Flow

```
long max(long x, long y)
{
    long max;
    if (x > y) {
        max = x;
    } else {
        max = y;
    }
    return max;
}
```

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

```
max:
???
movq    %rdi, %rax
???
???
movq    %rsi, %rax
???
ret
```

Control Flow

```
long max(long x, long y)
{
    long max;
    if (x > y) {
        max = x;
    } else {
        max = y;
    }
    return max;
}
```

Conditional jump

Unconditional jump

max:
if $x \leq y$ then jump to else
 movq %rdi, %rax
 jump to done
else:
 movq %rsi, %rax
done:
 ret

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

Conditionals and Control Flow

- ❖ Conditional branch/*jump*
 - Jump to somewhere else if some *condition* is true, otherwise execute next instruction
- ❖ Unconditional branch/*jump*
 - Always jump when you get to this instruction
- ❖ Together, they can implement most control flow constructs in high-level languages:
 - **if** (*condition*) **then** { ... } **else** { ... }
 - **while** (*condition*) { ... }
 - **do** { ... } **while** (*condition*)
 - **for** (*initialization*; *condition*; *iterative*) { ... }
 - **switch** { ... }

x86 Control Flow

- ❖ Condition codes
- ❖ Conditional and unconditional branches
- ❖ Loops
- ❖ Switches

Processor State (x86-64, partial)

- ❖ Information about currently executing program
 - Temporary data (`%rax, ...`)
 - Location of runtime stack (`%rsp`)
 - Location of current code control point (`%rip, ...`)
 - Status of recent tests (**CF, ZF, SF, OF**)
 - Single bit registers:

Registers

<code>%rax</code>	<code>%r8</code>
<code>%rbx</code>	<code>%r9</code>
<code>%rcx</code>	<code>%r10</code>
<code>%rdx</code>	<code>%r11</code>
<code>%rsi</code>	<code>%r12</code>
<code>%rdi</code>	<code>%r13</code>
<code>%rsp</code>	<code>%r14</code>
<code>%rbp</code>	<code>%r15</code>



current top of the Stack

<code>%rip</code>

Program Counter
(instruction pointer)

CF

ZF

SF

OF

Condition Codes

Condition Codes (Implicit Setting)

- ❖ *Implicitly set by arithmetic operations*
 - (think of it as side effects)
 - Example: **addq** src, dst \leftrightarrow r = d+s
 - **CF=1** if carry out from MSB (*unsigned overflow*)
 - **ZF=1** if $r==0$
 - **SF=1** if $r<0$ (if MSB is 1)
 - **OF=1** if *signed overflow*
 $(s>0 \ \&\& \ d>0 \ \&\& \ r<0) \ | \ | \ (s<0 \ \&\& \ d<0 \ \&\& \ r>=0)$
 - **Not set by lea instruction (beware!)**



Condition Codes (Explicit Setting: Compare)

- ❖ *Explicitly* set by **Compare** instruction
 - **cmpq** `src1, src2`
 - **cmpq** `a, b` sets flags based on $b-a$, but doesn't store

- **CF=1** if carry out from MSB (good for *unsigned* comparison)
- **ZF=1** if $a==b$
- **SF=1** if $(b-a) < 0$ (if MSB is 1)
- **OF=1** if *signed* overflow

$$(a>0 \ \&\& \ b<0 \ \&\& \ (b-a) >0) \quad || \\ (a<0 \ \&\& \ b>0 \ \&\& \ (b-a) <0)$$


Condition Codes (Explicit Setting: Test)

- ❖ *Explicitly* set by **Test** instruction
 - **testq** src2, src1
 - **testq** a, b sets flags based on a&b, but doesn't store
 - Useful to have one of the operands be a *mask*
 - Can't have carry out (**CF**) or overflow (**OF**)
 - **ZF=1** if a&b==0
 - **SF=1** if a&b<0 (signed)



Using Condition Codes: Jumping

- ❖ j^* Instructions
 - Jumps to *target* (an address) based on condition codes

Instruction	Condition	Description
<code>jmp target</code>	1	Unconditional
<code>je target</code>	ZF	Equal / Zero
<code>jne target</code>	$\sim ZF$	Not Equal / Not Zero
<code>js target</code>	SF	Negative
<code>jns target</code>	$\sim SF$	Nonnegative
<code>jg target</code>	$\sim (SF \wedge OF) \ \& \ \sim ZF$	Greater (Signed)
<code>jge target</code>	$\sim (SF \wedge OF)$	Greater or Equal (Signed)
<code>jl target</code>	$(SF \wedge OF)$	Less (Signed)
<code>jle target</code>	$(SF \wedge OF) \ \ ZF$	Less or Equal (Signed)
<code>ja target</code>	$\sim CF \ \& \ \sim ZF$	Above (unsigned " $>$ ")
<code>jb target</code>	CF	Below (unsigned " $<$ ")

Using Condition Codes: Setting

- ❖ **set*** Instructions
 - Set low-order byte of *dst* to 0 or 1 based on condition codes
 - Does not alter remaining 7 bytes

Instruction	Condition	Description
sete <i>dst</i>	ZF	Equal / Zero
setne <i>dst</i>	$\sim ZF$	Not Equal / Not Zero
sets <i>dst</i>	SF	Negative
setns <i>dst</i>	$\sim SF$	Nonnegative
setg <i>dst</i>	$\sim (SF \wedge OF) \ \& \ \sim ZF$	Greater (Signed)
setge <i>dst</i>	$\sim (SF \wedge OF)$	Greater or Equal (Signed)
setl <i>dst</i>	$(SF \wedge OF)$	Less (Signed)
setle <i>dst</i>	$(SF \wedge OF) \mid ZF$	Less or Equal (Signed)
seta <i>dst</i>	$\sim CF \ \& \ \sim ZF$	Above (unsigned ">")
setb <i>dst</i>	CF	Below (unsigned "<")

Reminder: x86-64 Integer Registers

- ❖ Accessing the low-order byte:

%rax	%al	%r8	%r8b
%rbx	%bl	%r9	%r9b
%rcx	%cl	%r10	%r10b
%rdx	%dl	%r11	%r11b
%rsi	%sil	%r12	%r12b
%rdi	%dil	%r13	%r13b
%rsp	%spl	%r14	%r14b
%rbp	%bpl	%r15	%r15b

Reading Condition Codes

❖ set* Instructions

- Set a low-order byte to 0 or 1 based on condition codes
- Operand is byte register (e.g. al, dl) or a byte in memory
- Do not alter remaining bytes in register
 - Typically use movzbl (zero-extended mov) to finish job

```
int gt(long x, long y)
{
    return x > y;
}
```

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

```
cmpq    %rsi, %rdi    #
setg    %al             #
movzbl  %al, %eax     #
ret
```

Reading Condition Codes

❖ set* Instructions

- Set a low-order byte to 0 or 1 based on condition codes
- Operand is byte register (e.g. al, dl) or a byte in memory
- Do not alter remaining bytes in register
 - Typically use movzbl (zero-extended mov) to finish job

```
int gt(long x, long y)
{
    return x > y;
}
```

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

```
cmpq    %rsi, %rdi    # Compare x:y
setg    %al             # Set when >
movzbl  %al, %eax     # Zero rest of %rax
ret
```

Aside: **movz** and **movs**

movz SD *src, regDest* *# Move with zero extension*

movs SD *src, regDest* *# Move with sign extension*

- Copy from a *smaller* source value to a *larger* destination
- Source can be memory or register; Destination *must* be a register
- Fill remaining bits of dest with **zero** (**movz**) or **sign bit** (**movs**)

movzSD / **movsSD**:

S – size of source (**b** = 1 byte, **w** = 2)

D – size of dest (**w** = 2 bytes, **l** = 4, **q** = 8)

Example:

movzbq %al, %rbx

0x??	0xFF	←%rax						
0x00	0xFF	←%rbx						

Aside: `movz` and `movs`

`movz __ src, regDest` # Move with zero extension

`movs __ src, regDest` # Move with sign extension

- Copy from a *smaller* source value to a *larger* destination
- Source can be memory or register; Destination *must* be a register
- Fill remaining bits of dest with **zero** (`movz`) or **sign bit** (`movs`)

`movzSD` / `movsSD`:

S – size of source (**b** = 1 byte, **w** = 2)

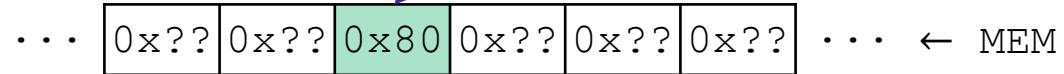
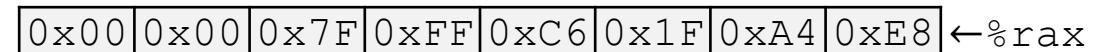
D – size of dest (**w** = 2 bytes, **l** = 4, **q** = 8)

Note: In x86-64, any instruction that generates a 32-bit (long word) value for a register also sets the high-order portion of the register to 0. Good example on p. 184 in the textbook.

Example:

`movsb1 (%rax), %ebx`

Copy 1 byte from memory into
8-byte register & sign extend it



Summary

- ❖ Control flow in x86 determined by status of Condition Codes
 - Showed **Carry**, **Zero**, **Sign**, and **Overflow**, though others exist
 - Set flags with arithmetic instructions (implicit) or Compare and Test (explicit)
 - Set instructions read out flag values
 - Jump instructions use flag values to determine next instruction to execute