

x86-64 Programming II

CSE 351 Winter 2020

Instructor:

Ruth Anderson

Teaching Assistants:

Jonathan Chen

Justin Johnson

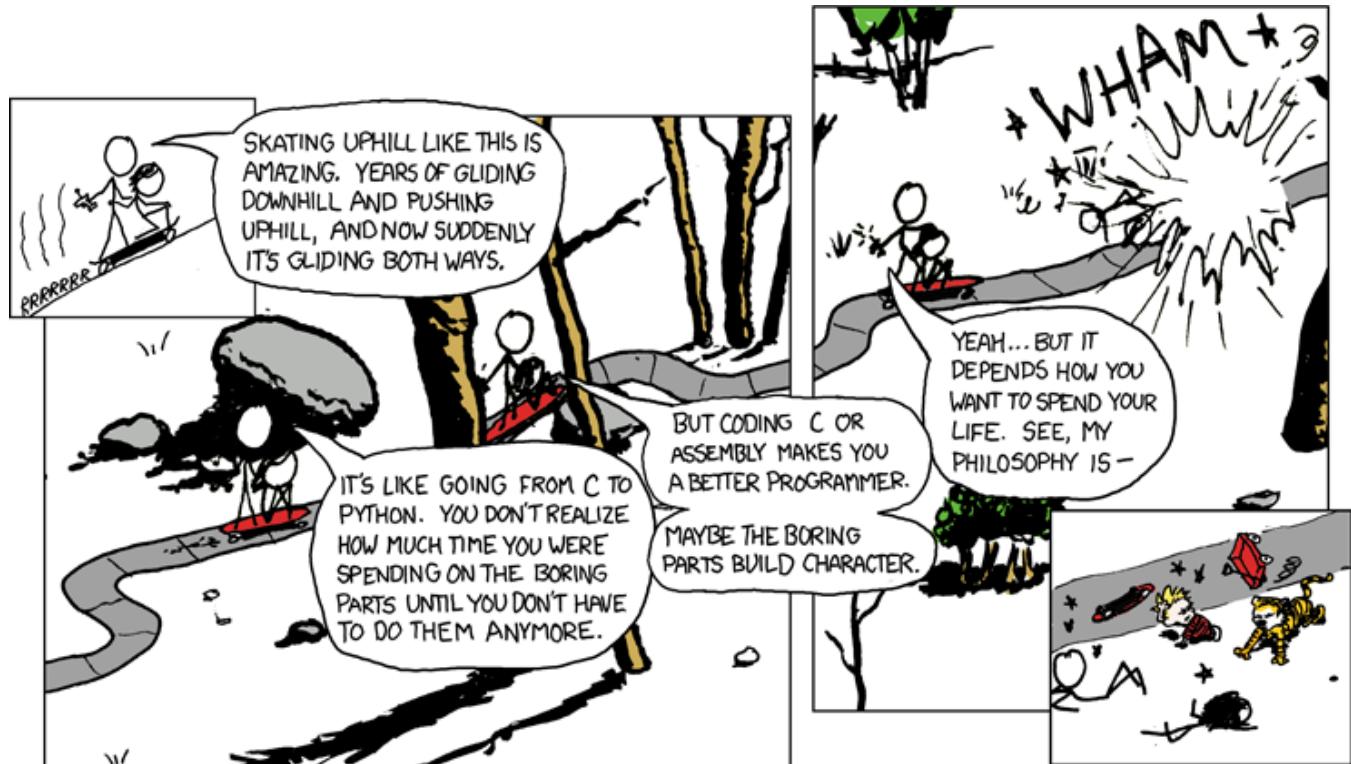
Porter Jones

Josie Lee

Jeffery Tian

Callum Walker

Eddy (Tianyi) Zhou



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

Administrivia

- ❖ Lab 1~~E~~ 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	0x10
%rbx	0x8
%rcx	0x4
%rdx	0x100
%rdi	0x100
%rsi	0x1

Memory

0x400	0x120
0xF	0x118
0x8	0x110
0x10	0x108
0x1	0x100

Word Address

0x100 0x10

```
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;
}
```

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

```
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
 - 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` $s \in \{1, 2, 4, 8\}$

~~B.~~ `movq (,%rdi,9), %rax`

C. `leaq (%rdi,%rdi,8), %rax`

D. `movq (%rdi,%rdi,8), %rax`

E. We're lost...

$$\%rax = 9 * \%rdi$$

$$\%rax = *(9 * \%rdi)$$

Control Flow

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

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

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

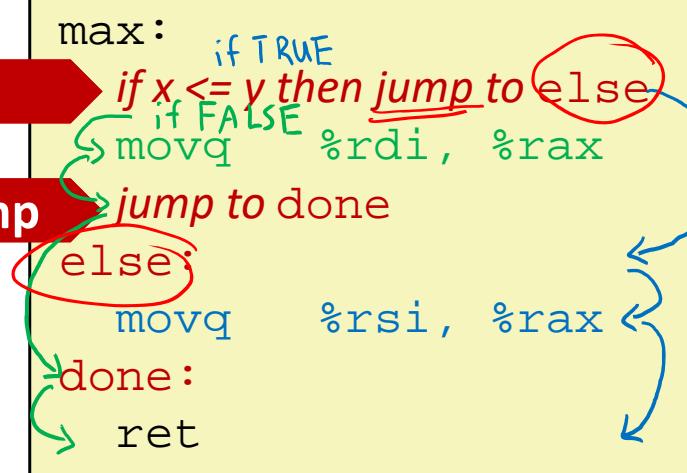
Control Flow

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

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

Conditional jump

Unconditional jump



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) "flags"
 - 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)

Carry Zero Sign Overflow

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` $\leftrightarrow r = d + s$

Example

- $CF=1$ 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

$$(s>0 \quad \&\& \quad d>0 \quad \&\& \quad r<0) \quad | \quad | \quad (s<0 \quad \&\& \quad d<0 \quad \&\& \quad r>=0)$$

- Not set by `lea` instruction (beware!)

$$\begin{array}{r} \text{%eax} = 100\ldots0 \\ + 10\ldots0 \\ \hline \text{result} = 100\ldots0 \end{array}$$

$$\%eax + \%eax$$

$$\text{result} = dst + src$$

$$\text{addl \%eax, \%eax}$$

example if %eax holds 0x80 00 00 00:

$$\text{addl \%eax, \%eax}$$

0x0 stored in %eax

CF = 1

ZF = 1

SF = 0

OF = 1 ($0+0=0$)

↑ signs don't match!



Condition Codes (Explicit Setting: Compare)

- ❖ *Explicitly* set by **Compare** instruction

- **cmpq** src1, src2 like subq a, b → $b - a$
- **cmpq** a, b sets flags based on $b - a$, but doesn't store ^{the} result

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

$$\begin{aligned} & (a > 0 \quad \&\& \quad b < 0 \quad \&\& \quad (b - a) > 0) \quad || \\ & (a < 0 \quad \&\& \quad b > 0 \quad \&\& \quad (b - a) < 0) \end{aligned}$$


Carry Flag



Zero Flag



Sign Flag

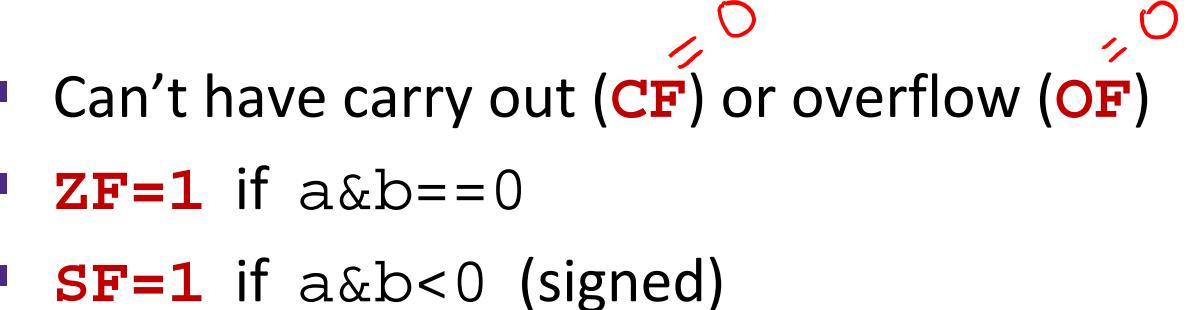


Overflow Flag

Condition Codes (Explicit Setting: Test)

- ❖ *Explicitly* set by **Test** instruction

- **testq** src2, src1 like *andq a, b*
- **testq** a, b sets flags based on a&b, but doesn't store ^{the} result
 - 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

don't worry about the details

Instruction	Condition	Description (always compared to 0)
<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>	$\sim (SF \wedge OF) \wedge \sim ZF$	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>	$(SF \wedge OF) \mid ZF$	Less or Equal (Signed)
<code>ja target</code>	$\sim CF \wedge \sim ZF$	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

Same instruction
suffixes as
 j^* instructions!

False $\rightarrow 0b\ 0000\ 000 = 0x\ 00$
True $\rightarrow 0b\ 0000\ 001 = 0x\ 01$

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

Annotations:

- A red circle highlights the "%al" box in the first row.
- A red arrow labeled "↑_{8B}" points to the "%rsp" row.
- A red arrow labeled "↑_{1B}" points to the "%al" box in the first row.
- A red curved arrow points from the "↑_{1B}" label to the "%r15" row.

Reading Condition Codes

e, ne, g, l, ...

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

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

```
int gt(long x, long y)
{
    return x > y; // x - y > 0
}
```

```
cmpq %rsi, %rdi          # set flags based on x-y
setg %al                  # %al = (x>y)
movzbl %al, %eax          # %rax = (x>y)
ret
```

zero-extend → *a(y), b(x)*

lowest byte ←

whole register ←

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

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

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

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

Set cmA codes based on:
 $x-y$ $x-y > 0$
 $x > y$

Aside: movz and movs

2 width specifiers: b, w, l, q
1 2 4 8 bytes

`movz__ src, regDest` # Move with zero extension
`movs__ src, regDest` # Move with sign extension

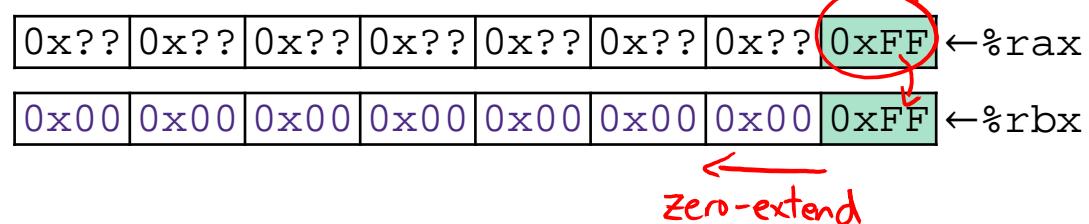
- Copy from a *smaller* source value to a *larger* destination
- Source can be memory or register; Destination *must* be a register
- Fill remaining bits of dest with **zero** (`movz`) or **sign bit** (`movs`)

`movzSD` / `movsSD`:

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

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

Example:
`movzbq %al, %rbx`
Zero-extend ↑ 1 byte 8 bytes



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

0x00	0x00	0x7F	0xFF	0xC6	0x1F	0xA4	0xE8	←%rax
------	------	------	------	------	------	------	------	-------

...	0x??	0x??	0x80	0x??	0x??	0x??	...	← MEM
-----	------	------	------	------	------	------	-----	-------

0x00	0x00	0x00	0x00	0xFF	0xFF	0xFF	0x80	←%rbx
------	------	------	------	------	------	------	------	-------

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