

x86-64 Programming II

CSE 351 Winter 2019

Instructors:

Max Willsey

Luis Ceze

Teaching Assistants:

Britt Henderson

Lukas Joswiak

Josie Lee

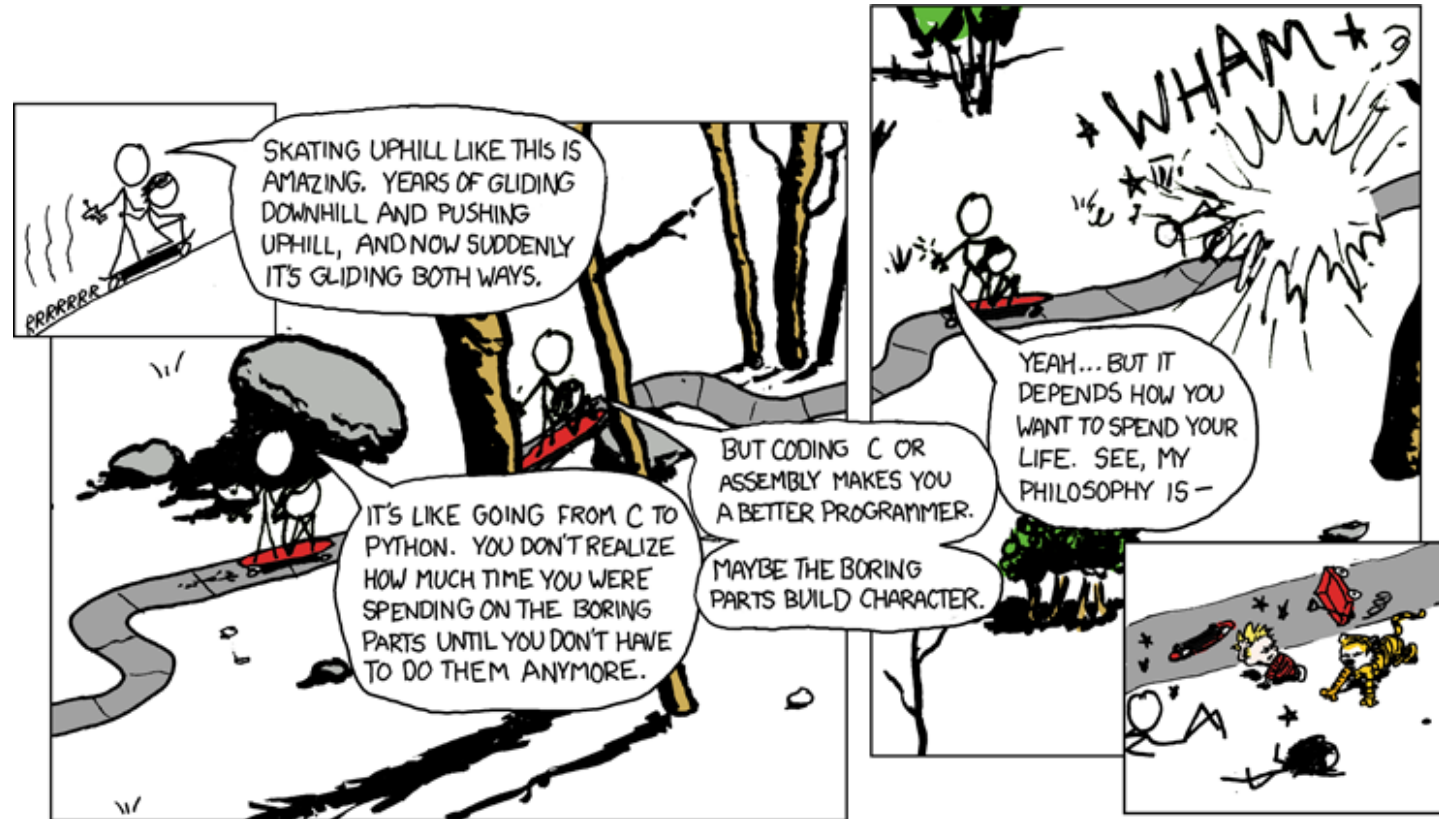
Wei Lin

Daniel Snitkovsky

Luis Vega

Kory Watson

Ivy Yu



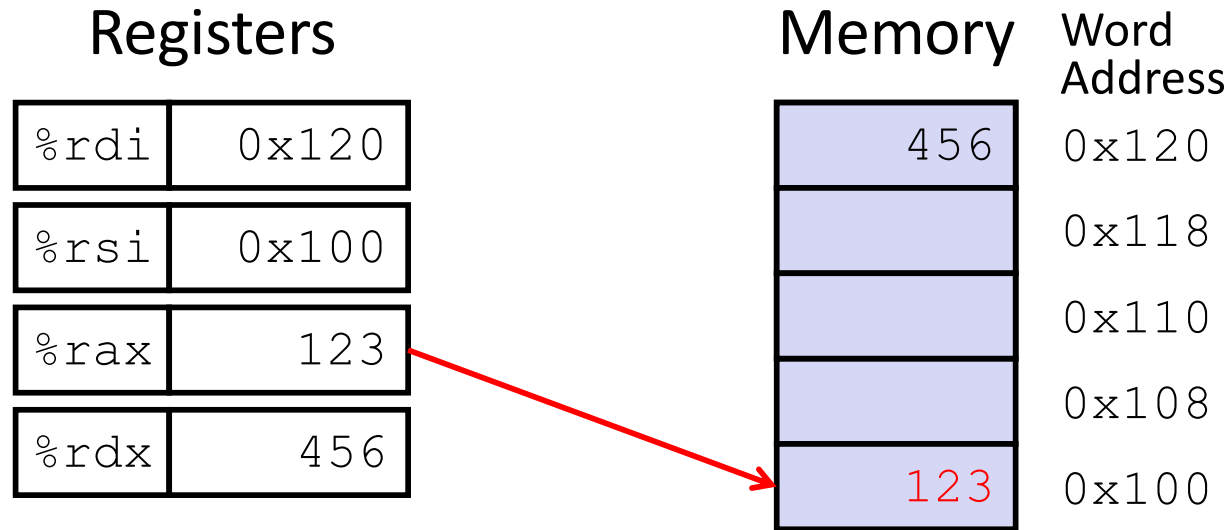
<http://xkcd.com/409/>

Administrivia

- ❖ Lab 2 (x86-64) due Feb 8
 - Learn to read x86-64 assembly and use GDB
- ❖ Homework 2 due Feb 1 (this Friday)

- ❖ Midterm is in ~two weeks (Feb 13)!
 - No lecture that day
 - You will be provided a fresh reference sheet
 - Study and use this NOW so you are comfortable with it when the exam comes around
 - You get 1 *handwritten*, double-sided cheat sheet (letter)
 - Find a study group! Look at past exams!

Understanding swap ()



```

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

Memory Addressing Modes: Basic

❖ **Indirect:** (R) $\text{Mem}[\text{Reg}[R]]$

- Data in register R specifies the memory address
- Like pointer dereference in C
- Example: `movq (%rcx), %rax`

❖ **Displacement:** $D (R)$ $\text{Mem}[\text{Reg}[R]+D]$

- 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`

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)

- $D(Rb, Ri) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] + D] \quad (S=1)$
- $(Rb, Ri, S) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] * S] \quad (D=0)$
- $(Rb, Ri) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri]] \quad (S=1, D=0)$
- $(, Ri, S) \quad \text{Mem}[\text{Reg}[Ri] * S] \quad (Rb=0, D=0)$

Address Computation Examples

<code>%rdx</code>	<code>0xf000</code>
<code>%rcx</code>	<code>0x0100</code>

$D(Rb, Ri, S) \rightarrow$
 $Mem[Reg[Rb]+Reg[Ri]*S+D]$

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

(if not specified)

default values:

$S = 1$

$D = 0$

$Reg[Rb] = 0$

$Reg[Ri] = 0$

Address Computation Examples

%rdx	0xf000
%rcx	0x0100

$D(Rb, Ri, S) \rightarrow$

$Mem[Reg[Rb] + Reg[Ri] * S + D]$

↑ ignore the memory access for now

Expression	Address Computation	Address (8 bytes wide)
$0x8$ (^D <u>%rdx</u>)	$Reg[Rb] + D = 0xf000 + 0x8$	$0xf008$
(^{Rb} %rdx, ^{Ri} %rcx)	$Reg[Rb] + Reg[Ri] * 1$	$0xf100$
(^{Rb} %rdx, ^{Ri} %rcx, ^S 4)	$*4$	$0xf400$
$0x80$ (^D , ^{Ri} %rdx, ^S 2)	$Reg[Ri] * 2 + 0x80$	$0x1e080$

$0xf000 * 2$

$0xf000 \ll 1 = 0x1e000$

1111 0000
1 1110 0000...0

Summary

- ❖ There are 3 types of operands in x86-64
 - Immediate, Register, Memory
- ❖ There are 3 types of instructions in x86-64
 - Data transfer, Arithmetic, Control Flow
- ❖ **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

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: `leaq` vs. `movq`

Registers

<code>%rax</code>	
<code>%rbx</code>	
<code>%rcx</code>	<code>0x4</code>
<code>%rdx</code>	<code>0x100</code>
<code>%rdi</code>	
<code>%rsi</code>	

Memory

	Word Address
<code>0x400</code>	<code>0x120</code>
<code>0xF</code>	<code>0x118</code>
<code>0x8</code>	<code>0x110</code>
<code>0x10</code>	<code>0x108</code>
<code>0x1</code>	<code>0x100</code>

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

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

```

Peer Instruction Question

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

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

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

```
max:
    if x <= y then jump to else
    movq    %rdi, %rax
    jump to done
else:
    movq    %rsi, %rax
done:
    ret
```

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** {...}

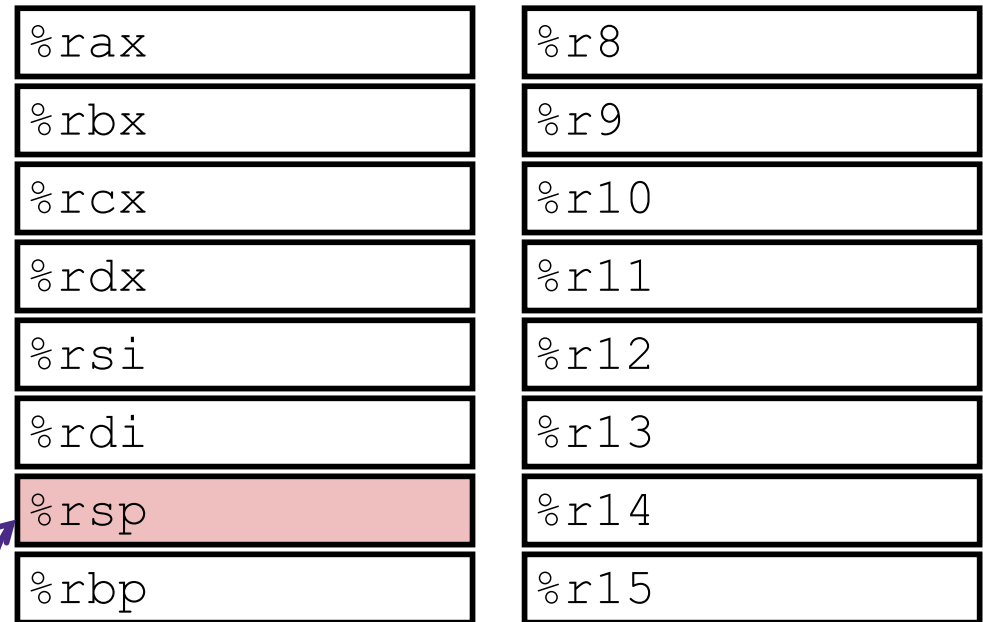
x86 Control Flow

- ❖ **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

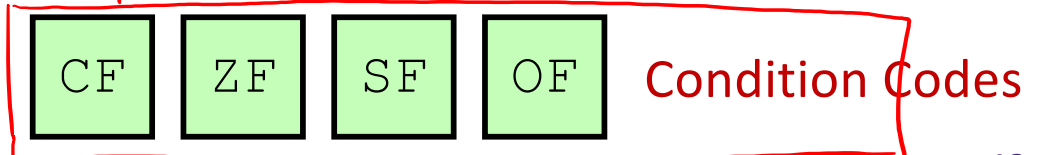


current top of the Stack



Program Counter
(instruction pointer)

Carry Zero Sign Overflow



Condition Codes (Implicit Setting)

❖ *Implicitly* set by arithmetic operations

- (think of it as side effects)

- Example: `addq src, dst` \leftrightarrow `r = d+s`
result = dst + src

- **CF=1** if carry out from MSB (*unsigned* overflow)

- **ZF=1** if `r==0`

example if %eax holds 0x 80 00 00 00:

- **SF=1** if `r<0` (if MSB is 1)

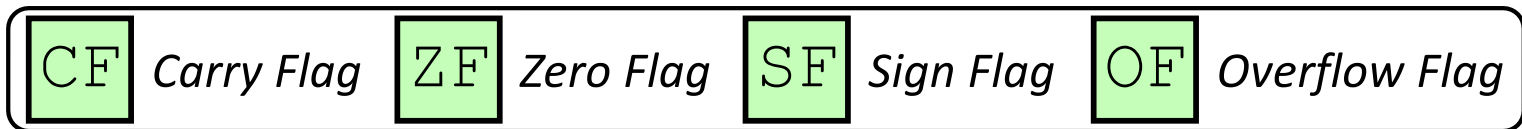
addl %eax,%eax # 0x0 stored in %eax
 # CF = 1
 # ZF = 1
 # SF = 0
 # OF = 1 (⊖+⊖=⊕)

- **OF=1** if *signed* overflow

`(s>0 && d>0 && r<0) || (s<0 && d<0 && r>=0)`

↑ signs don't match!

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)

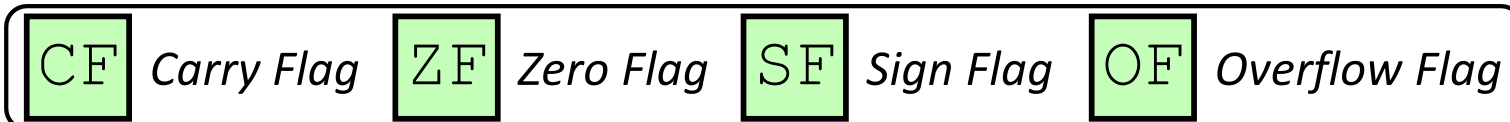
- **ZF=1** if $a==b$

- **SF=1** if $(b-a) < 0$ (if MSB is 1)

- **OF=1** if *signed overflow*

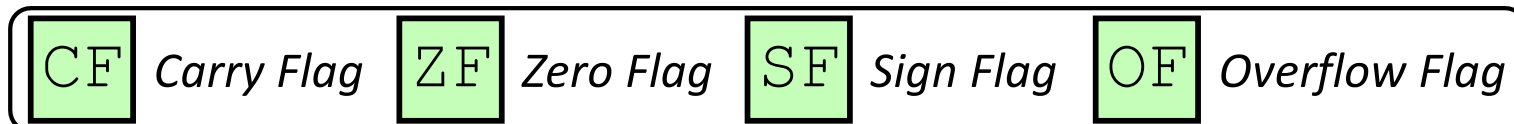
$(a > 0 \ \&\& \ b < 0 \ \&\& \ (b-a) > 0) \ ||$

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



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>jje 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>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

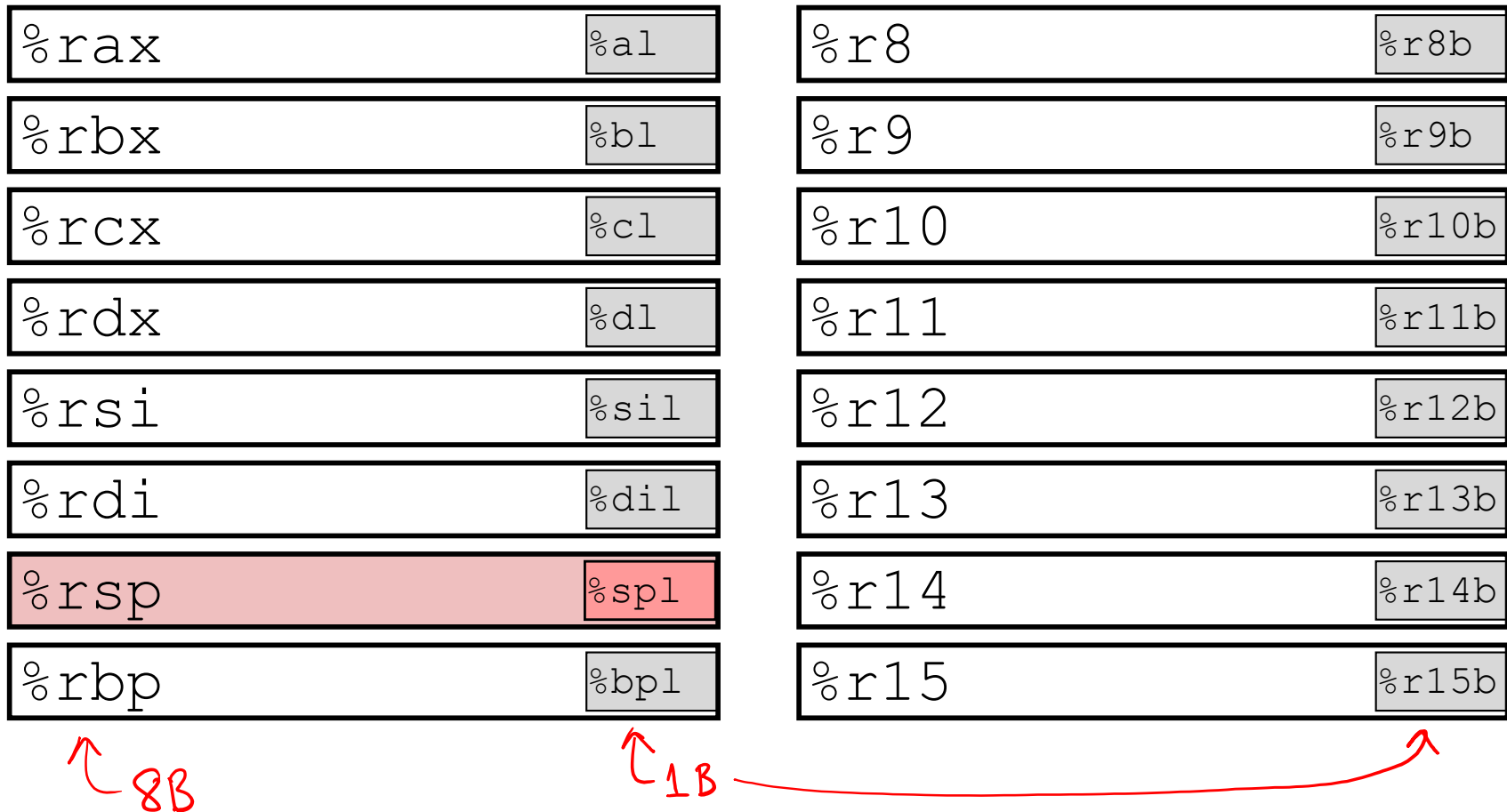
False → 0b 0000 0000 = 0x 00
 True → 0b 0000 0001 = 0x 01

Same instruction suffixes as j* instructions!

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

Reminder: x86-64 Integer Registers

❖ Accessing the low-order byte:



Reading Condition Codes

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

❖ 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; // x-y > 0
}
```

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

Handwritten annotations:
 - Red arrow from `x > y` to `setg`
 - Red arrow from `setg` to `movzbl`
 - Red arrow from `movzbl` to `ret`
 - Red arrow from `setg` to `%al` with label "lowest byte"
 - Red arrow from `movzbl` to `%eax` with label "whole register"
 - Red arrow from `movzbl` to `%eax` with label "zero-extend"
 - Red text above `cmpq`: `a(y), b(x)`
 - Red text next to `setg`: `# %al = (x > y)`
 - Red text next to `movzbl`: `# %rax = (x > y)`
 - Red text next to `ret`: `#`

Reading Condition Codes

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

❖ 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;
}
```

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

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)

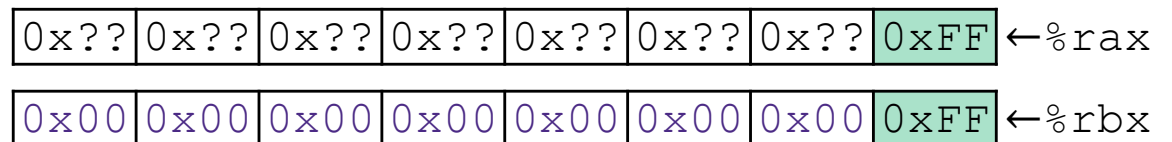
movzSD / movsSD:

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

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

Example:

movzbq %al, %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`)

`movz`SD / `movs`SD:

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

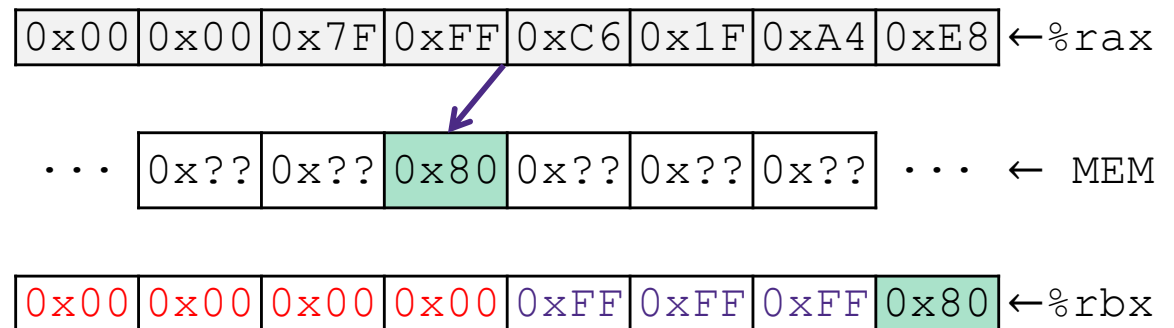
D – size of dest (**w** = 2 bytes, **l** = 4, **q** = 8)

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:

```
movsbl (%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 **C**arry, **Z**ero, **S**ign, and **O**verflow, 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