

# x86-64 Programming II

CSE 351 Autumn 2020

## Instructor:

Justin Hsia

## Teaching Assistants:

Aman Mohammed

Ami Oka

Callum Walker

Cosmo Wang

Hang Do

Jim Limprasert

Joy Dang

Julia Wang

Kaelin Laundry

Kyrie Dowling

Mariam Mayanja

Shawn Stanley

Yan Zhe Ong



<http://xkcd.com/99/>

# Administrivia

- ❖ Lab 2 (x86-64) released today
  - 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
  - Some leeway was given on Lab 1a, do not expect the same leniency moving forward
- ❖ Midterm is in two weeks (take home, 10/31–11/2)
  - Find groups of 5 for the group stage
  - Make notes and use the [midterm reference sheet](#)
  - Form study groups and look at past exams!

# Extra Credit

- ❖ All labs starting with Lab 2 have extra credit portions
  - These are meant to be fun extensions to the labs
- ❖ Extra credit points *don't* affect your lab grades
  - From the course policies: “they will be accumulated over the course and will be used to bump up borderline grades at the end of the quarter.”
  - Make sure you finish the rest of the lab before attempting any extra credit

# Complete Memory Addressing Modes

## ❖ General:

$$ar[i] \leftrightarrow *(ar + i) \rightarrow \text{Mem}[ar + i * \text{size of (data type)}]$$

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

- Rb: Base register (any register)
- Ri: Index register (any register except %rsp)
- S: Scale factor (1, 2, 4, 8) – *why these numbers?* data type widths
- D: Constant displacement value (a.k.a. immediate)

## ❖ Special cases (see CSPP Figure 3.3 on p.181)

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

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

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

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

↑ so reg name not interpreted as Rb

# Address Computation Examples

default values:

$S = 1$

$D = 0$

$Reg[Rb] = 0$

$Reg[Ri] = 0$

%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)
$0x8$ ( <sup>D</sup> <u>%rdx</u> )	$Reg[Rb] + D = 0xf000 + 0x8$	$0xf008$
( <sup>Rb</sup> %rdx, <sup>Ri</sup> %rcx)	$Reg[Rb] + Reg[Ri] * 1$	$0xf100$
( <sup>Rb</sup> %rdx, <sup>Ri</sup> %rcx, <sup>S</sup> 4)	$*4$	$0xf400$
$0x80$ ( <sup>D</sup> , <sup>Ri</sup> %rdx, <sup>S</sup> 2)	$Reg[Ri] * 2 + 0x80$	$0x1e080$

$0xf000 * 2$

$0xf000 \ll 1 = 0x1e000$

1111 0000  
1 1110 0000...0

# Reading Review

- ❖ Terminology:
  - Address Computation Instruction (`lea`)
  - Condition codes: Carry Flag (`CF`), Zero Flag (`ZF`), Sign Flag (`SF`), and Overflow Flag (`OF`)
  - Test (`test`) and compare (`cmp`) assembly instructions
  - Jump (`j*`) and set (`set*`) families of assembly instructions
  
- ❖ Questions from the Reading?

# Review Questions

❖ Which of the following x86-64 instructions correctly calculates  $\%rax = 9 * \%rdi$ ? *no memory access, so must be lea  $S \in \{1, 2, 4, 8\}$*

A. ~~leaq~~ (~~,~~ ~~%rdi~~, ~~9~~), %rax *invalid syntax*

B. ~~movq~~ (~~,~~ %rdi, 9), %rax *invalid syntax*

**C.** leaq (%rdi, %rdi, 8), %rax *%rax = 9 \* %rdi*

D. ~~movq~~ (%rdi, %rdi, 8), %rax *%rax = Mem[9 \* %rdi]*

❖ If %rsi is 0x B0BACAFE 1EE7 F0 0D, what is its value after executing `movswl %si, %esi`? *MSB of %si is a 1*

*sign extension* *destination is 4 bytes* *source is 2 bytes*

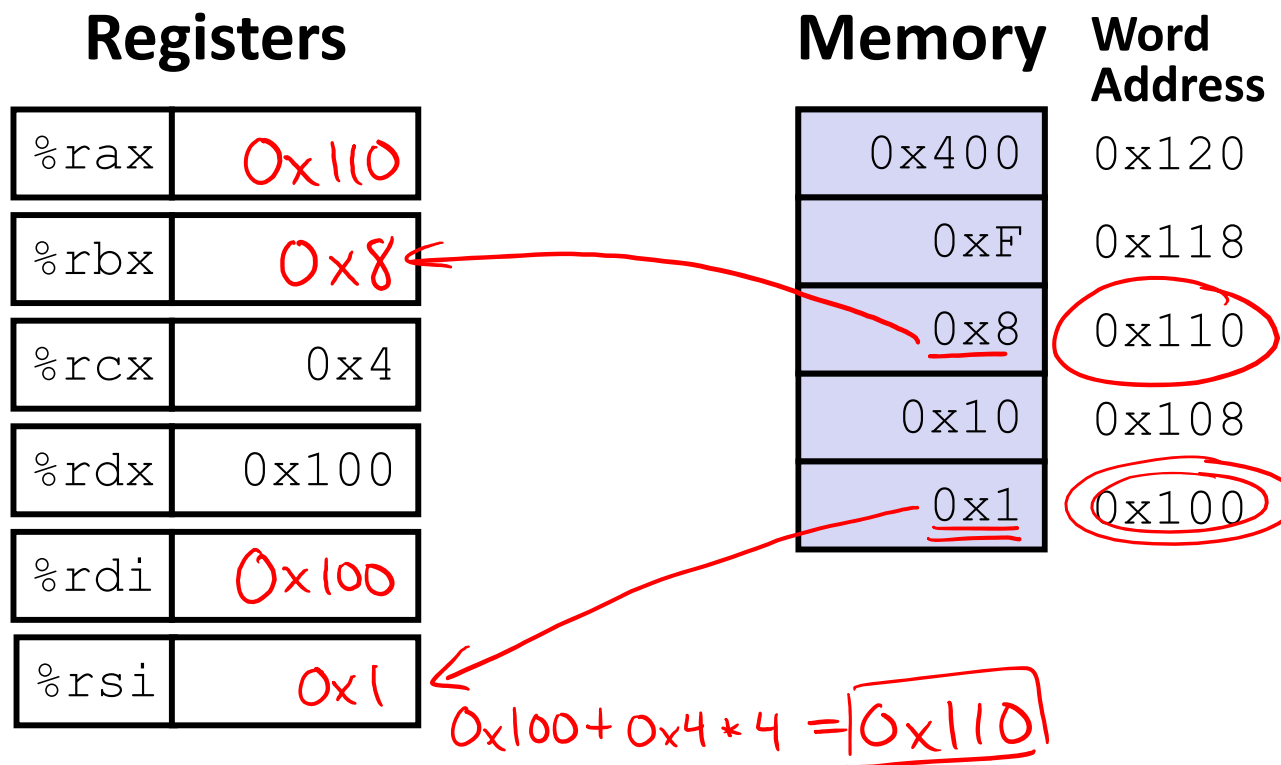
0x 0000 0000 FFFF F00D  
*x86-64 rule when destination is 32 bits* *↑ sign extension* *↑ original data*

# Address Computation Instruction

- ❖  $\overset{\text{"Mem" Reg}}{\text{leaq src, dst}}$ 
  - "lea" stands for *load effective address*
  - src is address expression (any of the formats we've seen)
  - dst is a register  $\hookrightarrow$  calculates  $\text{Reg}[R_b] + \text{Reg}[R_i] * S + D$
  - Sets dst to the *address* computed by the src expression  
(**does not go to memory!** – it just does math) ~~Mem!~~
  - Example: `leaq (%rdx, %rcx, 4), %rax`
- ❖ Uses:
  - Computing addresses without a memory reference
    - e.g., translation of `p = &x[i];`  $\leftarrow$  address-of operator
  - Computing arithmetic expressions of the form  $\underline{x + k * i + d}$   $\text{Reg}[R_b] + \text{Reg}[R_i] * S + D$ 
    - Though  $k$  can only be 1, 2, 4, or 8



# Example: lea vs. mov



leaq <sup>Rb</sup> (%rdx, <sup>Ri</sup> %rcx, <sup>S</sup> 4), %rax	→ 0x110	("addr")
movq (%rdx, %rcx, 4), %rbx	→ 0x8	(data)
leaq (%rdx), %rdi	→ 0x100	("addr")
movq (%rdx), %rsi	→ 0x1	(data)

0x100

# 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 <sup>st</sup> argument (x)
%rsi	2 <sup>nd</sup> argument (y)
%rdx	3 <sup>rd</sup> argument (z)

← replaced by lea & shift

```

arith:
    leaq    (%rdi,%rsi), %rax    # rax = x + y (t1)
    addq   %rdx, %rax          # rax = x + y + z (t2)
    leaq   (%rsi,%rsi,2), %rdx  # rdx = 3y
    salq   $4, %rdx           # rdx = 48y (t4)
    leaq   4(%rdi,%rdx), %rcx
    imulq  %rcx, %rax
    ret
    
```

← multiplying two variables

## 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

limited registers means they often get reused!

```

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
    
```

*comment (AT &T syntax)*

*SE{1,2,4,8}*

# Control Flow

Register	Use(s)
%rdi	1 <sup>st</sup> argument (x)
%rsi	2 <sup>nd</sup> 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 # if case
    ???
    ???
    movq    %rsi, %rax # else case
    ???
    ret
```

# Control Flow

Register	Use(s)
%rdi	1 <sup>st</sup> argument (x)
%rsi	2 <sup>nd</sup> 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 TRUE
    if x <= y then jump to else
    if FALSE
    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** {...}

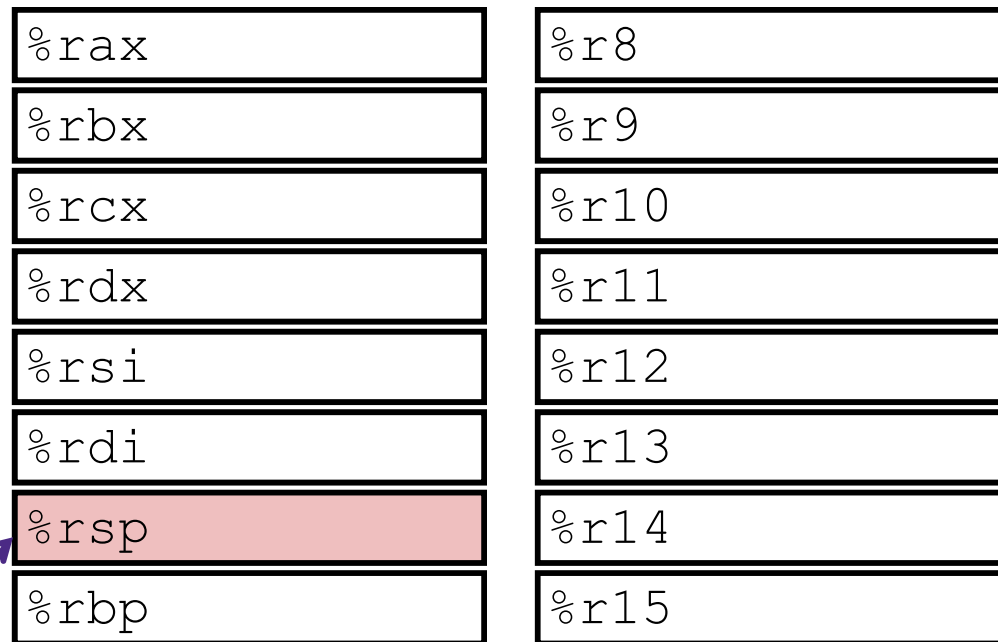
# x86 Control Flow

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

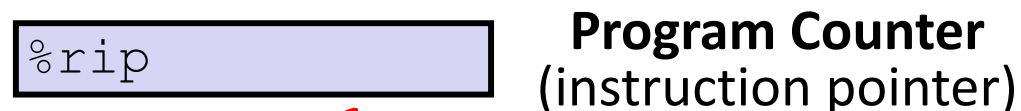
# Processor State (x86-64, partial)

- ❖ 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:

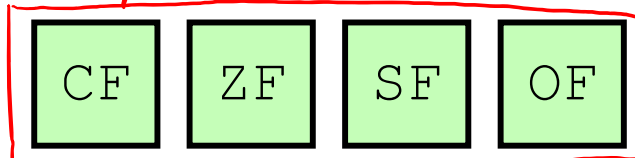
## Registers



current top of the Stack



Carry Zero Sign Overflow



**Condition Codes**



# Condition Codes (Implicit Setting)

## ❖ *Implicitly* set by **arithmetic** operations

- (think of it as side effects)

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

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

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

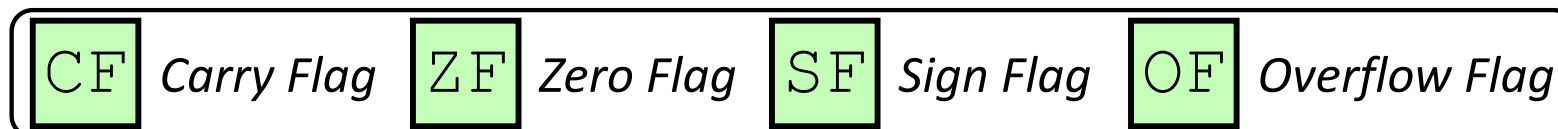
- **SF=1** if  $r < 0$  (if MSB is 1)

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

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

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

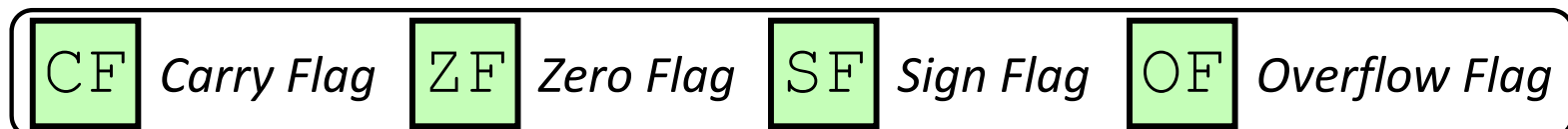
*↑ signs don't match!*



# Condition Codes (Explicit Setting: Compare)

## ❖ Explicitly set by **Compare** instruction

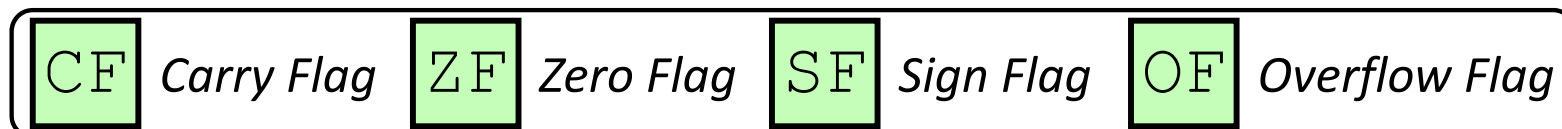
- **cmpq** src1, src2    *like subq a, b → b - a*
- **cmpq** a, b sets flags based on  $b - a$ , but doesn't store
- **CF=1** if carry out from MSB (good for *unsigned* comparison)
- **ZF=1** if  $a == b$  ( $b - a == 0$ )
- **SF=1** if  $(b - a) < 0$  (if MSB is 1)
- **OF=1** if *signed* overflow  
 $(a > 0 \ \&\& \ b < 0 \ \&\& \ (b - a) > 0) \ ||$   
 $(a < 0 \ \&\& \ b > 0 \ \&\& \ (b - a) < 0)$



# Condition Codes (Explicit Setting: Test)

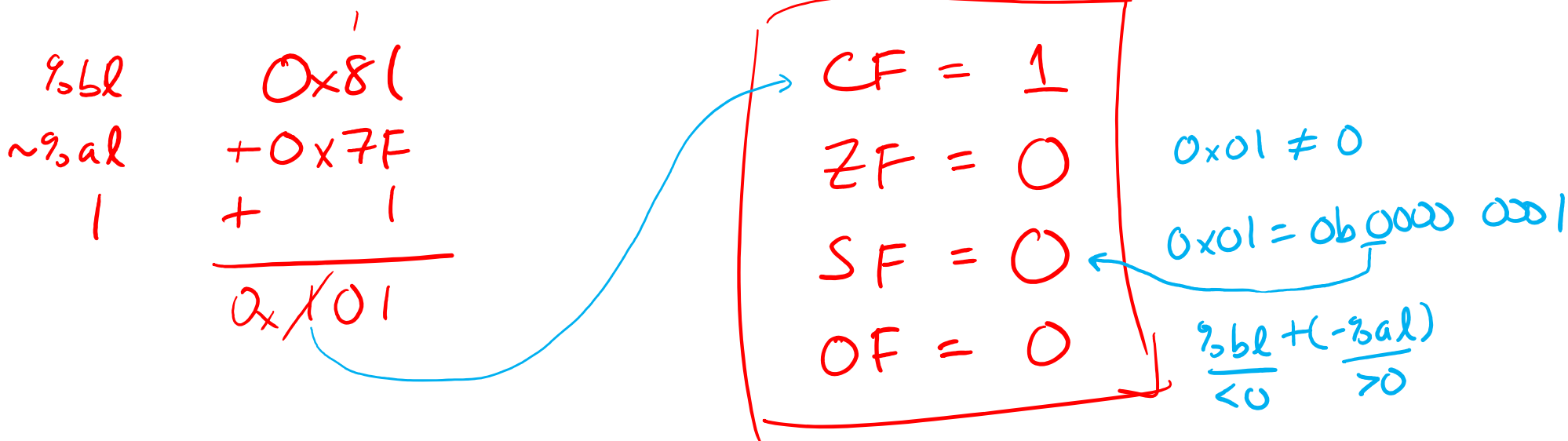
❖ *Explicitly* set by **Test** instruction

- **testq** src2, src1 *like andq a, b*
- **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**<sup>0</sup>) or overflow (**OF**<sup>0</sup>)
- **ZF=1** if a&b==0
- **SF=1** if a&b<0 (signed)



# Example Condition Code Setting

- Assuming that `%a1 = 0x80` and `%b1 = 0x81`, which flags (CF, ZF, SF, OF) are set when we execute `cmpb %a1, %b1`?  $\rightarrow$  computes  $\%b1 - \%a1 = \%b1 + \sim \%a1 + 1$   
 $\sim \%a1 = \sim 0x80 = 0x7F$



# Using Condition Codes: Jumping

❖ j\* Instructions

- Jumps to **target** (an address) based on condition codes

*don't worry about the details*

Instruction	Condition	Description <i>(always compared to 0)</i>
<u>jmp</u> target	1	Unconditional
<u>j<del>e</del></u> target	ZF	Equal / Zero
<u>j<del>n</del>e</u> target	~ZF	Not Equal / Not Zero
<u>j<del>s</del></u> target	SF	Negative
<u>j<del>n</del>s</u> target	~SF	Nonnegative
<u>j<del>g</del></u> target	$\sim (SF \wedge OF) \ \& \ \sim ZF$	Greater (Signed)
<u>j<del>g</del>e</u> target	$\sim (SF \wedge OF)$	Greater or Equal (Signed)
<u>j<del>l</del></u> target	$(SF \wedge OF)$	Less (Signed)
<u>j<del>l</del>e</u> target	$(SF \wedge OF) \   \ ZF$	Less or Equal (Signed)
<u>j<del>a</del></u> target	$\sim CF \ \& \ \sim ZF$	Above (unsigned ">")
<u>j<del>b</del></u> target	CF	Below (unsigned "<")

# Using Condition Codes: Setting

## ❖ set\* Instructions

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

- 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!

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

# Reading Condition Codes

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

## ❖ *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) 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    a(y), b(x) %rsi, %rdi # set flags based on x-y
setg    %al # %al = (x > y)
zero-extend → movzbl %al, %eax # %rax = (x > y)
ret
```

← lowest byte  
← whole register

# Reading Condition Codes

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

## ❖ `set*` Instructions

- Set a low-order byte to 0 or 1 based on condition codes
- Operand is byte register (e.g., `%al`) 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  
2 width specifiers: b, w, l, q  
1 2 4 8 bytes

`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`)

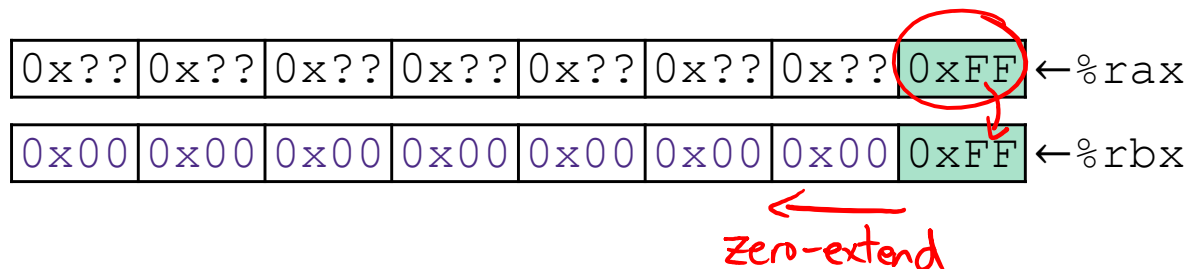
`movzSD` / `movsSD`:

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)

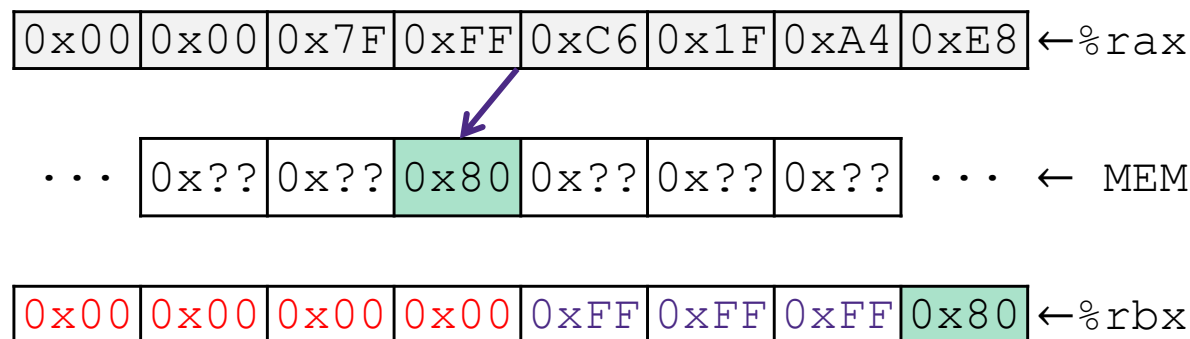
D – size of dest (**w** = 2 bytes, **l** = 4, **q** = 8)

**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:

`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