

# x86-64 Programming III

CSE 351 Autumn 2021

## Instructor:

Justin Hsia

## Teaching Assistants:

Allie Pflieger

Anirudh Kumar

Assaf Vayner

Atharva Deodhar

Celeste Zeng

Dominick Ta

Francesca Wang

Hamsa Shankar

Isabella Nguyen

Joy Dang

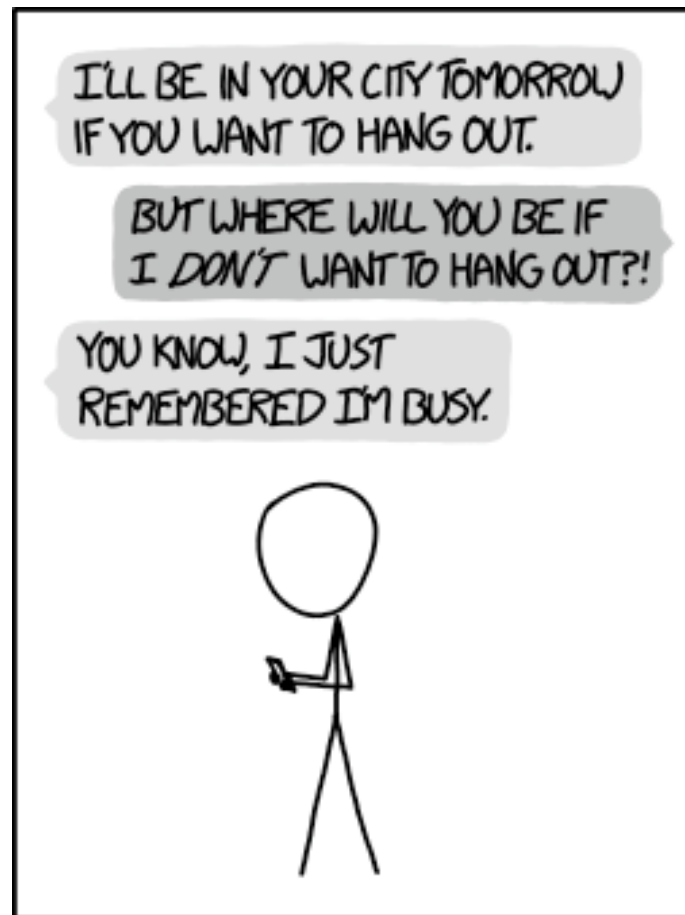
Julia Wang

Maggie Jiang

Monty Nitschke

Morel Fotsing

Sanjana Chintalapati



<http://xkcd.com/1652/>

# Relevant Course Information

- ❖ Lab 1a regrade requests open on Gradescope
- ❖ Lab 1b submissions close tonight
- ❖ Lab 2 due next Friday (10/29)
  
- ❖ Section tomorrow on Assembly
  - Use the midterm reference sheet!
  - Optional GDB Tutorial slides and Lab 2 phase 1 walkthrough
  
- ❖ Midterm (take home, 11/3–11/5)
  - Make notes and use the [midterm reference sheet](#)
  - Form study groups and look at past exams!

# Move extension: `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`

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

# Move extension: `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

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

# GDB Demo

- ❖ The `movz` and `movs` examples on a real machine!
  - `movzbq %al, %rbx`
  - `movsbl (%rax), %ebx`
- ❖ You will need to use GDB to get through Lab 2
  - Useful debugger in this class and beyond!
- ❖ Pay attention to:
  - Setting breakpoints (`break`)
  - Stepping through code (`step/next` and `stepi/nexti`)
  - Printing out expressions (`print` – works with regs & vars)
  - Examining memory (`x`)

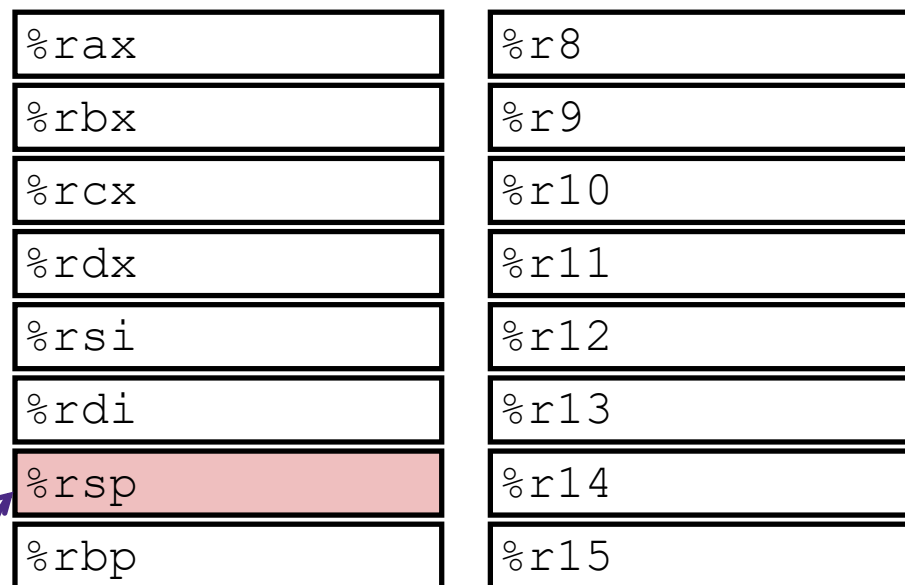
# x86 Control Flow

- ❖ **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:

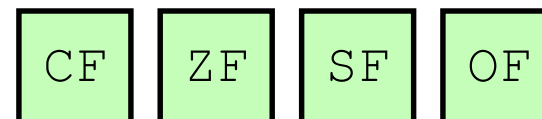
## Registers



current top of the Stack

`%rip`

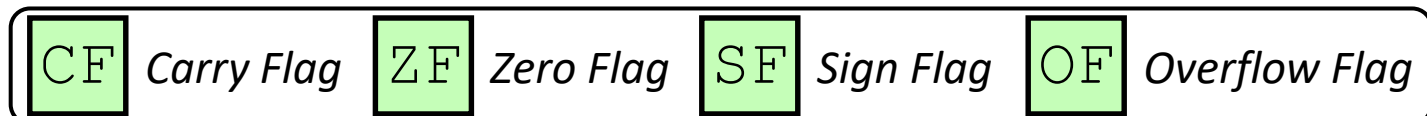
**Program Counter**  
(instruction pointer)



**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!)**

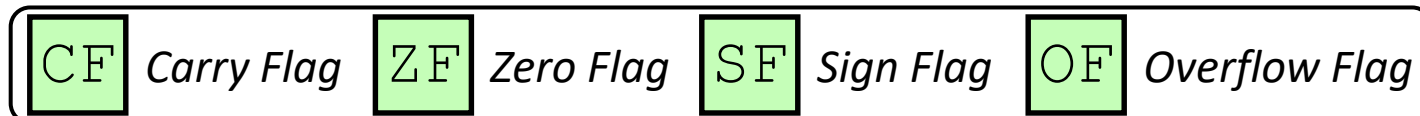




# Condition Codes (Explicit Setting: Compare)

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

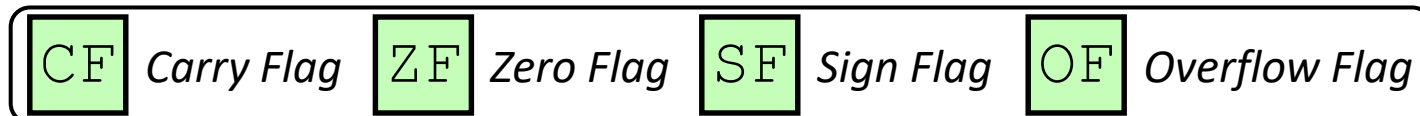
- `cmpq src1, src2`
- `cmpq a, b` sets flags based on  $b-a$ , but doesn't store
- **CF=1** if carry out from MSB (good for *unsigned* comparison)
- **ZF=1** if  $a==b$
- **SF=1** if  $(b-a) < 0$  (if MSB is 1)
- **OF=1** if *signed* overflow  
 $(a > 0 \ \&\& \ b < 0 \ \&\& \ (b-a) > 0) \ ||$   
 $(a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



# 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)



# 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`**?

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

## ❖ 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    #
setg    %al           #
movzbl  %al, %eax     #
ret
```

# Reading Condition Codes

Register	Use(s)
%rdi	1 <sup>st</sup> argument (x)
%rsi	2 <sup>nd</sup> 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) 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
```

# Choosing instructions for conditionals

- ❖ All arithmetic instructions set condition flags based on result of operation (op)
  - Conditionals are comparisons against 0
- ❖ Come in instruction *pairs*

```

addq 5, (p)
je:   *p+5 == 0
jne:  *p+5 != 0
jg:   *p+5 > 0
jl:   *p+5 < 0

```

```

orq a, b
je:   b|a == 0
jne:  b|a != 0
jg:   b|a > 0
jl:   b|a < 0

```

		(op) s, d
<b>je</b>	"Equal"	d (op) s == 0
<b>jne</b>	"Not equal"	d (op) s != 0
<b>js</b>	"Sign" (negative)	d (op) s < 0
<b>jns</b>	(non-negative)	d (op) s >= 0
<b>jg</b>	"Greater"	d (op) s > 0
<b>jge</b>	"Greater or equal"	d (op) s >= 0
<b>jl</b>	"Less"	d (op) s < 0
<b>jle</b>	"Less or equal"	d (op) s <= 0
<b>ja</b>	"Above" (unsigned >)	d (op) s > 0U
<b>jb</b>	"Below" (unsigned <)	d (op) s < 0U



# Choosing instructions for conditionals

- ❖ Reminder: `cmp` is like `sub`, `test` is like `and`
  - Result is not stored anywhere

		<b>cmp a,b</b>	<b>test a,b</b>
<b>je</b>	“Equal”	$b == a$	$b \& a == 0$
<b>jne</b>	“Not equal”	$b != a$	$b \& a != 0$
<b>js</b>	“Sign” (negative)	$b - a < 0$	$b \& a < 0$
<b>jns</b>	(non-negative)	$b - a \geq 0$	$b \& a \geq 0$
<b>jg</b>	“Greater”	$b > a$	$b \& a > 0$
<b>jge</b>	“Greater or equal”	$b \geq a$	$b \& a \geq 0$
<b>jl</b>	“Less”	$b < a$	$b \& a < 0$
<b>jle</b>	“Less or equal”	$b \leq a$	$b \& a \leq 0$
<b>ja</b>	“Above” (unsigned >)	$b >_U a$	$b \& a > 0U$
<b>jb</b>	“Below” (unsigned <)	$b <_U a$	$b \& a < 0U$

```

cmpq 5, (p)
je:   *p == 5
jne:  *p != 5
jg:   *p > 5
jl:   *p < 5

```

```

testq a, a
je:   a == 0
jne:  a != 0
jg:   a > 0
jl:   a < 0

```

```

testb a, 0x1
je:   aLSB == 0
jne:  aLSB == 1

```

# Choosing instructions for conditionals

	<b>cmp a,b</b>	<b>test a,b</b>
<b>je</b> "Equal"	$b == a$	$b\&a == 0$
<b>jne</b> "Not equal"	$b != a$	$b\&a != 0$
<b>js</b> "Sign" (negative)	$b - a < 0$	$b\&a < 0$
<b>jns</b> (non-negative)	$b - a \geq 0$	$b\&a \geq 0$
<b>jg</b> "Greater"	$b > a$	$b\&a > 0$
<b>jge</b> "Greater or equal"	$b \geq a$	$b\&a \geq 0$
<b>jl</b> "Less"	$b < a$	$b\&a < 0$
<b>jle</b> "Less or equal"	$b \leq a$	$b\&a \leq 0$
<b>ja</b> "Above" (unsigned >)	$b >_U a$	$b\&a > 0U$
<b>jb</b> "Below" (unsigned <)	$b <_U a$	$b\&a < 0U$

Register	Use(s)
%rdi	argument x
%rsi	argument y
%rax	return value

```

if (x < 3) {
    return 1;
}
return 2;

```

```

cmpq $3, %rdi
jge T2

```

```

T1: # x < 3:
    movq $1, %rax
    ret
T2: # !(x < 3):
    movq $2, %rax
    ret

```

# Practice Question 1

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

- A. `cmpq %rsi, %rdi`  
`jle .L4`
- B. `cmpq %rsi, %rdi`  
`jg .L4`
- C. `testq %rsi, %rdi`  
`jle .L4`
- D. `testq %rsi, %rdi`  
`jg .L4`
- E. We're lost...

```
long absdiff(long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

absdiff:

```

_____
_____
                                     # x > y:
movq    %rdi, %rax
subq    %rsi, %rax
ret

.L4:                                     # x <= y:
movq    %rsi, %rax
subq    %rdi, %rax
ret
```

# Reading Review

- ❖ Terminology:
  - Label, jump target
  - Program counter
  - Jump table, indirect jump
- ❖ Questions from the Reading?

# Labels

**swap:**

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

**max:**

```
movq %rdi, %rax
cmpq %rsi, %rdi
jg done
movq %rsi, %rax
done:
ret
```

- ❖ A jump changes the program counter (`%rip`)
  - `%rip` tells the CPU the *address* of the next instruction to execute
- ❖ **Labels** give us a way to refer to a specific instruction in our assembly/machine code
  - Associated with the *next* instruction found in the assembly code (ignores whitespace)
  - Each *use* of the label will eventually be replaced with something that indicates the final address of the instruction that it is associated with

# x86 Control Flow

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

# Expressing with Goto Code

```
long absdiff(long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff_j(long x, long y)
{
    long result;
    int ntest = (x <= y);
    if (ntest) goto Else;
    result = x-y;
    goto Done;
Else:
    result = y-x;
Done:
    return result;
}
```

- ❖ C allows goto as means of transferring control (jump)
  - Closer to assembly programming style
  - Generally considered bad coding style

# Compiling Loops

C/Java code:

```
while ( sum != 0 ) {  
    <loop body>  
}
```

Assembly code:

```
loopTop:    testq %rax, %rax  
            je      loopDone  
            <loop body code>  
            jmp     loopTop  
  
loopDone:
```

- ❖ Other loops compiled similarly
  - Will show variations and complications in coming slides, but may skip a few examples in the interest of time
- ❖ Most important to consider:
  - When should conditionals be evaluated? (*while* vs. *do-while*)
  - How much jumping is involved?



# Compiling Loops

## While Loop:

C: **while** ( sum != 0 ) {  
     <loop body>  
 }

x86-64:

```

loopTop:  testq %rax, %rax
          je    loopDone
          <loop body code>
          jmp  loopTop

loopDone:
  
```

## Do-while Loop:

C: **do** {  
     <loop body>  
 } **while** ( sum != 0 )

x86-64:

```

loopTop:
  <loop body code>
  testq %rax, %rax
  jne  loopTop

loopDone:
  
```

## While Loop (ver. 2):

C: **while** ( sum != 0 ) {  
     <loop body>  
 }

x86-64:

```

          testq %rax, %rax
          je    loopDone
loopTop:
  <loop body code>
  testq %rax, %rax
  jne  loopTop

loopDone:
  
```

# For-Loop → While-Loop

For-Loop:

```
for (Init; Test; Update) {  
    Body  
}
```



While-Loop Version:

```
Init;  
while (Test) {  
    Body  
    Update;  
}
```

Caveat: C and Java have `break` and `continue`

- Conversion works fine for `break`
  - Jump to same label as loop exit condition
- But not `continue`: would skip doing `Update`, which it should do with for-loops
  - Introduce new label at `Update`

## Practice Question 2

- ❖ The following is assembly code for a for-loop; identify the corresponding parts (Init, Test, Update)

- $i \rightarrow \%eax$ ,  $x \rightarrow \%rdi$ ,  $y \rightarrow \%esi$

Line

```
1      movl    $0, %eax
2  .L2:  cmpl    %esi, %eax
3      jge    .L4
4      movslq  %eax, %rdx
5      leaq   (%rdi,%rdx,4), %rcx
6      movl   (%rcx), %edx
7      addl   $1, %edx
8      movl   %edx, (%rcx)
9      addl   $1, %eax
10     jmp    .L2
11  .L4:
```

# Summary

- ❖ Control flow in x86 determined by 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
  - Most control flow constructs (*e.g.*, if-else, for-loop, while-loop) can be implemented in assembly using combinations of conditional and unconditional jumps