

# **x86-64 Programming II**

CSE 351 Autumn 2017

## **Instructor:**

Justin Hsia

## **Teaching Assistants:**

Lucas Wotton

Michael Zhang

Parker DeWilde

Ryan Wong

Sam Gehman

Sam Wolfson

Savanna Yee

Vinny Palaniappan

# Administrivia

- ❖ Lab 2 (x86-64) released tomorrow (10/17)
  - Learn to read x86-64 assembly and use GDB
- ❖ Homework 2 due Friday (10/20)
  
- ❖ Midterm is in two Mondays (10/30, 5pm in KNE 120)
  - 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!

# 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
    ???
    ???
    movq    %rsi, %rax
    ???
    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 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 {...}`

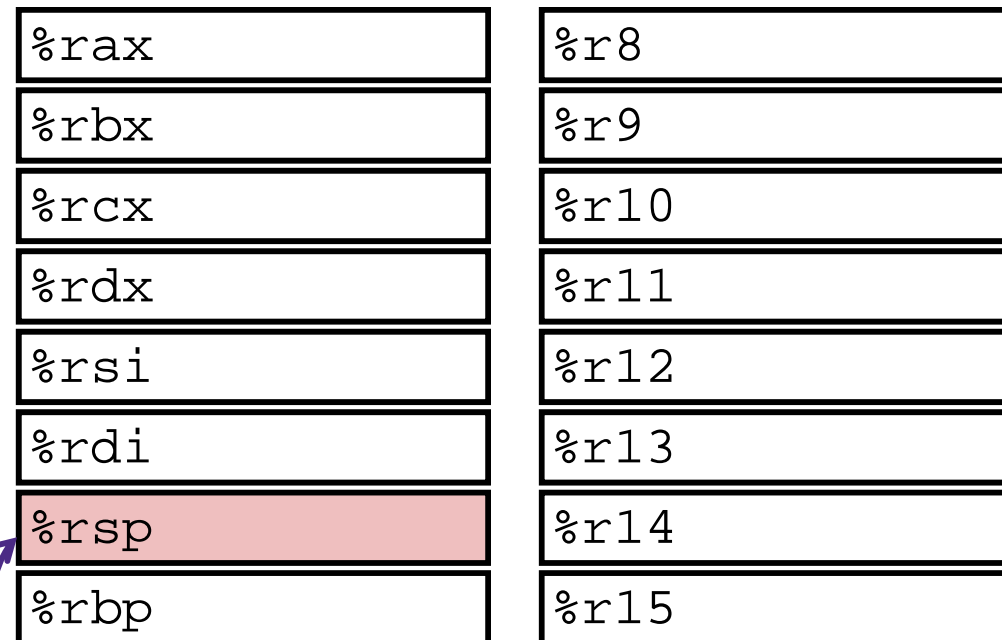
# x86 Control Flow

- ❖ **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` )
    - Single bit registers:

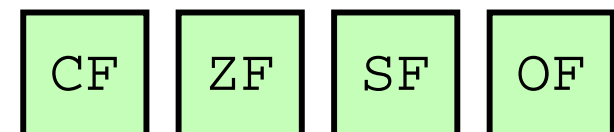
## Registers



current top of the Stack



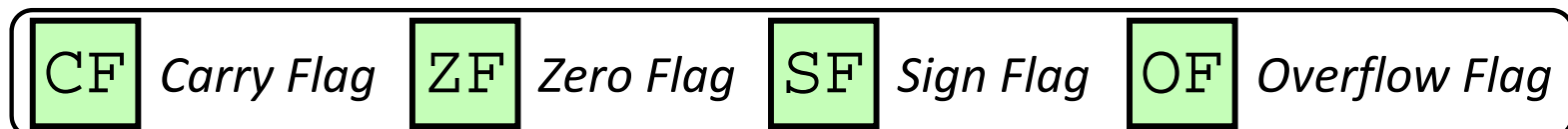
**Program Counter**  
(instruction pointer)



**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  $r<0$  (assuming signed, actually just if MSB is 1)
  - **OF=1** if two's complement (signed) overflow  
( $s>0 \ \&\& \ d>0 \ \&\& \ r<0$ )  $\| \ \|$  ( $s<0 \ \&\& \ d<0 \ \&\& \ r>=0$ )
  - **Not set by lea instruction (beware!)**

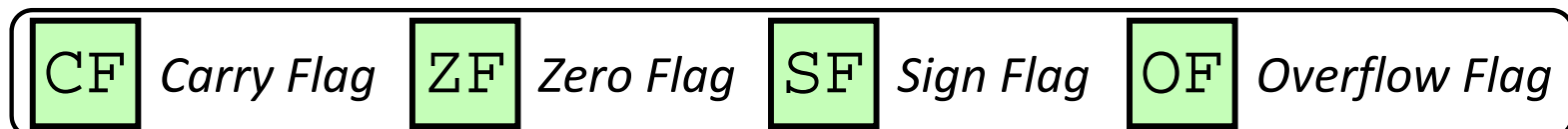




# Condition Codes (Explicit Setting: Compare)

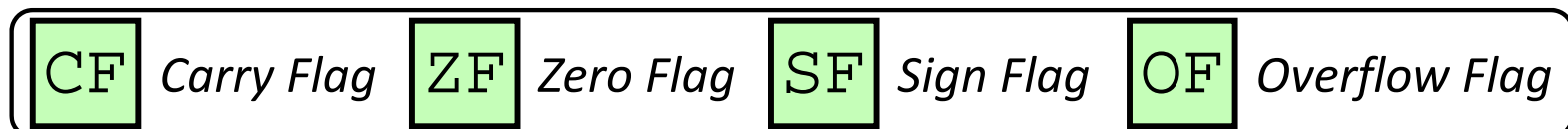
## ❖ *Explicitly* set by **Compare** instruction

- `cmpq src1, src2`
- `cmpq a, b` sets flags based on  $b-a$ , but doesn't store
- **CF=1** if carry out from MSB (used for unsigned comparison)
- **ZF=1** if  $a==b$
- **SF=1** if  $(b-a) < 0$  (signed)
- **OF=1** if two's complement (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: `testq %rax, %rax`
    - Tells you if (+), 0, or (-) based on ZF and SF



# 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>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 "<")

# Reminder: x86-64 Integer Registers

## ❖ Accessing the low-order byte:

<code>%rax</code>	<code>%al</code>
<code>%rbx</code>	<code>%bl</code>
<code>%rcx</code>	<code>%cl</code>
<code>%rdx</code>	<code>%dl</code>
<code>%rsi</code>	<code>%sil</code>
<code>%rdi</code>	<code>%dil</code>
<code>%rsp</code>	<code>%spl</code>
<code>%rbp</code>	<code>%bpl</code>

<code>%r8</code>	<code>%r8b</code>
<code>%r9</code>	<code>%r9b</code>
<code>%r10</code>	<code>%r10b</code>
<code>%r11</code>	<code>%r11b</code>
<code>%r12</code>	<code>%r12b</code>
<code>%r13</code>	<code>%r13b</code>
<code>%r14</code>	<code>%r14b</code>
<code>%r15</code>	<code>%r15b</code>

# 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, 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    #
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, 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)

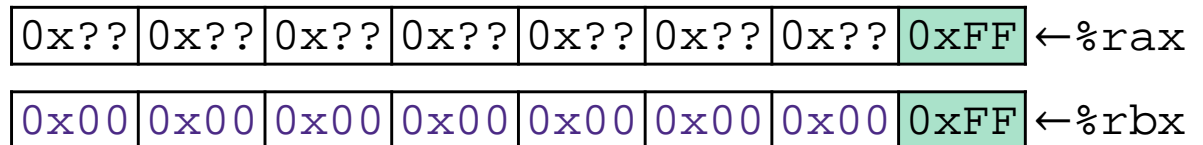
movzSD / movsSD:

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)

movzSD / movsSD:

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

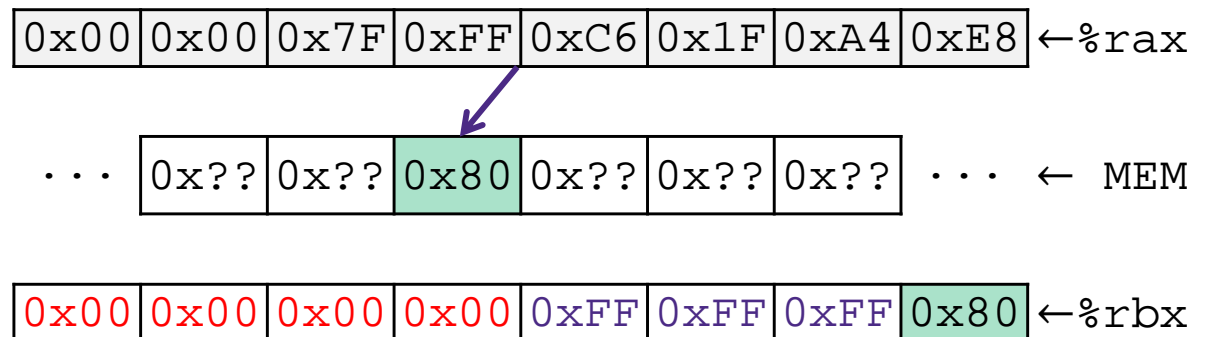
D – size of dest (**w** = 2 bytes, **l** = 4, **q** = 8)

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



# 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

	<code>cmp a,b</code>	<code>test a,b</code>
<b>je</b> "Equal"	<code>b == a</code>	<code>b&amp;a == 0</code>
<b>jne</b> "Not equal"	<code>b != a</code>	<code>b&amp;a != 0</code>
<b>js</b> "Sign" (negative)	<code>b-a &lt; 0</code>	<code>b&amp;a &lt; 0</code>
<b>jns</b> (non-negative)	<code>b-a &gt;= 0</code>	<code>b&amp;a &gt;= 0</code>
<b>jg</b> "Greater"	<code>b &gt; a</code>	<code>b&amp;a &gt; 0</code>
<b>jge</b> "Greater or equal"	<code>b &gt;= a</code>	<code>b&amp;a &gt;= 0</code>
<b>jl</b> "Less"	<code>b &lt; a</code>	<code>b&amp;a &lt; 0</code>
<b>jle</b> "Less or equal"	<code>b &lt;= a</code>	<code>b&amp;a &lt;= 0</code>
<b>ja</b> "Above" (unsigned >)	<code>b &gt; a</code>	<code>b&amp;a &gt; 0U</code>
<b>jb</b> "Below" (unsigned <)	<code>b &lt; a</code>	<code>b&amp;a &lt; 0U</code>

```

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

	<code>cmp a,b</code>	<code>test a,b</code>
<b>je</b> "Equal"	<code>b == a</code>	<code>b&amp;a == 0</code>
<b>jne</b> "Not equal"	<code>b != a</code>	<code>b&amp;a != 0</code>
<b>js</b> "Sign" (negative)	<code>b-a &lt; 0</code>	<code>b&amp;a &lt; 0</code>
<b>jns</b> (non-negative)	<code>b-a &gt;= 0</code>	<code>b&amp;a &gt;= 0</code>
<b>jg</b> "Greater"	<code>b &gt; a</code>	<code>b&amp;a &gt; 0</code>
<b>jge</b> "Greater or equal"	<code>b &gt;= a</code>	<code>b&amp;a &gt;= 0</code>
<b>jl</b> "Less"	<code>b &lt; a</code>	<code>b&amp;a &lt; 0</code>
<b>jle</b> "Less or equal"	<code>b &lt;= a</code>	<code>b&amp;a &lt;= 0</code>
<b>ja</b> "Above" (unsigned >)	<code>b &gt; a</code>	<code>b&amp;a &gt; 0U</code>
<b>jb</b> "Below" (unsigned <)	<code>b &lt; a</code>	<code>b&amp;a &lt; 0U</code>

Register	Use(s)
<code>%rdi</code>	argument x
<code>%rsi</code>	argument y
<code>%rax</code>	return value

```

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

```

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

# Question

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

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

- 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...

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

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

# Choosing instructions for conditionals

		<code>cmp a,b</code>	<code>test a,b</code>
<code>je</code>	"Equal"	<code>b == a</code>	<code>b&amp;a == 0</code>
<code>jne</code>	"Not equal"	<code>b != a</code>	<code>b&amp;a != 0</code>
<code>js</code>	"Sign" (negative)	<code>b-a &lt; 0</code>	<code>b&amp;a &lt; 0</code>
<code>jns</code>	(non-negative)	<code>b-a &gt;= 0</code>	<code>b&amp;a &gt;= 0</code>
<code>jg</code>	"Greater"	<code>b &gt; a</code>	<code>b&amp;a &gt; 0</code>
<code>jge</code>	"Greater or equal"	<code>b &gt;= a</code>	<code>b&amp;a &gt;= 0</code>
<code>jl</code>	"Less"	<code>b &lt; a</code>	<code>b&amp;a &lt; 0</code>
<code>jle</code>	"Less or equal"	<code>b &lt;= a</code>	<code>b&amp;a &lt;= 0</code>
<code>ja</code>	"Above" (unsigned >)	<code>b &gt; a</code>	<code>b&amp;a &gt; 0U</code>
<code>jb</code>	"Below" (unsigned <)	<code>b &lt; a</code>	<code>b&amp;a &lt; 0U</code>

```

if (x < 3 && x == y) {
    return 1;
} else {
    return 2;
}

```

```

T1: # x < 3 && x == y:
    movq $1, %rax
    ret
T2: # else
    movq $2, %rax
    ret

```

# Choosing instructions for conditionals

		<code>cmp a,b</code>	<code>test a,b</code>
<code>je</code>	"Equal"	<code>b == a</code>	<code>b&amp;a == 0</code>
<code>jne</code>	"Not equal"	<code>b != a</code>	<code>b&amp;a != 0</code>
<code>js</code>	"Sign" (negative)	<code>b-a &lt; 0</code>	<code>b&amp;a &lt; 0</code>
<code>jns</code>	(non-negative)	<code>b-a &gt;= 0</code>	<code>b&amp;a &gt;= 0</code>
<code>jg</code>	"Greater"	<code>b &gt; a</code>	<code>b&amp;a &gt; 0</code>
<code>jge</code>	"Greater or equal"	<code>b &gt;= a</code>	<code>b&amp;a &gt;= 0</code>
<code>jl</code>	"Less"	<code>b &lt; a</code>	<code>b&amp;a &lt; 0</code>
<code>jle</code>	"Less or equal"	<code>b &lt;= a</code>	<code>b&amp;a &lt;= 0</code>
<code>ja</code>	"Above" (unsigned >)	<code>b &gt; a</code>	<code>b&amp;a &gt; 0U</code>
<code>jb</code>	"Below" (unsigned <)	<code>b &lt; a</code>	<code>b&amp;a &lt; 0U</code>

❖ <https://godbolt.org/g/KntpyG>

```

if (x < 3 && x == y) {
    return 1;
} else {
    return 2;
}

```

```

cmpq $3, %rdi
setl %al
cmpq %rsi, %rdi
sete %bl
testb %al, %bl
je T2

```

```

T1: # x < 3 && x == y:

```

```

    movq $1, %rax
    ret

```

```

T2: # else

```

```

    movq $2, %rax
    ret

```

# 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