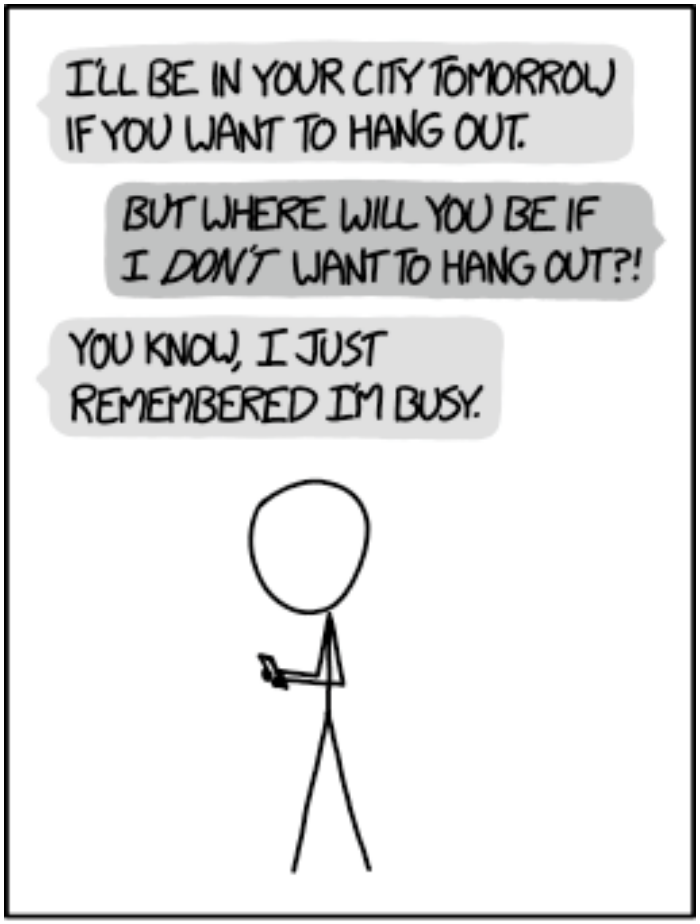
x86-64 Programming III

CSE 351 Spring 2022

Instructor:
Ruth Anderson

Teaching Assistants:
Melissa Birchfield
Jacob Christy
Alena Dickmann
Kyrie Dowling
Ellis Haker
Maggie Jiang
Diya Joy
Anirudh Kumar
Jim Limprasert
Armin Magness
Hamsa Shankar
Dara Stotland
Jeffery Tian
Assaf Vayner
Tom Wu
Angela Xu
Effie Zheng



http://xkcd.com/1652/

Relevant Course Information

- hw8 due TONIGHT (4/18) @ 11:59 pm
- Lab 1b, due TONIGHT, Monday 4/18 @ 11:59 pm
  - Submit `aisle_manager.c`, `store_client.c`, and `lab1Bsynthesis.txt`
  - Submissions that fail the autograder get a **ZERO**
    - No excuses – make full use of tools & Gradescope's interface
- Lab 2 (x86-64) due next Friday (4/29)
  - Learn to read x86-64 assembly and use GDB

Move extension: `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`:

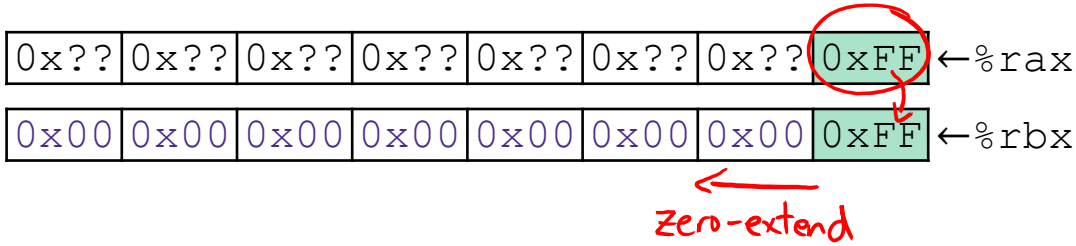
S – size of source (**`b`** = 1 byte, **`w`** = 2)

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

Example:

```
movzbq %al, %rbx
```

zero-extend; 1 byte; 8 bytes



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`)

movz*SD* / movs*SD*:

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

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

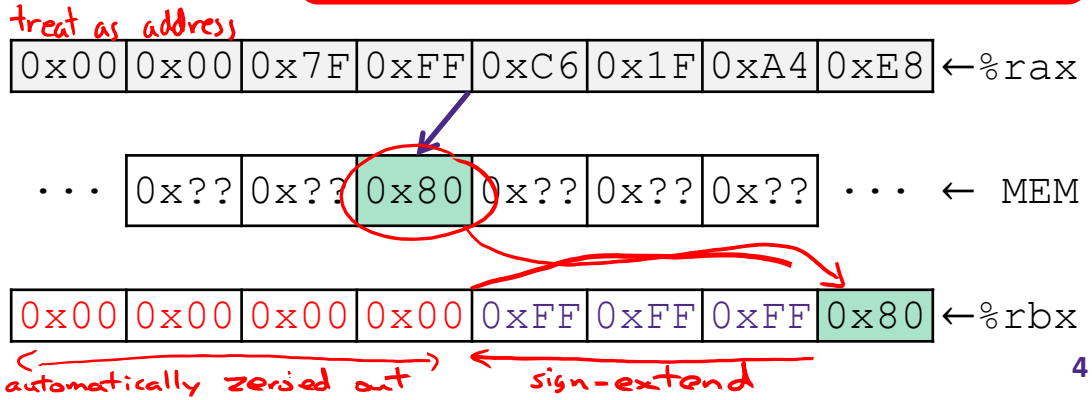
Note: In x86-64, *any instruction* that generates a 32-bit (long word) value for a register also sets the high-order portion of the register to 0. Good example on p. 184 in the textbook.

Example:

```
movsbl (%rax), %ebx
```

1 byte, sign-extend, 4 bytes

Copy 1 byte from memory into 8-byte register & sign extend it

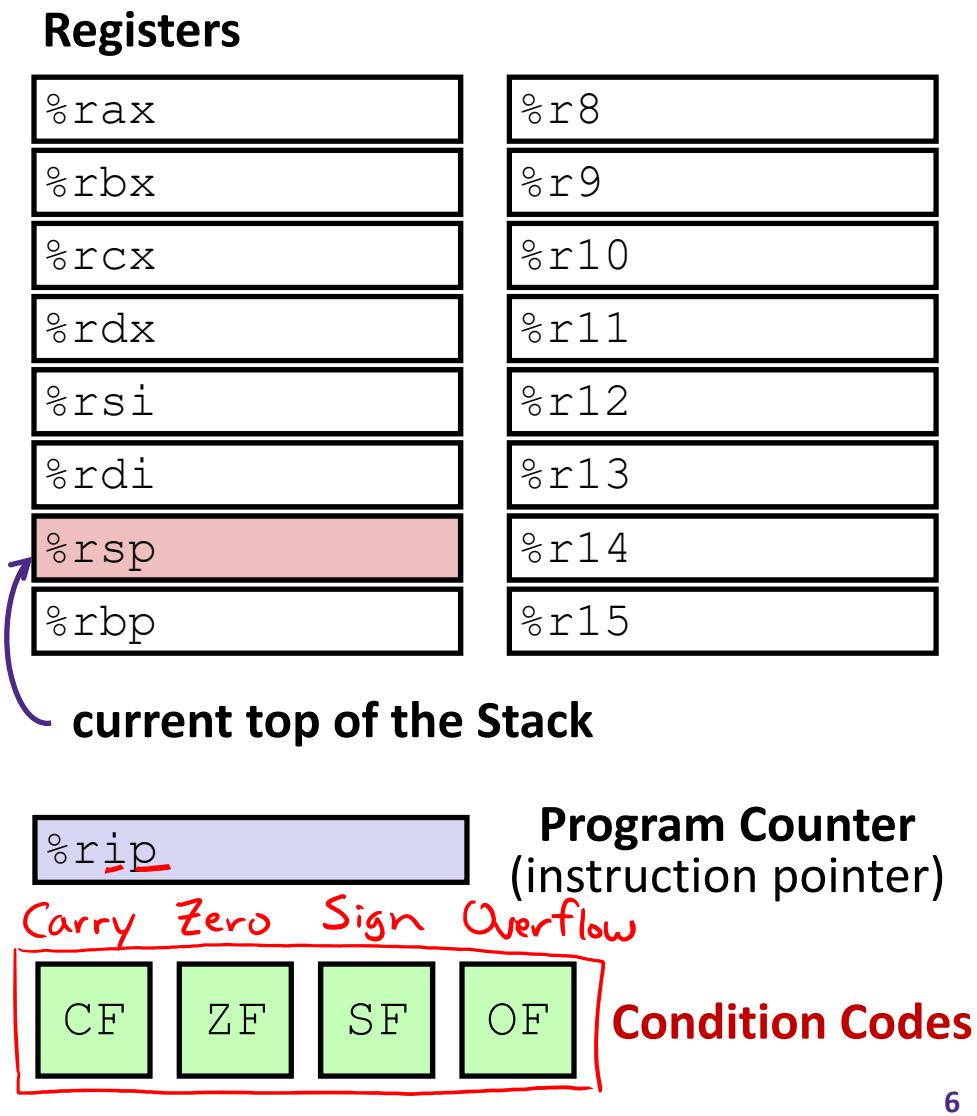


x86 Control Flow

- **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:



Condition Codes (Implicit Setting)

- *Implicitly* set by **arithmetic** operations
  - (think of it as side effects)
  - Example: **`addq`** `src, dst` ↔ `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!)

CF	*Carry Flag*	ZF	*Zero Flag*	SF	*Sign Flag*	OF	*Overflow Flag*

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)

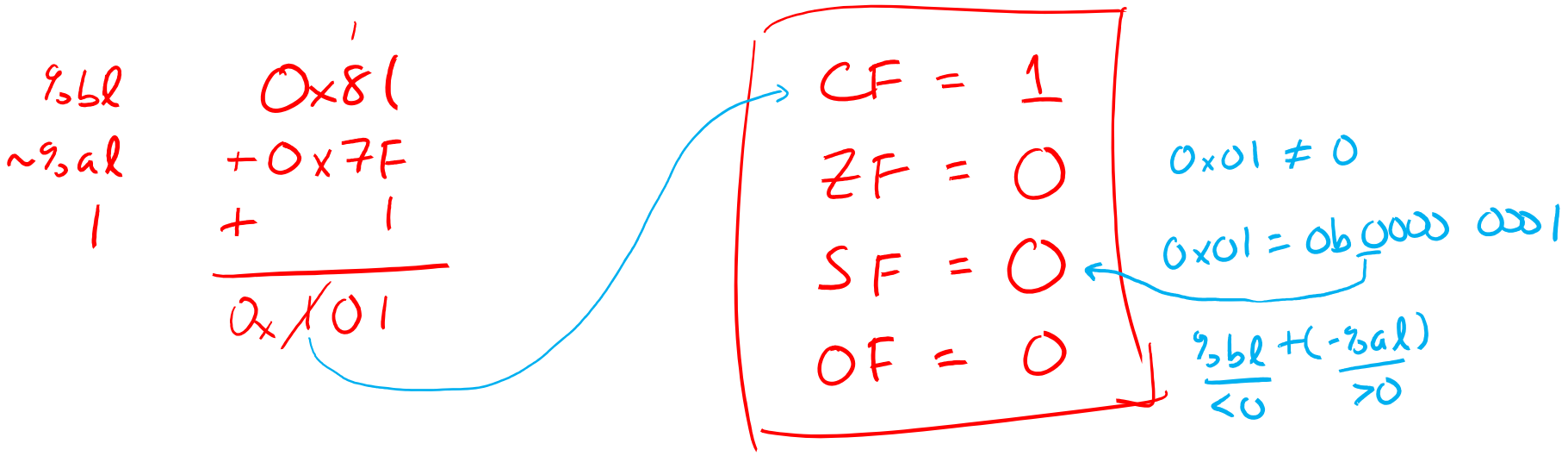
- *Explicitly* set by **Test** instruction
  - **`testq`** `src2, src1` like andq a, b
  - **`testq`** `a, b` sets flags based on `a&b`, but doesn't store
    - Useful to have one of the operands be a ***mask***
  - Can't have carry out (**CF**) or overflow (**OF**) = 0 = 0
  - **ZF=1** if `a&b==0`
  - **SF=1** if `a&b<0` (signed)

CF	*Carry Flag*	ZF	*Zero Flag*	SF	*Sign Flag*	OF	*Overflow Flag*

Example Condition Code Setting

- Assuming that %al = 0x80 and %bl = 0x81, which flags (CF, ZF, SF, OF) are set when we execute **`cmpb %al, %bl`**? → computes %bl - %al = %bl + ~%al + 1

~%al = ~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 (always compared to 0)
`jmp` *target*	`1`	Unconditional
`je` *target*	`ZF`	Equal / Zero
`jne` *target*	`~ZF`	Not Equal / Not Zero
`js` *target*	`SF`	Negative
`jns` *target*	`~SF`	Nonnegative
`jg` *target*	`~(SF^OF)&~ZF`	Greater (Signed)
`jge` *target*	`~(SF^OF)`	Greater or Equal (Signed)
`jl` *target*	`(SF^OF)`	Less (Signed)
`jle` *target*	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja` *target*	`~CF&~ZF`	Above (unsigned “>”)
`jb` *target*	`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

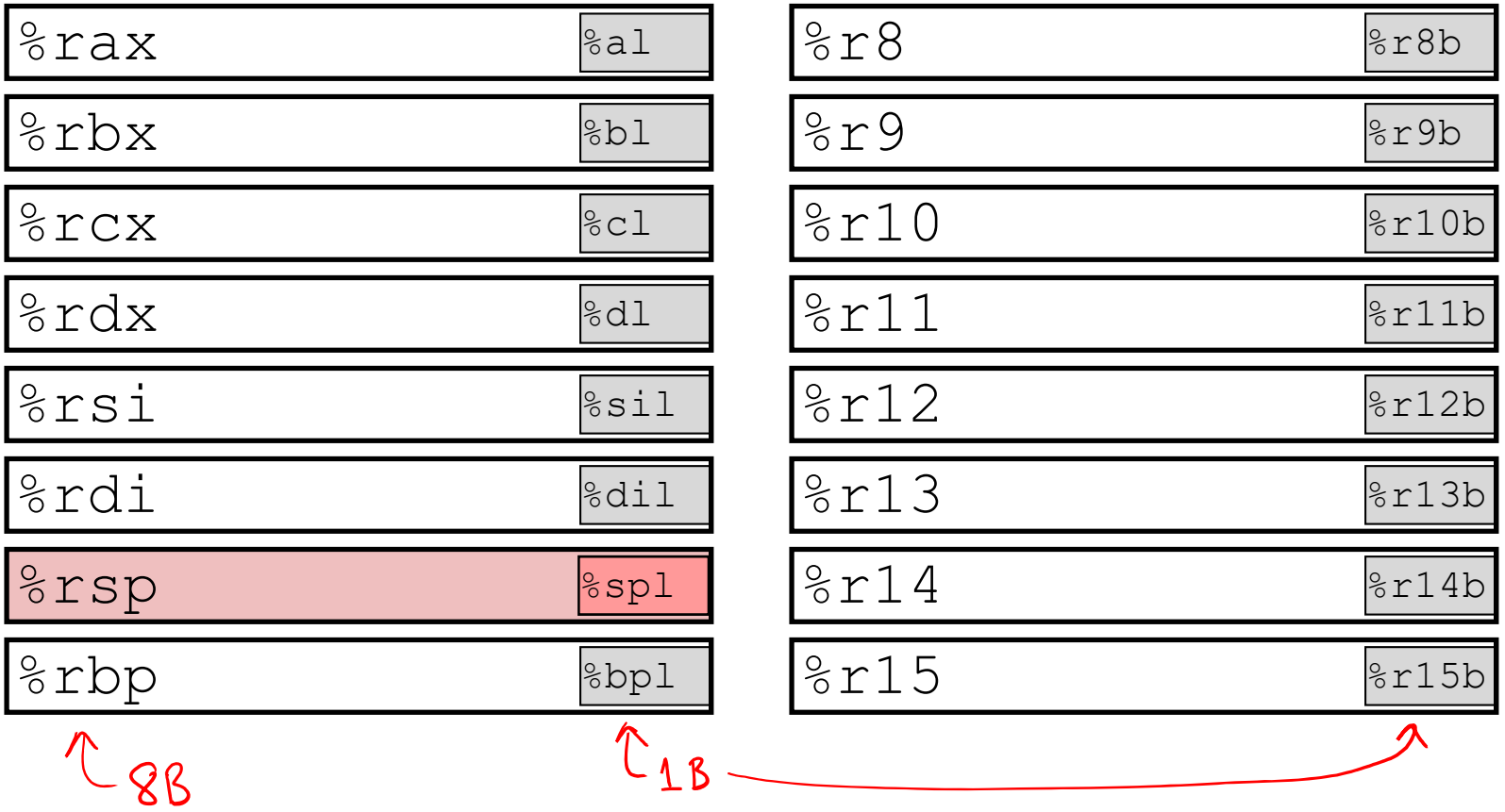
False → 0b 0000 0000 = 0x 00
True → 0b 0000 0001 = 0x 01

same instruction suffixes as j* instructions!

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

Reminder: x86-64 Integer Registers

- Accessing the low-order byte:



Reading Condition Codes

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

- set* Instructions (e, ne, g, l, ...)
  - 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;
}
```

(rdi, rsi)

x - y > 0

x > y

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

(y, x)

zero-extend → movzbl

← lowest byte

← whole register

Reading Condition Codes

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd 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
```

Reading Review

- Terminology:
  - Label, jump target
  - Program counter
  - Jump table, indirect jump

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
```

Pick one!

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

Pick one!

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

Choosing instructions for conditionals

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

		`cmp a,b`	**`test a,b`**
`je`	"Equal"	`b == a`	`b&a == 0`
`jne`	"Not equal"	`b != a`	`b&a != 0`
`js`	"Sign" (negative)	`b-a < 0`	`b&a < 0`
`jns`	(non-negative)	`b-a >=0`	`b&a >= 0`
`jg`	"Greater"	`b > a`	`b&a > 0`
`jge`	"Greater or equal"	`b >= a`	`b&a >= 0`
`jl`	"Less"	`b < a`	`b&a < 0`
`jle`	"Less or equal"	`b <= a`	`b&a <= 0`
`ja`	"Above" (unsigned >)	`b-a > 0U`	`b&a > 0U`
`jb`	"Below" (unsigned <)	`b-a < 0U`	`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:   a_LSB == 0
jne:  a_LSB == 1
```

Choosing instructions for conditionals

		cmp a,b	test a,b
je	"Equal"	b == a	b&a == 0
jne	"Not equal"	b != a	b&a != 0
js	"Sign" (negative)	b-a < 0	b&a < 0
jns	(non-negative)	b-a >=0	b&a >= 0
jg	"Greater"	b > a	b&a > 0
jge	"Greater or equal"	b >= a	b&a >= 0
jl	"Less"	b < a	b&a < 0
jle	"Less or equal"	b <= a	b&a <= 0
ja	"Above" (unsigned >)	b > a	b&a > 0U
jb	"Below" (unsigned <)	b < 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	1st argument (x)
%rsi	2nd 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
```

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

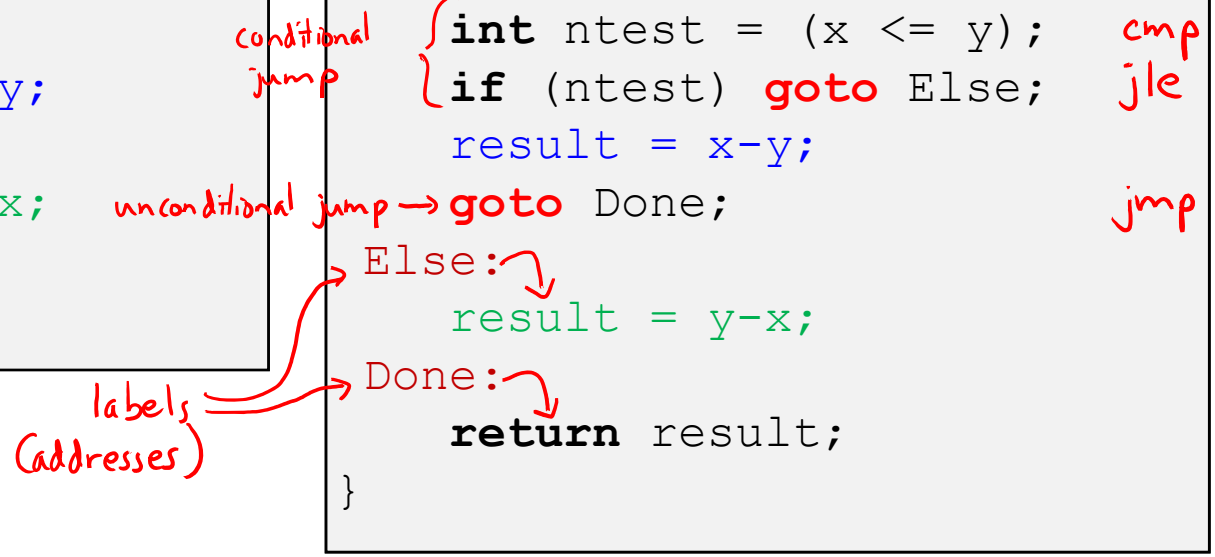
x86 Control Flow

- 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>
}
```

Test

Assembly code:

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

!Test

- 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

all jump instructions update the program counter (%rip)

While Loop:

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

x86-64:
```
loopTop:    testq %rax, %rax
            je    loopDone
            <loop body code>
            jmp   loopTop
loopDone:
```
(sum == 0; testq/je = ~Test)

Do-while Loop:

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

x86-64:
```
loopTop:
            <loop body code>
            testq  %rax, %rax
            jne    loopTop
loopDone:
```
(testq/jne = Test)

While Loop (ver. 2):

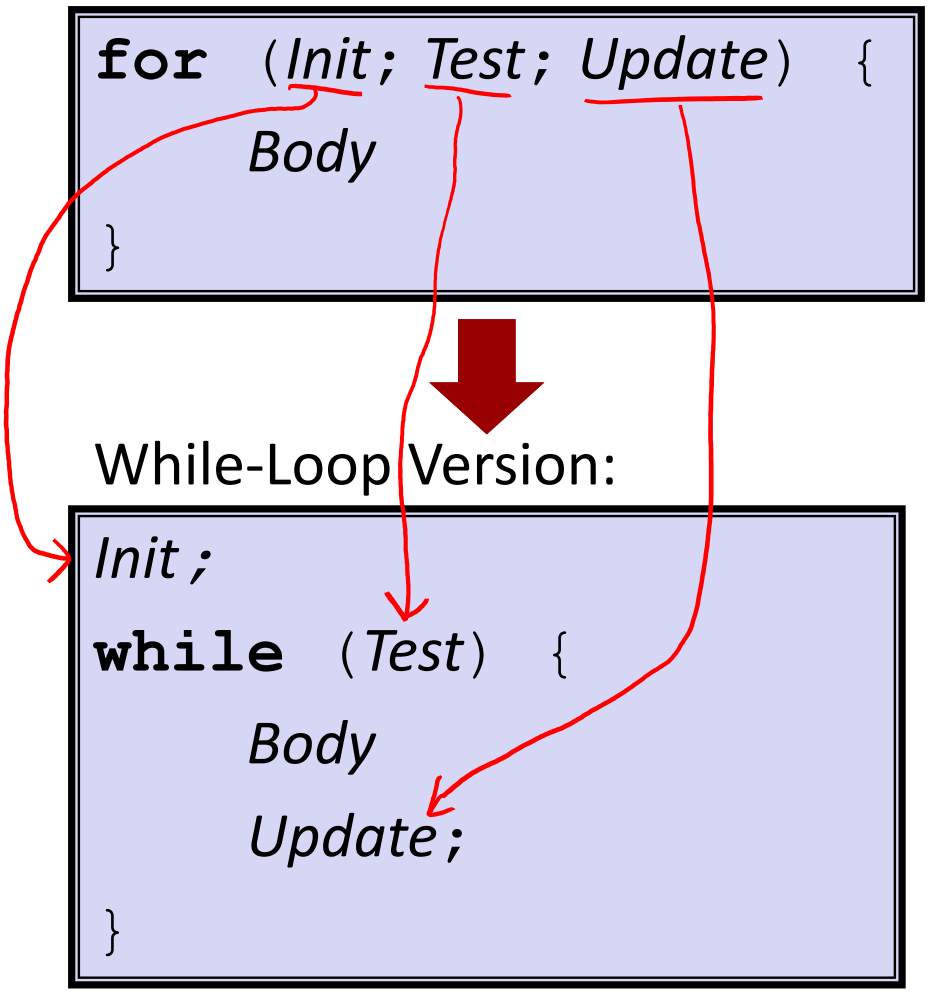
C:
```
while ( sum != 0 ) {
    <loop body>
}
```
(Test)

x86-64:
```
            testq %rax, %rax
            je    loopDone
loopTop:
            <loop body code>
            testq %rax, %rax
            jne   loopTop
loopDone:
```
(testq/je = ~Test; do-while loop; testq/jne = Test)

For-Loop → While-Loop

For-Loop:

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



While-Loop Version:

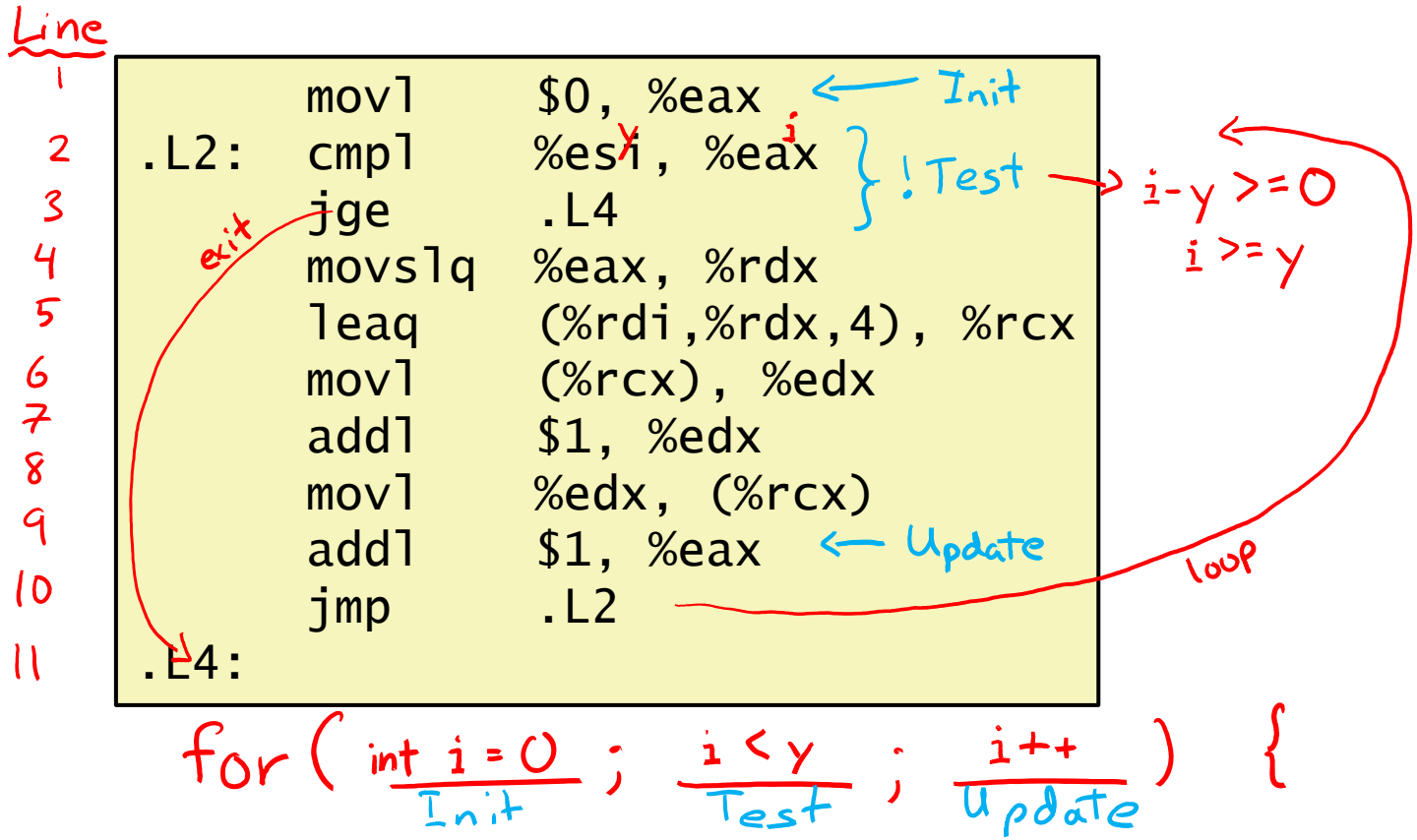
```
Init;
while (Test) {
    Body
    Update;
}
```

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 → %eax, x → %rdi, y → %esi



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