x86-64 Programming III

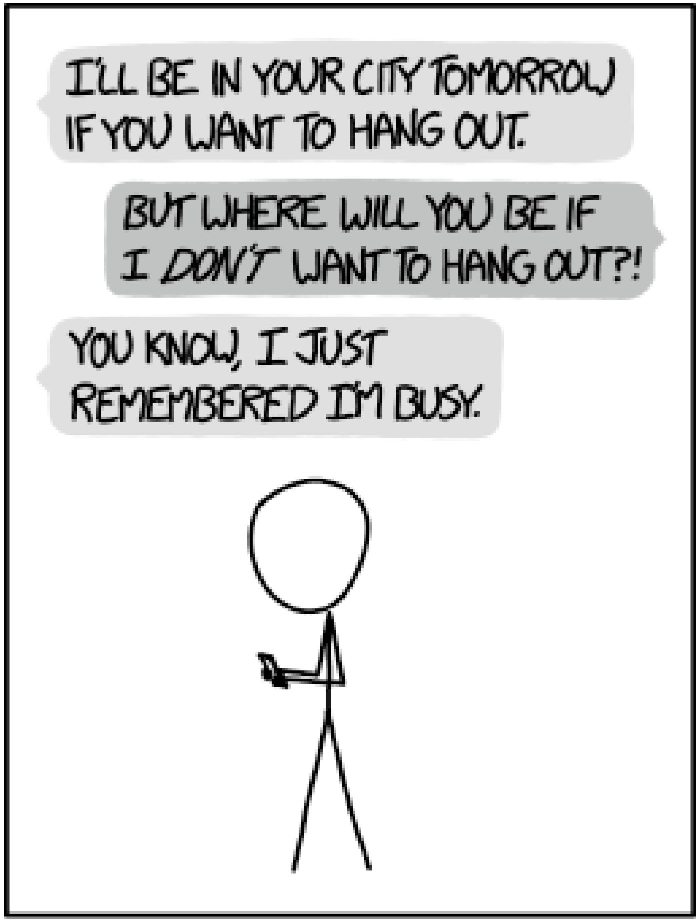
CSE 351 Autumn 2022

Instructor:

Justin Hsia

Teaching Assistants:

Angela Xu
Arjun Narendra
Armin Magness
Assaf Vayner
Carrie Hu
Clare Edmonds
David Dai
Dominick Ta
Effie Zheng
James Froelich
Jenny Peng
Kristina Lansang
Paul Stevans
Renee Ruan
Vincent Xiao



WHY I TRY NOT TO BE PEDANTIC ABOUT CONDITIONALS.

http://xkcd.com/1652/

Relevant Course Information

- Lab 1a regrade requests open on Gradescope
- Lab 1b submissions close tonight
- Lab 2 due next Friday (10/28)

- Section tomorrow on Assembly
  - Use the midterm reference sheet, bring your laptop!
  - 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`/`movs` (Review)

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:

8 bytes

`movzbq %al, %rbx`

zero-extend 1 byte

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

zero-extend

Move extension: `movz`/`movs` (Review)

`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)

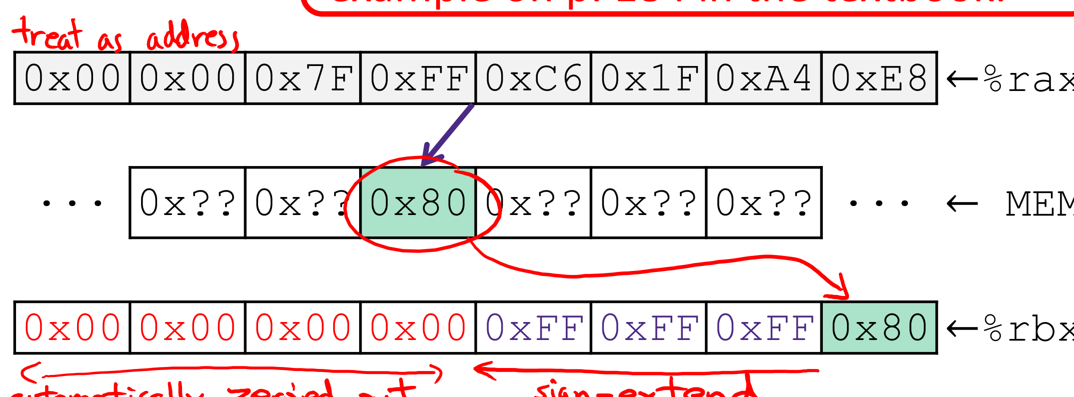
Note: In x86-64, *any instruction* that generates a 32-bit (long word) value for a register also sets the high-order portion of the register to 0. Good example on p. 184 in the textbook.

Example: 1 byte

`movsbl (%rax), %ebx`

sign-extend 4 bytes

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



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!

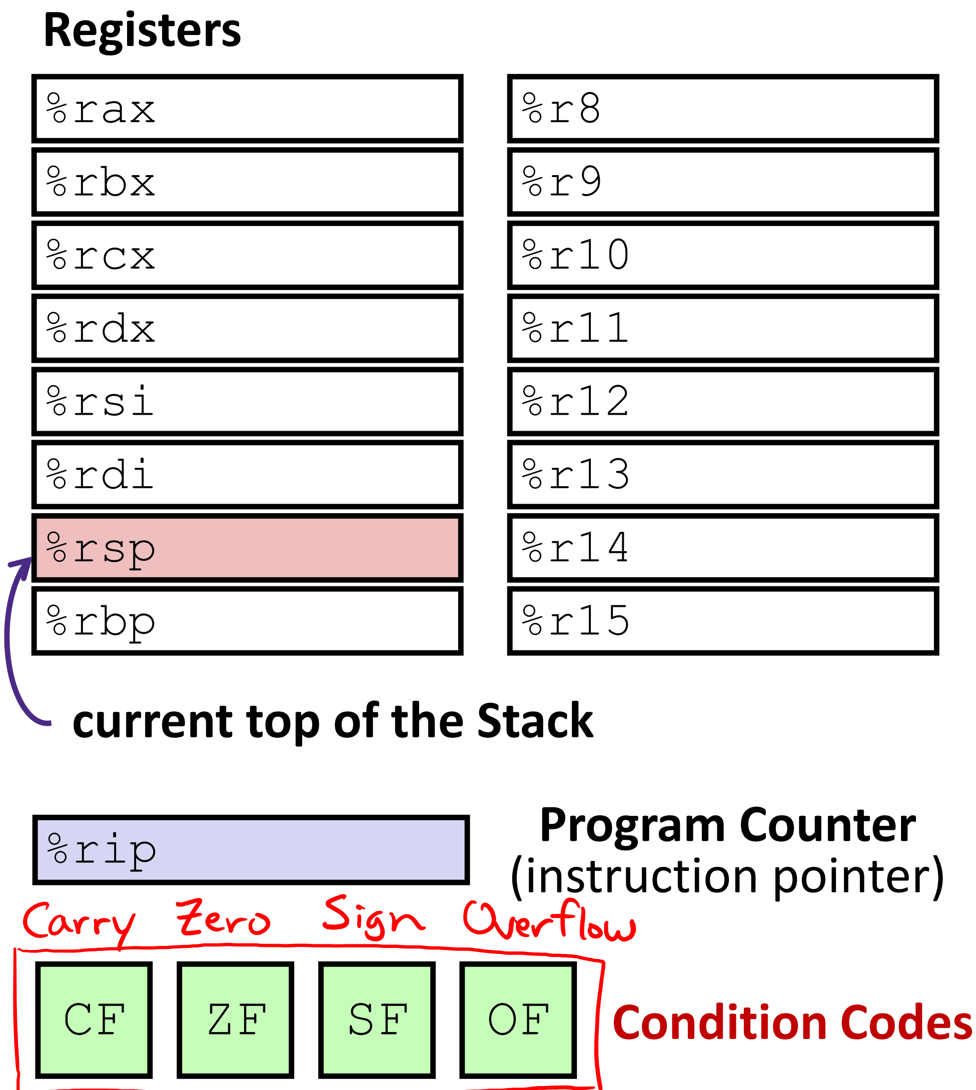
- Pay attention to:
  - Setting breakpoints (`break`)
  - Stepping through code (`step`/`next` and `stepi`/`nexti`)
  - Printing out expressions (`print` – works with regs & vars)
  - Examining memory (`x`)

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**) "flags"
    - Single bit registers:



Condition Codes (Implicit, RD9)

- *Implicitly* set by **arithmetic** operations
  - (think of it as side effects)
  - Example: **`addq`** `src, dst` ↔ `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>=0)`

    signs don't match!

  - *Not* set by `lea` instruction (beware!)

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

Condition Codes (Explicit: Compare, RD9)

- *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)`

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

Condition Codes (Explicit: Test, RD9)

- *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`** = 0) or overflow (**`OF`** = 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

%bl	0x81
~%al	+ 0x7F
1	+ 1
	0x~~1~~01

CF = 1
ZF = 0
SF = 0
OF = 0

0x01 ≠ 0

0x01 = 0b 0000 0001

$\underbrace{\%bl}_{<0} + \underbrace{(-\%al)}_{>0}$

Using Condition Codes: Jumping (RD9)

- `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 (RD9)

- `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 "<")

Reading Condition Codes

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

a(y), b(x)

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

zero-extend → movzbl

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*

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

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

		1 **(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
2 **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 >)	$b >_U a$	`b&a > 0U`
`jb`	"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:    a_LSB == 0
jne:   a_LSB == 1
```

Choosing instructions for conditionals

		① `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`
② `jge`	"Greater or equal"	`b >= a` (x, 3)	`b&a >= 0`
`jl`	"Less"	`b < a`	`b&a < 0`
`jle`	"Less or equal"	`b <= a`	`b&a <= 0`
`ja`	"Above" (unsigned >)	$b >_U a$	`b&a > 0U`
`jb`	"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;
```

do this if x ≥ 3

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

(if) (else)

labels

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` x-y

B. `cmpq %rsi, %rdi` / `jg .L4` x-y

~~C.~~ `testq %rsi, %rdi` / `jle .L4` x&y

~~D.~~ `testq %rsi, %rdi` / `jg .L4` x&y

E. We're lost…

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

x-y <= 0

less than or equal to (le)

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);       cmp
    if (ntest) goto Else;       jle
    result = x-y;
    goto Done;                  jmp
 Else:
    result = y-x;
 Done:
    return result;
}
```

conditional jump

unconditional jump

labels (addresses)

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

Compiling Loops (Review)

C/Java code:

Test

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

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 (Review)

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

~Test

Do-while Loop:

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

x86-64:
```
loopTop:
            <loop body code>
            testq  %rax, %rax
            jne    loopTop
loopDone:
```
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:
```
~Test

do-while loop

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`

```
        movl    $0, %eax
.L2:    cmpl    %esi, %eax
        jge     .L4
        movslq  %eax, %rdx
        leaq    (%rdi,%rdx,4), %rcx
        movl    (%rcx), %edx
        addl    $1, %edx
        movl    %edx, (%rcx)
        addl    $1, %eax
        jmp     .L2
.L4:
```

Init

y i

!Test

i-y >= 0

i >= y

exit

Update

loop

`for(` int i = 0 `;` i < y `;` i++ `)`

Init Test Update

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