x86-64 Programming III

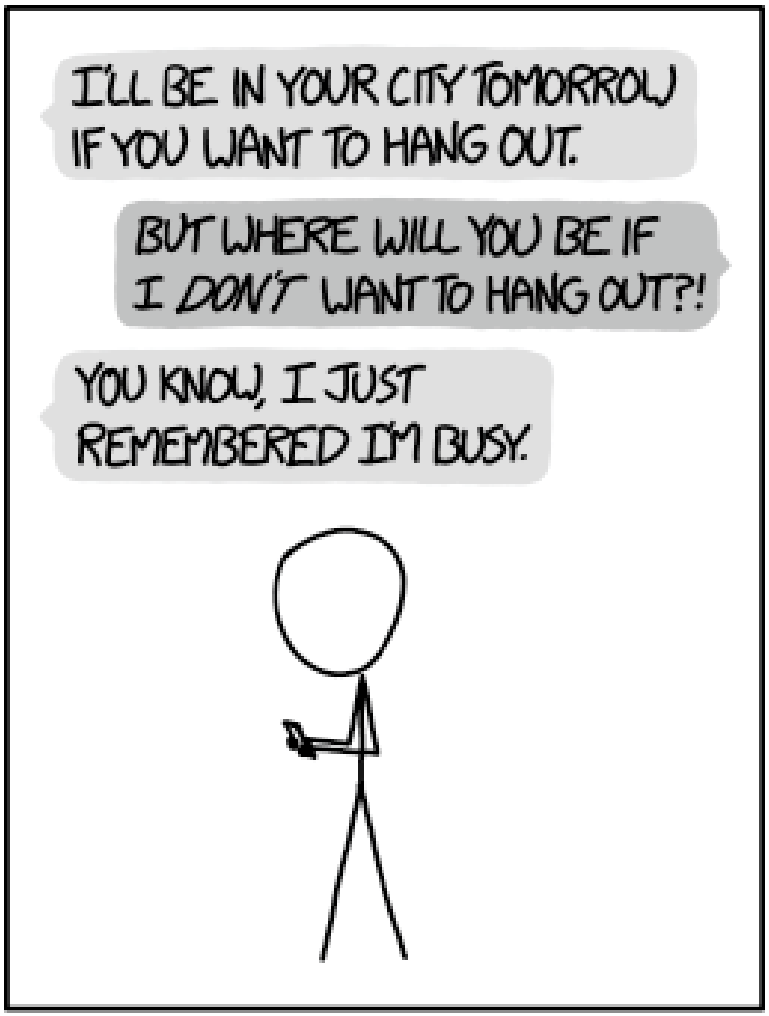
CSE 351 Autumn 2020

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



WHY I TRY NOT TO BE PEDANTIC ABOUT CONDITIONALS.

http://xkcd.com/1652/

Administrivia

- Lab 2 due next Friday (10/30)
- Section tomorrow on Assembly
  - Use the midterm reference sheet!
  - Optional GDB Tutorial slides and Lab 2 phase 1 walkthrough
- Midterm (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!

Aside: `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`:

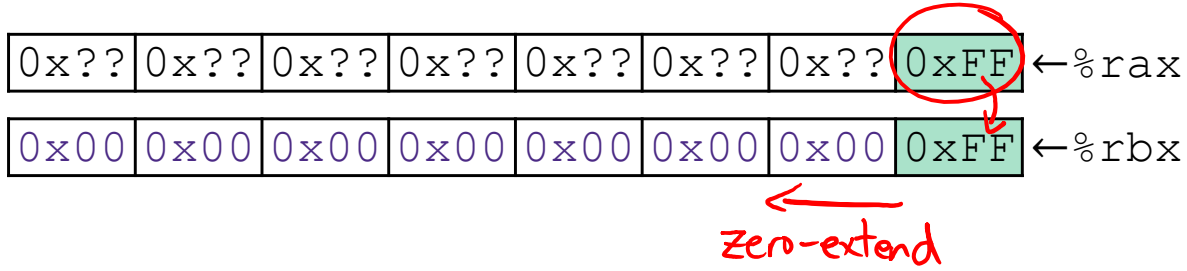
S – size of source (**`b`** = 1 byte, **`w`** = 2)

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

Example:

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

8 bytes; zero-extend; 1 byte



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`)

`movz`*SD* **/** **`movs`*SD***:

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

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

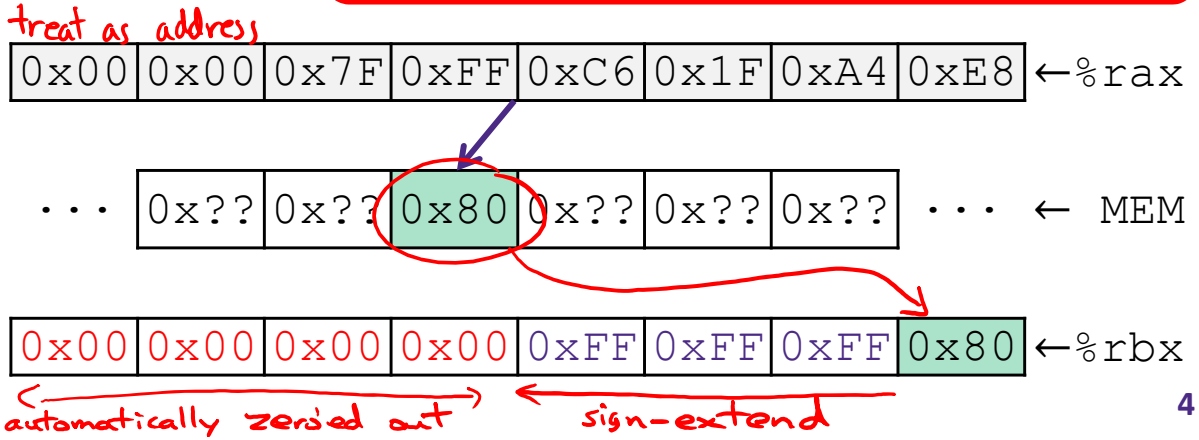
Note: In x86-64, *any instruction* that generates a 32-bit (long word) value for a register also sets the high-order portion of the register to 0. Good example on p. 184 in the textbook.

Example:

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

1 byte; sign-extend; 4 bytes

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



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`)

Reading Review

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

- Questions from the Reading?

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

		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

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

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    x-y <= 0
    subq    %rdi, %rax
    ret
```

less than or equal to (le)

Choosing instructions for conditionals

		cmp a,b	test a,b
je	"Equal"	② b == a	③ 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"	① 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

- https://godbolt.org/z/Tfrv33

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

%al %bl

do this if either %al or %bl are False

```
① cmpq $3, %rdi
  setl %al

② cmpq %rsi, %rdi
  sete %bl

③ testb %al, %bl
  je T2
```

%al = (x<3)

%bl = (x==y)

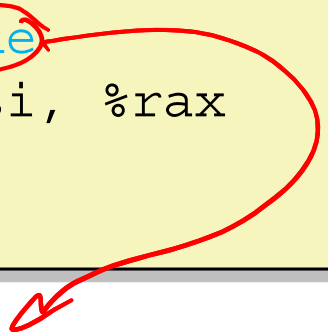
jump to T2 if (%al & %bl) == 0

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

conditional jump

cmp

jle

unconditional jump → goto Done;

jmp

labels (addresses)

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

Compiling Loops

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

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):

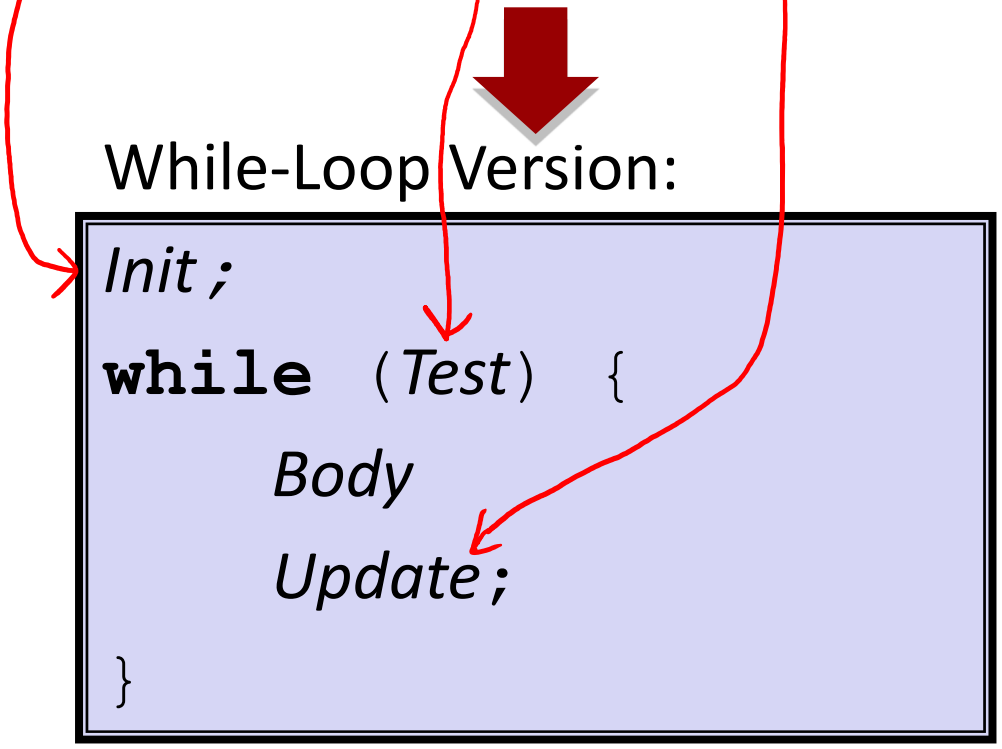
C:
```
while ( sum != 0 ) {
    <loop body>
}
```
(Test)

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