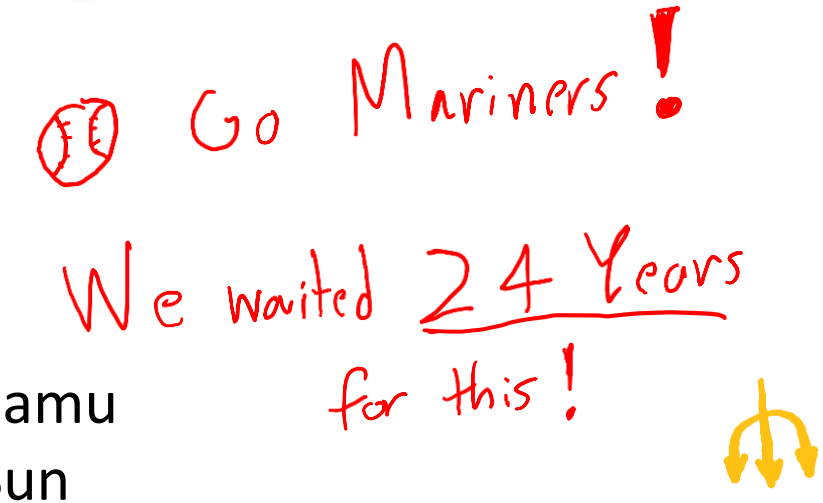
The Hardware/Software Interface

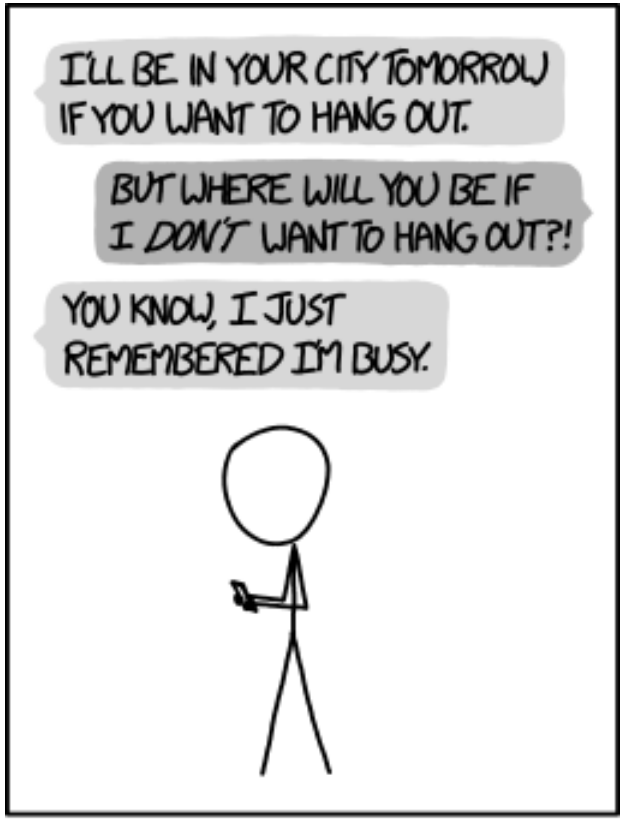
x86-64 Programming III

Instructors:
Amber Hu, Justin Hsia

Teaching Assistants:

Anthony Mangus	Divya Ramu
Grace Zhou	Jessie Sun
Jiuyang Lyu	Kanishka Singh
Kurt Gu	Liander Rainbolt
Mendel Carroll	Ming Yan
Naama Amiel	Pollux Chen
Rose Maresh	Soham Bhosale
Violet Monserate	





WHY I TRY NOT TO BE PEDANTIC ABOUT CONDITIONALS.

http://xkcd.com/1652/

Relevant Course Information

- Lab submissions that fail the autograder get a **ZERO**
  - No excuses – make full use of tools & Gradescope's interface
  - Leeway on Lab 1a won't be given moving forward

- Lab 2 (x86-64) released Wednesday
  - Learn to trace x86-64 assembly and use GDB

- Midterm is in two weeks (10/27, 5:30pm, location dependent on section)
  - No lecture that day
  - You will be provided a fresh midterm reference sheet
    - Study and use this NOW so you are comfortable with it when the exam comes around
  - Form study groups and look at past exams!

Lab Extra Credit

- Lab 2 and several other labs have "extra credit"
  - These are meant to be fun extensions to the labs – you are not expected to attempt these extra components

- Extra credit points *don't* affect your lab grades
  - From the course policies: "These will be accumulated over the course and may be used to bump up borderline grades at the end of the quarter at the discretion of the instructor(s)."
  - Make sure you finish the rest of the lab before attempting any extra credit

Lecture Outline (1/4)

- **Control Flow**
- Condition Codes
- Conditionals
- Instruction Set Philosophies, Revisited

Control Flow (1/2)

```
long max(long x, long y) {
  long max;
  if (x > y) {
    max = x;
  } else {
    max = y;
  }
  return max;
}
```

```
max:
   ???
   movq    %rdi, %rax  # if case
   ???
   ???
   movq    %rsi, %rax  # else case
   ???
   ret
```

Variable	Register
x	%rdi
y	%rsi
return value	%rax

Control Flow (2/2)

```
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
```

if TRUE

if FALSE

Variable	Register
x	%rdi
y	%rsi
return value	%rax

Conditionals and Control Flow

- **Conditional jump:** Jump to somewhere else if some *condition* is true, otherwise execute next instruction
- **Unconditional jump:** *Always* jump when you get to this instruction

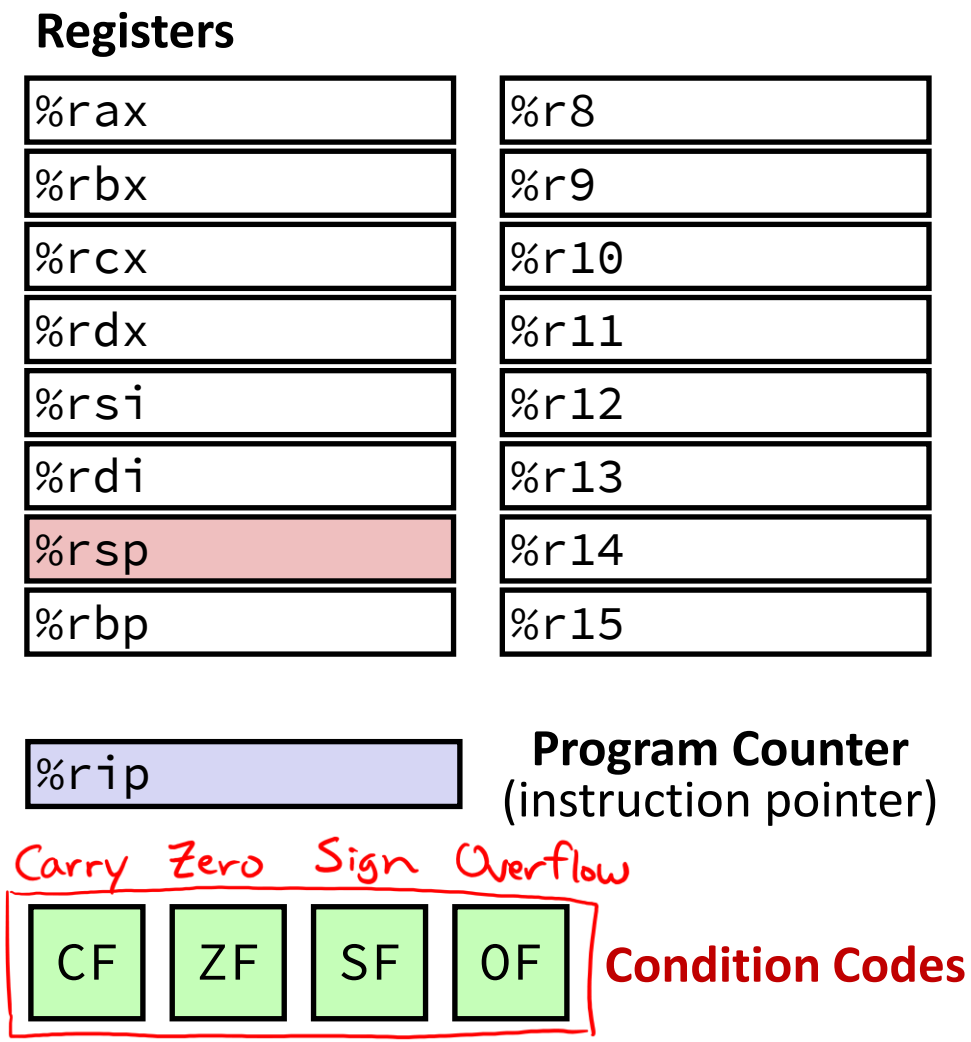
- With just these two types of jumps, we can implement most control flow constructs in higher-level languages:
  - **`if`** (*condition*) **`then`** {…} **`else`** {…}
  - **`while`** (*condition*) {…}
  - **`do`** {…} **`while`** (*condition*)
  - **`for`** (*initialization*; *condition*; *iterative*) {…}
  - **`switch`** {…}

Lecture Outline (2/4)

- Control Flow
- **Condition Codes**
- Conditionals
- Instruction Set Philosophies, Revisited

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:



Condition Codes (Review)

- Condition codes are set based on the result of the *previous* instruction(s)
  - These are automatically updated by your CPU as instructions are executed
  - Some instructions don't affect the condition codes (*e.g.*, `mov`, `ret`, `lea`)

- **Carry:** `CF=1` if result had *unsigned* overflow
- **Zero:** `ZF=1` if result was 0
- **Sign:** `SF=1` if result was negative (*i.e.*, same as sign bit)
- **Overflow:** `OF=1` if result had *signed* overflow

Implicit Setting (Review)

- *Implicitly* set by **arithmetic** and **logical** operations as side effect
  - Notable exception: `lea` (beware! 🙄)

- Example: `%al=0x80`, execute **addb** `%al, %al`

compute: 0b 1000 0000
+ 0b 1000 0000
---
1 0000 0000

stores 0x00 in %al ⟹

CF = 1 (carry out from MSB)
ZF = 1 (== 0)
SF = 0 (non-negative)
OF = 1 (⊖ + ⊖ = ⊕)

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

Explicit Setting (Review)

- *Explicitly* set by **`cmp`** and **`test`**, whose purposes are to change the condition codes *without storing their results*
  - `cmp` is equivalent result to `sub` (`-`); `test` is equivalent result to `and` (`&`)

- Example: `%al=0x83` and `%bl=0x8C`, execute **`testb`** `%al, %bl`

compute: 0b 1000 0011
& 0b 1000 1100
1000 0000

result is 0x80,
but %bl is unchanged!

⇒

CF = 0 (no carry out)
ZF = 0 (!= 0)
SF = 1 (MSB is 1)
OF = 0 (no overflow)

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

Lecture Outline (3/4)

- Control Flow
- Condition Codes
- **Conditionals**
- Instruction Set Philosophies, Revisited

Jump and Set Families (Review)

- `j* target`
  - Jumps to ***target*** (an address) based on condition codes

- `set* dst`
  - Set low-order byte of `dst` to 0x00 or 0x01 based on condition codes and does not alter remaining 7 bytes

- Typically come as the second in a pair of instructions:
  1) Set the condition codes
  2) Uses the condition codes to do something

Instruction Condition Table (Review)

- `j*/set*` Instructions – focus on description, not formula

don't worry about the details

Jump Instr	Set Instr	Condition	Description
jmp *target*	*n/a*	1	Unconditional
je *target*	**sete** *dst*	ZF	Equal to 0
jne *target*	**setne** *dst*	~ZF	Not Equal to 0
js *target*	**sets** *dst*	SF	Signed/Negative
jns *target*	**setns** *dst*	~SF	Not Signed/Nonnegative
jg *target*	**setg** *dst*	~(SF^OF)&~ZF	Greater Than 0 (Signed)
jge *target*	**setge** *dst*	~(SF^OF)	Greater Than or Equal to 0 (Signed)
jl *target*	**setl** *dst*	(SF^OF)	Less Than 0 (Signed)
jle *target*	**setle** *dst*	(SF^OF)|ZF	Less Than or Equal to 0 (Signed)
ja *target*	**seta** *dst*	~CF&~ZF	Above 0 (unsigned “>”)
jb *target*	**setb** *dst*	CF	Below 0 (unsigned “<“)

Choosing Instruction Pairs

- Find correct form of condition in table, use corresponding instrs
  - Instruction #1 = chosen operation, Instruction #2 = row heading
  - Recall: `cmp` performs `sub`
    `test` performs `and`

- Examples:
  - `x >= y` →
    ```
    x - y >= 0
    ```
    →
    ```
    cmp y, x
    jge target
    ```
  - `x == 3` →
    ```
    x - 3 == 0
    ```
    ```
    cmp 3, x
    je target
    ```
  - `x` →
    ```
    x & x != 0
    ```
    ```
    test x, x
    jne target
    ```

		(op) s, d
je /sete	"Equal"	d (op) s == 0
jne/setne	"Not equal"	d (op) s != 0
js /sets	"Signed" (negative)	d (op) s < 0
jns/setns	"Not signed" (nonnegative)	d (op) s >= 0
jg /setg	"Greater"	d (op) s > 0
jge/setge	"Greater or equal"	d (op) s >= 0
jl /setl	"Less"	d (op) s < 0
jle/setle	"Less or equal"	d (op) s <= 0
ja /seta	"Above" (unsigned >)	d (op) s > 0U
jb /setb	"Below" (unsigned <)	d (op) s < 0U

Set Example

```
int gt(long x, long y) {
  return x > y;
}
```

x - y > 0

Variable	Register
x	%rdi
y	%rsi
return value	%rax

cmp y, x → cmp %rsi, %rdi

set g ___ → setg %al

```
cmpq    %rsi, %rdi   # set flags on x - y
setg    %al          # %al = (x-y>0)
movzbl  %al, %eax    # %rax = (x>y)
ret
```

- %al is lowest byte of %rax
- Uses movz to finish job, since set* only changes lowest byte of register

Labels (Review)

```
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

Jump Example (Writing)

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

do this if x ≥ 3

Variable	Register
x	%rdi
return value	%rax

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

(if)

(else)

		① cmp a, b (3, x)	test a, b
je	"Equal"	b-a == 0	b&a == 0
jne	"Not equal"	b-a != 0	b&a != 0
js	"Signed" (negative)	b-a < 0	b&a < 0
jns	"Not signed" (nonnegative)	b-a >= 0	b&a >= 0
jg	"Greater"	b-a > 0	b&a > 0
② **jge**	"Greater or equal"	b-a >= 0 (x ≥ 3)	b&a >= 0
jl	"Less"	b-a < 0	b&a < 0
jle	"Less or equal"	b-a < 0	b&a <= 0
ja	"Above" (unsigned >)	b $>_U$ a	b&a > 0U
jb	"Below" (unsigned <)	b $<_U$ a	b&a < 0U

Jump Example (Reading)

```
mystery:
        movl    %edi, %eax      # ret = x
        testl   %edi, %edi      # x & x
        js      .L3             # jump if x < 0
.L2:    ret
.L3:    negl    %eax            # negate x
        jmp     .L2
```

Variable	Register
x	%edi
return value	%eax

```
int mystery(int x) {
  if ( x < 0 ) {
    return -x;
  } else {
    return x;
  }
}
```

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

Polling Question

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

```
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)

Lecture Outline (4/4)

- Control Flow
- Condition Codes
- Conditionals
- **Instruction Set Philosophies, Revisited**

Instruction Set Philosophies, Revisited

- *Complex Instruction Set Computing* (CISC):
  Add more and more elaborate and specialized instructions as needed
  - **Design goals:** complete tasks in as few instructions as possible; minimize memory accesses for instructions

- *Reduced Instruction Set Computing* (RISC):
  Keep instruction set small and regular
  - **Design goals:** build fast hardware; instructions should complete in few clock cycles (ideally 1); minimize complexity and maximize performance

- How different are these two philosophies, really?

Instruction Set Philosophies, Revisited

- *Complex Instruction Set Computing* (CISC):
  Add more and more elaborate and specialized instructions as needed
  - **Design goals:** complete tasks in as few instructions as possible; minimize memory accesses for instructions

- *Reduced Instruction Set Computing* (RISC):
  Keep instruction set small and regular
  - **Design goals:** build fast hardware; instructions should complete in few clock cycles (ideally 1); minimize complexity and maximize performance

- How different are these two philosophies, really?
  - Both pursue **efficiency** (**minimalism** is a means to an end)

Mainstream ISAs, Revisited

x86	
Designer	Intel, AMD
Bits	16-bit, 32-bit and 64-bit
Introduced	1978 (16-bit), 1985 (32-bit), 2003 (64-bit)
Design	CISC
Type	Register–memory
Encoding	Variable (1 to 15 bytes)
Branching	Condition code
Endianness	Little

ARM	
Introduced	1985
Encoding	AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2)

RISC-V	
Designer	University of California, Berkeley
Bits	32 · 64 · 128
Introduced	2010
Design	RISC
Type	Load-store
Encoding	Variable
Endianness	Little[1][3]

Does anything feel "off" about this landscape?

Windows desktop/laptops
(Core i3, i5, i7, Ryzen)
x86-64 Instruction Set

Smartphone-like devices
(iPhone, Android, Raspberry Pi)
Apple products (ca. 2020-)
(Macbook, Mac Mini)
ARM Instruction Set

Mostly research
(some traction in embedded)
RISC-V Instruction Set

Tech Monopolization (Oligopolization)

- How many “dominant” ISAs are there?
  - 2: x86, ARM
- How many “dominant” phone brands are there?
  - 3: Samsung, Apple, Xiaomi
- How many “dominant” operating systems are there?
  - 3-ish: Windows, iOS/macOS, Android/Linux
- How many “dominant” chip manufacturers are there?
  - 3: TSMC, Samsung, Intel

- It wasn’t always this way!
  - Combination of antitrust policies and (lack of) enforcement

Discussion Questions

- Discuss the following question(s) in groups of 3-4 students
  - I will call on a few groups afterwards so please be prepared to share out
  - Be respectful of others' opinions and experiences

- How do you feel about tech oligopolization?
  - What are the benefits and disadvantages of this landscape for (1) the dominant companies and (2) the consumers?

  - These big tech companies are now worth billions of dollars. What might we try if we wanted to break up the oligopolization?

Summary (1/2)

- 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 & logical instructions (implicit) or Compare/Test (explicit)
  - **Set instructions** (`set*`) read out flag values (0/1)
  - **Jump instructions** (`j*`) use flag values to determine next instruction to execute
  - Result of 1st instruction gets compared against 0 in a way determined by 2nd instruction:

			`(op) s, d`	`cmp a, b`	`test a, b`
je	**sete**	"Equal"	`d (op) s == 0`	`b-a == 0`	`b&a == 0`
jne	**setne**	"Not equal"	`d (op) s != 0`	`b-a != 0`	`b&a != 0`
js	**sets**	"Signed" (negative)	`d (op) s < 0`	`b-a < 0`	`b&a < 0`
jns	**setns**	"Not signed" (nonnegative)	`d (op) s >= 0`	`b-a >= 0`	`b&a >= 0`
jg	**setg**	"Greater"	`d (op) s > 0`	`b-a > 0`	`b&a > 0`
jge	**setge**	"Greater or equal"	`d (op) s >= 0`	`b-a >= 0`	`b&a >= 0`
jl	**setl**	"Less"	`d (op) s < 0`	`b-a < 0`	`b&a < 0`
jle	**setle**	"Less or equal"	`d (op) s <= 0`	`b-a < 0`	`b&a <= 0`
ja	**seta**	"Above" (unsigned >)	`d (op) s > 0U`	$b >_U a$	`b&a > 0U`
jb	**setb**	"Below" (unsigned <)	`d (op) s < 0U`	$b <_U a$	`b&a < 0U`

Summary (2/2)

- **Labels** (*e.g.*, `main`, `.L0`) refer to an instruction address and used as jump targets in assembly

Bonus: Compound Conditional Example

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

```
  cmpq  $2, %rdi
  setle %dl
  cmpq  %rsi, %rdi
  sete  %al
  testb %al, %dl
  je    T2
T1: # x <= 2 && x == y:
  movl  $1, %eax
  ret
T2: # else
  movl  $2, %eax
  ret
```

Variable	Register
x	%rdi
y	%rsi
return value	%rax

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