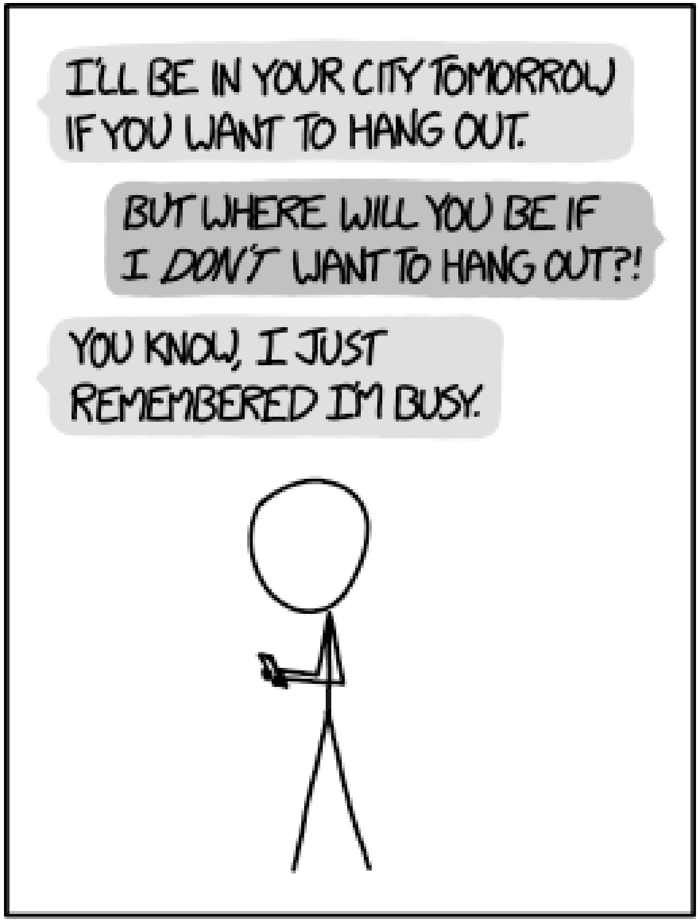
x86-64 Programming III

CSE 351 Autumn 2024

Instructor:
Ruth Anderson

Teaching Assistants:
Alexandra Michael
Connie Chen
Chloe Fong
Chendur Jayavelu
Joshua Tan
Nikolas McNamee
Nahush Shrivatsa
Naama Amiel
Neela Kausik
Renee Ruan
Rubee Zhao
Samantha Dreussi
Sean Siddens
Waleed Yagoub



WHY I TRY NOT TO BE PEDANTIC ABOUT CONDITIONALS.

http://xkcd.com/1652/

Relevant Course Information

- HW7 due TONIGHT, Monday (10/14) @ 11:59 pm
- Lab 1b, due TONIGHT, Monday (10/14) @ 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
- HW8 due Wednesday (10/16) @ 11:59 pm
- Lab 2 (x86-64) due next Friday (10/25)
  - coming soon!
  - 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`)

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

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` 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

0x00	0x00	0x7F	0xFF	0xC6	0x1F	0xA4	0xE8	←%rax

...	0x??	0x??	0x80	0x??	0x??	0x??	...	← MEM

0x00	0x00	0x00	0x00	0xFF	0xFF	0xFF	0x80	←%rbx

Review Question

- If `%rsi` is `0x B0BACAFE 1EE7 F0 0D`, what is its value after executing:

`movswl %si, %esi`?

x86 Control Flow

- **Condition codes**
- **Conditional and unconditional branches**
- Loops
- Switches

Conditionals and Control Flow

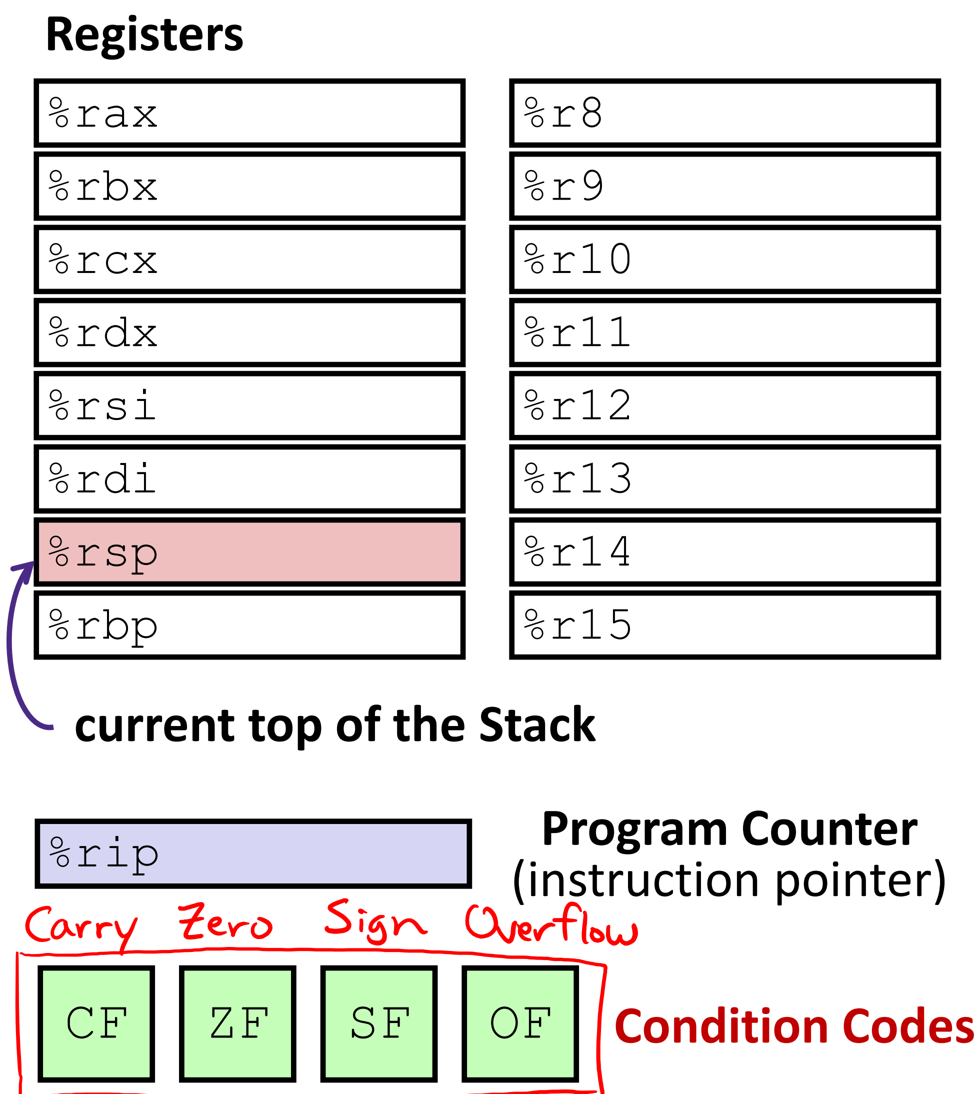
- Conditional branch/*jump*
  - Jump to somewhere else <u>if some condition is true</u>, otherwise execute next instruction
- Unconditional branch/*jump*
  - <u>Always</u> jump when you get to this instruction

- 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`** {…}

How to check conditions and implement?!

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

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

- Carry Flag (CF): Set to 1 if most recent op result *carried out* data which couldn't be stored; i.e. unsigned overflow (usually when *dest* < *src*)
- Zero Flag (ZF): Set to 1 if most recent op result computed to `0`
- Sign Flag (SF): Set to 1 if most recent op result is a signed (negative) value i.e. MSB produced is 1
- Overflow Flag (OF): Similar as carry flag, but applies to signed overflow i.e. if MSBs were both `0`, and now result MSB is 1, or vice versa

Use flag values to decide whether to jump: How do we set them?

Setting Condition Codes: Implicit Setting

- Implicitly set by **arithmetic** operations: a fun side effect!
  - 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

Suppose we only want to set the flags but not save the result…

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

Setting Condition Codes: Explicit Setting with `cmp`

- Explicitly set by the **Compare** instruction
  - **`cmpq`** `src1, src2` ↔ sets flags based on `b-a`

    Kind of like **`subq`** `a, b` but doesn't store result anywhere

  - **`CF=1`** if carry out from MSB (good for *unsigned* comparison)
  - **`ZF=1`** if `a==b`, because `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*

Setting Condition Codes: Explicit Setting with `test`

- Explicitly set by **Test** instruction
  - **`testq`** `src2, src1` ↔ sets flags based on `a&b`

    Kind of like **`andq`** `a, b` but doesn't store result anywhere

    Tip: Useful to have one of the operands be a mask

  - Can't have carry out (**`CF`**) or overflow (**`OF`**)—why?
  - **`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
```

CF: ZF: SF: OF:

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 “<“)

Note: We can't define our own `j*` instructions. But we can simulate customizing one by “saving” flags for a compound conditional, and jumping based on that…

Using Condition Codes: Setting

False → 0b 0000 0000 = 0x 00
True → 0b 0000 0001 = 0x01

- `set*` Instructions
  - Set low-order byte of `dst` to 0 or 1 based on condition codes
  - Does not alter remaining 7 bytes

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:

%rax	%al	%r8	%r8b
%rbx	%bl	%r9	%r9b
%rcx	%cl	%r10	%r10b
%rdx	%dl	%r11	%r11b
%rsi	%sil	%r12	%r12b
%rdi	%dil	%r13	%r13b
%rsp	%spl	%r14	%r14b
%rbp	%bpl	%r15	%r15b

8B

1B

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 the 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`	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 the 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
```

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*: ***first set flags, then jump!***

```
      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
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

Register	Use(s)
`%rdi`	argument `x`
`%rsi`	argument `y`
`%rax`	return value

		`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`

```
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
```

Control Flow

Register	Use(s)
`%rdi`	1st argument (`x`)
`%rsi`	2nd 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 TRUE
  if x <= y then jump to else
            if FALSE
  movq   %rdi, %rax
  jump to done
else:
  movq   %rsi, %rax
done:
  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;
}
```

```
absdiff:

   ____________________

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

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…

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

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