

CSCI 2330



TODO

- Assignment 2 assigned due October 7

Overview for today

- Introduction to machine code

Structure of course

- First third (Exam 1):
 - Data Representation
 - Memory
- Second third (Exam 2):
 - Machine Code ← here
 - Memory exploits
- Final third (Final exam):
 - Memory hierarchy
 - Processes

Computer systems

- We learn how computers *actually* work
 - Not how they're *supposed* to work
 - Not how we *want* them to work
- We saw that with integers
- We saw that with floats
- We will see that again with code itself

Computer systems

- We will see how this gives us a tremendous amount of power
- We will write realistic software exploits that take control of machines' execution

Programming language hierarchy

- Java, Python, Rust, etc.
- C
- Machine code

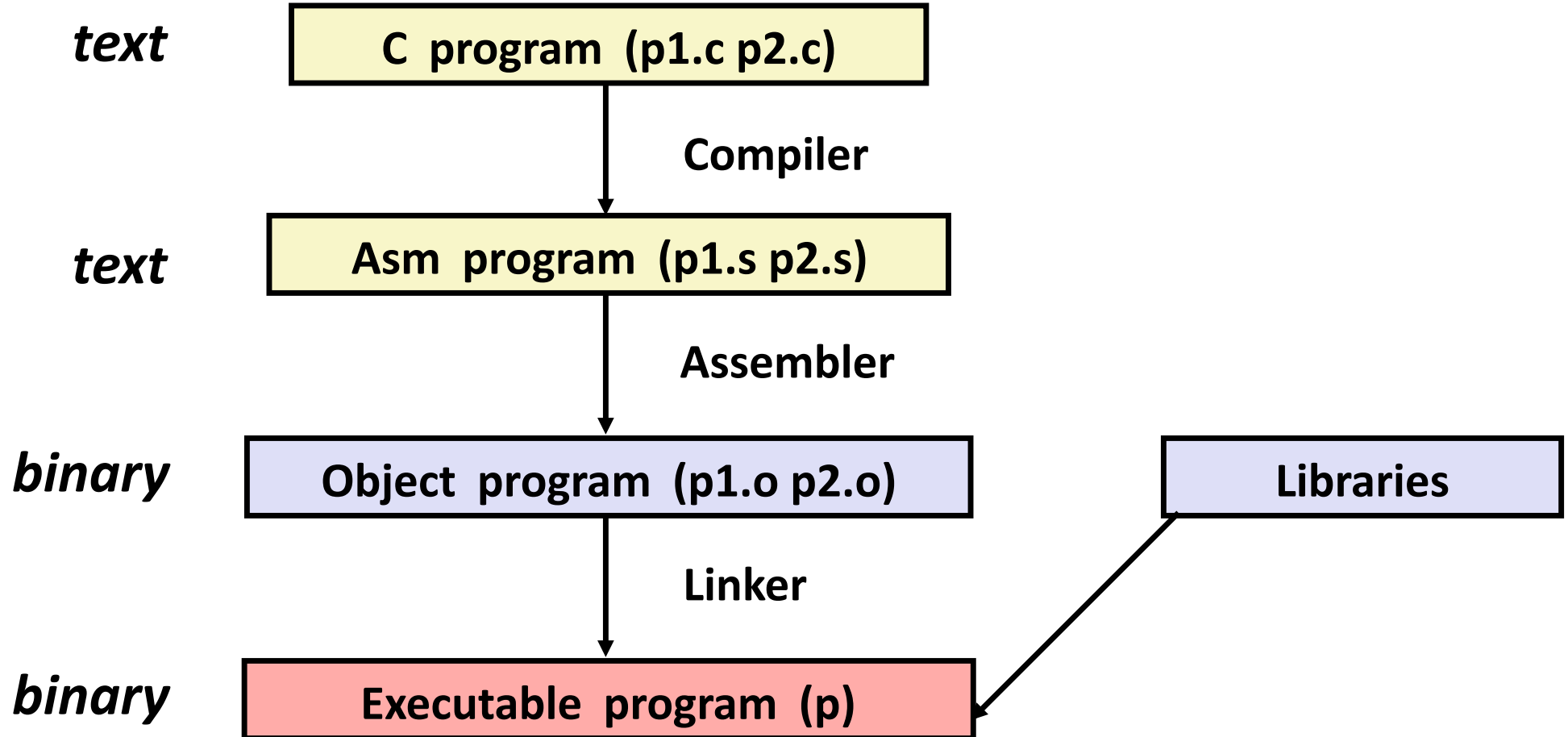
How are they related?

- Java, Python, Rust, etc. (*written in C*)
- C (*compiled to machine code*)
- Machine code

So what is machine code?

- Lowest level instructions that a computer can execute directly

How do we get machine code?



Compilation stages

- *The compiler compiles* each *.c file to *assembly* (a *.s file)
- Assembly – a human readable version of machine code

Compilation stages

- The *assembler assembles* the assembly code into machine code (a *.o object file)
- Object file – a simple container for machine code

Compilation stages

- *The linker links* the object files into an *executable*
- Executable – the thing you can actually run

Compilation stages

- gcc typically hides many of these steps
- gcc can do them all in one step!

hello.c

So that's where machine code
comes from

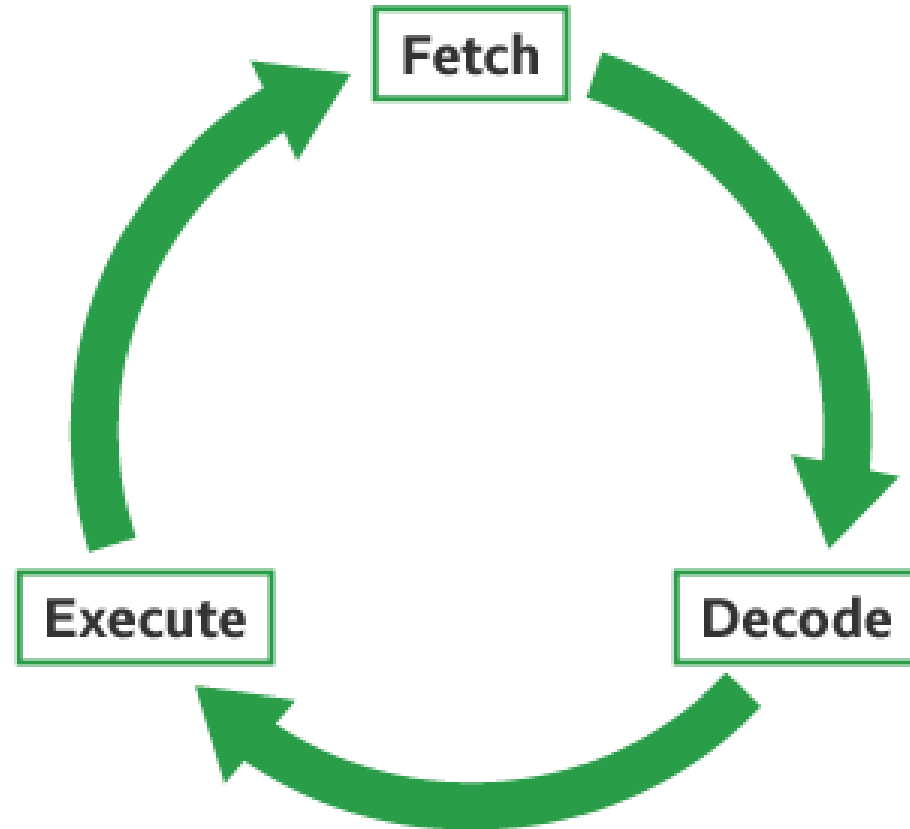


Machine code execution

- So that's where machine code comes from
- How do we actually execute it?



This is how computers work

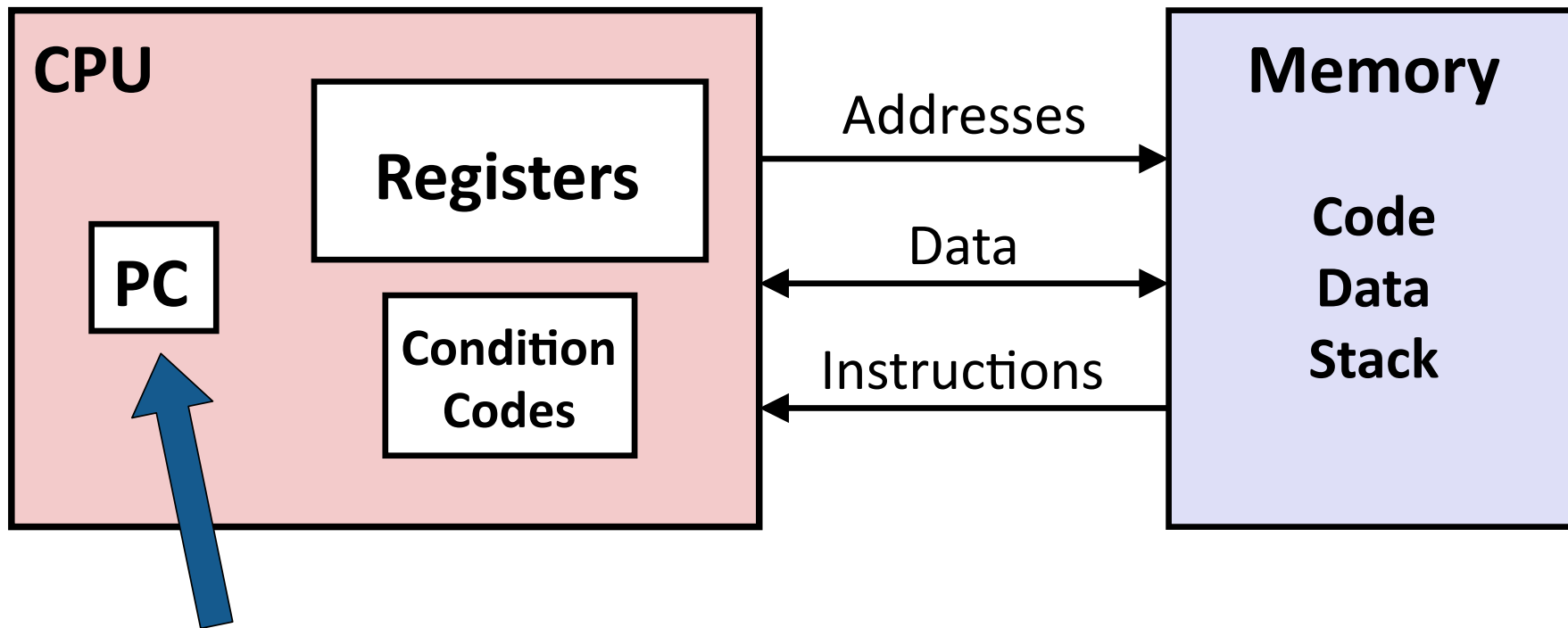


Fetch-decode-execute loop

- Fetch: fetch an instruction from memory
- Decode: decode instruction
- Execute: execute instruction
- Infinite loop!
- The rest is: what do those instructions do?

Anatomy of a computer

(for purposes of executing machine code)



Program Counter:
address of next instruction

CPU components

- Program counter (PC): the address of the next instruction to execute (a.k.a. *instruction pointer*)
- PC determines by its address which instruction to fetch next

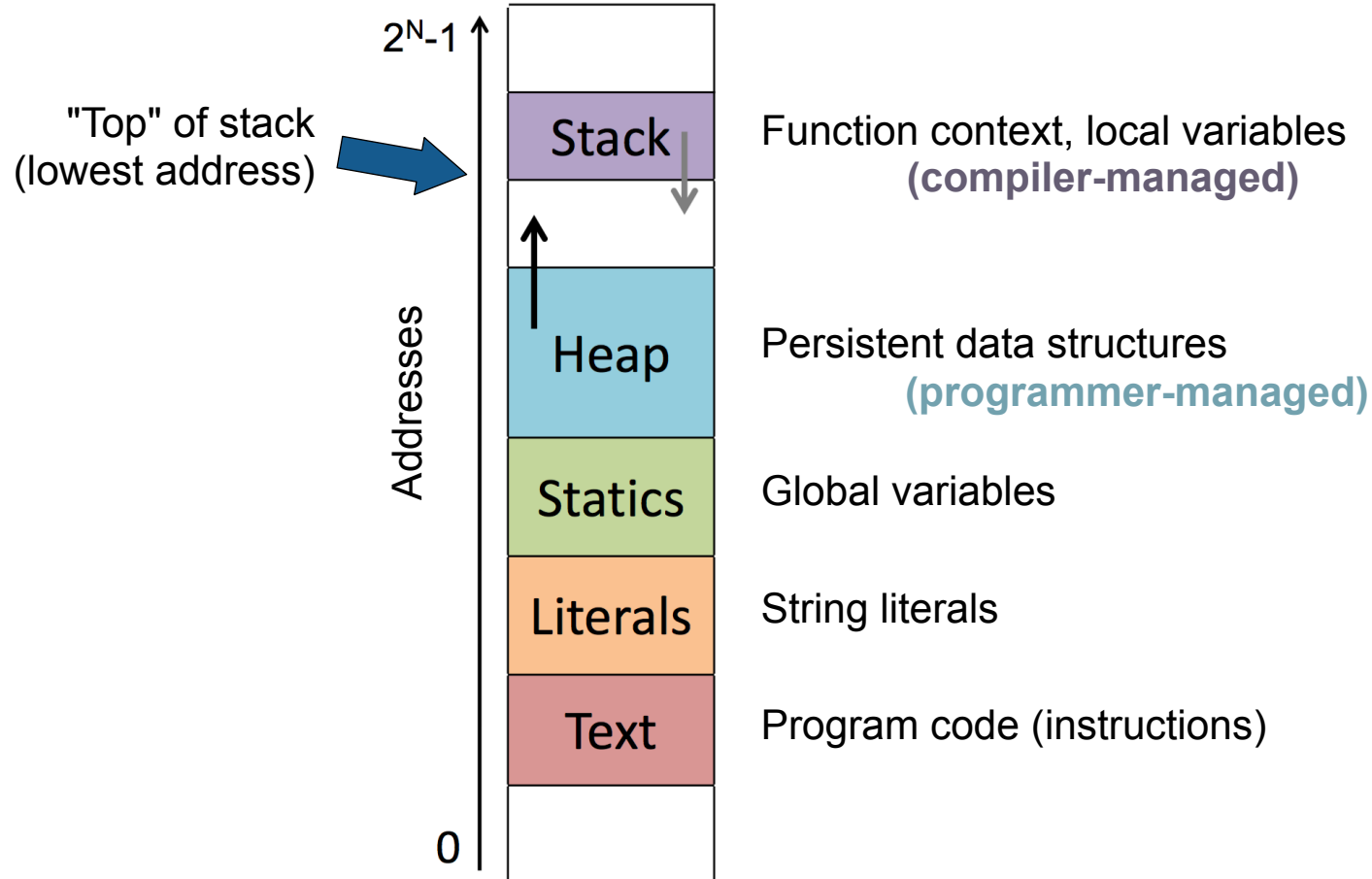
CPU components

- Registers: holds the data that the CPU directly acts on
- Registers are word-sized
- We usually copy data from memory to a register before operating on it
- CPUs typically have between 4 and 50 registers

CPU components

- Condition codes: status information about the most recently completed operation
- Useful for conditional control flow!

Memory contents



Memory contents

- Spent a lot of time talking about the **stack** and the **heap**
- Now we will look at **text** and even more closely at the **stack**
 - PC points into text
 - Special CPU instructions for managing stack
 - (Heap is managed entirely in software)

Von Neumann architecture

- Is it code? Is it data?
- It's a matter of interpretation...



Von Neumann architecture

- Is it code? Is it data?
- It's a matter of interpretation...
- PC pointing to it? code!
- Now you're trying to loading it into a register? data!



Instruction Set Architecture (ISA)

- Defines your word size
- Defines your registers
- Defines your set of instructions
- Examples: x86, x86-64, ARM32, ARM64

Word size

- Previously said it determined the size of an address
- Now we will see how it is actually the native data type of the machine
- It is the typical register size

x86-64 and our class

- We will be primarily studying x86-64 in this class
- krebs runs it
- If you're using a Bowdoin Macbook, it's running ARM!

x86 history

- 1978: Intel 8086 processor (16-bit word size)
- 1985: Intel 386 processor (32-bit word size)
- 2003: AMD Opteron processor (64-bit word size)
- Modern x86 processors can still run 8086 code!
- x86 has a lot of backwards compatibility

Instruction Set Architectures (ISAs)

- We said:
- Defines your word size
- Defines your registers
- Defines your set of instructions
- How is x86-64 defined?

x86-64

- Word size: 64-bit

x86-64

- Registers?
- 16 general purpose 64-bit integer registers

x86-64 registers

	%rax	%r8
	%rbx	%r9
	%rcx	%r10
	%rdx	%r11
	%rsi	%r12
	%rdi	%r13
Stack Pointer	%rsp	%r14
Frame/Base Pointer	%rbp	%r15
Program Counter (Instruction Pointer)	%rip	

Special registers

- `%rsp` – points to “top” of stack (lowest address)
 - Some instructions work specially with it
 - Your operating system may also read it to determine your current stack size
- `%rbp` – often points to base of current stack frame
 - This is purely out of convention (no special instructions, etc.)

Special registers

- `%rip` – the name of the program counter on x86-64
 - Stands for *register instruction pointer*
 - **NOT** one of the 16 general purpose registers
 - Cannot be used for anything but as a program counter

x86-64

- Now that we have defined the word size and registers, we “only” need to define the instructions!
- This will be the hard part!

Three categories of instructions

1. Data movement – moving data between registers and memory
 - E.g., load data from memory address 0x20 into register %rax
2. Arithmetic – perform arithmetic on data in registers or in memory
 - E.g., take the value in %rax and add it to the value in %rbx
3. Control flow – choosing next instruction
 - Calling a function
 - Conditional branches, loops, etc.

Data movement

- Moving data between memory and/or registers
- Basic instruction:
- `movq src, dest`
- Copies from source to destination
- Note the order of the operands! ⚠
- In C, `dest = src`, but here we say `movq src, dst`

MOV examples

copies the contents of rax into rbx (i.e., changes rbx)

movq %rax, %rbx

vice versa: copies the contents of rbx into rax (i.e., changes rax)

movq %rbx, %rax

Instruction operand types

- Named register (%rax, %rbx, etc.)
 - As in previous slide
- Immediate data
 - Called this because the data is encoded into the instruction itself (note the \$)

`movq $5, %rax` # loads the value 5 into %rax

`movq $0x10, %rax` # loads 16 (or 0x10) into %rax

`movq $-12, %rcx` # loads -12 into %rcx

Instruction operand types

- Memory

- By hard-coded address (note: **no \$**):

`movq %rax, 20` # copy from %rax into memory at address 20

`movq 0x50, %rbx` # copy from memory at address 0x50 into %rbx

- By register (e.g., dereferencing a pointer):

`movq (%rax), %rbx` # copy from memory at address %rax into %rbx

`movq %rcx, (%rdx)` # copy from %rcx into memory at address %rdx

Instruction operand types

- Memory

- By hard-coded address (note: **no \$**):

`movq %rax, 20` # copy from %rax into memory at address 20

`movq 0x50, %rbx` # copy from memory at address 0x50 into %rbx

- By register (e.g., dereferencing a pointer):

`movq (%rax), %rbx` # C equivalent: `rbx = *rax`

`movq %rcx, (%rdx)` # copy from %rcx into memory at address %rdx

Instruction operand types

- Memory

- By hard-coded address (note: **no \$**):

`movq %rax, 20` # copy from %rax into memory at address 20

`movq 0x50, %rbx` # copy from memory at address 0x50 into %rbx

- By register (e.g., dereferencing a pointer):

`movq (%rax), %rbx` # C equivalent: `rbx = *rax`

`movq %rcx, (%rdx)` # C equivalent: ???

Instruction operand types

- Memory

- By hard-coded address (note: **no \$**):

`movq %rax, 20` # copy from %rax into memory at address 20

`movq 0x50, %rbx` # copy from memory at address 0x50 into %rbx

- By register (e.g., dereferencing a pointer):

`movq (%rax), %rbx` # C equivalent: `*rax = rbx`

`movq %rcx, (%rdx)` # C equivalent: `*rdx = rcx`

Allowed MOV operand combinations

- **movq immediate, register**
- **movq immediate, memory**
- **movq register, register**
- **movq register, memory**
- **movq memory, register**
- What is missing?

Cannot move anything to immediate

- What would that even mean?
- `movq ..., $2` ❌
- C equivalent:
- `2 = ...` ❌

Cannot move memory to memory

- `movq (%rax), (%rdx)` ✗
- `movq (%rax), 0xdeadbeef` ✗
- `movq 0x12345678, (%rdx)` ✗
- `movq 0x12345678, 0xdeadbeef` ✗
- These instructions might make sense conceptually, but x86 does not allow them due to architectural limitations

Cannot move memory to memory

- `movq (%rax), (%rdx)` ✗
- `movq (%rax), 0xdeadbeef` ✗
- `movq 0x12345678, (%rdx)` ✗
- `movq 0x12345678, 0xdeadbeef` ✗
- So how do we move from (%rax) to (%rdx)?
 - Perhaps using multiple instructions?

Cannot move memory to memory

- `movq (%rax), (%rdx)` ❌
- `movq (%rax), 0xdeadbeef` ❌
- `movq 0x12345678, (%rdx)` ❌
- `movq 0x12345678, 0xdeadbeef` ❌
- So how do we move from (%rax) to (%rdx)?
- `movq (%rax), %rcx` ✅
- `movq %rcx, (%rdx)` ✅

swap.c

swap.c

- Note: not all variables are stored in memory
- Some only exist in registers!

MOV Exercises

One more wrinkle...

- What does the **q** in `movq` mean?
- It's an *instruction suffix*
- b = byte (8 bit / 1 byte)
- w = word (16 bit / 2 bytes)
- l = long / doubleword (32 bit / 4 bytes)
- q = quadword (64 bit / 8 bytes)

One more wrinkle...

- What does the **q** in **movq** mean?
- It's an *instruction suffix*
- b = byte (8 bit / 1 byte)
- w = word (16 bit / 2 bytes) (!?!?)
- l = long / doubleword (32 bit / 4 bytes)
- q = quadword (64 bit / 8 bytes)

x86 history

- 1978: Intel 8086 processor (16-bit word size)
- 1985: Intel 386 processor (32-bit word size)
- 2003: AMD Opteron processor (64-bit word size)
- Modern x86 processors can still run 8086 code!
- x86 has a lot of backwards compatibility

Instruction suffixes

- These suffixes allow us to work with smaller word sizes!
- Now we can work with ints, shorts, etc.
- But we need smaller registers first...

x86 virtual registers

- Each general purpose register has variants
 - 64-bit (**%rax**, the physical register)
 - 32-bit (**%rax** → **%eax**)
 - 16-bit (**%rax** → **%ax**)
 - 8-bit (**%rax** → **%al**)
- Naming is, of course, inconsistent...

64-Bit Register	Lowest 32 Bits	Lowest 16 Bits	Lowest 8 Bits
%rax	%eax	%ax	%al
%rbx	%ebx	%bx	%bl
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%rbp	%ebp	%bp	%bpl
%rsp	%esp	%sp	%spl
%r8	%r8d	%r8w	%r8b
%r9	%r9d	%r9w	%r9b
%r10	%r10d	%r10w	%r10b
%r11	%r11d	%r11w	%r11b
%r12	%r12d	%r12w	%r12b
%r13	%r13d	%r13w	%r13b
%r14	%r14d	%r14w	%r14b
%r15	%r15d	%r15w	%r15b

x86 instruction suffixes

- So, for example:
- `int *rcx = ...;`
- `int rax = *rcx;` → `movl (%rcx), %eax`
- `short *rcx = ...;`
- `short rax = *rcx;` → `movw (%rcx), %ax`
- `char *rcx = ...;`
- `char rax = *rcx;` → `movb (%rcx), %al`

What happens to other bits?

- **movl (%rcx), %eax**
 - Upper 32 bits of %rax are what now?
- **movw (%rcx), %ax**
 - Upper 48 bits of %rax are what now?
- **movb (%rcx), %al**
 - Upper 56 bits of %rax are what now?

What happens to other bits?

- **movl (%rcx), %eax**
 - Upper 32 bits of %rax are what now? Zero
- **movw (%rcx), %ax**
 - Upper 48 bits of %rax are what now? Unchanged
- **movb (%rcx), %al**
 - Upper 56 bits of %rax are what now? Unchanged

General principle

- Always use 64-bit *memory* operands on a 64-bit machine
- Change the size of the *register* operands when changing the instruction suffix
- Using the wrong register size is illegal

TODO

- Assignment 2 assigned due October 7