Computer Systems

CS107

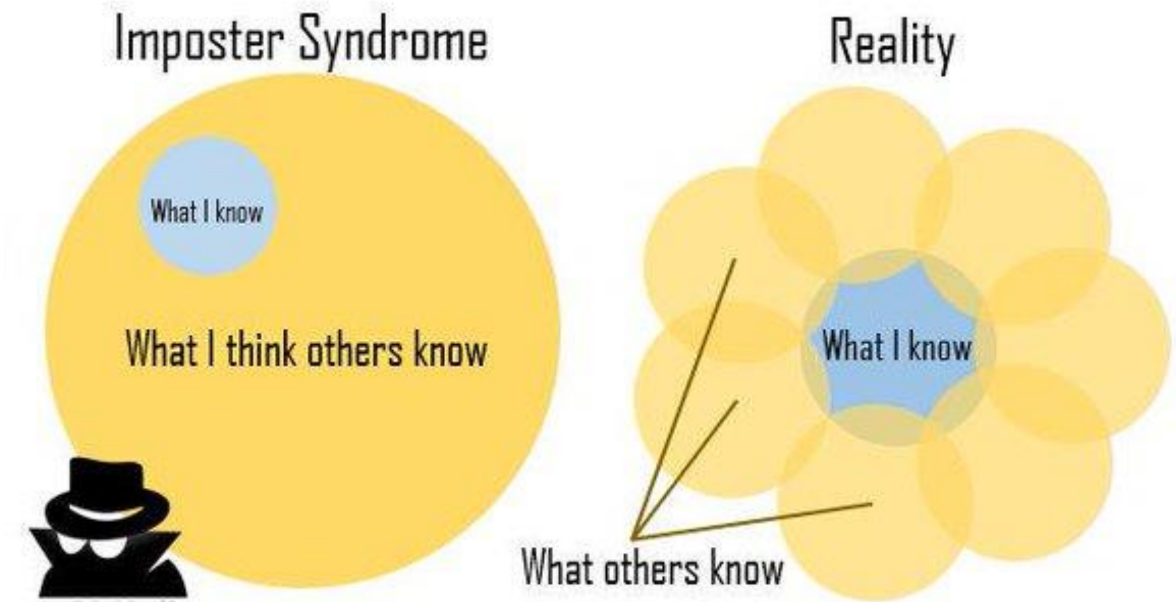
Cynthia Lee

Today's Topics

LECTURE:

- › Pop quiz on floating point!
  - Just kidding.
- › New: Assembly code

Two friendly reminders:



REMINDER:
Everything is bits!

Everything is bits!

- We've seen many data types so far:
  - Integers:
    - char/short/int/long (encoding as unsigned or two's complement signed)
  - Letters/punctutation/etc:
    - char (ASCII encoding)
  - Real numbers:
    - float/double (IEEE floating point encoding)
  - Memory addresses:
    - pointer types (unsigned long encoding)
  - **Now a new one…..the code itself!**
    - **Instructions** (AMD64 encoding)

What happens when we compile our code?

ANATOMY OF AN EXECUTABLE FILE

What happens when we compile our code?

```
int sum_array(int arr[], int nelems) {
    int sum = 0;
    for (int i = 0; i < nelems; i++) {
        sum += arr[i];
    }
    return sum;
}
```

```
> make
> ls
Makefile  sum  sum.c
> objdump -d sum
```

```
000000000040055d <sum_array>:
  40055d:       ba 00 00 00 00          mov    $0x0,%edx
  400562:       b8 00 00 00 00          mov    $0x0,%eax
  400567:       eb 09                   jmp    400572
  400569:       48 63 ca                movslq %edx,%rcx
  40056c:       03 04 8f                add    (%rdi,%rcx,4),%eax
  40056f:       83 c2 01                add    $0x1,%edx
  400572:       39 f2                   cmp    %esi,%edx
  400574:       7c f3                   jl     400569
  400576:       f3 c3                   repz retq
```

```
0000000000040055d <sum_array>:
  40055d:        ba 00 00 00 00            mov     $0x0,%edx
  400562:        b8 00 00 00 00            mov     $0x0,%eax
  400567                                           400572
  400569                                       .q  %edx,%rcx
  40056c                                           (%rdi,%rcx,4),%eax
  40056f                                           $0x1,%edx
  400572                                           %esi,%edx
  400574:        7c f3                     jl      400569
  400576:        f3 c3                     repz retq
```

Name of the function (same as in the C code) and the memory address where the code for this function starts

Stanford University

```
000000000040055d <sum_array>:
  40055d:    ba 00 00 00 00      mov    $0x0,%edx
  400562:    b8 00 00 00 00      mov    $0x0,%eax
  400567:    eb 09               jmp    400572
  400569:                        vslq   %edx,%rcx
  40056c:                        d      (%rdi,%rcx,4),%eax
  40056f:                        d      $0x1,%edx
  400572:                        p      %esi,%edx
  400574:                               400569
  400576:                        pz retq
```

Memory address where each of line of instruction is found—sequential instructions are found sequentially in memory

Stanford University

```
0000000000040055d <sum_array>:
  40055d:       ba 00 00 00 00        mov    $0x0,%edx
  400562:       b8 00 00 00 00        mov    $0x0,%eax
  400567:       eb 09                 jmp    400572
  400569:       48 63 ca              movslq %edx,%rcx
  40056c:       03 04 8f              add    (%rdi,%rcx,4),%eax
  40056f:                             add    $0x1,%edx
  400572:                             cmp    %esi,%edx
  400574:                             jl     400569
  400576:                             repz retq
```

Assembly code: "human-readable" version of each instruction

```
0000000000040055d <sum_array>:
  40055d:      ba 00 00 00 00        mov    $0x0,%edx
  400562:      b8 00 00 00 00        mov    $0x0,%eax
  400567:      eb 09                 jmp    400572
  400569:      48 63 ca
  40056c:      03 04 8f                         4),%eax
  40056f:      83 c2 01
  400572:      39 f2
  400574:      7c f3
  400576:      f3 c3
```

Machine code: raw hexadecimal version of each instruction, representing the binary as it would be read by the computer

Anatomy of an individual instruction

Anatomy of an individual instruction

```
40056c:    03 04 8f    add    (%rdi,%rcx,4),%eax
40056f:    83 c2 01    add    $0x1,%edx
400572:    39 f2       cmp    %esi,%edx
400574:    7c f3       jl     400569
```

Operation name
(sometimes called
“opcode”)

Operands
(like arguments)

Anatomy of an individual instruction

```
40056c:    03 04 8f    add    (%rdi,%rcx,4),%eax
40056f:    83 c2 01    add    $0x1,%edx
400572:    39 f2       cmp    %esi,%edx
400574:    7c f3       jl     400569
```

%[name] names a register—these are a small collection of memory slots right on the CPU that can hold variables' values

Anatomy of an individual instruction

```
40056c:    03 04 8f    add   (%rdi,%rcx,4),%eax
40056f:    83 c2 01    add   $0x1,%edx
400572:    39 f2       cmp   %esi,%edx
400574:    7c f3       jl    400569
```

$[number] means a constant value (this is the number 1)

Stanford University

```
000000000040055d <sum_array>:
  40055d:     ba 00 00 00 00        mov    $0x0,%edx
  400562:     b8 00 00 00 00        mov    $0x0,%eax
  400567:     eb 09                 jmp    400572
  400569:     48 63 ca              movslq %edx,%rcx
  40056c:     03 04 8f              add    (%rdi,%rcx,4),%eax
  40056f:     83 c2 01              add    $0x1,%edx
  400572:     39 f2                 cmp    %esi,%edx
  400574:     7c f3                 jl     400569
  400576:     f3 c3                 repz retq
```

adding 1 ++

Guessing game: which part of the sum_array C code do you think corresponds to the marked assembly instruction?

```
int sum_array(int arr[], int nelems) {
    int sum = 0;                                // (A)
    for (int i = 0; i < nelems; i++) {          // (B)
        sum += arr[i];                          // (C)
    }                                           // or (D) other?
    return sum;
}
```

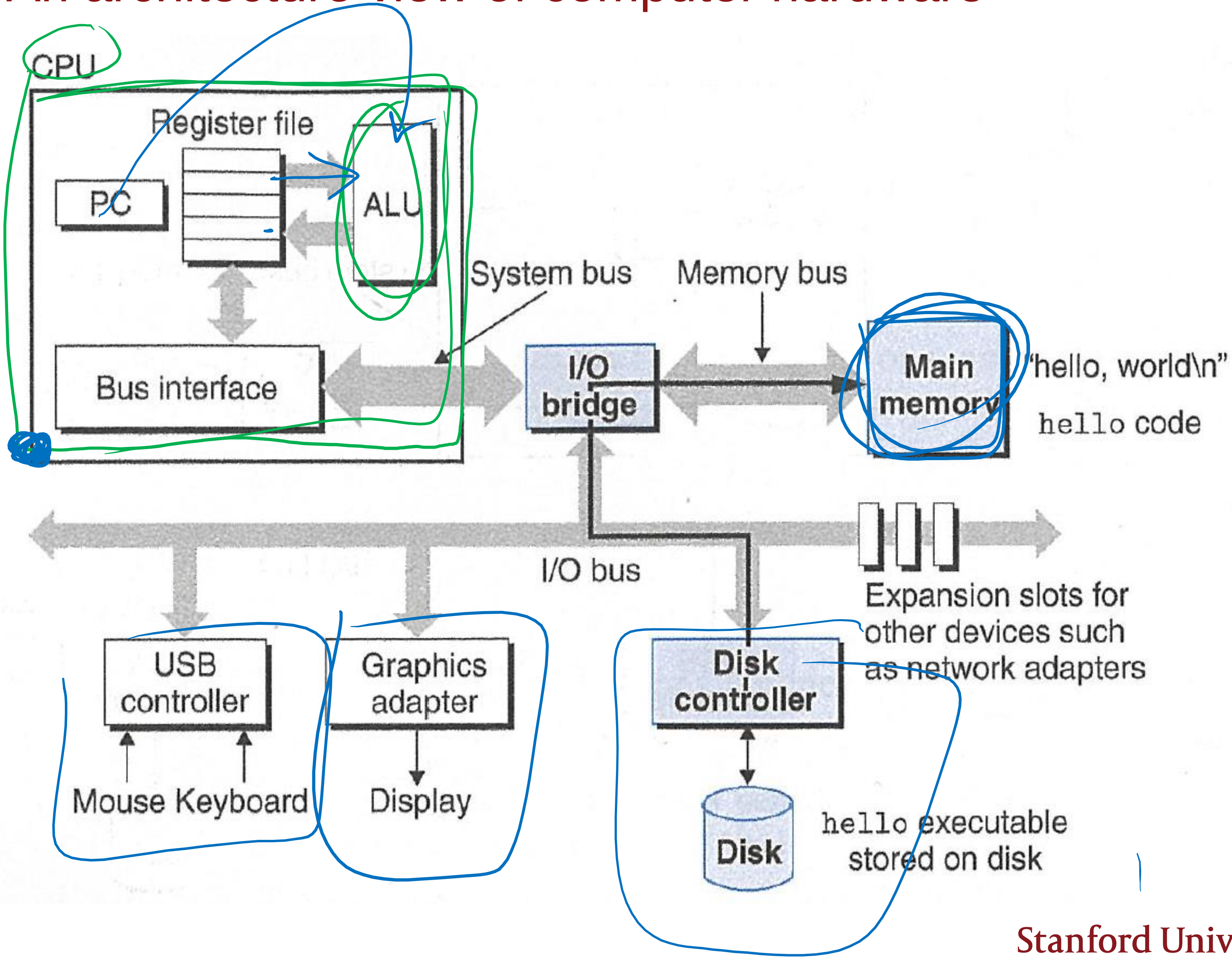

Stanford University

Registers and memory

ANATOMY OF THE COMPUTER

Stanford University

An architecture view of computer hardware



The mov instruction

Our first instruction

Dude, where's my data?

- **A main job of assembly language is to manage data:**
  - Data can be on the CPU (in registers) or in memory (at an address)
    - Turns out this distinction REALLY MATTERS for performance
    - https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html
  - Instructions often want to move data:
    - Move from one place in memory to another
    - Move from one register to another
    - Move from memory to register
    - Move from register to memory
  - Instructions often want to operate on data:
    - Add contents of register X to contents of register Y

- **Hence "mov" (move) instruction is paramount!**

mov

- **`mov    src,dst`**
  - Optional suffix (b,w,l,q): `movb, movw, movl, movq`
  - One confusing thing about "move" it makes it sound like it leaves the src "empty"—no!
    - Does a **copy**, like the assignment operator you are familiar with
  - `src,dst` options:
    - Immediate (AKA constant value)
    - Register
    - Memory

Basic addressing modes (Think: assembly version of VARIABLES)

- Notice that one major difference between high-level code and assembly instructions is the absence of programmer-chosen, descriptive variable names:

```
int total_goodness = nReeses + nButterfinger;
```

```
addl 8(%rbp),%eax
```

- We don't get to choose variable names, we have to talk directly about places in hardware
- **"Addressing modes" are allowable ways of <u>naming</u> these places**

Basic addressing modes

(Think: assembly version of VARIABLES)

Op	Source	Dest	Dest Comments
`movl`	`$0,`	`%eax`	Name of a register
`movl`	`$0,`	`0x8f2713e0`	Actual address literal (note address literals are different from other literals—don't need $ in front)
`movl`	`$0,`	`(%rax)`	Look in the register named, find an address there, and use it

Reminder: need to put $ in front of immediate values (constant literals)

Basic addressing modes (Think: assembly version of VARIABLES)

Op	Source	Dest	Dest Comments
`movl`	`$0,`	`%eax`	Name of a register
`movl`	`$0,`	`0x8f2713e0`	Actual address literal (note address literals are different from other literals—don't need $ in front)
`movl`	`$0,`	`(%rax)`	Look in the register named, find an address there, and use it
`movl`	`$0,`	`-24(%rbp)`	Add -24 to an address in the named register, and use that address

Displacement must be a constant; to have a variable base and variable displacement , use two steps: add then mov

Displacement can be positive or negative

Basic addressing modes

(Think: assembly version of VARIABLES)

Op	Source	Dest	Dest Comments
`movl`	`$0,`	`%eax`	Name of a register
`movl`	`$0,`	`0x8f2713e0`	Actual address literal (note address literals are different from other literals—don't need $ in front)
`movl`	`$0,`	`(%rax)`	Look in the register named, find an address there, and use it
`movl`	`$0,`	`-24(%rbp)`	Add -24 to an address in the named register, and use that address
`movl`	`$0`	`8(%rbp, %eax, 2)`	Address to use = (8 + address in rbp) + (2 * index in eax)

Any constant allowed

Only 1, 2, 4, 8 allowed

Matching exercise: Addressing modes use cases

- Match up which use cases make the most sense for which addressing modes (some guesswork expected)

short arr[size];

Op	Src	Dest	Use case?
movl	$0	8(%rbp, %eax, 2)	(c)
movl	$0,	%eax	(a)
movl	$0,	0x8f2713e0	(d)
movl	$0,	4(%rax)	(b)

Use cases	
(a)	Prepare to use 0 in a calculation
(b)	Zero out a field of a struct
(c)	Zero out a given array bucket
(d)	Zero out a global variable

Instruction Set Architectures

Some context and terminology

Stanford University

Instruction Set Architecture

- **The ISA defines:**
  - › Operations that the processor can execute
  - › Data transfer operations + how to access data
  - › Control mechanisms like branch, jump (think loops and if-else)
  - › Contract between programmer/compiler and hardware
- **Layer of abstraction:**
  - › Above:
    - Programmer/compiler can write code for the ISA
    - New programming languages can be built on top of the ISA as long as the compiler will do the translation
  - › Below:
    - New hardware can implement the ISA
    - Can have even potentially radical changes in hardware implementation
    - Have to "do" the same thing from programmer point of view
- **ISAs have incredible inertia!**
  - › Legacy support is a huge issue for x86-64

Two major categories of Instruction Set Architectures

- **CISC:**
  - › **Complex** instruction set computers
    - e.g., x86 **(CS107 studies this)**
  - › Have special instructions for each thing you might want to do
  - › Can write code with fewer instructions, because each instruction is very expressive
- **RISC:**
  - › **Reduced** instruction set computers
    - e.g., MIPS
  - › Have only a very tiny number of instructions, optimize the heck out of them in the hardware
  - › Code may need to be longer because you have to go roundabout ways of achieving what you wanted

large, varied "menu"

McDonald's

IN-N-OUT BURGER

offer a tiny menu, but do it very, very well