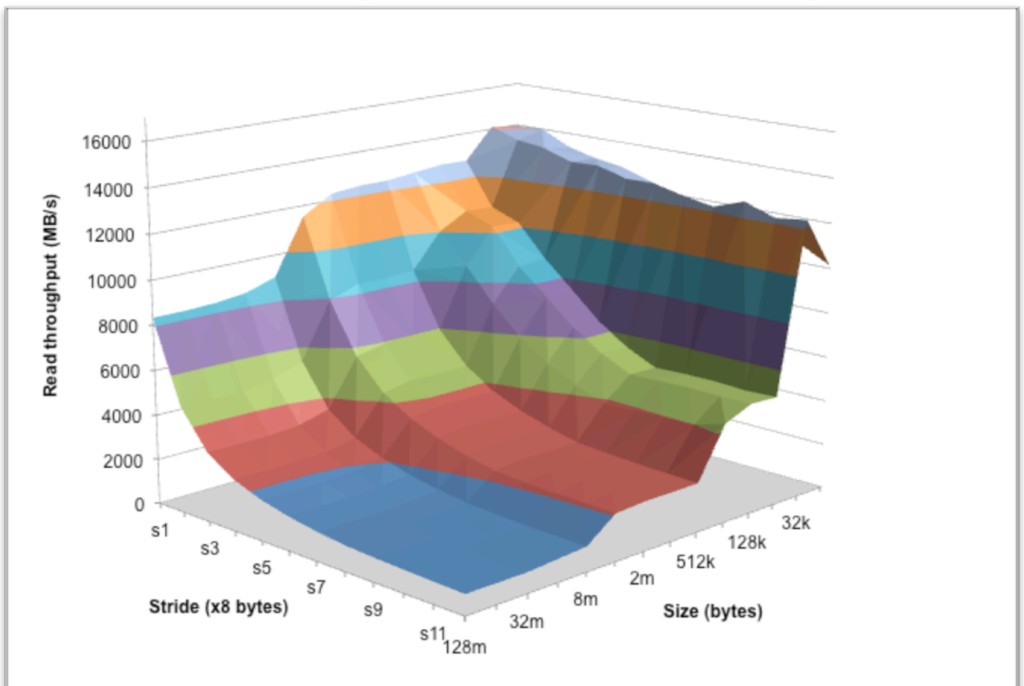
Introducing Computer Systems from a Programmer's Perspective

Randal E. Bryant, David R. O'Hallaron

Computer Science, Electrical & Computer Engineering

Carnegie Mellon University



Outline

Introduction to Computer Systems

- Course taught at CMU since Fall, 1998
- Some ideas on labs, motivations, …

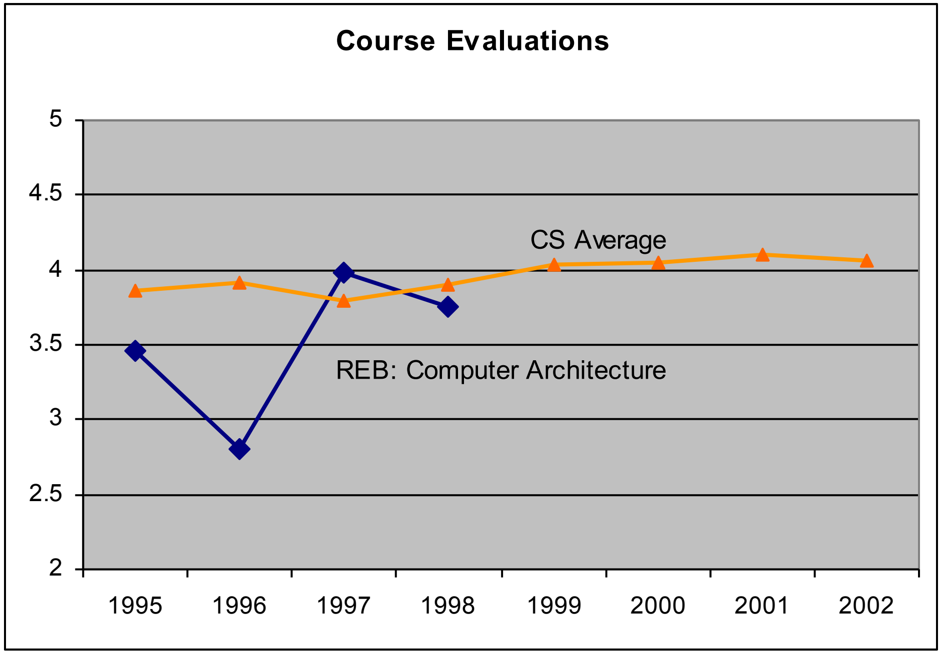
Computer Systems: A Programmer’s Perspective

- Our textbook, now in its third edition
- Ways to use the book in different courses

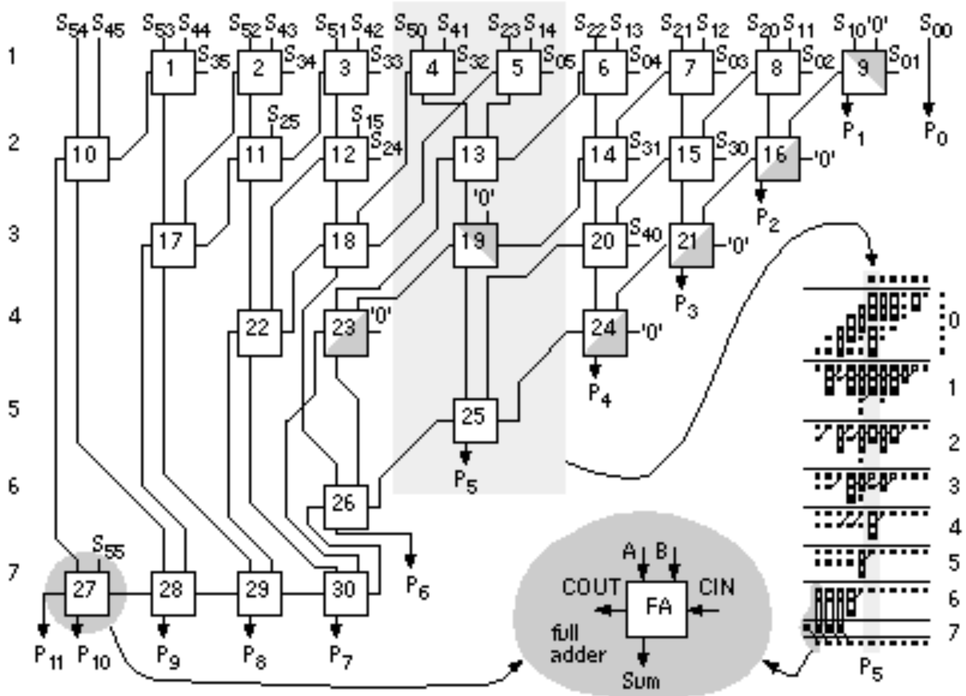
Background

1995-1997: REB/DROH teaching computer architecture course at CMU.

- Good material, dedicated teachers, but students hate it
- Don’t see how it will affect their lives as programmers



Computer Arithmetic
Builder's Perspective



- **How to design high performance arithmetic circuits**

Computer Arithmetic
Programmer's Perspective

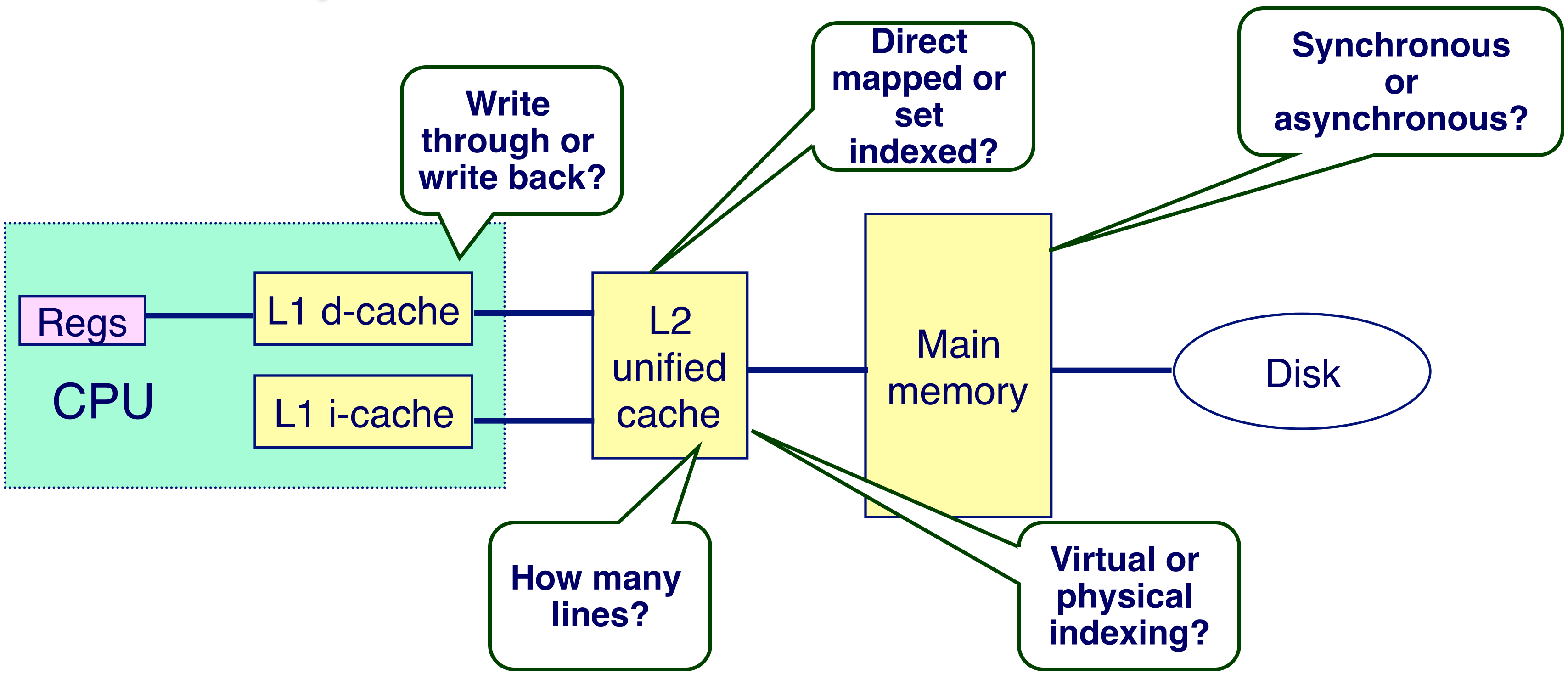
```
void show_squares()
{
  int x;
  for (x = 5; x <= 5000000; x*=10)
    printf("x = %d x^2 = %d\n", x, x*x);
}
```

x	=	5	x^2	=	25
x	=	50	x^2	=	2500
x	=	500	x^2	=	250000
x	=	5000	x^2	=	25000000
x	=	50000	x^2	=	-1794967296
x	=	500000	x^2	=	891896832
x	=	5000000	x^2	=	-1004630016

- **Numbers are represented using a finite word size**
- **Operations can overflow when values too large**
  - **But behavior still has clear, mathematical properties**

Memory System
Builder's Perspective

Builder's Perspective



- Must make many difficult design decisions
- Complex tradeoffs and interactions between components

Memory System
Programmer's Perspective

```
void copyij(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (i = 0; i < 2048; i++)
    for (j = 0; j < 2048; j++)
      dst[i][j] = src[i][j];
}
```

4.3 ms

```
void copyji(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (j = 0; j < 2048; j++)
    for (i = 0; i < 2048; i++)
      dst[i][j] = src[i][j];
}
```

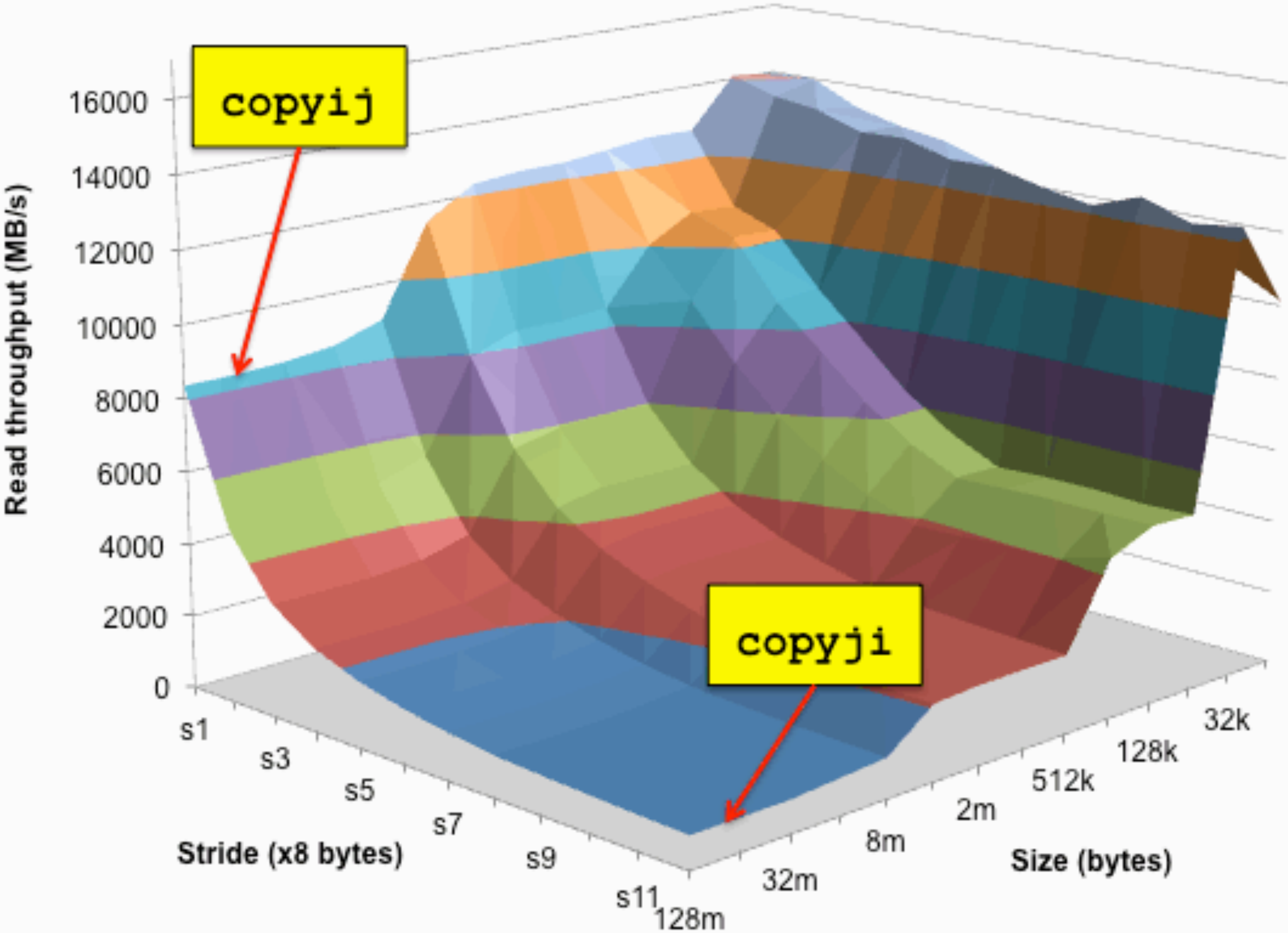
81.8 ms

19 times slower!

(Measured on 2 GHz Intel Core i7 Haswell)

- **Hierarchical memory organization**
- **Performance depends on access patterns**
  - **Including how step through multi-dimensional array**

The Memory Mountain



Background (Cont.)

1997: OS instructors complain about lack of preparation

- Students don't know machine-level programming well enough
  - What does it mean to store the processor state on the run-time stack?
- Our architecture course was not part of prerequisite stream

Birth of ICS

1997: REB/DROH pursue new idea:

- Introduce them to computer systems from a programmer's perspective rather than a system designer's perspective.
- Topic Filter: What parts of a computer system affect the correctness, performance, and utility of my C programs?

1998: Replace architecture course with new course:

- 15-213: Introduction to Computer Systems

Curriculum Changes

- Sophomore level course
- Eliminated digital design & architecture as required courses for CS majors

15-213: Intro to Computer Systems

Goals

- Teach students to be sophisticated application programmers
  - Immediate value, even if never take another systems course
- Prepare students for upper-level systems courses

Taught every semester to 400+ students

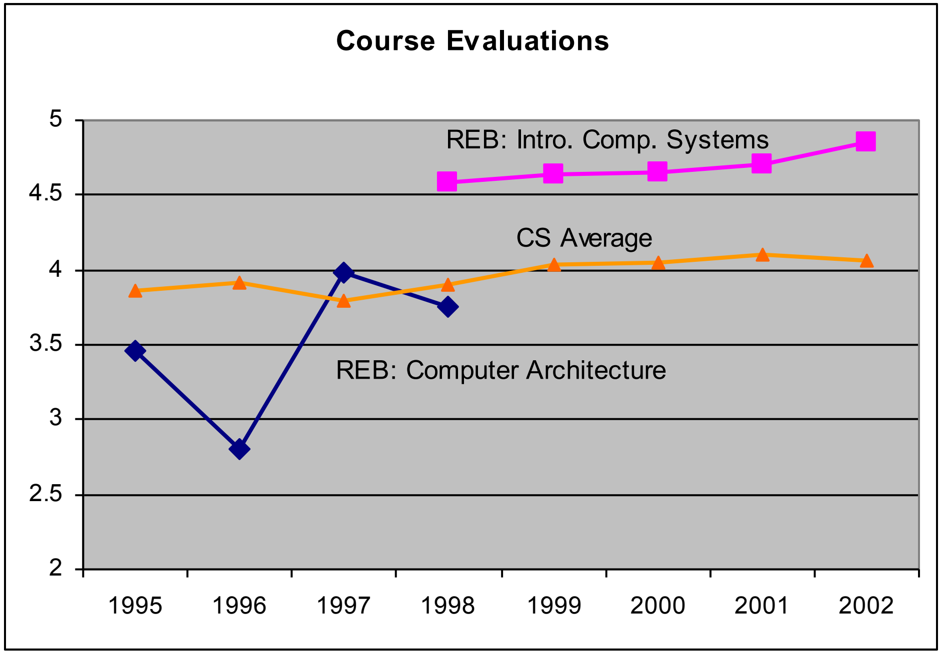
- All CS undergrads (core course)
- All ECE undergrads (core course)
- Many masters students
  - To prepare them for upper-level systems courses
- Variety of others from math, physics, statistics, …

Preparation

- Optional: Introduction to CS in Python or Ruby
- Imperative programming in C subset

ICS Feedback

Students



Faculty

- Prerequisite for most upper level CS systems courses
- Also required for ECE embedded systems, architecture, and network courses

Lecture Coverage

Data representations [3]

- It's all just bits.
- `int`'s are not integers and `float`'s are not reals.

x86-64 machine language [5]

- Analyzing and understanding compiler-generated machine code.

Program optimization [2]

- Understanding compilers and modern processors.

Memory Hierarchy [3]

- Caches matter!

Linking [1]

- With DLL's, linking is cool again!

Lecture Coverage (cont)

Exceptional Control Flow [2]

- The system includes an operating system that you must interact with.

Virtual memory [4]

- How it works, how to use it, and how to manage it.

Application level concurrency [3]

- Processes and threads
- Races, synchronization

I/O and network programming [4]

- Programs often need to talk to other programs.

Total: 27 lectures, 14 week semester

Labs

Key teaching insight:

- Cool Labs $\Rightarrow$ Great Course

A set of 1 and 2 week labs define the course.

Guiding principles:

- Be hands on, practical, and fun.
- Be interactive, with continuous feedback from automatic graders
- Find ways to challenge the best while providing worthwhile experience for the rest
- Use healthy competition to maintain high energy.

Lab Exercises

Data Lab (2 weeks)

- Manipulating bits.

Bomb Lab (2 weeks)

- Defusing a binary bomb.

Attack Lab (1 week)

- Buffer overflow and return-oriented programming exploits

Cache Lab (2 weeks)

- Write basic cache simulator and then optimize application

Shell Lab (1 week)

- Writing your own shell with job control.

Malloc Lab (2-3 weeks)

- Writing your own malloc package.

Proxy Lab (2 weeks)

- Writing your own concurrent Web proxy.

Data Lab

Goal: Solve some "bit puzzles" in C using a limited set of logical and arithmetic operators.

- Examples: `absval(x)`, `greaterthan(x,y)`, `log2(x)`

Lessons:

- Information is just bits in context.
- C `int`'s are not the same as integers.
- C `float`'s are not the same as reals.

Infrastructure

- Configurable source-to-source C compiler that checks for compliance.
- Instructor can automatically select from 45 puzzles.
- Automatic testing using formal verification tools

Let’s Solve a Bit Puzzle!

```
/*
 * abs - absolute value of x (except returns TMin for TMin)
 *   Example: abs(-1) = 1.
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 10
 *   Rating: 4
 */
int abs(int x) {
  int mask = x>>31;

  return  (x^mask)  +  1+~mask  ;
}
```

$$\begin{cases} 11\ldots1_2, = -1, & x < 0 \\ 00\ldots0_2, = \ \ 0, & x \geq 0 \end{cases}$$

$$\begin{cases} -x-1, & x < 0 \\ x, & x \geq 0 \end{cases} + \begin{cases} 1, & x < 0 \\ 0, & x \geq 0 \end{cases} = \begin{cases} -x, & x < 0 \\ x, & x \geq 0 \end{cases}$$

Verifying Solutions

```
int abs(int x) {
  int mask = x>>31;
  return (x ^ mask) + ~mask + 1;
}
```

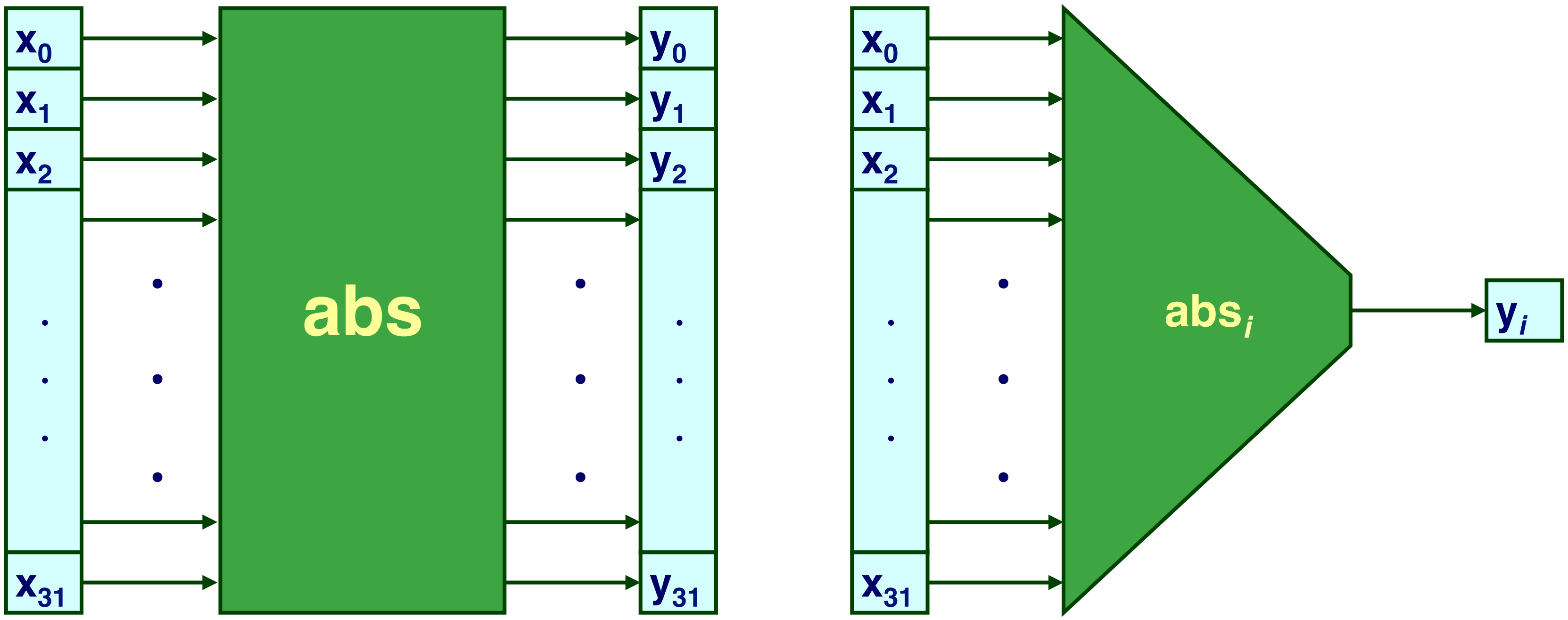
```
int test_abs(int x) {
  return (x < 0) ? -x : x;
}
```

Do these functions produce identical results?

How could you find out?

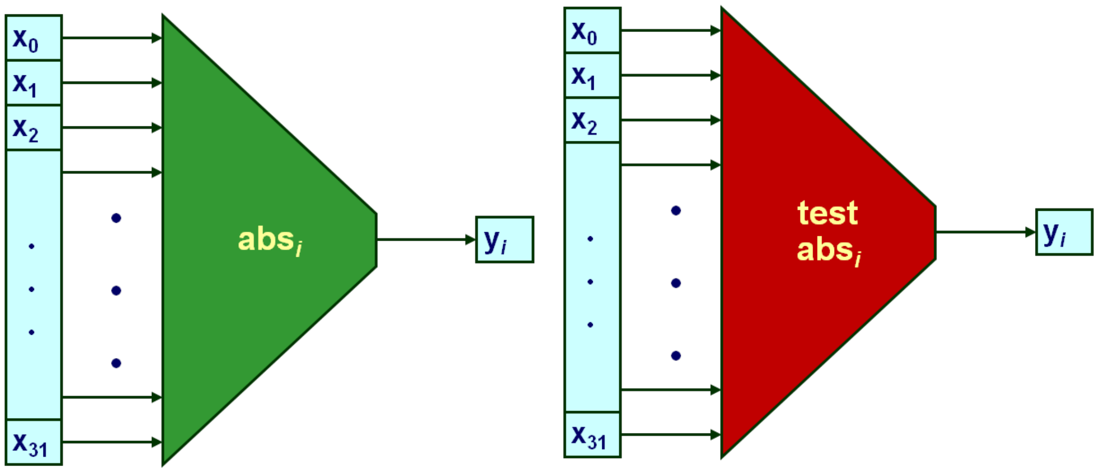
Bit-Level Program Model

```
int abs(int x) {
  int mask = x>>31;
  return (x ^ mask) + ~mask + 1;
}
```



- **View computer word as 32 separate bit values**
- **Each output becomes Boolean function of inputs**

Bit-Level Program Verification



- Determine whether functions equivalent for all outputs j
- Exhaustive checking:
  - Single input: $$\frac{2^{32} \text{ cases X } 50 \text{ cycles}}{2 \text{ X } 10^9 \text{ cycles / second}} \approx 60 \text{ seconds}$$
  - Two input: 2^{64} cases ➔ 8,800 years!
- Other approaches
  - BDDs, SAT solvers
  - Easily handle these functions (< 1.0 seconds)

Verification Example

```
int iabs(int x) {
  if (x == 1234567) x++;
  int mask = x>>31;
  return (x ^ mask) + ~mask + 1;
}
```

Almost Correct

- Valid for all but one input value
- Overlooked by our test suite

Counterexample Generation

```
int iabs(int x) {
  if (x == 1234567) x++;
  int mask = x>>31;
  return (x ^ mask) + ~mask + 1;
}
```

Detected By Checking Code

- Since covers *all* cases
- Generate counterexample to demonstrate problem

```
int main()
{
    int val1 = iabs(1234567);
    int val2 = test_iabs(1234567);
    printf("iabs(1234567) --> %d [0x%x]\n", val1, val1);
    printf("test_iabs(1234567) --> %d [0x%x]\n", val2, val2);
    if (val1 == val2) {
       printf(".. False negative\n");
    } else
       printf(".. A genuine counterexample\n");
}
```

Bomb Lab

- Idea due to Chris Colohan, TA during inaugural offering

Bomb: C program with six *phases*.

Each phase expects student to type a specific string.

- Wrong string: bomb *explodes* by printing BOOM! (- ½ pt)
- Correct string: phase *defused* (+10 pts)
- In either case, bomb sends message to grading server
- Server posts current scores anonymously and in real time on Web page

Goal: Defuse the bomb by defusing all six phases.

- For fun, we include an unadvertised seventh *secret phase*

The challenge:

- Each student get only binary executable of a *unique* bomb
- To defuse their bomb, students must disassemble and reverse engineer this binary

Properties of Bomb Phases

Phases test understanding of different C constructs and how they are compiled to machine code

- Phase 1: string comparison
- Phase 2: loop
- Phase 3: switch statement/jump table
- Phase 4: recursive call
- Phase 5: pointers
- Phase 6: linked list/pointers/structs
- Secret phase: binary search (biggest challenge is figuring out how to reach phase)

Phases start out easy and get progressively harder

Let's defuse a bomb phase!

```
0000000000400a6c <phase_2>:
  ... # function prologue not shown
  400a72:       mov    %rsp,%rsi
  400a75:       callq  4010ba <read_six_numbers>  # rd 6 ints into buffer
  400a7a:       cmpl   $0x1,(%rsp)
  400a7e:       je     400a85 <phase_2+0x19>
  400a80:       callq  400f6d <explode_bomb>
  400a85:       lea    0x4(%rsp),%rbx             # p = &buf[1]
  400a8a:       lea    0x18(%rsp),%rbp            # pend = &buf[6]
  400a8f:       mov    -0x4(%rbx),%eax            # LOOP: v = buf[0]
  400a92:       add    %eax,%eax                  # v = 2*v
  400a94:       cmp    %eax,(%rbx)                # if v == *p
  400a96:       je     400a9d <phase_2+0x31>      #    then goto OK:
  400a98:       callq  400f6d <explode_bomb>      #    else explode!
  400a9d:       add    $0x4,%rbx                  # OK: p++
  400aa1:       cmp    %rbp,%rbx                  # if p != pend
  400aa4:       jne    400a8f <phase_2+0x23>      #    then goto LOOP:
   ... # function epilogue not shown
  400aac:       c3                     retq       # YIPPEE!
```

Source Code for Bomb Phase

```
/*
 * phase2b.c - To defeat this stage the user must enter the geometric
 * sequence starting at 1, with a factor of 2 between each number
 */
void phase_2(char *input)
{
    int i;
    int numbers[6];

    read_six_numbers(input, numbers);

    if (numbers[0] != 1)
        explode_bomb();

    for(i = 1; i < 6; i++) {
        if (numbers[i] != numbers[i-1] * 2)
            explode_bomb();
    }
}
```

The Beauty of the Bomb

For the Student

- Get a deep understanding of machine code in the context of a fun game
- Learn about machine code in the context they will encounter in their professional lives
  - Working with compiler-generated code
- Learn concepts and tools of debugging
  - Forward vs backward debugging
  - Students *must* learn to use a debugger to defuse a bomb

For the Instructor

- Self-grading
- Scales to different ability levels
- Easy to generate variants and to port to other machines

Attack Lab

```
int getbuf()
{
    char buf[4];
    /* Read line of text and store in buf */
    gets(buf);
    return 1;
}
```

Task

- Each student assigned “cookie”
  - Randomly generated 8-digit hex string
- Generate string that will cause `getbuf` to return cookie
  - Instead of `1`

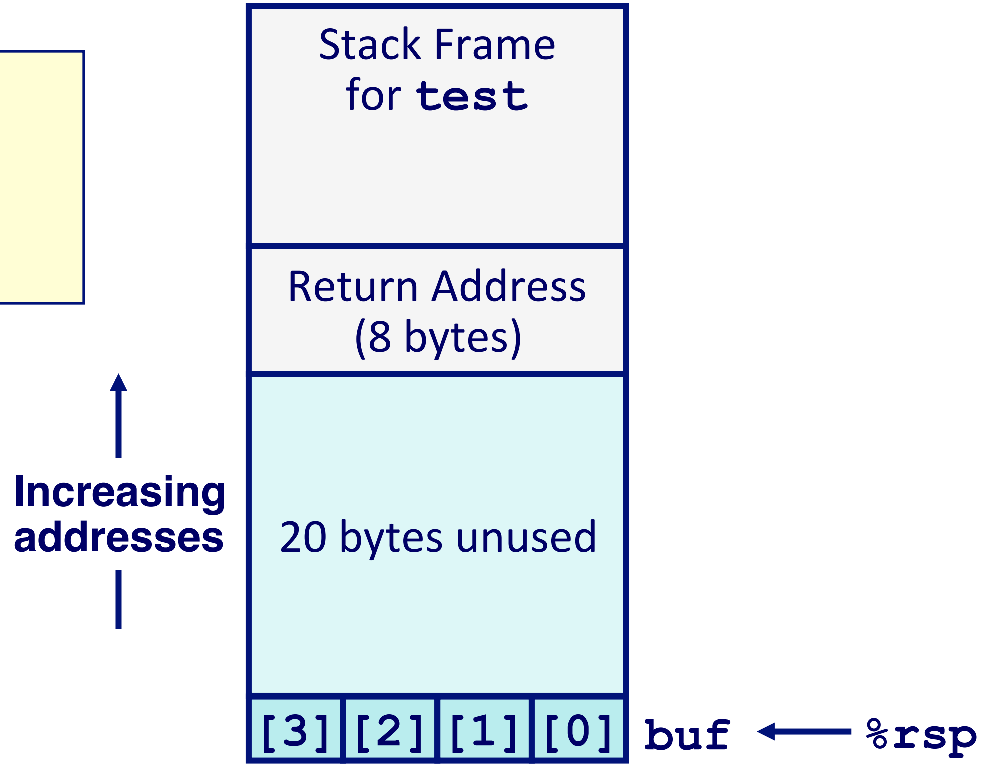
Buffer Code

Return address

```
void test(){
   int v = getbuf();
   ...
}
```

```
void getbuf() {
   char buf[4];
   gets(buf);
   return 1;
}
```

Stack when `gets` called



- **Calling function `gets(p)` reads characters up to '`\n`'**
- **Stores string + terminating null as bytes starting at `p`**
- **Assumes enough bytes allocated to hold entire string**

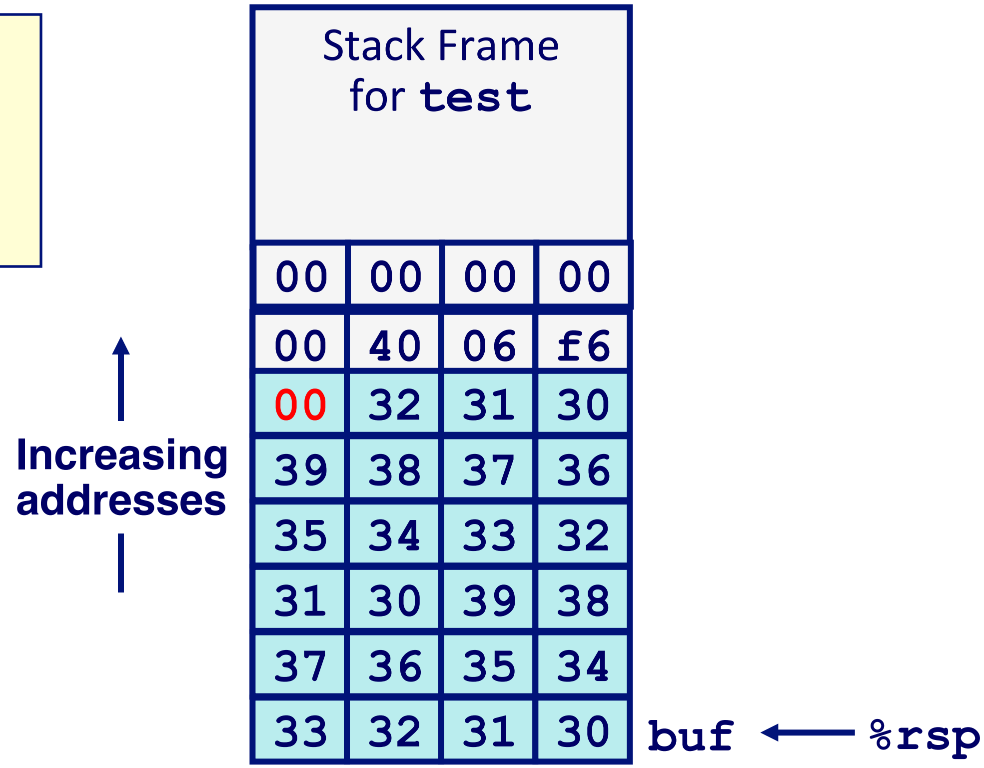
Buffer Code: Good case

Return address

```
void test(){
  int v = getbuf();
  ...
}
```

```
void getbuf() {
  char buf[4];
  gets(buf);
  return 1;
}
```

Input string
“01234567890123456789012”



- **Fits within allocated storage**
  - String is 23 characters long + 1 byte terminator

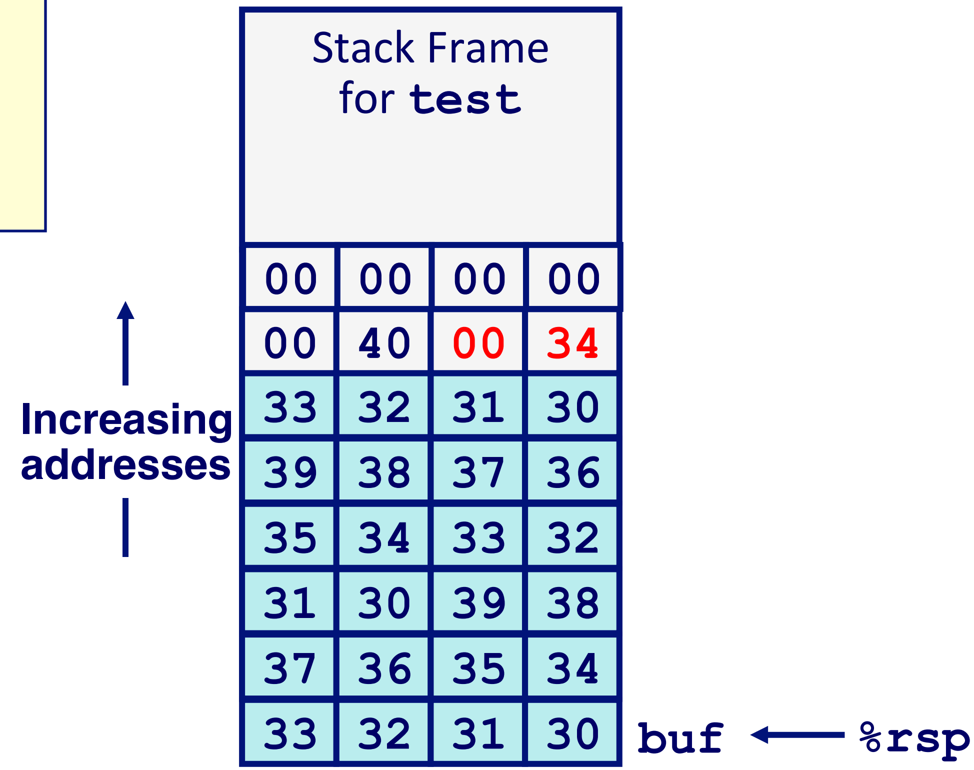
Buffer Code: Bad case

Input string
“01234567890123456789012 34”

Return address

```
void test(){
   int v = getbuf();
   ...
}
```

```
void getbuf() {
   char buf[4];
   gets(buf);
   return 1;
}
```



- Overflows allocated storage
  - Corrupts saved frame pointer and return address
- Jumps to address `0x400034` when `getbuf` attempts to return
  - Program executes some instruction and then segfaults

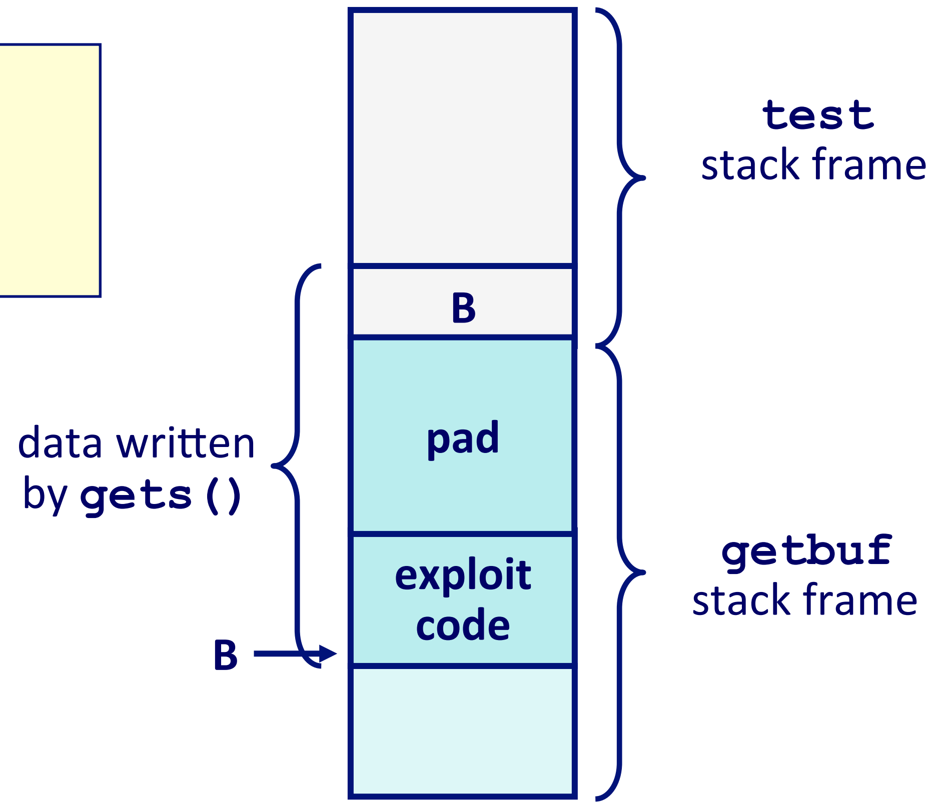
Malicious Use of Buffer Overflow

Return address

```
void test(){
  int v = getbuf();
  ...
}
```

```
void getbuf() {
  char buf[4];
  gets(buf);
  return 1;
}
```

Stack after call to **gets()**



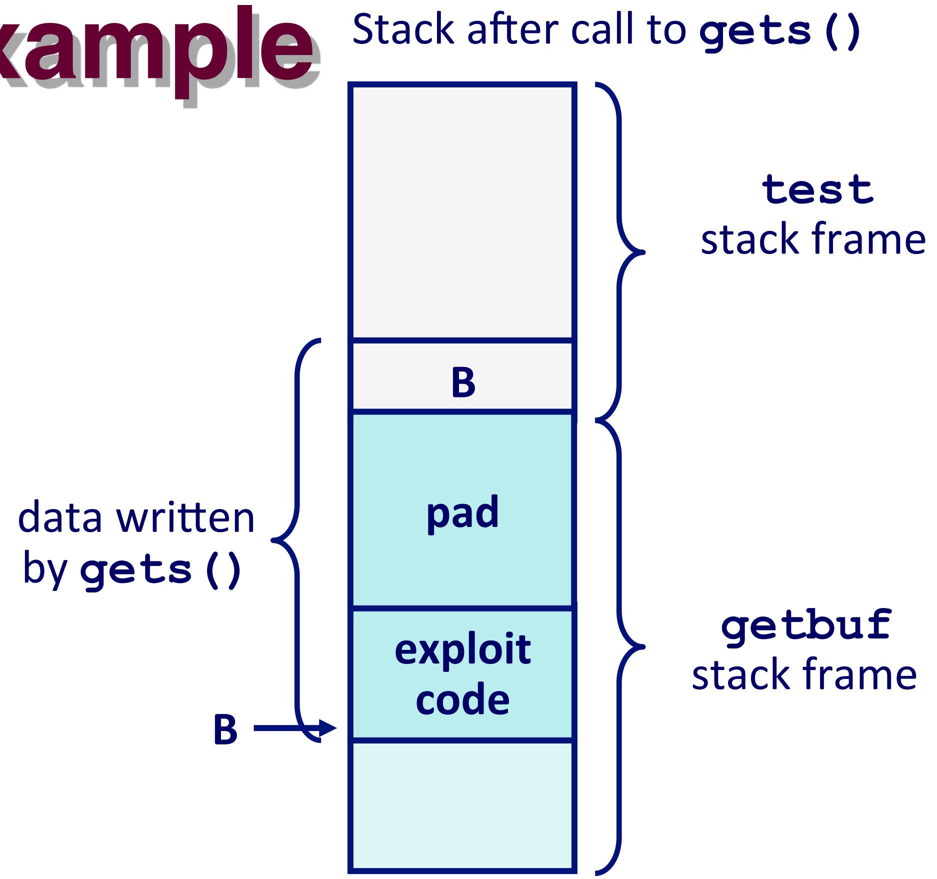
- Input string contains byte representation of executable code
- Overwrite return address with address of buffer
- When `getbuf()` executes return instruction, will jump to exploit code

Exploit String Example

```
void getbuf() {
  char buf[4];
  gets(buf);
  return 1;
}
```

- **Sets `0x59b997fa` as function argument**
- **Invokes function touch2**

```
/* Byte code for shell code
   movq $0x59b997fa,%rdi; ret */
48 c7 c7 fa 97 b9 59 c3
/* Pad with 16 bytes */
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
/* Address of shellcode */
78 dc 61 55 00 00 00 00
/* Address of touch2 */
0c 18 40 00 00 00 00 00
```



Why Do We Teach This Stuff?

Important Systems Concepts

- Stack discipline and stack organization
- Instructions are byte sequences
- Making use of tools
  - Debuggers, assemblers, disassemblers

Computer Security

- What makes code vulnerable to buffer overflows
- Common vulnerability in systems

Impact

- CMU student teams consistently win international Capture the Flag Competitions

Cache Lab

Goal: Understanding Cache Operations

- How memory locations map to cache blocks
- Performance implications for application programs

Activities

- Write cache simulator
  - Provides full understanding of mapping from memory address to cache location
- Minimize cache misses for simple application
  - Matrix transpose

Shell Lab

Goal: Write a Unix shell with job control

- (e.g., ctrl-z, ctrl-c, jobs, fg, bg, kill)

Lessons:

- First introduction to systems-level programming and concurrency
- Learn about processes, process control, signals, and catching signals with handlers
- Demystifies command line interface

Infrastructure

- Students use a scripted autograder to incrementally test functionality in their shells

Malloc Lab

Goal: Build your own dynamic storage allocator

```
void *malloc(size_t size)
void *realloc(void *ptr, size_t size)
void free(void *ptr)
```

Lessons

- Sense of programming underlying system
- Large design space with classic time-space tradeoffs
- Develop understanding of scary “action at a distance” property of memory-related errors
- Learn general ideas of resource management

Infrastructure

- Trace driven test harness evaluates implementation for combination of throughput and memory utilization
- Evaluation server and real time posting of scores

Proxy Lab

Goal: write concurrent Web proxy.

Lessons: Ties together many ideas from earlier

- Data representations, byte ordering, memory management, concurrency, processes, threads, synchronization, signals, I/O, network programming, application-level protocols (HTTP)

Infrastructure:

- Plugs directly between existing browsers and Web servers
- Grading is done via autograders and one-on-one demos
- Very exciting for students, great way to end the course

ICS Summary

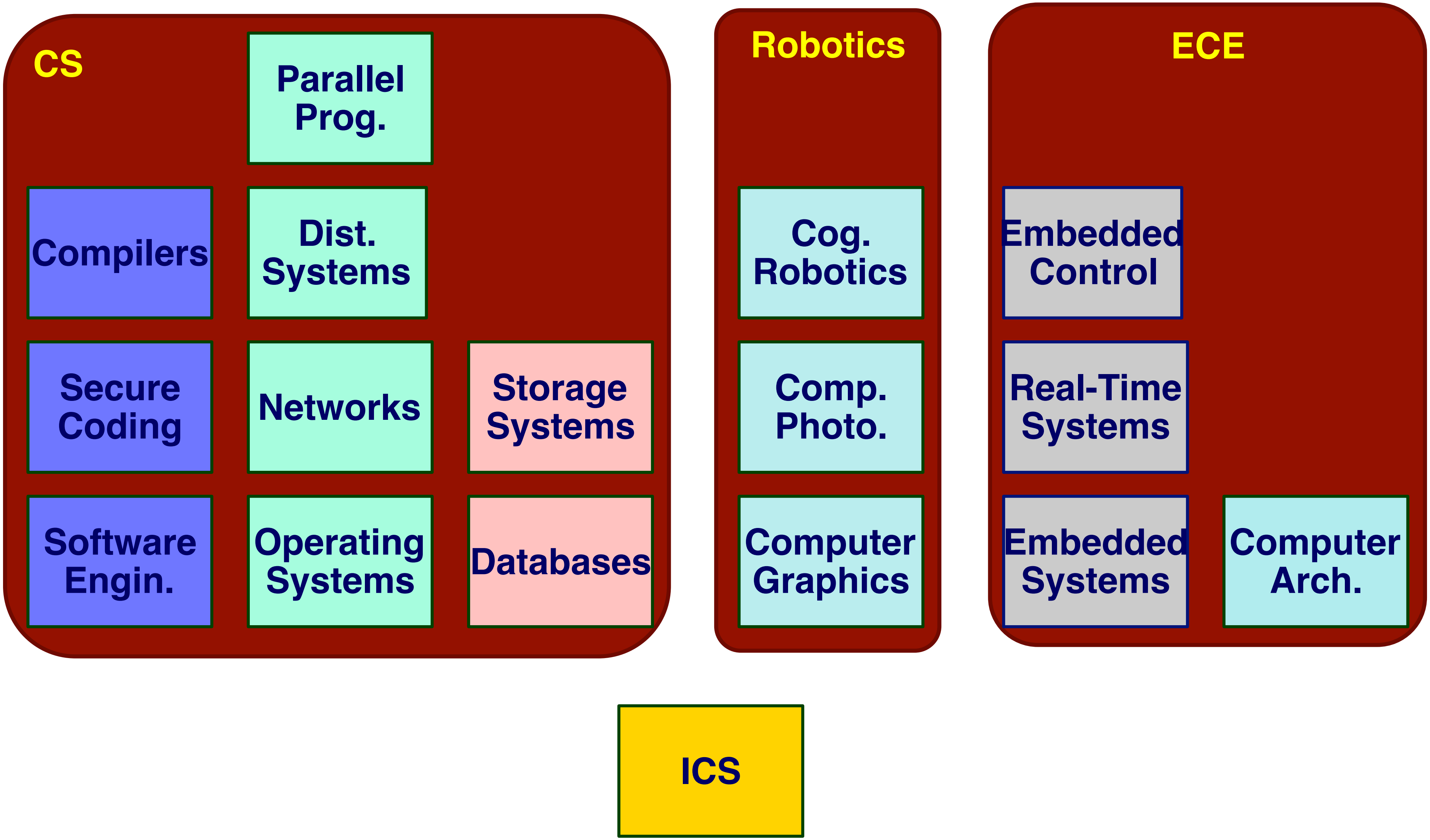
Principle

- *Introduce students to computer systems from the programmer's perspective rather than the system builder's perspective*

Themes

- What parts of the system affect the correctness, efficiency, and utility of my C programs?
- Makes systems fun and relevant for students
- Prepare students for builder-oriented courses
  - Architecture, compilers, operating systems, networks, distributed systems, databases, …
  - Since our course provides complementary view of systems, does not just seem like a watered-down version of a more advanced course
  - Gives them better appreciation for what to build

CMU Courses that Build on ICS



Fostering “Friendly Competition”

Desire

- Challenge the best without frustrating everyone else

Method

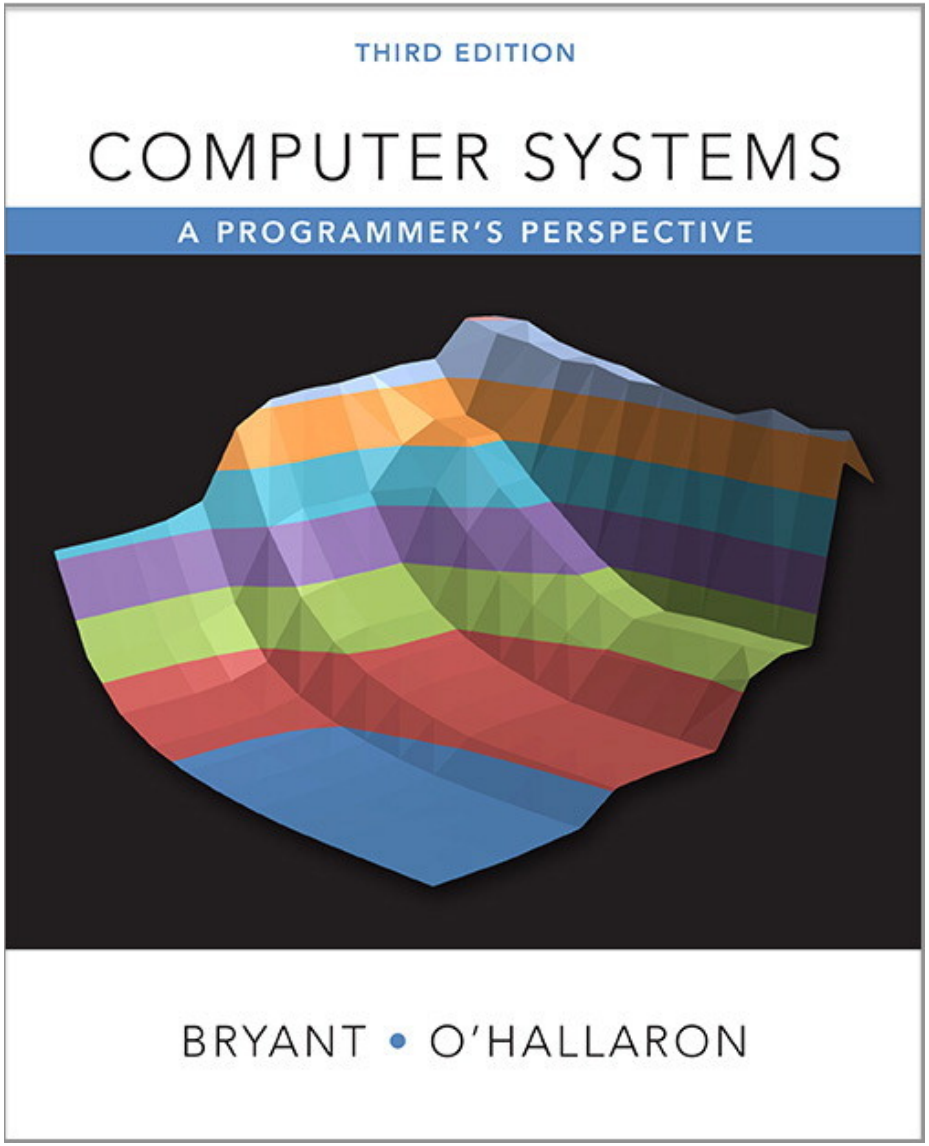
- Web-based submission of solutions
- Server checks for correctness and computes performance score
  - How many stages passed, program throughput, …
- Keep updated results on web page
  - Students choose own *nom de guerre*

Relationship to Grading

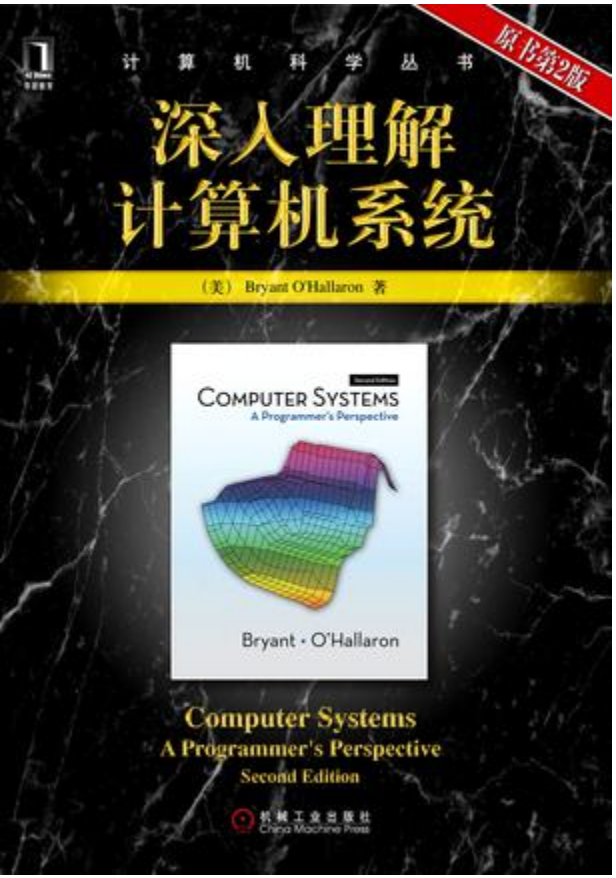
- Students get full credit once they reach set threshold
- Push beyond this just for own glory/excitement

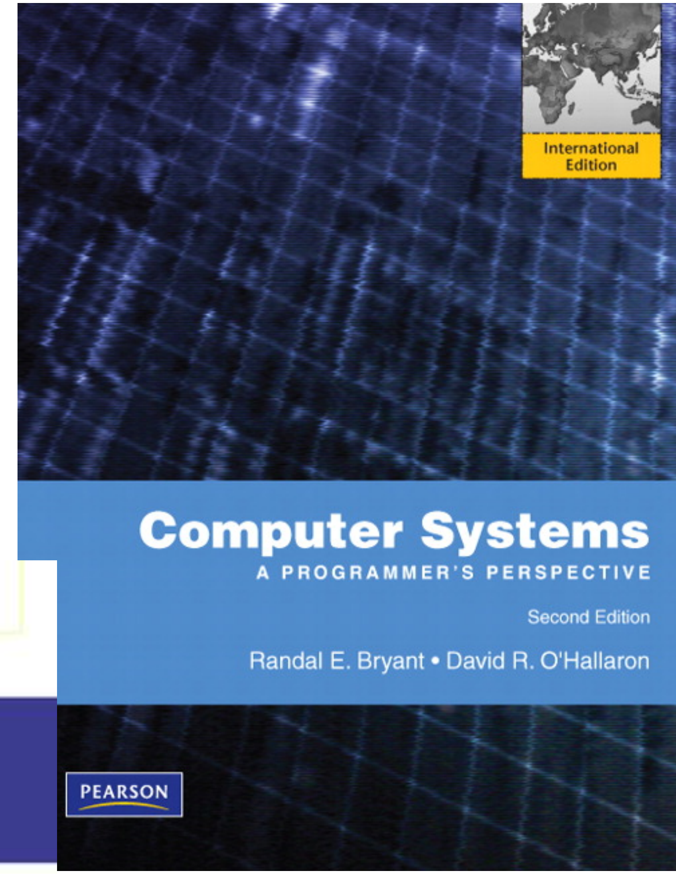
Shameless Promotion

- `http://csapp.cs.cmu.edu`
- Third edition published 2015
- In use at 289 institutions worldwide



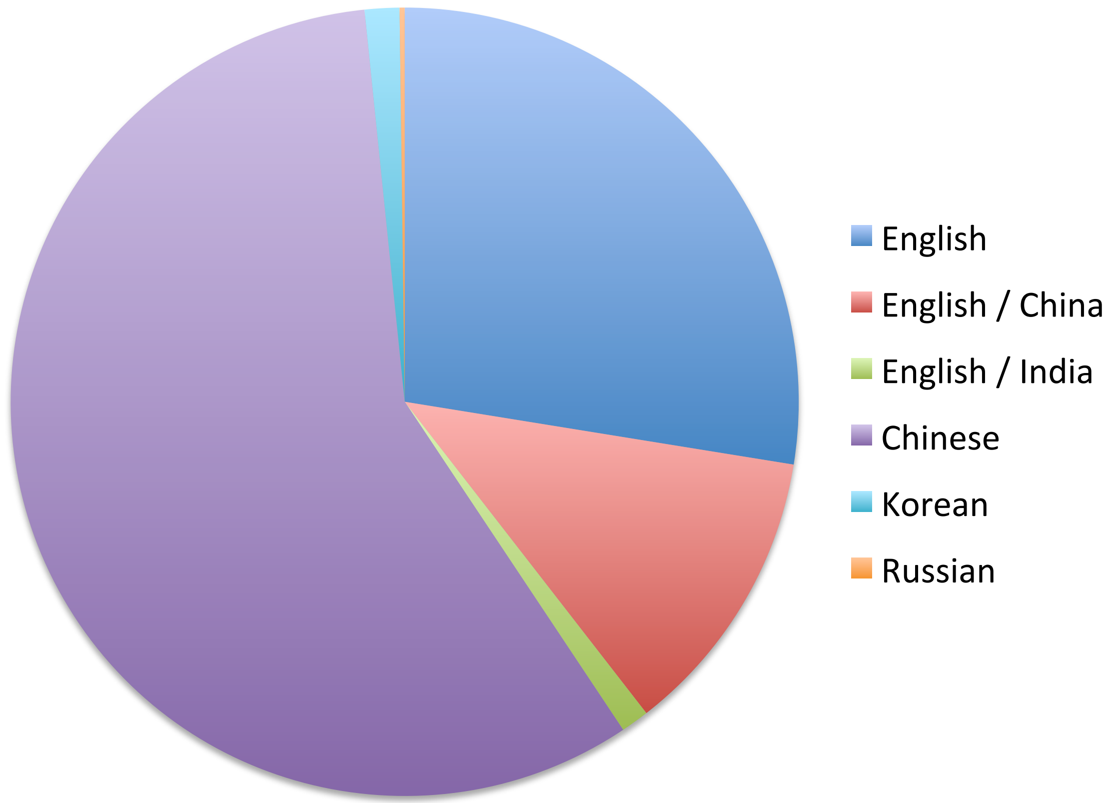
International Editions (No 3rd edition yet)





Overall Sales

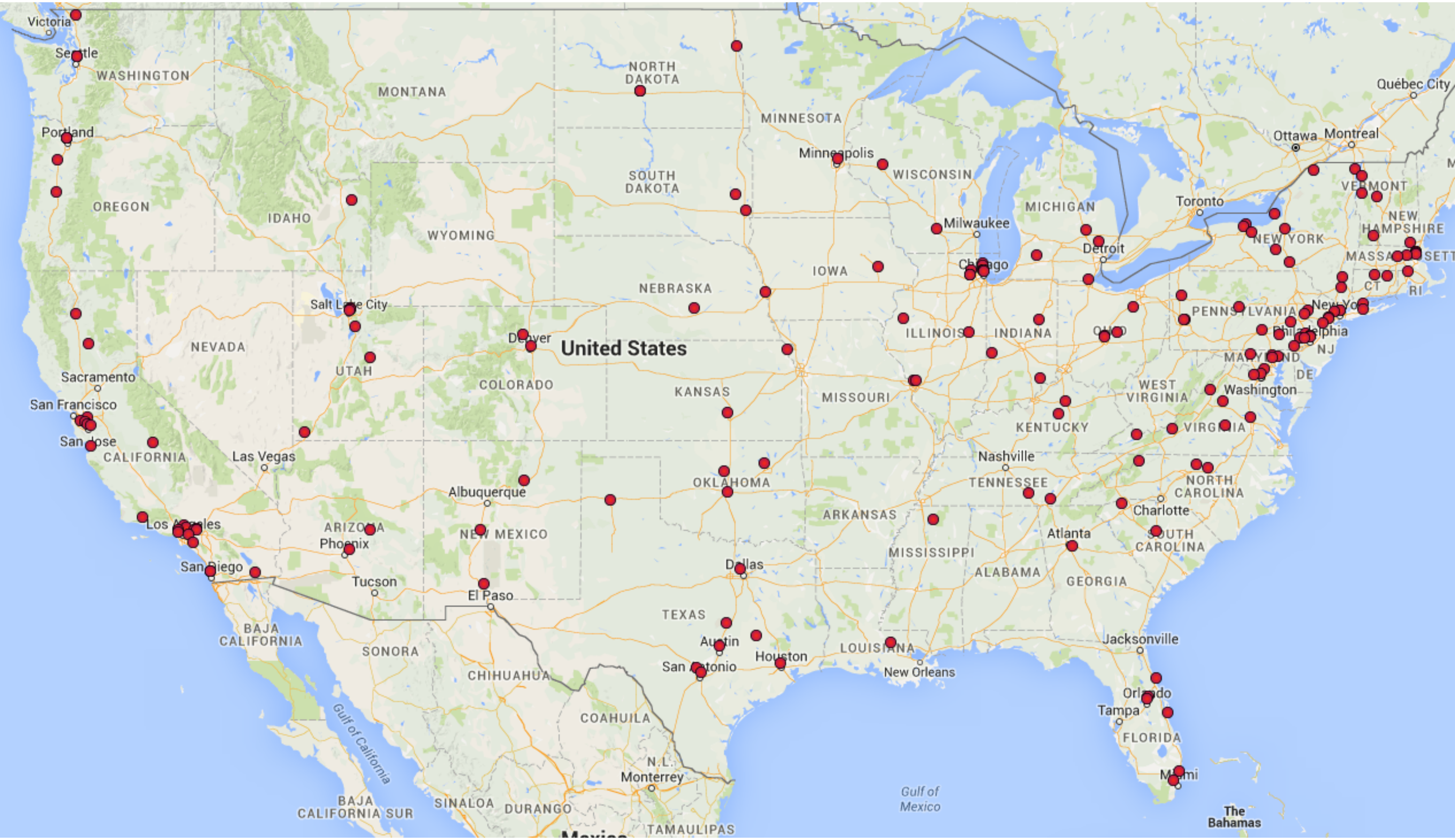
- All Editions
- As of 6/30/2015
- 175,835 total



Worldwide Adoptions

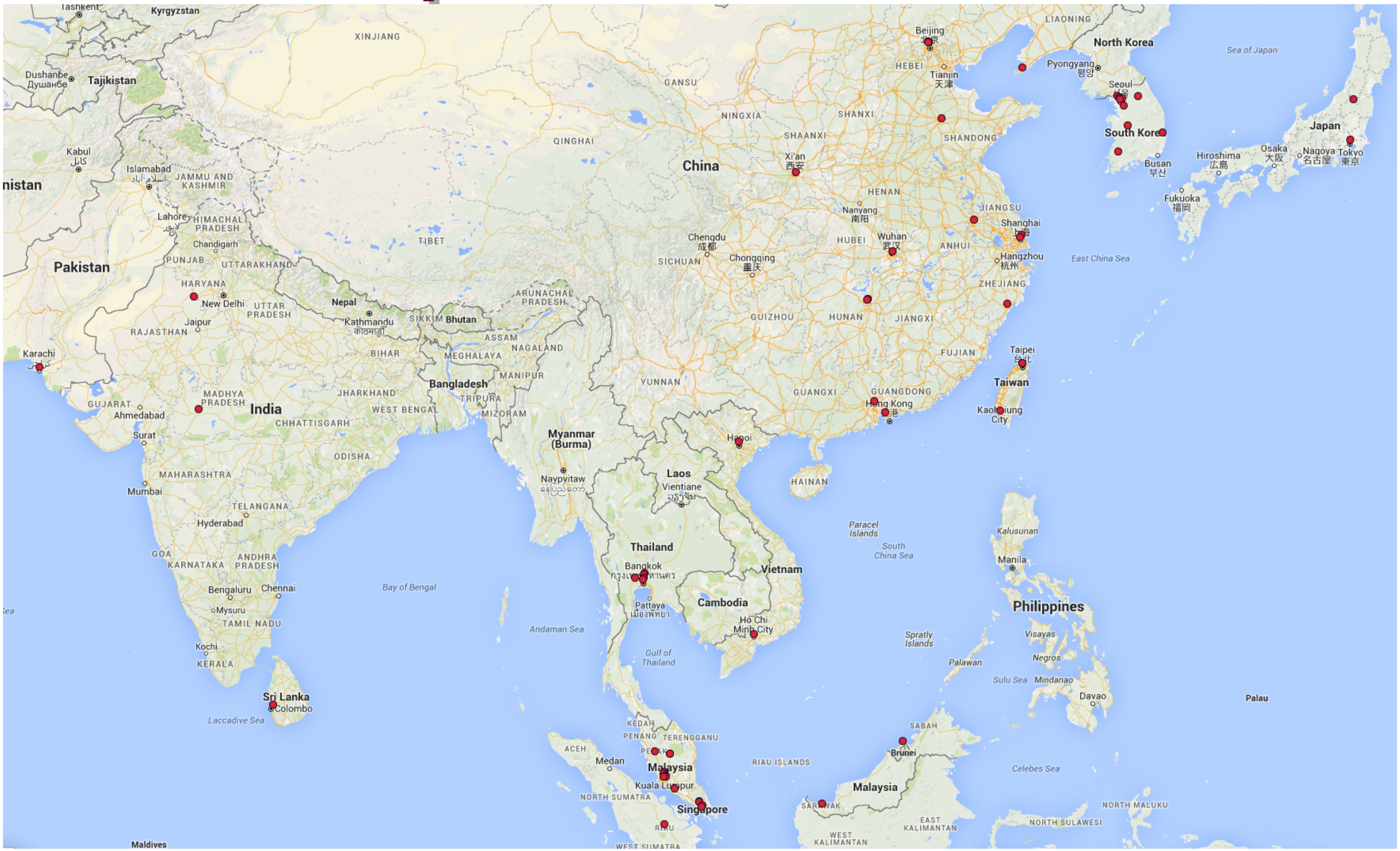
289 total

US Adoptions



176 total

Asian Adoptions



European Adoptions

CS:APP3e

Vital stats:

- 12 chapters
- 267 practice problems (solutions in book)
- 226 homework problems (solutions in instructor's manual)
- 544 figures, 342 line drawings
- Many C & machine code examples

Turn-key course provided with book:

- Electronic versions of all code examples.
- Powerpoint and PDF versions of each line drawing
- Password-protected Instructors Page
  - Instructor's Manual
  - Lab Infrastructure
  - Powerpoint lecture notes
  - Exam problems.

Coverage

Material Used by ICS at CMU

- Pulls together material previously covered by multiple textbooks, system programming references, and man pages

Greater Depth on Some Topics

- Dynamic linking
- I/O multiplexing

Additional Topic

- Computer Architecture
- Added to cover all topics in “Computer Organization” course

Architecture

Material

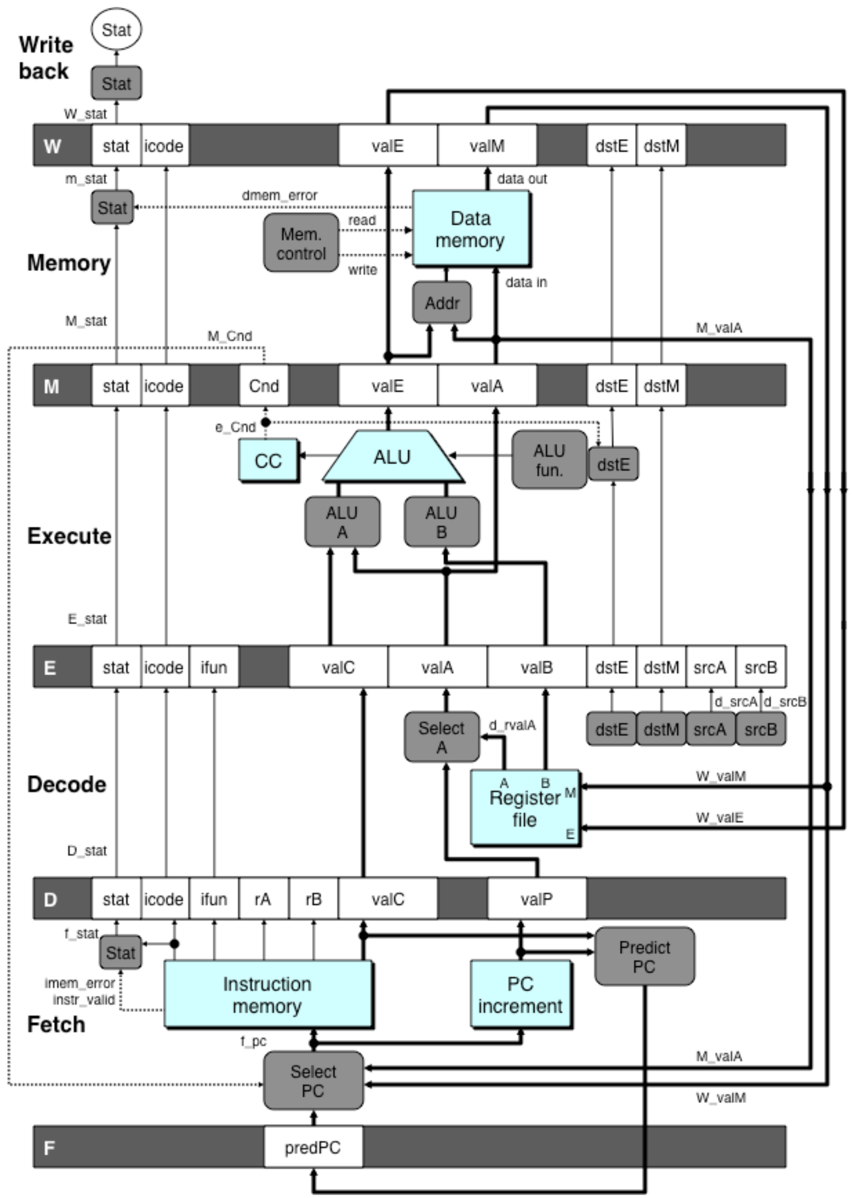
- Y86-64 instruction set
  - Simplified/reduced x86-64
- Implementations
  - Sequential
  - 5-stage pipeline

Presentation

- Simple hardware description language to describe control logic
- Automatic translation to simulator and to Verilog

Labs

- Modify / extend processor design
  - New instructions
  - Change branch prediction policy
- Optimize application + processor



Web Asides

- Supplementary material via web
- Topics either more advanced or more arcane

Examples

- Boolean algebra & Boolean rings
- IA32 programming
- Combining assembly & C code
- Processor design in Verilog
- Using SIMD instructions
- Memory blocking

Courses Based on CS:APP

Computer Organization

ORG Topics in conventional computer organization course, but with a different flavor

ORG+ Extends computer organization to provide more emphasis on helping students become better application programmers

Introduction to Computer Systems

ICS Create enlightened programmers who understand enough about processor/OS/compilers to be effective

ICS+ What we teach at CMU. More coverage of systems software

Systems Programming

SP Prepare students to become competent system programmers

Courses Based on CS:APP

Chapter	Topic	Course				
		ORG	ORG+	ICS	ICS+	SP
1	Introduction	●	●	●	●	●
2	Data representations	●	●	●	●	○
3	Machine language	●	●	●	●	●
4	Processor architecture	●	●			
5	Code optimization		●	●	●	
6	Memory hierarchy	○	●	●	●	○
7	Linking			○	○	●
8	Exceptional control flow			●	●	●
9	Virtual memory	○	●	●	●	●
10	System-level I/O				●	●
11	Concurrent programming				●	●
12	Network programming				●	●

○ **Partial Coverage** ● **Complete Coverage**

Conclusions

ICS Has Proved Its Success

- Thousands of students at CMU over 13 years
- Positive feedback from alumni
- Positive feedback from systems course instructors

CS:APP is International Success

- Supports variety of course styles
- Many purchases for self study