HARVARD

School of Engineering
and Applied Sciences

CS 61:
Systems programming and machine organization

Lecture 1: Introduction

Prof. Stephen Chong

September 2, 2010

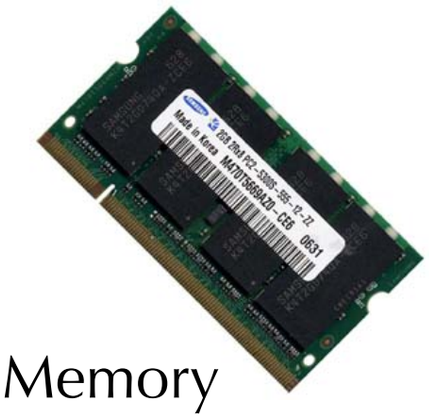
Welcome to CS 61!

- What is this course about?
  - Intro to computer systems
- What happens when I run a program?
  - Delving into mysteries of how machines really work
  - Get "under the hood" of programming at machine level
  - Understand what affects performance of your programs
    - Processor architecture
    - Caching, memory management
    - Processes, threads, synchronization
    - …
- Write rock solid (and fast) systems code

What we're going to cover

- Answer "What happens when I run a program?"
- Learn how computer systems work
  - How processors work and what affects their performance
    - Linking, loading, execution of programs
    - Memory
  - How to read x86 assembly code
- Learn about OS-level programming
  - UNIX system programming: files, processes, pipes, signals
  - Concurrency: threads and synchronization
- Do some very cool labs
  - Get hands dirty in assembly, buffer overruns, low-level memory management...

What happens when you run a program?



Memory

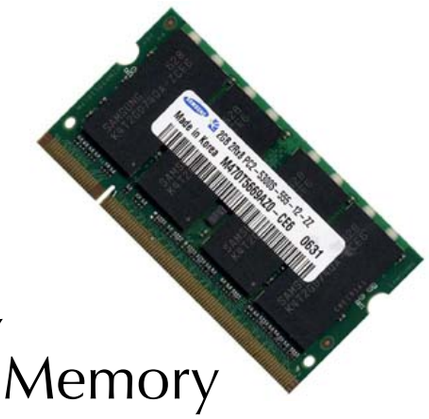
Load program

Disk storage

What happens when you run a program?

Processor

Load instruction



Memory

Write results

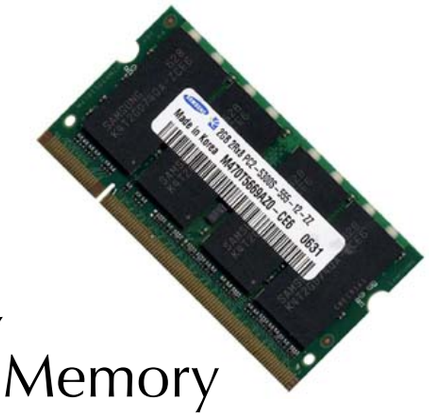
What happens when you run a program?

Processor

Load next instruction

Repeat forever...

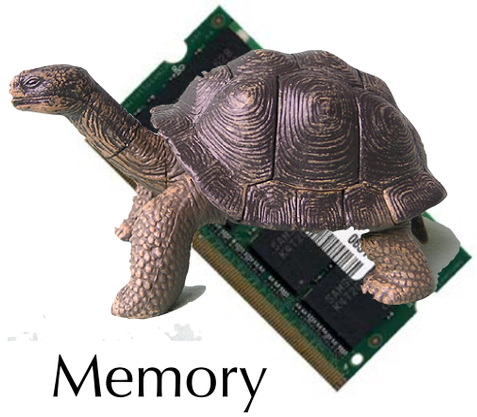
Write results



Memory

But all is not well...

Processor



Memory

- Memory is slow compared to CPU!
  - About 40 nanoseconds to access memory
  - About 0.4 nanoseconds per CPU cycle

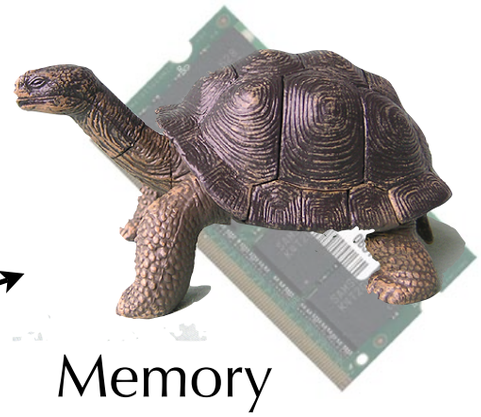
Caches

Processor

~1.5ns

Cache

~40ns



Memory

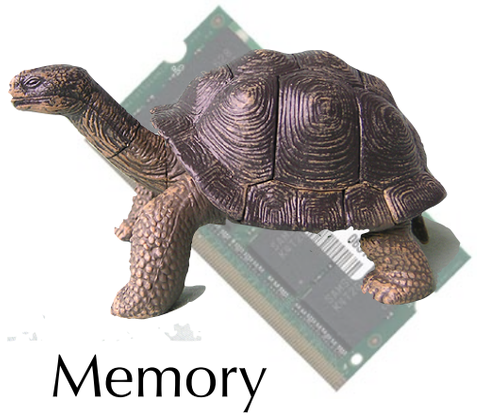
- Cache stores recently-used memory
- Very fast to access, but not very big (kilobytes).
  - About 10x faster than main memory
  - About 100x more expensive!
- Often located on the same chip as the CPU itself.

Cache effects

- Caches generally speed up memory access considerably.
- But they have limitations:
  - Not very big, so large data sets will not fit.
  - Often optimized for specific memory access patterns.
- Upshot: Programs need to be aware of cache locality
  - Want data structures to “fit” into the cache
  - Code should be structured to make best use of cache size and structure.

Caches all the way down...

Processor

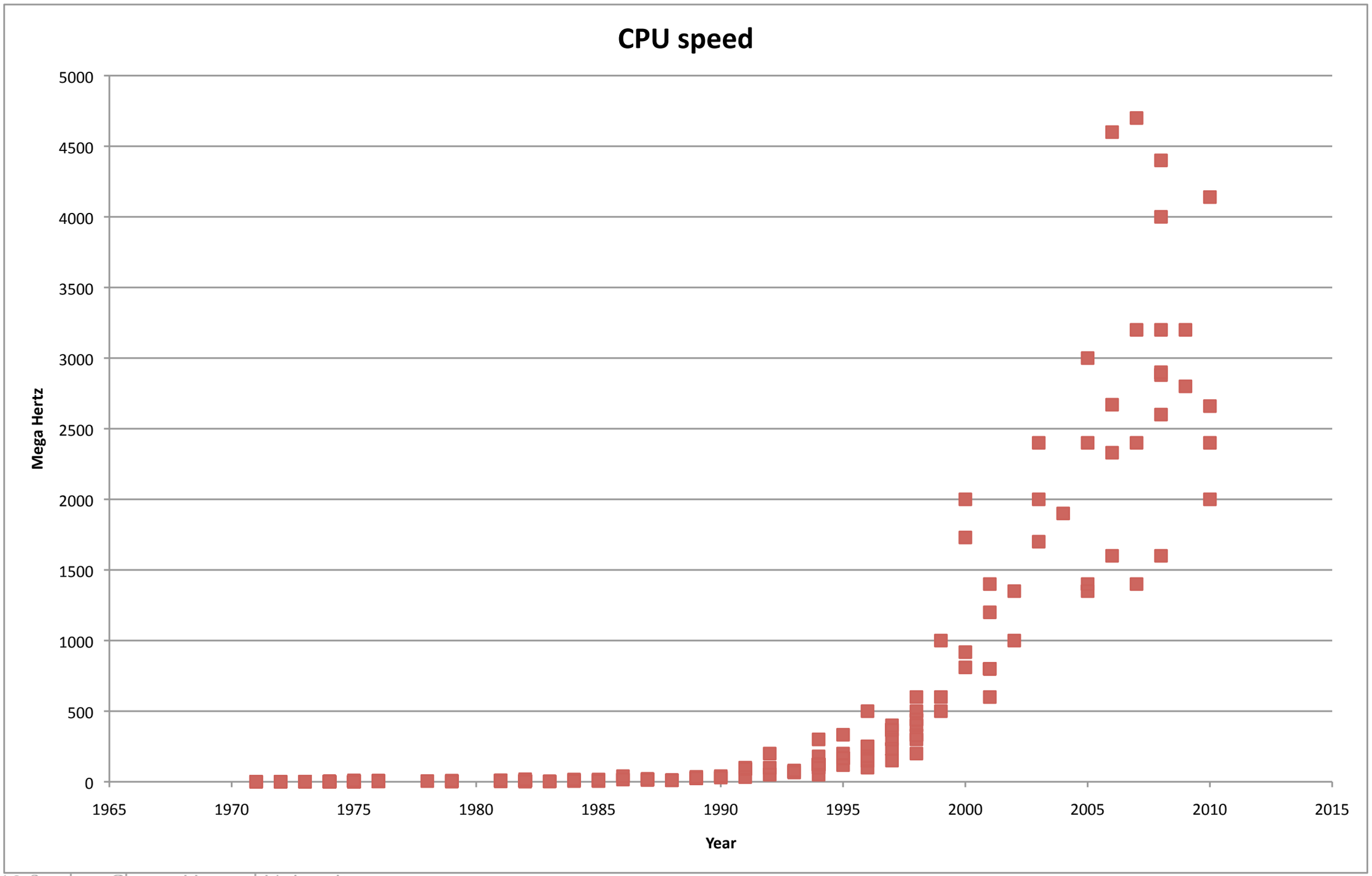


Memory

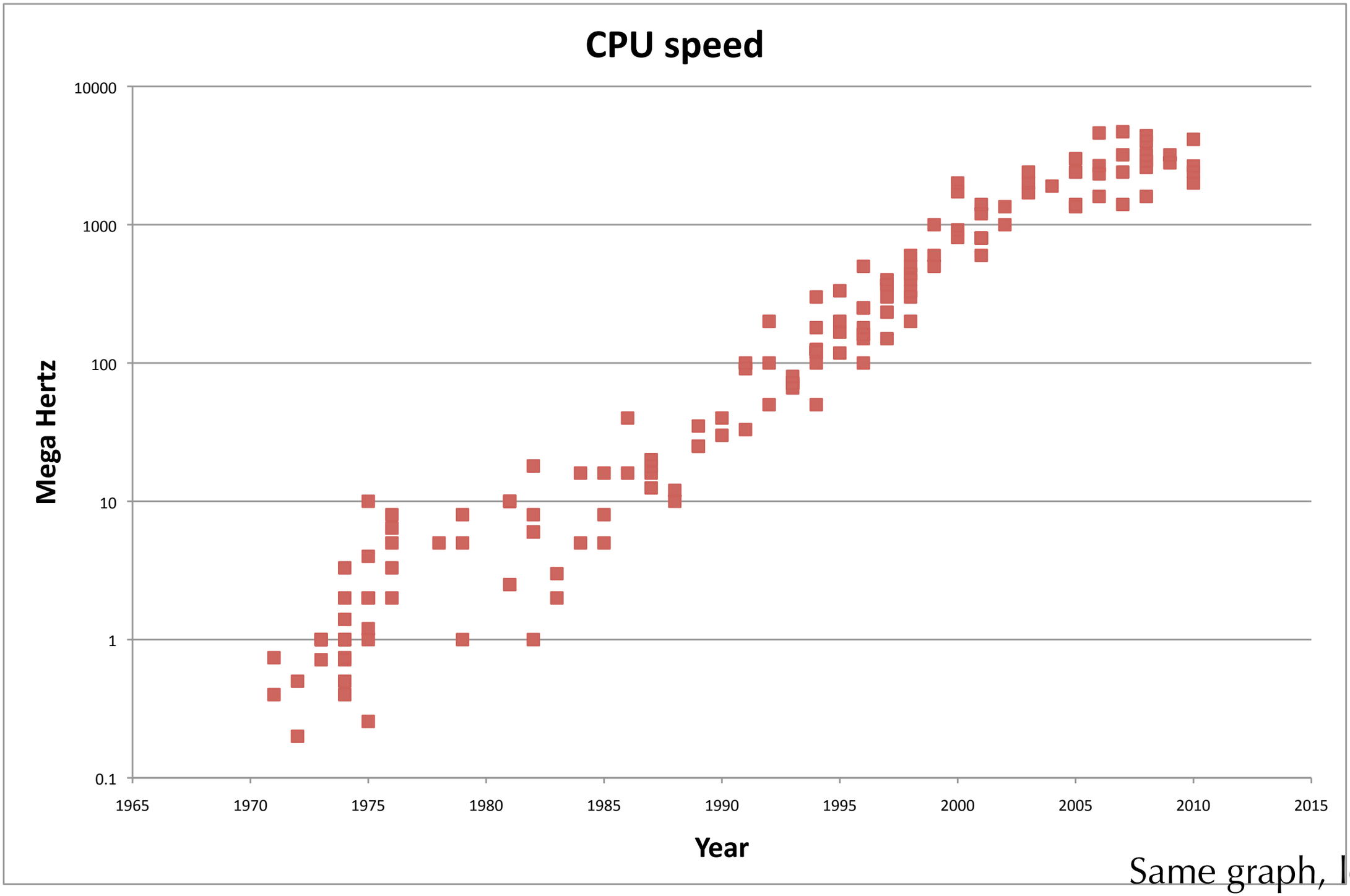
Disk storage

- Hard drive much slower than memory!
  - About 3ms = 3,000,000ns = millions of cycles!
- Main memory is a cache for disk storage...

CPUs getting faster...

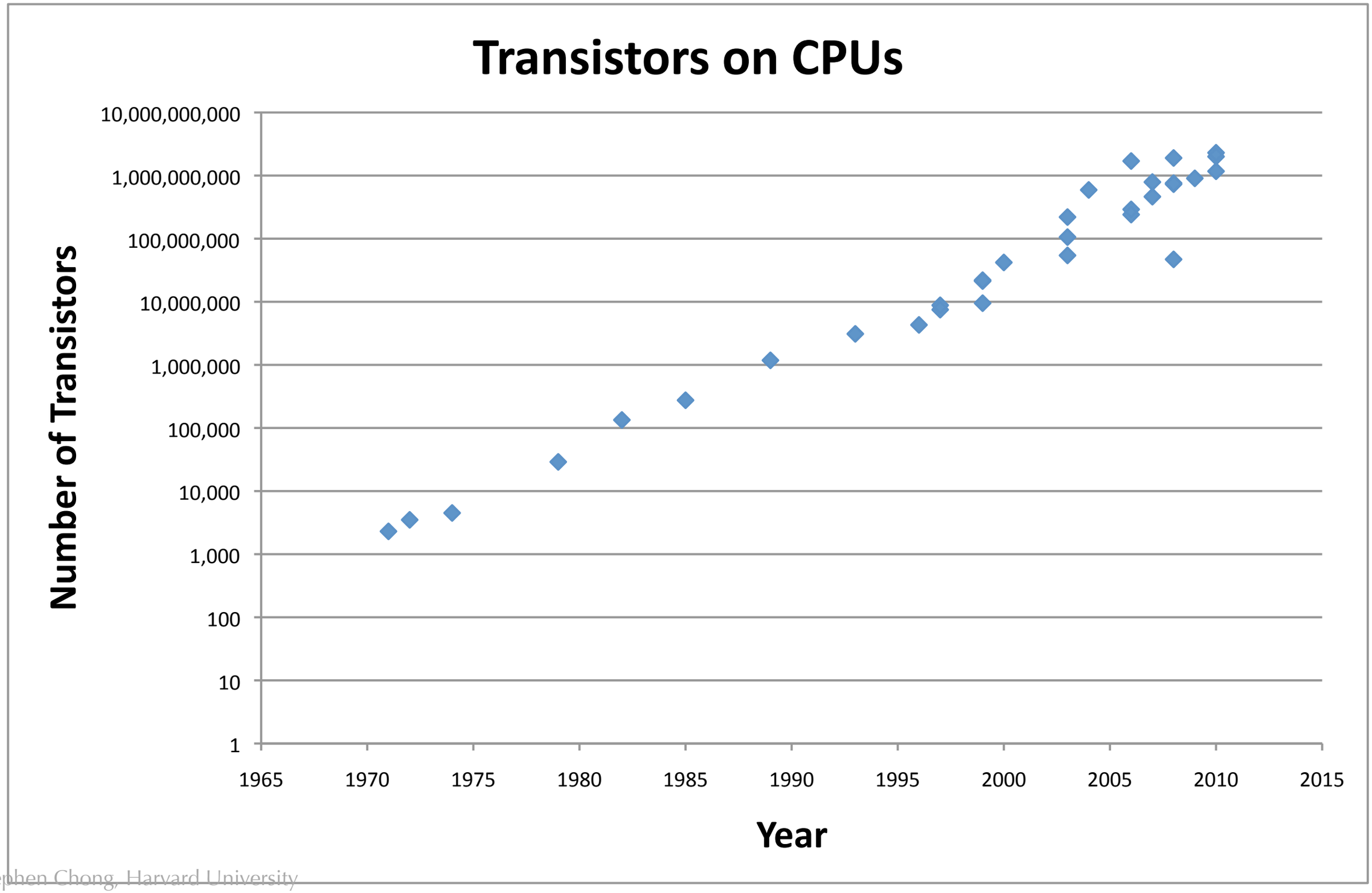


CPUs getting faster...



Same graph, log scale

…and bigger



Moore's Law

- Moore's Law (1965): *Density of transistors on a chip will double every 24 months.*
  - Has remained essentially true for 40 years.
  - More transistors = more computing power
  - Smaller transistors = faster clock speeds

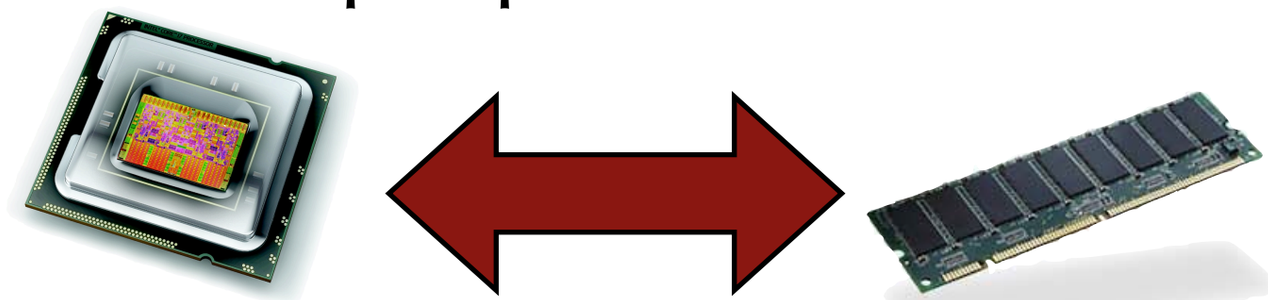
- Problems…
  - Power densities: More components = more heat
  - Increased cost for driving down transistor sizes
  - Fundamental limits of the size of transistors

Where are those transistors going?

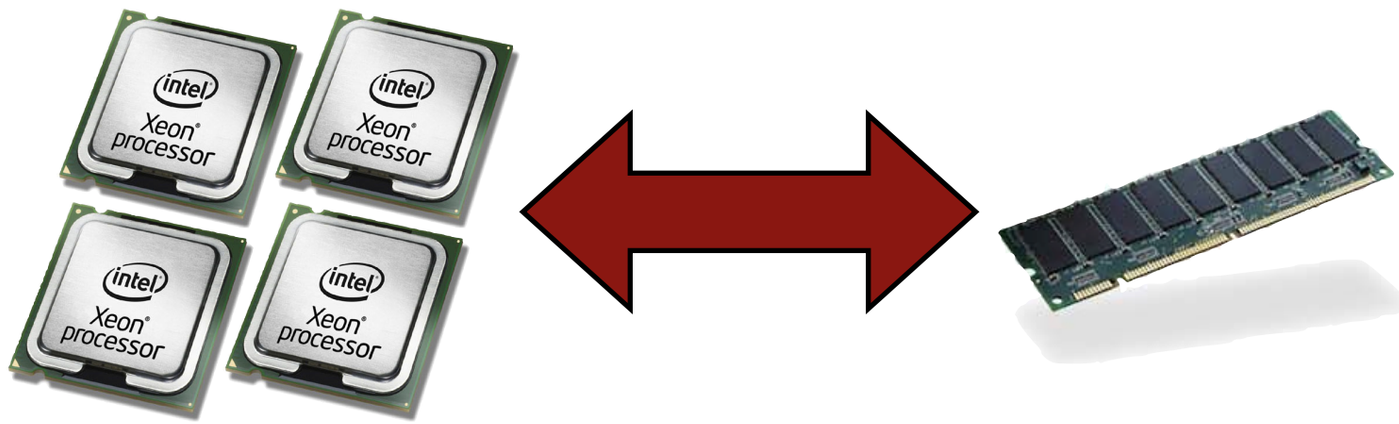
- Larger caches
  - 12 MB cache – Intel Core i7-970
- More functionality
  - 4 bit (1971: Intel 4004; 46 instructions)
    to 8 bit (1972: Intel 8008)
    to 16 bit (1978: Intel 8086)
    to 32 bit (1985: Intel 80386)
    to 64 bit (2003: AMD's Opteron; more than 500 instructions)
- More parallelism
  - Multicores (2001 onwards)

Multi-cores

- Multicore processors
  - One chip with multiple processors on it



- Symmetric multiprocessors (SMPs)
  - Machine with multiple CPUs sharing same memory



Why multi-cores?

- Eventually we will not be able to make smaller transistors.
  - Or, making them any smaller will not be cost effective.
- Chip manufacturers struggling to find new ways to maintain increases in processor performance.
  - Moore's Law has become a self-fulfilling prophesy
- One idea: Parallelization
  - Instead of speeding up individual operations, what about running multiple operations in parallel?

Performance gain with parallelism

- Parallelism can result in tremendous speedups!
  - Quad-core processor can do four times the work of a single-core processor in same amount of time
- But, some important limitations...
  - Can program be split across multiple parallel threads?
    - Can't just run a regular program and expect any speedup.
  - Memory is shared between processors...
    - Leading to possible contention
    - For example, if one CPU writes to a memory value being read by another CPU
    - Must use software techniques (e.g., locks) to avoid stomping on each other
    - This slows things down.

Dependencies

- Consider function to multiply array of integers
  - a points to array of integers of length n
  - Result stored in d

```
void arr_mult(int *a, int n, int *d) {
  int i;
  *d = 1;
  for (i = 0; i < n; i++) {
    *d = *d * a[i];
  }
}
```

- Runs slow. Why?
  - Accesses memory location d in every iteration
  - Cache helps, but still slow

Dependencies

- Old version

```
void arr_mult(int *a, int n, int *d) {
  int i;
  *d = 1;
  for (i = 0; i < n; i++) {
    *d = *d * a[i];
  }
}
```

- New version runs faster. Why?

```
void arr_mult2(int *a, int n, int *d) {
  int i;
  int t = 1;
  for (i = 0; i < n; i++) {
    t = t * a[i];
  }
  *d = t;
}
```

What this means for you...

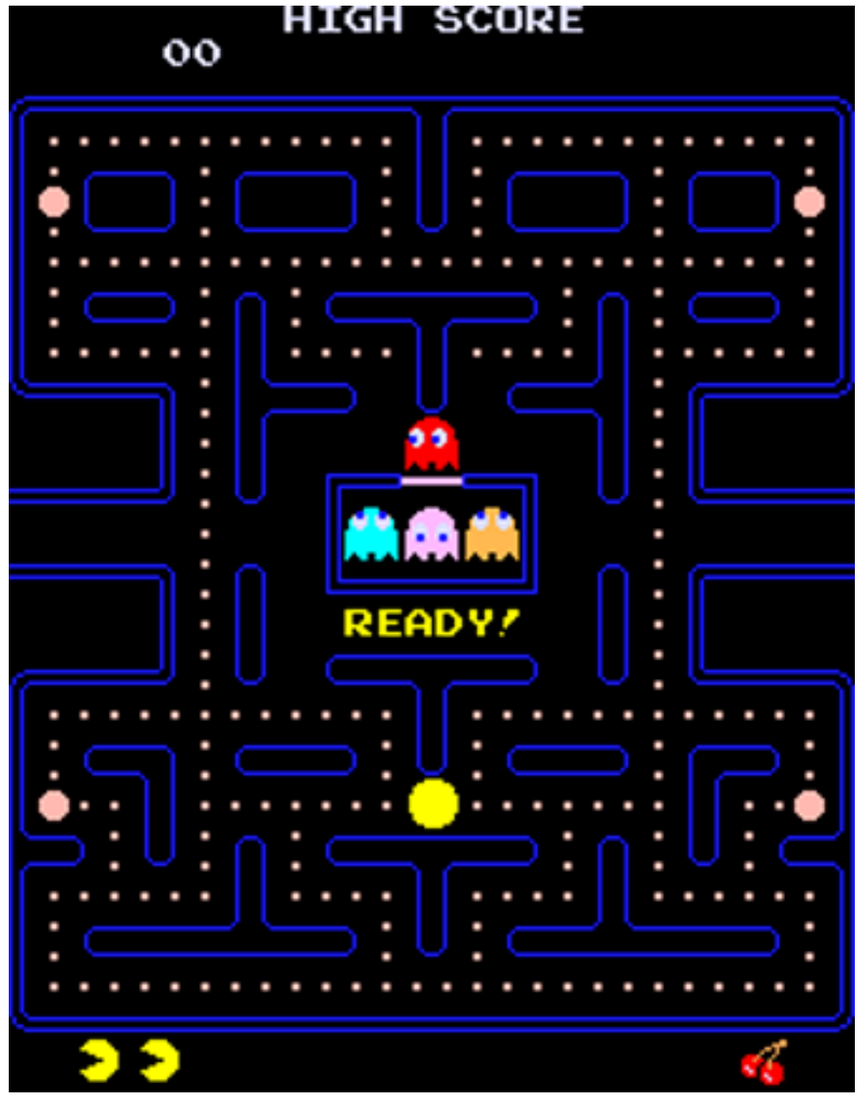
- Many things affect performance and reliability of programs
  - Memory hierarchy, caching
  - Concurrency and parallelism
  - Dependencies in program
  - Compiler optimizations
  - Buffer overruns
  - Operating system abstractions
  - ...
- These low-level details matter
  - Need to understand them to write robust efficient programs.
- This stuff is hard!
  - Mind-twisting (at first)
  - Easy to screw up!

Examples of screwing up…

- Mars Pathfinder: 1996
  - Lander and rover, cost about $265 million.
  - Subtle concurrency bug in the control software.
    - Turned out to be a classic case of **priority inversion**.
  - Software bug discovered and patched while the spacecraft was on Mars!

Examples of screwing up…

- Pacman: 1980
  - Classic video game!
  - After 255 levels, things get weird…



Examples of screwing up…

- Pacman: 1980
  - Classic video game!
  - After 255 levels, things get weird…
  - Level counter stored in 1 byte
  - 255 is highest number storable in 1 byte
    - Adding 1 to 255 gives… 0!
    - Called **overflow**
  - Bonus symbol drawing routine gets confused, draws more symbols than it should
    - **Buffer overflow**

Examples of screwing up…

- iPhone jail-breaking: 2010 (and others from 2007)
  - Apple only allows signed applications to run on iPhone
  - Hacking phone to allow unsigned applications to run called *jail-breaking*

  - Connect to a web site (jailbreakme.com)
  - Exploited **buffer overflow vulnerability** in font handling for PDF documents
  - Executed code that exploited another vulnerability to gain root access
  - (Lab 2 involves stack buffer overflows!)

Machine code

- Important to understand how computer represents and executes programs
  - May not be what you think!
- Ken Thompson "Reflections on Trusting Trust" 1984
  - Co-inventor of UNIX
  - Won Turing Award in 1983
  - During award lecture, revealed surprising exploit...

The Thompson Hack

- Put backdoor into login.c to allow easy access
  - Allow Ken Thompson access to any UNIX system
  - But login.c contains suspicious code…
- Modify compiler to insert code when it notices login.c is being compiled
  - login.c looks normal, no hack
  - But compiler contains suspicious code…
- Modify compiler to insert code when it notices compiler is being compiled!
  - login.c and compiler look normal
- Then delete compiler source code
- Result: source code of compiler and login.c is not what is executed…
  - Backdoor only observable in binary executable
- (Proof of concept implementation, did not distribute)

Hacking into my account...

- Say I left a program in my home directory that would run a shell as me if you gave it the right password.

```
% cd ~stephenchong
% ./scshell
Enter the password: **********
Congratulations! Running shell...

$ whoami
stephenchong
```

How would you figure it out?

- Brute force guessing? Won't work...

```
% cd ~stephenchong
% ./scshell
Enter the password: lameguess
Sorry, wrong!
Emailing President Faust…
Triggering Ad Board enquiry...

%
```

How would you figure it out?

- What if you could read the executable?

```
% cat scshell
ELF?4d4 ($!444??44444 HHH??(((  Q?
td/lib/ld-linux.so.2G  ?K?? )|??
893?.??__gmon_start__libc.so.
6_IO_stdin_usedgetsexitputsprintfQ
(,048<U??S???[??0????????t????DX
[???5 ?%$?%(h??????%,??????
%0h?????%4h????%8h ?????%<h(????
1?^????PTRh?h?QVh
\??????????????????U???=P
```

How would you figure it out?

- What if you could read the executable?

```
% od -x scshell
0000000 457f 464c 0101 0001 0000 0000 0000 0000
0000020 0002 0003 0001 0000 8380 0804 0034 0000
0000040 0e64 0000 0000 0000 0034 0020 0007 0028
0000060 0024 0021 0006 0000 0034 0000 8034 0804
0000100 8034 0804 00e0 0000 00e0 0000 0005 0000
0000120 0004 0000 0003 0000 0114 0000 8114 0804
0000140 8114 0804 0013 0000 0013 0000 0004 0000
```

Disassembly?

```
% objdump -d scshell
...
08048404 <check_password>:
 8048404:  55                      push   %ebp
 8048405:  89 e5                   mov    %esp,%ebp
 8048407:  83 ec 14                sub    $0x14,%esp
 804840a:  c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%ebp)
 8048411:  eb 2a                   jmp    804843d
 8048413:  8b 15 4c 97 04 08       mov    0x804974c,%edx
 8048419:  8b 45 fc                mov    -0x4(%ebp),%eax
 804841c:  8d 04 02                lea    (%edx,%eax,1),%eax
 804841f:  0f b6 10                movzbl (%eax),%edx
 8048422:  8b 45 08                mov    0x8(%ebp),%eax
 8048425:  0f b6 00                movzbl (%eax),%eax
 8048428:  38 c2                   cmp    %al,%dl
 804842a:  74 09                   je     8048435
...
```

Disassembly!

```
push   %ebp
mov    %esp,%ebp
sub    $0x14,%esp
movl   $0x0,-0x4(%ebp)
jmp    804843d
mov    0x804974c,%edx
mov    -0x4(%ebp),%eax
...
```

Put the contents of register %ebp on the stack

Copy the stack pointer to register %ebp

Subtract 20 bytes from the stack pointer

Copy the value 0 to local variable

Jump to address 0x804843d

Move the value 0x804974c to register %edx

Disassembly!

```
push   %ebp
mov    %esp,%ebp
sub    $0x14,%esp
movl   $0x0,-0x4(%ebp)
jmp    804843d
mov    0x804974c,%edx
mov    -0x4(%ebp),%eax
...
```

Put the contents of register %ebp on the stack
Copy the stack pointer to register %ebp
Subtract 20 bytes from the stack pointer
Copy the value 0 to local variable
Jump to address 0x804843d
Move the value 0x804974c to register %edx

- Looks complicated!
  - Until you take CS 61…
- What this? `mov 0x804974c,%edx` Looks interesting…
  - Looks like a memory address!

Disassembly

```
% gdb scshell
(gdb)
```

- I like to use gdb for examining memory (and disassembly)

Disassembly

```
% gdb scshell
(gdb) x 0x804974c
0x804974c <passwd>:  0x080485a0
(gdb)
```

- 0x804974c has the symbol “passwd”!
  - Looks like another address...

Disassembly

```
% gdb scshell
(gdb) x 0x804974c
0x804974c <passwd>:  0x080485a0
(gdb) x 0x080485a0
0x80485a0: 0x656b6154
(gdb)
```

- Hmmm, doesn't look like an address.
- Let's try examining the memory starting at 0x804805a0 as if it were a string...

Disassembly

```
% gdb scshell
(gdb) x 0x804974c
0x804974c <passwd>:  0x080485a0
(gdb) x 0x080485a0
0x80485a0: 0x656b6154
(gdb) x/s 0x080485a0
0x80485a0:  "Take CS61!"
```

- Bingo!

Success!!

```
% cd ~stephenchong
% ./scshell
Enter the password: Take CS61!
Congratulations! Running shell...


$
```

High score contest

- There is a binary in my home directory on nice.fas
  - Input 4 numbers on command line
  - It (deterministically) gives you back a score
  - Get the highest score you can
  - Need to disassembly it to figure out how to get high score…
    - Don't brute force! It emails me results…
- Check out CS 61 webpage for details and hints…
- Highest scores announced in next lecture

Why CS 61

- Learn how machines really work!
- Debug the hardest (and most interesting) bugs.
  - Stuff that only makes sense when you can read assembly
  - Use gdb and objdump like an expert
- Measure and improve the performance of your programs.
- Understand memory hierarchies, processor pipelines, and parallelism.
- Write concurrent, multi-threaded programs like a pro.
  - The basis for every application and server on the Internet today.

Where CS 61 fits into curriculum

- Prerequisites: CS50, CS51, or C programming experience
  - Need to know C for this course!
    - Must be comfortable with pointers, arrays, malloc/free, etc.
  - Talk to me if you are unsure if you're ready.
- This is an introduction to computer systems – not an "advanced" course.
  - Don't need to be a CS concentrator to take this class.
  - Will set you up for CS161 (OS), CS153 (compilers), and CS141 (architecture)

Where CS 61 fits into curriculum

- For CS concentrators
  - Require at least two of CS 50, CS 51, and CS 61
- CS as a secondary field
  - Counts as one of the four half-courses

CS 50 or CS 61?

- Take CS50 if…
  - You are new to programming
  - You have done only a little programming in C, C++, or Java
  - You have programmed only in Visual Basic or Python
  - You have never used UNIX before

- Take CS61 if…
  - You have taken CS50 or CS51 already
  - You have taken the CS AP (and done well)
  - You feel comfortable programming in Java, C, or C++

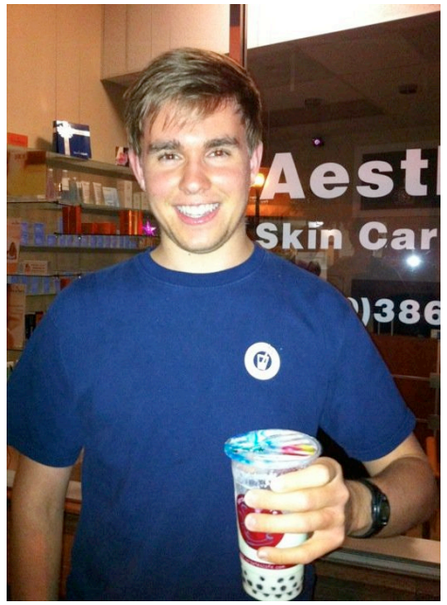
CS 51 or CS 61?

- Take both! They're complementary...
- CS51 focuses on concepts of program design, data structures, and algorithms
  - sets you up for later theory and programming classes
- CS61 is more "nuts and bolts" – how machines work
  - sets you up for later systems, architecture, and compiler classes

Lectures and sections

- Lectures Tues, Thurs 2:30pm-4pm
  - Attendance required!
- Weekly 90 minute sections
  - Attendance required!
  - 5 sections, times to be determined
    - More details next lecture

Teaching fellows



Jim Danz

Daniel Margo

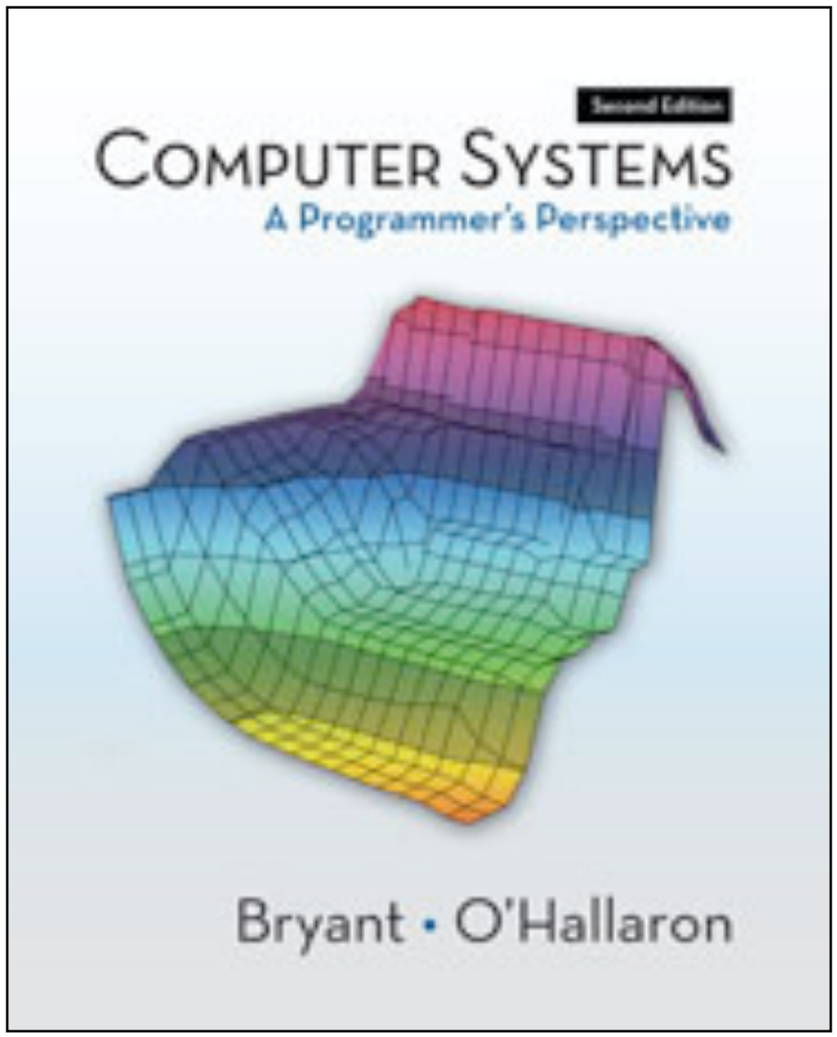
Stefan Muller

Nick Murphy

Jason Waterman (left)

Textbook

- ***Computer Systems: A Programmer's Perspective, Second Edition***
  by Bryant and O'Hallaron



- Required for course
  - Will use it heavily
- Second edition preferred
  - First edition usable with some supplemental reading

Syllabus

- Intel x86 assembly language programming
  - All about how the CPU works and how to get the best performance.
  - Basic operations: accessing data, arithmetic operations, etc.
  - Control flow (jumping and loops)
  - Procedures and the stack
  - Arrays and other data structures
- Performance optimization
  - Measuring the performance of programs
  - Optimizing C programs: strength reduction, common subexpression elimination, loop unrolling
- Linking and loading
  - How an executable is made, and how it starts running on the machine

Syllabus

- Memory management
  - Memory hierarchies, caches, virtual memory
  - Dynamic memory allocation
- UNIX systems programming
  - Dealing with files, pipes, signals, and processes
- Concurrency and threads
  - Synchronization: locks, monitors, semaphores, and condition variables
  - Synchronization problems and deadlock
- Cloud applications and concurrent servers
  - How are web servers, databases, cloud services, etc. really built.

Programming labs

- Lab 1: Binary Bomb
  - Defuse a bomb by typing the right password. But we give you no hints.
- Lab 2: Exploiting buffer overflows
  - Just like the Code Red worm that nearly took down the Internet!
- Lab 3: Dynamic memory allocation
  - Writing your own versions of malloc(), free(), and realloc().
- Lab 4: Write your own UNIX shell
  - Get into the gritty details of UNIX systems programming.
- Lab 5: Implementing a concurrent server
  - Build your own multithreaded Web application. Compete with Facebook maybe?

Extensions, collaborations

- Each student has five late days for labs
  - Can use at most 3 late days per lab
  - Extends deadline by 24 hours
  - **Late days must be requested before deadline!**
  - Labs not accepted after deadline
  - (Exceptions made, of course, for medical or other emergencies)
- Collaboration
  - Work in groups of up to size 2 for labs
    - May change group each lab
    - Late for group = late day for each member
  - All work expected to be your own.
  - Allowed to discuss problems, collaborate in thinking through and planning solutions.

Assessment

- Labs (~40%)
- Mid-term exam (~20%)
- Final exam (~25%)
- In-section quizzes (~10%)
  - Fairly frequent quizzes, no makeups
  - Lowest two scores dropped
- Participation (~5%)

Next week...

- GDB tutorial
  - Tues September 7, 4pm–5pm, in Pierce 307
    - Immediately after class
  - Assigned sections will start Tues 14 Sept.
- Next lecture: Intro to assembly!

Website

- Course website: **http://cs61.seas.harvard.edu/**
  - Labs
  - Sections
  - Lecture notes
  - Office hours
  - Announcements
  - Info about high score contest
  - Course policies
  - ...
- Questions? Talk to me!
  - Am I prepared for this course? Is this course suitable for me?