EECS 370 – Lecture 1

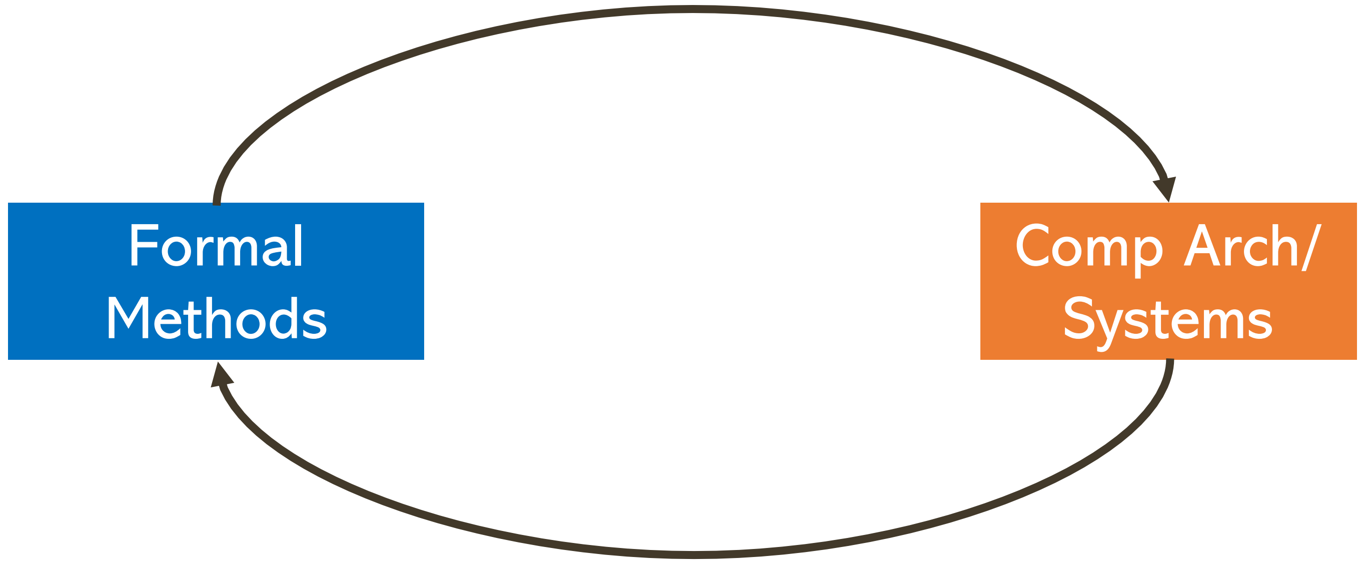
Introduction

Who am I?

- Prof. Manerkar ("Muh-nair-cur" is pretty close ☺)
- Faculty in CSE
- PhD from Princeton, Masters from UMich, undergrad from UWaterloo
- Research: intersection of formal methods and computer architecture
  - Formal methods: techniques based on rigorous mathematical analysis
  - Can automatically prove correctness (even across infinite scenarios!)
  - Can synthesize correct implementations
- Finding bugs in real systems can get a lot of attention!

My research in a nutshell

Bring more formal methods into
the architectural design process



Make formal methods community aware of
latest architectural advances, better system
support for formal methods

Who am I?

- Dr. Mark Brehob
  - Full-time teacher (lecturer)
    - Been here for 25 years and I've taught a wide variety of courses (100, 101, 203, 270, 281, 370, 373, 376, 452, 470, 473)
  - PhD is in the intersection of computer architecture and theoretical computer science as it relates to caches.
  - See http://web.eecs.umich.edu/~brehob/

Who are you?

Class resources

- Course homepage: eecs370.github.io/
  - All assignments will be posted here.
  - Also links for administrative requests (SSD, Medical emergencies, etc.)

- Ed: edstem.org
  - Use for general questions on lectures, projects and homework assignments. Can discuss with your classmates.

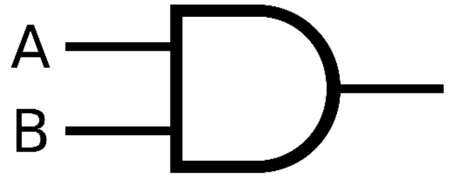
- Gradescope: https://www.gradescope.com
  - Turn in homework assignments.
  - Contact us if you don't have access

For other issues, submit admin form linked on website

Where does EECS 370 fit in our curriculum?

- Software view
  - EECS 183/ENGR 101, EECS 280, EECS 281
  - Turning specs into high-level language
- Hardware view
  - EECS 270, **EECS 370**
  - gates $\rightarrow$ logic circuits $\rightarrow$ computing structures
- Prereqs:
  - C or C++ programming experience (EECS 280)
  - Basic logic (EECS 203 or EECS 270)
- Projects are in C — not C++

Logic done quickly



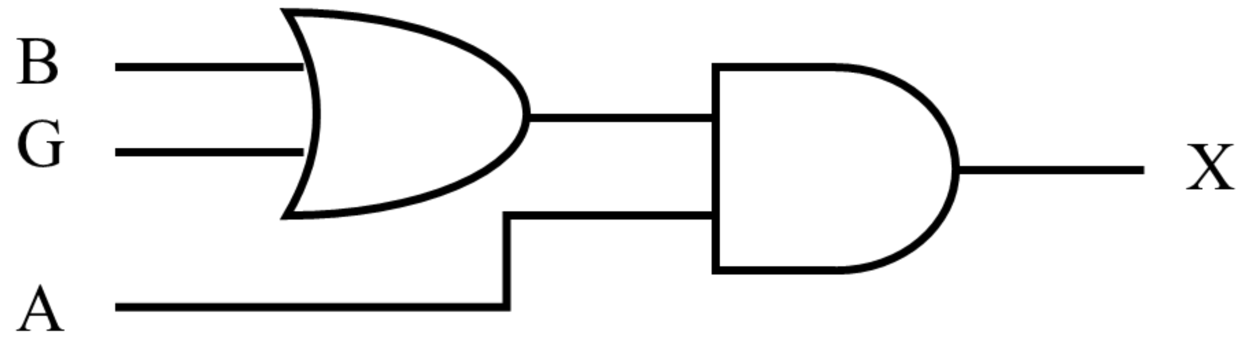
	Math/ Philosophy	Engineering	Gate
A and B			
A or B			
not A			
A xor B			

A	B	Out
F	F	
F	T	
T	F	
T	T	

Example:
X: Get an apple and either grapes or a banana

- $X = A \wedge (G \vee B)$

- X=A*(G+B)



A	G	B	Out
F	F	F	
F	F	T	
F	T	F	
F	T	T	
T	F	F	
T	F	T	
T	T	F	
T	T	T	

A	G	B	Out
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Goals of the course

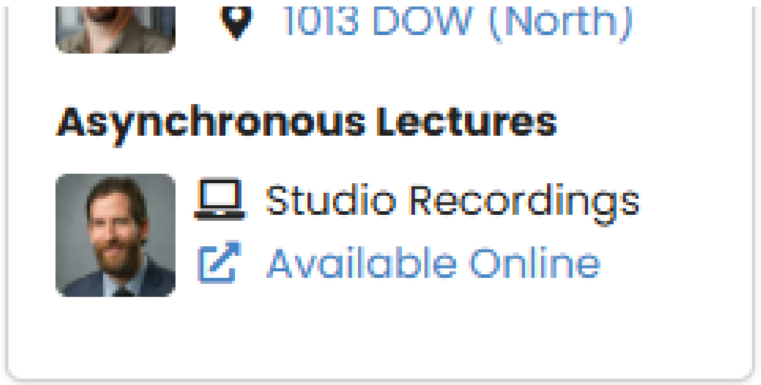
- To understand how computer systems are organized and what tradeoffs are made in the design of these systems
  - Instruction set architecture
  - Processor microarchitecture
  - Memory systems

Course Logistics: Lecture

- In line with the general CoE and University policies, default mode of instruction is **in-person**
- Lectures (Tu/Th)
  - Attendance is not required
  - Feel free to ask questions at anytime!
    - Interaction is highly encouraged (and critical!)

Course Logistics: Studio Recordings

- These should be viewed as a supplement to the class, not a replacement for the lectures.
  - They are a bit dated and coverage continues to change.
  - You are responsible for the material covered in lecture.



1013 DOW (North)

Asynchronous Lectures

Studio Recordings

Available Online

Course Logistics: Labs!

- Labs (M, Tu, or Fr)
  - Work on assignments during lab
  - Turn in hard copy at end of lab period
  - Assignment graded for correctness
  - 20% of lab grade comes from doing a lecture quiz online (more details later)
    - Due 11:55 pm Thursday **before** the lab.
  - **Attendance required to get credit (waived for lab 1)**
  - **Fill out lab survey to let us know if you can't make your section**
- **Labs start this week!**

Course Logistics: Assignments and Exams

- Programming assignments (40%)
- Homework assignments (5% total)
  - Total of 4 homework assignments, no drops
- Labs (5% total)
  - Total of 12, drop 2 lowest
  - Part of each lab is doing lecture quiz on Gradescope beforehand
- One midterm and a final exam (50% total)

<u>In-person only</u>

• *Midterm*	*Thursday, March 12th*	*7:00pm - 9:00pm*
• Final	Thursday, April 23rd	10:30 am - 12:30 pm

Course Logistics: Programming assignments

- 4 programming assignments simulating the execution of a simple microprocessor
  - Assembler / functional processor simulation
  - Linker
  - Pipeline simulation
  - Cache simulation

- Using C to program, C is a subset of C++ without several features like classes
- The challenge is to understand computer organization enough that you can build a complete computer emulator

Auto-grading assignments

- We use a program to grade your programming assignments
  - Program submitted using autograder.io
- **Assignments due at 11:55 pm on due date**.
  You may use late days just for projects over the course of the semester
- Help on C available from staff

Course Logistics: Academic integrity

Suspected code copying reported to Honor Council

Each year we hope to have zero cases. Some terms we report almost 10% of the class.

- We want you to collaborate as you learn!
  - But **DON'T SHARE CODE**

DO	**DO NOT**
Share high level strategies	Share code
Help someone debug	Debug for someone
Explain compiler errors to someone	Fix someone's compiler error
Discuss test strategies	Share test cases

See the website for more details and examples.

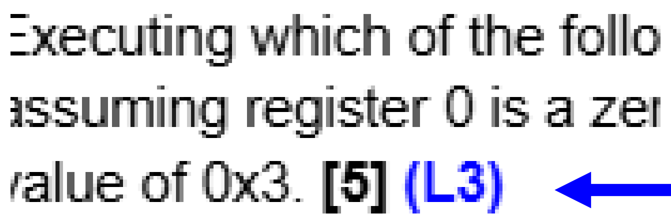
AI Tools (e.g. ChatGPT)

Coolguy Whit – drawception.com

- ChatGPT, Co Pilot, and similar things are great tools!
- Think of it as another student:
    - A great resource to ask questions to
    - It might sometimes be wrong
    - Taking work that it wrote and representing it as your own is an honor code violation
- So feel free to use it as a starting point, verify any answers it gives, and submit your own work
    - **All code you submit should be your own!**

Course Logistics: Homework

- There are 4 homework assignments
    - The assignments have 0 to 2 questions per lecture.
    - Each problem is labeled with the lecture number after which you should be able to answer it.



Executing which of the follo
assuming register 0 is a zer
value of 0x3. **[5]** (L3)

Lecture 3

- We'd recommend doing the homework as you go and then reviewing it before you turn it in.

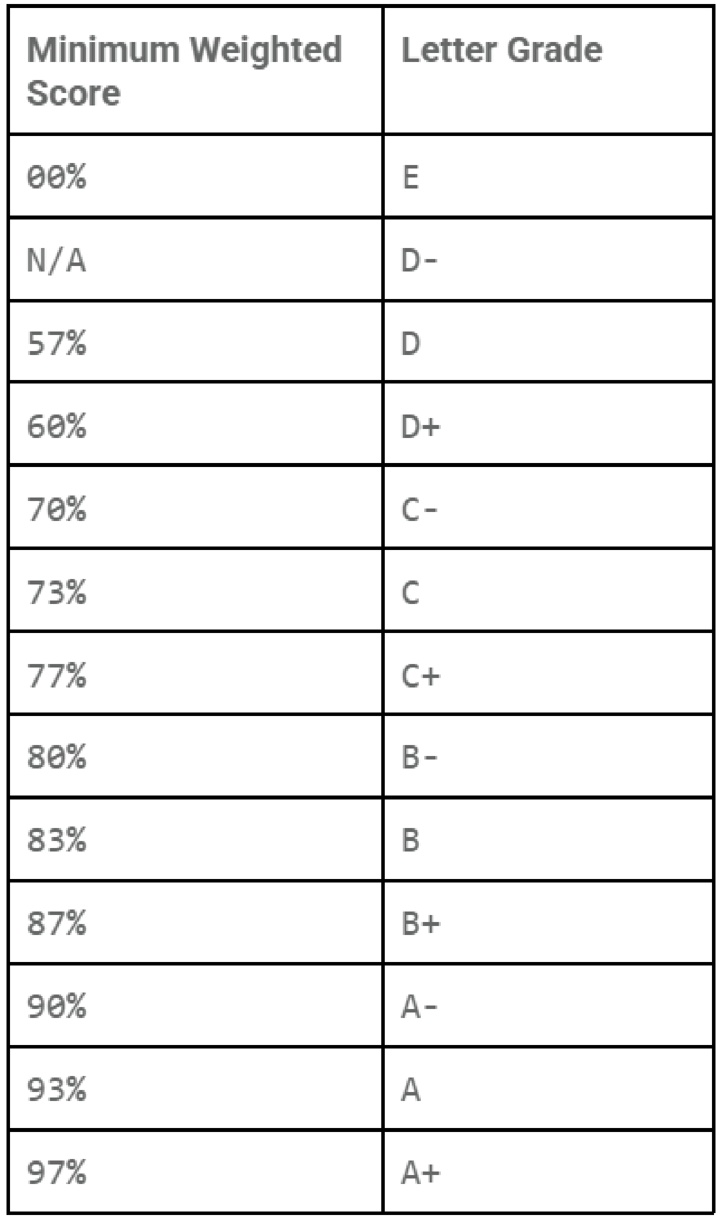
Course Logistics: Admin Requests

- If anything comes up that may interfere with your work in the class (e.g. illness, family emergency, etc) fill out the admin form on the course website
  - We'll probably instruct you to use late day / drop, but it gets it on the record in-case something comes up later

Course Logistics:
How we assign course grades

- Grade on a straight scale
- We may adjust this in your favor if exams are more difficult than expected
- Average is usually ~B-B+

- **<u>Require a 50% or higher on exams to pass</u>**



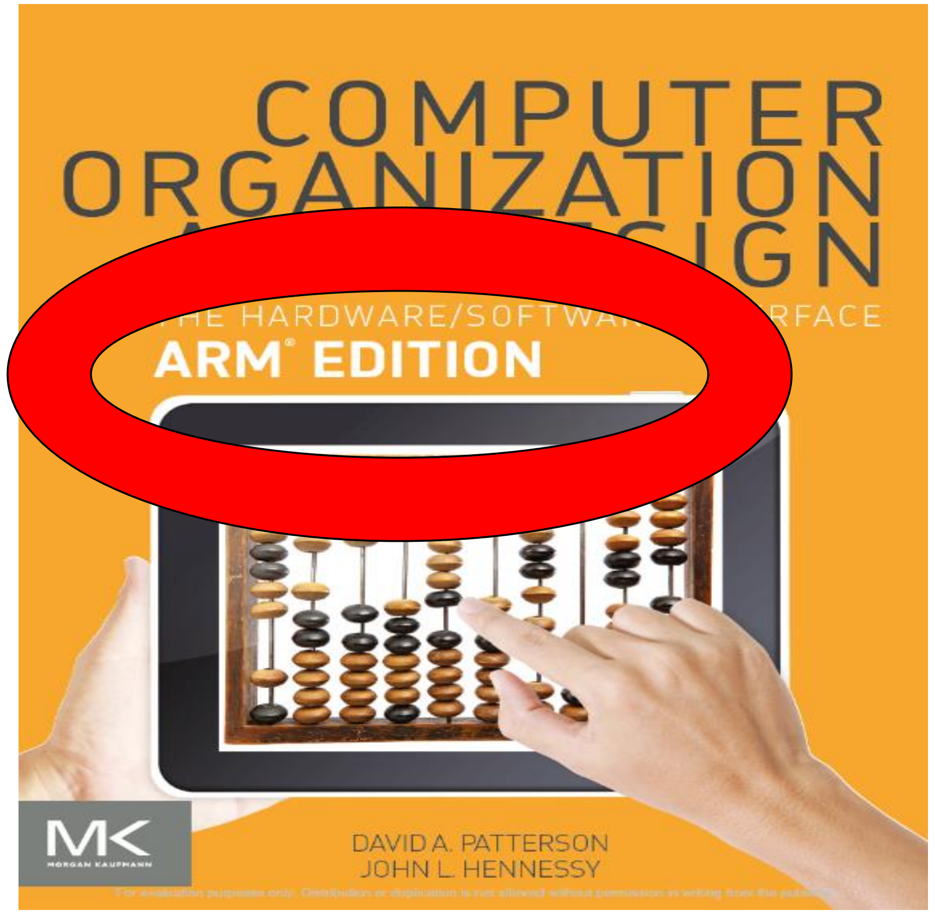
Minimum Weighted Score	Letter Grade
00%	E
N/A	D-
57%	D
60%	D+
70%	C-
73%	C
77%	C+
80%	B-
83%	B
87%	B+
90%	A-
93%	A
97%	A+

Course Logistics: Course textbook

Computer Organization and Design
ARM Edition

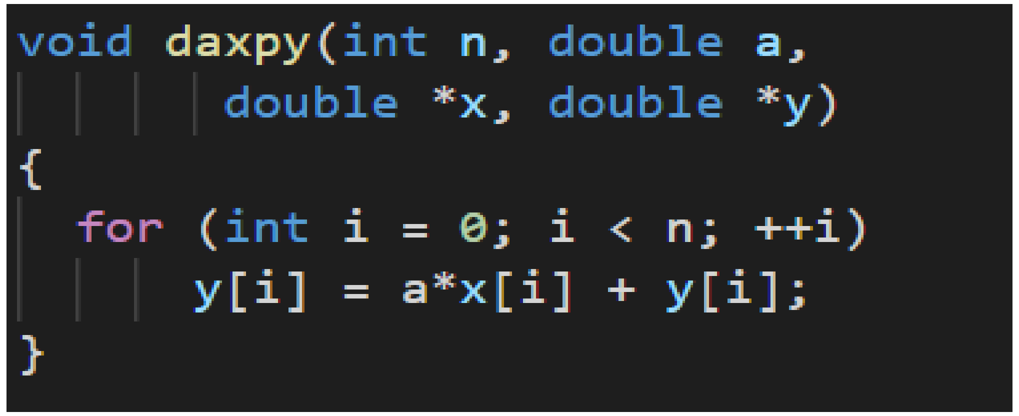
by Patterson and Hennessy

- Not required, but it's good



What is 370 about?

- You're used to writing programs like this



```
void daxpy(int n, double a,
           double *x, double *y)
{
  for (int i = 0; i < n; ++i)
      y[i] = a*x[i] + y[i];
}
```

- But what's actually happening ***inside*** the computer when we compile / run this?
  - Come to think it… what does "compiling" even mean??

- 370 in a nutshell: **How do computers execute programs?**

You will need to know this stuff if…

- You work in designing processors at Intel, ARM, NVIDIA, etc
- You write optimized library code
- You work on designing operating systems or compilers
- You work in computer security

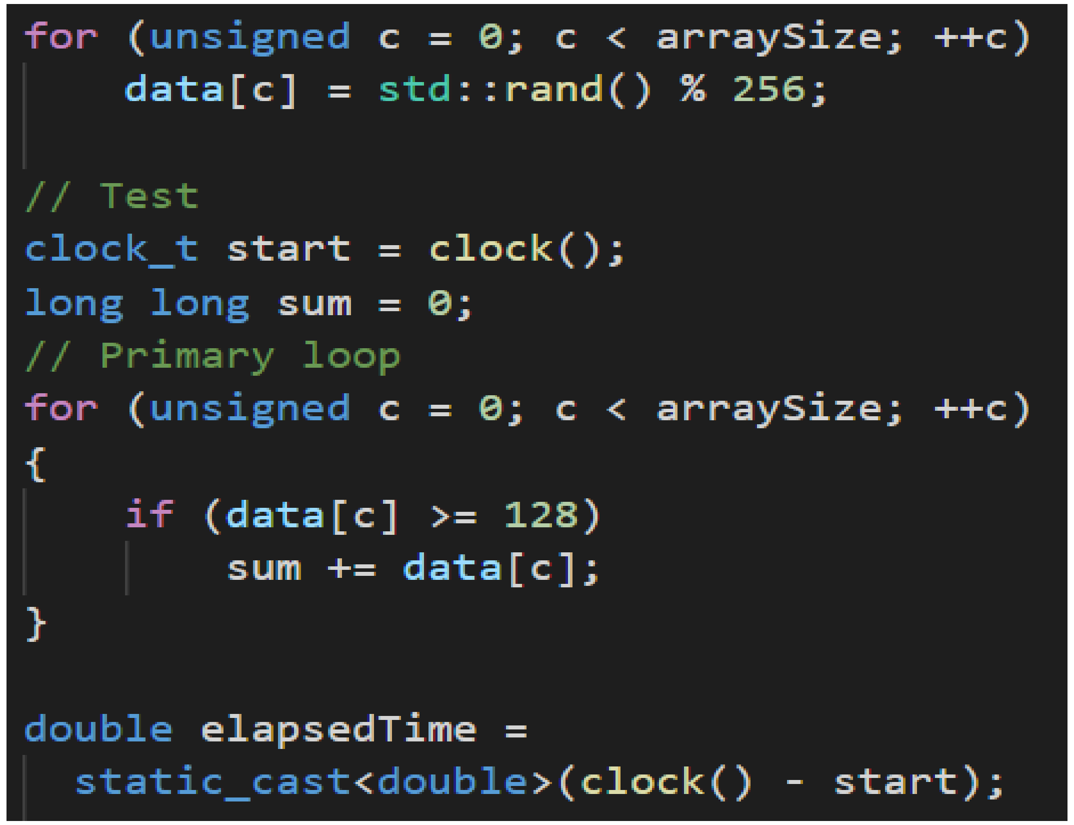
- You work in designing embedded systems (IOT, etc.)
- You want to be at the forefront of AI systems

You might need to know this stuff if…

- Even if you just write software for the rest of your life
  - Important to know what your computer is doing when it executes your code!
  - It can make a big difference in your performance
- Our favorite example comes from the #1 StackOverflow question of all time…

Example

- Consider 1 version of code that loops through an array of random values and adds all elements larger than 128

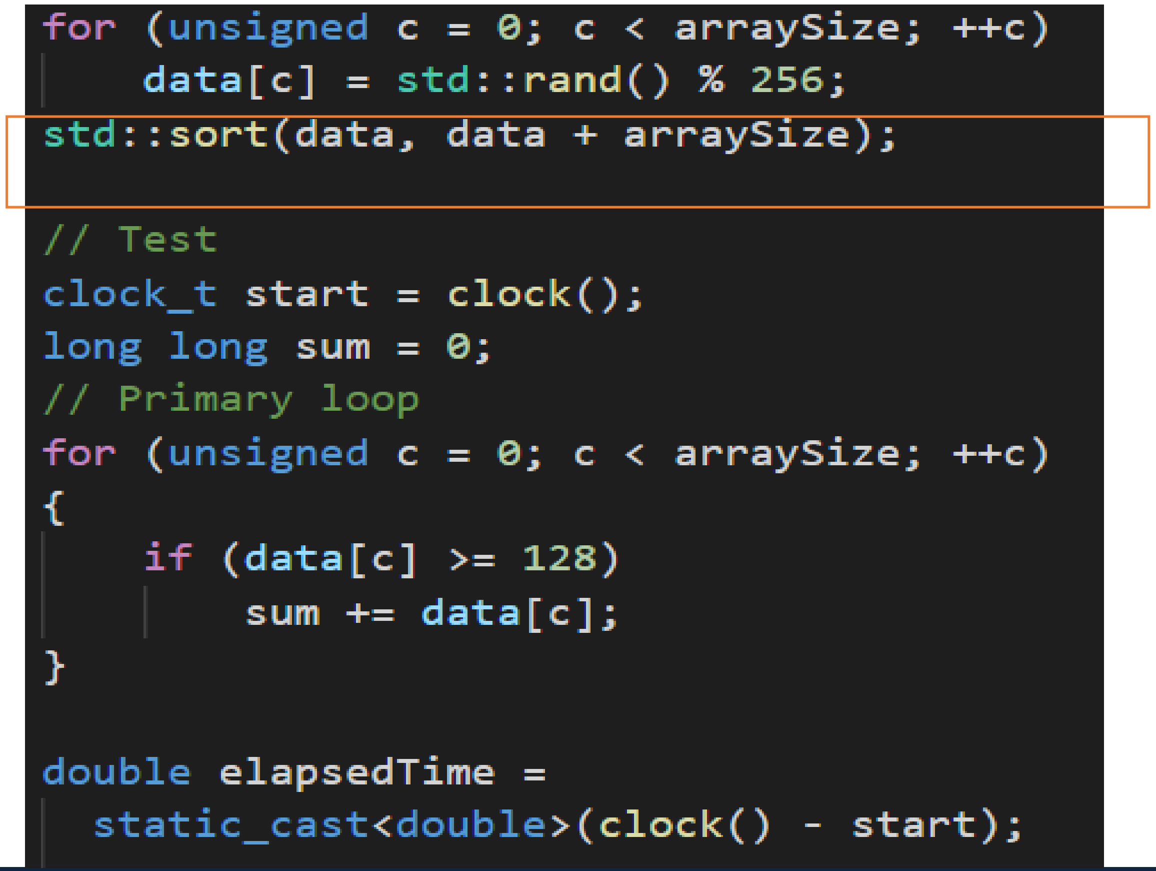


```
for (unsigned c = 0; c < arraySize; ++c)
    data[c] = std::rand() % 256;

// Test
clock_t start = clock();
long long sum = 0;
// Primary loop
for (unsigned c = 0; c < arraySize; ++c)
{
    if (data[c] >= 128)
        sum += data[c];
}

double elapsedTime =
  static_cast<double>(clock() - start);
```

https://stackoverflow.com/questions/11227809/

Example

- What will happen to the execution time of the loop if we sort the array beforehand?



```
for (unsigned c = 0; c < arraySize; ++c)
    data[c] = std::rand() % 256;
std::sort(data, data + arraySize);

// Test
clock_t start = clock();
long long sum = 0;
// Primary loop
for (unsigned c = 0; c < arraySize; ++c)
{
    if (data[c] >= 128)
        sum += data[c];
}

double elapsedTime =
  static_cast<double>(clock() - start);
```

What do you think will happen?

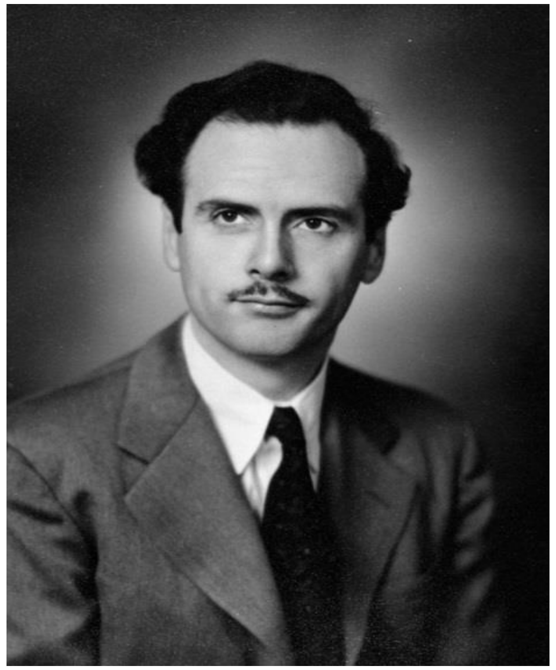
A. Sorted array sums much faster

B. Sorted array sums much slower

C. No significant difference between sorted and unsorted arrays

Why take 370?

- Inherent value in understanding how the tools we use work



"We shape our tools and thereafter
our tools shape us"

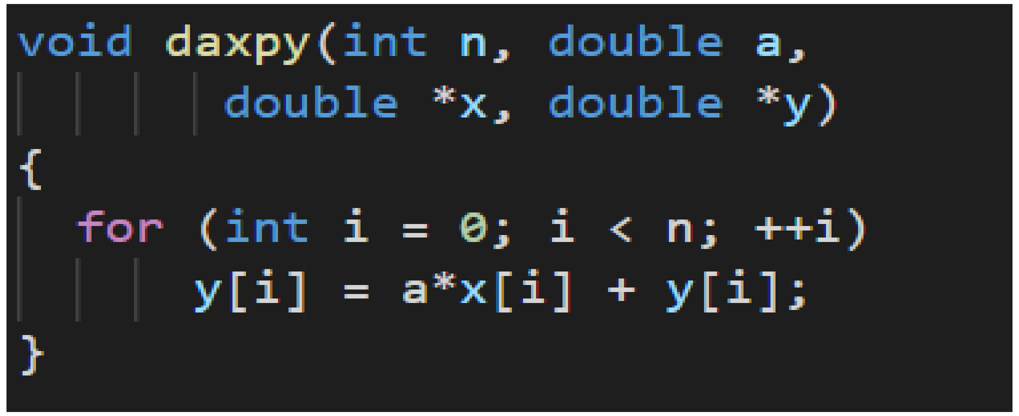
- Marshal McLuhan

Course overview

- Introduction (this lecture)
- ISAs and Assembly (~6 lectures)
- Processor implementation (~9 lectures)
- Memory (~7 lectures)
- Exams/catch-up (3 lectures)

How programs work in a nutshell

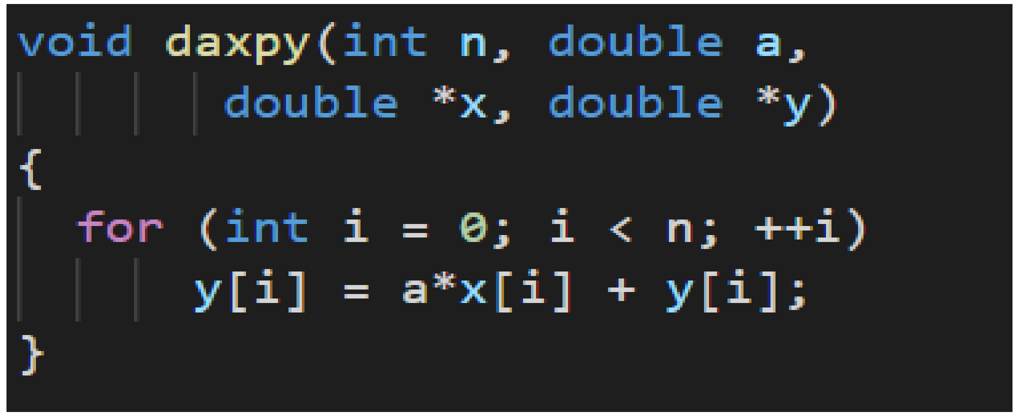
- We're used to seeing programs like this
- But it's difficult to do all this in hardware efficiently
  - Tons of different keywords and variable names…
  - Matching up different parentheses
  - Do we have to hardwire all this in logical circuits???



```
void daxpy(int n, double a,
           double *x, double *y)
{
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}
```

How programs work in a nutshell

- High level languages are intended to be easy for humans to understand / write
  - **NOT** for hardware to execute
- To be executed, must **compile** this complicated code into a set of **very simple** and easy to understand instructions that hardware can execute efficiently

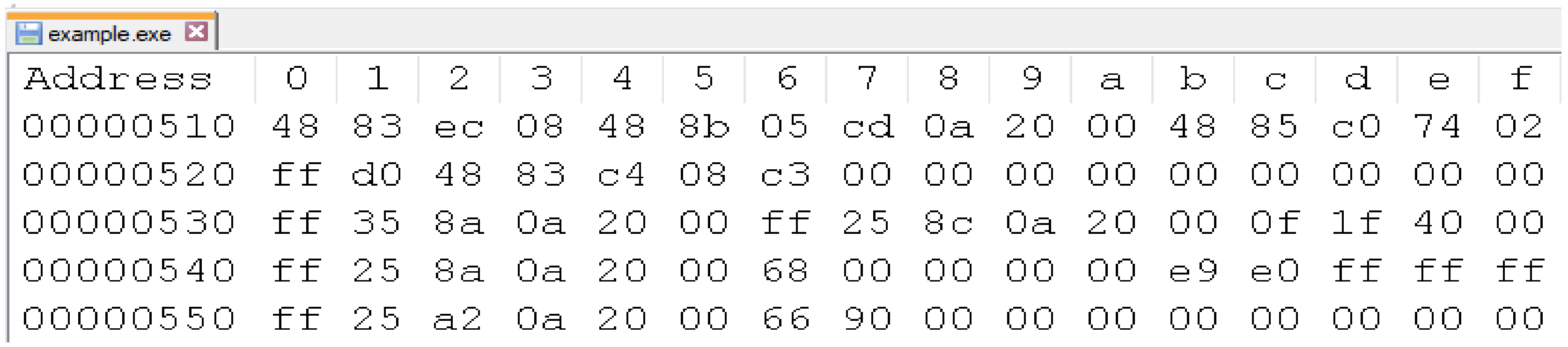


```
void daxpy(int n, double a,
           double *x, double *y)
{
  for (int i = 0; i < n; ++i)
      y[i] = a*x[i] + y[i];
}
```

How programs work in a nutshell

$xxd –b [file]
will show you the binary contents… not very useful

- THIS is what computers can understand and execute



example.exe

```
Address   0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00000510 48 83 ec 08 48 8b 05 cd 0a 20 00 48 85 c0 74 02
00000520 ff d0 48 83 c4 08 c3 00 00 00 00 00 00 00 00 00
00000530 ff 35 8a 0a 20 00 ff 25 8c 0a 20 00 0f 1f 40 00
00000540 ff 25 8a 0a 20 00 68 00 00 00 00 e9 e0 ff ff ff
00000550 ff 25 a2 0a 20 00 66 90 00 00 00 00 00 00 00 00
```

- This is called "machine code": just a bunch of 0s and 1s (easier for us to read if we convert it into hex digits)
- It's what's produced when you type:

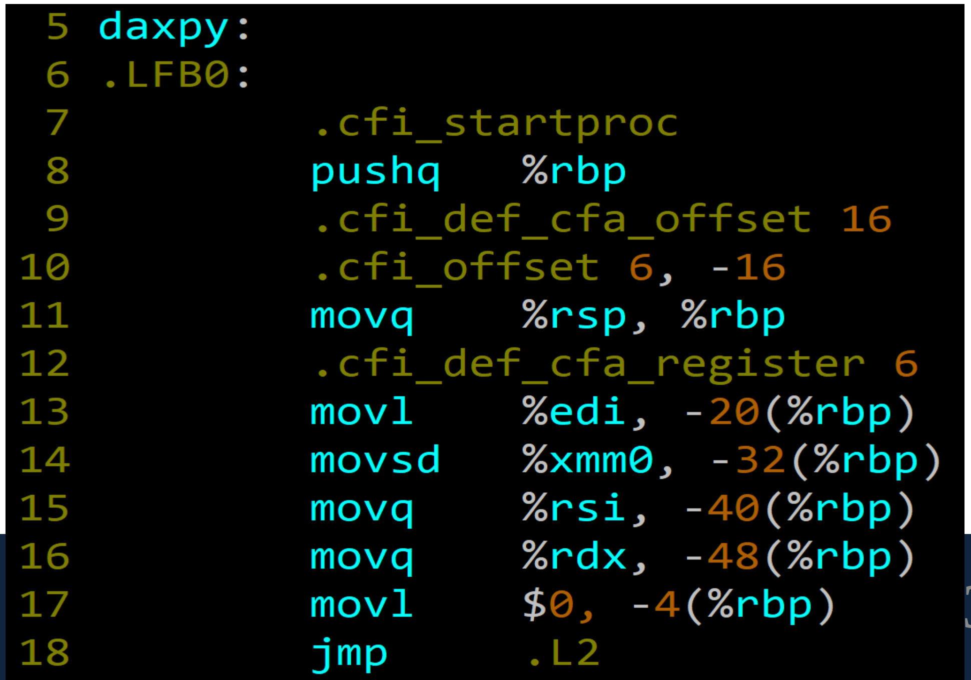
  g++ example.c -o example.exe

- Early programmers literally programmed by flipping switches on and off

How programs work in a nutshell

- Humans sometimes work at an intermediate level by writing **assembly code**
- Usually has a 1-1 correspondence with machine code instructions
  - Gives the programmer fine control over the final executable
- But it's (relatively) easy to read
- You can view generated assembly code with -s flag in g++:

  g++ -s example.c

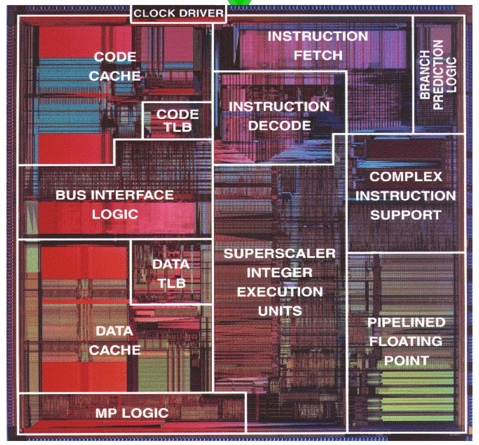


```
daxpy:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    %edi, -20(%rbp)
        movsd   %xmm0, -32(%rbp)
        movq    %rsi, -40(%rbp)
        movq    %rdx, -48(%rbp)
        movl    $0, -4(%rbp)
        jmp     .L2
```

Source Code to Execution (simplified)

$$\begin{tikzcd}
\boxed{\text{C, C++, etc.}} \arrow[r, "\text{Compiler}"] & \boxed{\text{Assembly}} \arrow[r, "\text{Assembler}"] & \boxed{\text{Executable}} \arrow[d]\\
 & & \text{}
\end{tikzcd}$$

- There are some things missing here… we'll fill those in later
- First quarter of the course will cover this process
- Remainder of the class will be… how do you build this thing?? (and memory)



Architectures

- Not just one type of machine code produced for all types of computers
- Just like how there are several different programming languages (C/C++, Java, Python, etc)…
  - there are also many different types of **architectures** that code can be compiled to run on
- Popular architectures:
  - x86, ARM, RISC-V
- Code compiled for one architecture will not run on another

x86

- Designed by Intel (AMD designed 64-bit version)
- Beefy, complex, fast, power-hungry
- Used in:
  - Desktops
  - Most laptops
  - Servers
  - PlayStation 4/5, Xbox One

ARM

- Designed by… Arm
- Versatile: can be used for higher performance or low-power usage
- Used in:
  - Most smartphones
  - Recent Macbooks
  - Recent supercomputer clusters
  - Nintendo Switch

CRAY CS500 HIGH PERFORMANCE FUJITSU A64FX ARM V8.2-SVE SERVER
• 2 nodes per blade and 8 nodes in a 2U Chassis
• First HPC processor with direct attached high bandwidth memory (HBM2)
• Accelerating compute for memory intensive HPC workloads
CRAY

RISC-V

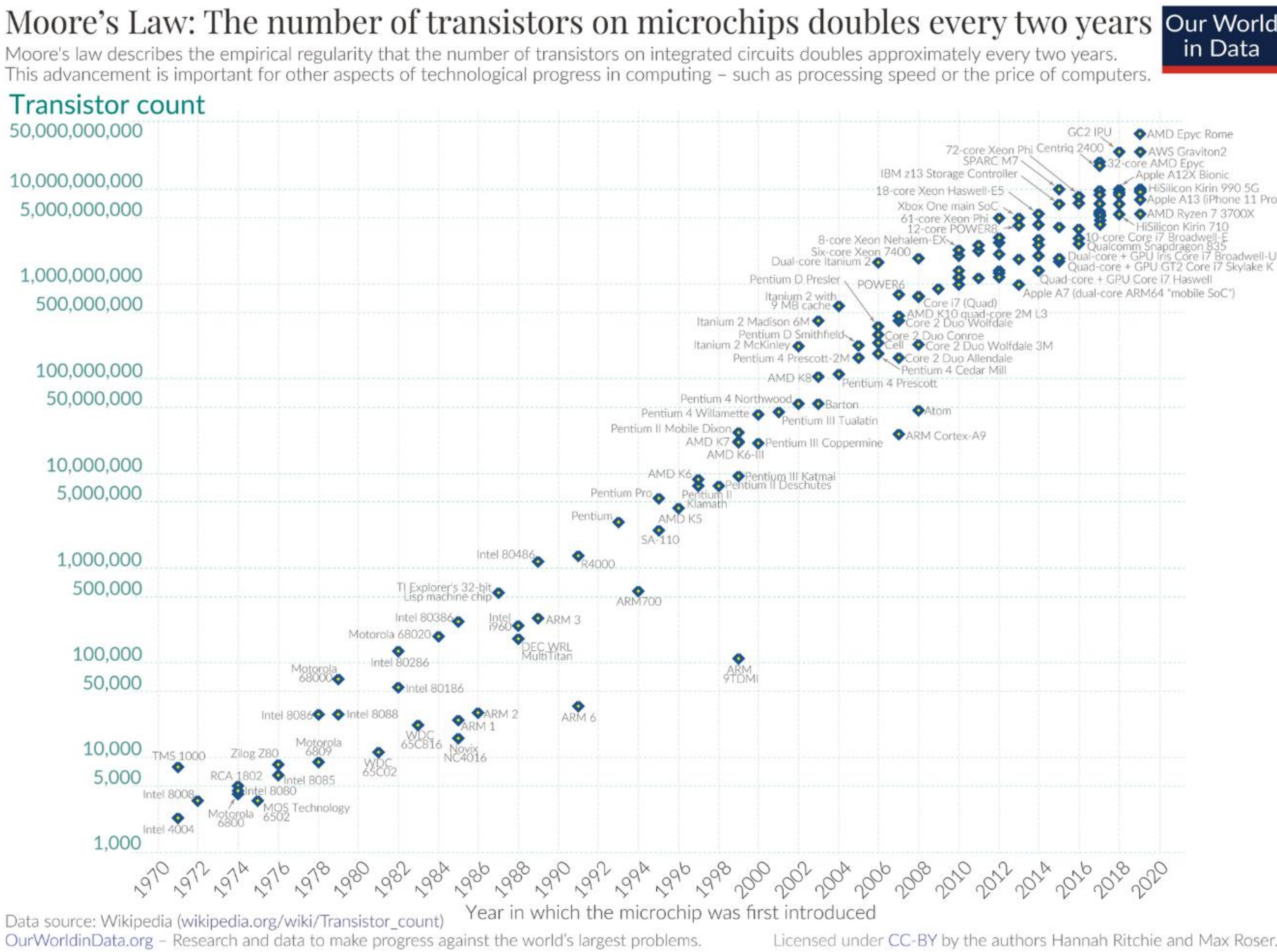
- Open source
- Very popular in academia
  - Don't need to pay super-expensive licensing fees
- Starting to make its way into actual products

Architectures Discussed in this Class

- We primarily focus on:
  - A subset of ARM called "LEG" (hardy-har-har)
  - A made-up ISA we call LC2K (Little Computer 2000)
    - Extremely simple, lets us focus on the concepts
    - Not practical for real applications

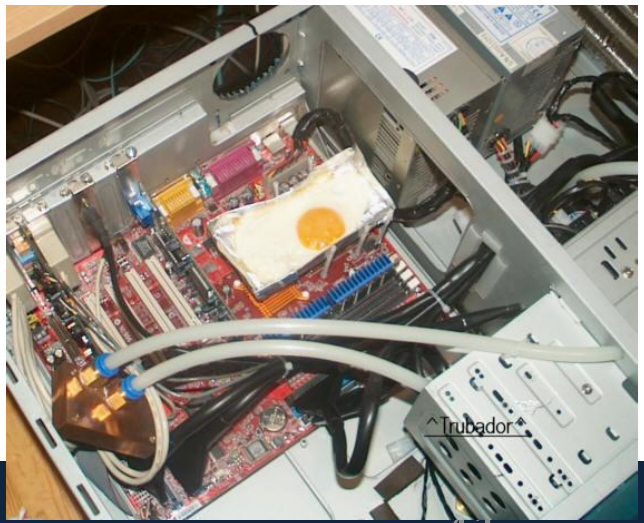
The Trend

- Moore's Law



The End of Moore's Law?: Dennard Scaling

- Dennard Scaling: as transistors get smaller their power density stays constant
- Translation: as the number of transistors on a chip grows (Moore’s Law), the power stays roughly constant
- Mid-2000’s Dennard Scaling broke. Why? Transistors got so small that they began to leak a lot of power. Leaking lots of power caused a chip heat up a lot.
- Conclusion: you can put lots of transistors on a chip, but you can’t use them all at full power at the same time.
  - You'll melt the processor!
- This is why newest processors focus on having multiple **cores**
  - More energy-efficient, but you have to parallelize your code!
  - Take EECS 570 to learn more ☺



Reminders

- Make sure you have a CAEN account!
- Lab 1 starts this Friday
  - learn about C programming, debugging methods and tools, and more

Next Time

- Introduce Instruction Set Architectures (ISAs)