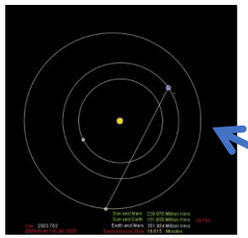
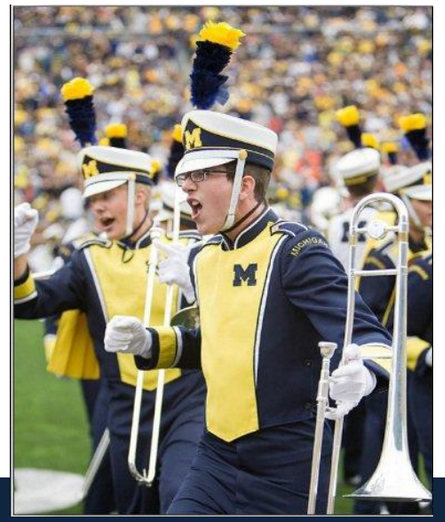
EECS 370 – Lecture 1

Introduction

Who am I?

- Bachelors (CE), Masters (CSE), and PhD (CSE) all from Michigan!
- Research in computer architecture
  - How to make devices like Graphics Processing Units (GPUs) better at non-graphics stuff?
- Currently work on curriculum design for quantum computing





Who am I?

- When not doing school stuff, you can find me:
  - Triathlon training
  - Cooking
  - Designing retro games / emulators
- Call me:
  - Jon
  - Dr / Prof Beaumont
  - “Yo Teach!”

Who are You?

<u>Poll:</u> Who are you!

I'll make use of polls during class – these are optional

I have you log in with your name because it helps me learn names

(you can also ask questions asynchronously on Slido)

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

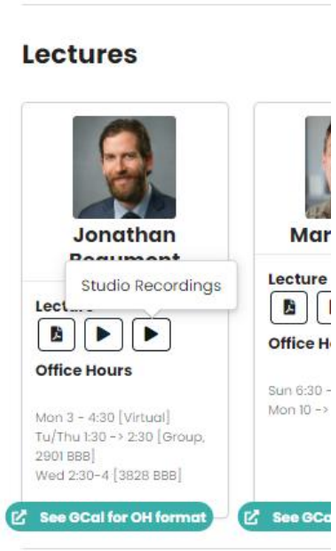
For other issues, submit admin form linked on website

Course Logistics

- 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!)
- Office hours may have some remote options, but mostly in-person
  - Will be posted on website

Async Recordings

- If you're watching lectures asynchronously…
- I have studio recordings
  - Much better quality than lecture recordings
  - I won't walk off screen, etc



Labs!

- Labs (M or F)
  - Work on assignments during lab
  - Turn in hard copy at end of lab period
    - Lab 1 is exception – have until Sunday night to turn in electronically
  - 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**

Your work in 370

- Programming assignments (40%)
- Homeworks (5% total)
  - Total of 4 homeworks
- Labs (5% total)
  - Total of 11, drop 1 lowest
- One midterm and a final exam (50% total)
  - **In-person**
  - Midterm – Thu October 9th, 6pm – 8 pm
  - Final – Thu Dec 11th, 10:30 am-12:30 pm

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 (mostly) subset of C++ without several features like classes

- Submitted to autograder.io
  - You'll know your score, but not what the tests are doing
  - Late days available at your discretion

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

Academic integrity

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

Suspected code copying reported to Honor Council

– usual penalty is 0 on assignment and 1/3 letter grade reduction in course

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

AI Tools (e.g. ChatGPT)

Coolguy Whit – drawception.com

- ChatGPT is a great tool!
- 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

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+
- **Require a 50% or higher on exams to pass**
  - (This is different than previous semesters, for folks retaking the class)

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 textbooks

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 computer's 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
- My 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?

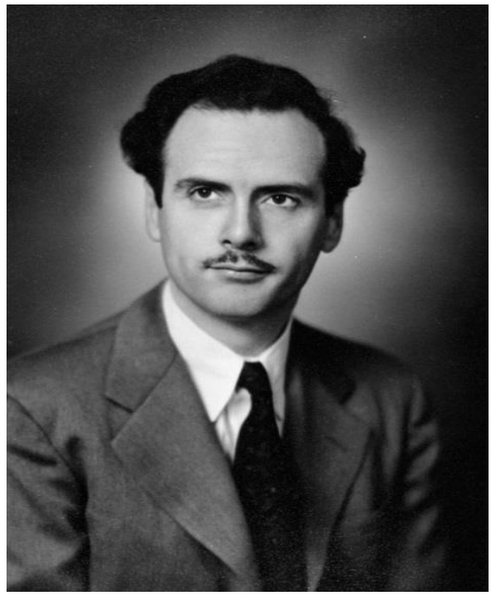
```
for (unsigned c = 0; c < arraySize; ++c)
    data[c] = std::rand() % 256;
std::sort(data, data + arraySize);

// 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);
```

Poll: 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
D. I have no idea!

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

Let's take a break

- In the meantime…
- What processor are you using?
  - Windows: Control Panel > System and Security > System
  - Mac: Click top-left Apple icon > About this Mac
  - Linux: lscpu
- How many cores does it have?
  - Windows: Ctrl-Shift-Esc > Performance

Poll: What hardware do you have with you today?

How programs work in a nutshell

- We're used to seeing programs like this
- But computer hardware is limited and can't understand code this complicated
  - 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*

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

**Another option is interpreting programs, like is done in Python. We won't cover that in this class*

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)
- Early programmers literally programmed by flipping switches on and off
- It's what's produced when you type:

  g++ example.c -o example.exe

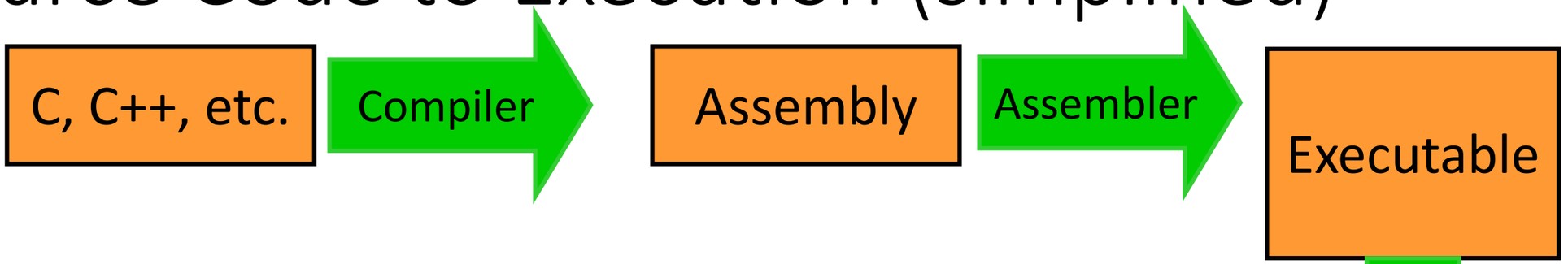
How programs work in a nutshell

- Humans often 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++:

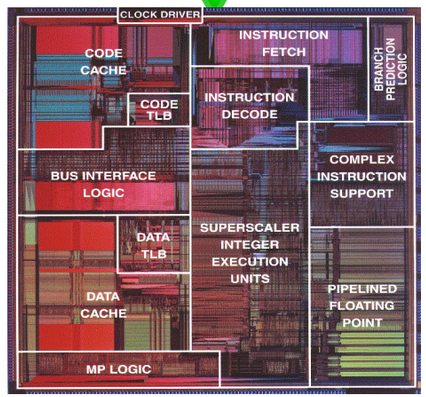
  g++ -s example.c

```
 5 daxpy:
 6 .LFB0:
 7         .cfi_startproc
 8         pushq   %rbp
 9         .cfi_def_cfa_offset 16
10         .cfi_offset 6, -16
11         movq    %rsp, %rbp
12         .cfi_def_cfa_register 6
13         movl    %edi, -20(%rbp)
14         movsd   %xmm0, -32(%rbp)
15         movq    %rsi, -40(%rbp)
16         movq    %rdx, -48(%rbp)
17         movl    $0, -4(%rbp)
18         jmp     .L2
```

Source Code to Execution (simplified)



- 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 non-Apple laptops
  - Servers
  - PlayStation 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 (2)

RISC-V

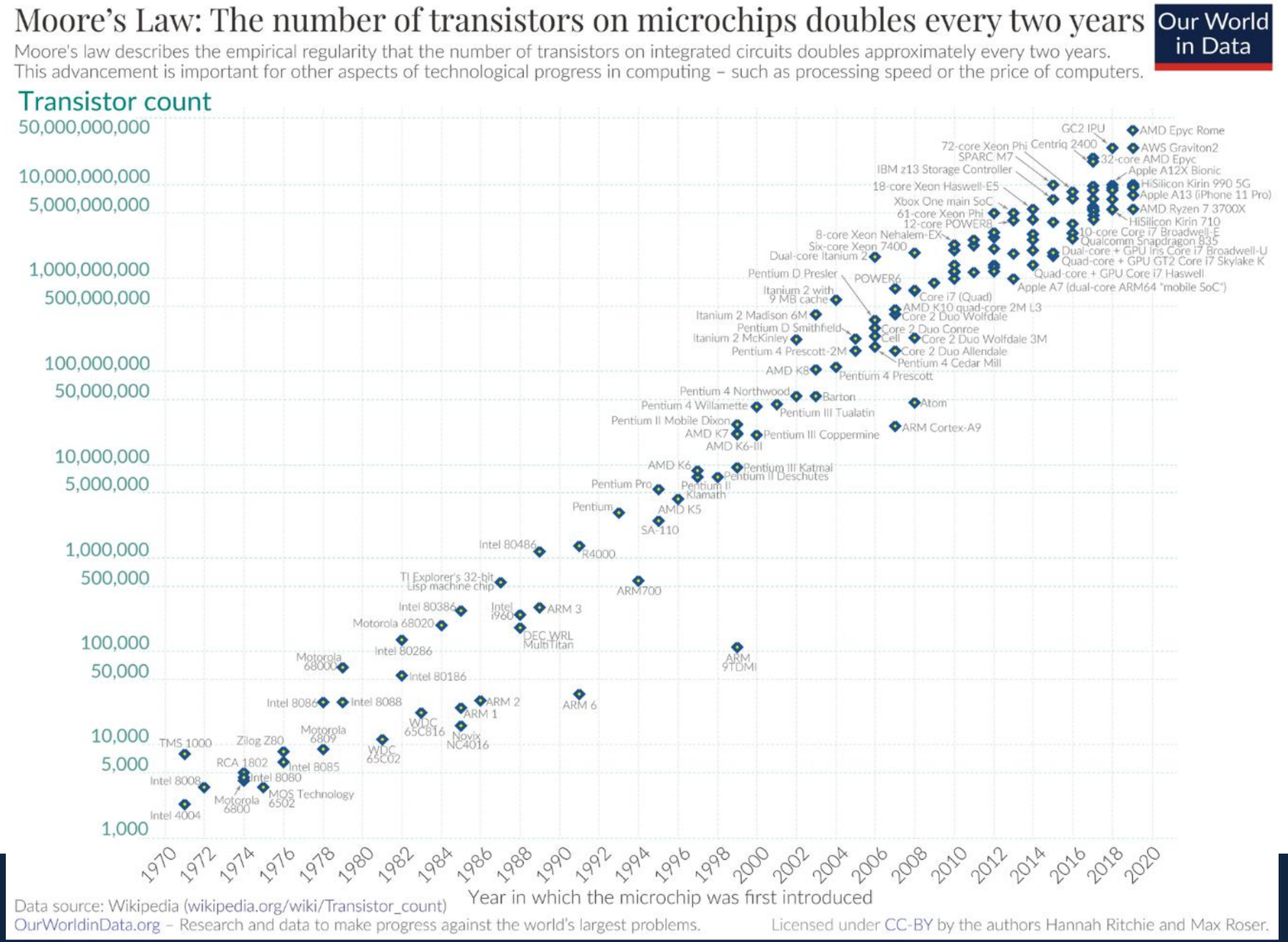
- Open source
- Very popular in academia
  - Don't need to pay super-expensive licensing fees
- Starting to make its way into actual products

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

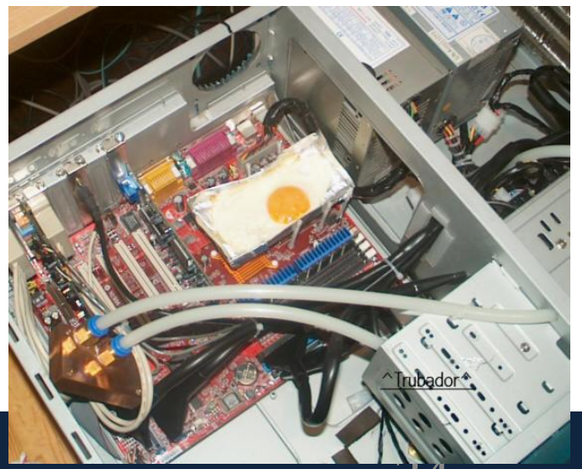
The Trend of Computing

- 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**



Reminder

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

Next Time

- Introduce Instruction Set Architectures (ISAs)