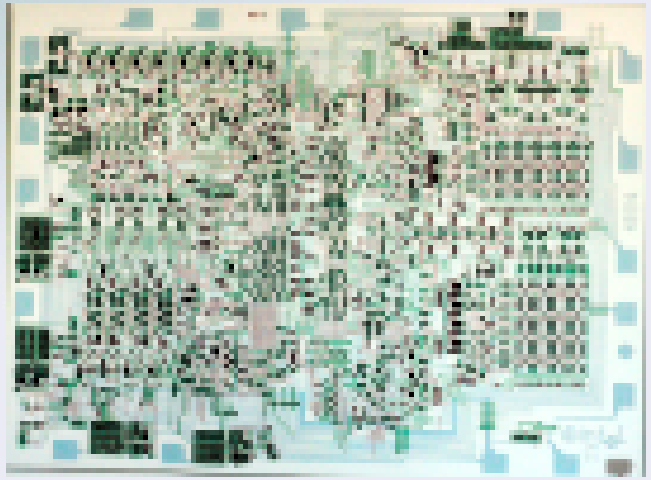


Lecture 1a: Why?

CSCI11: **Computer Architecture and Organization**

C-ID: Computer Architecture and Organization

General Description

"The organization and behavior of real computer systems at the assembly-language level. The mapping of statements and constructs in a high-level language onto sequences of machine instructions is studied, as well as the internal representation of simple data types and structures. Numerical computation is examined, noting the various data representation errors and potential procedural errors."

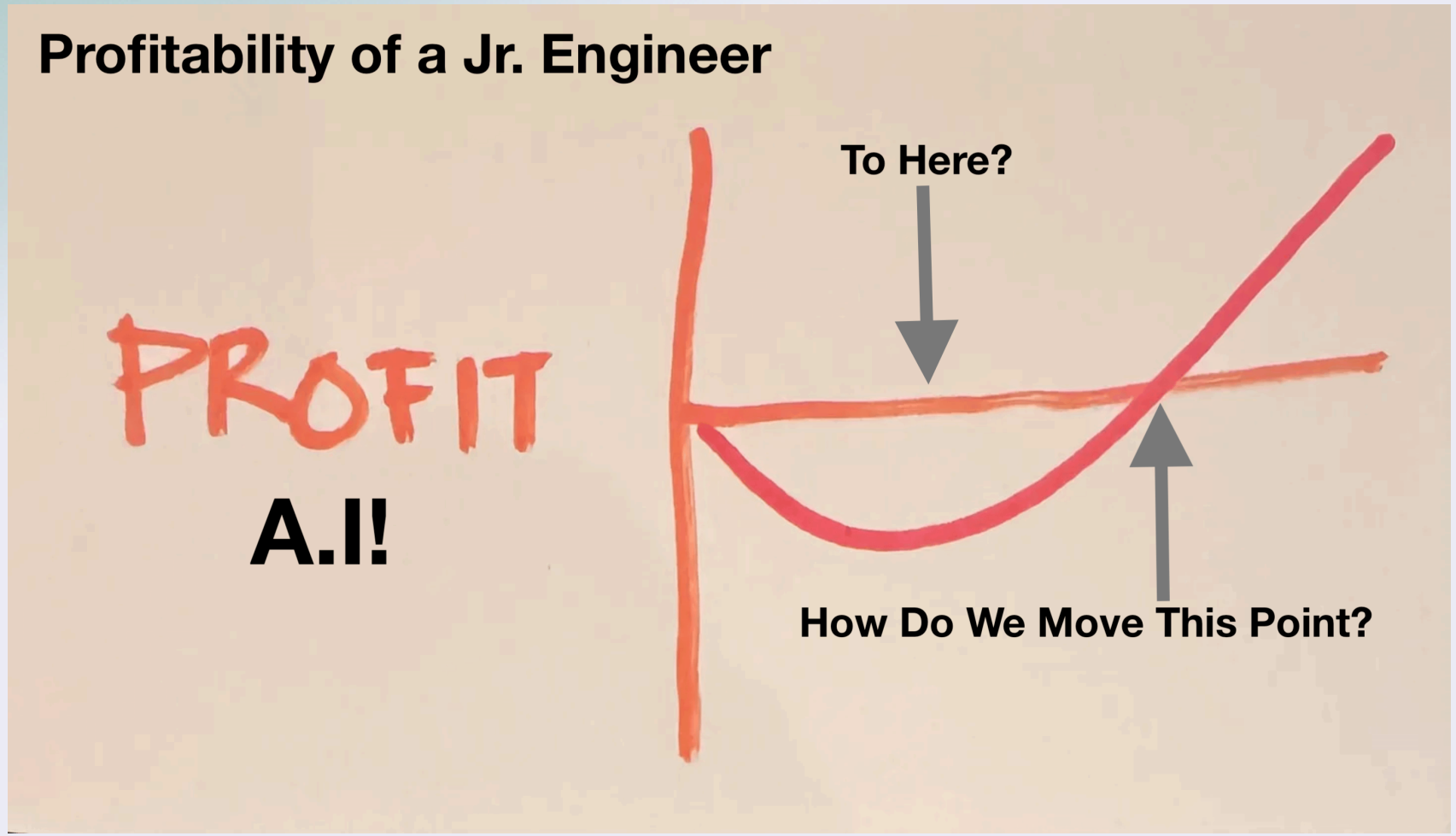
C-ID: Computer Architecture and Organization

Objectives

At the conclusion of this course, the student should be able to:

- *Write simple assembly language program segments;*
- *Demonstrate how fundamental high-level programming constructs are implemented at the machine-language level;*

With AI, why do we need to learn the lowest level of programming constructs?



The Bet On Juniors Just Got Better

Junior Engineers are breaking things

"Friends who are engineering managers and directors and VPs of engineering tell me they're seeing a pattern with junior employees: they write AI-assisted code, ship it, and don't recognize what's wrong with it. The problems surface later—when the feature breaks in production, or when another engineer tries to build on the code and can't make sense of it."

On the Consumption of AI-Generated Content at Scale

Its a significant issue...

"In all of the debates about the value of AI-assistance in software development there's one depressing anecdote that I keep on seeing: the junior engineer, empowered by some class of LLM tool, who deposits giant, untested PRs on their coworkers—or open source maintainers—and expects the "code review" process to handle the rest.

This is rude, a waste of other people's time, and is honestly a dereliction of duty as a software developer"

Your job is to deliver code you have proven to work

Ultimately...

"Programming isn’t the job
Here’s my take: ***AI can replace most of programming, but programming isn’t the job.***

Programming is a task. It’s one of many things you do as part of your work. But if you’re a software engineer, your actual job is more than typing code into an editor.

The mistake people make is conflating the task with the role. It’s like saying calculators replaced accountants. Calculators automated arithmetic, but arithmetic was never the job. The job was understanding financials, advising clients, making judgment calls, etc. The calculator just made accountants faster at the mechanical part."

AI Can Write Your Code. It Can’t Do Your Job.

So What is the Solution?

1. Be of **value -** bring something to your job, which people count on you for
2. Be **proficient -** perform at a high level
3. Be **dependable -** deliver on a consistent basis

Net, net: In a STEM job, your job is to solve problems, to think.

An Example:

1. **Software Engineer -** expert in assembly language programming, *lived it/nerd* (**proficient**)
2. **Technology Salesman -** expert in the sale of disc drives in retail stores, *new markets* (**value**)
3. **Technology Marketing -** channel marketing strategy, *drove execution* (**dependable**)

Assembly Language vs. Computer Architecture

1. While the Description and Objectives call out **Assembly Language**, that's not the point.
2. Understanding **Computer Architecture**, and how computers can be optimized **is the point**.
3. And to do that, it helps to start at the beginning.

CSCI11: What?

- Mon and Wed 6-8:30 BMC205 **Be on-time**
- Instructor: Lief Koepsel
- Syllabus: Review

Textbook (Optional, however, invaluable)

- *Introduction to Computing Systems: From Bits and Gates to C and Beyond*, Patt and Patel, 3rd edition
  Note: The class will follow the chapters in the book.

This class In a nutshell:

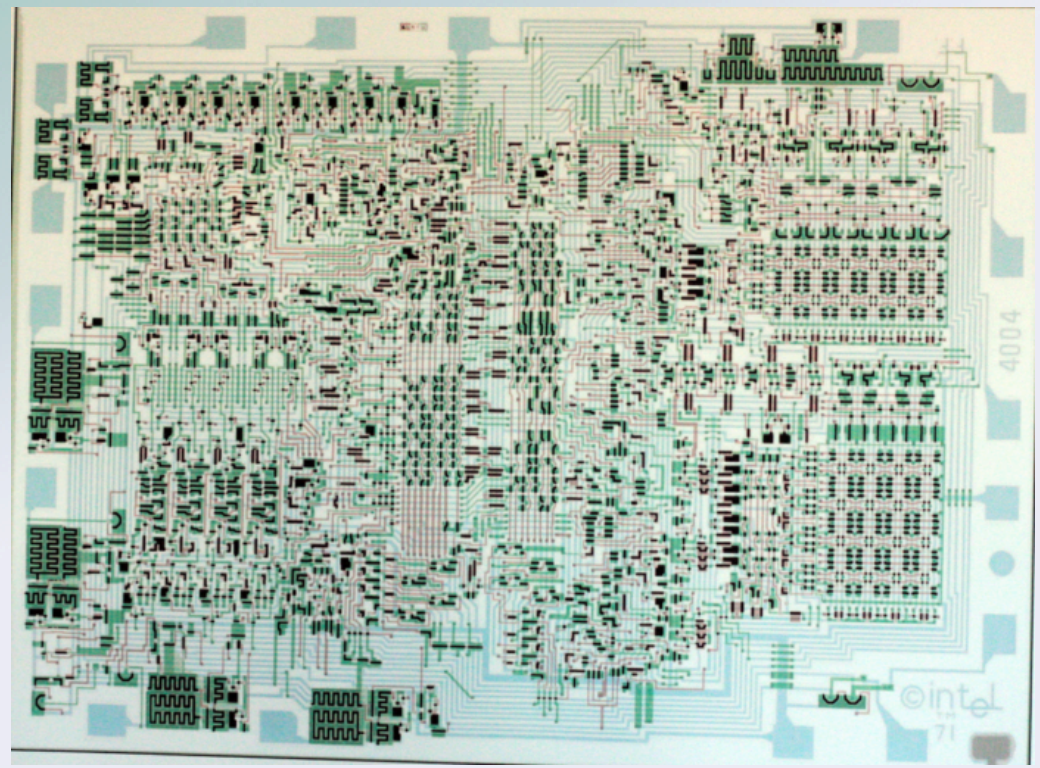
1. Understand the concept of digital design
2. Translate digital design into microArchitecture (*assembly language*)
3. Understand how a higher level language (*C*) compiles into assembly language

From this, you will understand how a computer works

How We Got Here

(*with help from a presentation by Don Fussell UT/Austin*)

A historical journey



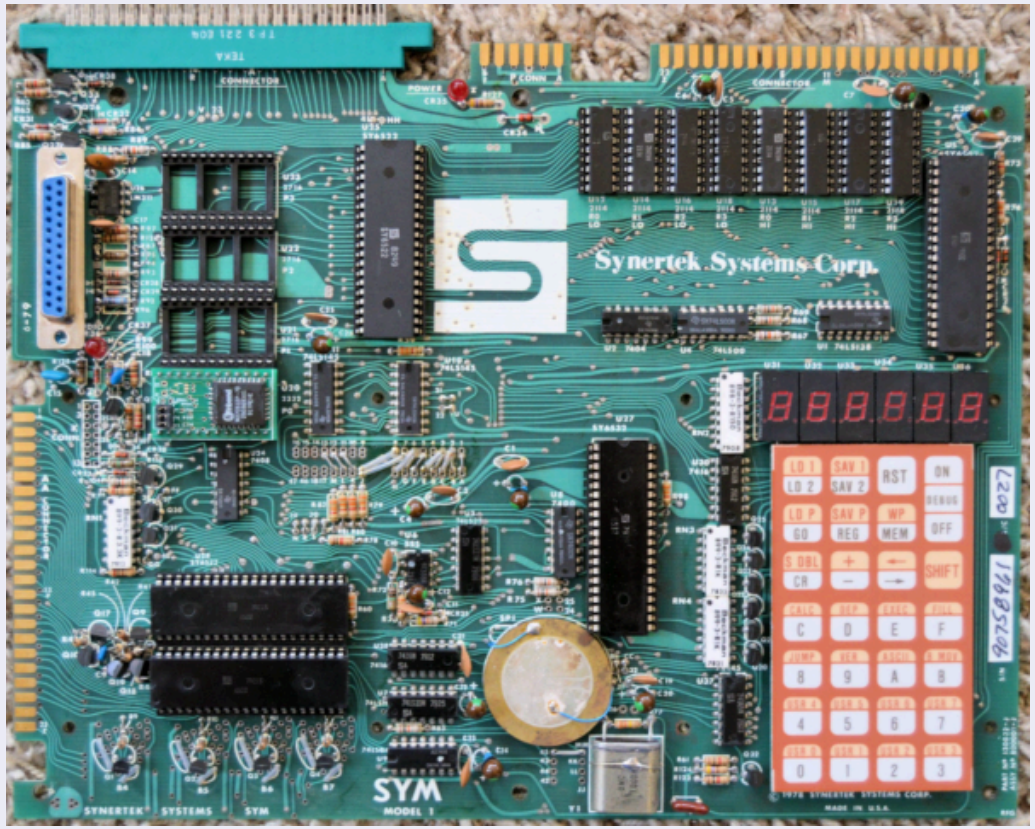
It all started with the 4004, about 55 years ago

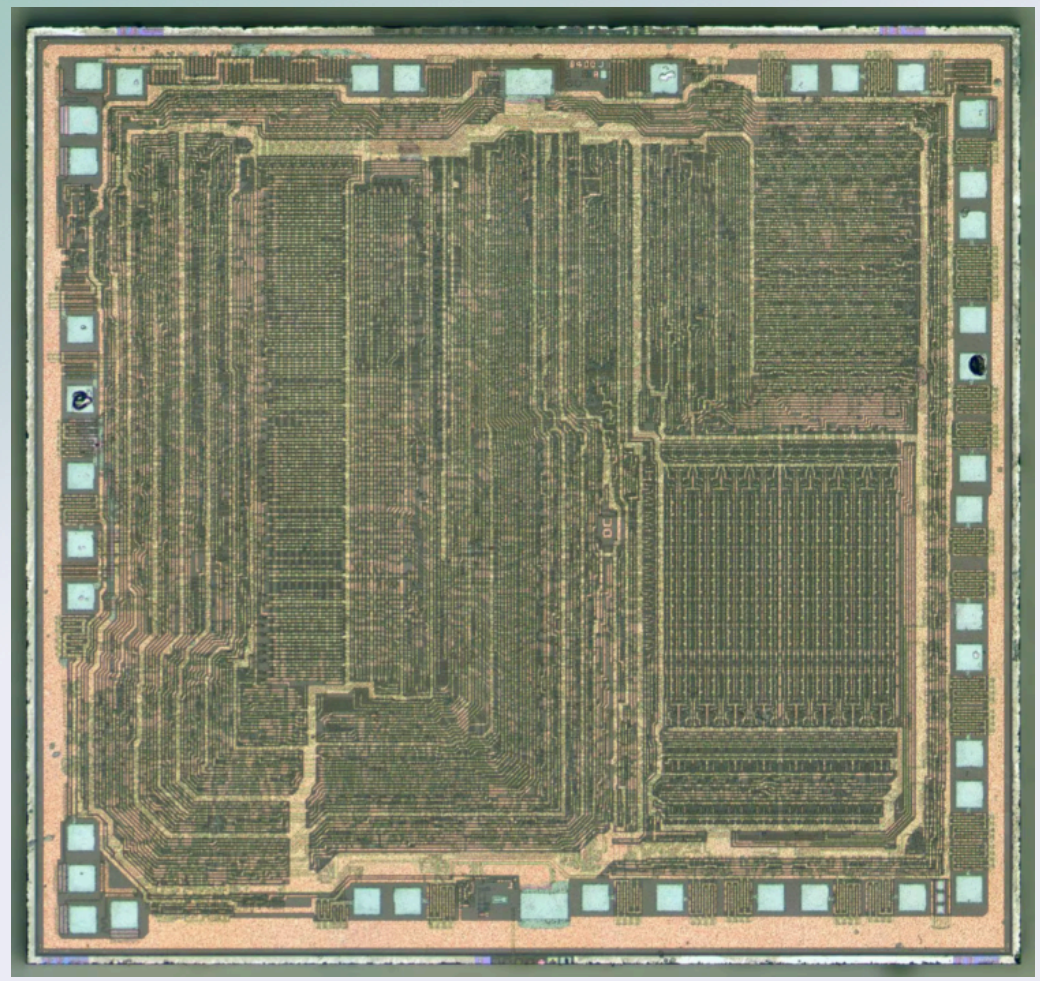
Intel 4004 (1971)

- The first microprocessor
- 2,300 transistors
- 108 KHz
- 10,000nm process
- Intended for a desktop calculator

MOS 6502 (1975)

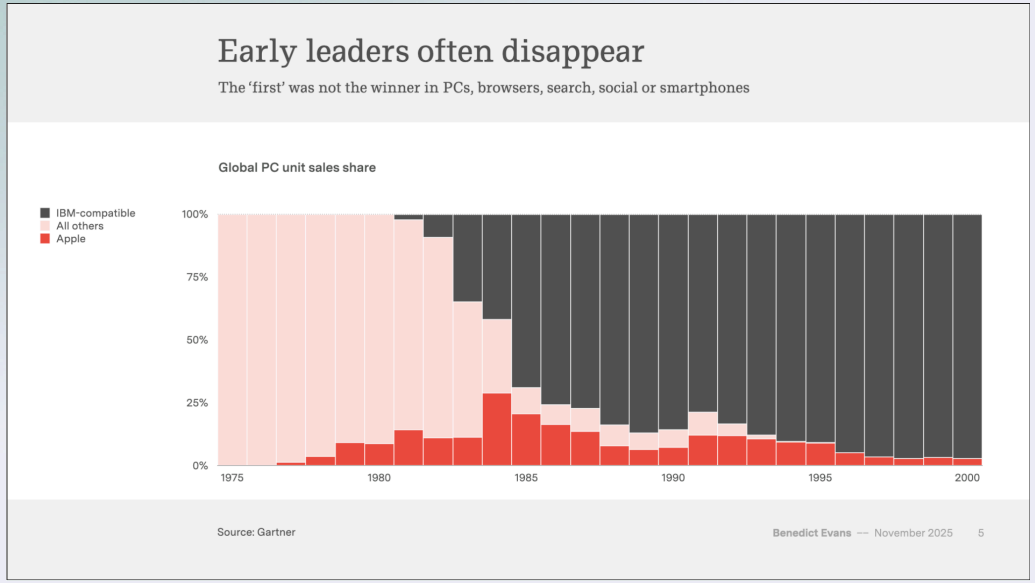
- Apple I/II Microprocessor
- 4,528 transistors
- 2 MHz
- 8,000nm process
- My personal favorite
- Synertek SYM-1





Zilog Z80 (1976)

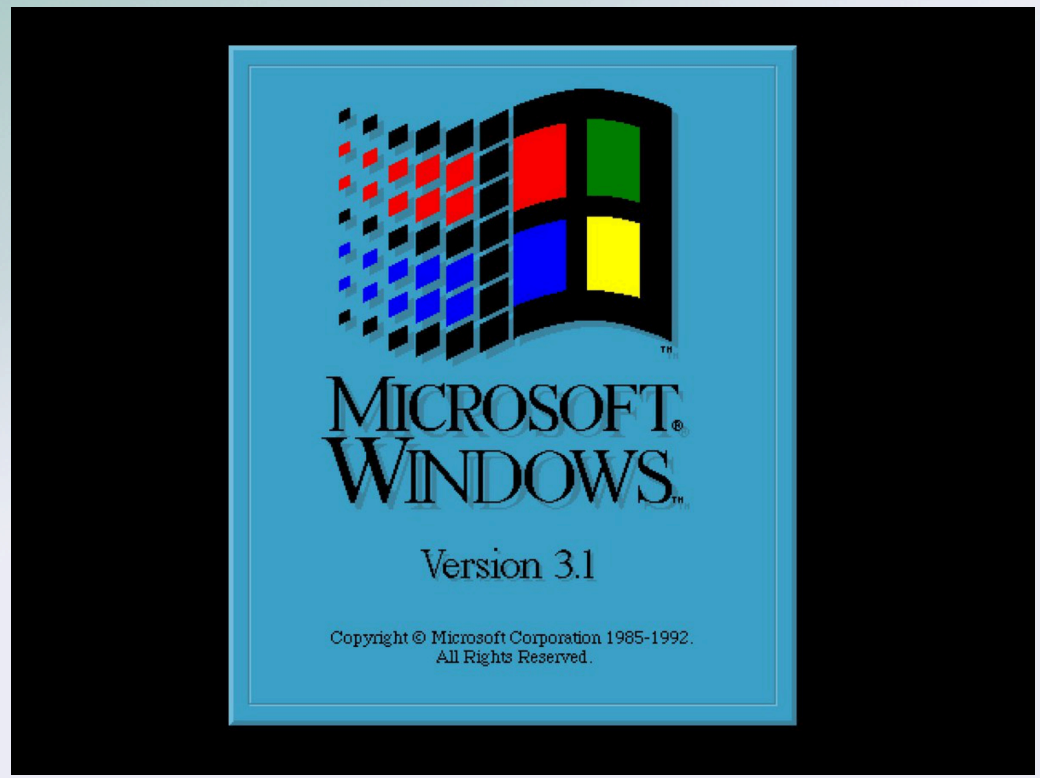
- 8,500 transistors
- 4 MHz
- 4,000nm process
- CP/M OS Microprocessor
- Gary Kildall/Pacific Grove Resident



Intel 8088 (1979)

- IBM PC processor
- 29,000 transistors
- 10 MHz
- 3,000nm process

Benedict Evans

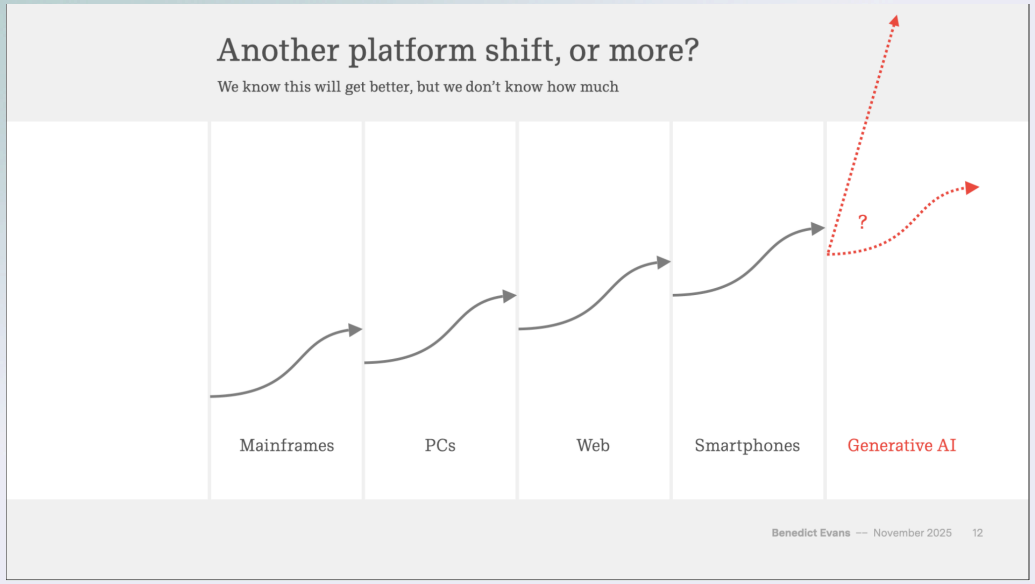


Intel Pentium (1993)

- First Intel processor to execute more than one instruction per cycle
- 3.1 million transistors
- 66 MHz
- 800nm process
- **Wintel**

Intel Pentium IV (2001)

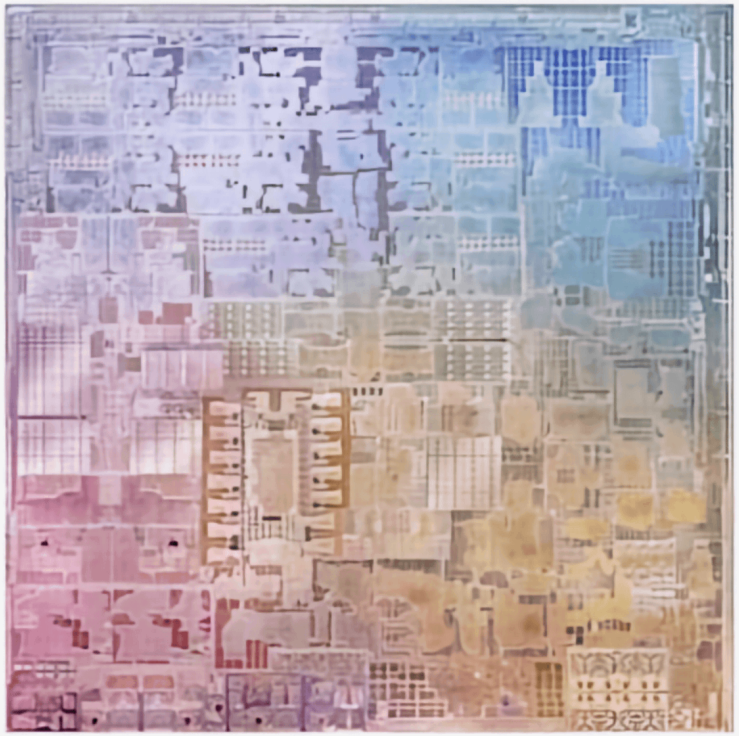
- 42 million transistors
- 2GHz
- 130nm process
- Could fit ~15,000 4004s on this chip!



Apple A8 (2014)

- Apple 6
- 2 billion transistors
- 2 core 64/32 processors
- 20nm process

Benedict Evans



M1 (2020)

- 5nm process
- 8-core CPU, 4 high performance, 4 efficiency
- 8-core GPU
- 16-core Neural Engine
- 16 billion transistors
- chiplet

nvidia H100 (2022)

- 4nm process
- 80 billion transistors
- **CUDA Cores** 16,896 CUDA cores across 132 Streaming Multiprocessors (SMs), lightweight, general-purpose parallel processing units optimized for data parallelism
- **Tensor Cores**: 456 Tensor Cores (in the PCIe variant) designed specifically for machine learning and matrix operations

NVIDIA H100 is a Compute Monster with 80 Billion... | TechPowerUp
NVIDIA H100 PCIe 80 GB Specs | TechPowerUp GPU Database

Fundamental Ideas Become Road Blocks

Moore's Law

- Moore's Law
- **Physics -** As density increased, power and heat increased. As density increases, there are diminishing returns due to inability to shrink the transistors

Microprocessor Journey to Date

- Increase **speed** of microprocessor (*Clock Speed*)
- Increase **capability** of microprocessor (*more transistors*)
- Increase **word size** of microprocessor (*8bit -> 32bit -> 64bit*)
- Increase **capability** of machine instruction (*SSE, MMX*)
- **Simplify** everything and start all over (*CISC -> RISC*)
- Increase the **number of processors** (*2 -> 4 -> 8...144*)
- Increase the **width of the memory bus** (*GPU 4096+*)

How Did We Get Here?

Abstraction

- Allows us to manage seemingly insurmountable complexity
- Billion's of components require abstraction

Hardware v. Software

- Separation of hardware and software is artificial
- To really know software, you must understand the hardware
- To really know hardware, you must understand the software

CPU: Layers of Abstraction

1. Application
2. Program
3. ISA (Instruction Set Architecture)
4. microArchitecture
5. Logic
6. Transistors
7. Physics/Chemistry

Abstraction hides details

Examples:

- "Drive to school" - abstraction of multiple complex systems
- "Make dinner" - abstraction of a multitude of choices

Computing examples:

- Java methods
- C functions
- LC-3 instructions
- Logical gates
- Transistors

The point:

- Often best to operate at highest level of abstraction
- Dangerous to completely ignore lower levels of abstraction

Hardware v. Software

Artificial distinction

At first (*...tried to solve by increasing complexity*)

- CISC vs RISC (complex vs "reduced" instruction set computing)
- MMX/SSE - a single instruction became more powerful
- Intel's Itanium - EPIC (explicitly parallel instruction computing)

Today (*...parallelism at the CPU (core-level)*)

- ARM and RISC V are predominant (RISC processors)
- Apple's Vertical Stack - Software specifically written for M-Series
- nvidia GPU's move from graphics calculations to AI matrix algebra

The point:

- We really care about computation
- Hardware best understood by those who know software
- Software best understood by those who know hardware

Two Very Important Ideas

1. Universality

- All computers can compute the same thing

2. Layered Abstraction

- We can build very complex systems from simple components

1. Universal Computing Device

All computers can computing exactly the same things, given enough time and memory

- PC = Workstation = Supercomputer
- AI consumes significant memory

By the end of class, you will understand how to create a computer from scratch.

1. Turing Machine

Mathematical model of a device that can perform any computation – Alan Turing (1937)

- Ability to read/write symbols on an infinite "tape"
- State transitions, based on current state and symbol

Every computation can be performed by some Turing machine. (Turing's thesis)

Examples:

$T_{add,b} \rightarrow a+b$ (Turing machine that adds)

$T_{mul,b} \rightarrow a*b$ (Turing machine that multiplies)

1. From Theory to Practice

In theory:

- Computers can compute anything that's possible to compute
- Given enough memory and time

In practice:
Solving real problems requires computing under constraints

- **Time** Weather forecast, next frame of animation...
- **Cost** Cell phone, automotive engine controller...
- **Power** Cell phone, handheld video game...
- **Memory** AI, agents...

2. The Statement of the Problem

Problem
Algorithms
Language
ISA
Microarchitecture
Circuits
Devices

Problem Statement

Use "natural language", may be ambiguous, imprecise

Algorithm

Step-by-step procedure, guaranteed to finish, definiteness, effective computability, finiteness

Program

Express algorithm using a computer language, high or low level language

Instruction Set Architecture (ISA)

The set of instructions the computer can perform, data types, addressing mode

Microarchitecture

An implementation of the ISA, such as both Intel and AMD create microprocessors based on the x86 ISA

Circuits

The combination of devices which create the microArchitecture, the images provided earlier i.e; the 4004

Devices

Each individual element which comprise the circuit, a logic gate, a memory cell in a specific technology such as NMOS, CMOS or bi-polar

2. Layered Abstraction

How do we solve a problem using a computer?

Systematic sequence of transformations between layers of abstraction...

Problem → Algorithm

- Software Design: choose algorithms and data structures

Algorithm → Program

- Programming: use language to express design

Program → Instruction Set Architecture

- Compiling/Interpreting: convert language to machine instructions

Many Choices at Each Level

"Solve a system of equations"

- Gaussian elimination
- Jacobi iteration
- Red-black SOR
- Multigrid

Languages:

- FORTRAN
- C
- C++
- Java
- Javascript
- Forth

Architectures:

- x86
- ARM
- GPU

Processors (ISA):

- **Intel** x86, Intel, AMD, Zhaoxin, *CISC*
- **ARM -** Apple M-series, Qualcomm Snapdragon, **Raspberry Pi**, *RISC*
- **RISC V -** Open Source, Alibaba, Espressif, GigaDevice, **Raspberry Pi**, *RISC*

Technology:

Process	Strengths (for µP/µC)	Weaknesses (for µP/µC)
NMOS	good for 1970s/80s microprocessors	limited transistor count before cooling becomes prohibitive
CMOS	Near-zero static power	roughly 2× transistors per gate vs NMOS (cost)
Bipolar	very fast switching (bit-slice)	expensive, area-hungry logic
GaAs	Very fast, 60MIPS RISC	power-hungry, difficult to manuf.

Tradeoffs:

- cost
- performance
- power
- heat

Starting from the "Bottom"

Bits and Bytes

- How do we represent information using electrical signals?

Digital Logic

- How do we build circuits to process information?

Processor and Instruction Set

- How do we build a processor out of logic elements?
- What operations (instructions) will we implement?

Starting from the "Bottom" (*continued*)

Assembly Language Programming

- How do we use processor instructions to implement algorithms?
- How do we write modular, reusable code? (subroutine)

C language Programming

- How do we maintain the speed we achieved using Assembly Language?
- How do we attract additional development?

Course Components

Part 1: Hardware

- Overview, binary logic, binary tools, data, transistors, gates, digital logic structures, von Neumann machine model

Part 2: Assembly language

- LC-3, ISA, (structured) programming, input/output, relationship to hardware

Part 3: C programming

- High level languages, C language syntax, **the stack and stack frame**, operators, control structures, functions, pointers, data structures, all in relationship to assembly language

Objectives

- Understand role & relationship of hardware and software

Exposure to:

- Machine organization
- Assembly language programming
- C programming
- Understand how to build entire (slow) computing system
    - Hardware and software

Why Study Hardware?

Important:

- HW drives the industry
- HW determines the SW which determines the HW

Timely:

- RISC processors
- multi-core processors
- parallel execution (memory bus width > 4096 bits)

Why Learn Assembly Programming?

- Helps understand capabilities of machine
- Assembly language enables speed
- Many system components written in assembly
  - E.g., microcontrollers, device drivers, media kernels, digital signal processing (DSP) code

Why Learn C Programming?

What is C?

- High-level language, somewhat transportable, "Portable assembly language"
- Invented in 1970s to write the Unix operating system
- In between assembly and Java/Python

Very common:

- Operating systems and many general applications

Still the right tool for many tasks:

- Foundation for C++/Java/Python
- Embedded applications (largest market)