Computer architecture and the hardware/software interface

> To understand a program you must become both the machine and the program.

Alan Perlis

Among foundational computer science courses, computer architecture is *bedrock* for software engineers. By learning how microprocessors work and interact with the software layer, your future understanding of operating systems and software applications will stand on much sturdier ground.

RECOMMENDED RESOURCES

While it's possible to complete this course simply by participating in classes, you will see the most value if you do the suggested prework.

"P&H" below refers Patterson and Hennessy's Computer Organization and Design—a classic text, commonly used in undergraduate computer architecture courses. The authors are living legends, having pioneered RISC and created MIPS, acronyms that will become familiar to you shortly if they are not already!

For those who prefer video-based courseware, our recommended supplement to our own course is the Spring 2015 session of Berkeley's 61C course "Great ideas in computer architecture" available on Youtube.

For students who have some extra time and would like to do some more project-based preparatory work, we recommend the first half of *The Elements of Computing Systems* (aka Nand2Tetris) which is available for free online.

For those with no exposure to C, we strongly recommend working through some of K&R C before the course commences. We will have one class covering C, but the more familiar you are, the better.

CLASSES

1 - The big picture: how data is represented, stored and manipulated

This class provides an overview of the entire system. Given a simple program (say to add two numbers together) how is it stored on disc and loaded into memory? How are its instructions encoded in a form that the microprocessor can understand? How is its *data* stored, transported and operated over?

Suggested prework: watch Feynman's Esalen introductory lecture or read the first of his Lectures on Computation

Further study:

- P&H 1.3-1.5 and 2.4
- This 61C lecture from 55:51
- Book: Code by Charles Petzold

2 - An overview of C, the portable assembly language

This class is mostly a crash course in C that will allow us to write code closer to the metal. Rather than covering the language exhaustively, we will focus on aspects that are most relevant to our course, such as types, structs, arrays and pointers.

Suggested prework: read Chapter 1 of The C Programming Language (K&R C) and do some of the exercises, or read Learn C in X minutes.

Further study:

- The rest of K&R C!
- C programming problems on exercism.io
- Brian Harvey's intro/rant

3 - MIPS: a simple instruction set architecture

This class uses MIPS as a channel to explore ISAs. We will consider some of the tradeoffs that computer engineers make when designing an ISA, and learn MIPS thoroughly enough that we will be able to write some small programs. MIPS will later be the lens through which we look at the layout of a microprocessor—today's class will focus on the software side of the divide and how MIPS instructions are encoded; future classes will describe how they are actually executed.

Suggested prework: Read P&H 2.1-2.3 and 2.5-2.9, or watch these lectures: 1 2

Further study: exercism.io MIPS problems

4 - Compiling, assembling, linking and loading

This class is a more detailed view of how programs are built and executed, building on ideas introduced in class 1. We will see what the specific steps are in assembling a program to machine code, how multiple object files are linked, and the details of preparing the combined executable to be run. Much of this class will center around designing an assembler and writing a linker for the Mach-O executable format.

Suggested prework: Read P&H 2.12 or watch this lecture, read Let's Build a Mach-O Executable

Further study:

- Bradfield "languages, compilers and interpreters" course
- the Dragon book

5 - The processor, clock and datapath

This class is a turning point! We have enough of an understanding of how our logic becomes instructions for the machine that we can now examine how they specifically flow through and are executed by the processor.

Suggested prework: Read P&H 4.1-4.4 or watch this lecture

Further study: this lecture although this is a greater level of detail than is likely to be interesting to software engineers.

6 - Logic gates and using them to build logic gates

In the previous class we handwaved around how some logical and arithmetic operations are actually performed by the microprocessor. In this class we will examine logic gates in more depth, and combine simple logic gates into more complex ones, up to the ALU itself.

Suggested prework: Read P&H appendix B1-B6 or watch this lecture

Further study:

- The first three chapters of Nand2Tetris are a great way to practice designing logic gates
- The free Zachtronics game called KOHCTPYKTOP: Engineer of the People which has you build out circuitry, even laying down the P- and N-type silicon yourself.

7 - Pipelining

Pipelining was one of the earliest, most ingenious and most effective improvements to microprocessor designs. This class explains how it works and explores some branch prediction strategies, focusing on what a practicing software engineer might want to know to write super high performance code.

Suggested prework: Read P&H 4.5-4.8 or watch this lecture

Further study: P&H 4.10-4.11

8 - Memory hierarchy

This is the most practical class in the course, as a program's rate of L1/L2/L3 cache misses is a common cause of poor performing low level code. We cover the reason for the innovation, how the caches operate, and how to measure and work with them.

Suggested prework: watch this amazing rant by Mike Acton and read P&H 5.1-5.4

Further study:

- Dan Luu's What's new in CPUs since the 80s
- For the bold, What Every Programmer Should Know About Memory
- Much of Computer Architecture: A Quantitative Approach (the more advanced "H&P" version of P&H) is on memory hierarchy design

9 - Parallelism, the Flynn taxonomy and Amdahl's law

We are constantly trying to squeeze more compute out of our computers. Two important ways that we do this are to operate over more data per operation (SIMD or vector operations) as well as to run multiple threads in parallel on separate "cores". This class examines these innovations and aims to build a working familiarity with both, given that they continue to increase in importance with the rising popularity of machine learning, virtual reality and other highly parallel compute tasks.

Suggested prework: watch these two lectures 1 2 and the talk The Future of Microprocessors and read An Easy Introduction to CUDA C and C++

Further study: Udacity's Intro to Parallel Computing with CUDA