Computer Architecture and the Hardware/Software Interface

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

Alan Perlis

As software engineers, we study computer architecture to be able to understand *how our programs ultimately run*. Our immediate reward is to be able to write faster, more secure and more space-efficient programs.

Longer term, the value of understanding computer architecture may be even greater. Every abstraction between you and your hardware leaks, to some degree. This course should provide you with a set of primitives from which you can build sturdier mental models and reason more effectively from first principles.

We'll start mostly on the hardware side of the hardware/software interface, building our undestanding of how the machine works and writing assembly language programs to explore a typical instruction set architecture. Once we have an appreciation of program execution at a low level, we'll move on to higher level considerations like C language programming and the compile/assemble/link/load pipeline, the basic responsibilities of an operating system as well as one of the the most important performance consquences of modern architecture: CPU utilization.

RECOMMENDED RESOURCES

Please endeavor to complete all of the prework for each class. Doing so will help us cover more content overall, and to spend more time on the kind of interactive activities that we can only do in person. We've done our best to keep the prework short, interesting and relevant, so please let us know if there's anything that seems off topic or unreasonably long or uninteresting.

"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 the Internet Archive.

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.

Finally, an alternative textbook which we like is *Computer Systems: A Programmer's Perspective*. If you find the P&H book too hardware-focused, CS:APP may be worth a try. It uses a different architecture (a simplified version of x86) but the book is good enough that the extra translation effort may be worthwhile.

CLASSES

1 What Is a "Computer"? and What Does It Mean to "Program" It?

Our first class will provide an overview of the entire system. By the end, you should be able to explain at a high level:

- How a program is stored on disc and loaded into memory;
- How its *instructions* are encoded in a form that the microprocessor can understand;
- How its *data* is stored, transported and operated over; and,
- How all these components work together to do computation.

While we will touch on some low level details, in this class we'll focus on *simulating* the low level behavior of computers using a high level language. In subsequent classes we will continue to zoom in, describing these low level behaviors at the level of assembly language and circuitry.

Prework

Please watch Richard Feynman's introductory lecture (1:15 hr) and test yourself by explaining the key ideas to a friend. Feynman describes a computer mostly in metaphors, as he talks about cards, storage and filing systems, and the responsibilities of a computer, think about these questions:

- Modern computers are a series of electronics; physically speaking, what are the "cards" Feynman is discussing?
- What kind of hardware might be needed to to "file" and "process" these "cards"?
- What is the difference between storing data, and processing data?
- How does the filing system know which cards to fetch and process?
- How does the system know what kind of processing to perform on a given card?

In class, we're going to discuss computers much more *literally*. This will include defining a number of important hardware components. To prepare for that discussion please read the introductory section of the following wikipedia articles:

- Register
- Computer Data Storage
- Arithmetic Logic Unit (ALU)
- In Central Processing Unit, read the introductory section as well as the section under the heading "Decode"

Throughout the course we will examine each of these things in detail. Focus on expanding your vocabulary for now; as class goes on we'll learn more about how all these components work.

Further Resources

If you enjoyed the Feynman lecture above, you may be excited to know that he taught an entire introductory course on computation available in book form as *Feynman Lectures on Computation*. There are several other books that provide a good high-level introduction: a popular one is *Code* by Charles Petzold, another is *But How Do It Know* by J Clark Scott.

For those looking for an introduction to computer architecture from a more traditional academic perspective, we recommend P&H chapters 1.3-1.5 and 2.4, as well as this 61C lecture from 55:51 onwards.

2 Binary Data Representations

In this class we will discuss several data types and their binary representations. Because computers operate using electrical signals discretized into "high" and "low", the data that computers store and operate over must also have binary representations. Furthermore, these values must be manipulable by machine hardware in meaningful ways; binary representations of numbers are coupled with hardware implementations of addition, subtraction, multiplication, and so on.

By the end of this class you should be able to:

- Translate signed and unsigned integer values to and from binary;
- Translate IEEE Floating Point numbers to and from binary, and described their uses and limitations;
- Translate UTF-8 encoded binary data into their "code point" integer values and glyphs, and explain its variable length encoding scheme;
- Explain the concept of "endianness" and its implications; and,
- Examine and understand sophisticated binary formats such as a TCP headers, Protocol Buffers, serialization formats, and more.

Prework

- Watch Binary Addition & Overflow (7 min) and test yourself by converting the numbers 12 and 9 to binary, adding the binary values together, and converting the result back to decimal to verify your calculation. Do these by hand. Ask yourself, what would the result be if it were constrained to 4 bits? How many numbers can be represented in total with 4, 8, 16 or 32 bits respectively?
- Watch Why We Use Two's Complement (16 min) and test yourself by converting the numbers 12 and -9 to binary using the two's complement representation, adding the binary values together, and converting the result back to decimal. Similarly compute (by hand!) -3 - 4 ("negative three, minus four"). Calculate what are the largest and smallest numbers representable in 32 bit 2's complement.

Ask yourself, what are two positive numbers that cause an overflow in 8 bit 2's complement. What about two negative numbers?

Class will start with a **very short** review of binary integers, both unsigned and two's complement, so please watch these above two videos carefully. Additionally, watch these two videos:

- IEE Floating Point(9 min) IEE Floating Point format is a ubiquitous and powerful format used to both represent infinitesimally small numbers, and extraordinarily large numbers. The format borrows quite a lot from the idea of scientific notation; keep that in mind while studying the format you will appreciate the format much more.
- The Unicode Miracle:(10 min) Unicode, and UTF-8 specifically, is a fantastic piece of low level thinking that allows for flexibly encoding data in a way that is "backwards compatible" with ASCII, yet flexible enough to represent 1,112,064 unique characters (compared to ASCII's 128 or extended ASCII's 256). Even more impressive, if you only encode characters supported by ASCII the UTF-8 encoding takes the same amount of space on disk.

We will spend more time discussing floating point and Unicode in class, so it is okay if those topics still feel fuzzy. You should, however, be able to convert any unsigned integer to it's decimal version before class begins.

Further Resources

- Watch 15 and Hexadecimal numbers (8 min) and test yourself by converting the numbers 9, 136 and 247 to hexadecimal. If you are familiar with CSS, ask yourself what is the relationship between rgb and hex representations of colors? If you were given a hex code, could you guess the color?
- Watch the first 5-ish minutes (remainder optional) of Byte ordering. Consider this: in a TCP segment, the source and destination ports are stored as two-byte integers. If you saw port 8000 represented as 0x1f40, would you conclude that it is stored as big endian or little endian? How would you represent port 3000?

3 Programming "Against the Metal" with Assembly Languages

This class introduces the MIPS architecture. MIPS will be the lens through which we look at the layout of a microprocessor and the design of its instruction set. We use MIPS as it's a *simple but real* architecture, used for instance in some game consoles and the Mars rover, and a precursor to the ARM architecture used in most phones. MIPS is small enough to understand within the timeframe of the course, but the knowledge we attain from studying it will transfer readily to other architectures like the ubiquitous x86.

In this class, we'll write MIPS assembly code as a way to explore the set of instructions available for a typical MIPS computer. By the end of the class, you should be able to write simple programs in MIPS assembly, as well as to explain at a high level:

- Which high level programming statements compile to single instructions, and which to multi-step procedures;
- How a conditional statement is executed, at a low level;
- How a loop is executed, at a low level; and,
- What a calling convention is, and how a function call is made.

Prework

The exercises we'll start to solve in class will come from a set that we maintain on exercism.io. Please follow the instructions to install the exercism.io client and MARS simulator, which we will use to run our assembly code. Please make sure that you can at least fetch and run the first exercise, and ideally make an attempt at solving it, too!

For a background in MIPS, watch these lectures: 1 and 2 (2.5 hrs total) or read P&H 2.1-2.3 and 2.5-2.9. You may also find it useful to play with the MIPS converter.

Please also read this brief article about how a computer boots.

Further Resources

The best option for further study is to do some more of the MIPS exercises on exercism.io. As an alternative approach, there are two *games worth playing*: SHENZHEN I/O and Human Resource Machine. Both use simpler instruction sets (and assembly languages) than MIPS, but solving the programming problems in either will help train you to think at the level of a typical machine.

For a more practical approach, you may want to start moving towards understanding the Intel x86 architecture. This is a very complicated instruction set, and real expertise will come with repeat use in a professional context. But a good resource for getting a little background is chapter 3 of the book *Computer Systems: A Programmer's Perspective*.

4 A Brief Tour of Logic Circuits

You probably know that a computer works by combining a series of electric signals through operations like AND, OR and NOT. But how do these operations combine to perform something like an addition? How can we remember values between one CPU cycle and another? This class starts to demystify these and other aspects of how instruction are actually executed.

By the end of this class, you should be able to:

- Draw truth tables for simple circuits;
- Explain how NAND gates are combined to form more complex combinational circuits;
- Explain how circuits (like adders) can be constructed to affect *arithmetic* through combinational logic; and,
- Explain how a flip-flop works, and how they can be used to give our circuits "memory".

While this course may be the last time you're asked to design even a small circuit, your model of how logic affects computation will over time become a key aspect of your mental framework for first principles reasoning as a software engineer.

Prework

At an absolute minimum, come to class able to draw and reproduce truth tables for the following 5 logic gates:

- NAND
- AND
- OR
- NOR
- XOR (sometimes called EXOR)

A simple reference can be found here.

It will be helpful to come to class with a general idea of how logic gates could be combined all the way up to something like an adder. The video How Computers Add in One Lesson (15 mins) is a surprisingly good starting point given the brevity. To test your understanding, consider this: the video shows a simple design of a multi-bit adder using a sequence of full adders, where the carry bit of one becomes an input to the next. Does this seem slower than it needs to be? Can you come up with a design that would be faster?

You should also aim to come to class with some sense of how we build up to memory chips. The key question is, how do we *remember* even one bit of information? Have a think about that question, then please watch this video (10 mins), part of Ben Eater's excellent series building an 8 bit computer on a breadboard. Feel free to also watch the subsequent videos building up to flip-flops, but don't worry if you get stuck on the details: what's important is overcoming the hurdle of starting to reason about *sequential* rather than *combinational* logic.

If you have more time, we suggest working through the programming exercises for the first chapter or two of Nand2Tetris.

Further Resources

The first three chapters of Nand2Tetris correspond roughly to the topics covered in this class. Building these gates in Hardware Description Language and seeing them run in the Nand2Tetris emulator should give you a much more solid understanding.

The game designer who made SHENZHEN I/O (referenced above) also made the less polished but still fun (and free!) game KOHCTPYKTOP: Engineer of the People which has you build out circuitry, even laying down the P- and N-type silicon yourself.

For those preferring more conventional coverage of the topics, see P&H appendix B1-B6 or this lecture from 61C.

5 The Structure of a Simple CPU

Now that we have an understanding both of the basic combinational and sequential circuits at our disposal, and of the set of instructions a CPU might support, we may now attempt to bridge the gap between the two. Again we'll use MIPS as our example, and together sketch out the datapath and control unit for a CPU that could theoretically execute a subset of MIPS instructions.

We cover how the the core of MIPS instruction set (the datapath) is implemented with the combinatorial and sequential logic circuits we learned about in the previous class.

Prework

Please either read P&H 4.1-4.4 or watch this lecture before class. The details here aren't as important as the high level responsibilities of the pieces. To test your understanding, you may want to try explaining "how a computer works" to a curious 12 year old. By this point, no aspect of that explanation should be a complete mystery to you, although there may be many more details that you'd like to fill in!

Further Resources

This lecture goes into more detail on the same topics.

As mentioned earlier, Ben Eater's Youtube series *Designing an 8-bit breadboard computer* is great, and will give you another perspective on what it looks like to bring components together to behave overall as what we can call "a computer".

Two other resources that your instructor may have referred to in this class are the 6502 emulator and the Scott computer emulator.

6 Intro to Operating Systems: Booting, Loading, and Virtual Memory

In this class we peak one level above the architecture of a computer into the operating system. Having completed our discussion of the mechanical/physical/electronic aspects of computers we have a foundation on which to begin understanding the wonders of operating systems. Modern operating systems do so much for us, so well, that it's easy to take them for granted. In this class we touch on some of the fundamental components of operating systems in order to elucidate how applications and programs are scheduled and managed in a modern system.

By the end of class you will be able to:

- Describe what happens when a computer turns on.
- Describe how an OS loads and executes a program.
- Describe the concept of "virtual memory" and some tactics for implementing it.
- Simulate loading and executing a program in a virtual memory space.

Prework

Please read the following chapters of Operating Systems: Three Easy Pieces, available online:

- Chapter 6: Limited Direct Execution
- Chapter 15: Address Translation

These chapters will give us enough just enough information to tackle the simulation a (very) simplified operating system. This is just a taste of what a full blown operating system has to achieve, but it is enough for us form a good mental model of the responsibilities of the operating system as it relates to both computer architecture and program execution. We will focus our discussion of operating systems on two areas: loading & executing programs, and memory virtualization.

Further Resources

The entire OSTEP textbook is quite readable and has been made available for free by the publisher. A great way to push your knowledge further specifically in the context of operating systems and memory management would be to read OSTEP chapters:

- Chapter 14: Memory API
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging

Afterwards, attempt the `malloc` exercise from *Computer Systems: A Programmers Perspective*. All the CS:APP labs are available here – you want to click the malloc self study handout and the malloc writeup links for any of the labs you are interested in. You'll have to learn a bit of C to complete the exercises, but they are a wonderful challenge once you get them running.

7 An Overview of C, the Portable Assembly Language

This class is mostly a crash course in the C programming language and its associated tooling. C allows us to write higher-level code that can be ported between different CPU instruction sets and operating systems. 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. We will also reexamine the operating system's role in the "compile -> assemble -> link -> load" pipeline that's required to run even the simplest of C programs using the lenses of our tools: compilers (`gcc/clang`), debuggers (`gdb/lldb`), and binary data examination tools (`xxd/objdump`).

Prework

Please come to class with at least a little familiarity with C syntax. If you can get a hold of a copy of The C Programming Language (K&R C), read the first chapter (25 pages) and do some of the exercises along the way. If not, read Learn C in X minutes and test yourself on the first 5 or so C programming problems on exercism.io. If everybody arrives comfortable with the types and basic control flow, we can spend more time on more interesting topics like pointers, arrays and functions.

Further Resources

Once you have written your first few C programs, you have taken a big step on a long road. K&R C is the canonical text, and for good reason: it is brief, well-written and co-authored by the creator of the language. In our opinion, it is one of the few programming languages books that *every* software engineer should read.

As with any programming language, understanding comes both through study and practice. If you have a project idea that lends itself to C, that's great! Otherwise, solving a set of discrete problems such as the C programming problems on exercism.io may be a good first step.

Brian Harvey's notes on C for students of the introductory computer architecture course at Berkeley provides an interesting alternative perspective and is well worth the read, even if you feel comfortable with C already.

A useful resource for understanding pointers specifically is Ted Jensen's tutorial.

If you seek a better understanding of the CALL pipeline, you may soon find yourself in the world of programming languages and compiler theory. We predictably suggest our *Languages, Compilers and Interpreters* course as an introduction, and the canonical text is *the Dragon book*.

8 Improving Performance with Caching

In this class, we explore one of the most important practical aspects of modern computer architectures: the "memory hierarchy". Modern computers use a series of levels of 'caching' to mitigate costly data retrieval operations.

For many common workloads, a program's rate of L1/L2/L3 cache misses can be the greatest cause of poor performance. We cover the reason for the innovation, how the caches operate, and most importantly how to measure and work with them. You will practice by profiling and optimizing code to make better use of the caches on your own computer.

Prework

As preparation for this class, please read the blog post Why do CPUs have multiple cache levels? by Fabian Giesen, and watch this talk by Mike Acton. The first should help you rationalize why we've settled (for now) on the configuration of CPU caches that you'll typically see, and the second serves as motivation for why this matters for programmers.

In class we'll also be using Cachegrind (a Valgrind tool) to measure the cache utilization of the code that we'll be optimizing. Please make sure that you have Valgrind installed and able to run Cachegrind on a real program.

Further Resources

What Every Programmer Should Know About Memory may be ambitiously named given its depth, but certainly you should aspire to know much of what's contained. If you are excited about making the most, as a programmer, of your CPU caches, then this will be a great starting point.

P&H "covers" the memory hierarchy in sections 5.1-5.4, but it is designed to be a kind of appetizer for the chapters on memory hierarchy design in *Computer Architecture: A Quantitative Approach* (the more advanced "H&P" version of P&H).