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 by probe the hardware/software interface from a number of directions, writing C and assembly code as well as well as designing simple circuits. Once we have a good understanding of the overall system, we'll explore three particular aspects that are particular important when writing fast software: pipelining, the memory hierarchy and parallelism.

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 operate over; and,
- How common "types" of data such as integers and strings are represented in binary.

This class also 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.

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?
- Watch How To Read Text In Binary (3 min). The presenter asserts that the *skill* of reading ASCII encoded values is "almost useless", which may be true, but the *understanding* of how text is encoded in binary will be very useful for this course :). You may also want to read the very short article Four Column ASCII. Then challenge yourself by writing "hello" in binary (in ASCII encoding) without checking the table.
- 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?
- Watch Richard Feynman's introductory lecture (1:15 hr) and test yourself by explaining the key ideas to a friend.

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 - Programming "Against the Metal" with Assembly Languages

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;
- What a calling convention is, and how a function call is made; and,
- How the programs we write interact with the operating system.

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*.

3 - 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'll also cover the "compile -> assemble -> link -> load" pipeline that's required to run even the simplest of C programs.

By the end of the class, you should be able to write simple C programs and anticipate at a high level the sequence of instructions to which they will compile.

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.

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*.

4 - A Brief Tour of Logic Circuits

You probably know that a computer works by combining binary 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

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'll aim for a simplified model without pipeline stages, which will be introduced in the following class.

We cover how the the core of MIPS instruction set (the datapath) is implemented with the combinatorial and sequential logic circuits we learnt 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 - Improving Performance with 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.

Prework

Please arrive in class having read P&H 4.5-4.8 or watched this lecture. To test your understanding, take at a piece of code you recently wrote (or write a simple program from scratch) and ask yourself which of the underlying operations are going to give rise to pipeline hazards? Which are amenable to some kind of branch prediction?

Further Resources

P&H goes into greater depth, and does so well, so we suggest continuing through to chapters 4.10-4.11.

For more on branch prediction, Dan Luu has some great notes providing a historical perspective and overall introduction.

If you're interested in the state of the art of branch prediction, you may wish to start by exploring perceptron (basically, neural network) based methods, starting with the paper Neural Methods for Dynamic Branch Prediction.

7 - Improving Performance with Caching

In this class, we explore one of the most important practical aspects of modern computer architectures: the “memory hierarchy” or use 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).

8 - Improving Performance with Parallelism

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.

Prework

For some context, please skim-watch these two lectures: 1 2. Again the concepts are more important than the details, and you can stop short of the OpenMP content as we won't be using it.

We *will* be writing a short program or two in CUDA, so please also read An Easy Introduction to CUDA C and C++. If you have access to an Nvidia GPU, please also try setting up CUDA and running a sample program.

Further Resources

For some more historical context (as well as ideas about what the future may hold) see the talk The Future of Microprocessors.

If you'd like to dive deeper into the world of parallel computing, our suggest starting point is the course *Intro to Parallel Computing with CUDA* on Udacity.