Zone 01 Athens

Computer Architecture and the Hardware/Software Interface

April 2024 with Matthew Stephenson

> *To understand a program you must become both the machine and the program.*
>
> Alan Perlis, Epigrams

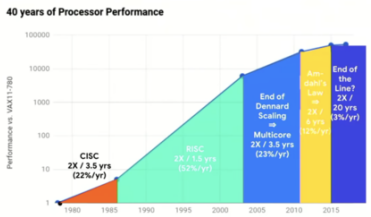
As software engineers, we study computer architecture to be able to understand *how our programs ultimately run*. Our immediate reward is to be able to write faster, more memory-efficient and more secure code.

Longer term, the value of understanding computer architecture may be even greater. Every abstraction between us and the hardware is leaky, to some degree. This course aims to provide a set of first principles from which to build sturdier mental models and reason more effectively.

We'll start mostly on the hardware side of the hardware/software interface, developing our understanding of how the machine works and writing assembly language programs to explore a typical instruction set architecture.[2] With a better understanding 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 most important performance consequences of modern architecture: CPU cache 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



Moore's Law gave some programmers an excuse to ignore computer architecture, by relying on faster underlying hardware each year. That era is over, and as John Hennessy argues in the talk from which the above graph is sourced, much of the burden for progress from this point is shifting to software systems.

[2] We'll primarily use MIPS as our example architecture, and introduce x86-64 towards the end of the course as a point of comparison. We chose MIPS because of its relative simplicity: it was one of the reduced instruction set computer (RISC) architectures, and has an order of magnitude smaller instruction set than x86-64.

MIPS is no longer a popular architecture for general purpose computing, but was used in many game consoles including the Nintendo 64, PlayStation and PlayStation 2, as well as the Mars rover. In many ways, MIPS inspired ARM, which is now the most popular computer architecture in the world by virtue of it being used in most phones.

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, very commonly used in undergraduate computer architecture courses. Chapter references are for the 5th edition, but older editions should be close in content.[3]

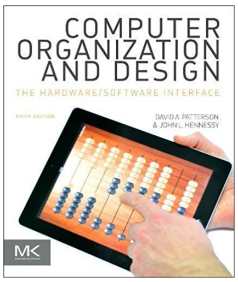
For those who prefer video-based courseware, our recommended supplement to our own course is the Spring 2015 session of Berkeley's 61C course "Great ideas in computer architecture" available on the Internet Archive.

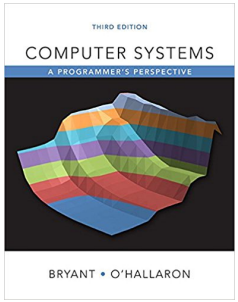
For students who have some extra time and would like to do some more project-based preparatory work, we recommend the first half of *The Elements of Computing Systems* (a.k.a. Nand2Tetris) which is available for free online.

For those with no exposure to C, we strongly recommend working through some of *The C Programming Language* (a.k.a "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.

[3] *Computer Organization and Design* is one of the most successful, lasting textbooks in all of computer science. The authors are living legends, having pioneered RISC and created MIPS. David Patterson now works as a researcher at Google after 40 years as a Professor at UC Berkeley, and John Hennessy was most recently President of Stanford before becoming Chairman of Alphabet.





1. The Fetch-Decode-Execute Cycle

TBD

Our first class aims to help you develop a high level understanding of the major components of a modern computer system—including those within the CPU itself—and how each is involved in the execution of a program.[5]

We will see that a useful model of program execution is that the CPU will repeatedly fetch an instruction from memory, decode it to determine which logic gates to utilize, then execute it and store the results. By the end of the class, you should be able to confidently describe this fetch-decode-execute process, including how the primary components of the system play a role in each step. You

[5] Broadly, the architecture of most modern computers is often but controversially called the "von Neumann architecture" after the prodigious mathematician and computer science pioneer John von Neumann. Specifically, von Neumann was a consultant on the EDVAC project and wrote a report about it that you could say went viral.

EDVAC was one of the first binary digital computers, and a successor to ENIAC which used decimal encodings but is considered by many to be the first digital computer.

should also be able to *simulate* this process by implementing a very simple virtual machine, which will you start in class and finish as post-class work.

Along the way, we will discuss how computers come to be architected this way, how they have evolved to perform this basic function at tremendous speed, and how the binary encodings of both instructions and data are tightly related to hardware components. These are themes that we will continue to explore throughout the course.

Pre-class Work

Prior to class, please do two things:

1. A diagram you have drawn of the main components of a computer, and how they are connected; and,
2. A paragraph or two of prose describing your understanding of the fetch-decode-execute cycle, and how the relevant components of the computer are involved in each step.

In both cases, please go into as much detail as possible! We hope that you will spend at least an hour or two researching the topic and pushing your understanding as you draw the diagram and write the description.

There are many resources that you could use as a starting point, but there are two in particular that we recommend: Richard Feynman's introductory lecture (1:15 hr) and the article How Computers Work: The CPU and Memory.[6] The first is very conceptual; the second is more concrete. Both are useful angles.

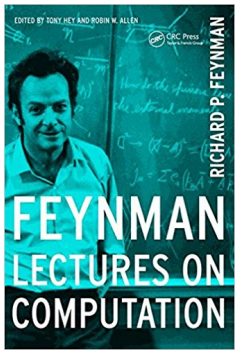
While watching the Feynman lecture, you may wish to ask yourself which actual physical components correspond to each of Feynman's metaphors. For instance, what are the physical equivalents of the "cards" that Feynman describes, and what hardware is used to "file" as opposed to "process" these cards?

While reading the article, please look up any terms or concepts where your understanding is at all vague.

We hope that this reading takes you down a long path of discovery, but don't forget to incorporate your discoveries into your diagram and description, and submit them to your instructor!

[6] Richard Feynman is of course better know for his contributions to physics, but he did spend some time thinking about computing, including a lecture series on computation referenced in the further readings for this class, and some work on the Connection Machine. Like von Neumann and Oppenheimer above, Feynman too worked on the Manhattan Project—he was 24 when he joined.

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 onward.

2. Lab: Writing a Virtual Machine

TBD

This lab-style class aims to consolidate your understanding of the fetch-decode-execute cycle by having you implement a simple virtual machine.

Pre-class Work

In preparation, please review any new concepts from last class, and read the in-class exercise instructions.

In-class Exercise

In class, you will begin to write a virtual machine for a very simple architecture that we have designed. The purpose of this exercise is to consolidate your understanding of the fetch-decode-execute model by writing a program that emulates it!

Post-class Work

Please complete your virtual machine implementation as we will be building upon it in a later class.

3. Binary Encodings of Data

TBD

In this class, we'll 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 process must also have binary representations. Furthermore, it must be possible for hardware implementations of operations like addition and multiplication to manipulate these representations efficiently.

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 byte ordering or "endianness"; and,
- Examine and interpret sophisticated binary formats such as network protocol formats and binary files.

Pre-class Work

Please watch the videos below and answer the corresponding questions.

Watch **15 and Hexadecimal numbers** (8 min) and test yourself by converting the numbers 9, 136 and 247 to hexadecimal. Then answer: in CSS, how many colors can be represented in the hexadecimal form? How about in the RGB form? If you were given a hex code, could you guess the color?

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. 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 -3 - 4 ("negative three, minus four"). What are the largest and smallest numbers representable in 32 bit 2's complement?

Watch at least the first 5 minutes of **Byte ordering**. Then consider that 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 TCP uses big-endian or little-endian integers? How would you represent port 3000?

IEEE Floating Point (9 min) is a ubiquitous format used to represent numbers ranging from the unimaginably subatomic to well beyond astronomical. It borrows extensively from the idea of scientific notation; keeping that in mind may aid your understanding. What are the largest and smallest values representable with 64-bit floats?

Finally, watch **The Unicode Miracle** (10 min). UTF-8 is an impressive an highly successful scheme for encoding all of the more than 1 million valid Unicode code points, while remaining entirely backwards compatible with ASCII. Is there any additional space cost to encoding a purely ASCII document as UTF-8? What are the pros and cons detriments of UTF-8 compared to another encoding for Unicode such as UTF-32?

Please also read the in-class exercise instructions.

In-class Exercise

The class exercise will involve a series of challenges requiring you to convert between various data encoding formats.

Post-class Work

Much like the in-class exercises, the post-class work involves a series of small challenges requiring you to understand and convert between various binary data encodings.

Further Resources

What Every Computer Scientist Should Know About Floating-Point Arithmetic is a classic and stunningly detailed paper covering the

many rough edges and surprising behaviors of the standard.

Joel Spolsky on Unicode and character sets is another great resource for understanding character encoding from a programmer's perspective.

4. Introduction to Assembly Programming

TBD

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.[8]

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.

Pre-class Work

We will be writing short MIPS assembly programs in class. For a background in MIPS, watch the CS 61C "MIPS Intro" lectures 1 and 2 (2.5 hrs total) or read P&H 2.1-2.3 and 2.5-2.9.

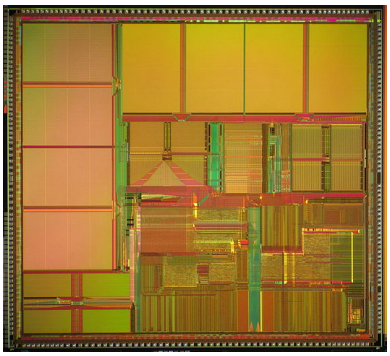
Since we don't have MIPS CPUs ready to execute our programs directly, you will be using the MARS emulator to assemble and run your programs. Please install it.

To very both that you have MARS working correctly and that you've understood the basics of MIPS, please write a program which can calculate the volume of a cylinder given its height and radius.[9]

Please also read the in-class exercise instructions.

[8] We use MIPS as it's a *simple but real* architecture, used for instance in the Playstation and other game consoles of the same era, and the Mars rover. It is also inspired 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.

Pictured below is a die shot of the QED RM7000 MIPS microprocessor, one of the first to feature an integrated L2 cache, clearly visible. The photo is by Paulo Rautakorpi, who has taken many others.



[9] For simplicity, you can hardcode the value of pi to the integer 3, and explicitly load the height and radius variables into registers at the start of your program.

As stretch goals, you may want to try using floating point instructions for the calculation, and/or to accept the parameters from the user as input during program execution.

In-class Exercise

The class exercise will involve writing a few short programs in MIPS assembly, to become familiar with the basic instructions and MARS emulator.

Post-class Work

To better understand function calls and stack use, the post-class work will involve implementing recursion in MIPS assembly.[10]

[10] In case you've not yet stopped to think about it: a typical computer is an inherently iterative device. The clock signals to the system that it must loop! If we wish for recursion, we must implement it.

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.[11] 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*.

[11] SHENZHEN I/O is a fantastic game by Zach Barth, where you play an American computer engineer who has moved to Shenzhen and is tasked with building small devices like e-cigarettes and garage door openers by wiring together hardware components and writing small assembly programs. Zach Barth is also responsible for the assembly programming game TIS-100, and other puzzle games including Opus Magnum and Infinifactory. You may also enjoy his talk Zachtronics: Ten Years of Terrible Games.

5. A Brief Tour of Logic Circuits

TBD

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? This class starts to demystify this 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; and,

- Explain how circuits (like adders) can be constructed to affect *arithmetic* through combinational logic.

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.

Pre-class Work

In preparation for class, please research and determine the truth tables for the following 6 logic gates: NOT, NAND, AND, OR, NOR and XOR. For all gates other than NAND, draw a circuit diagram to demonstrate how it can be constructed out of one or more NAND gates. A simple reference can be found here.[12]

[12] Note that this reference uses the shorthand EOR rather than XOR for exclusive or.

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?

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

In-class Exercise

The class exercise will involve building simple circuits.

Post-class Work

Complete any remaining component diagrams from the class exercise. As a stretch goal, research adders that perform better than the ripple carry adder we constructed, and draw a diagram of one.

Further Resources

The first three chapters of Nand2Tetris correspond roughly to the topics covered in this class. Building these gates in a hardware

description language[13] 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 KOHCT-PYKTOP: 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.

[13] Hardware systems are typically far too complicated to develop without extensive software tools. Typically, a computer engineer will use a hardware description language such as VHDL or Verilog to describe the structure of hardware components, which can then be named and incorporated into higher level components. VHDL/Verilog descriptions can be used for simulation and debugging, and ultimately to create the masks from which integrated circuits are printed.

One of the key features of the nand2tetris course is that you can develop a more concrete sense of how circuits are designed by actually doing so using a custom, simplified hardware description language.

6. Sequential Circuit Components

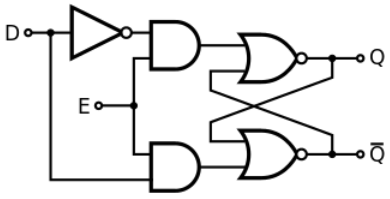
TBD

In our last class, we saw how meaningful computation like addition can be built up from more primitive combinational circuits, and ultimately NAND gates. However, a mystery still remains… How can we remember values between one CPU cycle and another?[14] This class extends our understanding of hardware components into the weird world of sequential circuits—those which retain state from one cycle to another.

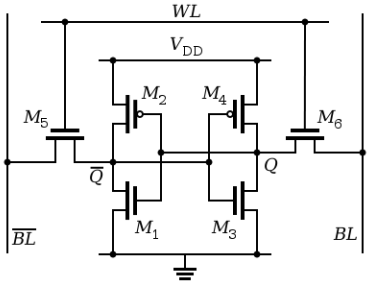
By the end of this class, you will understand how a flip-flop works, and how they can be used to give our circuits "memory".

[14] Many students are surprised to discover that it takes multiple transistors to remember a single bit of data, at least in the SRAM typically used for registers and CPU caches.

Here is a logical representation of such a circuit, constructed from AND and NOR gates:



Here is a typical CMOS SRAM implementation, using six transistors:



DRAM typically uses a simpler construction, with a single transistor to set the value and a capacitor to store it.

Pre-class Work

In preparation for class, please first ensure that you're comfortable with the concepts covered last class.

Then, please 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.

Please also read the in-class exercise instructions.

In-class Exercise

The class exercise will involve building simple sequential circuits like latches and flip-flops.

Post-class Work

Complete any remaining component diagrams from the class exercise. As a stretch goal, research adders that perform better than the ripple carry adder we constructed, and draw a diagram of one.

Further Resources

The first three chapters of Nand2Tetris correspond roughly to the topics covered in this class. Building these gates in a hardware description language[15] 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 KOHCT-PYKTOP: 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.

[15] Hardware systems are typically far too complicated to develop without extensive software tools. Typically, a computer engineer will use a hardware description language such as VHDL or Verilog to describe the structure of hardware components, which can then be named and incorporated into higher level components. VHDL/Verilog descriptions can be used for simulation and debugging, and ultimately to create the masks from which integrated circuits are printed.

One of the key features of the nand2tetris course is that you can develop a more concrete sense of how circuits are designed by actually doing so using a custom, simplified hardware description language.

7. The Structure of a Simple CPU

TBD

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.[16]

[16] To a close approximation, a CPU is made up of the datapath—which stores and processes the meaningful data for instructions—and the control path—which orchestrates the whole process by sending appropriate signals.

Pre-class Work

Please either read P&H 4.1-4.4 or watch this lecture before class. To test your understanding, write a paragraph describing how the

instruction `lw $r1, 10($r2)` progresses through each stage of execution.

At this point, you may also wish to test your overall understanding, by trying to explain "how a computer works" to a curious 12 year old. By now, no aspect of that explanation should be a complete mystery to you by now, although there may be many more details that you'd like to fill in!

Please also read the in-class exercise instructions.

In-class Exercise

This class involves designing a simple single-cycle CPU.

Post-class Work

Please complete the class exercise, filling in any missing details and incorporating your total understanding into a final diagram.

Further Resources

This CS 61C lecture goes into more detail on the same topics.

For a look at how CPU designs have evolved over time, watch this 2016 talk by Sophie Wilson who is one of the inventors of the ARM architecture. The talk discusses the history of computer architecture, and makes some predictions for the future.

This article by former MIT professor, Rodney Brooks, discusses the history of Moore's law, why we've reached the end of Moore's Law, and what its end might mean for the future of computer architecture. This is a good complement to the Sophie Wilson talk.

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".

8. Introduction to Operating Systems: Booting, Loading, and Virtual Memory

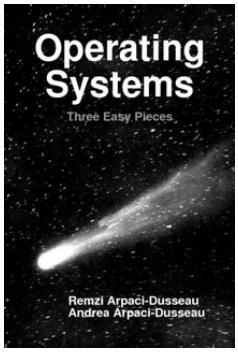
TBD

In this class we peek one level above the architecture of a computer into the operating system. Modern operating systems are so effective at fulfilling their responsibilities 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 develop a model of how programs are typically initialized, scheduled and managed.

By the end of class you will be able to:

- Describe what happens when a computer turns on;
- Describe how an operating system loads and executes a program;
- Describe the concept of "virtual memory" and some strategies for implementing it; and,
- Simulate loading and execution of a program in a virtual machine that uses address translation.

Pre-class Work



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

- Chapter 4: The Process
- Chapter 6: Limited Direct Execution
- Chapter 15: Address Translation

Test your understanding by writing a paragraph or two explaining a typical operating system's process model, and another explaining a typical virtual memory model. These two concepts will be our focus for our quick tour of what is an incredibly complex domain, so if we all have a reasonable high-level understanding before class, we should be able to explore some more interesting details together.

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

Further Resources

We of course recommend our own *Operating Systems* course. In the meantime, the entire OSTEP textbook is very approachable and has been made available for free by the authors. Focusing on memory virtualization, a great way to further your understanding would be to first read these 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—click the malloc self study handout and the malloc writeup links for any of the labs you are interested in.

For more on the initialization process, see Demystifying the init system, where the author strives to understand the init process by writing a simple version of one in ruby.

9. Lab: Writing a Virtual Operating System

TBD

In this lab-style class, you will continue to explore operating systems concepts like CPU scheduling and virtual memory, by extending your virtual machine into a kind of "virtual operating system" with these features.

Pre-class Work

In preparation, please review any unfamiliar concepts from last class, and read the in-class exercise instructions.

In-class Exercise

The exercise for this class involves adding virtual threads and virtual memory to your virtual machine implementation from earlier.

Post-class Work

It's unlikely that you will have completed the virtual operating system in class. Please complete it afterwards.

10. An Overview of C, the Portable Assembly Language

TBD

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.[19] 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 re-examine 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`).[20]

Pre-class Work

Please come to class familiar with the basic syntax of C.[21] If you can get a hold of a copy of *The C Programming Language* (K&R C),[22] 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 a few C programming problems on exercism.io.

Then, write two short programs: one called `strlen` which determines the length of a given string, and one called `linecount` which counts the number of lines in the file with a given filename.

In-class Exercise

In this class you will be asked to write a short C program to convert a string of binary digits into its decimal value. You will then work use the tools in your compiler toolchain to do step-through debugging, disassembling and performance optimization.

[19] Higher level than assembly, that is! C provides functions, arrays and pointers, structs, looping and conditional constructs and a syntax that isn't tied to any one computer architecture, but little else.

[20] The GNU toolchain is a very well established collection of programming tools including GCC (the "GNU Compiler Collection") and GDB ("The GNU Project Debugger"). LLVM is a newer collection of tools—including clang and lldb—designed to be more flexible and modular than some of its GNU equivalents. It was originally developed by Chris Lattner, who also designed the Swift programming language.

[21] Our assumption is that everybody will be comfortable with types and basic control flow, and that we can spend more time on more interesting topics like pointers, arrays and functions.

[22] *The C Programming Language* is in many ways the ideal programming book. It is short and very clearly written, co-authored by Brian Kernighan who co-created the AWK language and contributed to Unix, and Dennis Richie who designed C and wrote the first implementation. Until C was standardized in 1989, the book itself was considered the standard.

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.

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.[23]

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

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

Finally, for a look at how the lessons of C can be extended to higher level languages, consider these two articles about how JavaScript manages classes and closures in memory.

[23] Brian Harvey was—until his recent retirement—a legendary teacher at Berkeley, having taught the CS 61A *Structure and Interpretation of Computer Programs* course for almost three decades. He also contributed to the Logo and Snap programming languages.

11. Lab: C Programming and Reverse Engineering

TBD

This lab-style class continues our crash course in the C programming language, this time focused on reverse engineering C programs by examining disassembled binaries. This will also provide an introduction to the x86-64 instruction set architecture.

Pre-class Work

In preparation, please review any unfamiliar concepts from last class, and read the in-class exercise instructions.

In-class Exercise

In class, you will reverse engineer two compiled object files by disassembling them.

Further Resources

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.

12. The Memory Hierarchy

TBD

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 hardware caches to mitigate the cost of accessing main memory. For many common workloads, a program's rate of hardware cache misses can be the greatest cause of poor performance. We cover the reason for the innovation, how the caches operate, and most practically 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.

Pre-class Work

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.

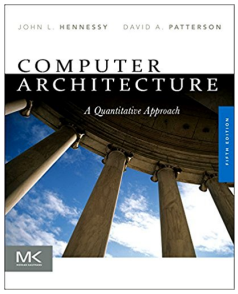
In class we'll also be using Cachegrind (a Valgrind tool) to measure the cache utilization of the code that we'll be optimizing. Please make sure that you have Valgrind installed and able to run Cachegrind on a real program.[24]

[24] If you are using a recent version of macOS, you may need to compile Valgrind from source to obtain a working version. The latest release (or the one available via Homebrew) is unlikely to be up to date. As a worst case option, you may want to consider running Valgrind on Linux in VirtualBox.

In-class Exercise

The exercise for this class involves optimizing a few programs based on your understanding of cache behavior.

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). Berkeley CS 61C covers caches over three classes: 1 2 and 3.

Epilogue

We hope that this is just the beginning of your computer architecture journey! All of the "further resources" above are well worth exploring, in our opinion, but we particularly recommend *Computer Systems: A Programmer's Perspective* and its corresponding labs.

For just a glimpse of what the future of computing may resemble, see this talk at Google I/O 2018 by John Hennessy. In particular, we should expect a greater emphasis on hardware-conscious software systems as well as domain-specific hardware in the future, as the gains from Dennard scaling wind down. Machine learning and 3-D computer graphics in particular present both challenges and opportunities for system designers.

As a foundational course, your study of *Computer Architecture* should set you up well for many other 01-edu student led courses, in par-ticular the systems track courses like Databases, Operating Systems, Languages and Compilers and Distributed Systems (available upon request; will create if there's enough demand)