Bradfield
SCHOOL of COMPUTER SCIENCE

Introduction to Computer Systems

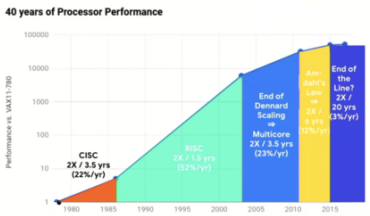
August 2023 with Elliott Jin and Oz Nova

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

As software engineers, we study computer systems 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 systems 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.

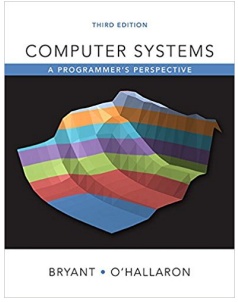


Moore's Law gave some programmers an excuse to ignore computer architecture, by relying on faster underlying hardware each year. That era is over, and as John Hennessy argues in the talk from which the above graph is sourced, much of the burden for progress from this point is shifting to software systems.

[2] We'll consider a subset of Intel's x86-64 architecture, which for decades has been the most popular architecture for servers, consumer desktops, and most laptops.

Recommended Resources

Please endeavor to complete all of the pre-class work for each class, which generally consists of a programming exercise as well as supplementary reading. Depending on your preferred style of learing, you may wish to dive straight into the programming, or to first do the background reading. In either case, if everyone makes an



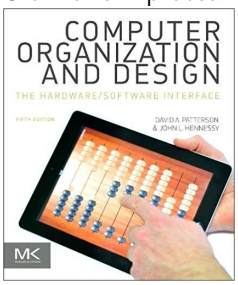
earnest attempt to complete the exercise, our live classes will end up being richer, deeper and more insightful.

The main text for this class is *Computer Systems: A Programmer's Perspective* ("CS:APP"). Please note that the cheaper "international" version of the book has different exercises often with incorrect solutions.

For a more hardware-focussed supplementary text, we suggest *Computer Organization and Design* by Patterson and Hennessy ("P&H")—a classic text, very commonly used in undergraduate computer architecture courses.[4] Chapter references are for the 5th edition, but older editions should be close in content.

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.

[4] P&H is one of the most successful textbooks in all of computer science. The authors received the 2017 Turing Award for their pioneering work on RISC (reduced instruction set computing). 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. Introduction

Monday, 7 August 2023

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 should also be able to *simulate* this process by implementing a very simple virtual machine.

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.

[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.
Sadly, J. Presper Eckert and John Mauchly who in fact created the "von Neumann architecture" for the ENIAC are not as well known.
Below is a photo of John von Neumann standing with J. Robert Oppenheimer in front of the IAS machine, built under von Neumann's direction based on the design in the EDVAC report. IAS was built at the Institute of Advanced Studies at Princeton. Oppenheimer and von Neumann previously worked together on the Manhattan Project.

Pre-class Work

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

As supplementary reading, we suggest *CS:APP* chapter 1 "A Tour of Computer Systems", and in particular sections 1.1 to 1.4.

Further Resources

There are many additional resources that you might find useful to better understand the execution of a computer at a very high level, as we are aiming to do with this class. Two shorter resources in particular that we recommend are 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.

There are also several books that provide a good high-level introduction to how computers work: 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.

If you'd like other comparable examples of writing a simple VM, see the article Write your Own Virtual Machine as well as A Python Interpreter Writen in Python from *50 Lines or Less*.

[6] Richard Feynman is of course better know for his contributions to physics, but he did spend some time thinking about computing, including the lecture series *Feynman Lectures on Computation* 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.

2. Introduction

Monday, 7 August 2023

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.[7]

We will see that a useful model of program execution is that the CPU will repeatedly fetch an instruction from memory, decode it to

[7] 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

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 should also be able to *simulate* this process by implementing a very simple virtual machine.

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

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

As supplementary reading, we suggest *CS:APP* chapter 1 "A Tour of Computer Systems", and in particular sections 1.1 to 1.4.

Further Resources

There are many additional resources that you might find useful to better understand the execution of a computer at a very high level, as we are aiming to do with this class. Two shorter resources in particular that we recommend are Richard Feynman's introductory lecture (1:15 hr) and the article How Computers Work: The CPU and Memory.[8] The first is very conceptual; the second is more concrete. Both are useful angles.

There are also several books that provide a good high-level introduction to how computers work: 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.

If you'd like other comparable examples of writing a simple VM,

[8] Richard Feynman is of course better know for his contributions to physics, but he did spend some time thinking about computing, including the lecture series *Feynman Lectures on Computation* 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.

see the article Write your Own Virtual Machine as well as A Python Interpreter Writen in Python from *50 Lines or Less*.

3. Binary Encodings of Data

Thursday, 10 August 2023

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

In preparation for class, please complete the following short exercises. We will use the class primarily to troubleshoot any issues and clarify confusing concepts.

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. Binary Encodings of Data

Thursday, 10 August 2023

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

In preparation for class, please complete the following short exercises. We will use the class primarily to troubleshoot any issues and clarify confusing concepts.

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.

5. Introduction to Assembly Programming

Monday, 14 August 2023

This class introduces the x86-64 architecture,[9] and provides an opportunity for us to write programs "close to the metal" in x86-64 assembly language.

Assembly language has become an abstraction layer, and a modern microprocessor may perform many microoperations to execute a single machine code instruction, but assembly language remains one of the best ways for us as software engineers to reason about the execution of our programs. In this class, we'll primarily write short programs to help us understand how our higher level programming constructs are interpreted by the machine.

By the end of the class, you should understand:

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

Pre-class Work

In preparation for class, you will write a series of short assembly programs in x86-64 assembly, using *CS:APP* chapters 3.1-3.6 as a reference.

[9] This is the 64-bit version of Intel's x86 architecture, an astonishingly popular architecture first introduced with the Intel 8086, released in 1978. Modern x86 machines are largely backwards compatible with programs written for these older machines, including the Intel 386 or 486 which were the first computers that many of us used.

6. Introduction to Assembly Programming

Monday, 14 August 2023

This class introduces the x86-64 architecture,[10] and provides an opportunity for us to write programs "close to the metal" in x86-64 assembly language.

Assembly language has become an abstraction layer, and a modern microprocessor may perform many microoperations to execute a single machine code instruction, but assembly language remains one of the best ways for us as software engineers to reason about the execution of our programs. In this class, we'll primarily write short

[10] This is the 64-bit version of Intel's x86 architecture, an astonishingly popular architecture first introduced with the Intel 8086, released in 1978. Modern x86 machines are largely backwards compatible with programs written for these older machines, including the Intel 386 or 486 which were the first computers that many of us used.

programs to help us understand how our higher level programming constructs are interpreted by the machine.

By the end of the class, you should understand:

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

Pre-class Work

In preparation for class, you will write a series of short assembly programs in x86-64 assembly, using *CS:APP* chapters 3.1-3.6 as a reference.

7. More Assembly Programming

Thursday, 17 August 2023

In this class, we'll continue our exploration of assembly programming, including topics such as recursion, array access and floating point operations.

Pre-class Work

In preparation for class, please revise any unclear concepts from last class, and work through the exercises linked below.

Further Resources

One of the best options for further study is to complete some of the x86-64 exercises on exercism.io. As an alternative approach, there are two *games worth playing*: SHENZHEN I/O and Human Resource Machine.[11] Both use much simpler instruction sets than x86-64, but solving the programming problems in either will help train you to think at the level of a typical machine.

[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.

8. More Assembly Programming

Thursday, 17 August 2023

In this class, we'll continue our exploration of assembly programming, including topics such as recursion, array access and floating point operations.

Pre-class Work

In preparation for class, please revise any unclear concepts from last class, and work through the exercises linked below.

Further Resources

One of the best options for further study is to complete some of the x86-64 exercises on exercism.io. As an alternative approach, there are two *games worth playing*: SHENZHEN I/O and Human Resource Machine.[12] Both use much simpler instruction sets than x86-64, but solving the programming problems in either will help train you to think at the level of a typical machine.

[12] 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.

9. Low Level Optimization

Monday, 21 August 2023

One of the most enjoyable aspects of understanding computer architecture is using a machine's characteristics to your advantage to write faster programs. While we generally expect our compilers to do this for us, there is a limit to what a compiler can understand, so we still have a responsibility, when writing high level code, to understand the performance implications of the machine code that will be generated.

In this class, we explore some techniques for program optimization, both directly in assembly and as a "nudge" to the compiler. We will also be forced to increase the resolution of our model of the computer, incorporating ideas like micro-operations, branch prediction and out-of-order execution.

Pre-class Work

In preparation for class, please work through the following short exercises, as well *CS:APP* chapter 5. In this case, we suggest that you at least read *CS:APP* chapter 5 even if you manage to complete the exercises without doing so, as it covers a significant amount of conceptual background.

Further Resources

We highly recommend Agner Fog's optimization manuals, in particular "Optimizing subroutines in assembly language" for an indepth look at some of the optimization techniques we discussed.

10. Low Level Optimization

Monday, 21 August 2023

One of the most enjoyable aspects of understanding computer architecture is using a machine's characteristics to your advantage to write faster programs. While we generally expect our compilers to do this for us, there is a limit to what a compiler can understand, so we still have a responsibility, when writing high level code, to understand the performance implications of the machine code that will be generated.

In this class, we explore some techniques for program optimization, both directly in assembly and as a "nudge" to the compiler. We will also be forced to increase the resolution of our model of the computer, incorporating ideas like micro-operations, branch prediction and out-of-order execution.

Pre-class Work

In preparation for class, please work through the following short exercises, as well *CS:APP* chapter 5. In this case, we suggest that you at least read *CS:APP* chapter 5 even if you manage to complete the exercises without doing so, as it covers a significant amount of conceptual background.

Further Resources

We highly recommend Agner Fog's optimization manuals, in particular "Optimizing subroutines in assembly language" for an indepth look at some of the optimization techniques we discussed.

11. The Memory Hierarchy

Thursday, 24 August 2023

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

In preparation for class, please complete the following exercises, where you will investigate the cache utilization of a couple of short programs, and refactor them to improve their performance. For supporting material, we suggest chapters 6.1-6.3 of *CS:APP*.

Further Resources

For an alternative explanation of the memory hierarchy, see the blog post Why do CPUs have multiple cache levels? by Fabian Giesen. This should help you rationalize why we've settled (for now) on the configuration of CPU caches that you'll typically see.

For motivation and context on why this matters for programmers, watch this talk by Mike Acton.

For more on Cachegrind, see the cachegrind manual.

12. The Memory Hierarchy

Thursday, 24 August 2023

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

In preparation for class, please complete the following exercises, where you will investigate the cache utilization of a couple of short programs, and refactor them to improve their performance. For supporting material, we suggest chapters 6.1-6.3 of *CS:APP*.

Further Resources

For an alternative explanation of the memory hierarchy, see the blog post Why do CPUs have multiple cache levels? by Fabian Giesen. This should help you rationalize why we've settled (for now) on the configuration of CPU caches that you'll typically see.

For motivation and context on why this matters for programmers, watch this talk by Mike Acton.

For more on Cachegrind, see the cachegrind manual.

13. The Memory Hierarchy 2

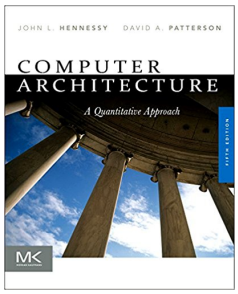
Monday, 28 August 2023

Since understanding the memory hierarchy has become so important, we devote a second class to understanding it in greater depth, including the implications of how caches are implemented, and some more techniques for writing cache friendly code.

Pre-class Work

In prepration for class, please revise any unclear material from last class, and work through the remainder of chapter 6 of *CS:APP*. When you are ready, please attempt the below exercise, where you will improve the performance of a program written in Go.

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.

14. The Memory Hierarchy 2

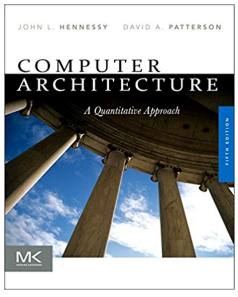
Monday, 28 August 2023

Since understanding the memory hierarchy has become so important, we devote a second class to understanding it in greater depth, including the implications of how caches are implemented, and some more techniques for writing cache friendly code.

Pre-class Work

In prepration for class, please revise any unclear material from last class, and work through the remainder of chapter 6 of *CS:APP*. When you are ready, please attempt the below exercise, where you will improve the performance of a program written in Go.

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.

15. Exceptional Control Flow

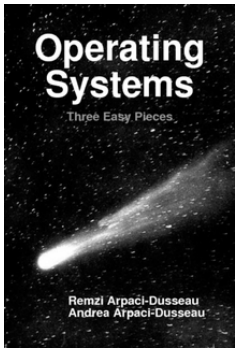
Thursday, 31 August 2023

In this class we peek one level above the architecture of a computer into the operating system. Our current model of program execution is that the CPU simply fetches, decodes and executes instructions forever. In order to understand how an operating system can coordinate multiple concurrent processes, manage process lifecycles and handle errors (all with the help of the hardware) we must understand exceptional control flow.

Pre-class Work

In preparation for class please, you will write a simple shell, focusing on the challenge of managing multiple shell jobs using signals. For the relevant background, work through any unfamiliar sections of chapter 8 of *CS:APP*.

Further Resources



In many ways, this class serves as a preview of our module on *Operating Systems*, where we use the *Operating Systems: Three Easy Pieces* book as a reference. You may find it useful, now, as an additional resource, in particular Chapter 4: The Process and Chapter 6: Limited Direct Execution.

For a better understanding of how a computer boots, see this article.

16. Exceptional Control Flow

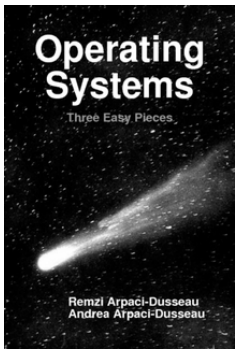
Thursday, 31 August 2023

In this class we peek one level above the architecture of a computer into the operating system. Our current model of program execution is that the CPU simply fetches, decodes and executes instructions forever. In order to understand how an operating system can coordinate multiple concurrent processes, manage process lifecycles and handle errors (all with the help of the hardware) we must understand exceptional control flow.

Pre-class Work

In preparation for class please, you will write a simple shell, focusing on the challenge of managing multiple shell jobs using signals. For the relevant background, work through any unfamiliar sections of chapter 8 of *CS:APP*.

Further Resources



In many ways, this class serves as a preview of our module on *Operating Systems*, where we use the *Operating Systems: Three Easy Pieces* book as a reference. You may find it useful, now, as an additional resource, in particular Chapter 4: The Process and Chapter 6: Limited Direct Execution.

For a better understanding of how a computer boots, see this article.