Essential Computer Science

A Programmer's Guide to Foundational Concepts

Paul D. Crutcher
Neeraj Kumar Singh
Peter Tiegs

Apress®

Essential Computer Science

A Programmer's Guide to Foundational Concepts

Paul D. Crutcher
Neeraj Kumar Singh
Peter Tiegs

Apress®

Essential Computer Science: A Programmer's Guide to Foundational Concepts

Paul D. Crutcher
Welches, OR, USA

Neeraj Kumar Singh
Bangalore, Karnataka, India

Peter Tiegs
Hillsboro, OR, USA

ISBN-13 (pbk): 978-1-4842-7106-3
ISBN-13 (electronic): 978-1-4842-7107-0
https://doi.org/10.1007/978-1-4842-7107-0

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Susan McDermott
Development Editor: Laura Berendson
Coordinating Editor: Rita Fernando

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, New York, NY 10004. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484271063. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

To my wife, Lisa, for her unending support and encouragement; to my sons currently studying computer science, Kyle and Cameron, may this be a bit of inspiration for their journey in life; and to the memory of my father, Edwin Lee Crutcher, who passed away while I worked on this book. I love you, Dad!

—Paul

To my wife, Shilpi, for her unwavering support.

—Neeraj

To Karen, Jane, Josephine, Henri, and Jeanette, my family, for all of their support, patience, and encouragement.

—Peter

Table of Contents

About the Authors

Paul D. Crutcher is a senior principal engineer at Intel Corporation, managing the Platform Software Architecture team in the Client Computing Group (CCG). Paul has worked at Intel for more than 25 years and has also worked at two smaller software companies. Paul has a degree in computer science, with expertise spanning software development, architecture, integration, and validation based on systems engineering best practices in multiple areas. He holds several patents and has multiple papers and presentations to his credit.

Neeraj Kumar Singh is a principal engineer at Intel with more than 15 years of system software and platform design experience. His areas of expertise are hardware/software (HW/SW) codesign, system/platform architecture, and system software design and development. Neeraj is the lead author of two other books, *System on Chip Interfaces for Low Power Design* and *Industrial System Engineering for Drones: A Guide with Best Practices for Designing*, in addition to many other papers and presentations.

Peter Tiegs is a principal engineer at Intel with around 20 years of software experience. Inside Intel, he often consults on DevOps topics such as build automation and source code branching. Over the last decade, Peter evangelized continuous integration and delivery as well as agile practices at Intel. Peter has written software at all levels of the stack from embedded C code to Vue.js. His programming language of choice is Python.

About the Contributors

Chockalingam Arumugam is a system software architect with expertise in design, development, and delivery of software solutions that work across OSs. He holds a master's degree in software systems from Birla Institute of Technology and a bachelor's degree in electronics and communications from Anna University. He is a hands-on technologist on OS-agnostic software development and has over 12 years of experience in the industry. In recent years, he has been specializing in cloud-based telemetry solutions.

Through his career, he has worked on a broad set of domains, including device drivers, firmware/platform services, desktop/universal applications, web applications, and services. He specializes in the areas of Platform Health Analytics, Windows crash decode, and thermal and power management debug and has led multiple engagements in these areas. These solutions are used extensively in the industry for client platform validation and debug. He is currently based out of Bangalore, India, and works at Intel Corporation.

Prashant Dewan is a principal engineer at Intel and is very passionate about computer security. At Intel, he has worked on multiple security technologies and has filed 100+ patents in the area of computer security. He has a master's and doctorate in computer science from Arizona State University.

About the Technical Reviewer

Kenneth Knowlson is a senior principal engineer in the Client Computing Group (CCG) division at Intel. He leads a group of principal engineers in the Analytics and DevOps subgroup, within CCG, leading the organization's strategic and technical direction in these dynamic areas. Prior to joining CCG, Ken invented the processes and procedures for "pre-silicon" (Pre-Si) software and system development at Intel. The Pre-Si initiative is focused on accelerating time to market by shifting SW and FW development "left," before Si is available, enabling products to come to market much faster than they would otherwise. Pre-Si uses technologies like Virtual Platform, FPGA, and System-Level Emulation to approximate the final Si-based product. Ken also has a long history at Intel creating and delivering consumer-connected media products streaming media space.

Ken holds bachelor's degrees in mathematics and physics from the University of California Santa Cruz. Ken enjoys swimming and running and also holds black belts in taekwondo and hapkido, although he no longer practices.

Acknowledgments

We would like to express gratitude to the people who helped us through this book, some of them directly and many others indirectly. It's impossible to not risk missing someone, but we will attempt anyway.

First and foremost, we would like to sincerely thank our technical reviewer, Ken Knowlson, for meticulous reviews; it helped the book significantly. Thank you, Ken!

We would like to acknowledge Prashant Dewan for writing Chapter 6 and Chockalingam A. for his help on Chapter 4 of the book.

Thank you so much Rita Fernando, Susan McDermott, and all of the Apress publishing team for the outstanding work, help, guidance, and support; you have gone the extra mile to make the book what it is.

Above all, we thank our family and friends for their understanding and support and for being continuous sources of encouragement.

Introduction

According to code.org, there are 500,000 open programming positions available in the United States alone – compared to an annual crop of just 50,000 graduating computer science majors. The US Department of Labor predicted there will be 1.4 million computer science jobs by 2020, however, only enough people to fill roughly 30% of these jobs. To bridge the gap, many people not formally trained in computer science are employed in programming jobs. While they are able to start programming and coding quickly, it often takes them time to acquire the necessary understanding and gain the requisite skills to become an efficient computer engineer or advanced developer.

The goal of the book is to provide the essential computer science concepts and skills necessary to develop a sound understanding of the field. It focuses on the foundational and fundamental concepts upon which expertise in specific areas can be developed, including computer architecture, programming language, algorithm and data structure, operating systems, computer networks, distributed systems, security, and more.

This is a must-read for computer programmers lacking formal education in computer science. Secondarily, it is a refresher for all, including people having formal education in computer science as well as anyone looking to develop a general understanding of computer science fundamentals.

Overall, we authors have attempted to make it as lucid as possible, so people with limited or even no background in computer science can pick up the book and go through the journey to develop a good understanding of computer science. We're excited to have you on board.

CHAPTER 1

Fundamentals of a Computer System

There are many resources online to get you started programming, but if you don't have training in computer science, there are certain fundamental concepts that you may not have learned yet that will help you avoid getting frustrated, such as choosing the wrong programming language for the task at hand or feeling overwhelmed. We wrote this book to help you understand computer science basics, whether you already started programming or you are just getting started. We will touch on the topics someone with a computer science degree learns above and beyond the semantics and syntax of a programming language. In this first chapter, we will cover a brief history and evolution of a computer system and the fundamentals of how it operates. We will cover some low-level computer architecture and programming concepts in this chapter, but subsequent chapters will cover higher-level programming concepts that make it much easier to program the computer.

von Neumann Architecture

You've probably heard stories about computers the size of an entire room in the 1940s into the 1970s, built with thousands of vacuum tubes, relays, resistors, capacitors, and other components. Using these various

P. D. Crutcher et al., *Essential Computer Science*,
https://doi.org/10.1007/978-1-4842-7107-0_1

components, scientists invented the concept of gates, buffers, and flip-flops, the standard building blocks of electronic circuits today. In the 1970s, Intel invented the first general-purpose microprocessor, called the 8088, that IBM used to make the first PC that was small enough for personal use. Despite the continuous advancements that have made it possible to shrink the microprocessor, as you'll see, the core elements of today's desktop or laptop computer are consistent with the first computers designed in the 1940s!

In 1945, John von Neumann documented the primary elements of a computer in the "First Draft of a Report on the EDVAC" based on the work he was doing for the government. EDVAC stands for Electronic Discrete Variable Automatic Computer, which was the successor to the Electronic Numerical Integrator and Computer (ENIAC), the first general-purpose computer developed during World War II to compute ballistic firing tables. EDVAC was designed to do more general calculations than calculating ballistic firing tables. As depicted in Figure 1-1, von Neumann described five subdivisions of the system: central arithmetic and central control (C), main memory (M), input (I), output (O), and recording medium (R). These five components and how they interact is still the standard architecture of most computers today.

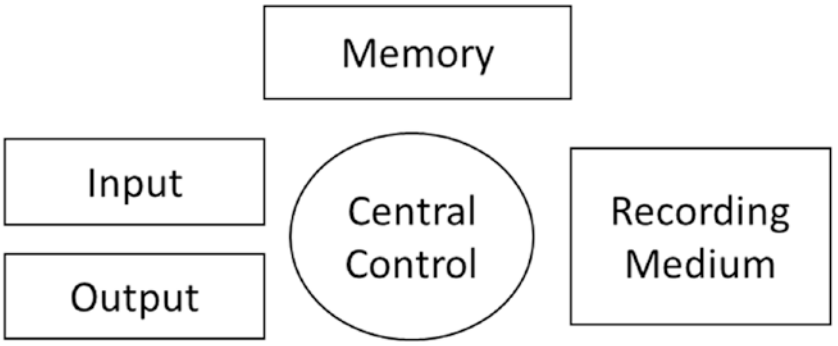

Figure 1-1. *Primary Architecture Elements of a Computer*

In his paper, von Neumann called the central arithmetic and control unit the central control organ and the combination of central control and main memory as corresponding to associative neurons. Even today, people refer to the central processing unit, or CPU, as the "brain" of the computer. Don't be fooled, though, because a computer based on this architecture does exactly what it is programmed to do, nothing more and nothing less. Most often the difficulties we encounter when programming computers are due to the complex nature of how your code depends on code written by other people (e.g., the operating system), combined with your ability to understand the nuances of the programming language you're using. Despite what a lot of people might think, there's no magic to how a computer works, but it can be complicated!

CPU: Fetch, Decode, Execute, and Store

The CPU's job is to fetch, decode, execute, and store the results of instructions. There are many improvements that have been invented to do it as efficiently as possible, but in the end, the CPU repeats this cycle over and over until you tell it to stop or remove power. How this cycle works is important to understand as it will help you debug multi-threaded programs and code for multicore or multiprocessor systems.

Note Threads are a mechanism used to simulate executing a set of instructions in parallel (at the same time), whereas multiple cores in the same system actually do execute instructions in parallel.

The basic blocks of a CPU are shown in Figure 1-2. The CPU needs a clock that sends an electric pulse at a regular interval, called a frequency. The frequency of the clock dictates how fast the CPU can execute its internal logic. The control unit drives the fetch, decode, execute, and store

function of the processor. The arithmetic and logic unit, or ALU, performs math operations and digital logic operations like AND, OR, XOR, and so on. The CPU has an internal memory unit for registers and one or more high-speed memory caches to store data proactively pulled in from main memory.

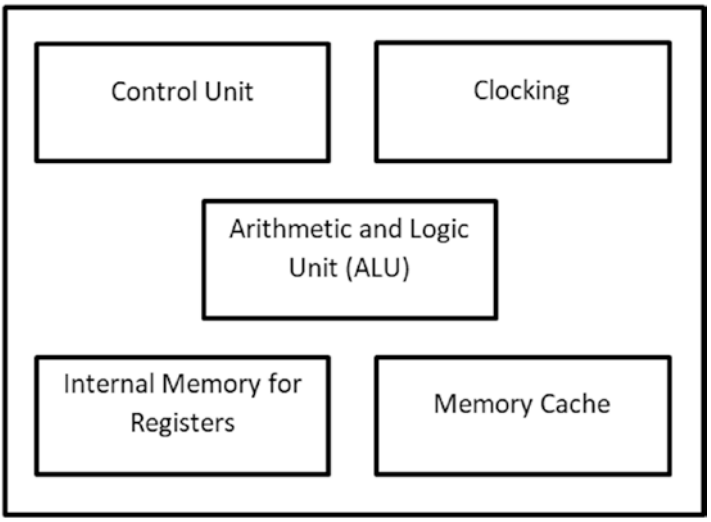

Figure 1-2. *Basic Blocks Inside a CPU*

Fetch

The CPU fetches instructions from memory using addresses. Consider your home's mailbox; it has an address and, if it's anything like my mailbox, contains junk mail and a letter from my mom, if I'm lucky. Like the mail in your mailbox, instructions sit in memory at a specific address. Your mailbox is probably not much bigger than a shoebox, so it has a limit to how much mail the mail carrier can put into it. Computer memory is similar in that each address location has a specific size. This is an important concept to grasp because much of computer programming has

to do with data and instructions stored at an address in memory, the size of the memory location, and so on.

When the CPU turns on, it starts executing instructions from a specific location as specified by the default value of its instruction pointer. The instruction pointer is a special memory location, called a register, that stores the memory address of the next instruction.

Here's a simple example of instructions in memory that add two numbers together:

Address	Instruction	Human-Readable Instruction
200	B80A000000	MOV EAX,10
205	BB0A000000	MOV EBX,10
20A	01D8	ADD EAX,EBX

The first column is the address in memory where the instruction is stored, the second column is the instruction itself, and the third column is the human-readable version of the instruction. The address and instruction numbers are in hexadecimal format. Hexadecimal is a base 16 number system, which means a digit can be 0–F, not just 0–9 as with the decimal system. The address of the first instruction is 200, and the instruction is "mov eax,10," which means "move the number 10 into the EAX register." B8 represents "move something into EAX," and 0A000000 is the value. Hexadecimal digit A is a 10 in decimal, but you might wonder why it's in that particular position.

It turns out that CPUs work with ones and zeros, which we call binary. The number 10 in binary is 1010. B8 is 10111000 in binary, so the instruction B80A000000 in binary would be 1011 1000 0000 1010 0000 0000 0000 0000 0000 0000. Can you imagine having to read binary numbers? Yikes!

In this binary format, a single digit is called a "bit." A group of 8 bits is called a "byte." This means the maximum value of a byte would be 1111 1111, which is 255 in decimal and FF in hexadecimal. A word is 2 bytes, which is 16 bits. In this example, the "MOV EAX" instruction uses a byte for

the instruction and then 4 words for the data. If you do the math, 4 words is 8 bytes, which is 32 bits. But if you are specifying the number 10 (or 0A in hexadecimal) to be moved into the EAX register, why is it 0A000000? Wouldn't that be 167,772,160 in decimal? It would, but it turns out processors don't expect numbers to be stored in memory that way.

```
bit          0 or 1
byte         8 bits
word         2 bytes = 16 bits
dword        2 words = 4 bytes = 32 bits
```

Most CPUs expect the lower byte of the word to be before the upper byte of the word in memory. A human would write the number 10 as a hexadecimal word like this: 000A. The first byte, 00, would be considered the most significant byte; and the second byte, 0A, would be the least significant. The first byte is more significant than the second byte because it's the larger part of the number. For example, in the hexadecimal word 0102, the first byte 01 is the "most significant" byte. In this case, it represents the number 256 (0100 in hexadecimal is 256). The second 02 byte represents the number 2, so the decimal value of the hexadecimal word 0102 is 258. Now, let's look at the "MOV EAX,10" instruction as a stream of bytes in memory:

```
200:  B8    <- Instruction (MOV EAX)
201:  0A    <- Least significant byte of 1st word
202:  00    <- Most significant byte of 1st word
203:  00    <- Least significant byte of 2nd word
204:  00    <- Most significant byte of 2nd word
205:  ??    <- Start of next instruction
```

The instruction is a single byte, and then it expects 4 bytes for the data, or 2 words, also called a "double word" (programmers use DWORD for short). A double word, then, is 32 bits. If you are adding a hexadecimal number that requires 32 bits, like 0D0C0B0A, it will be in this order in

memory: 0A0B0C0D. This is called "little-endian." If the most significant byte is first, it's called "big-endian." Most CPUs use "little-endian," but in some cases data may be written in "big-endian" byte order when sent between devices, for instance, over a network, so it's good to understand the byte order you're dealing with.

For this example, the CPU's instruction pointer starts at address 200. The CPU will fetch the instruction from address 200 and advance the instruction pointer to the location of the next instruction, which in this case is address 205.

The examples we've been studying so far have been using decimal, binary, and hexadecimal number conventions. Sometimes it is hard to tell what type of number is being used. For example, 10 in decimal is 2 in binary and 16 in hexadecimal. We need to use a mechanism so that it is easy to tell which number system is being used. The rest of this book will use the following notation:

Decimal: No modifier. Example: 10

Hexadecimal: Starts with 0x or ends in h. Example: 0x10 or 10h

Binary: Ends in b. Example: 10b

Instruction Set Architecture

Instructions are defined per a specification, called instruction set architecture, or ISA. There are two primary approaches to instruction set architecture that have evolved over time: complex instruction sets and reduced instruction sets. A system built with a complex instruction set is called a complex instruction set computer, abbreviated as CISC. Conversely, a system built with a reduced instruction set is referred to as a reduced instruction set computer, abbreviated as RISC. A reduced instruction set is an optimized set of instructions that the CPU can execute quickly, maybe in a single cycle, and typically involves fewer memory accesses.

Complex instructions will do more work in a single instruction and take as much time to execute as needed. These are used as guiding principles when designing the instruction set, but they also have a profound impact on the microarchitecture of the CPU. Microarchitecture is how the instruction set is implemented. There are multiple microarchitectures that support the same ISA, for example, both Intel and AMD (Advanced Micro Devices) make processors that support the x86 ISA, but they have a different implementation, or microarchitecture. Because they implement the same ISA, the CPU can run the exact same programs as they were compiled and assembled into binary format. If the ISA isn't the same, you have to recompile and assemble your program to use it on a different CPU.

Note A compiler and an assembler are special programs that take code written by humans and convert it into instructions for a CPU that supports a specific instruction set architecture (ISA).

Whether it is complex or reduced, the instruction set will have instructions for doing arithmetic, moving data between memory locations (registers or main memory), controlling the flow of execution, and more. We will use examples based on the x86 ISA to understand how the CPU decodes and executes instructions in the following sections.

Registers

CPUs have special memory locations called registers. Registers are used to store values in the CPU that help it execute instructions without having to refer back to main memory. The CPU will also store results of operations in registers. This enables you to instruct the CPU to do calculations between registers and avoid excess memory accesses. Table 1-1 is the original x86 ISA base register set.

Table 1-1. *x86 Base Register Set*

	64 bits (x86_64)	32 bits (x86)	16 bits(8086) 8 bits	8 bits
Accumulator	RAX	EAX	AX	
			AH	AL
Base register	RBX	EBX	BX	
			BH	BL
Counter	RCX	ECX	CX	
			CH	CL
Data	RDX	EDX	DX	
			DH	DL
Base pointer	RBP	EBP	BP	
				BPL
Source index	RSI	ESI	SI	
				SIL
Destination index	RDI	EDI	DI	
				DIL
Stack pointer	RSP	ESP	SP	
				SPL
General purpose	R8-R15	R8D-R15D	R8W-R15W	
				R8B-R15B

It's important to understand how the registers are used by the CPU for the given ISA. For example, the 32-bit counter, in this case ECX, will be automatically decremented by the loop instruction. Another example is the stack pointer where you can directly manipulate it, but it's modified by many other instructions (we will explore the concept of a stack later in this chapter).

The x86 register set has evolved over time and is meant to be backward compatible with older versions of x86 CPUs. You can see the progression from the original 16-bit processor to 32-bit and the now more common 64-bit memory address sizes. As the memory address size increased, so did the register size, and new names were given to allow using the different register sizes with the appropriate instructions. Even when in 64-bit mode, the 32-bit register names enable programs written for 32 bits to run on 64-bit machines.

A typical ISA will have multiple register sets. For example, x86 has a floating-point register set and another register set for handling large data sets. The popular ARM architecture also has multiple register sets. The register set and the ISA go hand in hand!

Decode, Execute, and Store

Decoding is when the CPU interprets the instruction and transfers the data needed to execute the instruction into the CPU to prepare to execute the instruction.

Instructions are formatted in a particular way to enable efficient decoding. The instruction format specifies the opcode (the operation to be performed), the operands (the registers or data needed for the operation), and the addressing mode. The number and order of the operands depends on the instruction addressing mode as follows:

Register Direct: Both operands are registers:

```
ADD EAX, EAX
```

Register Indirect: Both operands are registers, but one contains the address where the operand is stored in memory:

```
MOV ECX, [EBX]
```