

INSIDE THE MACHINE

UNDERSTANDING THE REPRESENTATIONS,
EXECUTION AND EVOLUTION
THAT SHAPED MODERN COMPUTING



APPLICATIONS

OS SERVICES

SYSTEM CALLS

KERNEL

HARDWARE

PREVIEW

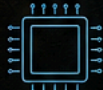
full edition available
for purchase at
www.ardiri.com

PART I FOUNDATIONS



Bits, numbers, memory
and data—how information
is represented.

PART II EXECUTION



The CPU, instructions,
memory hierarchy, buses
and I/O—how the machine
executes work.

PART III EVOLUTION



Parallel machines,
virtual machines, secure
machines, intelligent machines
and the limits of the classical
computer.

FROM BITS TO SYSTEMS. FROM THEORY TO ENGINEERING. FROM IDEAS TO IMPACT.

INSIDE THE MACHINE

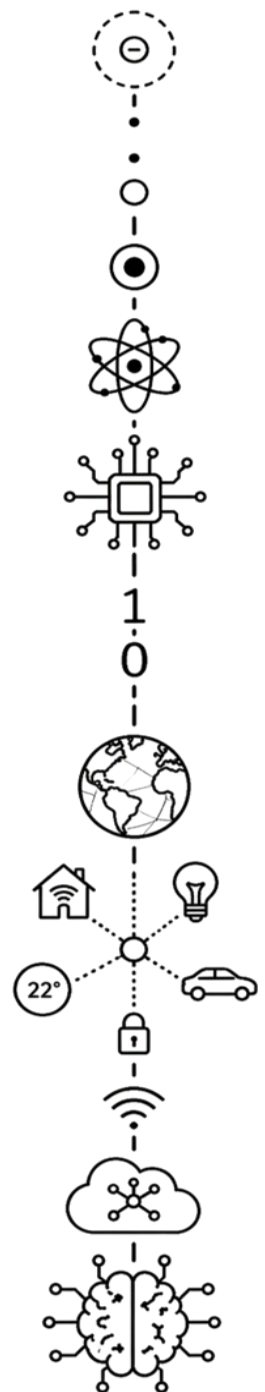
UNDERSTANDING THE REPRESENTATIONS,
EXECUTION AND EVOLUTION THAT SHAPED
MODERN COMPUTING

Aaron Ardiri

© 2026 Aaron Ardiri
All rights reserved.

First Edition - 2026

ISBN 978-91-531-9117-9



Dedicated to **John Fornehed**.

You once described evaluating technology like opening a Russian matryoshka doll—removing one layer only to discover another beneath it.

That simple analogy stayed with me for years and became the inspiration for this book. My goal was never to simply explain how computers work, but to keep peeling back the layers until the history, the reasoning and the engineering beneath them become visible.

FOREWORD

I met Aaron in the late 1990s, we shared a passion for handheld computing.

At the time, he was writing games for the Palm OS platform, and I reached out with a few questions about sprite animation. That conversation marked the beginning of a friendship and a professional relationship that has now spanned almost three decades.

Inspired by the work Aaron was doing, I decided I wanted to write a Palm OS game of my own. He introduced me to the concept of game engines and showed me how it was possible to build surprisingly sophisticated software on hardware with incredibly limited resources. A few months later, my first game, *Hexxagon*, was published under his mobile game studio umbrella.

Over the years that followed, we collaborated on a variety of commercial projects. Aaron focused on mobile software development while I built the desktop companion applications. Looking back, what we achieved with the hardware and tools available at the time often felt like magic.

Although our careers eventually took us in different directions, we remained in contact. Whenever Aaron had a new idea, he would often ask for my opinion—and being Dutch, I was never shy about giving honest and unfiltered feedback.

When Aaron was completing his autobiography we had open conversations, during one of these, he mentioned that his next project would be a book called *Inside the Machine*. I remember telling him that this was the book I was really looking forward to reading. Not long afterwards, he asked whether I would review an early manuscript and provide feedback throughout its development.

I approached the first draft with some hesitation.

A book about computer architecture could easily become dry, overly academic, or inaccessible to anyone without a strong technical background. Instead, I found myself reading late into the evening, eager to continue to the next chapter.

Through carefully chosen analogies, practical examples, historical context, and clear explanations, Aaron has achieved something that is surprisingly rare: he makes complex engineering concepts approachable without oversimplifying them.

What impressed me most was not simply the technical content, but the way it is presented. Rather than treating computer architecture as a collection of isolated topics, the book gradually peels back the layers of abstraction that surround modern computing.

Each chapter builds naturally upon the last, revealing not only *how* computers work, but *why* they were designed that way. By the time I reached the final chapters, I realised I wasn't simply learning about processors, memory, virtualization, or security—I was seeing the entire machine as one coherent system.

More than once while reading, I caught myself thinking, *this is the book I wish I had been given during my computer science studies*. It would have saved me years of gradually piecing these concepts together from different sources.

Whether you are a student beginning your journey, a software developer wanting to understand the hardware beneath your code, an embedded engineer working close to the metal, or simply someone fascinated by how computers really work, I believe you will find something valuable in these pages. This is not a book that expects you to memorize facts or architectures.

Instead, it invites you to think differently about the machines we use every day, revealing the engineering decisions, trade-offs, and ideas that have shaped computing over decades. By the time you reach the final page, I suspect you will look at every computer a little differently.

What once appeared to be a collection of mysterious components will instead become a carefully engineered system whose elegance lies in the way each layer builds upon the last. That, to me, is the greatest strength of this book.

I thoroughly enjoyed reading *Inside the Machine*, and I hope you enjoy the journey just as much. I have no doubt that, by the end, you will have gained not only a deeper understanding of modern computers, but also a greater appreciation for the remarkable engineering that makes them possible..

Bas van de Sande

Coding Architect and Trainer

<https://www.linkedin.com/in/basvandesande/>

July 2026

Table of Contents

About the Author.....	1
Foundations.....	3
Everything Is Just Bits.....	4
Number Systems.....	21
Integers.....	41
Memory and Representation.....	63
Endianness.....	87
Floating-Point Numbers.....	106
Fixed-Point Arithmetic.....	135
Bit Manipulation.....	159
Execution.....	185
The Central Processing Unit.....	186
Machine Instructions.....	212
The Fetch-Decode-Execute Cycle.....	233
Pipelining.....	252
Registers and the Stack.....	270
Memory Hierarchy.....	289
Buses and Communication.....	307
Interrupts and Exceptions.....	327
From Power-On to Running Code.....	348
Persistent Storage.....	367
Evolution.....	385
Parallel Machines.....	386
Virtual Machines.....	405
Secure Machines.....	426
Intelligent Machines.....	445
Beyond the Classical Computer.....	465
Appendixes.....	483
Binary Reference Tables.....	484
IEEE-754 Quick Reference.....	488
CPU Architectures.....	491

About the Author



Aaron Ardiri

<http://www.ardiri.com/>

I am a software engineer, systems architect, entrepreneur and technology leader with more than three decades of experience designing software systems that span embedded devices, mobile computing, cybersecurity, cloud platforms and the Internet of Things.

Over the years, I have mentored developers, spoken at conferences around the world and helped engineers bridge the gap between theory and practice. **Inside the Machine** grew from a simple observation: many programmers learn how to use computers without ever being taught why computers behave the way they do. This book is my attempt to close that gap.

Beginning my programming journey in the late 1980s, I developed a fascination with understanding how computers worked beneath the surface. That curiosity led me from BASIC and assembly language to computer graphics, operating systems, embedded systems, networking, security and distributed computing—an approach that has shaped both my career and the philosophy behind this book.

Throughout my career, I have held technical leadership positions ranging from startup founder to Chief Technology Officer, working with organizations

including BlackBerry, Ericsson and numerous technology ventures across Europe and Australia. I am the founder and CEO of RIoT Secure AB, where I have led the development of secure device lifecycle management platforms for connected embedded systems. My work has also included pioneering mobile software, designing virtual machines, building secure communication protocols and developing technologies that power modern IoT deployments.

Although *Inside the Machine* explores the technical foundations of modern computing, it is ultimately a book about curiosity. It reflects the lessons I have learned throughout a career spent asking one simple question:

"How does it really work?"

That same curiosity inspired my autobiography, *Technology, Entrepreneurship and the Reality of Building Systems*, which tells the personal story behind the engineering journey—from childhood experiments and early programming to founding companies, building products and navigating the realities of entrepreneurship.

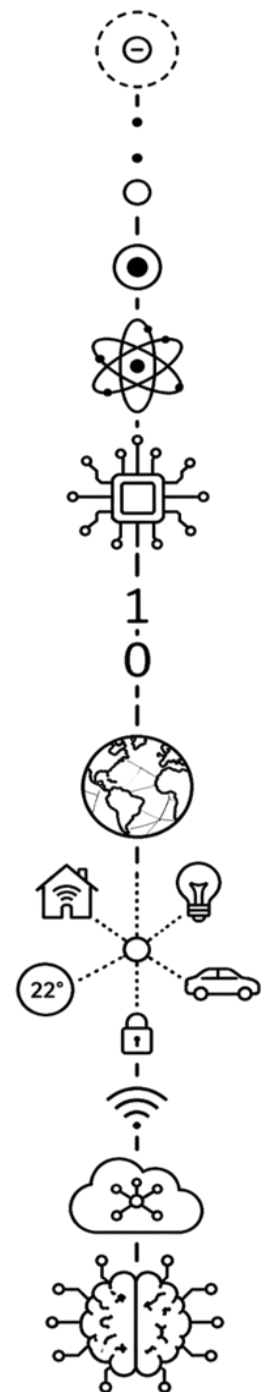
Together, the two books offer complementary perspectives. *Technology, Entrepreneurship and the Reality of Building Systems* tells the personal story behind the engineer, while *Inside the Machine* explores the technical foundations that shaped my understanding of computing. One explains the journey; the other explains the ideas that inspired it.

I continue to design software, explore new technologies and share my passion for engineering through writing, speaking and mentoring. I believe that the most valuable skill an engineer can develop is not mastering a particular programming language or framework, but cultivating the curiosity to understand how things work.

If *Inside the Machine* has encouraged you to look beyond programming languages and software abstractions—to see the representations, trade-offs and engineering decisions that make modern computing possible—then it has achieved exactly what I hoped. The most valuable engineers are rarely those who memorize the most facts, but those who continually ask why systems are designed the way they are. I wish you every success on your own engineering journey—and I hope you never stop asking, "How does it really work?"

ACT 1:

Foundations



Everything Is Just Bits

There is a question that I've asked countless times throughout my career, whether I was writing a Game Boy emulator in assembly language, designing secure communication protocols for IoT devices, building a virtual machine or simply experimenting with a new programming language.

How does it really work?

It is an innocent question, yet one that has guided engineers for generations. Every significant advance in computing has come from someone looking beyond the abstraction in front of them and asking what lies beneath.

Curiosity has always been one of the most valuable tools an engineer can possess. If you stop and look around, you'll probably find yourself surrounded by computers. Your phone, laptop, smartwatch, television, car and even many household appliances now contain processors capable of performing billions of operations every second.

They display photographs, stream music, render three-dimensional graphics, navigate the world using satellites and communicate with devices on the other side of the planet. To us, these all appear to be completely different tasks, each requiring its own specialized technology.

Yet to the computer, none of those distinctions exist.

A photograph of your family, your favorite song, the operating system running your device and the program used to display these words are all stored in exactly the same way. They are not separated into different kinds of memory, nor does the processor recognize one as an image and another as a document. At its lowest level, a computer has no concept of photographs, music, video or text. It only understands bits.

That statement may seem almost disappointingly simple, but it is the single most important idea in computing. Every computer system, from the smallest embedded microcontroller to the largest supercomputer, begins with this remarkably small building block.

Every abstraction we have created over the past seventy years exists to give those bits meaning. Programming languages, operating systems, communication protocols and processor architectures all provide different ways of interpreting and manipulating the same underlying patterns of zeros and ones.

Once you appreciate this, something rather interesting begins to happen. Technologies that once appeared unrelated start to reveal common foundations. A floating-point number and a pixel are no longer fundamentally different. A character in a text file and an instruction executed by the processor become variations of the same theme.

Concepts that once seemed isolated begin to connect, and computing gradually transforms from a collection of independent subjects into a single, coherent discipline. That is the journey I invite you to take. We will begin with the humble bit, exploring how computers represent information before building our understanding one layer at a time.

Along the way we will discover how computers represent numbers, organize memory and manipulate binary information. We will then step inside the processor itself to understand how instructions are executed, before exploring how modern operating systems coordinate the work performed by the machine.

By the end, ideas that once appeared unrelated will reveal themselves as different expressions of the same underlying principles. This is not about memorizing algorithms or programming language syntax.

Technologies change far too quickly for that. Instead, it is about developing an intuition for the machine itself—understanding why things work, recognizing the patterns that appear again and again, and learning to look beyond the abstractions that modern software places in front of us.

By the time you reach the final chapter in the first act, I hope you will no longer see a computer as a mysterious box executing incomprehensible instructions.

Instead, I hope you'll recognize something far more elegant: a remarkably simple machine, built from equally simple ideas, carefully layered together over decades of engineering innovation. And like every journey worth taking, this one begins with a single, deceptively simple question.

What does a computer actually store?

Information Begins as Bits

Imagine you're standing in a dark room with a single light switch.

The switch has only two possible positions: on or off. There is no in-between state, no half-way position and no ambiguity. At any given moment, the light is either illuminated or it isn't.

Now imagine replacing that light switch with a tiny electronic switch, one so small that millions—or even billions—of them can fit onto a piece of silicon no larger than your fingernail.

Individually they appear almost trivial, capable of representing only two distinct states. Collectively, however, they form the foundation of every computer ever built.

This simple idea lies at the heart of digital computing. A bit, short for **binary digit**, is the smallest unit of information a computer can represent. Like the light switch, it has only two possible values, traditionally written as 0 and 1.

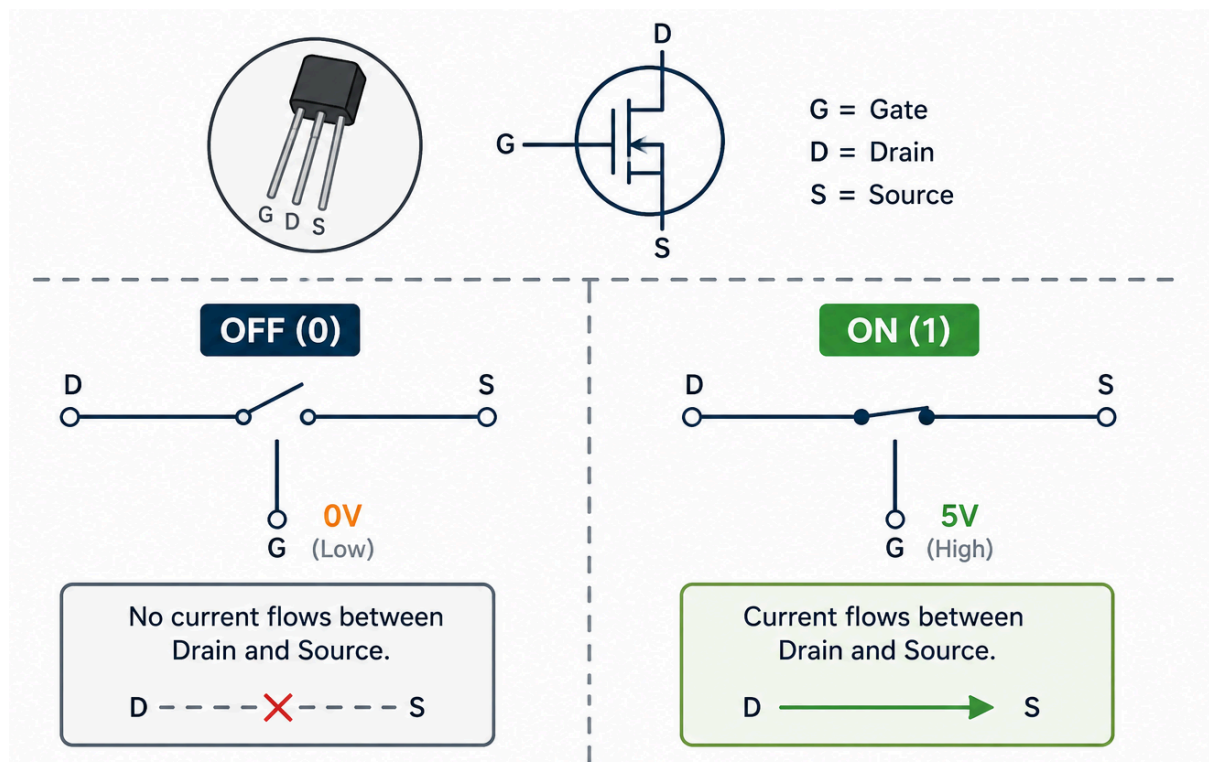


Figure 1.1. A transistor can be viewed as an electronic switch with two stable states. Although modern processors contain billions of transistors, each one ultimately represents a simple binary choice: on or off.

These symbols are not special in themselves; they are simply convenient labels for two distinguishable states. Inside a computer those states might correspond to a low or high voltage, a transistor being switched off or on, a magnetic region pointing in one direction or another, or even the presence or absence of light traveling through an optical fiber.

Physical technology may change, but the underlying idea remains the same.

Why only two states?

At first glance it might seem unnecessarily restrictive.

Wouldn't it be more efficient if a computer stored ten different values instead of just two?

After all, we count using ten digits every day, so surely a decimal computer would be more natural. Early computer pioneers explored exactly this question. Some experimental machines were built using decimal arithmetic, while others experimented with three or more electrical states.

In practice, however, binary proved to be remarkably resilient. Electronic circuits are never perfectly clean. Voltages fluctuate, components age and electrical noise is an unavoidable part of the physical world.

Distinguishing reliably between two well-separated voltage levels is comparatively straightforward; distinguishing between ten different levels is considerably more difficult.

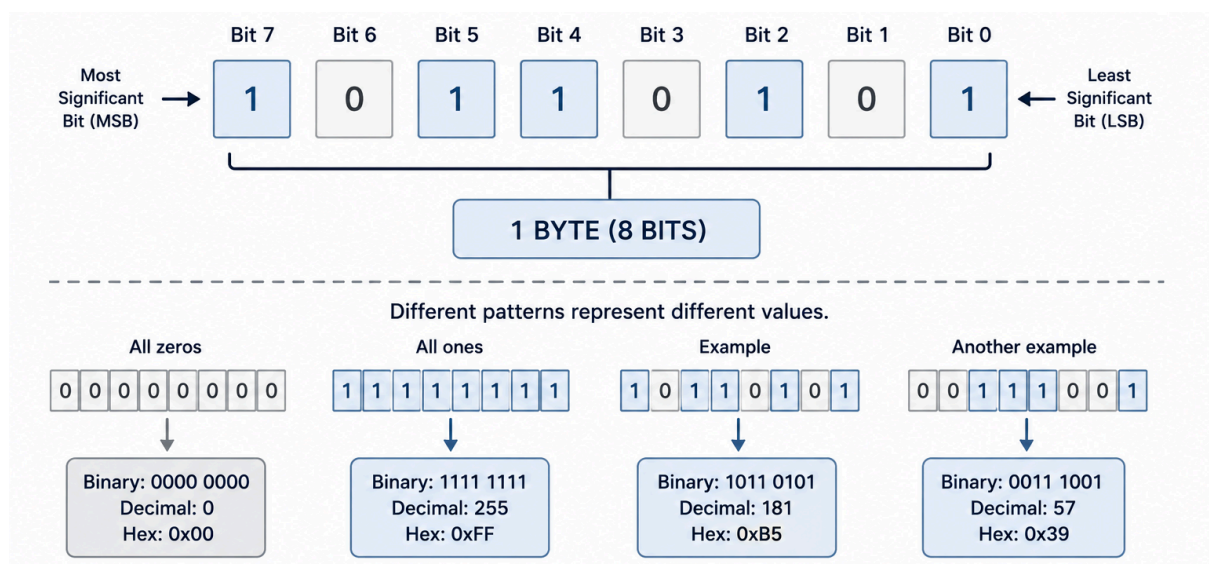


Figure 1.2. A single bit can represent one of two possible states. Individually this is quite limited, but when many bits are combined they can represent increasingly complex information.

By reducing every decision to a simple yes-or-no question, binary systems became faster, more reliable and easier to manufacture.

This simplicity is one of computing's greatest strengths, and it is also worth recognizing that a single bit is not particularly useful on its own.

With only two possible values it can answer a question such as *true* or *false*, *yes* or *no* or *on* or *off*, but it cannot represent the age of a person, the color of a pixel or the letter you're currently reading.

Those richer forms of information emerge only when many bits are combined together, much as individual letters combine to form words and sentences.

For now, it is enough to appreciate that every computer, from the smallest microcontroller embedded inside a household appliance to the largest supercomputer performing scientific simulations, begins with exactly the same fundamental building block.

Beneath the billions of transistors, the sophisticated operating systems and the complex software applications lies an astonishingly simple principle: the ability to represent one of two possible states.

Everything else in modern computing is built upon that simple foundation.

Building Larger Values

A single bit can only represent one of two possible states. While this makes it ideal for answering simple questions such as *yes* or *no*, *true* or *false* or *on* or *off*, it quickly becomes apparent that one bit alone cannot describe very much about the world around us.

Suppose, for example, that you wanted to represent the result of rolling a six-sided die. Two possibilities are no longer sufficient. Likewise, if you wished to store a person's age, the day of the month or the brightness of a single pixel, a lone bit simply doesn't provide enough information.

Number of Bits	Possible Values	Typical Example
1	2	Yes/No
2	4	Four compass directions
3	8	Days of the week (one spare)
4	16	A hexadecimal digit (0-F)
8	256	One byte
16	65,536	A 16-bit integer
32	4,294,967,296	A 32-bit integer
64	18,446,744,073,709,551,616	A 64-bit integer

Table 1.1. *The relationship between the number of bits and the number of unique values they can represent. Every additional bit doubles the available combinations, a deceptively simple idea that forms the foundation of binary representation throughout computing.*

Fortunately, the solution is surprisingly simple. Rather than relying upon a single bit in isolation, computers group many bits together, allowing increasingly complex information to be represented.

Every additional bit doubles the number of possible combinations. Two bits can represent four different values, three bits can represent eight, and four bits can represent sixteen. This pattern continues indefinitely, with each extra bit increasing the amount of information that can be stored.

The progression is remarkable. By the time we reach sixty-four bits, we can represent more than eighteen quintillion unique values—an unimaginably large number built from nothing more than sixty-four binary choices.

This exponential growth is one of the reasons binary has remained the foundation of computing for more than half a century. Simple building blocks, when combined together, become extraordinarily powerful.

It is difficult to overstate the importance of this observation. Complexity is not created by making individual components more sophisticated. More often than not, it emerges simply by combining vast numbers of remarkably simple building blocks.

Number of bits	Bit patterns	Number of possible values
1 bit (2^1)	0 1	2 ($2^1 = 2$)
2 bits (2^2)	00 01 10 11	4 ($2^2 = 4$)
3 bits (2^3)	000 001 010 011 100 101 110 111	8 ($2^3 = 8$)
4 bits (2^4)	0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111	16 ($2^4 = 16$)

Figure 1.3. Each additional bit doubles the number of possible values that can be represented. This exponential growth allows surprisingly large quantities of information to be stored using relatively few bits.

One useful way to think about this is to imagine a series of switches. A single switch has only two possible positions. Add a second switch and there are now four possible combinations. With three switches there are eight combinations, and with four there are sixteen.

Nothing about the individual switches has changed; we have simply increased the number of ways they can be combined. This exponential growth explains why binary is so effective. It may appear limited when viewed one bit at a time, yet surprisingly expressive once many bits are considered together.

As computers evolved, certain group sizes became common enough to acquire their own names. Four bits are known as a **nibble**, capable of representing sixteen unique values. Although the term is encountered less frequently today, it remains useful because a nibble corresponds exactly to a single hexadecimal digit, a numbering system that we will explore later.

Eight bits form a **byte**, perhaps the most familiar unit of storage in computing. A single byte can represent 256 unique values, enough to encode a character, a small integer or one component of a color in a digital image. Modern computer memory is almost universally organized as an array of bytes, making it the fundamental building block of storage and communication.

Larger quantities of information are represented by combining multiple bytes together. Historically these groupings became known as **words**, although their exact size depends upon the architecture of the processor.

Early computers commonly used 16-bit words, while modern desktop and server processors typically operate on 64-bit words. Embedded systems continue to use a wide variety of word sizes depending upon their performance requirements and hardware constraints.

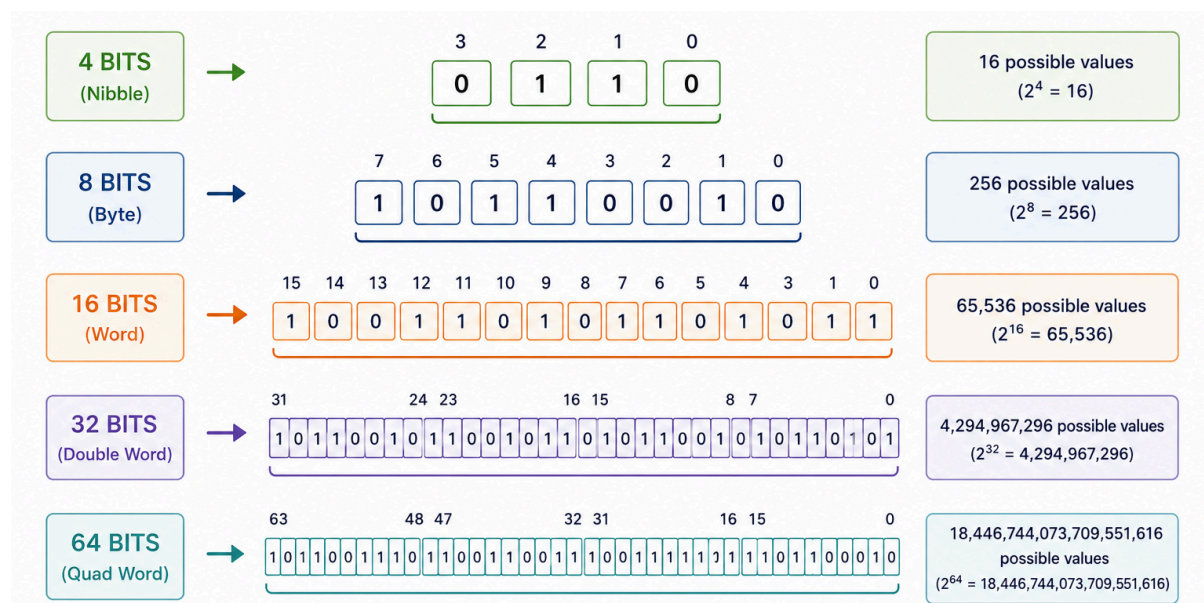


Figure 1.4. Bits are grouped into progressively larger units. Four bits form a nibble, eight bits form a byte, while multiple bytes are combined into words whose size depends upon the processor architecture.

It is important to recognize that these groupings are conventions rather than rules. A byte is simply eight consecutive bits. A word is simply a collection of bytes that happens to match the natural size of a particular processor. Neither has any inherent meaning beyond the way hardware and software choose to interpret them.

This distinction will become increasingly important. As we shall soon discover, a byte containing the value `01000001` might represent the number 65, the letter `A`, one color component within a pixel or even part of a machine instruction. The bits themselves remain unchanged; only their interpretation differs.

Before exploring those interpretations, however, we first need to understand one of the most fundamental ideas in computing: representation.

Meaning Comes from Interpretation

At this point we have discovered that computers store information using bits and that by combining those bits together we can represent an astonishing number of different values.

It would be perfectly reasonable to assume that, once those values have been stored, the computer somehow understands what they represent.

One of the most profound ideas in computing is that bits have no inherent meaning. A computer neither knows nor cares whether a particular sequence of bits represents a number, a letter, a color or part of a computer program.

It simply stores patterns and manipulates them according to instructions. The meaning of the values that the pattern represents exists entirely in the interpretation we choose to apply.

To appreciate this, consider the following sequence of eight bits.

`01000001`

By itself, this pattern tells us absolutely nothing. It is simply a collection of eight binary digits. There is nothing within the sequence that identifies it as a character, an integer or anything else.

Without additional context, it is impossible to determine what they represent.

Suppose, however, that we decide to interpret the pattern as an unsigned binary number. Suddenly those same eight bits become the decimal value 65. If instead we interpret them using the ASCII character encoding, they become the capital letter `A`.

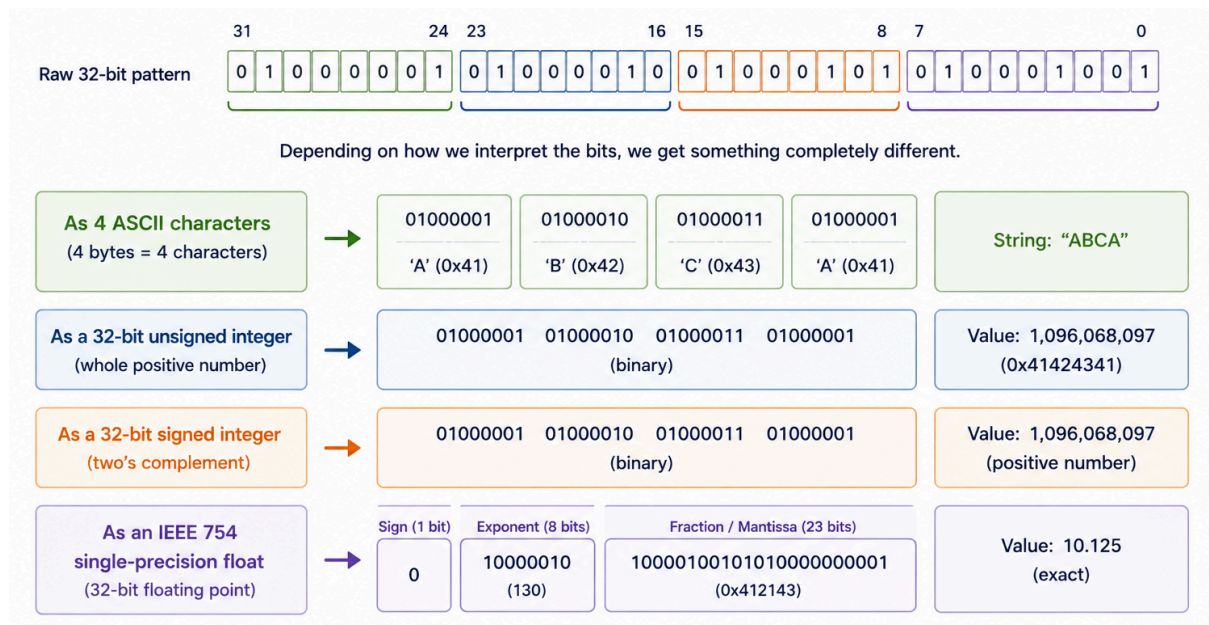


Figure 1.5. The same sequence of bits can represent many different kinds of information. The underlying data never changes; only the interpretation of it.

Within a digital photograph they might describe the intensity of a single pixel, while inside a machine instruction they could form part of an operation the processor is about to execute. They may even represent one small piece of a floating-point number, where their purpose is entirely different once again.

Nothing about the bits has changed, only our interpretation has.

This distinction is easy to overlook because modern programming languages shield us from it almost completely. We declare variables with names such as `age`, `temperature` or `letter`, and the language naturally encourages us to think in terms of those concepts.

The compiler understands the data types we have chosen, generates the appropriate machine instructions and ensures that each value is treated correctly throughout the program.

The processor itself, however, possesses no such understanding. When the program finally executes, the named variables have disappeared.

The compiler has translated them into addresses, registers and machine instructions, leaving the processor to manipulate nothing more than patterns of bits. It performs arithmetic, copies data between memory locations and evaluates conditions without ever knowing whether the values involved represent temperatures, bank balances or the color of a single pixel on the screen.

From Experience

One of the things that fascinated me when I first began writing Game Boy emulators was the realization that the original hardware had no understanding of graphics or sound in the way we naturally think about them. The console wasn't drawing sprites or playing music—it was simply reading and writing binary values according to a precisely defined set of rules.

Once I understood that, emulation became far less mysterious. The challenge was no longer to imitate what appeared on the screen, but to faithfully reproduce how the hardware interpreted and manipulated those underlying bits. The graphics, sound and gameplay emerged naturally from that behavior.

That experience fundamentally changed the way I think about computers. Ever since, whenever I've encountered a new system, I've found myself asking the same question: *What are the bits really representing?*

An analogy from everyday life helps illustrate the point.

Imagine finding the number 42 written on a scrap of paper. Without any additional information, the number is almost meaningless. It might represent a person's age, a page number, the answer to a mathematics problem or the speed limit on a particular road.

The value itself never changes; only the context in which it is interpreted gives it meaning. Computers behave in exactly the same way. A sequence of bits is simply information waiting to be interpreted.

Did You Know? – Definition of Bit

The term **bit** is a contraction of **binary digit**. It was popularized by the American mathematician and information theorist John Tukey in 1947 and later adopted by Claude Shannon in his pioneering work on Information Theory. Today the word bit is universally recognized as the smallest unit of information in computing, although its origins predate the first commercially successful computers.

This simple observation turns out to be one of the most important ideas.

We will encounter integers, floating-point numbers, images, sound, communication protocols, encrypted messages and even complete operating systems.

At first glance these appear to be entirely different technologies, each solving its own unique problem. Yet beneath the abstractions they all share the same foundation. They are simply different ways of interpreting binary data.

Once this idea takes hold, the complexity of computing begins to dissolve.

Technologies that once appeared unrelated suddenly reveal themselves as variations of the same underlying theme, and the computer becomes a far more elegant machine than it first appears.

Memory Doesn't Know

We have seen that the same sequence of bits can represent many different kinds of information. Depending upon how those bits are interpreted, they may become a number, a character, part of an image or even an instruction executed by the processor. This naturally leads to another question.

Where is that information actually stored?

The answer seems obvious, yet the implications are surprisingly subtle—**memory**.

Every computer contains memory, whether it is a tiny microcontroller with only a few kilobytes of RAM or a powerful server equipped with hundreds of gigabytes.

Programs are loaded into memory, variables occupy memory, images are decoded into memory and operating systems constantly move information into and out of memory as applications execute.

It is tempting to imagine that memory somehow understands what it is storing. After all, programming languages encourage us to think in terms of integers, floating-point numbers, strings and objects.

We declare variables, assign them meaningful names and manipulate them using carefully designed language constructs. From the programmer's perspective, memory appears to be filled with rich and varied data types.

The hardware sees something entirely different.

To a memory chip, there is no such thing as an integer. It has no concept of characters, images, colors or network packets. It cannot distinguish between a floating-point number and a machine instruction, nor does it recognize the difference between a variable named `temperature` and one named `counter`.

Memory simply stores bits, nothing more.

Imagine writing two completely unrelated values into memory. One might represent the age of a person, while the other represents the red component of a pixel in a digital photograph.

To us these values are fundamentally different. One describes a property of a human being, the other contributes to an image displayed on a screen. A memory chip has no concept of context. It does not know that one region contains part of a web browser while another contains the text of a book. It has no understanding of integers, floating-point numbers or strings.

Those are ideas that exist entirely within the software we write.

From the perspective of the hardware, memory performs one fundamental task. It stores patterns of bits at particular locations and faithfully returns those same patterns when asked. It neither knows nor cares what those patterns represent.

Perhaps the easiest way to appreciate this is to imagine removing every label from a filing cabinet. The folders remain exactly where they were, each containing its original contents, but without labels it becomes impossible to know which folder contains tax records, family photographs or old letters.

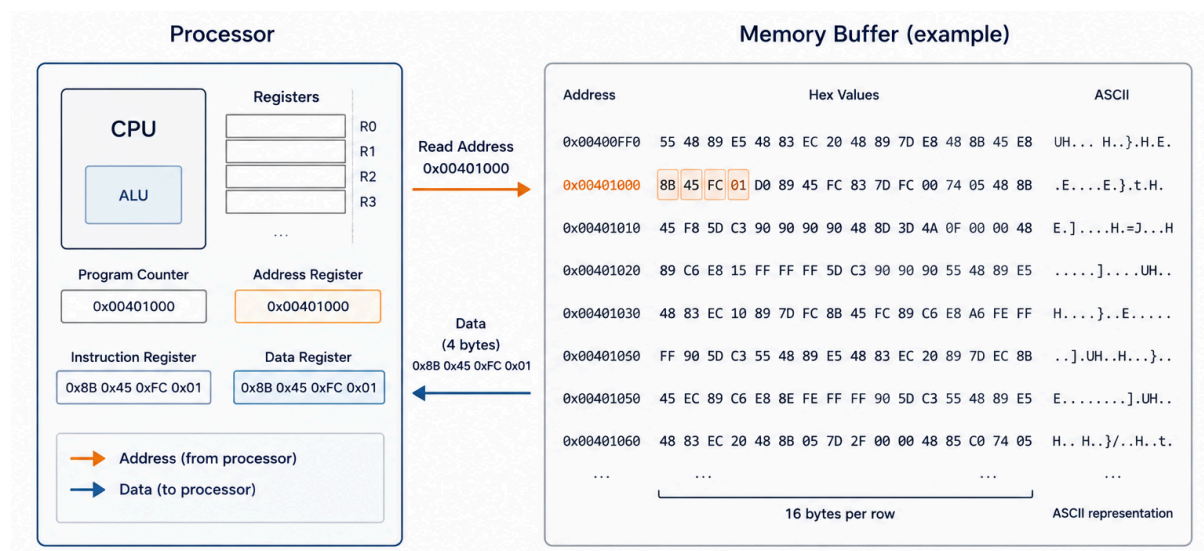


Figure 1.6. Memory stores binary patterns at specific locations. Whether those bits represent an integer, a character, a pixel or part of an executable program depends entirely upon the software interpreting them. A hexadecimal editor reveals the raw contents of memory or a file. Without context, these bytes are simply patterns of data waiting to be interpreted.

The information still exists; only the context has disappeared.

Memory behaves in much the same way. Once the labels provided by our programming language have been stripped away, all that remains are addresses and binary values. The processor reads those values, performs calculations upon them and writes new values back into memory, never pausing to consider what they represent.

One of the first surprises awaiting anyone who opens a debugger or a hexadecimal editor is just how little meaning the raw data appears to contain. Gone are the carefully named variables and neatly organized data structures, replaced instead by long rows of hexadecimal values.

This simple observation has consequences that reach far beyond memory itself. It explains why file formats require specifications, why communication protocols must define exactly how data is encoded and why two programs written by different developers can exchange information successfully only when they agree upon a common representation.

We will encounter countless examples where the same collection of bits acquires entirely different meanings depending upon the rules used to interpret it. Memory, however, remains blissfully unaware of all of them—it simply stores the bits.

Engineering Insight – Systems Programmer Milestone

One of the biggest milestones in becoming a systems programmer is realizing that memory is completely passive. It never stores an integer, a character or an image—it stores only bits. Everything else is an interpretation layered on top by software. Once this idea becomes intuitive, concepts such as pointers, type casting, serialization and communication protocols begin to feel like natural extensions of the same underlying principle.

The Language of the Machine

If memory stores nothing more than patterns of bits, an intriguing question begins to emerge. We have seen how those bits can represent numbers, characters and images, but what about the program itself? Surely the instructions that tell the computer what to do must be something different.

They are not—a computer program is simply another collection of bits.

When we write software, we naturally think in terms of programming languages. We write variables, functions, loops and expressions that are intended to be read by other human beings. The language is designed to help us express ideas clearly, allowing us to concentrate on solving problems rather than worrying about the underlying hardware.

The processor, however, sees none of these human abstractions. It has no idea of for loops, classes or functions. It knows nothing of variables or meaningful names. Like memory, the processor understands only binary patterns, each representing a particular operation that has been defined by the designers of the processor architecture.

Some patterns instruct the processor to add two numbers together. Others tell it to load a value from memory, compare two values or branch to another location in the program. Individually these operations are deceptively simple, often performing just a single task before moving on to the next instruction.

Yet by executing billions of these tiny operations every second, modern processors create the illusion of extraordinarily complex software.

The remarkable thing is that the instructions themselves are stored in memory no differently from any other data. To the hardware, there is no special distinction between a program and the information that program manipulates.

Both are represented using the same binary foundation that we have explored. It is only when the processor fetches those bits and interprets them as instructions that they become executable code.

This idea is both simple and profound.

The same memory that stores a photograph can also store the program used to display it. A document and the word processor used to edit it may sit side by side in memory, each represented by nothing more than carefully organized sequences of bits.

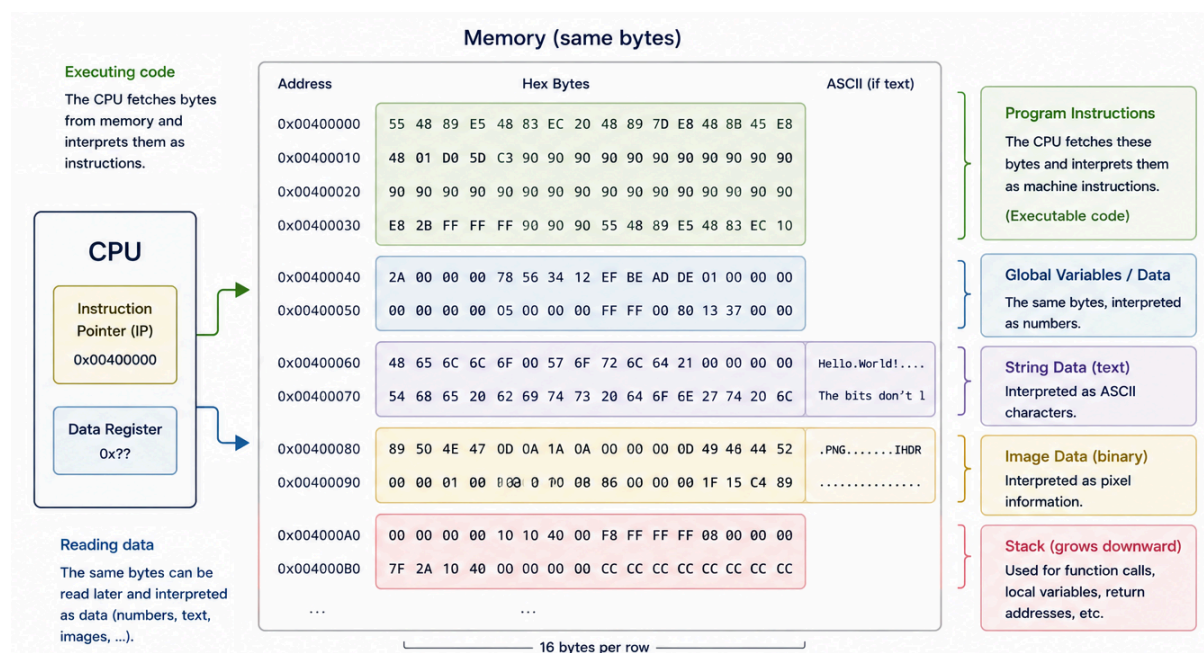


Figure 1.7. Programs and data share the same underlying representation. Both are stored as binary information in memory, differing only in how the processor interprets them. A compiler translates human-readable source code into machine instructions that can be executed directly by the processor.

The distinction lies not in the storage itself, but in the way the processor chooses to interpret those patterns. Writing programs directly as binary numbers would be unimaginably difficult.

Early programmers often worked with machine language or assembly language, carefully constructing instructions that mapped closely to the processor's architecture.

While this provided complete control over the hardware, it also demanded an intimate understanding of the underlying machine. Modern programming languages free us from that burden.

Every processor, regardless of manufacturer or architecture, ultimately performs the same fundamental cycle. It fetches an instruction from memory, decodes what that instruction means and executes it before immediately moving on to the next.

Modern processors perform this cycle billions of times each second, yet the underlying principle has remained remarkably unchanged since the earliest electronic computers.

A compiler translates the high-level code we write into the machine instructions that a processor can execute, preserving the intent of the program while hiding much of the complexity beneath.

Although the tools have evolved dramatically over the past seventy years, the underlying principle has remained unchanged. Every program, regardless of the language in which it was written, ultimately becomes a sequence of binary instructions executed one after another by the processor.

Historical Note – The Stored Program Concept

One of the most important ideas in the history of computing is the stored-program concept. Before its introduction, many early computers were programmed by physically rewiring plug boards or manually configuring switches to change their behavior. Altering a program often meant altering the hardware itself.

During the 1940s, mathematician John von Neumann popularized the idea that both programs and data could reside together in the same memory. Instead of rewiring the machine, a computer could simply load a different sequence of instructions into memory and execute them.

This seemingly simple insight transformed computing forever. It made software practical, enabled operating systems, compilers and programming languages to flourish, and established an architectural model that remains the foundation of almost every general-purpose computer built today. Although modern processors have evolved far beyond those early machines, the stored-program concept continues to underpin contemporary computing.

It is perhaps one of the most elegant ideas in computing.

Programs are not separate from data; they are simply another form of data. Once this is understood, the boundary between software and information begins to disappear, revealing a computer for what it truly is: a machine that stores, interprets and transforms patterns of bits according to carefully defined rules.

Everything Starts Here

At first glance, a single bit hardly seems worthy of an entire chapter. It can represent only one of two possible states, stores very little information by itself and appears insignificant when compared to the complexity of modern computers.

Yet we have discovered that this humble building block is far more important than it first appears. Every piece of information stored by a computer begins here, with a single bit that, by itself, appears almost insignificant.

Whether it is a number, a character, a photograph, a piece of music or an executable program, everything ultimately becomes a carefully organized sequence of bits.

Memory stores those bits without understanding them, processors manipulate them without attaching meaning to them, and software provides the rules that allow those patterns to represent the rich digital world we interact with every day.

It is an elegant foundation—rather than inventing different mechanisms for every kind of information, computers rely upon a single, universal representation from which everything else emerges.

This simple idea has remained largely unchanged since the earliest days of digital computing, surviving generations of processors, programming languages and operating systems.

If there is one lesson to take away, it is not simply that computers use binary. It is that *representation matters*. Once information has been reduced to bits, everything that follows depends upon how those bits are organized, interpreted and manipulated.

Change the representation, and you change the behavior of the system.

That single idea will quietly accompany us. We will discover how computers count using binary, why hexadecimal became the language of engineers and how negative numbers can be represented without requiring a separate sign.

We will explore the structure of memory, learn why the order of bytes matters, and uncover the clever techniques used to represent fractions and many other forms of information.

Our journey will continue by exploring how computers represent numbers, organize memory and manipulate information. From there we will step inside the processor itself to discover how instructions are executed, how memory is accessed and how operating systems coordinate the countless activities taking place inside every modern computer.

By the end of this book, we will have traveled from the smallest unit of information to the engineering principles that allow computers to represent information, execute instructions and coordinate complex systems. Along the way, you will discover that topics which initially appeared unrelated are, in fact, different expressions of the same underlying principles.

Everything begins with bits. Understanding how those bits are represented, transformed and coordinated is what allows us to truly look inside the machine.

Number Systems

The Same Number, Different Languages

If you spend any amount of time reading source code, browsing technical documentation or examining the output of a debugger, you will almost certainly encounter numbers that seem strangely unfamiliar.

Alongside the decimal values we use every day, you may come across expressions such as `0xFF`, `0b10101010` or perhaps an unfamiliar sequence of digits that appears to bear little resemblance to anything you've seen before. These values can seem almost cryptic.

It is easy to assume that computers somehow understand several different kinds of numbers, switching effortlessly between one representation and another depending upon the situation.

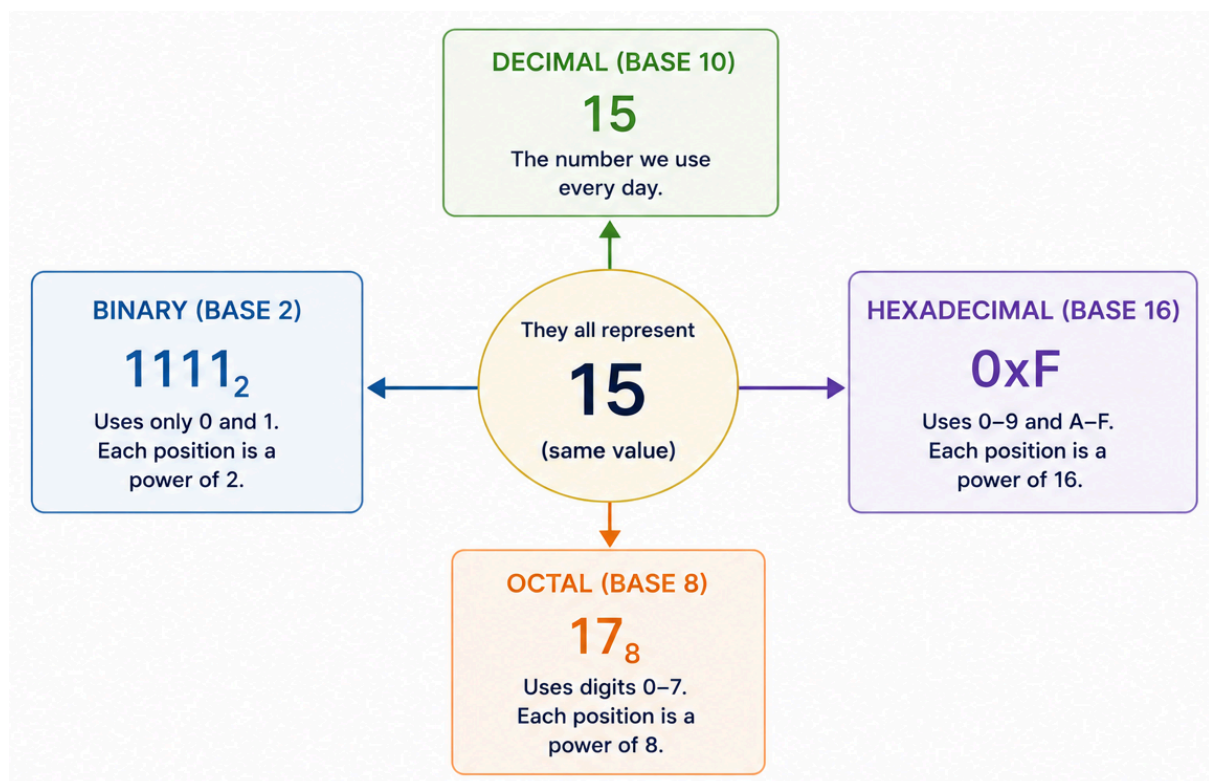


Figure 2.1. The same quantity can be written using different numbering systems. The value remains unchanged even though its representation appears very different.

Consider the following four values.

15 1111₂ 0xF 17₈

At first they appear to describe four completely different numbers. In fact, they all represent exactly the same quantity. Nothing about the value has changed. Only the notation used to write it is different.

This distinction is one of the first conceptual hurdles when learning how computers represent information. We naturally associate a number with the symbols used to write it, rarely considering that those symbols are merely one representation of the underlying value.

The number fifteen exists independently of whether we write it as 15, 1111₂, 0xF or 17₈. Each notation simply follows a different set of rules for expressing the same underlying quantity.

Language provides a useful analogy. The English word *three*, the French word *trois* and the German word *drei* all describe exactly the same concept. Although the words themselves look and sound completely different, the quantity they represent never changes.

Number systems behave in much the same way. They are simply different representations for expressing the same numerical value.

Once this idea becomes intuitive, many of the mysteries surrounding low-level programming begin to disappear. Binary is no longer a strange language invented for computers, and hexadecimal is no longer an obscure notation reserved for experienced programmers.

They are simply alternative ways of writing numbers, each chosen because it solves a particular engineering problem more effectively than decimal. That raises an obvious question.

If the decimal system serves us so well in everyday life, why do computers rely upon binary at all?

It has very little to do with mathematics and everything to do with engineering.

Why Do We Count in Tens?

It is difficult to imagine counting in anything other than tens.

From our earliest years we learn that after nine comes ten, after ninety-nine comes one hundred, and after nine hundred and ninety-nine comes one thousand. The decimal system feels so natural that most of us never stop to question it. It simply appears to be the way numbers work.

Yet there is nothing inherently inevitable about it.

The universe does not favor ten over any other number, nor is there a mathematical law that declares decimal to be the "correct" way to count. In fact, the numbering system we use every day is largely the result of human anatomy rather than mathematical necessity.

Most people have ten fingers.

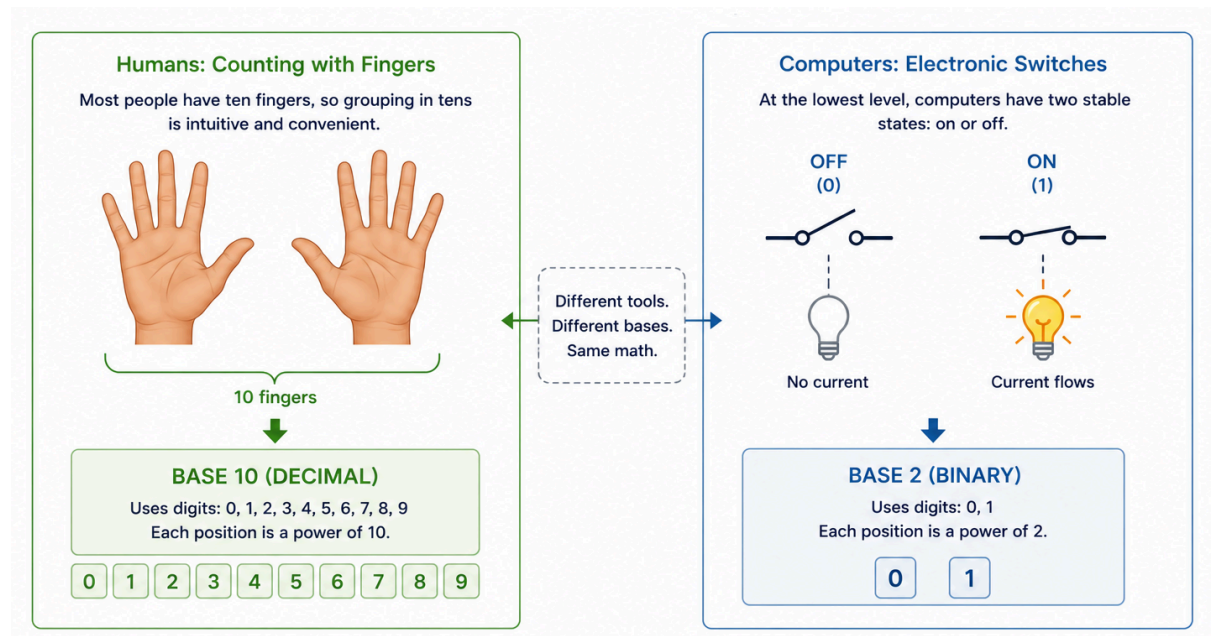


Figure 2.2. The decimal system is believed to have evolved largely because humans naturally count using their ten fingers. Other civilizations adopted different numbering systems based upon their own traditions and needs.

Long before calculators, computers or even written mathematics, our ancestors counted using the tools that were always available to them: their hands. Fingers provided a convenient way to keep track of quantities, making groups of ten a natural choice for recording and communicating numbers.

Over time this approach became deeply embedded in language, trade and education, eventually evolving into the decimal system that much of the world continues to use today.

Of course, not every civilization reached the same conclusion. Throughout history, different cultures experimented with a variety of numbering systems, each reflecting their own practical needs and traditions.

The ancient Babylonians famously developed a system based on sixty, traces of which still survive whenever we measure time in seconds and minutes or divide a circle into three hundred and sixty degrees.

The Maya independently developed a numbering system based largely on twenty, thought to have arisen from counting both fingers and toes.

The important observation is not that these systems were different, but that they all worked. Whether a society counted in tens, twenties or sixties, the underlying mathematics remained exactly the same.

Addition, subtraction, multiplication and division did not suddenly change because the symbols or groupings were different. The rules for writing numbers had changed, but the quantities those numbers represented had not.

This idea may seem almost trivial at first, yet it marks an important shift in perspective. We often think of decimal notation as being synonymous with numbers themselves, when in reality it is simply only one way of writing them.

Just as spoken languages developed differently around the world while describing the same physical reality, numbering systems evolved independently while representing the same underlying quantities.

Once you begin to separate the **value** from the **notation**, everything starts to fall naturally into place. Decimal is not special because it is mathematically superior. It is the notation that humans became accustomed to using. Computers, however, were designed under very different constraints.

Unlike people, they do not have fingers.

Nor do they need them.

Positional Notation

Whether we write numbers in decimal, binary or hexadecimal, they all share one remarkably elegant idea. Rather than assigning a unique symbol to every possible quantity, each digit derives its meaning from two things: the symbol itself and the position in which it appears.

This is known as **positional notation**, and it is one of the most important ideas in mathematics.

To appreciate just how powerful it is, consider the decimal number 4,382. Initially it appears to be nothing more than four digits written side by side. Yet each digit contributes a very different amount to the final value. The 2 represents two ones, the 8 represents eight tens, the 3 represents three hundreds, and the 4 represents four thousands.

The value of each digit depends entirely upon where it is placed.



Figure 2.3. In positional notation, each digit represents a multiple of the base raised to an increasing power. In decimal, the place values are powers of ten.

This relationship can also be expressed mathematically.

$$\begin{aligned}
 &4,382 \\
 &= (4 \times 10^3) + (3 \times 10^2) + (8 \times 10^1) + (2 \times 10^0) \\
 &= 4,000 + 300 + 80 + 2
 \end{aligned}$$

Although this may appear to be a mathematical exercise, it reveals something much more fundamental. The number 4,382 is not a single symbol with a predefined meaning. It is simply the sum of four separate values, each determined by multiplying a digit by the appropriate power of ten.

Once you recognize this pattern, something rather interesting happens.

There is nothing inherently special about the number ten. The entire system works perfectly well if we replace every power of ten with powers of another number. The symbols may change, the place values may change, but the underlying principle remains exactly the same.

Whether we choose two, eight, ten, sixteen or even sixty as our base, positional notation continues to describe numbers in precisely the same way.

This observation marks an important turning point in our journey. Until now, decimal has probably felt synonymous with numbers themselves. In reality, decimal is merely one particular implementation of a much broader idea. Positional notation provides the framework; the choice of base simply determines how quickly each column grows in value.

That realization removes much of the mystery surrounding other numbering systems. Binary is no longer an entirely different way of thinking about numbers. It is simply positional notation with a different set of place values. In other words, once you understand how decimal works, you have already learned the most important part of binary.

Engineering Insight – Engineer Habits

One of the most useful habits an engineer can develop is to look beyond the notation and search for the underlying pattern. Decimal, binary, octal and hexadecimal often appear to be completely different systems, yet they are all built upon exactly the same principle. Once you recognize that pattern, learning a new number system becomes far less about memorization and far more about understanding.

Binary

Armed with an understanding of positional notation, we are finally ready to explore the numbering system that lies at the heart of every digital computer.

At first the sequences may appear unfamiliar, but the underlying idea is one we have already encountered. Nothing fundamental is about to change.

Imagine, for a moment, a world in which you possessed only two digits instead of ten. Rather than writing numbers using the symbols 0 through 9, you are limited to just 0 and 1.

At first this seems restrictive—how could you possibly count beyond one?

The answer is exactly the same way we count in decimal. When we reach the largest available digit, we simply carry into the next position. In decimal, the sequence progresses naturally from 8 to 9, and then to 10 because we have exhausted all ten available digits.

Binary behaves no differently. After 0 comes 1, and having exhausted the only remaining digit, the next value becomes 10. The timing of the carry changes, but the underlying process is identical.

As we continue counting, a familiar pattern quickly begins to emerge. The numbers appear to grow rather quickly. This is hardly surprising. With only two symbols available, additional digits must be introduced far more frequently than they are in decimal.

Yet despite their appearance, these values represent exactly the same quantities we have always used. Nothing about the underlying mathematics has changed; only the notation has.

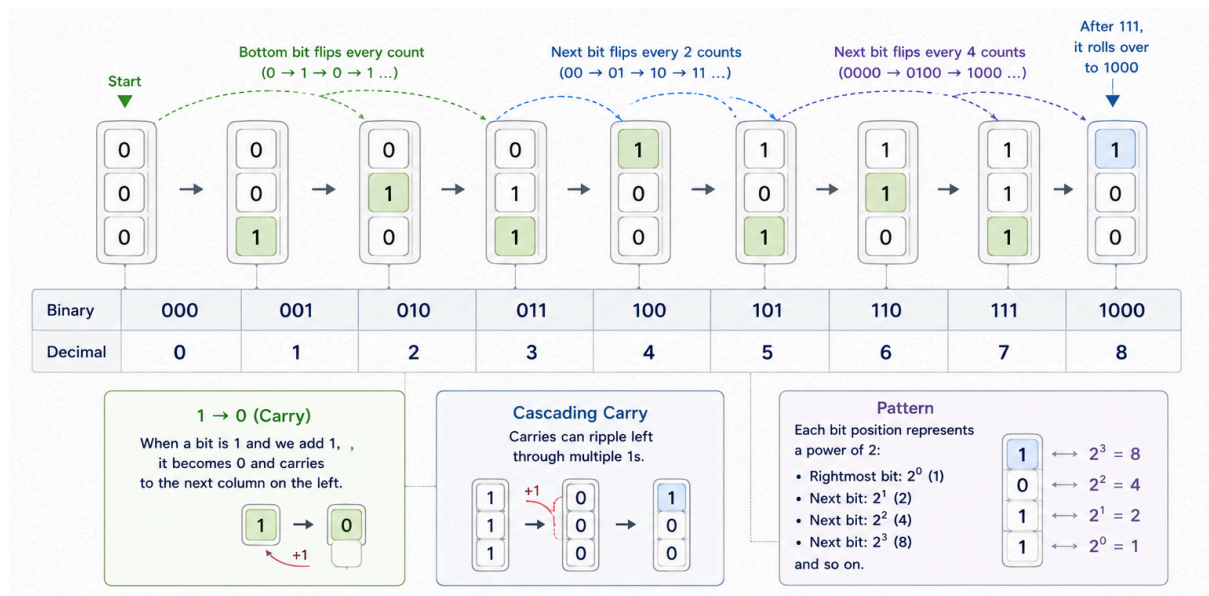


Figure 2.4. Counting in binary follows exactly the same rules as decimal. Once the highest available digit has been used, the value carries into the next position.

The reason becomes clear once we examine the place values. In decimal, each column represents an increasing power of ten. The rightmost digit counts ones, the next counts tens, followed by hundreds and thousands. Binary follows precisely the same pattern, except that each position represents an increasing power of two.

Consider the binary value 101101_2 . Using the same approach we applied to decimal, we can expand it according to its place values.

$$\begin{aligned}
 &101101_2 \\
 &= (1 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + \\
 &\quad (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) \\
 &= 32 + 8 + 4 + 1 \\
 &= 45
 \end{aligned}$$

There is nothing mysterious about this calculation. We have simply replaced powers of ten with powers of two. Every digit contributes to the final value according to its position, exactly as it did in decimal.

Once this becomes familiar, reading binary numbers becomes no more difficult than reading decimal ones.

Of course, familiarity takes time. Few people grow up reading binary in everyday life, so it initially appears awkward and cumbersome. Experienced programmers rarely attempt to interpret long binary values digit by digit.

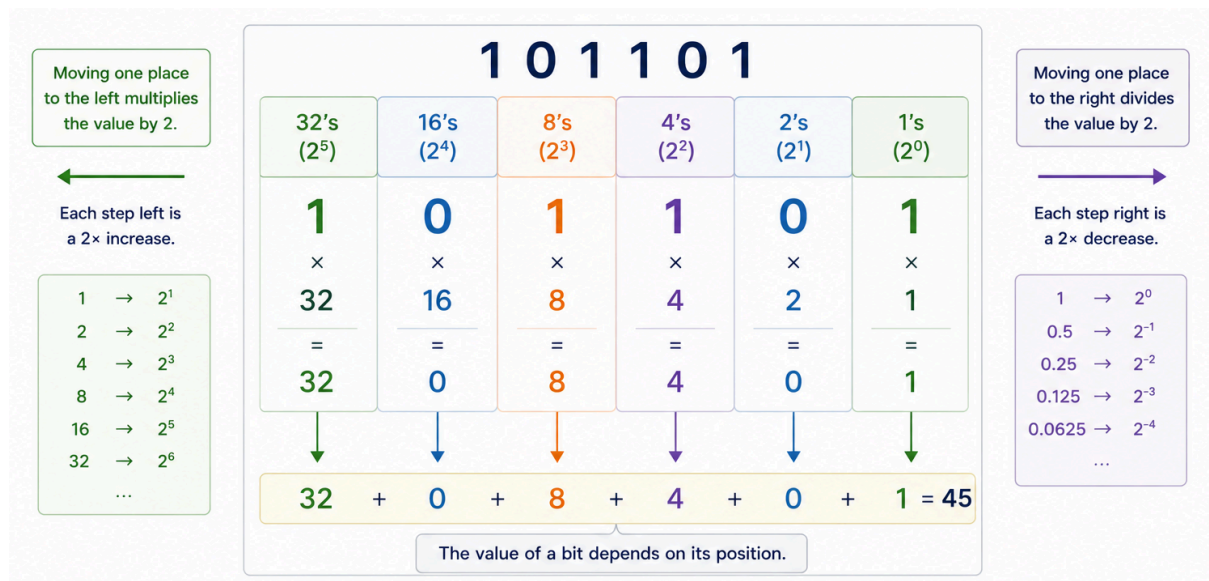


Figure 2.5. Binary uses powers of two rather than powers of ten. The underlying principle of positional notation remains unchanged.

Instead, they begin to recognize common patterns almost instinctively. Values such as 11111111 or 10000000 soon become as recognizable as familiar decimal numbers. That observation hints at another important idea.

Although computers work naturally with binary, humans do not.

As binary values become larger, they also become increasingly difficult to read, compare and remember. A thirty-two-bit memory address or machine instruction may consist of dozens of binary digits, making it easy to lose your place or misread a single bit.

If binary is so well suited to computers, why do programmers so often write values using an entirely different notation?

It lies not in the needs of the computer, but in the needs of the programmer.

Why Binary?

By this point, binary no longer seems quite so mysterious. We have seen that it follows exactly the same rules as decimal, differing only in the number of symbols available and the place values assigned to each position. From a mathematical perspective, there is nothing inherently superior about either system.

Which naturally raises an important question.

If decimal works so well, why do computers insist on using binary?

It is tempting to assume that the answer lies somewhere within mathematics. After all, computers perform calculations at extraordinary speed, so perhaps binary offers some hidden mathematical advantage over decimal.

In reality, the decision has remarkably little to do with mathematics—it is almost entirely an engineering problem. To understand why, imagine trying to build a machine capable of recognizing ten different electrical voltage levels.

One voltage might represent the digit 0, another the digit 1, continuing all the way to 9. At first this seems perfectly reasonable. If humans can count using ten symbols, why shouldn't a computer?

The difficulty is that the physical world is rarely perfect.

Electrical signals fluctuate, components age, temperatures change, and tiny imperfections creep into every circuit. A voltage intended to represent one value may drift slightly higher or lower, making it increasingly difficult to distinguish between neighboring levels.

As more states are introduced, the margin for error becomes smaller.

Now imagine a much simpler system. Instead of trying to distinguish between ten different voltages, the circuit needs to recognize only two. A signal is either sufficiently low to represent 0, or sufficiently high to represent 1.

Small variations in voltage no longer matter very much because the gap between the two states is so large. As long as the signal remains on the correct side of the threshold, the value is interpreted correctly. This tolerance to electrical noise is one of the key reasons binary systems are so robust and reliable in real hardware.

This simple observation changed the course of computing. Rather than attempting to build increasingly complicated hardware capable of distinguishing many different states, engineers embraced simplicity.

By reducing every decision to a choice between two stable conditions, digital circuits became faster, more reliable and significantly easier to manufacture.

At the heart of this approach lies one of the most important inventions of the twentieth century: the **transistor**. Although modern processors contain billions of these microscopic devices, each performs an astonishingly simple task. It behaves much like an electronic switch, controlling whether electrical current is allowed to flow or prevented from flowing.

Those two stable conditions naturally correspond to the binary values 0 and 1, providing the physical foundation upon which every digital computer is built.

The elegance of binary is therefore not that it makes arithmetic easier. Its true strength is that it matches the physical realities of electronic hardware. Mathematics adapted to engineering, rather than the other way around.

This distinction is worth remembering because it appears repeatedly throughout computing. Many of the technologies we use every day were not chosen because they are mathematically perfect, but because they represent the best compromise between theory and the practical constraints of the real world. Binary is one of the earliest—and perhaps the most enduring—examples of that philosophy.

It is also the reason why we began with a discussion of bits rather than numbers. A bit is not an abstract mathematical invention. It is the natural consequence of building reliable electronic circuits from components that can exist in one of two stable states.

Once that connection becomes clear, the relationship between hardware and software begins to feel much less mysterious. Binary is not the language of computers because computers somehow "prefer" it. Binary is the language of computers because it is the language that the hardware can speak with extraordinary reliability.

Historical Note – Why Not Decimal Computers

During the early years of computing, engineers experimented with machines that represented numbers in a variety of ways. Some mechanical calculators used decimal wheels, and a handful of early electronic computers—including **IBMs 650**—performed calculations internally using decimal arithmetic rather than binary.

Although these systems worked, they required significantly more complex hardware than equivalent binary designs. As transistor technology matured, binary architectures proved simpler, faster, cheaper and more reliable. By the late twentieth century, binary had become the universal foundation of digital computing, not because it was the only possible choice, but because it was the most practical one.

When Binary Stops Being Friendly

By now, binary should feel considerably less intimidating than it did originally.

We have seen that it is neither mysterious nor mathematically unusual. It follows exactly the same rules as decimal, differing only in the number of symbols available and the place values assigned to each position.

For the computer, this simplicity is ideal. For people, however, it presents an entirely different challenge. Imagine being asked to identify the following value while stepping through a program in a debugger.

11111111101001011001100110101110

To a processor, this sequence is perfectly ordinary. It is simply thirty-two bits, each occupying its own position and contributing to the final value. The hardware has no difficulty interpreting it because it processes one bit at a time, millions or even billions of times every second.

Most humans are considerably less efficient. Our eyes naturally search for patterns. We instinctively group information into manageable chunks, recognizing familiar shapes rather than analyzing every individual symbol. Reading a long binary value, however, demands concentration.

It is surprisingly easy to lose your place, miscount a digit or overlook a single bit that completely changes the value being represented. The problem is not that binary is difficult—the problem is that it was never designed for people.

This distinction highlights an important engineering principle that extends far beyond number systems. A solution that is ideal for one part of a system is not necessarily ideal for every part of it.

Binary allows electronic hardware to operate with extraordinary reliability, yet the same notation becomes increasingly awkward as soon as a programmer needs to read or write it.

Consider a typical memory address on a modern computer. Written in binary, it may occupy thirty-two or sixty-four individual digits. Even relatively small values quickly become cumbersome, making debugging, documentation and communication unnecessarily difficult.

Although computers have no trouble processing such values, engineers need representations that allow them to recognize patterns at a glance. This need for readability appears throughout low-level computing.

Engineers rarely work directly with long strings of binary digits. Instead, they rely upon representations that make information easier to inspect. Hexadecimal, grouped binary, register layouts, memory dumps and processor documentation all present exactly the same underlying information in forms better suited to human understanding. The information itself never changes—only its representation does.

Numbers are no different.

The challenge, then, is not to replace binary but to find a notation that preserves its simplicity while making it significantly easier for people to read. As it turns out, the solution is remarkably elegant.

From Experience

One of the habits I developed while working with embedded systems was to stop thinking of binary as something I should read directly. Whenever I encountered a long sequence of bits in a debugger or data sheet, my instinct was to immediately regroup it into something more recognizable.

That simple change dramatically reduced mistakes. Instead of counting individual zeros and ones, I could focus on the patterns they represented. Over time those patterns became familiar enough that values such as `0xFF`, `0x8000` or `0xDEADBEEF` conveyed meaning almost instantly, whereas their binary equivalents would still require careful inspection.

Looking back, I wasn't becoming better at reading binary. I was learning to choose the right representation for the task at hand.

Hexadecimal Notation

Fortunately, engineers discovered that while long sequences of ones and zeros are difficult for people to read, they can be divided into small groups that are surprisingly easy to recognize.

The magic number happens to be four. Consider the following binary value.

```
1101011010111111
```

Viewed as a single uninterrupted sequence, it is not particularly inviting. Our eyes struggle to identify where one group ends and the next begins, making even relatively small values difficult to compare or remember.

Now separate the same value into groups of four bits.

```
1101 0110 1011 1111
```

Nothing about the value has changed—only the way it is presented.

Suddenly the individual patterns become much easier to distinguish. Instead of sixteen unrelated symbols competing for our attention, we now have a handful of much smaller groups that can be processed almost instinctively.

This observation may seem almost insignificant, yet it leads to one of the most elegant conventions in computer science. There are exactly sixteen possible combinations of four bits.

```
0000 - 0001 - 0010 - 0011 ... 1111
```

Perhaps not coincidentally—sixteen is also small enough that each combination can be represented by a single symbol. The first ten combinations naturally map onto the digits we already know.

0 1 2 3 4 5 6 7 8 9

That leaves six additional values still to represent. Rather than inventing entirely new symbols, engineers simply continued with the first six letters of the alphabet.

A B C D E F

The result is what we now call **hexadecimal**.

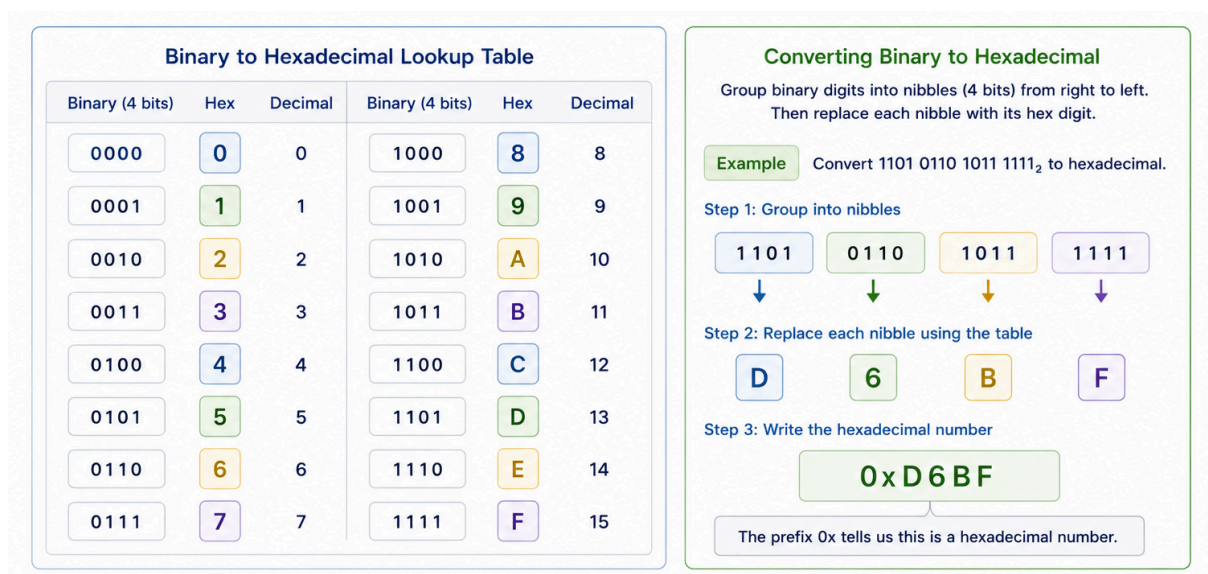


Figure 2.6. Every hexadecimal digit represents exactly four bits. This one-to-one relationship makes hexadecimal a compact and human-friendly representation of binary.

Once this relationship is understood, converting between binary and hexadecimal becomes almost effortless. Each group of four bits is translated independently, without affecting any of the others. For example,

1101 0110 1011 1111 → D 6 B F → 0xD6BF

The reverse process is just as straightforward. Each hexadecimal digit immediately expands back into its corresponding four-bit pattern. Experienced programmers often appear to "read" hexadecimal with ease.

They are not mentally converting every value into decimal before understanding it. Instead, they recognize each hexadecimal digit as a compact representation of four familiar bits. With practice, hexadecimal becomes almost an extension of binary itself.

Did You Know? – Why A Through F?

When hexadecimal notation was first adopted, engineers needed six additional symbols beyond the familiar digits 0 through 9. Rather than inventing entirely new characters, they simply continued with the letters A through F, representing the values ten through fifteen.

Although this convention may seem arbitrary at first, it quickly became universal. Today it appears in virtually every programming language, debugger, processor architecture and hardware datasheet, making hexadecimal one of the few notations recognized by engineers across every area of computing.

It is worth emphasizing that computers never perform calculations in hexadecimal. This is one of the most common misconceptions among new programmers. Processors store binary values in memory, perform binary arithmetic and manipulate binary instructions.

Hexadecimal exists for one reason only: it allows humans to work with binary information far more comfortably. It is a notation designed entirely for our benefit. Once you appreciate this distinction, hexadecimal begins to appear everywhere. Memory addresses are almost always written in hexadecimal.

Machine instructions, bit masks, color values, network protocols and binary file formats all rely heavily upon it. Whenever engineers need to inspect raw binary data, hexadecimal provides a representation that is both compact and remarkably easy to interpret.

It is difficult to overstate how useful this becomes in practice. Although decimal remains the language of everyday life, hexadecimal gradually becomes the language of low-level software development.

Engineering Insight – Hexadecimal

Hexadecimal is often described as a numbering system, but experienced engineers rarely think of it that way. Instead, they think of it as a **window into binary**. Every hexadecimal digit corresponds to exactly four bits, making it possible to inspect memory, registers and machine instructions without mentally processing long strings of zeros and ones.

Once this relationship becomes intuitive, hexadecimal stops feeling like another notation to memorize and instead becomes a remarkably efficient way of reading the machine's native language.

Given enough experience, values such as `0xFF`, `0x1000` or `0xDEADBEEF` begin to feel every bit as familiar as the decimal numbers we use outside computing.

The most surprising realization, however, is that we have not invented a new kind of number at all—we have simply found a more convenient representation of binary information.

Octal Notation

Having discovered the elegance of hexadecimal, it is natural to wonder whether other shorthand notations exist for binary. As it turns out, they do.

Long before hexadecimal became the dominant notation used by programmers, many engineers preferred **octal**, a numbering system based on eight symbols rather than sixteen. This may seem like an unusual choice.

Why eight instead of sixteen?

The answer, once again, lies not in mathematics but in the practical realities of the hardware being designed at the time. Just as hexadecimal groups binary digits into sets of four, octal groups them into sets of three.

101 110 011 → 5 6 3 → 563₈

Three bits can represent exactly eight different combinations, making octal a natural shorthand for binary in much the same way that hexadecimal uses groups of four bits.

During the 1960s and 1970s, many computers were designed with word sizes that happened to be convenient multiples of three bits. Machines such as the PDP-8 and several other minicomputers naturally aligned with octal notation, making it an efficient way for programmers to examine memory contents, machine instructions and addresses.

As processor architectures evolved, however, hardware increasingly adopted word sizes that were multiples of four and eight bits. Eight-bit bytes became the standard building block of modern computing, and hexadecimal suddenly offered a much more natural correspondence.

Every byte could be represented by exactly two hexadecimal digits, making memory dumps, machine instructions and registers significantly easier to interpret. Hexadecimal has largely replaced octal in low-level programming.

That does not mean octal has disappeared entirely.

If you have ever used a Unix or Linux system, you have almost certainly encountered file permissions such as 755, 644 or 777. Although they appear to be ordinary decimal numbers, they are in fact octal values, each digit

representing three permission bits that control whether a file may be read, written or executed. For many, this is the only place octal is encountered.

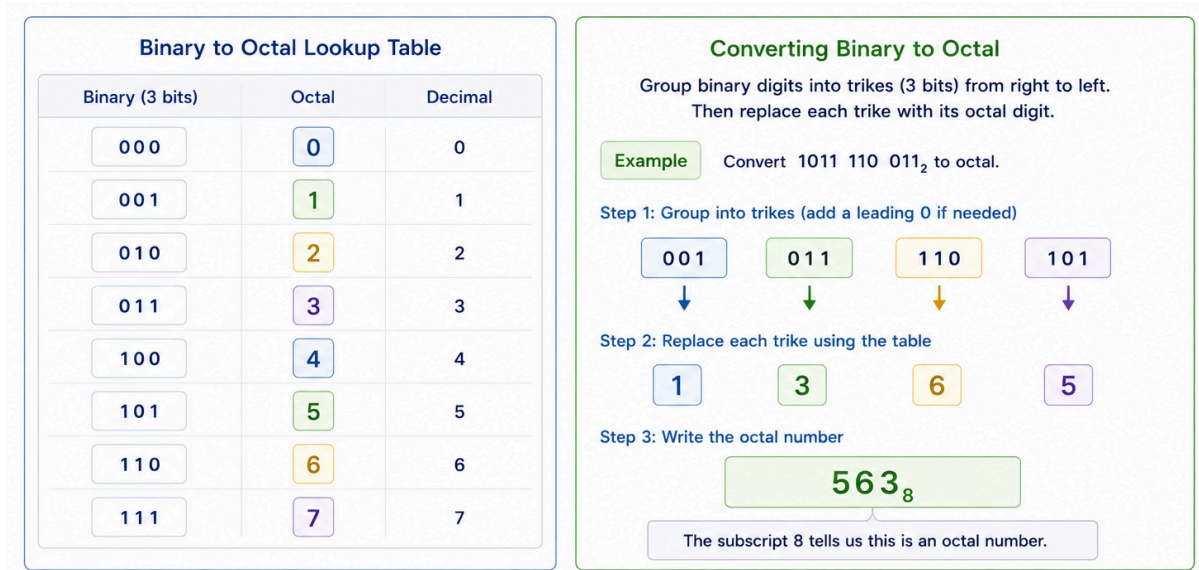


Figure 2.7. Octal groups binary digits into sets of three, allowing each group to be represented by a single octal digit.

The important lesson, however, extends beyond octal itself.

Throughout the history of computing, engineers have repeatedly adapted the representation of information to match the needs of the hardware and the people working with it.

Hexadecimal eventually became the preferred notation because processor architectures evolved in a way that made it more convenient, not because octal was somehow incorrect.

Like every numbering system we have encountered so far, octal is simply another language for describing exactly the same binary values.

Historical Note – The Age of Octal

During the early decades of computing, octal was far more common than hexadecimal. Digital Equipment Corporation's PDP series of minicomputers, along with several other influential systems of the 1960s and 1970s, were documented almost entirely in octal. Programmers became so accustomed to the notation that many could read octal values as fluently as later generations would read hexadecimal.

As eight-bit processors and standardized byte-oriented architectures became widespread, hexadecimal gradually displaced octal because its four-bit grouping aligned perfectly with modern hardware. Although octal survives in a handful of specialized applications, its role today is largely historical.

Converting Between Bases

Initially, converting between different numbering systems can seem like an exercise in memorizing rules. Textbooks typically present pages of algorithms for repeatedly dividing, multiplying or expanding numbers until they eventually produce the desired result.

While those techniques certainly work, they often leave the reader wondering why the process succeeds in the first place. By now, however, we are in a rather fortunate position.

We no longer think of binary, decimal, octal and hexadecimal as completely different kinds of numbers. Instead, we understand that they are simply different ways of writing exactly the same quantity. Once that idea has taken root, converting between them becomes far less mysterious.

The easiest conversion is between binary and hexadecimal.

As we have already seen, every hexadecimal digit corresponds to exactly four binary digits. This relationship is perfect. There are sixteen possible four-bit combinations and sixteen hexadecimal symbols, allowing each group to be translated independently of every other group.

Consider the following binary value.

```
1101011010111111
```

Rather than interpreting the entire sequence at once, we simply translate each group of four bits, and convert them individually to hexadecimal digits.

```
1101  0110  1011  1111  →  0xD6BF
```

No arithmetic is required because the grouping itself preserves the value.

The same idea applies when converting between binary and octal. Instead of grouping bits into sets of four, we group them into sets of three. Each three-bit group maps directly to a single octal digit, making the conversion just as straightforward.

Converting between decimal and binary requires a little more work because there is no such direct correspondence. Decimal is built upon powers of ten, while binary is built upon powers of two. Nevertheless, the underlying principle remains exactly the same.

Every digit contributes to the final value according to its position, and converting between the two systems simply means expressing the same quantity using a different set of place values.

For relatively small numbers, it is often easiest to expand the binary value into powers of two, just as we did earlier.

$$\begin{aligned} & 101101_2 \\ &= (1 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + \\ & \quad (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) \\ &= 45 \end{aligned}$$

The reverse process—converting from decimal to binary—follows the same reasoning in the opposite direction. Rather than asking which powers of two produced the original value, we determine which powers of two combine to recreate the decimal number.

The mechanics of these algorithms are less important than the idea behind them. Whether performed by hand, by a calculator or by a compiler, every conversion is simply another way of expressing an identical quantity.

This perspective becomes increasingly valuable as software grows more complex. A debugger may display a memory address in hexadecimal, a protocol analyzer may show binary flags, while a log file reports decimal values intended for human consumption.

Although the notation changes from tool to tool, the underlying information remains exactly the same. Experienced engineers quickly learn to move between these representations without thinking of them as different numbers.

From Experience

Early in my career I found myself constantly converting hexadecimal values back into decimal because it felt familiar. Every memory address, register value or bit mask became a small arithmetic exercise before I could understand what I was looking at.

Over time that habit gradually disappeared. I stopped thinking about the notation itself and began thinking about the information it represented. A value such as `0x8000` no longer meant "thirty-two thousand seven hundred and sixty-eight." Instead, it immediately suggested a single bit set in the most significant position. Likewise, `0xFF` stopped being "two hundred and fifty-five" and simply became "all eight bits are set."

Looking back, I realized I wasn't becoming faster at converting between number systems. I was learning to recognize the patterns that mattered.

In many ways, learning number systems is rather like becoming multilingual. A fluent speaker does not mentally translate every sentence before understanding it; they simply recognize the meaning directly. The same thing happens with experience in software development. After enough exposure, values such as `0xFF`, 11111111_2 , and `255` cease to feel like separate representations. They become different expressions of the same idea.

Looking Ahead

In the beginning,, a binary number and a hexadecimal value may have seemed like entirely different kinds of information. One appeared to belong to the world of computers, the other to the world of programmers, while decimal remained the familiar language of everyday life.

By now, those distinctions have largely disappeared.

We have discovered that binary, octal, decimal and hexadecimal are not competing ways of representing different quantities. They are simply different notations for expressing exactly the same value. Each evolved to solve a particular problem.

Decimal reflects the way humans have counted for thousands of years. Binary aligns perfectly with the physical realities of digital electronics. Hexadecimal provides a compact, human-friendly view of binary, while octal reminds us that engineering solutions often reflect the constraints of the era in which they were developed.

The numbers themselves never changed—only the notation did.

This is a surprisingly powerful idea because it reaches far beyond number systems. Throughout computing, we continually encounter different representations of the same underlying information.

An integer may be stored as binary in memory, displayed as hexadecimal in a debugger and manipulated by machine instructions inside the processor. A memory address, a machine instruction and a communication protocol may all appear very different, yet each is ultimately constructed from the same patterns of bits.

Representation changes—meaning does not.

We will find that the same principle appears again and again. We will explore how computers represent negative numbers without using a minus sign, how

fractions are approximated using floating-point arithmetic and why the order of bytes in memory sometimes matters.

Each of these topics builds directly upon the ideas introduced so far.

Understanding a computer has never been about memorizing isolated facts—it is about recognizing patterns. Once you begin to see those patterns, concepts that once appeared unrelated gradually reveal themselves to be different expressions of the same underlying principles.

Binary and hexadecimal are no longer strange notations to memorize. They become familiar tools, each chosen because it represents information in the clearest and most useful way for a particular task. That, perhaps, is the most important lesson so far.

Numbers have many different languages—however, the value never changes.

PREVIEW EDITION

to purchase a copy of the book
visit the following website

www.ardiri.com

the book is available in both
digital and physical form.

Every abstraction hides a machine.

Software feels intangible.
Applications appear effortless.
Artificial intelligence seems almost magical.

Yet beneath every operating system, every programming language, every cloud service and every intelligent application lies the same remarkable machine—one built from bits, logic, memory and carefully engineered abstractions that have evolved over decades.

Inside the Machine peels back those layers one by one, revealing how modern computers really work. Beginning with the smallest building block and progressing through processors, memory, storage, operating systems, virtualization, security and intelligent hardware, it connects concepts that are too often taught in isolation into a single coherent journey through computer architecture.

Whether you're a student, developer, engineer or simply curious about how computers really work, this book will change the way you think about the technology you use every day.

Because once you understand the machine, you'll never look at software in quite the same way again.

ABOUT THE AUTHOR



Aaron Ardiri is passionate about making complex technology accessible, he writes to help readers understand not only how computers work, but why they were designed the way they are.

**“It’s not about computers.
It’s about what’s inside the machine that makes them work.”**



PART I FOUNDATIONS

Bits, numbers, memory and data—how information is represented.

PART II EXECUTION

The CPU, instructions, memory hierarchy, buses and I/O—how the machine executes work.

PART III EVOLUTION

Parallel machines, virtual machines, secure machines, and intelligent machines

ISBN 978-91-531-9117-9



9 789153 191179

1 0 1
0 1 0
1 0 1

