Computer Architecture for Everybody

From Stonehenge to Silicon

Charles R. Severance

2026

Contents

0.1 From Stonehenge to Silicon: Why Computers Began with Numbers

Long before computers were used for communication, entertainment, or social interaction, they were built for something far more basic: measuring the world and predicting what would happen next. The earliest forms of computation were not abstract symbols stored in memory, but physical structures designed to model natural processes. These devices helped track the seasons, predict the motion of planets, guide travelers, and support engineering and trade. In this early period, computing was inseparable from mathematics and measurement, and numbers were the primary objects being manipulated.

This focus on numerical calculation remained dominant for most of computing history. Until the late twentieth century, computers were rare, expensive, and primarily used for scientific, military, and industrial purposes. There was no internet, and few people interacted directly with computing machines. Only later did computers become tools for information exchange and personal communication. To understand modern computer architecture, it helps to begin with this earlier world, where computation meant physically transforming numbers.

0.1.1 Curved Motion and the Need for Prediction

Much of the natural world moves along curves rather than straight lines. The arc of a thrown object, the path of the Moon across the sky, and the orbit of planets all follow curved trajectories. Predicting such motion is difficult because small errors accumulate over time, and precise prediction requires repeated calculation. While the human brain is good at intuitive estimates, accurate forecasting requires systematic measurement and mathematical modeling.

The practical need to predict motion—for agriculture, navigation, and astronomy—drove the development of early computational tools. These tools were not general-purpose machines, but specialized devices that captured particular physical relationships and made them easier to reason about. Each device encoded a small piece of mathematics in wood, stone, or metal.

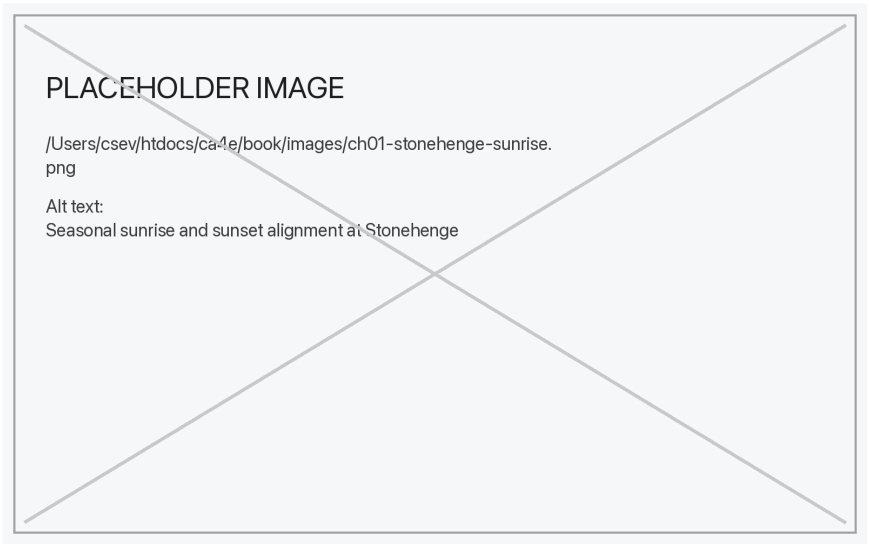

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch01-stonehenge-sunrise.png
Alt text:
Seasonal sunrise and sunset alignment at Stonehenge

Figure 1: Seasonal sunrise and sunset alignment at Stonehenge

---

0.1.2 Astronomy and Physical Data Tables

From the earliest civilizations, careful observation of the sky revealed repeating patterns. The positions of the Sun, Moon, and stars changed in predictable ways over the course of days, months, and years. Recording these observations allowed calendars to be built, planting seasons to be planned, and ceremonial events to be scheduled.

Over time, physical structures were constructed to embody these patterns directly. Instead of writing numbers in tables, builders placed stones or architectural features so that sunlight or shadows would align at particular times of year. These structures functioned as durable, physical data sets that could be read simply by observing the environment.

0.1.3 Stonehenge as a Measuring Device

Stonehenge, constructed in several phases between roughly 5100 and 3600 years ago, illustrates this idea clearly. The arrangement of stones aligns with sunrise and sunset positions that change throughout the year. By observing which stones line up with the Sun on a given day, seasonal transitions can be predicted.

Rather than storing numbers, Stonehenge stored geometric relationships. Over many generations, observations were effectively "written" into the landscape. In modern terms, the structure can be thought of as a calibrated data table, built incrementally through centuries of refinement.

Similar solar and lunar alignment structures exist across many cultures, from temples in India to monuments in Central America. All reflect the same underlying idea: physical construction can encode numerical patterns from nature.

> Physical computing did not begin with machines. It began with architecture.

0.1.4 Continuous Representations of Number

Many early computational devices represented numbers not as symbols, but as physical positions. A value might correspond to the angle of a gear, the distance of a slider, or the rotation of a dial. Because these values could vary smoothly, such systems are called continuous or analog.

In these devices, mathematical relationships are built into geometry. Adding distances can perform multiplication when scales are logarithmic. Rotating disks can solve trigonometric problems by turning angles into lengths. Instead of executing arithmetic step by step, the device produces results through physical alignment.

Accurate printed scales are essential for this approach. The precision of the computation depends directly on the precision of the markings and the mechanical stability of the device. In effect, the manufacturing process becomes part of the calculation.

0.1.5 Gears as Models of the Solar System

The Antikythera mechanism, dating to around 2100 years ago, represents one of the most sophisticated examples of ancient analog computing. This

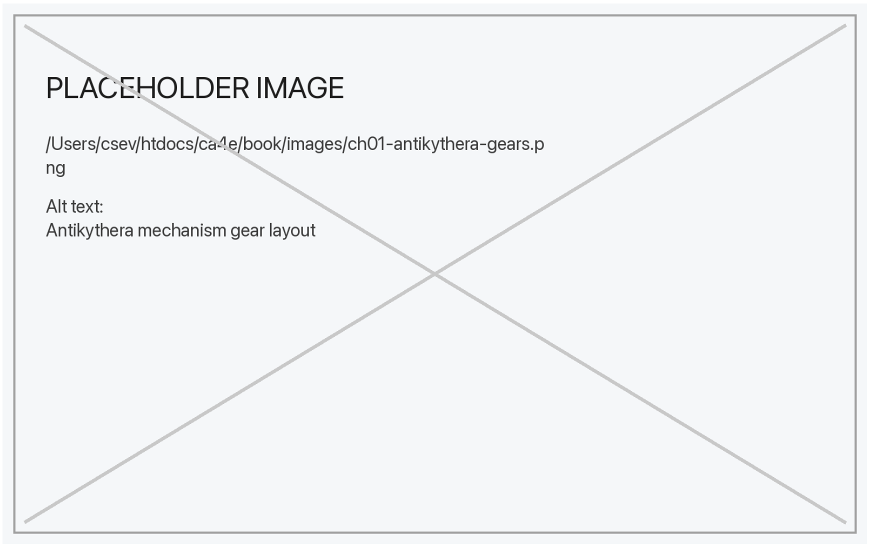

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch01-antikythera-gears.p
ng
Alt text:
Antikythera mechanism gear layout

Figure 2: Antikythera mechanism gear layout

device used interlocking gears to model the motion of the Sun, Moon, and known planets. Some of its gear trains represented long astronomical cycles spanning decades or even centuries.

Rather than calculating planetary positions numerically, the mechanism physically enacted the model of the cosmos that astronomers had developed. Turning a crank advanced time, and the gears moved accordingly. The computation occurred through mechanical interaction, not through written arithmetic.

This illustrates a broader principle that remains true today: computing systems implement models of reality. Whether those models are built from bronze gears or silicon transistors, the purpose is the same—to predict behavior by simulating it.

0.1.6 Practical Analog Computing

Analog computation did not remain confined to astronomy. Devices such as slide rules transformed multiplication into addition by using logarithmic scales. By aligning and sliding rulers, complex calculations could be performed quickly and reliably.

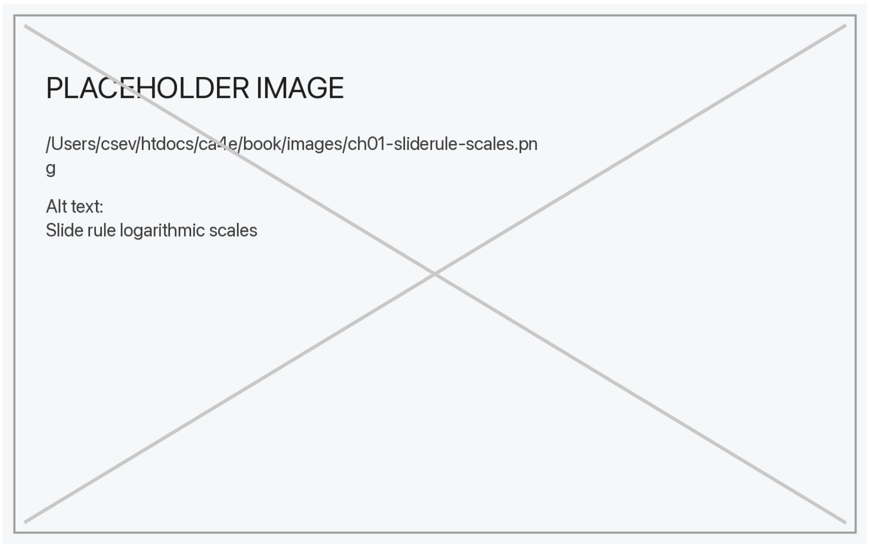

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch01-sliderule-scales.png
Alt text:
Slide rule logarithmic scales

Figure 3: Slide rule logarithmic scales

A particularly practical example is the E6B flight computer, still used by pilots to compute wind correction angles and ground speed. By rotating dials and aligning scales, trigonometric relationships are solved graphically. The device does not know anything about airplanes; it simply encodes geometric laws that apply to moving vectors.

In each case, the key idea is that physical movement stands in for mathematical transformation. The device performs computation because its shape embodies mathematical relationships.

0.1.7 Discrete States and Digital Devices

Not all physical computation is continuous. Some devices operate using discrete, stable states. A mechanical latch, for example, stays in one of two positions until enough force is applied to switch it. These stable configurations allow information to be stored physically.

Clocks provide a clear example of digital behavior in mechanical systems. A pendulum provides regular timing, while ratchets and gears count discrete events. When one gear completes a full rotation, it advances the

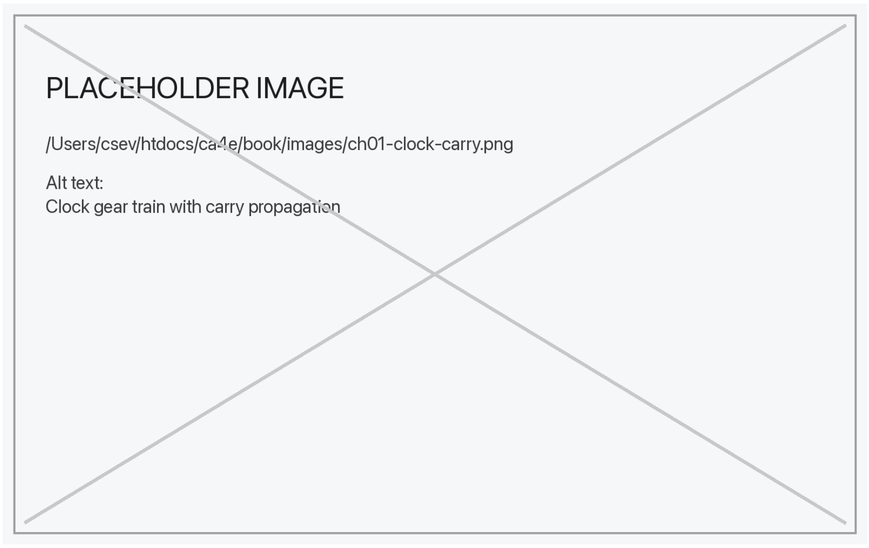

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch01-clock-carry.png
Alt text:
Clock gear train with carry propagation

Figure 4: Clock gear train with carry propagation

next gear, producing the familiar progression from seconds to minutes to hours.

This mechanism also introduces the concept of carry. When one digit overflows, the next digit is incremented. Mechanical systems must physically propagate this carry through connected components, a process that takes time and introduces delays. Similar effects still occur in electronic circuits, where signals must travel between components.

0.1.8 Iteration and Mechanical Automation

Once addition and counting are possible, repeated operations can be automated. Multiplication becomes repeated addition, and polynomial evaluation becomes a structured sequence of arithmetic steps. Adding motors or cranks allows machines to perform long sequences without human intervention.

Charles Babbage's Difference Engine, designed in the nineteenth century, exploited this idea. It used repeated addition to approximate complex mathematical functions and generate accurate tables. Although technology at the time could not easily produce all the required parts, a complete version built in the late twentieth century demonstrated that the design itself was sound.

Architectural ideas often appear long before manufacturing technology can fully support them.

0.1.9 Human Computers and Early Programming

Before electronic machines became common, teams of people performed large calculations using mechanical aids and strict procedures. These workers were called computers, and their job was to execute long sequences of operations reliably.

When electronic computers emerged, much of this procedural knowledge transferred directly. Programming languages such as FORTRAN reflected existing mathematical workflows, including loops and accumulation of results. Writing a program was, in many ways, formalizing what human computers had already been doing manually.

Thus, programming did not arise as a completely new activity. It evolved naturally from structured numerical work that had existed for generations.

0.1.10 From Mechanical to Electronic Switching

Mechanical systems are limited by friction, wear, and inertia. As machines grew faster and more complex, these physical limits became obstacles. Vacuum tubes offered a way to perform switching electronically, without moving parts.

Although tubes are inherently analog devices, additional circuitry allowed them to behave digitally by latching into stable high or low voltage states. Machines such as Colossus used thousands of tubes to perform computations far faster than electromechanical systems could achieve.

Heat and power consumption remained serious challenges, but electronic switching marked a fundamental shift. Computation was no longer constrained by mechanical motion.

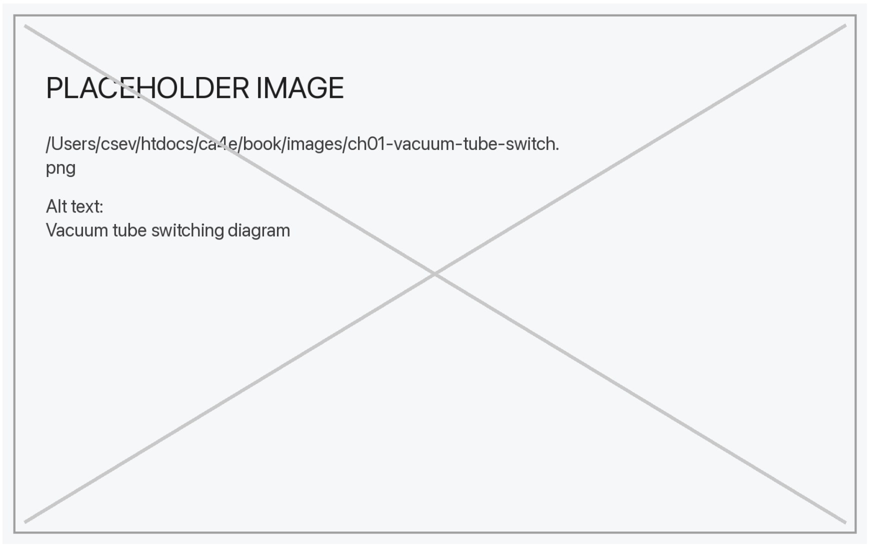

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch01-vacuum-tube-switch.png
Alt text:
Vacuum tube switching diagram

Figure 5: Vacuum tube switching diagram

0.1.11 Why Architecture Begins with Numbers

Across thousands of years, computational devices were developed to support astronomy, navigation, engineering, and accounting. These systems manipulated numbers that represented physical quantities: time, distance, angle, and mass. The architectural ideas that emerged—counting, carrying, iteration, state, and switching—remain central to computer design today.

Only later did computers become tools for processing text, images, and communication. Those capabilities were built on top of numerical foundations that had already been refined through centuries of physical computing devices.

Understanding this origin helps explain why modern processors still devote most of their structure to arithmetic and control of repeated operations.

0.1.12 What Comes Next

The transition from mechanical and electronic switching to solid-state devices transformed both the speed and scale of computation. In the next chapter, attention turns to how transistors replaced tubes and gears, and how tiny electrical switches became the building blocks of modern computer systems.

0.2 From Tubes to Transistors: How Solid-State Electronics Changed Everything

The earliest electronic computers replaced mechanical motion with electrical switching, but they still relied on bulky, fragile components that consumed large amounts of power. These components, called vacuum tubes or valves, made it possible to build fully electronic machines, yet they also imposed severe limits on speed, size, and reliability. The transition from tubes to transistors did far more than improve existing designs—it reshaped what computers could be and made modern computing possible.

This chapter follows that transition. It begins with electronic amplification, moves through the physics of semiconductors, and ends with logic gates built from complementary transistor pairs. Along the way, the story shifts from analog behavior to digital abstraction, laying the groundwork for everything that follows in computer architecture.

0.2.1 Electronic Switching with Vacuum Tubes

Vacuum tubes were the first practical electronic devices capable of amplifying and switching signals. In the United Kingdom they were commonly called valves, a name that reflects their function: controlling the flow of electrons much like a valve controls the flow of water. Invented in the early twentieth century, tubes were originally developed to amplify weak analog signals, especially for long□distance telephone communication, where signals had to be boosted every few miles.

By adjusting the design of a tube, engineers could make it behave more like a switch than an amplifier. When the input voltage crossed a threshold,

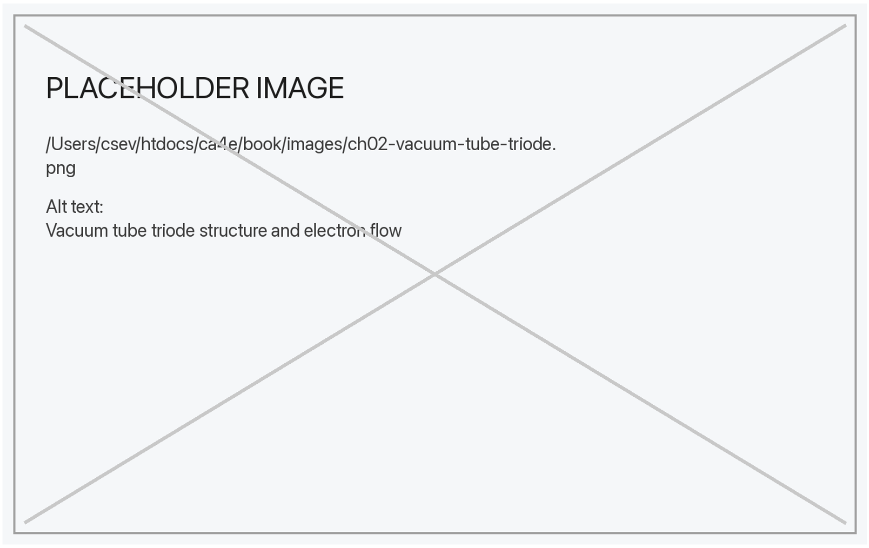

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch02-vacuum-tube-triode.png
Alt text:
Vacuum tube triode structure and electron flow

Figure 6: Vacuum tube triode structure and electron flow

the output would move rapidly from low to high voltage. This made it possible to represent logical states using electrical levels: low voltage for zero and high voltage for one.

Electronic switching was dramatically faster than mechanical relays, and it had no moving parts. However, tubes required internal heaters to release electrons from the cathode, which meant high power consumption and significant heat. They were also physically large, expensive to manufacture, and prone to failure over time.

Despite these limitations, tubes enabled the first generation of fully electronic computers, including machines such as Colossus, which performed calculations at speeds that mechanical systems could not approach.

0.2.2 The Limits of Tube-Based Computing

As computers grew larger and more complex, the drawbacks of vacuum tubes became increasingly serious. Thousands of tubes meant thousands of potential failure points. Cooling systems became massive engineering projects in their own right. Electrical power consumption limited how dense and how fast circuits could become.

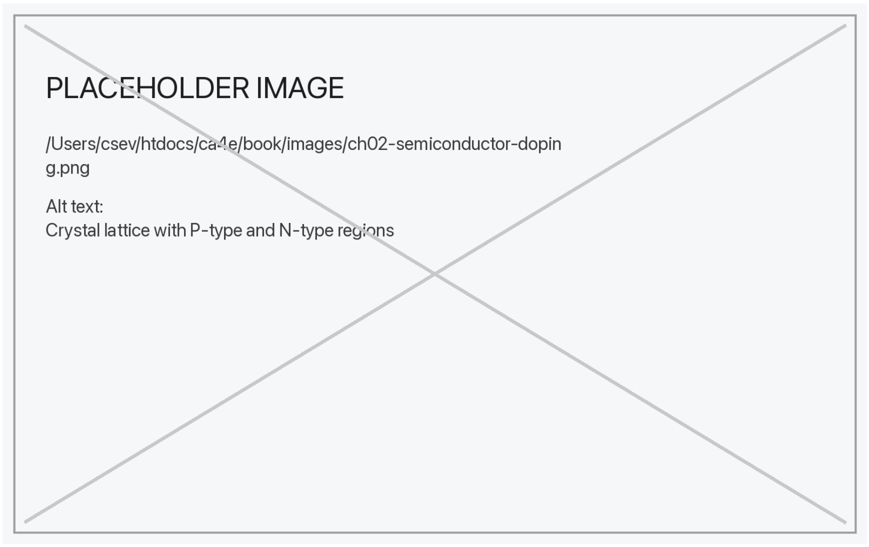

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch02-semiconductor-doping.png
Alt text:
Crystal lattice with P-type and N-type regions

Figure 7: Crystal lattice with P-type and N-type regions

The fundamental problem was not the logic itself, but the physical mechanism used to implement it. A better switching device was needed—one that did not rely on heating metal structures or maintaining vacuum environments.

That solution emerged from solid-state physics.

0.2.3 Semiconductors and Controlled Conductivity

Most materials fall into one of two categories when it comes to electricity: conductors, which allow current to flow easily, and insulators, which resist current strongly. Semiconductors occupy the middle ground. Under the right conditions, they can be made to conduct or block current in controlled ways.

Silicon, one of the most common elements in the Earth's crust, becomes useful for electronics when small amounts of other elements are added to it. This process, called doping, changes how electrons move through the crystal lattice. Regions with extra electrons are called N-type, while regions missing electrons are called P-type.

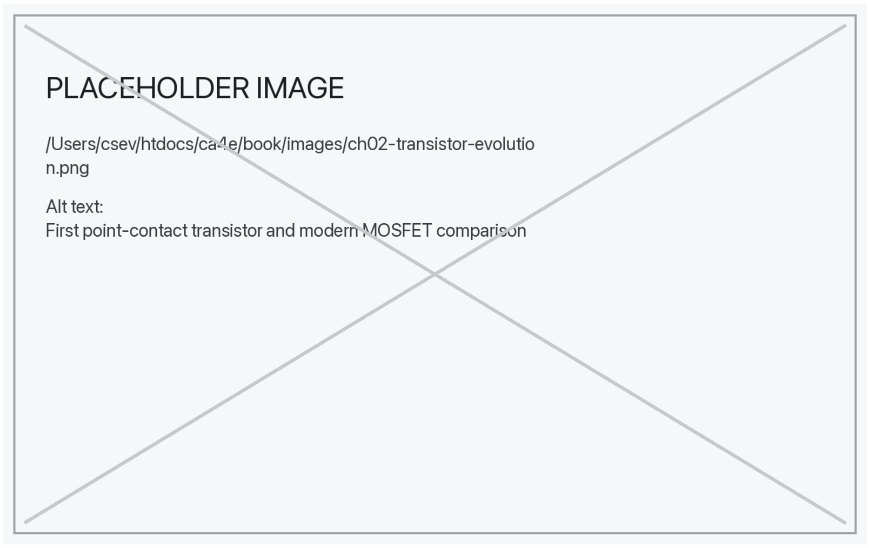

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch02-transistor-evolutio
n.png
Alt text:
First point-contact transistor and modern MOSFET comparison

Figure 8: First point-contact transistor and modern MOSFET comparison

When these regions meet, electrical behavior emerges that can be precisely controlled by external voltages. At the boundaries between P-type and N-type material, electric fields form that regulate the movement of charge carriers. These microscopic interactions make it possible to build devices that switch and amplify signals without moving parts or heaters.

0.2.4 Transistors as Electronic Switches

The transistor was developed as a solid-state replacement for the triode vacuum tube. Like tubes, transistors could amplify signals and could also be designed to behave as switches. But unlike tubes, transistors were small, required little power, and generated far less heat.

Early transistors were built using bipolar junction designs, where current flowing through one region controlled current in another. Later designs used metal-oxide-semiconductor structures, where an electric field at a control terminal called the gate regulated conduction through a channel of semiconductor material.

In both cases, the key property was the same: a small input signal could reliably control a larger output current. This allowed transistors to serve as building blocks for both analog amplifiers and digital logic circuits.

As manufacturing techniques improved, transistors became smaller, faster, and cheaper. What began as laboratory prototypes soon became mass-produced components found in radios, calculators, and eventually computers.

0.2.5 Analog Amplifiers and Digital Switching

Transistors did not immediately replace tubes in all applications. Audio amplification, for example, has long relied on both technologies, and some musicians and audio engineers still prefer tube-based amplifiers for their characteristic distortion patterns.

However, digital logic places very different demands on electronic components. In digital systems, what matters most is not the precise shape of a waveform, but whether a signal is interpreted as high or low. Designs that move quickly and decisively between these two states are far more useful than those that reproduce subtle analog variations.

To support digital operation, transistor designs were tuned to behave like fast, reliable switches rather than smooth amplifiers. Circuits were engineered so that small deviations in input voltage would be corrected at the output, restoring clean logical values. This behavior, called signal regeneration, is essential for building large digital systems where noise and small errors would otherwise accumulate.

0.2.6 NMOS, PMOS, and Complementary Pairs

Different transistor structures conduct under different conditions. NMOS transistors conduct when their gate voltage is high, while PMOS transistors conduct when their gate voltage is low. Each type has advantages and disadvantages when used alone.

Early integrated circuits often used only one type of transistor, resulting in designs that consumed power even when not switching and generated significant heat. As chip densities increased, this became a serious limitation.

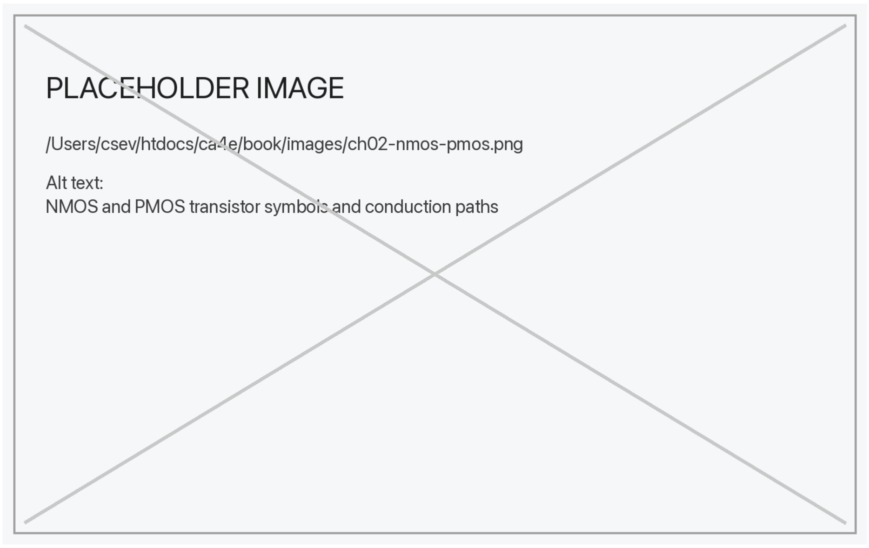

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch02-nmos-pmos.png
Alt text:
NMOS and PMOS transistor symbols and conduction paths

Figure 9: NMOS and PMOS transistor symbols and conduction paths

The solution was to pair NMOS and PMOS transistors in complementary configurations. In such arrangements, when one transistor is on, the other is off. This drastically reduces static power consumption and improves switching behavior. The resulting technology is called CMOS, short for Complementary Metal□Oxide□Semiconductor.

Once CMOS manufacturing became practical in the late twentieth century, it transformed computer design. Entire processors could be placed on single chips, power requirements dropped dramatically, and circuit densities increased by orders of magnitude.

From this point forward, nearly all digital logic in mainstream computing has been built using CMOS technology.

0.2.7 From Transistors to Logic Gates

While circuits are built from transistors, designers rarely think in terms of individual switching devices when creating complex systems. Instead, transistors are grouped into higher-level structures called logic gates. A gate accepts one or more binary inputs and produces a binary output according to a logical rule.

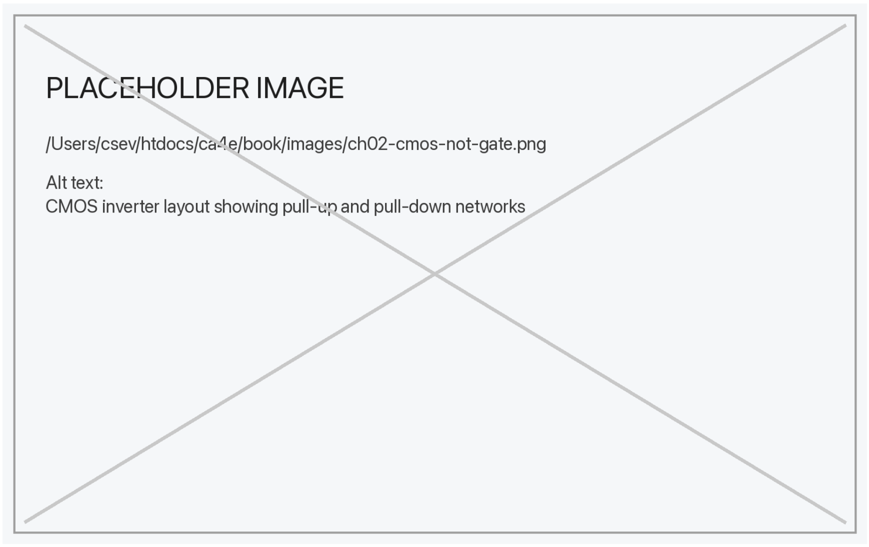

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch02-cmos-not-gate.png
Alt text:
CMOS inverter layout showing pull-up and pull-down networks

Figure 10: CMOS inverter layout showing pull-up and pull-down networks

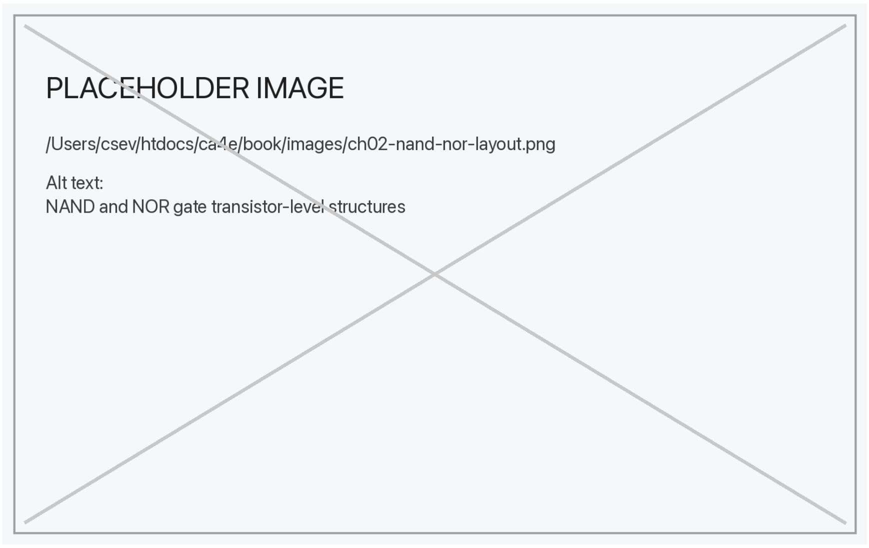
The simplest gate is the inverter, or NOT gate, which produces the opposite of its input. In CMOS, an inverter is built from one NMOS transistor and one PMOS transistor arranged so that exactly one conducts at any time. When the input is low, the PMOS transistor pulls the output high. When the input is high, the NMOS transistor pulls the output low.

This complementary behavior provides fast switching and low power usage, making it ideal for large-scale digital systems.

0.2.8 Building Complexity from Simple Gates

Once reliable gates are available, more complex logical functions can be constructed by combining them. Gates such as AND, OR, and exclusive OR implement familiar logical operations. Other gates, such as NAND and NOR, are especially important because entire digital systems can be built using only one of these gate types.

Designing digital circuits at the gate level allows engineers to reason about behavior using binary logic instead of voltage levels and transistor physics. This abstraction makes it possible to build systems containing billions of transistors without managing each device individually.


PLACEHOLDER IMAGE

/Users/csev/htdocs/ca4e/book/images/ch02-nand-nor-layout.png

Alt text:
NAND and NOR gate transistor-level structures

Figure 11: NAND and NOR gate transistor-level structures

From half adders and full adders to registers and processors, nearly every component in a computer can be described as an arrangement of logic gates operating in synchronized patterns.

0.2.9 Why CMOS Made Modern Computers Possible

Before CMOS, logic circuits were relatively slow, generated large amounts of heat, and could not be densely packed. Computers filled rooms and required elaborate cooling and power infrastructure. With the rise of CMOS, these constraints relaxed dramatically.

As transistor sizes shrank and manufacturing improved, entire central processing units moved from cabinets to circuit boards, and then onto single integrated chips. Power efficiency improved, clock speeds increased, and reliability soared.

This shift did not change the logical structure of computation, but it changed the physical feasibility of building large and fast systems. The same architectural ideas—state, control, and iteration—could now be implemented at scales that earlier engineers could only imagine.

0.2.10 Summary: Solid-State Foundations of Digital Logic

The move from vacuum tubes to transistors replaced fragile, power-hungry components with small, efficient solid-state devices. Advances in semiconductor physics and manufacturing enabled reliable electronic switching without mechanical motion or heated filaments.

Complementary transistor designs made CMOS the dominant technology for digital logic, allowing dense, low-power circuits to become practical. By grouping transistors into logic gates, designers gained an abstraction that supports the construction of complex systems without constant reference to underlying physics.

With these foundations in place, attention can now shift from individual devices to how large collections of gates are organized into functional computing units.

0.2.11 What Comes Next

With solid-state logic established, the next step is to explore how gates are combined into arithmetic circuits and memory structures. The following chapter examines how simple logical components are assembled into adders, registers, and the building blocks of processors.

0.3 Very Large Scale Integration: From Individual Devices to Entire Chips

The invention of the transistor made electronic switching reliable and efficient, but early circuits still consisted of individual components wired together on circuit boards. Each transistor, resistor, and capacitor had to be manufactured separately and then connected by hand or by automated assembly. While this approach worked for small systems, it quickly became impractical as circuits grew more complex.

Very Large Scale Integration (VLSI) describes the set of technologies that made it possible to place enormous numbers of transistors onto a single piece of silicon. Instead of assembling computers from individual components, entire processors could be manufactured as unified physical systems. This shift did not merely improve performance; it fundamentally changed how computers were designed, built, and scaled.

0.3.1 From Discrete Components to Integrated Circuits

Early transistor-based electronics were constructed much like mechanical systems: individual parts were mounted on boards and connected with wires or metal traces. Even when integrated circuits first appeared, they typically contained only a handful of transistors or logic gates.

These early chips reduced size and improved reliability, but they did not yet enable full processors to be placed on a single device. Complex systems still required many chips working together, with signals traveling across boards and connectors. Speed was limited by how fast electrical signals could move between packages, and cost increased with every additional component.

The next major leap required changing not just circuit design, but manufacturing itself.

0.3.2 Planar Transistors and Two-Dimensional Manufacturing

Before the 1960s, transistors were often built as small three-dimensional structures that had to be individually assembled and wired. This made large-scale manufacturing slow, expensive, and difficult to automate.

The planar transistor process, developed at Fairchild Semiconductor, transformed transistor fabrication into a two-dimensional surface process. Instead of assembling parts, chemical and photographic techniques were used to shape transistor structures directly on the surface of silicon wafers.

By depositing materials in layers and selectively removing regions using masks, entire arrays of transistors could be created simultaneously. This approach allowed thousands, and eventually billions, of devices to be produced in a single manufacturing run.

Planar fabrication made integration scalable.

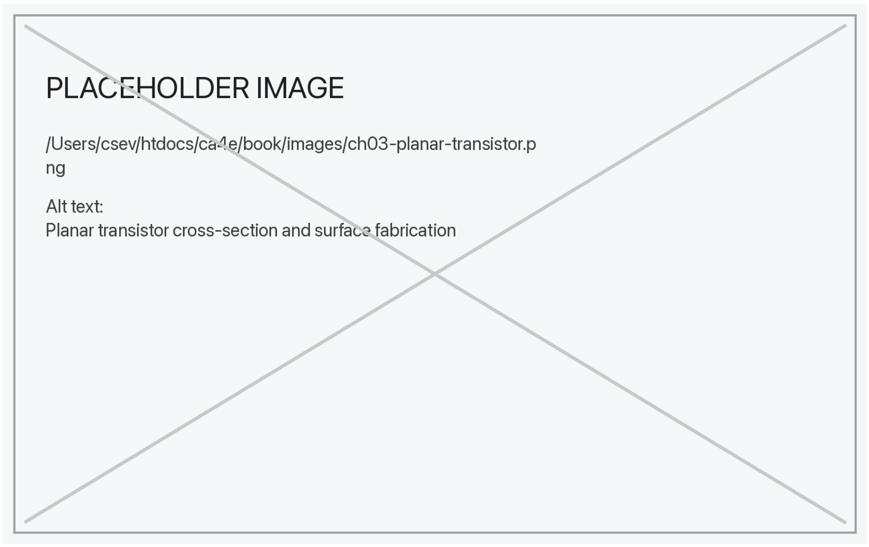

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch03-planar-transistor.png
Alt text:
Planar transistor cross-section and surface fabrication

Figure 12: Planar transistor cross-section and surface fabrication

---

0.3.3 Wafers, Masks, and Photolithography

Modern chip manufacturing begins with thin circular slices of silicon called wafers. Each wafer contains many identical copies of a chip design, arranged in a grid pattern. After fabrication, the wafer is cut into individual chips, each of which becomes a processor, memory device, or controller.

Patterns are transferred onto wafers using photolithography. In this process, light-sensitive chemicals are exposed through precisely designed masks. Each mask defines where material will be added, removed, or altered during that step of fabrication.

Multiple masks are used in sequence to build up complex three-dimensional structures from stacked layers. Although the manufacturing process is layered, the design itself is typically described as a two-dimensional layout, where different materials occupy specific regions and intersections.

This manufacturing model is what allows enormous numbers of transistors to be created with extremely high consistency.

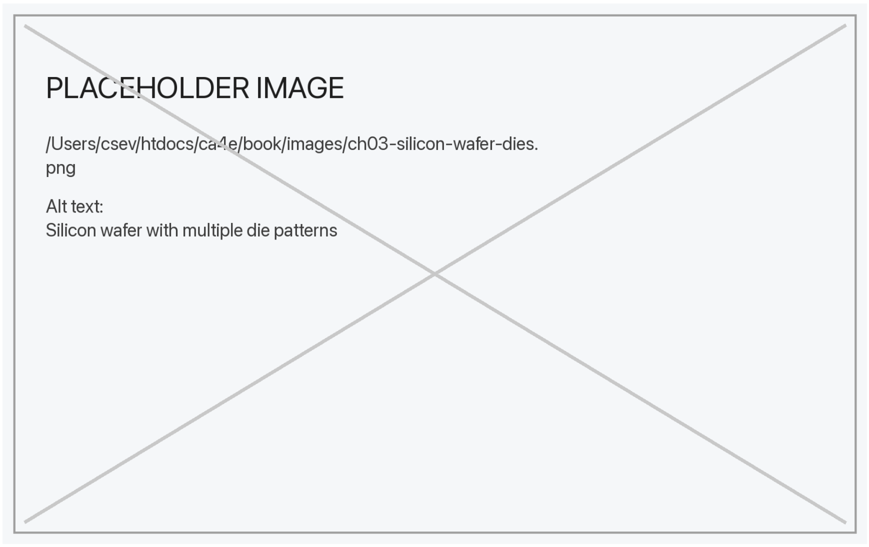

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch03-silicon-wafer-dies.png
Alt text:
Silicon wafer with multiple die patterns

Figure 13: Silicon wafer with multiple die patterns

0.3.4 Growth of Transistor Density Over Time

Once planar manufacturing was established, transistor counts began to grow rapidly. The Intel 4004 microprocessor, released in 1971, contained approximately 2,300 transistors and was the first commercially available single-chip general-purpose processor.

Over the following decades, transistor counts increased by orders of magnitude. By the early 2000s, chips contained billions of transistors. Modern graphics processors and specialized accelerators now contain tens or even hundreds of billions of devices, and experimental wafer-scale systems integrate trillions of transistors across entire wafers.

This steady growth is often associated with Moore's Law, an observation that transistor density tends to double over fixed time intervals. While physical limits now constrain how small transistors can become, integration remains the defining feature of modern computing hardware.

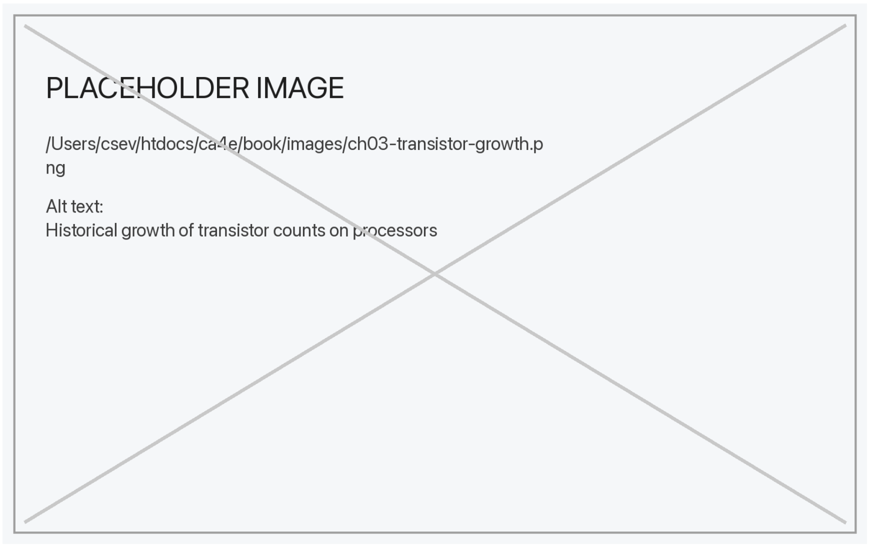

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch03-transistor-growth.png
Alt text:
Historical growth of transistor counts on processors

Figure 14: Historical growth of transistor counts on processors

0.3.5 Designing with Layout, Not Wires

As transistor counts increased, manual wiring became impossible. Instead, designers describe circuits using layout patterns that specify where materials will be placed on the wafer.

VLSI layout tools allow engineers to design using layers that represent different physical materials:

- metal for conductors
- doped silicon regions for transistor channels
- polysilicon for control gates

When certain layers cross, transistors are formed automatically by the manufacturing process. Vertical connections between layers are created using structures called vias, which allow signals to move between wiring layers.

Rather than drawing individual transistors explicitly, designers arrange geometric regions whose interactions produce the desired electrical behavior. Logical structure emerges from physical geometry.

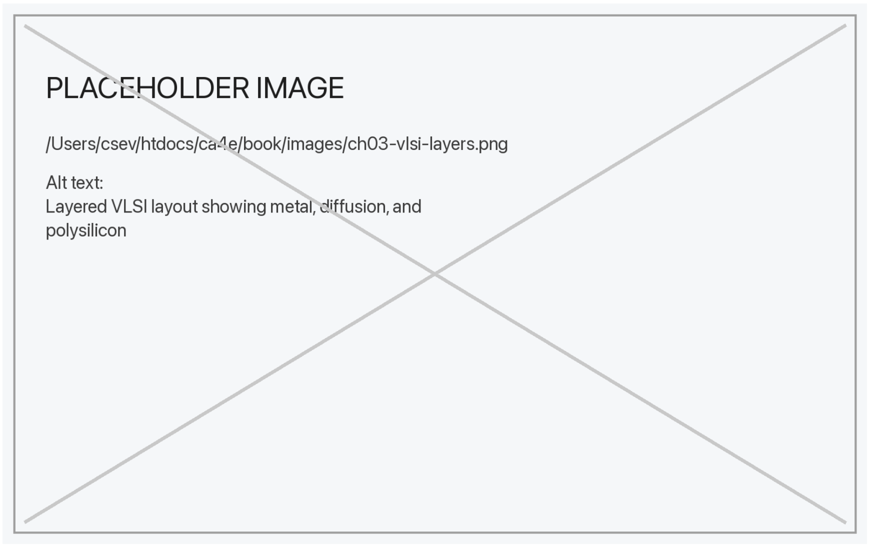

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch03-vlsi-layers.png
Alt text:
Layered VLSI layout showing metal, diffusion, and polysilicon

Figure 15: Layered VLSI layout showing metal, diffusion, and polysilicon

---

0.3.6 From Layout to Logic Gates

Just as transistors are grouped into logic gates in circuit design, layout patterns are arranged to produce gate behavior at the physical level.

A CMOS inverter, for example, is created by arranging one PMOS and one NMOS transistor so that exactly one conducts for any input value. When input voltage is low, the PMOS pulls the output high. When input voltage is high, the NMOS pulls the output low.

More complex gates such as NAND and NOR are built by combining multiple transistors in series and parallel arrangements. The same logical relationships studied in gate diagrams appear again in physical form as geometric structures on silicon.

This repetition of structure across abstraction levels is one of the defining characteristics of computer architecture: logical design mirrors physical implementation.

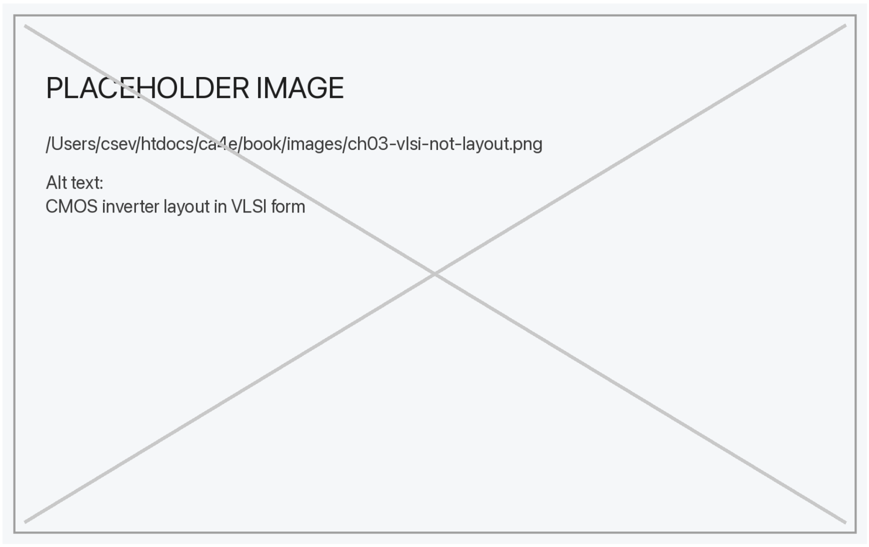

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch03-vlsi-not-layout.png
Alt text:
CMOS inverter layout in VLSI form

Figure 16: CMOS inverter layout in VLSI form

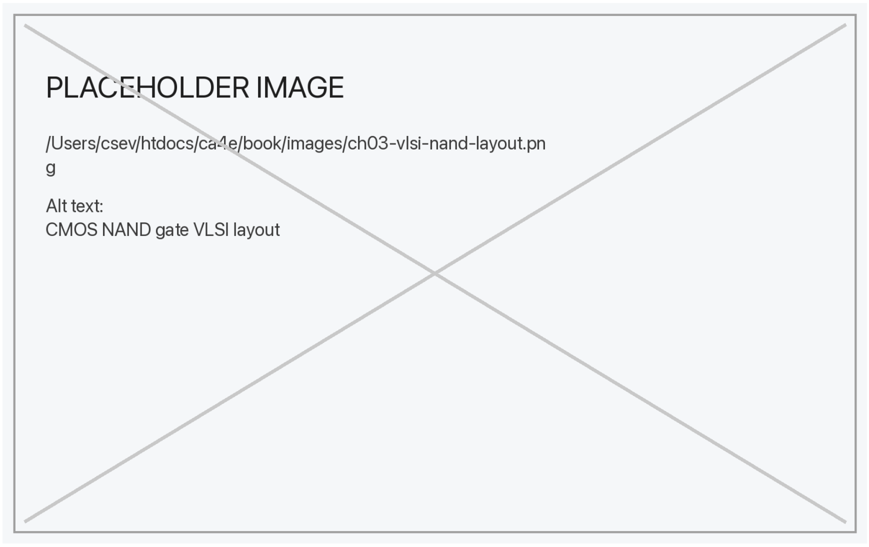

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch03-vlsi-nand-layout.png
Alt text:
CMOS NAND gate VLSI layout

Figure 17: CMOS NAND gate VLSI layout

0.3.7 Design Tools and Educational Models

Professional VLSI design requires sophisticated tools that model electrical behavior, timing, power consumption, and manufacturing constraints. One of the most widely used educational tools for learning layout is the open-source Magic VLSI system, originally developed at Berkeley in the early 1980s and still widely used in academic instruction.

Educational layout environments simplify many aspects of real fabrication. They may allow layer overlaps or ignore detailed spacing rules that would be impossible to manufacture reliably. These simplifications make it easier to understand core ideas, even if the resulting layouts would not be suitable for commercial fabrication.

Simplified emulators and browser-based tools used for instruction focus on conceptual correctness rather than full physical accuracy. They illustrate how geometry produces logical behavior without modeling every manufacturing constraint.

0.3.8 Why Integration Changes Architecture

When entire systems fit onto single chips, communication between components becomes far faster and more reliable. Signals no longer need to traverse long board traces or pass through connectors between packages. Memory, arithmetic units, and control logic can be tightly integrated and optimized together.

This physical proximity enables architectural features that would be impractical in discrete-component systems, such as deep pipelines, large on-chip caches, and complex parallel execution units.

Integration also shifts economic considerations. Instead of assembling systems from parts, the primary cost becomes chip design and fabrication, while replication becomes inexpensive once manufacturing is established.

As a result, architectural decisions increasingly reflect trade-offs between silicon area, power consumption, and performance rather than assembly complexity.

0.3.9 Summary: Manufacturing Enables Abstraction

Very Large Scale Integration made it possible to place entire computing systems onto single pieces of silicon. Planar fabrication, photolithography, and layered manufacturing transformed transistor construction into a massively parallel industrial process.

Layout-based design replaced manual wiring, and logical structures emerged from physical geometry. Transistors became gates, gates became arithmetic units, and complete processors became manufacturable as single devices.

With integration established, the focus of computer architecture can now shift away from manufacturing techniques and toward how logic itself is organized to perform computation.

0.3.10 What Comes Next

With logic gates available as reliable building blocks, the next step is to examine how arithmetic operations and memory storage are implemented using combinations of gates. The following chapter explores how numbers are added, stored, and manipulated using purely digital logic structures.

0.4 Digital Logic: Building Computation from Gates

With reliable transistors and scalable manufacturing in place, attention can shift from how devices are built to how computation itself is organized. Digital logic provides the abstraction that allows electrical behavior to be treated as mathematical structure. Voltages become symbols, wires become signals, and circuits become logical systems that can be reasoned about using rules rather than physical measurements.

This chapter introduces the logical building blocks of computers: gates, adders, and storage elements. These components form the foundation of processors and memory systems.

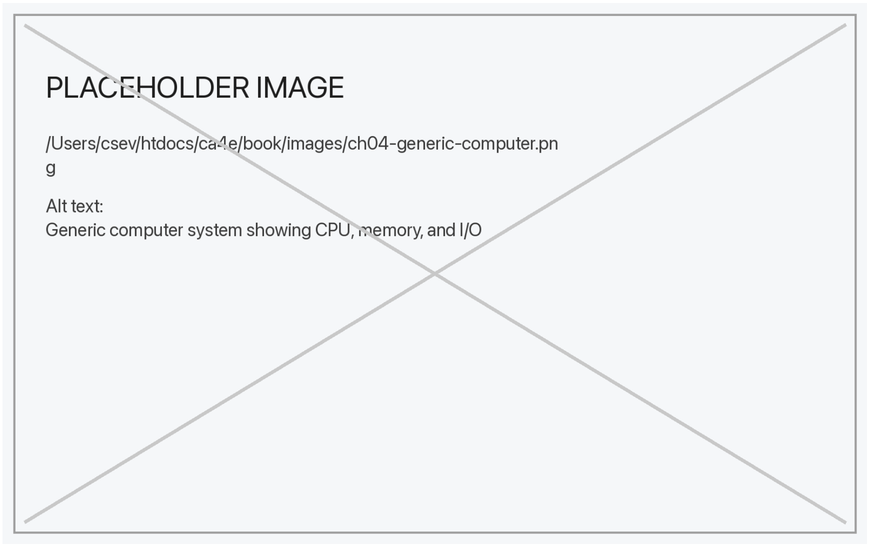

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch04-generic-computer.png
Alt text:
Generic computer system showing CPU, memory, and I/O

Figure 18: Generic computer system showing CPU, memory, and I/O

0.4.1 Computers as Interconnected Components

Inside a computer, information moves along wires that carry electrical signals. Each signal is interpreted as either a logical zero or one. Groups of wires connect major subsystems such as the central processing unit, memory, and input/output devices.

Although software describes computation in abstract terms, every operation ultimately becomes patterns of electrical signals traveling between physical components. Digital logic provides the framework that connects these physical movements to symbolic meaning.

0.4.2 What the CPU Does

The central processing unit (CPU) executes programs by repeatedly performing a simple cycle: fetch an instruction, interpret it, and perform the required operation. The CPU is not intelligent in the human sense. It does not understand goals or meaning. Instead, it follows mechanical rules at very high speed, executing billions of operations per second in modern systems.

Programs written in high-level languages are translated into machine instructions that the CPU can execute directly. Each instruction specifies small operations such as moving data, performing arithmetic, or testing conditions.

At the hardware level, these operations are implemented entirely using combinations of logic gates and storage elements.

0.4.3 Logic Gates as Building Blocks

Logic gates accept one or more binary inputs and produce a binary output. Each gate implements a simple logical rule.

The most common gates include:

- **NOT**, which inverts a signal
- **AND**, which produces one only if all inputs are one
- **OR**, which produces one if any input is one
- **XOR**, which produces one if inputs differ

Although these operations are simple, they are sufficient to construct any digital computation when combined appropriately.

0.4.4 Representing Numbers in Binary

To perform arithmetic using logic gates, numbers must be represented using electrical signals. Humans normally use base□10 representation, where each digit represents a power of ten. Digital systems instead use base□2 representation, where each digit represents a power of two.

For example:

- The base□10 number 6 is written as **110□** in binary.
- The base□10 number 7 is written as **111□**.

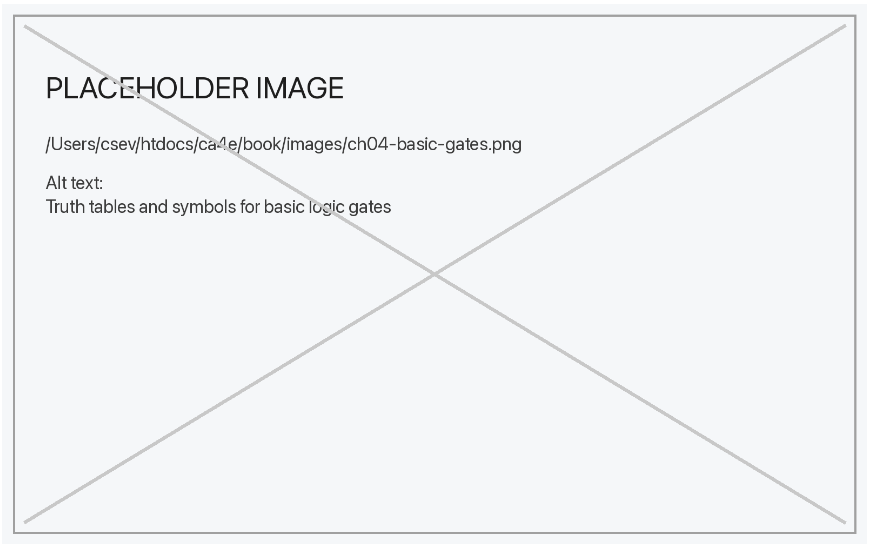

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch04-basic-gates.png
Alt text:
Truth tables and symbols for basic logic gates

Figure 19: Truth tables and symbols for basic logic gates

Each bit position corresponds to a weight:

- leftmost bit → 4
- middle bit → 2
- rightmost bit → 1

Binary representation allows numerical values to be manipulated using simple logical operations on individual bits.

0.4.5 Adding Numbers with Gates: Half Adders

The simplest arithmetic operation is addition. When adding two single bits, there are four possible input combinations. The result must produce both a sum bit and a carry bit.

A **half adder** is a circuit that adds two bits and produces:

- a **sum** output

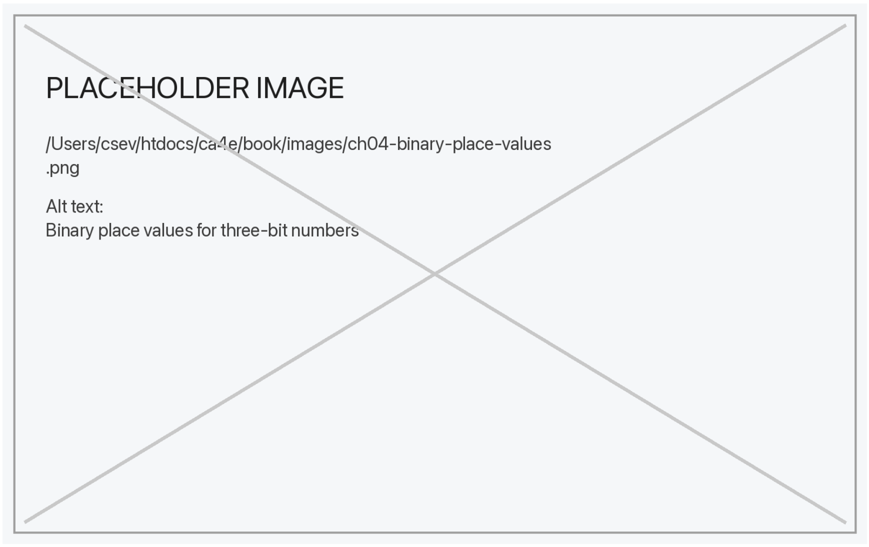

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch04-binary-place-values
.png
Alt text:
Binary place values for three-bit numbers

Figure 20: Binary place values for three-bit numbers

- a **carry** output

The sum output is produced by an XOR gate, while the carry output is produced by an AND gate.

This circuit performs correct binary addition for single-bit values.

0.4.6 Full Adders and Multi-Bit Addition

When adding multi-bit numbers, each bit position must also consider a carry value from the previous position. A **full adder** extends the half adder by adding three inputs:

- bit A
- bit B
- carry□in

It produces:

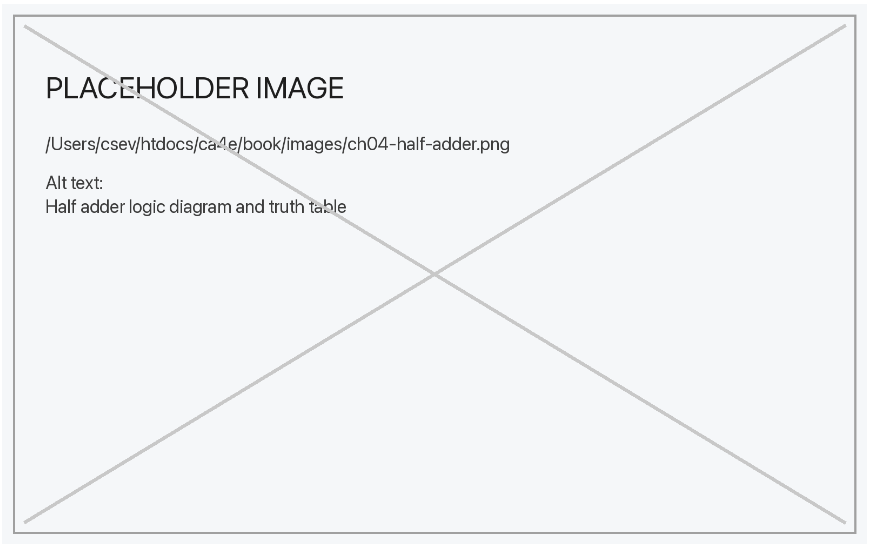

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch04-half-adder.png
Alt text:
Half adder logic diagram and truth table

Figure 21: Half adder logic diagram and truth table

- a sum bit
- a carry□out bit

By chaining full adders together, multi-bit addition can be performed. Each stage passes its carry output to the next stage, allowing numbers of arbitrary length to be added.

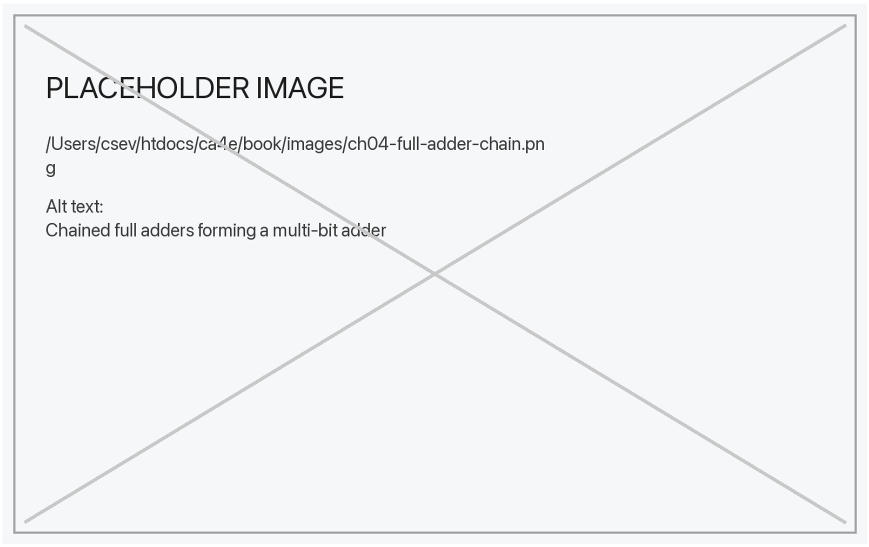
In practice, processors use more sophisticated adder designs to improve speed, but the fundamental principle remains the same.

0.4.7 Storing Data with Feedback

Computation requires not only processing data but also remembering it. Storage is implemented using circuits that maintain state over time.

The simplest storage element uses **feedback**, where part of the output is fed back into the input of the circuit. This allows a value to persist even when the original input signal is removed.

An example is the **set-reset (SR) latch**, built from two cross□connected NOR gates. Depending on the control inputs, the latch can store either a


PLACEHOLDER IMAGE

/Users/csev/htdocs/ca4e/book/images/ch04-full-adder-chain.pn
g

Alt text:
Chained full adders forming a multi-bit adder

Figure 22: Chained full adders forming a multi-bit adder

zero or a one.

Feedback loops introduce a new behavior: the circuit's output depends not only on current inputs, but also on past states.

0.4.8 Clocked Storage: Gated D Latches

While simple latches can store data, processors require more controlled storage that changes only at specific times. This is achieved by introducing a clock signal.

A **gated D latch** has:

- a data input (D)

- a clock or control input (C)

When the clock is active, the latch copies the data input into its internal state. When the clock is inactive, the stored value is held constant regardless of changes to the input.

This behavior allows many storage elements to update in synchronized steps, forming the basis of registers and memory systems.

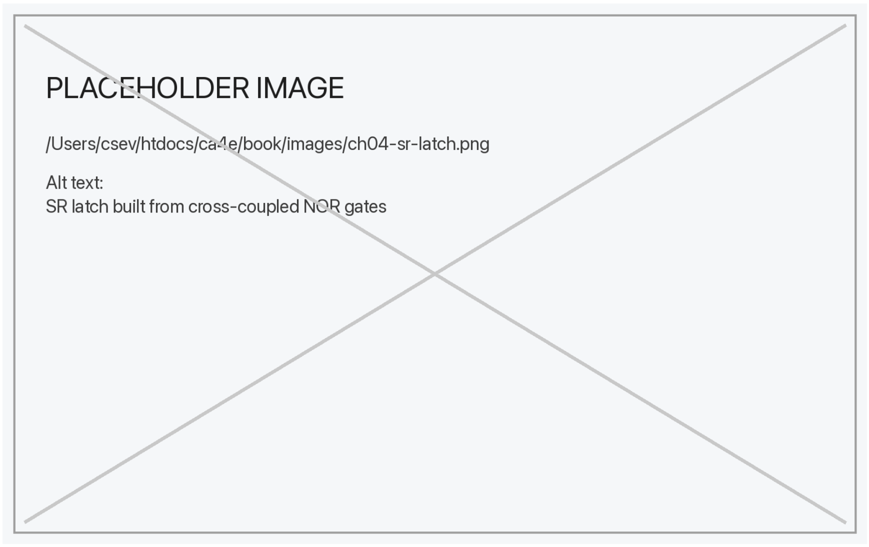

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch04-sr-latch.png
Alt text:
SR latch built from cross-coupled NOR gates

Figure 23: SR latch built from cross-coupled NOR gates

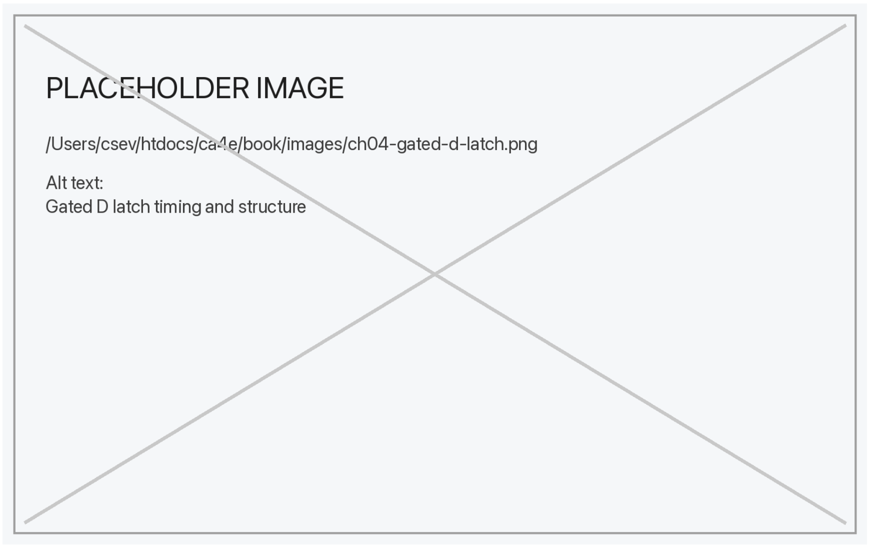

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch04-gated-d-latch.png
Alt text:
Gated D latch timing and structure

Figure 24: Gated D latch timing and structure

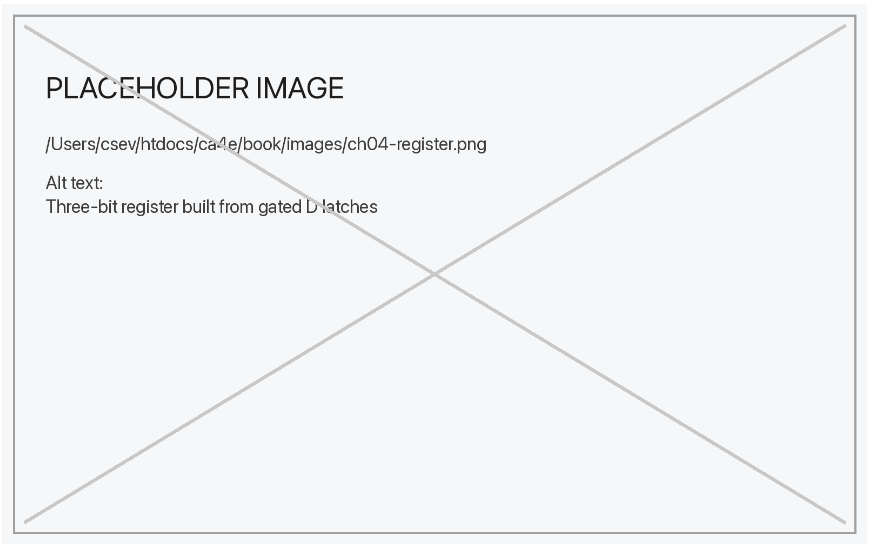

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch04-register.png
Alt text:
Three-bit register built from gated D latches

Figure 25: Three-bit register built from gated D latches

0.4.9 Registers from Multiple Latches

By grouping multiple gated D latches together, multi-bit storage units can be created. For example, three latches can store a three-bit number. Larger registers store entire machine words, allowing processors to hold intermediate values during computation.

Registers provide fast, temporary storage that supports arithmetic operations, branching decisions, and data movement within the CPU.

These structures form the immediate working memory of the processor.

0.4.10 Tools for Exploring Digital Logic

Modern educational tools allow digital circuits to be built and tested interactively. Gate-level simulators can display signal flow, timing, and logical behavior, making it easier to understand how complex systems emerge from simple components.

Some tools support layout-level construction, while others focus on abstract gate connectivity. Both approaches reinforce the relationship between physical hardware and logical structure.

By experimenting with gates, adders, and latches, it becomes clear that computation arises not from individual devices but from organized patterns of interaction.

0.4.11 Summary: From Gates to Computation

Digital logic transforms electrical behavior into mathematical structure. Logic gates implement simple rules, adders perform arithmetic, and latches store information over time.

By combining these components, complex machines can be built that execute programs, manipulate data, and respond to inputs. The physical realities of electronics remain present, but abstraction allows designers to reason about systems in logical terms.

With digital logic in place, the final step toward building a processor is introducing coordinated timing and instruction sequencing.

0.4.12 What Comes Next

The next chapter introduces clocked circuits and control logic. These mechanisms coordinate when data moves and when operations occur, allowing entire programs to be executed step by step inside the processor.

0.5 Clocked Circuits: Coordinating Computation Over Time

Combinational logic circuits, such as adders and logic gates, produce outputs that depend only on their current inputs. However, physical circuits do not update instantaneously. Signals require time to propagate through transistors and wires before stabilizing at their final values. As circuits grow larger and more complex, this delay becomes increasingly important.

To build reliable systems, digital computers introduce a coordinating signal called a **clock**. The clock defines discrete moments when values are allowed to change, allowing computation to proceed in synchronized steps rather than continuously drifting through intermediate states.

0.5.1 Propagation Delay and Circuit Length

When an input changes, it takes time for the effects of that change to travel through the circuit. Each transistor introduces a small delay, and long paths through many devices accumulate larger delays.

For example:

- a simple gate settles quickly
- a multi-bit adder takes longer
- a multiplier or complex control path may take much longer

If outputs are observed before signals have fully settled, incorrect values may be captured. Circuit designers therefore identify the **slowest path** through a system, known as the critical path, and ensure that enough time passes before results are used.

0.5.2 Clock Rate and System Timing

The clock rate of a processor specifies how often values are allowed to be updated. Each clock cycle provides time for signals to propagate and stabilize before being stored.

The maximum clock frequency is limited by:

- the physical length of signal paths
- the number of transistors in the longest logical path
- the switching speed of the transistors

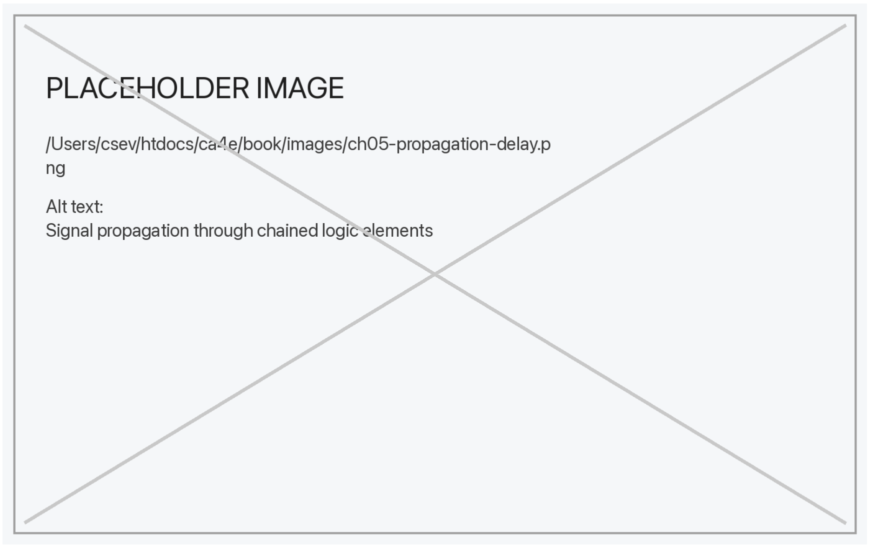

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch05-propagation-delay.png
Alt text:
Signal propagation through chained logic elements

Figure 26: Signal propagation through chained logic elements

- the size of the chip itself

Once the slowest element of the arithmetic and logic unit (ALU) is identified, the clock must be slow enough to accommodate that delay. Faster clocks allow more operations per second, but only if circuits can reliably settle within each cycle.

0.5.3 Separating Computation from Storage

To coordinate circuits, designers separate systems into two major categories:

- **combinational circuits**, which compute continuously
- **clocked storage elements**, which update only on clock events

Adders and logic gates belong to the first category. Latches and registers belong to the second.

During most of the clock cycle, combinational circuits compute based on stable stored inputs. At the clock edge, new values are captured into storage elements, and the next cycle of computation begins.

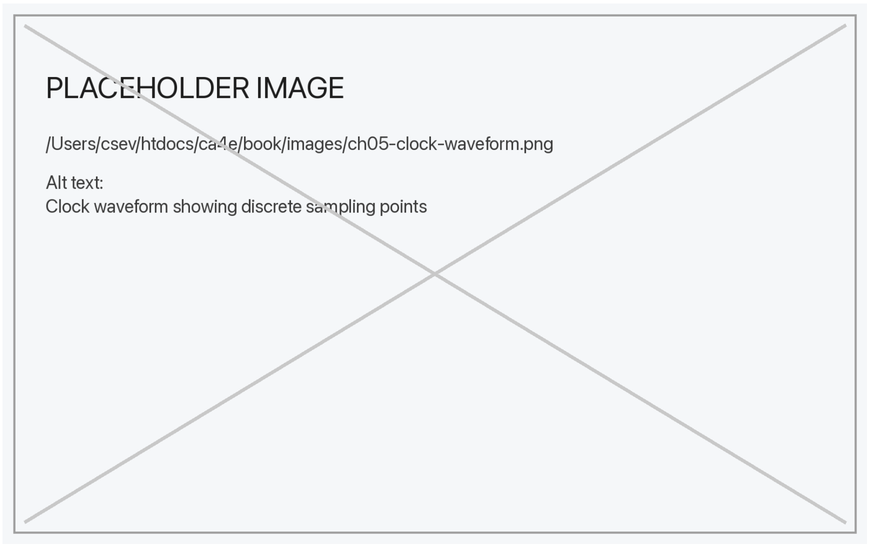

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch05-clock-waveform.png
Alt text:
Clock waveform showing discrete sampling points

Figure 27: Clock waveform showing discrete sampling points

This separation prevents unstable intermediate values from propagating into stored state.

0.5.4 Combining Adders, Registers, and Clocks

Consider a simple system that adds two numbers and stores the result:

- a register holds the current value
- an adder computes a new value
- a clock controls when the register updates

While the clock is low, the register holds its value and the adder continuously computes the sum based on that value. When the clock goes high, the computed sum is captured into the register. The next cycle begins with a new stable value.

This structure is repeated throughout processors to create step-by-step execution.

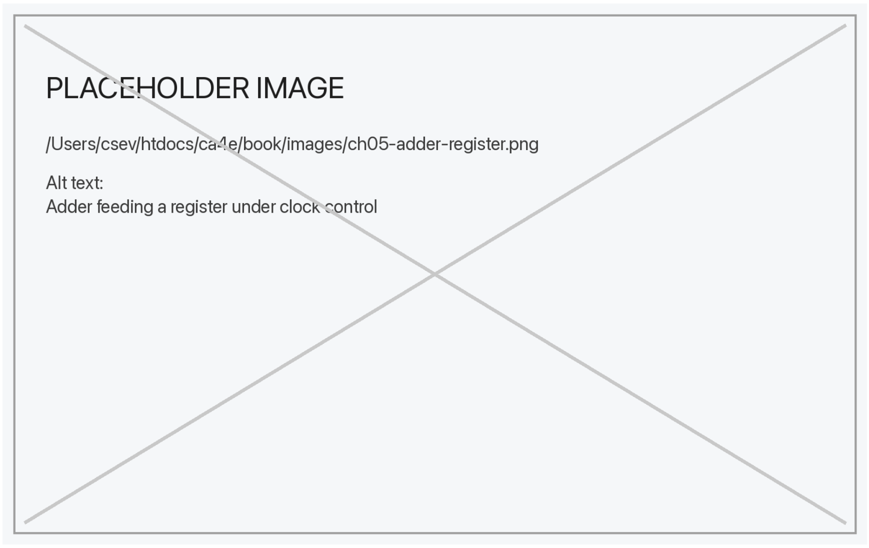

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch05-adder-register.png
Alt text:
Adder feeding a register under clock control

Figure 28: Adder feeding a register under clock control

0.5.5 Building a Counter

By feeding the output of a register back into one input of an adder and fixing the other input to the value one, a counting circuit can be created.

Each clock cycle:

1. the adder computes current value plus one

2. the register stores the new value

3. the process repeats

After reaching the maximum representable value, the counter overflows and wraps back to zero.

This simple structure forms the basis of timers, program counters, and many sequencing mechanisms inside computers.

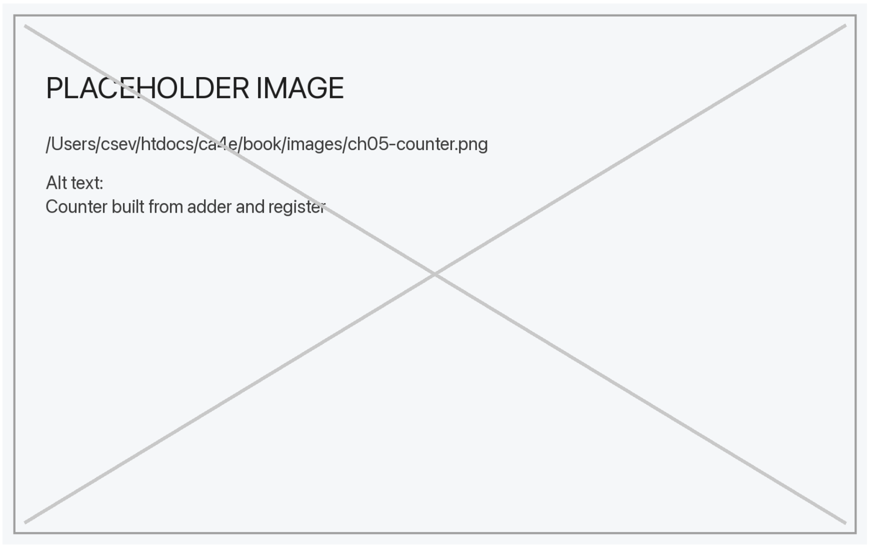

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch05-counter.png
Alt text:
Counter built from adder and register

Figure 29: Counter built from adder and register

0.5.6 From Hardware to Instructions

So far, all behavior has been determined by fixed wiring. To build programmable machines, behavior must be controlled by **instructions** rather than physical switches.

An instruction is a pattern of bits that specifies which operations should occur during a clock cycle. Instead of permanently connecting wires to force an action, instruction bits enable or disable parts of the circuit dynamically.

This is achieved through **control logic** that interprets instruction bits and routes signals accordingly.

0.5.7 A Two-Instruction CPU

A minimal processor can be built using:

- a small register
- an adder

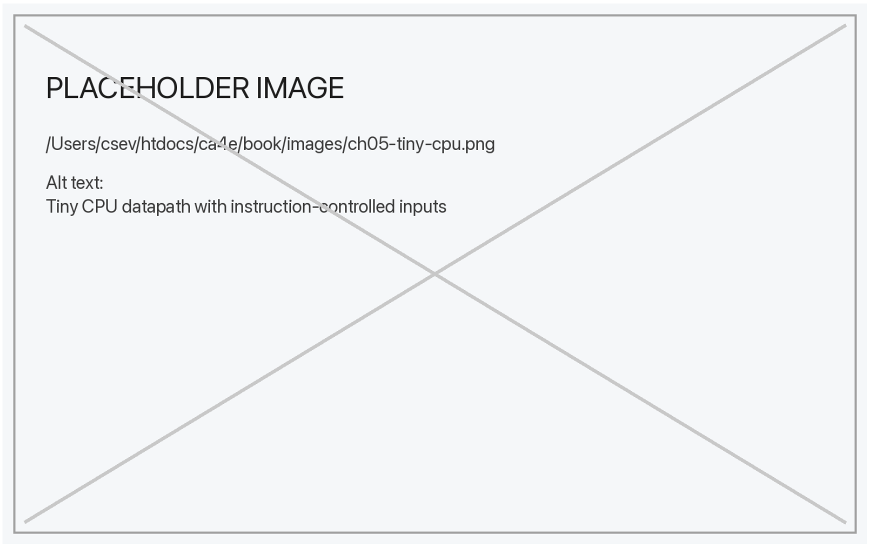

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch05-tiny-cpu.png
Alt text:
Tiny CPU datapath with instruction-controlled inputs

Figure 30: Tiny CPU datapath with instruction-controlled inputs

- control gates
- a clock

Suppose the machine supports two instructions:

- **0** — clear the register
- **1** — add one to the register

Instruction bits are connected to control gates that determine whether the adder output or zero is fed into the register. On each clock cycle, the selected value is stored.

Although extremely simple, this system demonstrates the core idea of programmable behavior: control signals modify data paths on each cycle.

0.5.8 Why Memory Is Required

Manual instruction selection is not sufficient for general computation. Real programs require sequences of many instructions executed automatically.

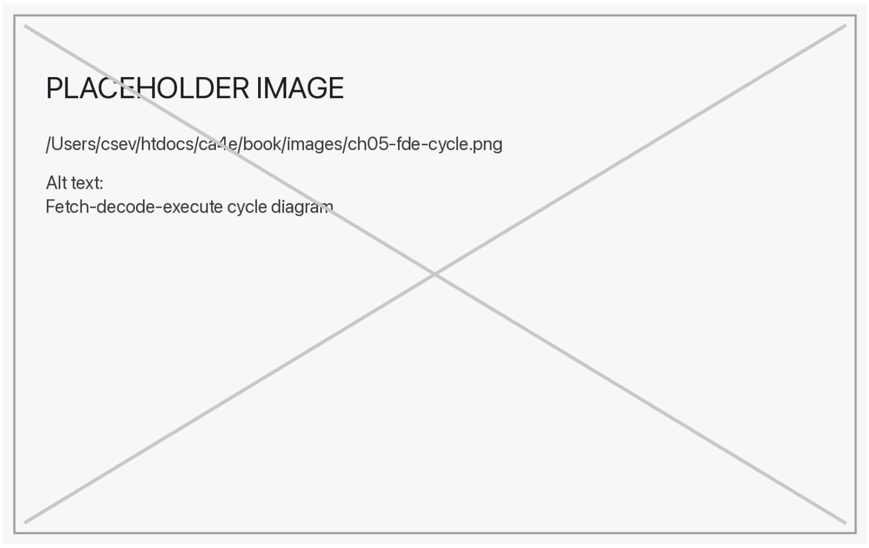

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch05-fde-cycle.png
Alt text:
Fetch-decode-execute cycle diagram

Figure 31: Fetch-decode-execute cycle diagram

To support this, computers store instructions in memory and retrieve them one by one during execution. This leads to the **fetch–decode–execute cycle**:

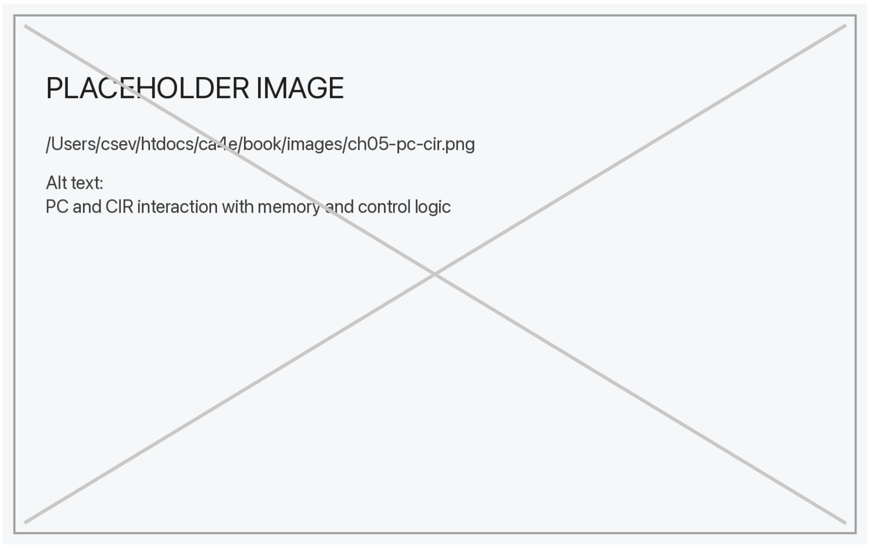
1. fetch instruction from memory
2. decode instruction bits into control signals
3. execute operation
4. repeat

This loop continues as long as the program runs.

0.5.9 Program Counter and Instruction Register

Two special registers manage instruction sequencing:

- the **Program Counter (PC)** stores the address of the next instruction


PLACEHOLDER IMAGE

/Users/csev/htdocs/ca4e/book/images/ch05-pc-cir.png

Alt text:
PC and CIR interaction with memory and control logic

Figure 32: PC and CIR interaction with memory and control logic

- the **Current Instruction Register (CIR)** holds the instruction being executed

On each cycle, the PC advances, memory is accessed, and the CIR loads the next instruction. Decoder circuits then examine the instruction bits and activate the appropriate control signals for the ALU and registers.

0.5.10 Decoding Instructions with Gates

Instruction decoding is performed using combinations of logic gates called **decoders**. A decoder converts bit patterns into individual control lines.

For example, if an instruction has three bits, a decoder can generate eight distinct control signals, one for each possible instruction value. These signals activate specific parts of the circuit for each instruction type.

This mechanism allows compact binary instructions to control large physical systems.

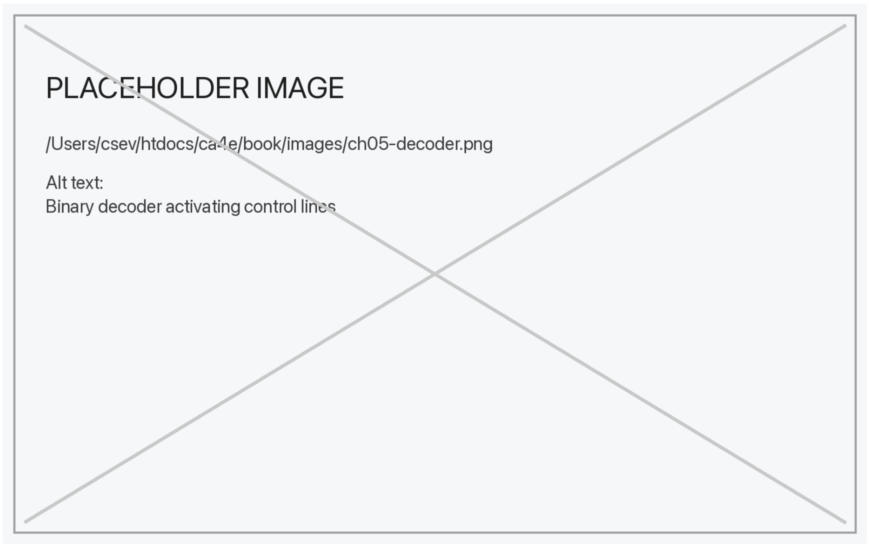

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch05-decoder.png
Alt text:
Binary decoder activating control lines

Figure 33: Binary decoder activating control lines

0.5.11 Abstraction Through Repetition

Although real processors contain billions of transistors, they are composed of repeated versions of the same fundamental structures:

- adders
- registers
- multiplexers
- decoders

Complex behavior emerges from organized combinations of simple components operating under synchronized timing.

This layered abstraction allows designers to reason about machines in terms of data paths and control flows rather than individual transistors.

0.5.12 Summary: Time Makes Programs Possible

Clocked circuits introduce controlled timing into digital systems, allowing stable computation to occur in discrete steps. By separating combinational logic from storage and coordinating updates with a clock, reliable large-scale systems become possible.

Control logic and instruction decoding transform fixed circuits into programmable machines. Registers, adders, and decoders cooperate to execute instruction sequences automatically.

With clocked execution in place, computers can now run programs rather than merely perform fixed calculations.

0.5.13 What Comes Next

The next chapter examines machine language and processor architecture in more detail, using a small but complete instruction set inspired by early microprocessors. Programs will be written directly in machine code and executed in an emulator to reveal how software and hardware meet at the lowest level.

0.6 CPU Architecture and Machine Code: How Programs Actually Run

Up to this point, digital systems have been built from logic gates, storage elements, and clocked circuits. These components make it possible to store values and perform arithmetic, but they do not yet explain how a general-purpose machine can follow a sequence of instructions. That capability emerges when computation is organized around a central processing unit (CPU) that repeatedly executes a simple control loop.

This chapter introduces a complete, working processor model and shows how machine instructions drive its behavior. Programs are examined not as abstract algorithms, but as concrete patterns of bits stored in memory that directly control hardware.

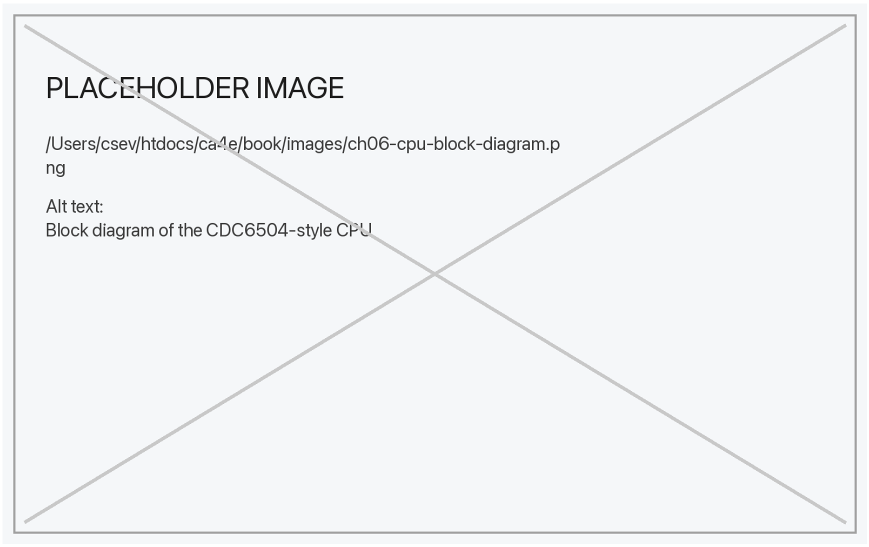

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch06-cpu-block-diagram.png
Alt text:
Block diagram of the CDC6504-style CPU

Figure 34: Block diagram of the CDC6504-style CPU

0.6.1 From Fixed Circuits to Programmable Machines

In earlier chapters, behavior was determined by wiring. If a circuit always adds two numbers or always increments a counter, then its function is fixed at design time. Programmable machines replace fixed control paths with instruction-controlled behavior.

Instead of hardwiring what happens each cycle, instruction bits select which operations occur and where data flows. The same physical hardware can perform many different tasks simply by changing the contents of memory.

This separation between hardware and program is the defining feature of general-purpose computers.

0.6.2 A Simple but Complete CPU

A minimal but realistic processor can be built from the following components:

- **Registers** that store temporary values

- a **Program Counter (PC)** that holds the address of the next instruction
- an **Instruction Register (IR)** that holds the current instruction
- an **Arithmetic Logic Unit (ALU)** that performs arithmetic and logic
- **Memory** that stores both instructions and data
- **Control logic** that interprets instruction bits

The CDC6504 emulator used in this book models a processor inspired by early microprocessors, with a small number of registers and a compact instruction set. Despite its simplicity, it contains all of the essential elements found in modern CPUs.

0.6.3 Registers: Fast, Small Storage

Registers are small storage locations located directly inside the CPU. They are much faster to access than memory and are used to hold intermediate results during computation.

Typical registers include:

- an **Accumulator (A)** for arithmetic results
- index registers such as **X** and **Y** for addressing and looping
- a **Status register** that holds condition flags

Flags record results of previous operations, such as whether a value was zero or whether an arithmetic overflow occurred. Later instructions can examine these flags to make decisions.

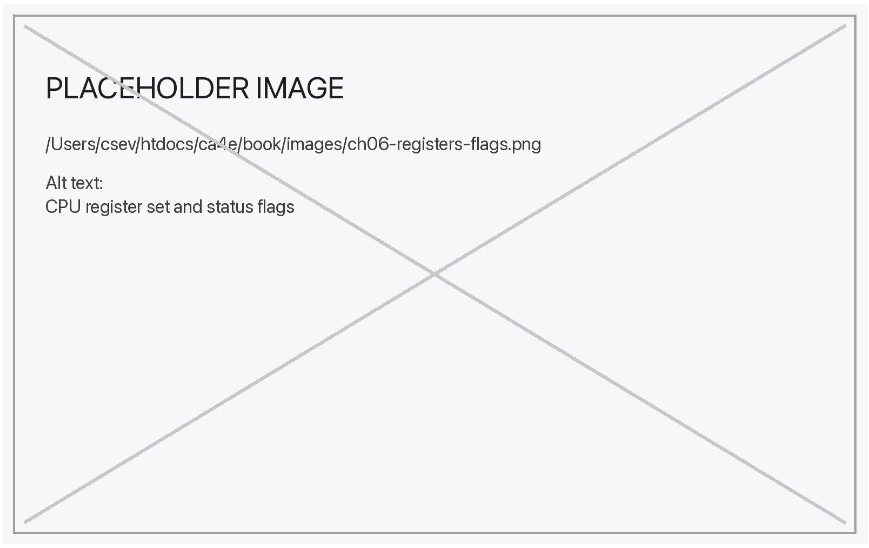

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch06-registers-flags.png
Alt text:
CPU register set and status flags

Figure 35: CPU register set and status flags

0.6.4 Memory: Where Programs and Data Live

Memory stores both instructions and data as sequences of bytes. Each byte is located at an address, and the Program Counter specifies which address should be read next.

Although modern computers often separate instruction and data memory internally, most early processors—and many simple designs—use a single memory space for both. In such systems, instructions are simply data that the CPU interprets in a special way.

This is why programs can be modified, copied, and even generated by other programs.

0.6.5 The Fetch–Decode–Execute Cycle

Every instruction executed by the CPU follows the same basic sequence:

1. **Fetch** the instruction from memory using the Program Counter

2. **Decode** the instruction to determine what operation to perform

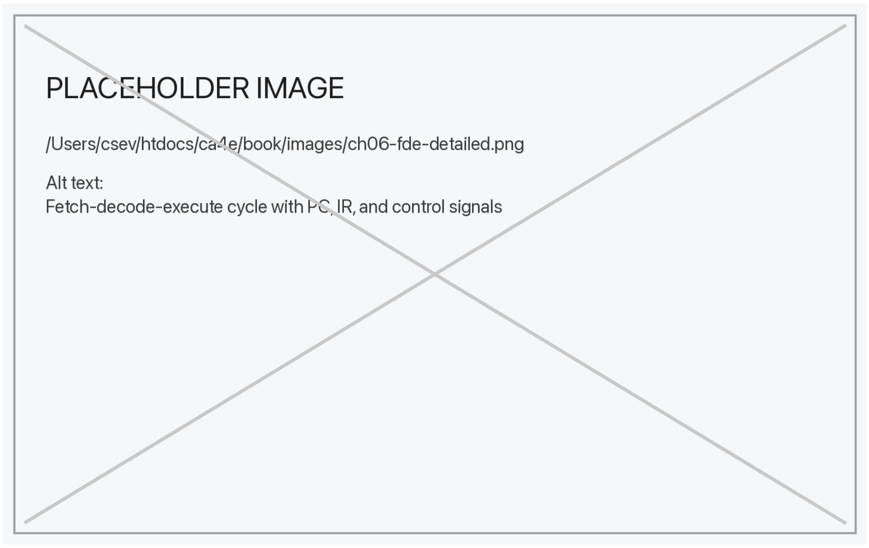

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch06-fde-detailed.png
Alt text:
Fetch-decode-execute cycle with PC, IR, and control signals

Figure 36: Fetch-decode-execute cycle with PC, IR, and control signals

3. **Execute** the operation using the ALU and registers
4. **Update** the Program Counter to the next instruction

This loop repeats continuously while the program runs.

Clock signals coordinate each stage so that values are stable when they are stored or used.

0.6.6 Why Hexadecimal Is Used

Machine instructions are sequences of bits, but long binary strings are difficult for humans to read. Hexadecimal notation groups bits into sets of four, making memory contents easier to inspect and write.

For example:

- binary: `1010 1111`
- hex: `AF`

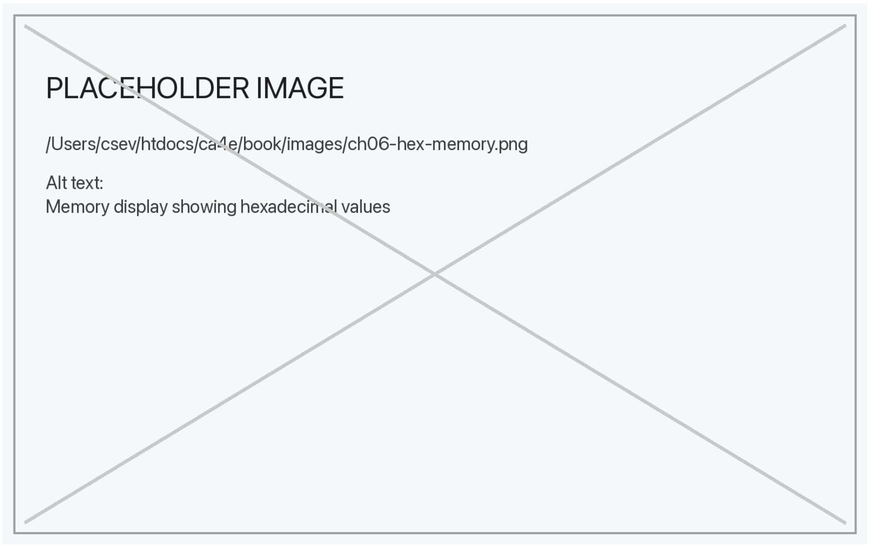

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch06-hex-memory.png
Alt text:
Memory display showing hexadecimal values

Figure 37: Memory display showing hexadecimal values

Each hexadecimal digit corresponds exactly to four binary bits. This mapping makes it convenient to display memory as rows of hex values while still representing precise machine data.

Assemblers, debuggers, and emulators commonly display memory using hexadecimal for this reason.

0.6.7 Instruction Formats and Operands

Each machine instruction contains:

- an **opcode** that specifies the operation

- zero or more **operands** that specify data or addresses

Some instructions operate directly on registers, while others reference memory. Common addressing modes include:

- **Immediate**: value is part of the instruction

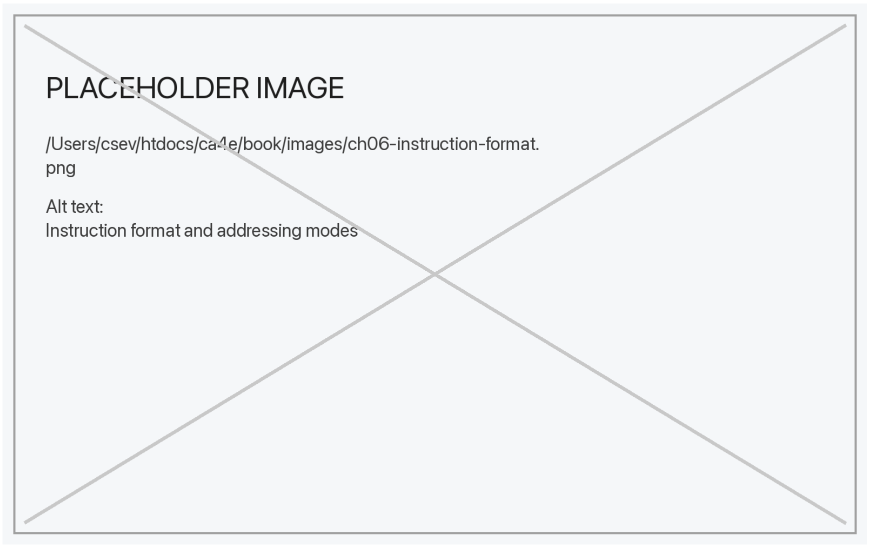

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch06-instruction-format.png
Alt text:
Instruction format and addressing modes

Figure 38: Instruction format and addressing modes

- **Direct**: instruction contains a memory address
- **Indexed**: address is computed using a register plus an offset

Different addressing modes allow programs to work with arrays, tables, and strings efficiently.

0.6.8 Control Logic and Decoding

Instruction decoding is performed by control logic that converts opcode bits into control signals. These signals determine:

- which registers receive data
- whether the ALU performs addition, subtraction, or comparison
- whether memory is read or written
- whether the Program Counter is modified

Conceptually, decoding is a large decision tree built from logic gates. In real processors, this logic is carefully optimized to minimize delay along critical paths.

Although decoding may appear complex, it is simply a systematic application of digital logic to route signals correctly.

0.6.9 Sequential Flow, Branches, and Loops

By default, the Program Counter advances to the next instruction after each cycle, causing instructions to execute sequentially. Branch instructions modify the Program Counter based on conditions stored in status flags.

This makes loops possible:

1. perform an operation

2. test a condition

3. branch back if the condition is not yet met

All higher-level control structures—such as while loops and if statements—ultimately reduce to conditional branches that alter the Program Counter.

0.6.10 Example: Counting with a Loop

Consider a simple loop that increments a register until it reaches a limit. In assembly language, this might look like:

```
LOAD A, #0
LOOP:
ADD  A, #1
CMP  A, #10
BNE  LOOP
```

Each line corresponds to one or more machine instructions. During execution, the Program Counter repeatedly jumps back to the label until the condition is satisfied.

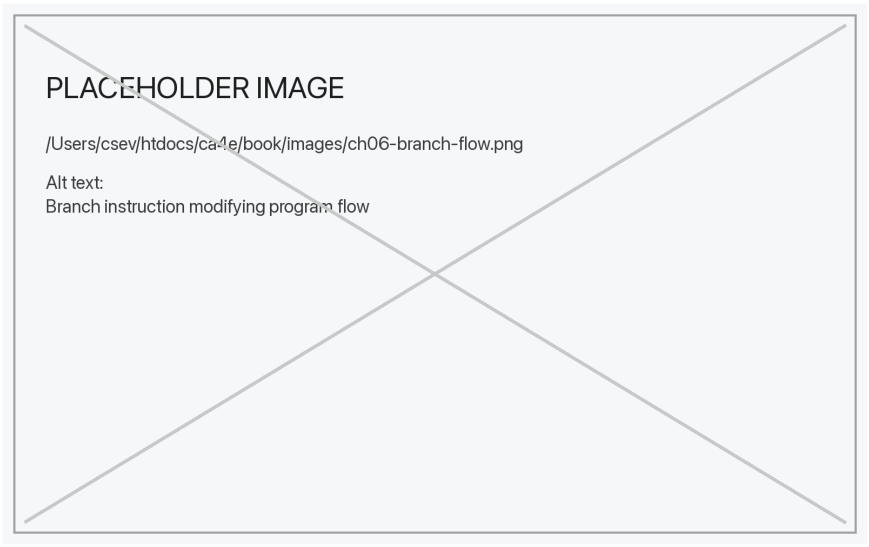

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch06-branch-flow.png
Alt text:
Branch instruction modifying program flow

Figure 39: Branch instruction modifying program flow

At the hardware level, this behavior is nothing more than controlled updates to registers and the Program Counter on each clock cycle.

0.6.11 Assembly Language as a Human Interface

Machine code is difficult to write directly. Assembly language provides symbolic names for instructions and allows labels to represent addresses. An **assembler** translates these symbolic programs into machine code.

Assembly language does not add new capabilities to the machine. It merely makes programs easier to write and understand.

For educational purposes, writing small programs in assembly reveals exactly how software controls hardware.

0.6.12 Strings, Characters, and Memory

Characters are stored as numeric codes, such as ASCII values. A string is simply a sequence of character codes stored in memory.

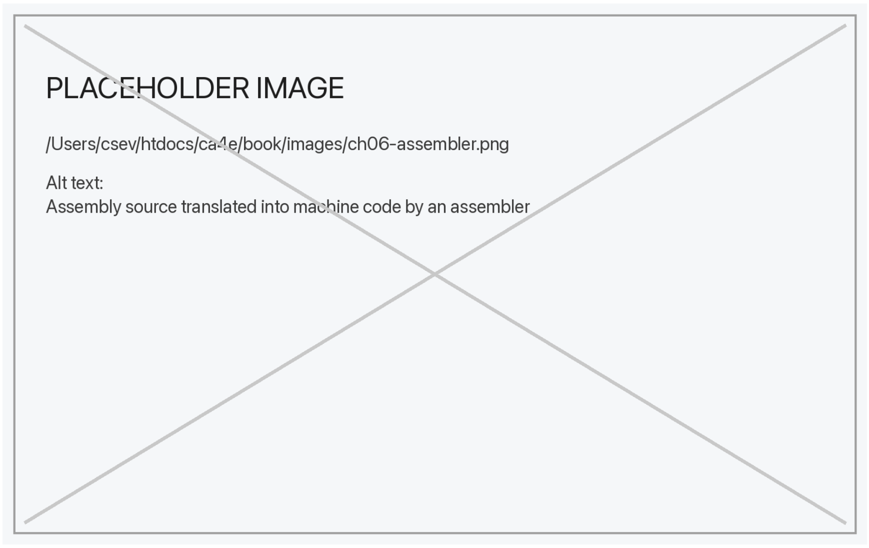

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch06-assembler.png
Alt text:
Assembly source translated into machine code by an assembler

Figure 40: Assembly source translated into machine code by an assembler

Programs that process text operate by:

- loading a character from memory
- testing or modifying it
- storing it back

Operations such as converting letters to uppercase are performed by arithmetic on character codes, not by any special text-handling hardware.

This illustrates that all data—numbers, characters, images—are treated uniformly as binary values by the processor.

0.6.13 From Python to Machine Instructions

High-level languages such as Python hide hardware details, but their execution still depends on machine instructions.

A loop written in Python becomes:

- comparisons
- branches
- register updates

at the machine level. Although modern systems add layers such as virtual machines and just-in-time compilation, the final execution always reduces to instructions executed by hardware.

Understanding machine code provides insight into performance, memory usage, and the real cost of software operations.

0.6.14 Why This Model Still Matters

Modern processors are far more complex than early microprocessors, with multiple cores, deep pipelines, and sophisticated memory systems. However, they still execute programs using the same fundamental principles:

- fetch instructions
- decode operations
- manipulate registers and memory
- update the Program Counter

The complexity lies in doing many of these steps in parallel and at extremely high speeds, not in changing the basic execution model.

Learning a small, complete CPU provides a foundation for understanding much larger systems.

0.6.15 Summary: Programs Are Physical Processes

Programs are not abstract mathematical objects running in isolation. They are physical processes enacted by electrical signals moving through circuits, coordinated by clocks, and shaped by control logic.

Machine instructions directly control data paths and storage elements. Assembly language offers a symbolic view of these instructions, while high-level languages build additional layers of abstraction on top.

By examining how a complete processor executes real programs, the relationship between hardware and software becomes concrete and observable.

0.6.16 What Comes Next

With a complete CPU model in place, attention can now turn to how real-world processors improve performance through techniques such as pipelining, caching, and parallel execution. The next chapter explores how architectural enhancements build on the same foundations while dramatically increasing speed.

0.7 WebAssembly and Emulation: Running Real Programs in the Browser

After building a complete mental model of how a CPU executes machine instructions, it becomes possible to recognize the same ideas appearing in unexpected places. One of the most surprising is the modern web browser. Today, browsers include built□in virtual machines capable of executing low□level binary code safely and efficiently.

This chapter explores how emulation and WebAssembly connect historical machine architectures to modern execution environments, and why the browser can now act as a practical platform for systems programming.

0.7.1 From Hardware to Emulators

An emulator is a program that imitates the behavior of a hardware processor. Instead of electrical signals moving through gates, software interprets instruction bytes and updates simulated registers and memory.

At a high level, an emulator performs the same steps as real hardware:

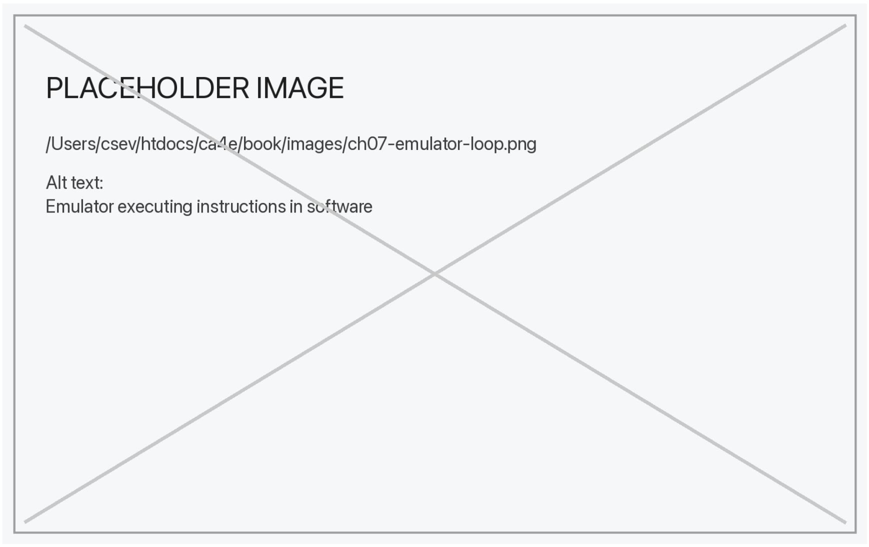

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch07-emulator-loop.png
Alt text:
Emulator executing instructions in software

Figure 41: Emulator executing instructions in software

1. read the next instruction
2. decode the opcode
3. perform the operation
4. update registers and memory
5. advance the program counter

This is the same fetch–decode–execute cycle implemented in software.

Because modern computers are extraordinarily fast, they can often emulate older machines faster than the original hardware ever ran.

0.7.2 Why Emulation Is Practical Today

Early microprocessors such as the MOS 6502 contained only a few thousand transistors and ran at clock speeds measured in megahertz. Modern processors contain billions of transistors and operate at several gigahertz.

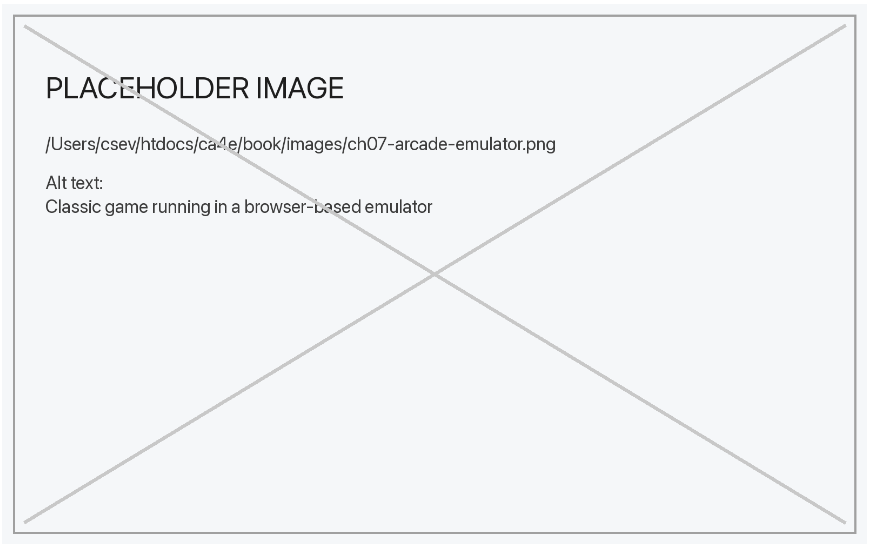

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch07-arcade-emulator.png
Alt text:
Classic game running in a browser-based emulator

Figure 42: Classic game running in a browser-based emulator

This enormous performance gap means that:

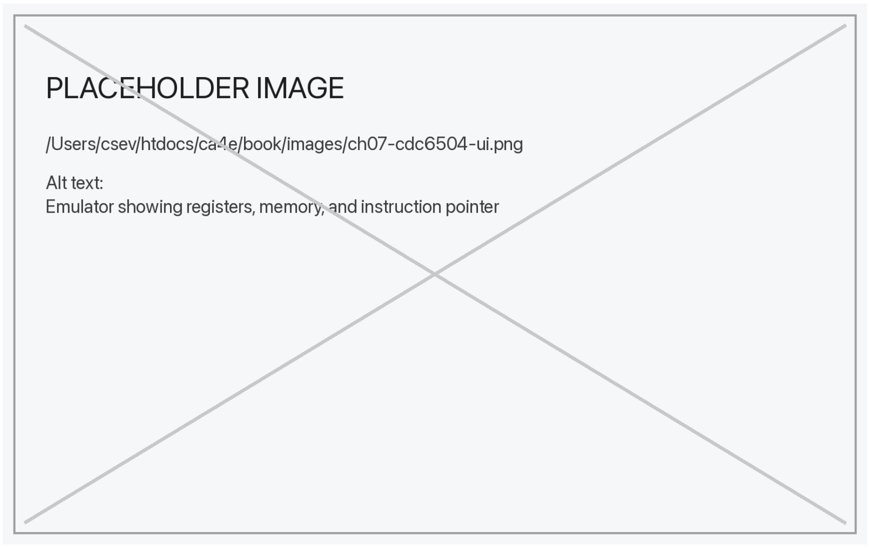
- JavaScript running in a browser can emulate historical CPUs in real time
- graphics and sound can be simulated accurately
- entire vintage game systems can be recreated in software

Web sites such as the Internet Archive host playable emulations of classic arcade machines that run entirely in the browser.

0.7.3 The CDC6504 Emulator

The CDC6504 emulator used in this course models a processor inspired by the MOS 6502 instruction set with a simplified architecture. It includes:

- registers (A, X, Y, status flags)


PLACEHOLDER IMAGE

/Users/csev/htdocs/ca4e/book/images/ch07-cdc6504-ui.png

Alt text:
Emulator showing registers, memory, and instruction pointer

Figure 43: Emulator showing registers, memory, and instruction pointer

- instruction memory
- data memory
- branching and arithmetic instructions

Each instruction is implemented as a small block of JavaScript that updates simulated hardware state.

Conceptually, each opcode becomes a function that performs the same register and memory updates that real circuitry would perform in silicon.

0.7.4 Machine Code Inside Software

When an emulator runs, machine code is not special. It is simply a sequence of numbers stored in an array. The emulator reads these numbers and interprets them as instructions.

For example:

- a value may represent “load accumulator”

- the next value may represent an address or constant
- branching instructions modify the program counter

The meaning comes entirely from how the emulator interprets the bits.

This mirrors exactly what real hardware does, except that software replaces physical wiring.

0.7.5 From Native CPUs to Virtual Machines

Historically, assembly language targeted specific processors:

- MOS 6502
- Intel x86
- ARM

Programs compiled for one architecture could not run on another without translation or emulation.

WebAssembly changes this model by defining a portable virtual instruction set that runs on top of browsers and other runtimes.

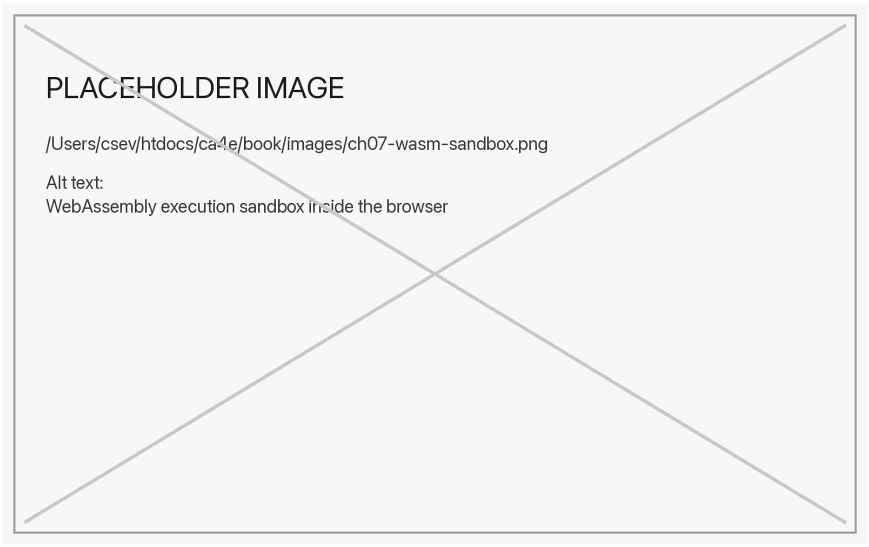
Instead of targeting physical hardware, compilers target a standardized virtual machine.

0.7.6 What Is WebAssembly?

WebAssembly (WASM) is a low□level, binary instruction format designed for safe and efficient execution in browsers and other environments. It is not tied to any specific physical CPU.

Key characteristics include:

- structured control flow
- validated instruction sequences


PLACEHOLDER IMAGE

/Users/csev/htdocs/ca4e/book/images/ch07-wasm-sandbox.png

Alt text:
WebAssembly execution sandbox inside the browser

Figure 44: WebAssembly execution sandbox inside the browser

- sandboxed memory access
- deterministic execution

WASM programs cannot access files, devices, or the operating system directly. All interaction with the outside world occurs through carefully controlled interfaces provided by the host environment.

0.7.7 From C to WASM

Languages such as C and C++ can be compiled to WebAssembly using modern toolchains. The compilation process is:

1. source code is compiled to WASM bytecode
2. the browser loads and validates the module
3. the runtime translates WASM to native machine code

4. the program executes at near□native speed

From the browser's perspective, WebAssembly is just another kind of executable content, similar to JavaScript but closer to machine operations.

0.7.8 A WASM "Hello, World"

A minimal WebAssembly program may look like this in textual form (WAT):

```
(module
  (import "console" "log" (func $log (param i32 i32)))
  (memory 1)
  (data (i32.const 0) "Hello, World!")
  (func $main (result i32)
    (call $log (i32.const 0) (i32.const 13))
    (i32.const 42)
  )
  (export "main" (func $main))
)
```

This code:

- allocates memory
- stores a string
- calls a logging function
- returns a numeric value

When compiled, it becomes a compact binary format that the browser can execute efficiently.

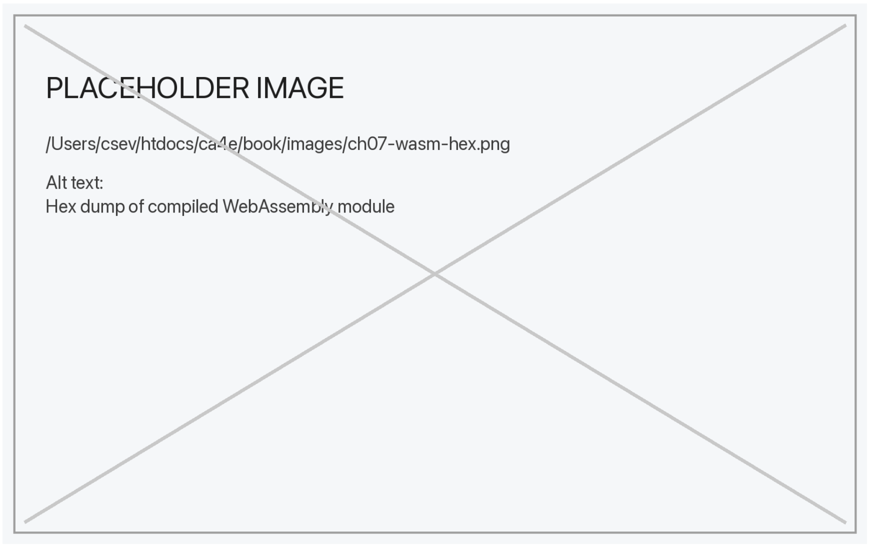

PLACEHOLDER IMAGE
/Users/csev/htdocs/ca4e/book/images/ch07-wasm-hex.png
Alt text:
Hex dump of compiled WebAssembly module

Figure 45: Hex dump of compiled WebAssembly module

0.7.9 WASM Compared to Traditional Assembly

Although WebAssembly resembles assembly language, it differs in important ways:

- **Sandboxed by design** — no direct memory or OS access
- **Safe execution model** — code is validated before running
- **Portable bytecode** — same program runs on any platform
- **Abstract machine** — targets a virtual stack machine
- **Host optimization** — browsers compile and optimize at runtime

Traditional assembly runs with full process privileges and depends entirely on the operating system for protection. WASM embeds safety into the execution model itself.

0.7.10 Loops, Functions, and the Stack

Like physical CPUs, WebAssembly supports:

- local variables
- function calls
- loops and branches
- a call stack

When a function is called, parameters and return addresses are managed using stack structures maintained by the runtime.

Even though the environment is virtual, the same concepts of control flow and memory organization still apply.

0.7.11 From Historical CPUs to Modern Devices

Over the last fifty years, many processor families have been developed, but two now dominate consumer computing:

- **ARM** processors, used in phones, tablets, and most laptops
- **x86** processors, used in many desktop and server systems

These architectures differ internally, but they all implement the same fundamental ideas explored in this book: registers, memory, instructions, and controlled execution.

0.7.12 Case Study: Apple's CPU Transitions

Apple has moved across several processor families:

- MOS 6502 — early Apple computers

- Motorola 68000 — early Macintosh systems
- PowerPC — mid□1990s through mid□2000s
- Intel x86 — 2006 through 2020
- Apple□designed ARM — 2020 to present

To ease transitions, Apple built machine□code translation systems that converted programs from one architecture to another at launch time. This allowed users to run older software while new native versions were developed.

These transitions demonstrate that software compatibility can be preserved even as hardware changes dramatically.

0.7.13 Why the 6502 Still Matters

The design philosophy of early microprocessors influenced later architectures. Engineers who designed early ARM processors had deep experience with the 6502, and many principles carried forward into modern instruction set design.

Although modern processors are vastly more complex, the core ideas remain recognizable.

Understanding a small historical CPU therefore provides insight into the design of modern systems.

0.7.14 Emulation as a Learning Tool

Emulators allow direct observation of how instructions change machine state:

- registers update
- memory changes

- branches alter execution flow

This visibility is rarely available on modern hardware, where pipelines and caches obscure internal behavior.

For learning computer architecture, emulation provides clarity that real machines no longer expose.

0.7.15 Summary: Abstraction Without Losing Reality

WebAssembly and emulation demonstrate that low□level computation remains relevant even in modern software systems. Programs still execute as sequences of instructions that manipulate memory and registers, whether those instructions are interpreted, emulated, or executed directly in silicon.

The same architectural principles govern both historical processors and modern virtual machines. Only the layers of abstraction have changed.

By tracing the path from physical circuits to browser□based execution, the continuity of computer architecture becomes clear.

0.7.16 Closing Thoughts

Computers have evolved enormously in speed and scale, but not in fundamental design. Logic gates became processors, processors became systems, and systems became virtual machines running inside browsers.

Understanding how computation works at the lowest level makes it possible to see through layers of abstraction and recognize the same mechanisms at work everywhere—from embedded devices to cloud servers to web pages running in a browser.

This perspective provides both practical insight and a deeper appreciation for the remarkable continuity of computing technology.