Computers Architecture

Islas Carreon Victor Jakxel

February 6, 2026

Contents

1 Foundations and Scope of Computer Architecture

To begin with, computer architecture is the discipline concerned with the conceptual design and fundamental operational structure of computer systems. It defines the attributes of a computing system that are visible to the programmer, as well as the principles that govern the interaction between hardware and software. As such, computer architecture serves as a bridge between abstract computational models and their physical realization.

This section talks about the conceptual scope of computer architecture, tries to clarify its distinction from related fields, and introduces the layered abstraction model that underpins modern computing systems. All of this is meant to be the first step into this article, before continuing into the real chaos and pain of the low-level world.

1.1 Definition of Computer Architecture vs Computer Organization vs Computer Engineering

For this discussion, there are three main concepts: computer architecture, computer organization, and computer engineering. They are closely related but distinct disciplines.

Computer architecture refers to the programmer-visible specification of a computer system. This includes the instruction set architecture (ISA), data types, registers, addressing modes, memory model, and input/output mechanisms. Architecture defines *what* a computer does and how software interacts with hardware, without prescribing a specific hardware implementation.

Computer organization describes the internal operational units and their interconnections that implement the architectural specification. This includes datapaths, control units, pipelines, cache structures, and memory hierarchies. Organization focuses on *how* architectural features are realized in hardware.

Computer engineering is a broader field that encompasses the design, development, and integration of computer systems at the electrical and physical level. It combines principles from electrical engineering and computer science, covering topics such as circuit design, semiconductor technology, embedded systems, and hardware–software co-design.

In summary, computer architecture defines system behavior and the programmer interface, computer organization implements that definition, and computer engineering provides the physical means to construct the system.

1.2 Role of Computer Architecture in the Hardware–Software Interface

Computer architecture defines the hardware–software contract. Software relies on architectural guarantees such as instruction semantics, memory consistency models, and exception-handling behavior. Hardware implementations must faithfully execute software according to these specifications.

This interface enables software portability across different implementations of the same architecture. For example, multiple microarchitectures may implement the same ISA. If you do not know what ISA means, it is simply the *Instruction Set Architecture*, which defines the set of instructions that a processor can execute. These implementations may differ significantly in performance, power consumption, and internal complexity. This separation of concerns is a fundamental principle that enables long-term software compatibility and ecosystem stability.

1.3 Historical Evolution of Computer Architectures

Early computer systems were based on the von Neumann architecture, characterized by a single memory shared between instructions and data. While simple and flexible, this design introduced the von Neumann bottleneck, limiting performance due to shared memory bandwidth.

Subsequent developments introduced Harvard and modified Harvard architectures, separating instruction and data paths to improve throughput. Advances in semiconductor technology enabled increasingly complex architectures, including pipelined processors, superscalar execution, and multicore systems.

Modern computer architectures emphasize parallelism, memory hierarchy optimization, and energy efficiency, making it possible for you to spend all day watching reels or funny cat videos.

1.4 Design Goals and Constraints

The design of a computer architecture involves balancing multiple competing objectives:

- **Performance**: maximizing instruction throughput and minimizing latency
- **Power efficiency**: reducing energy consumption and heat dissipation
- **Cost**: minimizing manufacturing and deployment expenses
- **Reliability**: ensuring correct operation under faults and variability
- **Scalability**: supporting future performance growth

Architectural decisions inevitably involve trade-offs among these factors, and no single design can optimize all objectives simultaneously.

1.5 Abstraction Layers in Computing Systems

Modern computing systems are organized as a hierarchy of abstraction layers. Each layer conceals lower-level complexity while exposing well-defined interfaces, enabling modular design, portability, and scalability across hardware and software platforms.

- **High-Level Software Abstraction**
  At the highest level, users interact with applications developed using high-level programming languages such as Python, Java, or C++. These languages provide abstractions for data structures, control flow, memory management, and concurrency. Compilers and interpreters translate high-level code into lower-level representations, while operating systems and runtime environments manage resources such as processes, memory, files, and I/O devices. This layer prioritizes productivity, portability, and maintainability over direct hardware control—basically the thing that keeps you awake during final exams in college.

- **Assembly and Machine-Level Abstraction**
  Below high-level languages lies the final boss, the chad section, where real men and women try to understand the instruction set architecture (ISA), which defines the interface between software and hardware. Assembly language offers a symbolic, somewhat human-readable representation of machine instructions, registers, and addressing modes. Machine language encodes these instructions as binary values executed directly by the processor. This layer exposes hardware capabilities more explicitly, enabling fine-grained control over performance, memory usage, and hardware features.

- **Microarchitectural Abstraction**
  The microarchitecture specifies how the processor implements the ISA internally. It includes components such as instruction pipelines, execution units, caches, branch predictors, and memory hierarchies. Although typically invisible to software, microarchitectural design choices significantly influence performance, energy efficiency, and latency. Techniques such as out-of-order execution, speculative execution, and parallelism are implemented at this level.

- **Digital Logic and Physical Implementation**
  At the lowest level, digital logic circuits constructed from transistors realize the functional units of the processor, including registers, arithmetic logic units, and control circuitry. This layer is governed by physical constraints such as clock frequency, signal propagation delay, power consumption, heat dissipation, and fabrication technology. The physical implementation ultimately determines the limits of performance, reliability, and scalability.

Together, these abstraction layers allow complex computing systems to be designed and reasoned about systematically, balancing usability, performance, and physical feasibility.

2 Binary Representation and Digital Foundations

All modern computer systems are built upon digital logic and binary representation. This foundation arises from the physical characteristics of electronic components, such as transistors, which naturally support two stable operating states corresponding to logical values. Binary abstraction enables reliable computation even in the presence of noise, efficient storage through discrete encoding, and systematic hardware design using formal mathematical models. This section introduces how data and computation are represented at the lowest logical level of computer architecture, and how these representations connect physical hardware to higher-level software abstractions.

2.1 Number Systems and Data Representation

Computers represent all forms of data numerically using positional number systems. While humans commonly use decimal notation, computers rely primarily on the binary system due to its direct correspondence with electronic states and its suitability for digital circuit implementation. Understanding these number systems is essential for interpreting memory contents, instruction encoding, and low-level program behavior.

2.1.1 Binary, Octal, and Hexadecimal Systems

The binary system uses base-2 notation, employing only the digits 0 and 1—yes, your entire digital life is ultimately based on just these two values. These digits map directly to low and high voltage levels in digital circuits. Binary representations are fundamental to arithmetic operations, data storage, and instruction execution. However, long sequences of binary digits can be difficult for humans to read and work with.

To improve readability and compactness, binary values are often expressed using octal (base-8) or hexadecimal (base-16) representations. Each octal digit corresponds to three binary bits, while each hexadecimal digit corresponds to four bits, allowing direct and lossless conversion. Hexadecimal notation is especially common in low-level programming, debugging tools, memory dumps, and system documentation due to its balance between compactness and clarity.

2.1.2 Signed and Unsigned Integer Representation

Unsigned integers represent non-negative values using straightforward binary encoding, allowing the full range of representable values to be interpreted as magnitudes. Signed integers, by contrast, must represent both positive and negative values within a fixed number of bits.

Most modern systems use two's complement representation for signed integers. In this scheme, the most significant bit indicates the sign, and negative values are represented by inverting the bits of the corresponding positive value and adding one. Two's complement simplifies hardware design by allowing addition and subtraction to be performed using the same circuitry, eliminating the need for separate sign-handling logic. For this reason, it has become the de facto standard across contemporary processor architectures.

2.1.3 Floating-Point Representation

Real numbers are commonly represented using floating-point formats defined by the ever-present and highly acclaimed IEEE 754 standard. A floating-point value consists of three fields: a sign bit, an exponent that determines the scale, and a significand (or mantissa) that encodes precision. This representation allows computers to efficiently approximate a wide range of real values, from very small fractions to extremely large magnitudes.

Despite its flexibility, floating-point arithmetic is inherently approximate. Rounding errors, limited precision, and special values such as NaN (Not a Number) and infinity can lead to unintuitive behavior in numerical computations. These limitations must be carefully considered in scientific computing, graphics, and financial applications, where numerical accuracy and stability are especially important.

2.2 Character and Symbol Encoding

Beyond numeric data—or just 0s and 1s—computers must also represent textual and symbolic information in a standardized and interoperable manner. Character encoding schemes define how characters are mapped to numeric values stored in memory.

2.3 ASCII Character Encoding

The American Standard Code for Information Interchange (ASCII) is one of the earliest and most influential character encoding standards in computing. Developed in the 1960s, ASCII was designed to provide a uniform method for representing textual data in computers and communication systems.

ASCII defines a mapping between numeric values and characters using a 7-bit encoding scheme, allowing for 128 distinct symbols. These include uppercase and lowercase letters, decimal digits, punctuation marks, whitespace characters, and control codes such as carriage return, line feed, and tab. An extended 8-bit variant was later adopted by various systems, enabling up to 256 symbols, although this extension was never standardized globally.

The primary objective of ASCII was interoperability. By standardizing character representation, ASCII enabled reliable data exchange between heterogeneous systems, terminals, and peripheral devices. Its simplicity and fixed-width encoding made it well suited for early hardware constraints and low-bandwidth communication channels.

Despite its limited character set, ASCII remains foundational in modern computing. Many programming languages, file formats, network protocols, and operating system interfaces assume ASCII compatibility. Even modern encodings such as UTF-8 are explicitly designed to preserve ASCII values for the first 128 code points, ensuring backward compatibility.

However, ASCII is inherently limited in scope. It cannot represent accented characters, non-Latin alphabets, or many symbolic scripts, making it unsuitable for internationalized applications. These limitations motivated the development of more expressive encoding standards.

2.4 Unicode and UTF Character Encodings

Unicode is a universal character encoding standard designed to represent the characters of virtually all written languages, along with symbols, technical notation, and pictographic characters. Its development was motivated by the need to overcome the fragmentation and incompatibility of earlier encoding schemes.

Unlike ASCII, Unicode does not prescribe a single fixed-width encoding. Instead, it defines a comprehensive set of abstract code points, each corresponding to a character. These code points are organized into planes, with the Basic Multilingual Plane (BMP) containing the most commonly used characters.

The primary objective of Unicode is universality. By providing a single, consistent character set, Unicode enables text to be represented, processed, and exchanged correctly across different systems, platforms, and languages. This design supports globalization, localization, and international software development.

Unicode code points are encoded into bytes using transformation formats known as UTF (Unicode Transformation Format). UTF-8 uses a variable-length encoding ranging from one to four bytes and is backward compatible with ASCII. UTF-16 encodes characters using one or two 16-bit units, while UTF-32 uses a fixed 32-bit representation for each character.

UTF-8 has become the dominant encoding for modern software systems, particularly on the web, due to its efficiency for ASCII-based text, byte-level compatibility, and lack of endianness issues. UTF-16 is commonly used in certain programming environments and operating systems, while UTF-32 is mainly used in internal processing where fixed-width representation simplifies indexing.

Despite its expressive power, Unicode introduces additional complexity. Variable-length encodings complicate string manipulation, memory usage, and performance considerations. Additionally, Unicode defines normalization forms, combining characters, and bidirectional text rules, all of which must be handled carefully to ensure correct text processing.

Overall, Unicode and its UTF encodings provide the foundation for modern, multilingual computing, enabling consistent text representation in a globally connected digital environment.

2.5 Boolean Algebra and Logic Design

Boolean algebra provides the mathematical foundation for digital circuit design by describing logical relationships using binary variables and operations. Logical operators such as AND, OR, NOT, XOR, and NAND correspond directly to physical gate implementations in hardware. Boolean algebra enables formal reasoning about circuit behavior, optimization, and correctness.

Logic Gates and Truth Tables

Logic gates implement Boolean functions using combinations of transistors arranged to produce specific input–output relationships. Truth tables formally describe the behavior of these gates by enumerating all possible input combinations and their corresponding outputs. They serve as a fundamental tool for circuit analysis, verification, and debugging.

Combinational Circuits

Combinational circuits produce outputs solely as a function of their current inputs, with no dependence on past states. Examples include adders, multiplexers, encoders, decoders, and comparators. These circuits form the computational core of arithmetic and logic units (ALUs) and are essential for performing data-processing operations within a processor.

Sequential Circuits and State Machines

Sequential circuits incorporate memory elements such as latches and flip-flops, enabling the system to retain state information across clock cycles. Finite state machines (FSMs) are a common abstraction used to model control logic, specifying how a system transitions between states in response to inputs and clock events. Sequential logic is central to the implementation of registers, counters, pipelines, and processor control units.

2.6 Clocks, Timing, and Synchronization

Clock signals coordinate the operation of synchronous digital systems by defining discrete time steps for state updates. Clock frequency determines the maximum rate at which a system can operate, while factors such as clock skew, jitter, and propagation delay introduce timing constraints. Proper timing analysis and synchronization are essential to ensure correct operation, prevent race conditions, and maintain reliability as system complexity and operating frequencies increase.

3 Computer Hardware Components

Computer hardware refers to the physical parts of a computer system, such as the central processing unit (CPU), random-access memory (RAM), motherboard, data storage devices, graphics card, sound card, and the computer case itself. It also includes external or peripheral devices such as the monitor, mouse, keyboard, speakers, and other input/output devices.

By contrast, software consists of written instructions that can be stored and executed by hardware. Hardware derives its name from the fact that it is physically rigid and relatively difficult to change, whereas software is considered "soft" because it can be modified, updated, or replaced easily without altering the physical system.

Hardware is typically directed by software to execute commands and instructions. A combination of hardware and software forms a usable computing system, although some specialized systems may operate with minimal or fixed software.

3.1 Computer Fundamental Principle

John von Neumann, while working on the ENIAC project at the University of Pennsylvania, proposed the von Neumann architecture, which has served as the foundation for most modern computer systems. This architecture is based on a simple but powerful idea: both program instructions and data are stored in the same memory.

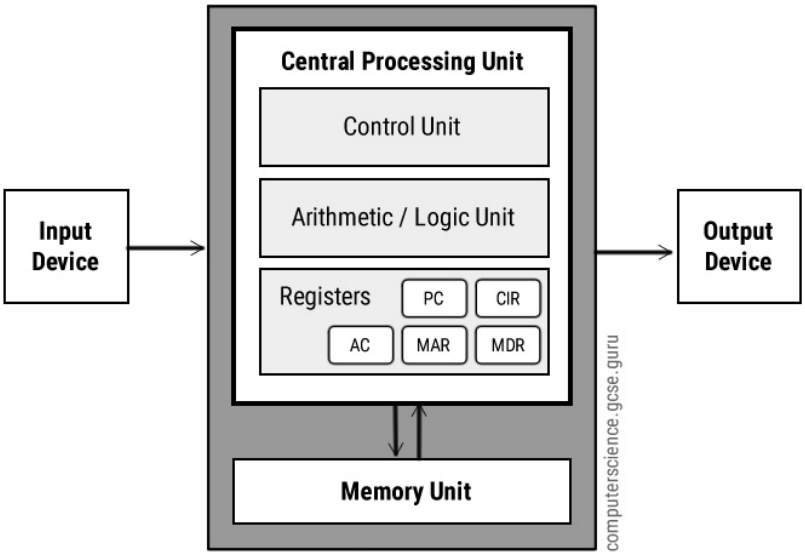

Figure 1: Von Neumann Architecture

The von Neumann architecture is composed of four main components:

- **Memory Unit**, which stores both data and program instructions
- **Central Processing Unit (CPU)**, which executes instructions
- **Input devices**, which provide data and commands to the system
- **Output devices**, which present results to the user

The CPU itself is typically divided into two main parts: the control unit, which directs the operation of the system, and the arithmetic logic unit (ALU), which performs calculations and logical operations.

3.1.1 How the von Neumann Architecture Works

At a high level, the von Neumann model works like a loop:

First, a program and its data are loaded into memory. The CPU then repeatedly performs a sequence known as the *fetch–decode–execute cycle*. During the **fetch** step, the CPU retrieves the next instruction from memory. In the **decode** step, the control unit interprets what the instruction means. Finally, in the **execute** step, the CPU performs the required operation, which may involve arithmetic, data movement, or input/output.

All communication between the CPU, memory, and I/O devices occurs over a shared set of buses. Because instructions and data use the same memory and bus, the CPU cannot fetch an instruction and access data at the same time. This limitation is known as the **von Neumann bottleneck**, and it can restrict overall system performance, especially in data-intensive workloads.

Despite this limitation, the simplicity and flexibility of the von Neumann architecture have made it extremely influential. Many modern systems are still based on this model, often with enhancements such as caches, pipelines, and separate instruction and data paths to reduce the impact of the bottleneck.

3.2 Central Processing Unit (CPU)

The Central Processing Unit (CPU) is often described as the brain of a computer. It is the component responsible for carrying out most of the thinking, calculating, and decision-making that allows a computer to function. Whether you are playing a game, typing a school assignment, or watching a video, the CPU is constantly processing instructions to make everything work smoothly.

The CPU is usually placed in a special slot called a *socket* on the computer's motherboard, which acts as the main circuit board connecting all computer components. The CPU is responsible for tasks such as:

- Performing mathematical calculations (such as addition and multiplication)
- Running applications and games
- Handling input/output (I/O) operations by communicating with memory and peripheral devices
- Temporarily storing and retrieving data needed during processing

Main Components of the CPU

The main components of the CPU are clearly illustrated again in the von Neumann architecture. These components include the Arithmetic Logic Unit (ALU), the Control Unit (CU), and registers.

- **Control Unit (CU):** The control unit manages and coordinates the operation of the CPU. It sends control signals such as clock, reset, and control instructions to other components. The CU ensures that instructions are executed in the correct order and that data moves properly between memory, registers, and the ALU.
- **Arithmetic Logic Unit (ALU):** The ALU performs all arithmetic operations (addition, subtraction, multiplication, and division) and logical operations (AND, OR, NOT, comparisons). Internally, many complex operations are reduced to simpler ones, such as using addition as the basis for multiplication.

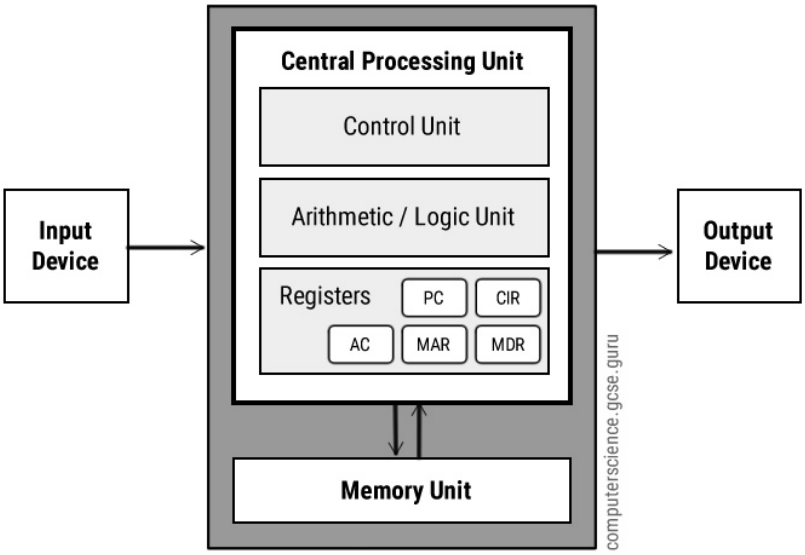

Figure 2: Von Neumann Architecture

- **Registers and Memory Unit:** Registers are very small and extremely fast memory locations inside the CPU used to store data and instructions temporarily. Modern CPUs also include cache memory, which is faster than main memory (RAM). During execution, the CPU fetches data from RAM or storage and places it into registers or cache for quick access.

Functions of the CPU

The CPU operates using a continuous process known as the *instruction cycle*, which consists of four main steps:

- **Fetch:** The CPU retrieves the next instruction from main memory (RAM).
- **Decode:** The control unit interprets the instruction and determines what actions are required.
- **Execute:** The CPU performs the operation, such as a calculation, data transfer, or logical comparison.
- **Store:** The result of the execution is stored back in memory or in a register for later use.

Types of CPUs

CPUs can be classified in several ways, depending on their design and purpose:

- **Single-core CPUs:** Contain one processing unit and can handle one task at a time.
- **Multi-core CPUs:** Include two or more cores, allowing multiple tasks to be processed simultaneously.
- **General-purpose CPUs:** Found in personal computers and laptops, designed to handle a wide variety of tasks.
- **Special-purpose CPUs:** Used in embedded systems, servers, or mobile devices, optimized for specific workloads.

How Does the CPU Make a Computer Faster?

A CPU makes a computer faster through several key factors. Higher clock speeds allow the CPU to execute more instructions per second. Multiple cores enable parallel processing, allowing several tasks to run at the same time. Cache memory reduces the time needed to access frequently used data, and advanced techniques such as pipelining and out-of-order execution improve efficiency by keeping the CPU busy instead of waiting for data.

Together, these features allow modern CPUs to perform billions of operations per second, making computers faster, more responsive, and capable of handling complex applications.

3.3 Primary Storage

Primary storage, also known as *main memory*, refers to the memory that the CPU can access directly and at very high speed while the computer is operating. It temporarily holds the data and instructions that are actively being used or processed. Because the CPU can only execute instructions that are already in primary storage, this type of memory is essential for real-time computation.

Most primary storage is volatile, meaning its contents are lost when power is removed. This makes it unsuitable for long-term storage but ideal for fast, short-term data access during program execution.

Importance of Primary Storage

Primary storage plays a critical role in system performance. The CPU operates much faster than secondary storage devices, so frequently used data must be kept close to the processor. If a system lacks sufficient primary storage, it may experience slowdowns, frequent program reloads, or increased reliance on virtual memory.

In modern systems, efficient use of primary storage allows multitasking, faster application response times, and smoother overall system operation.

Types of Primary Storage

- **RAM (Random Access Memory):**
  RAM is the primary working memory of the computer and plays a central role in system performance. It temporarily stores operating system components, running applications, and active data that the CPU needs immediate access to. RAM allows both read and write operations and provides uniform access time to any memory location, regardless of where the data is stored.

  One of the most important properties of RAM is its speed, which is significantly faster than secondary storage devices. However, RAM is volatile, meaning all stored data is lost when the computer is powered off. The capacity of RAM directly affects multitasking capabilities: systems with more RAM can run more applications simultaneously and handle larger datasets without slowing down. Modern RAM technologies, such as DDR4 and DDR5, improve bandwidth, reduce latency, and increase energy efficiency.

- **ROM (Read-Only Memory):**
  ROM stores essential system instructions required to initialize and start the computer, commonly known as firmware. These instructions include the boot process, hardware initialization routines, and low-level system configuration. Unlike RAM, ROM is non-volatile, so its contents remain intact even when the system is turned off.

  Traditionally, ROM could not be modified after manufacturing. However, modern systems use programmable variants such as EEPROM and flash-based ROM, allowing firmware updates to fix bugs, improve compatibility, or enhance security. ROM is optimized for reliability rather than speed, ensuring that critical startup instructions are always available and protected from accidental modification.

- **Cache Memory:**
  Cache memory is a small, high-speed memory located inside or very close to the CPU. Its primary function is to reduce the time required to access frequently used data and instructions. By keeping commonly accessed information close to the processor, cache memory significantly improves execution speed and overall system responsiveness.

  Cache memory is organized into multiple levels: L1 cache is the smallest and fastest, L2 cache is larger and slightly slower, and L3 cache is shared among CPU cores and balances capacity with speed. Cache memory operates transparently to software, managed automatically by the hardware. Although cache memory is expensive and limited in size, it is one of the most critical components for achieving high CPU performance.

Advantages and Limitations of Primary Storage

Primary storage offers very fast access speeds and direct communication with the CPU, but it is limited in capacity and relatively expensive compared to secondary storage. Additionally, its volatile nature means it cannot be used for permanent data storage.

Despite being tiny in size compared to secondary storage, cache memory can dramatically improve performance. A few megabytes of cache can save millions of slow memory accesses per second, making it one of the most valuable components inside a CPU.

—

3.4 Secondary Storage Devices

Secondary storage devices are used to store data and programs permanently. Unlike primary storage, secondary storage is non-volatile, meaning that information remains available even after the computer is turned off. These devices are designed for long-term data retention rather than high-speed access.

Secondary storage holds operating systems, applications, personal files, multimedia content, and system backups. When a program is executed, it is transferred from secondary storage into primary storage so the CPU can process it efficiently.

Importance of Secondary Storage

Secondary storage is essential for preserving information over time. Without it, computers would lose all data whenever they shut down. It also provides massive storage capacity at a relatively low cost, making it suitable for storing large datasets, software libraries, and user-generated content.

Secondary storage also plays a key role in system recovery, backups, and data sharing across devices.

Types of Secondary Storage Devices

- **Hard Disk Drives (HDDs):**
  HDDs store data on spinning magnetic platters. They offer high storage capacity at a low cost per gigabyte but are slower than modern alternatives and more vulnerable to physical damage due to their moving parts.

- **Solid State Drives (SSDs):**
  SSDs use flash memory to store data and have no moving parts. This results in much faster read and write speeds, lower power consumption, and increased durability. SSDs significantly improve boot times and application loading speeds.

- **Optical Storage Devices:**
  Optical media such as CDs, DVDs, and Blu-ray discs store data using laser technology. While their use has declined, they are still employed for software distribution, media playback, and archival storage.

- **Flash Storage Devices:**
  USB flash drives and memory cards provide portable and convenient storage. They are widely used for file transfer, backups, and temporary data storage.

- **Network and Cloud Storage:**
  Modern systems increasingly rely on network-attached storage (NAS) and cloud services. These solutions allow data to be accessed remotely, shared across devices, and protected through redundancy and backups.

Advantages and Limitations of Secondary Storage

Secondary storage offers large capacity, persistence, and affordability, but it is significantly slower than primary storage. Even the fastest SSDs cannot match the speed of RAM or cache memory, which is why data must be loaded into primary storage before processing.

Even with modern SSDs capable of reading gigabytes per second, accessing data from secondary storage is still thousands of times slower than accessing CPU registers. This is why operating systems rely heavily on caching and memory hierarchies to hide storage latency.

3.5 Input and Output Devices

Input and output (I/O) devices allow users and external systems to communicate with a computer. Input devices provide data and control signals to the system, while output devices present processed information in a human-readable or machine-readable form.

Input Devices

Input devices enable users to interact with the computer by entering data or commands. Common examples include:

- Keyboard and mouse
- Touchscreens and touchpads
- Microphones and webcams
- Scanners and game controllers

Output Devices

Output devices display or transmit the results of computation. Typical output devices include:

- Monitors and displays
- Printers
- Speakers and headphones
- Projectors

Some devices, such as touchscreens and network interfaces, function as both input and output devices. Efficient I/O handling is essential for system responsiveness and user experience.

3.6 Motherboards, Chipsets, and Interconnects

The motherboard is the main circuit board of a computer system. It physically connects and electrically links all major components, including the CPU, memory, storage devices, and peripheral interfaces. The motherboard provides the communication pathways that allow components to work together as a unified system.

Chipsets

The chipset is a collection of integrated circuits on the motherboard that manage data flow between the CPU, memory, storage, and peripheral devices. Traditionally, chipsets were divided into a northbridge and southbridge, but modern systems integrate many of these functions directly into the CPU.

Chipsets determine important system capabilities such as supported memory types, expansion slots, storage interfaces, and I/O ports.

Interconnects

Interconnects are the buses and links that carry data, addresses, and control signals between components. Examples include:

- PCI Express (PCIe) for graphics cards and high-speed devices
- Memory buses connecting the CPU to RAM
- SATA and NVMe interfaces for storage devices

High-speed and efficient interconnects are critical for system performance, especially in modern multi-core and data-intensive systems.

3.7 Power Delivery and Thermal Management

Power delivery and thermal management ensure that computer components receive stable electrical power and operate within safe temperature limits. Without proper management, systems can become unstable, inefficient, or even permanently damaged.

Power Delivery

The power supply unit (PSU) converts electrical power from an outlet into usable voltages for internal components. It must provide consistent and reliable power to the CPU, GPU, memory, storage, and peripherals. Modern systems also use voltage regulators on the motherboard to precisely control power delivery to sensitive components.

Thermal Management

As components operate, they generate heat. Thermal management systems are designed to remove this heat and maintain safe operating temperatures. Common cooling solutions include:

- Heat sinks and fans
- Liquid cooling systems
- Thermal paste and heat spreaders

Effective thermal management improves performance, extends hardware lifespan, and prevents overheating-related failures. In many systems, performance is directly linked to temperature, as CPUs and GPUs may reduce speed to avoid damage when they become too hot.

4 Processor Architecture

As the topics in this article become more complex and more significant within the architectural model, it is necessary to shift to a more detailed level of explanation. For this reason, the following sections expand on the functionality of key concepts and explain how they fit within fundamental computer science knowledge.

4.1 Instruction Set Architecture (ISA)

A CPU can be thought of as a machine that understands only a specific language. The **Instruction Set Architecture (ISA)** is that language. Just as a human language has words and grammatical rules that give meaning, an ISA defines which instructions exist, what they do, and how they are used. In other words, it specifies the vocabulary and grammar of the computer architecture.

An ISA tells programmers and compilers which commands the CPU understands, such as adding or subtracting numbers, loading and storing data, and comparing values. It also defines the types of data the CPU can operate on, how memory is accessed, and how data moves between memory and the CPU. Additionally, the ISA specifies the available registers, which are small and fast storage locations inside the CPU used to hold temporary data and intermediate results.

The ISA is critically important because software is written for an ISA rather than for a specific physical processor. This means that different CPUs can run the same programs as long as they implement the same ISA. As a result, hardware designers are free to change or optimize the internal design of a CPU, as long as it continues to follow the rules defined by the ISA.

Some real-world examples of widely used ISAs include:

- **x86:** Used by most Intel and AMD desktop and laptop computers
- **ARM:** Used in smartphones, tablets, and many modern laptops
- **RISC-V:** An open and modern ISA widely used in research and industry

In conclusion, the ISA acts as the interface between software and hardware. It defines the commands a CPU understands and specifies how those commands behave, enabling compatibility between programs and different hardware implementations.

4.1.1 Instruction Formats and Encodings

An **instruction format** describes how a single CPU instruction is laid out in bits. You can think of an instruction as a form with different fields. Kind of like a train ticket that includes a destination, a seat number, and a class. In the same way, each instruction has fixed fields that tell the CPU what to do and what data to use.

Some typical instruction format fields include:

1. **Opcode** Tells the CPU which operation to perform, such as `ADD`, `SUB`, `LOAD`, or `JUMP`.
2. **Operands** Indicate which data to use: registers, memory addresses, or immediate values.
3. **Addressing mode** (sometimes implicit) Explains how to interpret the operands, for example whether a value comes from a register, from memory, or is written directly in the instruction.

A simple conceptual example to start getting used to this idea is:

```
[ opcode | register1 | register2 | register3 ]
```

This could mean: add the values in `register2` and `register3`, and store the result in `register1`. Simple enough.

Instruction Encoding (the actual bits)

Instruction encoding is how the instruction format is translated into 0s and 1s. The CPU does not see something readable like:

```
ADD R1, R2, R3
```

What the CPU actually sees is something more like:

```
010011 001 010 011
```

Not very friendly, but much more useful for hardware.

For example, suppose:

- The opcode uses 6 bits
- Each register uses 3 bits

Then the encoding would look like this:

```
010011 | 001 | 010 | 011
 opcode   R1    R2    R3
```

This is the binary encoding of the instruction.

Now, not all ISAs use the same kind of instruction format. The two most common approaches are **fixed-length** and **variable-length** instructions.

Fixed-length instructions

In this format, all instructions have the same size (for example, 32 bits). This makes decoding easier and usually faster, which keeps the hardware simpler. Architectures like ARM and RISC-V use fixed-length instructions.

```
[ opcode | field | field | field ] \quad (always 32 bits)
```

Variable-length instructions

Here, instructions can have different sizes. This often makes programs more compact, but decoding becomes more complicated. The x86 architecture is a classic example of this approach.

```
[ prefix ][ opcode ][ mod ][ reg ][ immediate ][ ... ]
```

In the end, instruction formats and encodings affect almost everything: how complex the CPU is, how fast it can decode instructions, how much power it uses, how big programs are, and even how compilers are designed. That's why different ISAs make different trade-offs.

4.1.2 Addressing Modes

An **addressing mode** defines how an instruction finds its operands (the data it works on). In short, it answers a very simple question: *where is the data?*

The data used by an instruction can be located in different places: it can be written directly inside the instruction, stored in a CPU register, located in memory, or found at an address that is computed using some math. Addressing modes exist to describe all these possibilities.

Addressing modes give the CPU flexibility without requiring thousands of different instructions. The idea is simple: the **opcode** says *what* to do, and the **addressing mode** says *where* the data is.

There are many addressing modes, but some of the most important ones are listed below.

1. Immediate addressing

The value is stored directly inside the instruction.

```
ADD R1, R2, #5
```

Here, `#5` is the actual value. This is fast because no memory access is needed, but the value size is limited.

2. Register addressing

The operand is stored in a CPU register.

```
ADD R1, R2, R3
```

All data comes from registers, which makes this very fast.

3. Direct (absolute) addressing

The instruction contains the memory address of the data.

```
LOAD R1, 1000
```

This loads the value stored at memory address `1000` into `R1`.

4. Indirect addressing

The instruction refers to a memory location that holds another address.

```
LOAD R1, (1000)
```

The CPU first reads the address stored at `1000`, then uses that address to find the actual data.

5. Register indirect addressing

The address of the data is stored in a register.

```
LOAD R1, (R2)
```

The CPU uses the value in `R2` as the memory address.

6. Base + offset (displacement) addressing

The final address is computed by adding an offset to a register value.

```
LOAD R1, 8(R2)
```

The effective address is `R2 + 8`, commonly used for arrays and stack access.

7. Indexed addressing

The address is computed using an index register.

```
LOAD R1, (R2, R3)
```

The effective address is typically `R2 + R3`, which is useful in loops.

A simplified view of how addressing modes work internally is the following: for a memory instruction, the CPU decodes the instruction, determines the addressing mode, computes the **effective address**, accesses memory if needed, and finally executes the operation. The effective address is the final memory location calculated by the addressing mode.

4.1.3 Register File

A **register file** is a small, very fast storage area inside the CPU that holds the registers. If main memory is a warehouse, then registers are the items on the workbench, and the register file is the organized rack that holds them.

Accessing main memory is slow, while accessing registers is extremely fast. Registers allow the CPU to store temporary values, perform arithmetic quickly, avoid constant memory accesses, and execute instructions efficiently.

A register file is defined by the ISA and implemented in hardware. It specifies how many registers exist and how they are used. Some examples include:

- **RISC-V:** 32 integer registers
- **ARM:** 31 general-purpose registers
- **x86:** Fewer architectural registers (but many more hidden internally)

In general, **more registers means fewer memory accesses**, which usually means better performance.

Register width

Each register has a fixed size that matches the word size of the ISA:

- 32-bit CPU → 32-bit registers
- 64-bit CPU → 64-bit registers

Register file access

A register file can read multiple registers at the same time and write to one (or sometimes more) registers per clock cycle. A typical RISC design uses **two read ports and one write port**, which matches common instructions like:

```
ADD R1, R2, R3
```

This instruction reads `R2` and `R3`, and writes the result into `R1`.

Register file during instruction execution

Using the instruction `ADD R1, R2, R3` as an example, the register file is used as follows:

1. The instruction is decoded
2. The register file:
   - reads `R2`
   - reads `R3`
3. The ALU adds the two values
4. The result is written back to `R1`

In simple CPUs, all of this can happen within a single clock cycle.

Types of registers

There are two main types of registers associated with a register file:

- **General-purpose registers (GPRs):** Used for arithmetic, addresses, and pointers. Most instructions operate on these.
- **Special-purpose registers:** Often separate or logically distinct, such as:
  - Program Counter (PC)
  - Stack Pointer (SP)
  - Status / Flags register

  Some ISAs include the PC or SP inside the register file, while others do not.

Register file vs. main memory

Feature	Register File	Main Memory
Location	Inside the CPU	Outside the CPU
Speed	Fastest	Much slower
Size	Very small	Very large
Access	Direct	Via LOAD / STORE

Key takeaways

- Register file = collection of CPU registers
- Fastest storage in the system
- Multiple read ports, limited write ports
- Central to instruction execution
- Defined by the ISA, optimized by the microarchitecture

4.1.4 Data Types and Operand Sizes

A **data type** tells the CPU how to interpret a bunch of bits. By themselves, bits have no meaning at all; the data type is what gives them meaning. Some common CPU data types are the basic ones you already know: integers, floating-point numbers, booleans, and addresses (or pointers). An address is simply a number that represents a memory location, and its size is usually equal to the register size of the CPU.

What are operand sizes?

An **operand size** specifies how many bits are used to represent a value. In other words, it answers the question: *how big is this number?*

Common operand sizes are shown below:

Size	Bits	Name
1 byte	8	Byte
2 bytes	16	Halfword
4 bytes	32	Word
8 bytes	64	Double word

Operand size matters because the same instruction can behave very differently depending on how many bits it operates on. For example:

- An **8-bit add** works with small numbers and overflows quickly
- A **64-bit add** handles much larger numbers and offers more precision, but may be slightly slower

Another very important concept is **signed vs. unsigned** values. The bits may be exactly the same, but the way they are interpreted is different. A value can represent a negative number or a large positive number depending entirely on whether it is treated as signed or unsigned.

Memory access instructions must also know the operand size, because the CPU needs to know how many bytes to read or write. Using the wrong size means reading the wrong data. For example:

```
LOADB   -> 1 byte
LOADW   -> 2 bytes
LOAD    -> 4 or 8 bytes
```

What you should absolutely remember

- Data type = meaning of the bits
- Operand size = number of bits used
- The same bits can mean different things
- CPU instructions are size-aware
- Larger sizes mean more range and precision

In summary, data types define *what* a value represents, while operand sizes define *how many bits* are used to store and process it.

4.1.5 Control Flow Instructions

Control flow instructions decide which instruction the CPU executes next. Normally, the CPU runs instructions one after another in order. Control flow instructions change that order.

By default, during normal execution:

1. The CPU executes an instruction
2. The Program Counter (PC) moves to the next instruction
3. The process repeats

This is called **sequential execution**.

Without control flow, programs would only be able to run straight-line code. They could never make decisions, repeat actions, or structure code into functions. Control flow instructions enable `if/else` statements, loops, function calls, and even program start and exit.

There are different types of control flow instructions:

- **Jump (unconditional branch):** always changes the execution flow
- **Conditional branch:** jumps only if a condition is true
- **Compare instructions:** set conditions used by branches
- **Function call and return:** move execution to and from functions

Modern CPUs try to execute instructions ahead of time to improve performance. However, branches can cause problems because:

- The CPU may not know which path will be taken
- This can lead to stalls or wasted work

This is why modern processors use **branch prediction**, an advanced technique that guesses which path will be taken before the condition is fully resolved.

Key ideas

- Control flow instructions change the execution order
- They work by modifying the Program Counter (PC)
- Main types include jumps, conditional branches, calls, and returns
- They are essential for decisions, loops, and structured programs

In short, control flow instructions direct the CPU to choose which instruction runs next, making decisions, loops, and function calls possible.

4.2 RISC vs CISC Design Philosophies

4.2.1 RISC Principles

A **RISC** design is a way to build a processor that focuses on simplicity and speed. RISC stands for **Reduced Instruction Set Computer**. Instead of giving the processor many complex commands, RISC uses a small set of very simple instructions. Each instruction does one small thing, and most instructions take the same amount of time to execute (often one clock cycle).

You can think of RISC like LEGO bricks: each brick is simple, but by combining many of them you can build complex things—anything from a small house to a Death Star, or even a whole movie.

Load–store approach

In RISC architectures, only two instructions are allowed to access memory: **load** and **store**. A `load` instruction moves data from memory into a register, and a `store` instruction moves data from a register back to memory. All calculations happen using registers, which are very fast storage locations inside the CPU. This keeps the processor fast and predictable.

RISC designs simplify the hardware and rely on smarter software instead. The CPU is easier to design and can use techniques such as **pipelining**, which overlaps work like an assembly line. Compilers (software that translates code into instructions) do more of the work by arranging simple instructions efficiently.

Why it is fast and popular

RISC designs have several advantages: they use less power, run efficiently at high speeds, and scale well. This is why RISC is widely used today, from ARM processors in phones and tablets to Apple Silicon processors based on ARM, as well as many embedded systems.

4.2.2 CISC Principles

CISC follows almost the opposite design philosophy of RISC. CISC stands for **Complex Instruction Set Computer**. These processors provide many powerful instructions, but each instruction is more complex. A single instruction can perform multiple steps at once and may take different amounts of time to execute.

Using the LEGO analogy again, CISC is like having large prebuilt blocks. One block can do the work of several small bricks at once.

In CISC architectures, instructions can work directly with memory: loading data, operating on it, and storing it back in a single command. This reduces the number of instructions a program needs, which was especially important when memory was small and programs had to be compact.

Because the instructions are complex, the CPU hardware is also more complex. Many CISC instructions are internally broken down into simpler steps called **micro-operations**. As a result, compilers have an easier job because the CPU does more of the heavy lifting. In CISC, more work is done in hardware and less in software.

So why does CISC still matter today? CISC designs are excellent at running older software and can be extremely powerful thanks to modern optimizations. The most well-known examples are **x86** and **x86-64**, used by Intel and AMD CPUs in most desktop and laptop computers.

RISC vs. CISC comparison

RISC	CISC
Simple instructions	Complex instructions
Fixed instruction size	Variable instruction size
Load/store only	Memory operations inside instructions
Easier CPU design	More complex CPU design
Very power efficient	Backward compatible and powerful

4.2.3 Case Studies: ARM, x86, and RISC-V

To understand how RISC and CISC ideas work in practice, it helps to look at real processors. We start with ARM, then x86, and finally RISC-V.

ARM: Commercial RISC Built for Efficiency ARM is a commercial RISC architecture designed from the beginning with efficiency in mind. Its design philosophy is based on simple, mostly fixed-length instructions and a strong load–store model. From day one, ARM focused on low power consumption.

In practice, this means ARM achieves excellent performance per watt. It is very effective at pipelining and out-of-order execution while keeping energy usage low, which makes it ideal for battery-powered devices.

ARM processors are commonly used in:

- Smartphones and tablets
- Laptops using Apple Silicon (M1, M2, etc.)
- Embedded systems
- Some servers

x86: CISC on the Outside, RISC on the Inside The x86 architecture was originally designed as a CISC instruction set. It uses large, complex, variable-length instructions and places a strong emphasis on backward compatibility.

What actually happens inside modern x86 CPUs is more interesting. Today's x86 processors translate complex CISC instructions into simpler, RISC-like **micro-operations**. These micro-ops then flow through a pipeline that looks very similar to a RISC design.

So despite being CISC at the instruction-set level, the execution core behaves very much like RISC. Most of the complexity lives in the front end of the processor.

x86 processors are typically used in:

- Desktop and laptop computers
- High-performance servers
- Workstations

x86 proves that it is possible to keep a CISC interface for compatibility while using RISC ideas internally for performance.

RISC-V: Pure and Open RISC RISC-V takes the RISC philosophy to its logical extreme. It uses a very small base instruction set, with optional extensions added only when needed. This allows designers to build a CPU with exactly the features they want—no more, no less.

RISC-V is easier to design and verify, has no license fees, and is completely open. This makes it especially attractive for research, startups, and specialized hardware.

RISC-V is commonly used in:

- Microcontrollers
- Embedded systems
- Accelerators
- Emerging servers and research CPUs

In short, RISC-V is minimal, clean, and open.

ARM, x86, and RISC-V make different design choices that directly affect performance, power, and cost. These differences all trace back to RISC vs. CISC trade-offs.

1. Performance

Instruction decoding

- **x86**
  - Variable-length, complex instructions
  - Requires large and power-hungry decode logic
  - Instructions are broken into micro-operations
  - Decode stage can become a bottleneck
- **ARM / RISC-V**
  - Mostly fixed-length instructions
  - Simple and fast decoding
  - Instructions flow directly into execution

Effect: RISC designs waste fewer cycles and less silicon on decoding.

Pipeline efficiency

- **RISC (ARM, RISC-V)**
  - Uniform instruction timing
  - Predictable pipelines
  - Easier to keep execution units busy
- **x86**
  - Instruction timing varies
  - Requires more buffering, reordering, and speculation

Effect: RISC pipelines scale more cleanly to wide, high-frequency designs.

Real-world performance paradox
Despite the above, x86 CPUs often match or beat ARM in raw performance because of:

- Larger die area
- Extremely aggressive out-of-order execution
- Huge caches

Key point: x86 wins with brute force; RISC wins with efficiency.

2. Power

Decode and control logic

- **x86**
  - Decode can consume a significant fraction of total power
  - Microcode engines stay active
- **ARM / RISC-V**
  - Smaller control logic

– Lower switching activity

Effect: Simpler instructions mean fewer transistors toggling, which means lower power.

Energy per instruction

- **RISC:** More instructions, but each one is cheap
- **CISC:** Fewer instructions, but each one is energy-expensive

Result: RISC usually wins in energy per task, not instruction count.

3. Cost

Silicon area

- **x86:** Large decode logic, complex compatibility circuitry, large die
- **ARM:** Moderate complexity, optimized for mass production
- **RISC-V:** Minimal base ISA, smallest possible cores

Effect: Smaller dies mean more chips per wafer and lower cost.

Design and customization cost

- **x86:** Extremely expensive to design and verify
- **ARM:** License fees, but a mature ecosystem
- **RISC-V:** No license fees and full customization

Summary Table

Factor	ARM	x86	RISC-V
Decode complexity	Low	Very high	Very low
Pipeline efficiency	High	Medium	High
Power efficiency	Very high	Medium	Very high
Die size	Medium	Large	Small
Design cost	High (license)	Very high	Low
Customization	Limited	None	Excellent

The deep truth

- x86 spends complexity on compatibility
- ARM spends complexity on power efficiency
- RISC-V lets designers choose where to spend complexity

RISC designs minimize wasted work in power, silicon, and control logic, while x86 compensates with scale and aggressive optimization.

5 Microarchitecture

5.1 Datapath and Control Unit Design

At a high level, a CPU can be split into two main parts: the **datapath** and the **control unit**. The datapath does the actual work on data, while the control unit tells the datapath what to do and when to do it. You can think of it as *muscles versus brain.*

Datapath: what it is

The **datapath** is the collection of hardware blocks that store, move, and operate on data. Typical datapath components include:

- **Registers** (store values)
- **ALU** (add, subtract, AND, compare, etc.)
- **Multiplexers (MUXes)** that choose between different inputs
- **Buses and wires** that carry data
- Sometimes shifters, comparators, and other small units

The datapath answers questions like:

- "Add these two numbers"
- "Move this value from register A to register B"

The datapath can perform many different operations, but it does not decide *which* operation to perform by itself. Using a factory analogy, the datapath is like the factory floor full of machines and conveyor belts. It can cut, move, and assemble things—but only if someone tells it what to do.

Control Unit: the boss

The **control unit** is the "boss" of the CPU. It generates control signals that tell the datapath:

- Which operation to perform
- Which registers to read or write
- Which paths the data should take

It controls many parts of the CPU, including the ALU, register file, multiplexers, and the Program Counter (PC).

How it works

The control unit looks at the current instruction (its opcode) and the current state or clock cycle. Based on this information, it decides what should happen in the current cycle and what should happen in the next one.

The control unit is like a manager reading instructions and shouting commands to the workers: *"You add these numbers!" "You store the result over there!"*

Together, the datapath and control unit form a complete system: one that knows how to do the work, and one that knows when and how the work should be done.

5.2 Instruction Execution Cycle

Every instruction in a CPU goes through the same basic loop: **Fetch** → **Decode** → **Execute** → **Memory** → **Write-back**. Not every instruction uses every step, but conceptually the order is always the same.

1. Fetch (get the instruction)

In this stage, the CPU fetches the next instruction from instruction memory and places it into the **Instruction Register (IR)**. The control unit enables a memory read and updates the Program Counter (PC), for example:

$$PC = PC + 4$$

(assuming fixed-length instructions).

Think of this step as: *"Bring me the next command."*

2. Decode (understand the instruction)

During decode, the instruction is analyzed. The datapath extracts the instruction fields such as the opcode, register numbers, and immediate values. At the same time, the register file is accessed to read the source registers.

The control unit decodes the opcode and decides:

- Which ALU operation is needed
- Whether memory will be accessed
- Whether a register will be written

Think of this step as: *"What does this instruction want me to do?"*

3. Execute (do the operation)

This is where the main computation happens. The datapath performs arithmetic or logical operations using the ALU, computes addresses, or evaluates comparisons for branches.

The control unit selects the ALU operation and chooses the correct ALU inputs using multiplexers (MUXes).

Think of this step as: *"Do the actual work."*

4. Memory access (if needed)

If the instruction is a load or store, memory is accessed during this stage. Loads read data from memory, and stores write data to memory. Instructions that do not use memory simply skip this step.

5. Write-back

Finally, if the instruction produces a result, it is written back into the destination register in the register file.

The instruction execution cycle is a repeated process where the control unit guides the datapath through the stages of fetch, decode, execute, memory access, and write-back, over and over again for every instruction.

5.3 Pipelining

Pipelining allows a CPU to work on multiple instructions at the same time by breaking instruction execution into stages, much like an assembly line.

Imagine a car wash with four stations: wash, rinse, dry, and inspect. Every car must go through all four steps. However, while Car A is being rinsed, Car B can be washed, and Car C can be inspected. Once the pipeline is full, one car finishes at every step, even though each individual car still takes time to pass through all stations. This is a simple way to understand pipelining.

Now, how does this idea map to a CPU?

A CPU instruction is split into the same five stages discussed earlier: **fetch, decode, execute, memory, and write-back**. Instead of completing all five stages for one instruction before starting the next, the CPU overlaps these stages across multiple instructions.

Without pipelining:

```
Instruction 1: IF → ID → EX → MEM → WB
Instruction 2:                      IF → ID → EX → MEM → WB
```

Only one instruction is active at a time.

With pipelining:

```
Cycle 1: Inst1 IF
Cycle 2: Inst1 ID | Inst2 IF
Cycle 3: Inst1 EX | Inst2 ID | Inst3 IF
Cycle 4: Inst1 MEM| Inst2 EX | Inst3 ID | Inst4 IF
Cycle 5: Inst1 WB | Inst2 MEM| Inst3 EX | Inst4 ID | Inst5 IF
```

After the pipeline fills, one instruction completes every cycle. This increases **throughput**, meaning more work is done per unit of time, even though the latency of a single instruction stays the same.

Problems pipelining must handle (briefly)

1. **Data hazards** An instruction needs a result that is not ready yet.
2. **Control hazards** Branches and jumps change which instruction comes next.
3. **Structural hazards** Two stages need the same hardware resource at the same time.

Modern CPUs handle these problems using techniques such as forwarding, stalling, and branch prediction.

One-line summary

Pipelining is an assembly-line approach to instruction execution that improves CPU performance by overlapping the stages of multiple instructions.

5.3.1 Pipeline Stages

Most textbooks describe pipelining using a classic **5-stage RISC pipeline model**. Each instruction is broken into stages, and pipelining overlaps these stages across multiple instructions.

1. Instruction Fetch (IF)

- The CPU reads the instruction from instruction memory
- The Program Counter (PC) is updated to point to the next instruction

2. Instruction Decode (ID)

- The instruction is decoded (what operation to perform)
- Source registers are read from the register file
- Control signals are generated

3. Execute (EX)

- The ALU performs arithmetic or logical operations
- Branch conditions are evaluated
- Effective memory addresses are calculated

4. Memory Access (MEM)

- Data memory is read or written for load and store instructions

- Other instructions simply pass through this stage

5. Write Back (WB)

- The result of the instruction is written back to the register file

Key point: Each stage performs part of the work, and pipelining overlaps these stages across different instructions to increase throughput.

Pipeline Hazards

A **pipeline hazard** is anything that prevents the next instruction from executing in the following clock cycle. There are three main types of hazards.

A. Structural Hazards (hardware conflicts) **What happens?** Two pipeline stages need the same hardware resource at the same time.
Example: Instruction fetch and data memory access both need the same memory unit.
Result: One instruction must stall.

B. Data Hazards (instruction dependencies) **What happens?** An instruction depends on the result of a previous instruction that has not finished yet.
Example:

```
ADD R1, R2, R3   ; R1 = R2 + R3
SUB R4, R1, R5   ; uses R1 immediately
```

Types of data hazards:

- **RAW (Read After Write)** — most common and most important
- **WAR (Write After Read)**
- **WAW (Write After Write)**

In simple in-order pipelines, RAW hazards dominate.

C. Control Hazards (branches) **What happens?** The CPU does not know which instruction comes next because of a branch.
Example:

```
BEQ R1, R2, LABEL
```

The pipeline may fetch the wrong instruction before the branch outcome is known.

Hazard Mitigation Techniques

Modern CPUs use several techniques to reduce or eliminate pipeline stalls.

A. Structural hazard solutions

- **Resource duplication:** separate instruction and data memory, or multiple functional units

This approach is used in Harvard-style architectures.

B. Data hazard solutions 1. Pipeline stalling (bubble insertion)

- Pause the pipeline until the data is ready
- Simple, but reduces performance

2. Forwarding (bypassing)

- Send results directly from one pipeline stage to another
- Avoid waiting for write-back
- Most common solution

3. Register renaming

- Eliminates WAR and WAW hazards
- Used in out-of-order execution

C. Control hazard solutions 1. Pipeline flush

- Discard incorrectly fetched instructions
- Simple, but costly

2. Branch prediction

- Guess whether a branch will be taken or not
- Continue execution speculatively
- Can be static or dynamic (history-based)

3. Delayed branch

- Always execute the instruction after the branch
- Compiler schedules useful work

Used in early RISC designs.

Quick summary

Hazard	Problem	Solution
Structural	Hardware conflict	Duplicate resources
Data	Instruction dependency	Forwarding, stalls
Control	Unknown next instruction	Prediction, flush

One-sentence takeaway

Pipeline stages divide instruction work, hazards disrupt smooth flow, and mitigation techniques keep the pipeline full and efficient.

5.4 Superscalar and Out-of-Order Execution

Modern processors try to execute more than one instruction at the same time to improve performance. This idea is known as **superscalar execution**. A superscalar processor has multiple execution units, so it can issue and execute several instructions during the same clock cycle instead of just one.

However, instructions in a program are written in a specific order, and sometimes one instruction must wait for the result of another. To reduce waiting time, processors use **out-of-order execution**. This means the processor can execute instructions in a different order than they appear in the program, as long as the final result is the same. By doing this, the processor keeps its execution units busy and avoids unnecessary delays.

5.5 Branch Prediction

A branch instruction occurs when the program must choose between different paths, such as in `if` statements or loops. When the processor reaches a branch, it does not immediately know which path will be taken. If it waits for the decision, performance decreases.

To solve this problem, processors use **branch prediction**. Branch prediction is a technique where the processor guesses which path will be taken and continues executing instructions along that path. If the guess is correct, execution continues smoothly. If the guess is wrong, the processor must discard the incorrect instructions and restart, which causes a small performance penalty. Accurate branch prediction is very important for maintaining high performance.

5.6 Instruction-Level Parallelism

Instruction-Level Parallelism (ILP) refers to the ability of a processor to execute multiple instructions at the same time. Many instructions in a program are independent, meaning they do not rely on the results of others. These independent instructions can be executed in parallel.

Techniques such as superscalar execution, out-of-order execution, and branch prediction are all used to increase instruction-level parallelism. The more ILP a processor can exploit, the more work it can do in less time, leading to faster program execution.

6 Machine Language and Binary Code

6.1 Machine Instructions and Opcodes

This section marks the first real step toward assembly language. Before going too deep into assembly syntax, we need to understand machine instructions, opcodes, and a few basic concepts.

Opcodes are the core building blocks of assembly. They are the fundamental instructions that a CPU can execute. We have already seen opcodes mentioned in earlier sections, but here it is important to understand more clearly what they are, how they work, and why they matter.

The basic idea

A computer only understands numbers, not words. Machine instructions are those numbers that tell the CPU what to do. The **opcode** is the part of a machine instruction that specifies *which action* the CPU should perform.

You can think of a machine instruction as a complete command, and the opcode as the *verb* inside that command.

What is a machine instruction?

A **machine instruction** is a binary value (a sequence of 0s and 1s) that is read directly by the CPU. Each instruction tells the CPU to perform exactly one small, well-defined step.

What is an opcode?

Inside every machine instruction, the opcode tells the CPU *what action to perform.* The remaining bits of the instruction describe *how* to perform that action and *with which data* (registers, memory, or immediate values).

Some common opcode actions include:

- `MOV` — move data
- `ADD` — add numbers
- `SUB` — subtract numbers
- `LOAD` — read from memory
- `STORE` — write to memory
- `JUMP` — go to another instruction

- `COMPARE` — compare values

How the CPU uses opcodes
A CPU is essentially a very fast loop that repeatedly:

1. Fetches an instruction
2. Looks at the opcode
3. Activates a fixed hardware operation
4. Moves to the next instruction

That's it. There is no "thinking" or "understanding" involved—just pattern matching followed by action. The opcode is simply a selector that activates a specific piece of hardware inside the CPU.

6.2 Instruction Decoding and Execution

Instruction decoding and execution are simply the CPU figuring out what action you want and then doing it using its built-in hardware—nothing more.

The CPU reads a tiny command, recognizes its pattern, turns on the correct internal components, lets them do their job, and then moves on.

To visualize this, think of the CPU as a very fast kitchen. The instruction is a recipe card, the opcode is the name of the dish, the operands are the ingredients (numbers and registers), and the kitchen tools are already built and waiting. The CPU does not invent anything—it just follows recipes extremely fast.

Instruction decoding answers the question: *What kind of instruction is this?* You can think of decoding as the CPU looking at the first part of the instruction and recognizing which built-in action it corresponds to.

Internally, it feels like:

- "Ah, this pattern means ADD"
- "This one means LOAD"
- "This one means JUMP"

There is no thinking involved—just pattern matching.

Each opcode already has dedicated hardware wired for it. When the CPU decodes an instruction, it is really selecting a hardware path, not interpreting meaning the way a human would.

Decoding vs. Execution

- **Decoding** = choose the tool
- **Execution** = use the tool

The full flow looks like this: the CPU picks up the next instruction, glances at its beginning to recognize the action, flips the internal switches connected to that action, the hardware performs the work automatically, the result settles into a register or memory, and the CPU calmly walks to the next instruction.

In short: decoding is recognition, execution is electrical movement.

6.3 Endianness and Data Alignment

Two big ideas up front

- **Endianness** is about *byte order* (which byte goes first)
- **Data alignment** is about *where data is placed in memory*

They answer two different questions:

- Endianness asks: *"In what order?"*
- Alignment asks: *"At what address?"*

Endianness

The core idea of endianness is the rule a system uses to decide which part of a number is stored first in memory. Computers store data *byte by byte*, not all at once.

Imagine writing the number `1234` across a row of boxes:

- **Big-endian**: write it the way we read numbers

$$1 \mid 2 \mid 3 \mid 4$$

- **Little-endian**: write the smallest part first

$$4 \mid 3 \mid 2 \mid 1$$

Same number. Different order.

Endianness is simply a convention for how multi-byte numbers are laid out in memory. It exists because memory is a long row of single-byte boxes, while numbers are often larger than one byte. The CPU needs a consistent rule—and endianness *is* that rule.

Data Alignment

Data alignment is about placing data at memory addresses that the CPU can access efficiently.

Think of memory as a street with numbered houses.

Some furniture:

- fits in 1 house
- needs 2 houses
- needs 4 houses

The CPU prefers that large furniture starts at addresses that are multiples of its size.

For a 4-byte integer:

- **Aligned**: starts at address 0, 4, 8, 12, ...
- **Misaligned**: starts at address 1, 2, or 3

Alignment is about putting data where the hardware expects it, so it does not have to work harder.

Proper alignment means:

- faster memory access
- simpler hardware design
- fewer special cases

On some CPUs, misaligned accesses are slower; on others, they can even cause a crash. The key idea is that CPUs are built to grab chunks of data cleanly—not awkwardly split across boundaries.

Key distinction

Endianness decides the *order* of bytes. Alignment decides the *position* of bytes.

Why programmers care

Because this affects:

- reading raw memory
- sharing data between systems
- networking
- low-level programming
- performance and correctness

6.4 Executable Binary Formats

An executable binary format is simply an agreed-upon way to package a program so the operating system knows how to load and run it. There is nothing magical about it—it is a container with rules.

An executable file tells the OS:

- what code exists
- where that code and data should go in memory
- how execution should begin

Think of an executable as a moving checklist.
When you double-click a program, the OS silently asks:

- Where is the code?
- Where is the data?
- Where do I start running?
- What does this program need?

The binary format answers all of those questions.

What an executable is *not*

An executable is not just raw machine code.
It also contains:

- metadata
- structure
- rules for loading

Without a format, the OS would be blind.

What all executables describe

No matter the system, executable formats always describe the same core ideas.

1. Where things go in memory

Big idea: an executable is a map from file bytes to memory locations.
It tells the OS:

- "These bytes go here"
- "Those bytes go there"

2. Where execution begins
Big idea: every executable has a single official starting point.
This is the **entry point**—the first instruction the CPU executes.
3. What kind of program this is
Big idea: the OS must recognize the file before it can trust or run it.
The format identifies:

- the target CPU architecture
- OS expectations
- supported features

What happens when you run a program (friendly version)
The OS opens the file, recognizes its format, reads the instructions that describe the memory layout, loads code and data into the correct locations, prepares the program's environment, jumps to the entry point, and then lets the CPU take over.
That is execution in real life.
What's inside an executable (conceptually)
Don't think in terms of files—think in terms of sections.
Common ideas include:

- **Code** — machine instructions
- **Data** — global variables
- **Read-only data** — constants
- **Metadata** — rules and information for the OS

Big idea: executable formats separate things so the OS can treat them differently.
Why this structure matters
Because the OS needs to:

- protect memory
- share code between programs
- load programs quickly
- support debugging
- handle libraries

A flat blob of bytes would not work.
Static vs. Dynamic linking
Big idea: an executable can either carry everything it needs or reference shared libraries.

- **Static** — larger file, self-contained
- **Dynamic** — smaller file, depends on external libraries

Final takeaway
An executable binary format is a contract between the program and the operating system.
How this connects to everything before

- Opcodes → live inside the code section
- Decoding and execution → happen after loading
- Endianness and alignment → rules the format follows
- Executable format → tells the OS how to set the stage

7 Assembly Language

7.1 Assembly Syntax and Instruction Mnemonics

What is assembly? If you don't know yet—sorry, but it's time to put those hands to work.

Assembly language is a human-readable representation of machine code. While machine code is raw binary (like `10101010`), assembly uses symbolic instructions such as `MOV`, `ADD`, and `JMP`. Each assembly instruction typically maps *one-to-one* to a real CPU instruction.

Mnemonics (the "verbs")

Assembly instructions are written using **mnemonics**: short words that name CPU operations. Think of mnemonics as verbs telling the CPU what to do.

Some common examples:

Mnemonic	**Meaning**	**What it does**
MOV	Move	Copy data from one place to another
ADD	Add	Add two values
SUB	Subtract	Subtract values
MUL	Multiply	Multiply values
DIV	Divide	Divide values
CMP	Compare	Compare two values
JMP	Jump	Go to another instruction
CALL	Call	Call a function
RET	Return	Return from a function

A very important detail: `MOV` does *not* move data like a truck—it copies it. The source stays unchanged. This matters, because `MOV` is one of the most commonly used instructions.

Operands (the "nouns")

Mnemonics work on **operands**, which are the data involved in the instruction. Operands can be:

- Registers (fast CPU storage)
- Memory locations
- Immediate values (constants)

Examples:

```
MOV AX, 5        ; immediate value
MOV AX, BX       ; register
MOV AX, [1000]   ; memory
```

Basic assembly syntax

Most assembly instructions follow this pattern:

```
Mnemonic destination, source
```

Example:

```
MOV AX, BX
```

Meaning: copy the value from `BX` into `AX`.

A quick look at registers

Registers are small, very fast storage locations inside the CPU. Here are some classic x86 registers and their typical roles:

Register	Purpose
AX	General-purpose
BX	General-purpose
CX	Counter
DX	Data
IP	Instruction pointer
SP	Stack pointer

(We will explore registers in much more detail in the next section.)

Comments

Comments are used to explain code. The comment symbol depends on the assembler:

- x86 (MASM, NASM): `;`
- ARM assemblers: often `#`

Labels

Labels name positions in code so you can jump to them.

Example:

```
start:
    MOV AX, 1
    JMP start
```

Here, `start` is a label, and `JMP start` jumps back to it.

Control flow and decisions

Decision-making in assembly is done using comparisons and conditional jumps.

Example:

```
CMP AX, BX
JE equal
```

This means: compare `AX` and `BX`, and jump to `equal` if they are equal.

Common conditional jumps:

Instruction	Meaning
JE	Jump if equal
JNE	Jump if not equal
JG	Jump if greater
JL	Jump if less

Architecture matters

Assembly syntax depends heavily on the CPU architecture.

For example:

```
; x86
MOV AX, BX

; ARM
MOV R0, R1
```

Same idea, different register names. In this moment, for example, I am uing an Apple Silicon who use an ARM64 architecture, which differs significantly from x86. You should usually focus on the assembly syntax of the CPU you are actually using—but the core concepts transfer across architectures.

Putting it all together

```
MOV AX, 5
MOV BX, 3
ADD AX, BX
```

In plain human language:

1. Put 5 in `AX`
2. Put 3 in `BX`
3. Add `BX` to `AX`

Final result:

`AX = 8`

Final takeaway
Think of assembly as:

`VERB destination, source`

- Mnemonics = short names for CPU actions
- Syntax = how instructions are written
- Instructions operate on registers, memory, or constants
- Assembly is low-level but logical—not mysterious