From Silicon to Speed: Understanding Your Processor Through Rust

For experienced programmers who write code every day but have no idea what happens after compilation.

Part I — The Machine Under Your Code

Chapter 1: Why Should You Care?

- The abstraction trap: you write high-level code, but it runs on real silicon
- A simple Rust loop that runs 10x faster with one small change — why?
- What this book will teach you (and what it won't)
- How to read this book

Chapter 2: Bits, Gates, and How Computers Think

- Logic gates: NOT, AND, OR, XOR — truth tables and Rust equivalents (`!` , `&` , `|` , `^`)
- Building addition from gates: half-adder (XOR + AND), full-adder (5 gates), ripple-carry adder
- Registers as banks of 64 flip-flops, capturing on the clock edge
- The clock: synchronizing combinational logic and flip-flops, critical path, timing violations
- The cost of the clock: dynamic power (`P = CV²f`), clock distribution (25-35% of power)

Chapter 3: The ALU — The Calculator Inside

- What the ALU actually does: arithmetic + logic operations
- Integer operations vs. floating-point operations (separate units)
- Flags: carry, overflow, zero — how the CPU knows what just happened
- Two's complement: negate by flip-and-add-1, same adder for signed and unsigned
- Overflow: wrapping behavior, Rust's debug panic vs. release wrap, `wrapping_add` / `checked_add` / `saturating_add`
- SIMD: doing the same operation on multiple data at once

Chapter 4: Registers — The Fastest Memory You'll Never See

- What registers are and why they exist
- General-purpose vs. special-purpose registers (instruction pointer, stack pointer, flags)
- x86-64 register file: a guided tour

- Why you can't have 10,000 registers (speed vs. physical constraints)
- Register allocation: what the compiler does with your variables

Chapter 5: The Memory Hierarchy — A Story of Trade-offs

- The speed gap: CPU vs. RAM (and why it keeps growing)
- L1, L2, L3 cache: progressively larger, progressively slower
- Cache lines: the CPU doesn't read bytes, it reads 64-byte blocks
- How data gets loaded: the path from RAM to register
- TLB: the cache for address translations

Chapter 6: How the Cache Actually Works

- Direct-mapped, set-associative, and fully-associative caches
- Cache hits and cache misses
- Eviction policies: LRU and friends
- Write-back vs. write-through
- Cache coherence in multi-core systems (MESI protocol, simplified)
- False sharing: when cores fight over the same cache line

Chapter 7: The Instruction Cycle — Fetch, Decode, Execute

- The basic cycle every CPU follows
- What an instruction looks like in machine code
- CISC (x86) vs. RISC (ARM): different philosophies, converging reality
- Microcode: how x86 translates complex instructions into simpler ones

Chapter 8: The Instruction Pipeline — An Assembly Line for Instructions

- Why doing one instruction at a time wastes hardware
- Pipelining: overlap fetch/decode/execute across instructions
- Pipeline stages in a modern CPU (simplified 5-stage model)
- Hazards: data, control, and structural
- Stalls and bubbles: when the pipeline can't keep moving

Chapter 9: Branch Prediction — Guessing the Future

- Why branches are the pipeline's worst enemy
- Static prediction: always guess "taken" (or not)
- Dynamic prediction: the branch history table
- Two-level adaptive predictors and the idea behind them
- What happens on a misprediction: flushing the pipeline
- The cost of a misprediction in cycles

Chapter 10: Out-of-Order Execution and Superscalar Design

- In-order vs. out-of-order: reordering for speed
- The reorder buffer: tracking instructions that execute out of sequence
- Superscalar: issuing multiple instructions per cycle
- Register renaming: eliminating false dependencies
- Speculative execution: doing work you might throw away

Chapter 11: Multi-Core and Parallelism

- Why clock speeds stopped growing
- Multi-core: the modern answer to Moore's Law
- Symmetric multiprocessing (SMP)
- Hardware threads (hyper-threading / SMT)
- The memory consistency problem
- Memory ordering: what "happens before" means at the hardware level

---

Part II — Writing Rust That Respects the Hardware

Chapter 12: From Rust to Machine Code — The Compilation Pipeline

- rustc -> LLVM IR -> machine code
- What optimizations happen at each stage
- Reading assembly output: `cargo rustc -- --emit asm`
- Godbolt (Compiler Explorer): your new best friend
- Debug vs. release mode: what `-O2` and `-O3` actually do

Chapter 13: Data Layout and Cache Performance

- Struct layout in Rust: alignment, padding, and size
- `#[repr(C)]` vs. default Rust layout
- Array of structs (AoS) vs. struct of arrays (SoA)
- Why iterating a `Vec<MyStruct>` can be slow if `MyStruct` is big
- Practical: restructuring data for cache friendliness

Chapter 14: Branching in Practice

- `if/else`, `match`, and their cost when unpredictable
- Branchless programming techniques in Rust
- `likely` / `unlikely` hints and when they matter
- Sorting to make branches predictable
- Measuring branch mispredictions with `perf stat`

Chapter 15: SIMD in Rust

- Auto-vectorization: when the compiler does it for you
- Checking if your loop was vectorized
- `std::simd` (portable SIMD) and what it offers
- Architecture-specific intrinsics (`std::arch`)
- Practical: speeding up a numeric computation with SIMD

Chapter 16: Memory Access Patterns

- Sequential vs. random access and what the prefetcher expects
- Linked lists vs. arrays: a cache perspective
- Iteration order matters: row-major vs. column-major in 2D arrays
- Arena allocation and bump allocators for cache locality
- `Box` , `Vec` , `HashMap` — where does the data actually live?

Chapter 17: Concurrency and the Hardware

- `std::sync::atomic` and what it maps to in hardware
- Memory ordering: `Relaxed` , `Acquire` , `Release` , `SeqCst` — demystified
- Lock-free data structures: why they're hard and when they help
- Cache line padding to prevent false sharing (`CachePadded`)
- `Rayon` and work-stealing: leveraging multiple cores idiomatically

Chapter 18: Alignment, Allocation, and the OS

- Stack vs. heap: what the hardware actually sees
- Page tables and virtual memory
- Huge pages and when they help
- Custom allocators in Rust (`GlobalAlloc` , `jemalloc` , `mimalloc`)
- Alignment requirements and `#[repr(align(N))]`

Chapter 19: Profiling and Measuring

- Why you must measure, never guess
- `cargo bench` and Criterion.rs
- `perf` on Linux: stat, record, report
- Instruments on macOS
- Flame graphs: reading and interpreting them
- Key hardware counters: cache misses, branch misses, IPC
- Micro-benchmarking pitfalls: dead code elimination, loop hoisting

Chapter 20: Case Studies

- **Case 1**: Sorting algorithm showdown — why quicksort beats mergesort in practice (cache)
- **Case 2**: A matrix multiply that goes from 2 GFLOPS to 20 GFLOPS (tiling + SIMD)
- **Case 3**: A concurrent counter — from `Mutex` to `AtomicU64` to sharded counters
- **Case 4**: Parsing a large file — branchless, SIMD-accelerated, cache-friendly
- Each case: naive code, profile, identify bottleneck, optimize, measure again

Chapter 21: Zero-Copy by Default — Rust's Hidden Performance Advantage

- The problem: zero-copy is fast but dangerous (dangling pointers, use-after-free)
- The borrow checker as the mechanism that makes zero-copy safe at zero runtime cost
- References (`&T` , `&mut T`): 8-byte pointers, no copies, no runtime safety checks
- Slices (`&[T]` , `&str`): fat pointers as views into existing memory, subslicing is pointer arithmetic
- String slices: parsing without allocation
- Iterators: zero-allocation chains that compile to tight loops (x86-64 assembly example)
- Return value optimization (NRVO): constructing in the caller's stack frame

---

Appendices

Appendix A: x86-64 Assembly Cheat Sheet

- Registers, common instructions, calling convention

Appendix B: ARM64/AArch64 Assembly Cheat Sheet

- Apple Silicon and ARM server

Appendix C: Glossary

- Quick reference for all hardware terms used in the book

Appendix D: Tools Reference

- `perf` , `cargo-asm` , `cargo-show-asm` , Godbolt, Instruments, Cachegrind, Valgrind

Appendix E: Further Reading

- Computer Architecture: A Quantitative Approach (Hennessy & Patterson)
- The Rust Performance Book

- What Every Programmer Should Know About Memory (Drepper)

Chapter 3: The ALU — The Calculator Inside

In the previous chapter, you saw how simple logic gates — AND, OR, NOT, XOR — can be wired together to build circuits that add binary numbers. A handful of gates gives you a half-adder. Two half-adders give you a full-adder. Chain eight full-adders together and you can add two 8-bit numbers.

That was not a toy exercise. You were building the core of the component we are going to explore in this chapter: the **Arithmetic Logic Unit**, or ALU. The ALU is where computation actually happens. Every addition, every comparison, every bitwise operation your Rust program performs eventually arrives here, at a circuit made of logic gates, and gets resolved in a single clock cycle.

The ALU is conceptually simple. It takes two inputs, performs an operation, and produces an output. But the details of what it can do — and what it *cannot* do — shape the performance of every program you write.

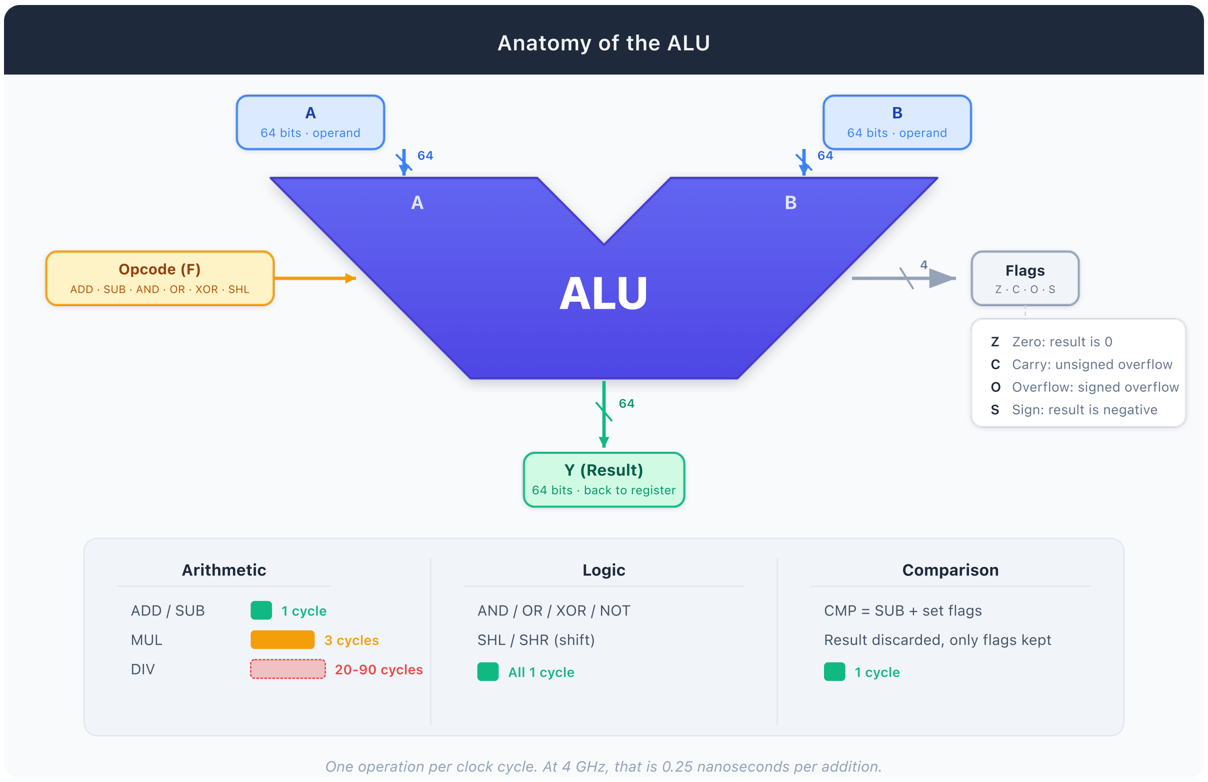
What the ALU Does

The ALU performs two categories of operations:

Arithmetic operations — addition, subtraction, multiplication, and sometimes division. These are the mathematical operations you use constantly. When you write `a + b` in Rust, the compiled machine code sends `a` and `b` to the ALU with an instruction that says "add these."

Logic operations — AND, OR, XOR, NOT, and shifts. When you write `a & b`, `a | b`, `a ^ b`, `a << 3`, or `!a` in Rust, these go to the ALU as well. Logic operations are bitwise: they operate on each bit independently.

That's it. The ALU does arithmetic and logic. It does not access memory. It does not make decisions about what to execute next. It does not know about your structs, your functions, or your types. It receives two binary values and an operation code, and it produces a binary result.



Let's look at each category more carefully.

Arithmetic: Not All Operations Are Equal

Addition and subtraction are the ALU's bread and butter. A 64-bit adder can produce a result in a single clock cycle. This is extremely fast — on a modern processor running at 4 GHz, that's one addition every 0.25 nanoseconds.

Subtraction is essentially free as well. The hardware for subtraction is almost identical to the hardware for addition — flip the bits of Input B, set the carry-in to 1, and use the same adder circuit. This works because of **two's complement**, the encoding scheme for negative numbers that we'll cover later in this chapter.

Multiplication is more expensive. Multiplying two 64-bit numbers requires significantly more gates than addition. On modern x86-64 processors, integer multiplication typically takes 3 clock cycles. That's still fast in absolute terms, but it's 3x slower than addition. In a tight inner loop that runs billions of times, that factor of 3 matters.

Division is the expensive one. Integer division on x86-64 can take 20-90+ clock cycles depending on the operand sizes. That's not a typo — division can be *ninety times* slower than addition. The reason is fundamental: division cannot be computed in a single pass through a fixed circuit the way addition can. It requires an iterative algorithm internally, similar in spirit to long division, where each step depends on the result of the previous step.

This is worth internalizing:

Operation	Typical latency (cycles)
Addition	1
Subtraction	1
Multiplication	3
Division	20-90+

These numbers are for 64-bit integers on a modern x86-64 processor. They vary by processor, but the ratios are consistent across generations: addition is cheap, multiplication is moderate, division is expensive.

This is why compilers go to great lengths to avoid division. When you write `x / 2` in Rust, the compiler does not emit a division instruction. It emits a right shift: `x >> 1`. When you write `x / 7`, the compiler replaces it with a multiplication by the modular multiplicative inverse of 7 — a clever trick that avoids division entirely. When you write `x % 8`, the compiler emits `x & 7`, a single-cycle bitwise AND.

You don't need to do these tricks yourself. The compiler handles them. But understanding *why* the compiler does this — because division is 20-90x slower than the alternatives — is the kind of hardware insight that helps you reason about performance.

Bitwise and Shift Operations

The logic side of the ALU — AND, OR, XOR, NOT — executes in a single cycle, just like addition. These operations are trivially parallel at the hardware level: each bit is independent. A 64-bit AND is just 64 AND gates operating simultaneously. There is nothing to "carry" from one bit to the next.

Shift operations (`<<` and `>>` in Rust) are also single-cycle. A **barrel shifter** — a circuit that can shift by any amount in one step — handles this. Left shift by `n` is equivalent to multiplying by 2^n. Right shift by `n` is equivalent to dividing by 2^n (for unsigned values). Both are single-cycle operations that replace what would otherwise be an expensive multiplication or division.

This is why you will sometimes see experienced systems programmers write `x << 3` instead of `x * 8`. On modern compilers, this is unnecessary — the compiler makes the substitution automatically — but the instinct comes from understanding that shifts are cheaper than multiplications at the hardware level.

Comparisons Are Subtractions in Disguise

When you write `if a > b` in Rust, the processor doesn't have a dedicated "greater than" circuit. Instead, it **subtracts** `b` from `a` and examines the result. It doesn't care about the actual difference — it only looks at the metadata that the subtraction produced.

That metadata is stored in a set of **flags**, which we'll cover in the next section. But the key insight is: comparison is subtraction. It costs exactly one cycle, same as subtraction, because it *is* subtraction — the result is just discarded.

Flags: The ALU's Side Channel

Every time the ALU performs an operation, it doesn't just produce a result. It also sets a handful of single-bit **flags** that describe properties of that result. These flags are stored in a special register (called RFLAGS on x86-64) and are used by subsequent instructions to make decisions.

The most important flags are:

Zero Flag (ZF) — Set to 1 if the result is exactly zero. When you write `if a == b`, the processor subtracts `b` from `a` and checks the Zero Flag. If `a - b == 0`, then `a == b`.

Carry Flag (CF) — Set to 1 if the operation produced a carry out of the most significant bit. For unsigned arithmetic, this indicates overflow. If you add two `u64` values and the result doesn't fit in 64 bits, the Carry Flag is set.

Overflow Flag (OF) — Set to 1 if the operation produced a signed overflow. This is different from the Carry Flag: it indicates that the result of a signed operation has the wrong sign. For example, adding two large positive `i32` values that produce a negative result.

Sign Flag (SF) — Set to 1 if the most significant bit of the result is 1. For signed numbers in two's complement, this means the result is negative.

Here's how they work together. Suppose the processor executes `CMP a, b` (compare `a` and `b`), which internally computes `a - b`:

Condition	Flags checked
`a == b`	ZF = 1
`a != b`	ZF = 0
`a < b` (signed)	SF != OF
`a > b` (signed)	ZF = 0 and SF = OF
`a < b` (unsigned)	CF = 1
`a > b` (unsigned)	CF = 0 and ZF = 0

You never interact with flags directly in Rust. The compiler generates the appropriate `CMP` instruction followed by a conditional jump that checks the right combination of flags. But understanding that comparisons are subtractions, and that branching is driven by flags, connects your high-level `if` statement to the actual hardware operations.

In Chapter 9, when we discuss branch prediction, this will matter. The processor is predicting which way these flag-based conditional jumps will go — before the subtraction has even finished.

Two's Complement: Negative Numbers in Binary

The flags section just showed you that the ALU uses different flag combinations for signed vs. unsigned comparisons. But how does the hardware represent negative numbers in the first place?

The naive approach would be to reserve one bit as a "sign bit" — 0 for positive, 1 for negative — and use the remaining bits for the magnitude. This is called **sign-magnitude** representation. It has problems: there are two representations of zero (+0 and -0), and addition doesn't work naturally. Adding +3 and -3 in sign-magnitude doesn't give you 0 without special-case logic.

Instead, virtually all modern hardware uses **two's complement**. The idea is elegant: to negate a number, flip all the bits and add 1.

Let's see it in action with 8-bit numbers:

```
5 in binary:   00000101

Flip all bits:  11111010
Add 1:          11111011  ← this is -5 in two's complement
```

Why does this work? Because of a beautiful property: if you add a number and its two's complement, you get zero (with a carry that overflows out of the register):

```
   00000101   ( 5)
 + 11111011   (-5)
 ----------
 1 00000000   overflow bit discarded → result is 0
```

The hardware doesn't need to know whether it's adding signed or unsigned numbers. **The same adder circuit works for both.** The bit pattern `11111011` is simultaneously -5 (if interpreted as `i8`) and 251 (if interpreted as `u8`). The hardware performs the same addition either way — only the interpretation of the result changes.

This is why Rust has separate types `i8` and `u8` that are the same size and use the same ALU operations. The difference is purely in how the compiler interprets the result and which comparisons it uses — signed comparisons check the Sign Flag and Overflow Flag, while unsigned comparisons check the Carry Flag, exactly as shown in the flags table above.

This is also why subtraction is essentially free. To compute `a - b`, the ALU flips the bits of `b`, sets the carry-in to 1 (which adds 1, completing the two's complement negation), and runs the same adder circuit it uses for addition.

The Two's Complement Range

For an N-bit two's complement number:

- The most significant bit is the sign bit: 0 = positive, 1 = negative.
- Positive range: 0 to 2^(N-1) - 1
- Negative range: -1 to -2^(N-1)

For 8-bit (`i8`):

```
01111111  =  127   (maximum positive)
00000001  =    1
00000000  =    0
11111111  =   -1
11111110  =   -2
10000000  = -128   (minimum negative)
```

Notice the asymmetry: there is one more negative value than positive. That's because zero takes one slot from the "positive" side. For `i8`, the range is -128 to 127, not -128 to 128. The same asymmetry exists in every signed integer type in Rust: `i16` goes to -32,768 but only to 32,767.

This asymmetry is the source of a subtle bug pattern. In Rust, the expression `-128_i8.abs()` doesn't return 128, because 128 cannot be represented as an `i8`. In debug mode, Rust panics on overflow. In release mode, it wraps silently. This is exactly the kind of hardware-level detail that matters in practice.

Overflow: What Happens When Numbers Don't Fit

What happens when you add 1 to the maximum value of a `u8`?

```
  11111111   (255)
+ 00000001   (  1)
----------
1 00000000   → carry overflows, result is 0
```

The hardware performs the addition normally. The carry bit falls off the edge of the 8-bit register. The result wraps around to 0. The ALU doesn't care — it sets the Carry Flag and moves on.

For signed overflow, consider adding 127 + 1 in `i8`:

```
  01111111   (127)
+ 00000001   (  1)
----------
  10000000   → -128 in two's complement!
```

The result "wraps" from the most positive value to the most negative. The ALU detects this by setting the Overflow Flag (the sign of the result doesn't match what you'd expect from the signs of the inputs).

Rust's behavior depends on the build mode:

- **Debug mode**: integer overflow panics. This is a safety check inserted by the compiler — it adds extra instructions to check the overflow after each arithmetic operation.
- **Release mode**: integer overflow wraps (for unsigned) or wraps in two's complement (for signed), matching the hardware behavior.
- **Explicit wrapping**: methods like `wrapping_add`, `checked_add`, `saturating_add`, and `overflowing_add` give you precise control over overflow behavior.

```rust
let a: u8 = 255;
let b: u8 = 1;

// Debug mode: panics!
// Release mode: wraps to 0
let c = a + b;

// Explicit: always wraps, no panic
let d = a.wrapping_add(b);  // 0

// Explicit: returns None on overflow
let e = a.checked_add(b);   // None

// Explicit: clamps to max
let f = a.saturating_add(b); // 255
```

Understanding two's complement tells you exactly what "wrapping" means: it means the hardware does the addition normally and drops the bit that doesn't fit. It's not a bug in the math — it's a consequence of fixed-width arithmetic.

Integer ALU vs. Floating-Point Unit

Everything we've discussed so far applies to **integer** operations. But there's a separate piece of hardware for floating-point math: the **Floating-Point Unit (FPU)**, sometimes called the floating-point execution unit.

The FPU is physically separate from the integer ALU. It has its own set of registers (on x86-64, the XMM/YMM/ZMM registers, which are 128/256/512 bits wide), its own adder, its own multiplier, and its own division logic. The reason for this separation is simple: floating-point arithmetic is fundamentally different from integer arithmetic. The IEEE 754 format requires handling sign bits, exponents, mantissas, rounding modes, and special values like NaN and infinity. The circuits that do this are specialized and complex.

The performance profile is different too:

Operation	Integer latency	Floating-point latency
Addition	1 cycle	3-5 cycles
Multiplication	3 cycles	3-5 cycles
Division	20-90 cycles	10-20 cycles

A few things stand out:

Floating-point addition is slower than integer addition. Integer addition is a single pass through an adder. Floating-point addition requires aligning the exponents, adding the mantissas, normalizing the result, and rounding — multiple stages.

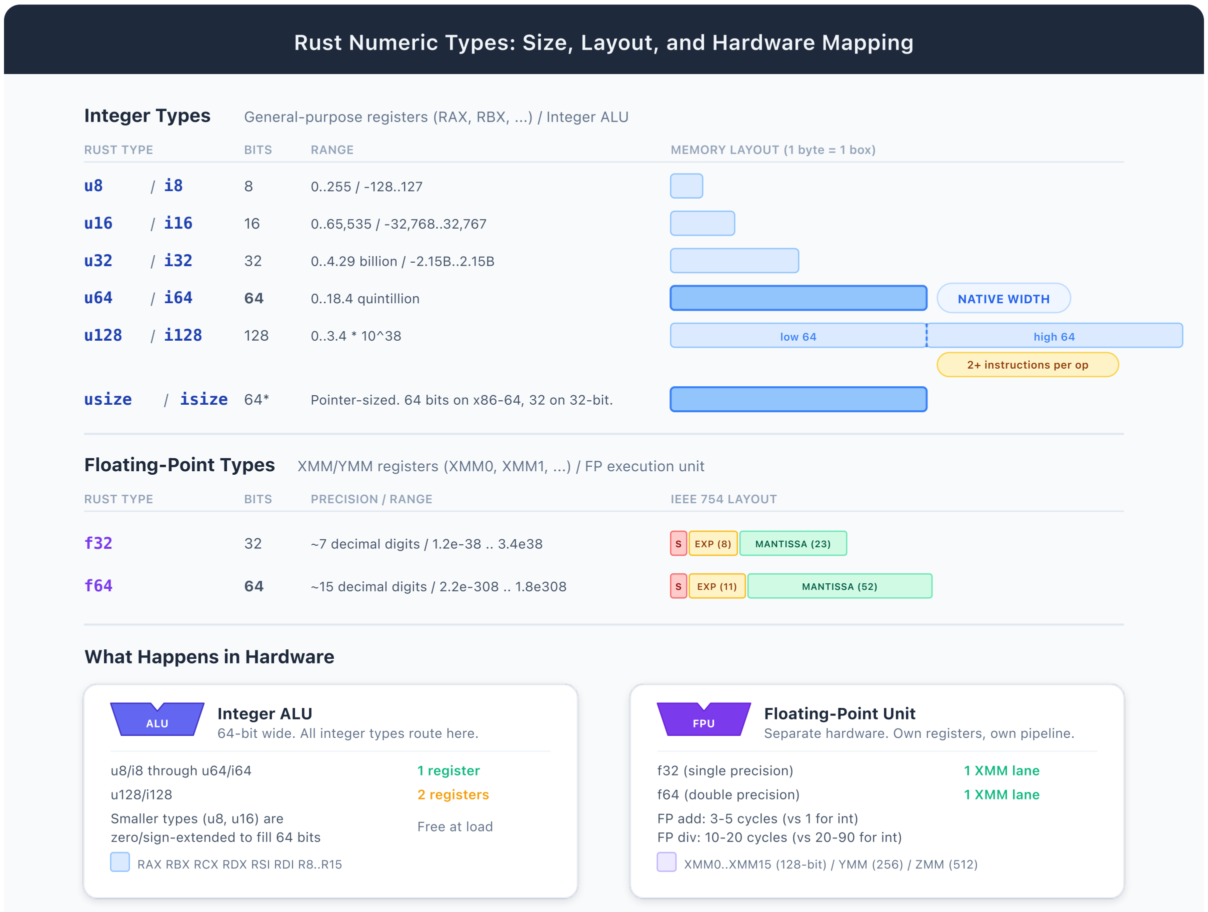
Floating-point multiplication is comparable to integer multiplication. Both take about 3-5 cycles. This surprises people who expect floating-point to always be slower.

Floating-point division is actually faster than integer division in many cases. This is because FPU division uses iterative approximation algorithms (like Newton-Raphson) that converge quickly for the fixed-width mantissa, while integer division must handle the full 64-bit range.

For Rust programmers, this means that switching between `i64` and `f64` is not a simple "integers are always faster" story. It depends on which operations dominate your workload. If your code is mostly addition, integers win. If it's heavy on division, floating-point might actually be faster.

Where Rust Types Meet Hardware

When you choose a numeric type in Rust, you are — whether you know it or not — choosing which execution unit will process your data, which registers will hold it, and how many bits of memory each value occupies.



Rust Numeric Types: Size, Layout, and Hardware Mapping

Integer Types General-purpose registers (RAX, RBX, ...) / Integer ALU

RUST TYPE	BITS	RANGE	MEMORY LAYOUT (1 byte = 1 box)
`u8` / `i8`	8	0..255 / -128..127	
`u16` / `i16`	16	0..65,535 / -32,768..32,767	
`u32` / `i32`	32	0..4.29 billion / -2.15B..2.15B	
`u64` / `i64`	64	0..18.4 quintillion	NATIVE WIDTH
`u128` / `i128`	128	0..3.4 * 10^38	low 64 / high 64; 2+ instructions per op
`usize` / `isize`	64*	Pointer-sized. 64 bits on x86-64, 32 on 32-bit.	

Floating-Point Types XMM/YMM registers (XMM0, XMM1, ...) / FP execution unit

RUST TYPE	BITS	PRECISION / RANGE	IEEE 754 LAYOUT
`f32`	32	~7 decimal digits / 1.2e-38 .. 3.4e38	S, EXP (8), MANTISSA (23)
`f64`	64	~15 decimal digits / 2.2e-308 .. 1.8e308	S, EXP (11), MANTISSA (52)

What Happens in Hardware

Integer ALU
64-bit wide. All integer types route here.

u8/i8 through u64/i64	1 register
u128/i128	2 registers
Smaller types (u8, u16) are zero/sign-extended to fill 64 bits	Free at load

RAX RBX RCX RDX RSI RDI R8..R15

Floating-Point Unit
Separate hardware. Own registers, own pipeline.

f32 (single precision)	1 XMM lane
f64 (double precision)	1 XMM lane

FP add: 3-5 cycles (vs 1 for int)
FP div: 10-20 cycles (vs 20-90 for int)

XMM0..XMM15 (128-bit) / YMM (256) / ZMM (512)

Choosing a numeric type in Rust is choosing which execution unit and registers will process your data.

Integer types

Rust gives you integers from 8 bits to 128 bits, signed and unsigned:

Type	Size	Range
`u8` / `i8`	8 bits (1 byte)	0..255 / -128..127
`u16` / `i16`	16 bits (2 bytes)	0..65,535 / -32,768..32,767
`u32` / `i32`	32 bits (4 bytes)	0..4.29 billion / -2.15B..2.15B
`u64` / `i64`	64 bits (8 bytes)	0..18.4 quintillion

Type	Size	Range
`u128` / `i128`	128 bits (16 bytes)	0..3.4 * 10^38
`usize` / `isize`	pointer-sized	64 bits on x86-64

All of these are processed by the **integer ALU** and stored in **general-purpose registers** (RAX, RBX, RCX, RDX, RSI, RDI, R8 through R15 on x86-64).

The ALU is 64 bits wide. This is the **native width** — the size it was designed to process in a single operation. The `u64` and `i64` types match this width perfectly: one value, one register, one instruction.

Smaller types fit inside the same 64-bit register. When you load a `u8` into a register, the hardware zero-extends it to fill all 64 bits. A `u16` gets zero-extended. An `i8` gets sign-extended (the sign bit is copied into the upper bits). This extension happens at load time and costs nothing. The ALU then operates on the full 64-bit register — it doesn't have a special 8-bit mode. This means that arithmetic on `u8` is not faster than arithmetic on `u64`. The ALU does the same work either way.

So why use smaller types? **Memory.** A `Vec<u8>` with a million elements uses 1 MB. A `Vec<u64>` with a million elements uses 8 MB. When your data lives in arrays, smaller types mean more values fit in a cache line, which can dramatically improve performance. We'll explore this in Chapters 5, 6, and 13.

`u128` and `i128` don't fit in a single register. The x86-64 ALU is 64 bits wide. It cannot add two 128-bit integers in one instruction. The compiler splits every `i128` operation across two registers and multiple instructions — for addition, it emits two instructions (add the lower halves, then add the upper halves with carry). For multiplication, it takes significantly more. This is why `i128` arithmetic is measurably slower than `i64`, even though the language makes them look equivalent.

`usize` and `isize` are pointer-sized: 64 bits on a 64-bit system, 32 bits on a 32-bit system. They are used for indexing into slices and collections. On x86-64, `usize` is identical to `u64` at the hardware level.

Floating-point types

Rust has two floating-point types, both following the IEEE 754 standard:

Type	Size	Precision	Range
`f32`	32 bits	~7 decimal digits	±1.2 * 10^-38 to ±3.4 * 10^38
`f64`	64 bits	~15 decimal digits	±2.2 * 10^-308 to ±1.8 * 10^308

Both are processed by the **floating-point unit** (FPU), not the integer ALU. They live in a completely separate set of registers: the **XMM registers** (XMM0 through XMM15), each 128 bits wide.

The internal layout of a floating-point number is very different from an integer. An `f32` is split into three fields: 1 **sign bit**, 8 **exponent bits**, and 23 **mantissa bits** (also called the significand). An `f64` has 1 sign bit, 11 exponent bits, and 52 mantissa bits. This structure is why floating-point arithmetic requires specialized hardware — the ALU must align exponents, operate on mantissas, normalize the result, and handle rounding, all in a pipeline that takes 3-5 cycles instead of 1.

`f32` vs `f64` performance. On modern x86-64 hardware, scalar `f32` and `f64` operations have the same latency — the FPU processes both at the same speed. The advantage of `f32` is not faster single operations but **SIMD throughput**: a 256-bit YMM register can hold 8 `f32` values but only 4 `f64` values. If your code vectorizes, `f32` can be 2x faster. If it doesn't vectorize, `f32` and `f64` perform identically. Use `f64` by default for precision, and switch to `f32` only when SIMD throughput matters and the reduced precision is acceptable.

The type choice checklist

When choosing a numeric type, you are making three hardware decisions at once:

1. **Execution unit.** Integer types go to the ALU. Floating-point types go to the FPU. They have different latency profiles.
2. **Register file.** Integer types use general-purpose registers. Floating-point types use XMM/YMM registers. These are physically separate — using both means you can keep more values "hot" simultaneously.
3. **Memory footprint.** Smaller types mean more values per cache line. In data-intensive code, this matters more than the operation cost.

SIMD: One Instruction, Many Results

So far, every ALU operation we've discussed works on a single pair of values. Feed in two 64-bit integers, get back one 64-bit result. This is called **scalar** execution.

But what if you need to add four pairs of numbers? In scalar mode, that's four separate ADD instructions, four trips through the ALU, four clock cycles.

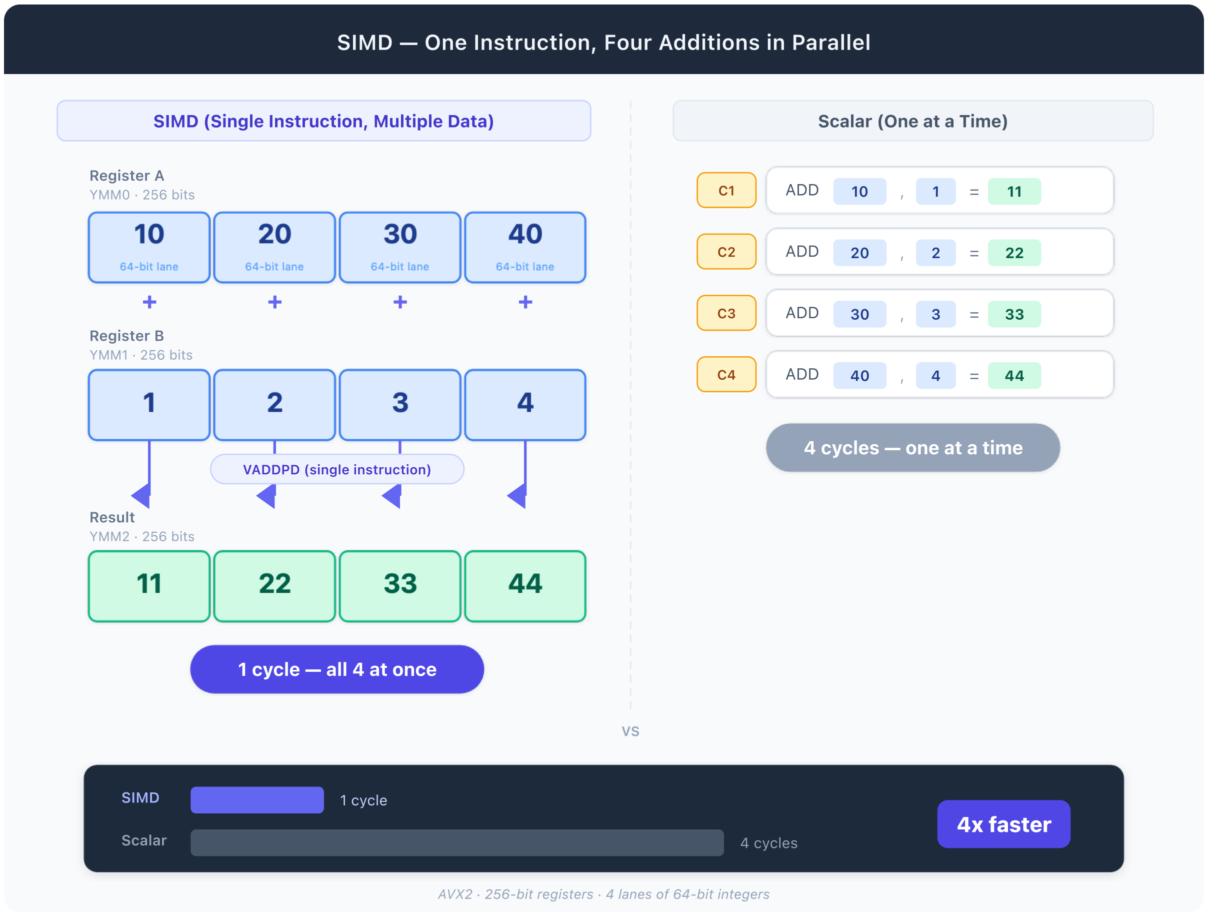
SIMD — Single Instruction, Multiple Data — is an extension that lets the processor perform the same operation on multiple data elements simultaneously. Instead of adding two 64-bit integers, a single SIMD instruction can add:

- Two 64-bit integers at once (128-bit registers, SSE2)
- Four 64-bit integers at once (256-bit registers, AVX2)
- Eight 64-bit integers at once (512-bit registers, AVX-512)

Or, working with smaller elements:

- Sixteen 8-bit integers at once (128-bit registers)
- Thirty-two 8-bit integers at once (256-bit registers)

The SIMD execution units are essentially wider versions of the regular ALU. Instead of a single 64-bit adder, they have a 256-bit or 512-bit adder that is partitioned into independent lanes. Each lane performs the same operation on its own data, completely independently, in the same clock cycle.



Here's what SIMD looks like in Rust using the `std::arch` intrinsics (don't worry about the syntax — we'll cover this in detail in Chapter 15):

```
use std::arch::x86_64::*;

unsafe fn add_four_f64(a: &[f64; 4], b: &[f64; 4]) -> [f64; 4] {
    // Load 4 f64 values into a single 256-bit register
    let va = _mm256_loadu_pd(a.as_ptr());
    let vb = _mm256_loadu_pd(b.as_ptr());

    // Add all 4 pairs in a single instruction
    let result = _mm256_add_pd(va, vb);

    // Store the result back
    let mut out = [0.0f64; 4];
    _mm256_storeu_pd(out.as_mut_ptr(), result);
    out
}
```

One instruction — `_mm256_add_pd` — does the work of four scalar additions. This is a 4x theoretical speedup for the addition itself.

In practice, you often don't need to write SIMD intrinsics by hand. The Rust compiler, through LLVM, can automatically **vectorize** loops when it detects the pattern. A simple loop like this:

```rust
fn add_slices(a: &[f64], b: &[f64], result: &mut [f64]) {
    for i in 0..a.len() {
        result[i] = a[i] + b[i];
    }
}
```

...will often be compiled into SIMD instructions automatically (in release mode). The compiler sees that each iteration is independent, that the operation is the same, and that the data is contiguous in memory — the perfect conditions for SIMD.

But auto-vectorization is fragile. It fails silently when the compiler can't prove the transformation is safe. If the loop body has branches, dependencies between iterations, or non-contiguous memory access, the compiler falls back to scalar code without telling you. In Chapter 15, we'll learn how to check whether vectorization happened and what to do when it doesn't.

The SIMD Hierarchy on x86-64

Over the years, Intel and AMD have introduced progressively wider SIMD instruction sets:

Instruction set	Register width	Year introduced
SSE2	128-bit (XMM)	2001
AVX	256-bit (YMM)	2011
AVX2	256-bit (YMM) + integer	2013
AVX-512	512-bit (ZMM)	2017

Every x86-64 processor supports SSE2 — it's part of the base specification. AVX2 is supported by virtually all processors from 2015 onward. AVX-512 has more limited support and comes with caveats (some processors reduce their clock speed when executing AVX-512 instructions, partially offsetting the gains).

For Rust, the default compilation target assumes only SSE2. If you want the compiler to use AVX2 for auto-vectorization, you need to tell it:

```
RUSTFLAGS="-C target-cpu=native" cargo build --release
```

This tells LLVM to use whatever instruction sets your current CPU supports. We'll revisit this in Chapter 12 and Chapter 15.

The ALU on ARM64: A Different Design Philosophy

Everything we've discussed so far has been x86-64 — the instruction set used by Intel and AMD processors. But if you're developing on Apple Silicon (M1/M2/M3/M4) or deploying to AWS Graviton servers, your code runs on **ARM64** (also called AArch64). The ALU concepts are the same — addition, multiplication, flags, SIMD — but the design philosophy is fundamentally different, and the performance characteristics shift in ways that matter.

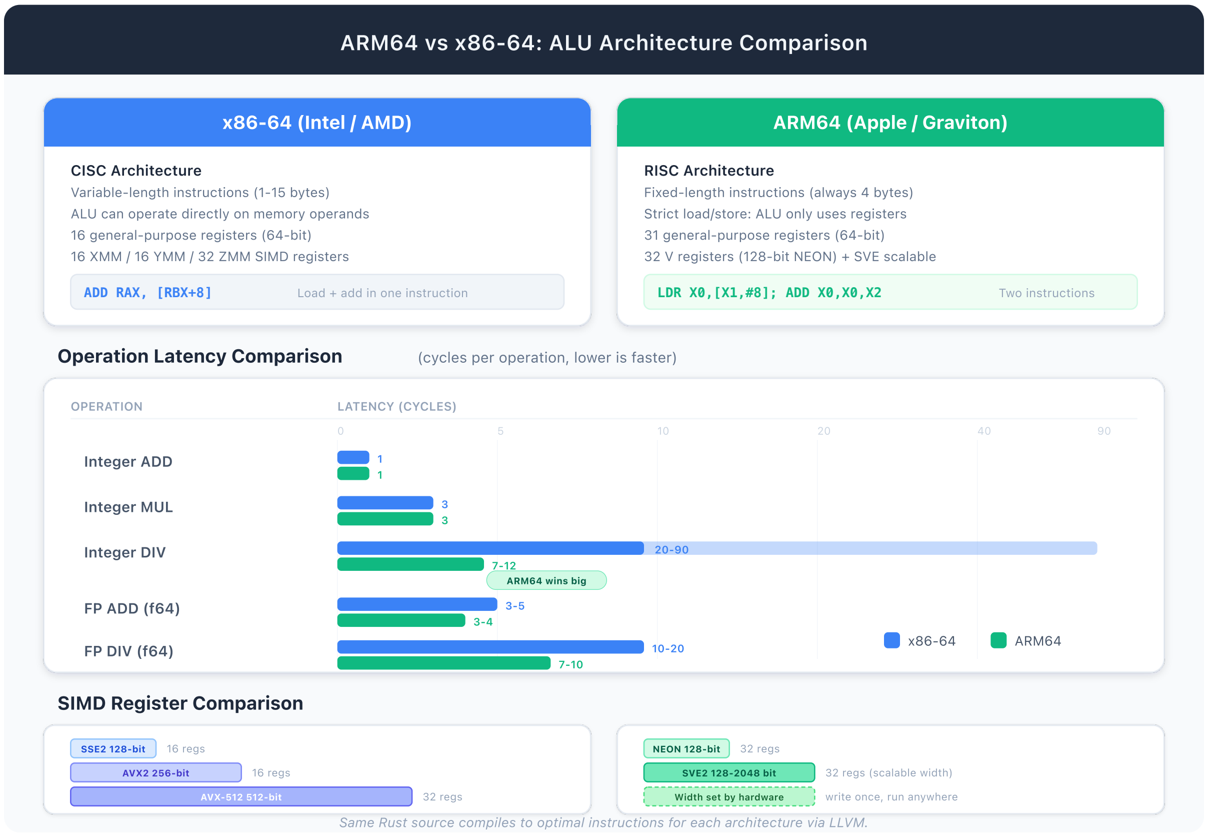
RISC vs. CISC: The Core Difference

x86-64 is a **CISC** (Complex Instruction Set Computer) architecture. Its instructions vary in length from 1 to 15 bytes and can do multiple things at once — a single instruction might load a value from memory, add it to a register, and store the result back. This makes the instruction set powerful but complex. The processor's decode stage must figure out the length and meaning of each variable-width instruction, which requires substantial hardware.

ARM64 is a **RISC** (Reduced Instruction Set Computer) architecture. Every instruction is exactly 4 bytes. Instructions do one thing: an ADD adds two registers. A LOAD loads from memory. A STORE stores to memory. You cannot add a register to a memory location in one instruction — you must load it first, add it, then store it back. This sounds slower on paper, but the simplicity means the processor can decode and execute instructions more efficiently, and the fixed width makes pipelining (Chapter 8) much easier.

For the ALU specifically, this means ARM64 has more instructions to accomplish the same task, but each instruction is simpler and often faster to issue.

ARM64 ALU: What's Different

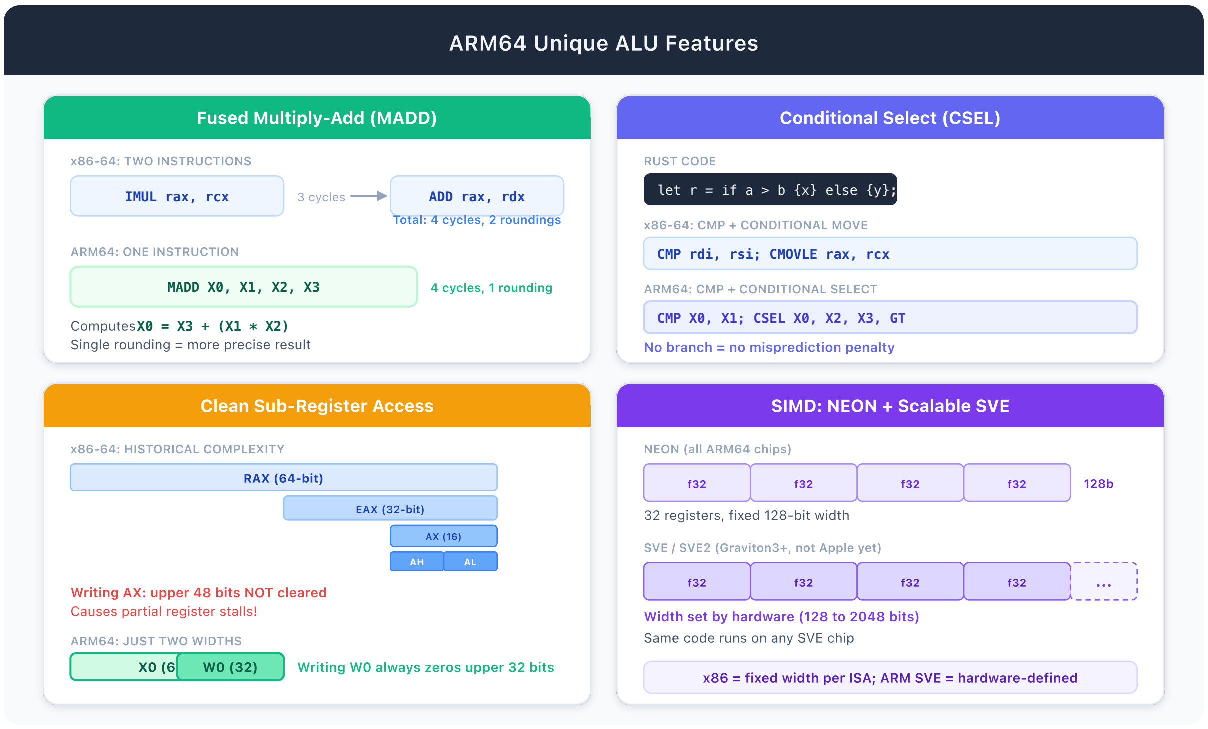

ARM64 vs x86-64: ALU Architecture Comparison
x86-64 (Intel / AMD)
CISC Architecture
Variable-length instructions (1-15 bytes)
ALU can operate directly on memory operands
16 general-purpose registers (64-bit)
16 XMM / 16 YMM / 32 ZMM SIMD registers
Load + add in one instruction
ARM64 (Apple / Graviton)
RISC Architecture
Fixed-length instructions (always 4 bytes)
Strict load/store: ALU only uses registers
31 general-purpose registers (64-bit)
32 V registers (128-bit NEON) + SVE scalable
Two instructions
Operation Latency Comparison
(cycles per operation, lower is faster)
OPERATION
LATENCY (CYCLES)
0
5
10
20
40
90
Integer ADD
1
1
Integer MUL
3
3
Integer DIV
20-90
7-12
ARM64 wins big
FP ADD (f64)
3-5
3-4
FP DIV (f64)
10-20
7-10
x86-64
ARM64
SIMD Register Comparison
SSE2 128-bit
16 regs
AVX2 256-bit
16 regs
AVX-512 512-bit
32 regs
NEON 128-bit
32 regs
SVE2 128-2048 bit
32 regs (scalable width)
Width set by hardware
write once, run anywhere

```
ADD RAX, [RBX+8]
```

```
LDR X0,[X1,#8]; ADD X0,X0,X2
```

Same Rust source compiles to optimal instructions for each architecture via LLVM.

The core arithmetic operations exist on both architectures, but ARM64 has some distinctive features:


ARM64 Unique ALU Features
Fused Multiply-Add (MADD)
x86-64: TWO INSTRUCTIONS
IMUL rax, rcx
3 cycles
ADD rax, rdx
Total: 4 cycles, 2 roundings
ARM64: ONE INSTRUCTION
MADD X0, X1, X2, X3
4 cycles, 1 rounding
Computes X0 = X3 + (X1 * X2)
Single rounding = more precise result
Conditional Select (CSEL)
RUST CODE
let r = if a > b {x} else {y};
x86-64: CMP + CONDITIONAL MOVE
CMP rdi, rsi; CMOVLE rax, rcx
ARM64: CMP + CONDITIONAL SELECT
CMP X0, X1; CSEL X0, X2, X3, GT
No branch = no misprediction penalty
Clean Sub-Register Access
x86-64: HISTORICAL COMPLEXITY
RAX (64-bit)
EAX (32-bit)
AX (16)
AH
AL
Writing AX: upper 48 bits NOT cleared
Causes partial register stalls!
ARM64: JUST TWO WIDTHS
X0 (6
W0 (32)
Writing W0 always zeros upper 32 bits
SIMD: NEON + Scalable SVE
NEON (all ARM64 chips)
f32
f32
f32
f32
128b
32 registers, fixed 128-bit width
SVE / SVE2 (Graviton3+, not Apple yet)
f32
f32
f32
f32
...
Width set by hardware (128 to 2048 bits)
Same code runs on any SVE chip
x86 = fixed width per ISA; ARM SVE = hardware-defined

Fused multiply-add (FMA). ARM64 has a dedicated `MADD` instruction that computes `a + (b * c)` in a single instruction. On x86-64, this requires two separate instructions (a multiply and an add). The FMA instruction isn't just syntactic sugar — the hardware performs both operations with a single rounding step, which is both faster and more numerically precise.

Conditional execution without branches. ARM64 has conditional select instructions (`CSEL` , `CSINC` , `CSNEG`) that choose between two register values based on condition flags — no branch needed. This eliminates branch misprediction penalties for simple conditional assignments. Where x86-64 might use a `CMOV` (conditional move) for `if a > b { x } else { y }` , ARM64 has a richer set of conditional operations.

No partial register complications. ARM64 registers are cleanly 64-bit (X0-X30) or 32-bit (W0-W30, the lower half). Writing to a W register always zeros the upper 32 bits. There's no equivalent of x86-64's historical 8-bit and 16-bit sub-register access that causes partial register stalls.

Bit manipulation built in. ARM64 includes instructions for bit field extraction, bit reversal, population count, and leading/trailing zero count as first-class operations. On x86-64, some of these require special instruction set extensions (like POPCNT or LZCNT).

Operation Latency: x86-64 vs ARM64

The ALU latency story is similar on both architectures, but not identical. Here's a comparison using a modern high-performance core from each:

Operation	x86-64 (Zen 4 / Raptor Lake)	ARM64 (Apple M4 / Cortex-X4)
Integer ADD	1 cycle	1 cycle
Integer MUL (64-bit)	3 cycles	3 cycles

Operation	x86-64 (Zen 4 / Raptor Lake)	ARM64 (Apple M4 / Cortex-X4)
Integer DIV (64-bit)	20-90 cycles	7-12 cycles
FP ADD (f64)	3-5 cycles	3-4 cycles
FP MUL (f64)	3-5 cycles	3-4 cycles
FP DIV (f64)	10-20 cycles	7-10 cycles
Fused multiply-add	4-5 cycles (FMA instr.)	4 cycles (MADD)
Bitwise AND/OR/XOR	1 cycle	1 cycle
Shift	1 cycle	1 cycle

The big surprise: **ARM64 integer division is significantly faster.** While x86-64 division can take up to 90 cycles in the worst case, ARM64 chips (especially Apple Silicon) have invested in faster division hardware, typically completing in 7-12 cycles. This narrows the gap between division and multiplication, though division is still the most expensive arithmetic operation on both architectures.

SIMD on ARM64: NEON and SVE

ARM64 has its own SIMD story, which diverges from x86-64's SSE/AVX lineage:

Feature	x86-64	ARM64
Base SIMD	SSE2 (128-bit)	NEON (128-bit)
Wide SIMD	AVX2 (256-bit)	SVE/SVE2 (128-2048 bit)
Widest SIMD	AVX-512 (512-bit)	SVE2 (up to 2048-bit)
Register count	16 XMM / 32 ZMM	32 V registers (128-bit)
Vector length	Fixed per ISA extension	Scalable (hardware-defined)

NEON is ARM64's base SIMD extension. Every ARM64 processor supports it. It provides 32 registers of 128 bits each — twice as many as x86-64's base SSE2. NEON handles the same kinds of parallel operations: add four `f32` values at once, compare sixteen `u8` values at once, etc.

SVE and SVE2 (Scalable Vector Extension) are ARM's answer to AVX-512, but with a twist. Instead of fixing the vector width at 256 or 512 bits, SVE defines the vector length as an implementation detail. A chip might implement 128-bit, 256-bit, or 2048-bit SVE registers — your code doesn't need to know. You write one loop, and it adapts to whatever hardware width is available. This is a fundamentally different approach from x86-64, where you must target SSE2, AVX2, or AVX-512 explicitly.

Apple Silicon currently implements NEON but not SVE. AWS Graviton3 and newer implement SVE2 with 256-bit vectors. The Rust compiler can auto-vectorize for NEON just as it does for SSE2/AVX2.

For Rust on ARM64:

```
# On Apple Silicon, enable NEON (enabled by default)
RUSTFLAGS="-C target-cpu=native" cargo build --release

# On Graviton3, this also enables SVE2
RUSTFLAGS="-C target-cpu=neoverse-v1" cargo build --release
```

The Full Comparison

Here's a comprehensive side-by-side of ALU-related features that matter for Rust programmers:

Feature	x86-64 (Intel/AMD)	ARM64 (Apple/Graviton)
Design philosophy	CISC (complex, variable-length)	RISC (simple, fixed-length)
Instruction width	1-15 bytes	Always 4 bytes
General-purpose regs	16 (64-bit)	31 (64-bit)
SIMD registers	16 XMM / 32 ZMM	32 V (128-bit)
Integer ADD	1 cycle	1 cycle
Integer MUL	3 cycles	3 cycles
Integer DIV	20-90 cycles	7-12 cycles
FP ADD/MUL	3-5 cycles	3-4 cycles
FP DIV	10-20 cycles	7-10 cycles
Fused multiply-add	FMA3 extension (2013+)	Built-in (MADD/MSUB)
Conditional select	CMOV (limited)	CSEL, CSINC, CSNEG
Base SIMD width	128-bit (SSE2)	128-bit (NEON)
Max SIMD width	512-bit (AVX-512)	Scalable (SVE2)
Bit manipulation	Extensions (BMI1/BMI2)	Built-in
Load-store discipline	Memory operands in ALU ops	Strict load/store separation
Energy efficiency	Moderate	High

What This Means for Rust

The good news: **you usually don't need to care.** The Rust compiler (through LLVM) generates optimal code for whichever architecture you're targeting. The same Rust source compiles to different instruction sequences on x86-64 and ARM64, each tuned to the target's strengths.

But there are cases where the architecture matters:

1. **Division-heavy code** runs faster on ARM64. If profiling shows division as a hotspot, ARM64's faster divider can help. But the best optimization is still to avoid division entirely (use shifts, multiplicative inverses).

2. **SIMD intrinsics are architecture-specific.** If you write `std::arch::x86_64::_mm256_add_pd`, that code won't compile on ARM64. Use `std::simd` (the portable SIMD API) or write separate implementations behind `#[cfg(target_arch)]` attributes.

3. **Register pressure is lower on ARM64.** With 31 general-purpose registers (vs. 16), the compiler has almost twice as much room. Complex functions that spill registers on x86-64 may run entirely in registers on ARM64. We'll see how this matters in Chapter 4.

4. **Branch-heavy code may differ.** ARM64's conditional select instructions can eliminate branches that x86-64 must predict. This can make code with unpredictable conditions (like sorting comparisons) faster on ARM64. We'll explore branch prediction deeply in Chapter 9.

Putting It All Together

Let's trace a simple Rust expression through the hardware to see the ALU in action:

```rust
let x: i64 = a + b * c;
```

Assuming `a`, `b`, and `c` are already in registers (we'll discuss how they get there in Chapter 4), the processor does:

1. **Multiply** `b * c` — send `b` and `c` to the integer ALU with the MUL opcode. This takes 3 cycles. The result goes into a temporary register.
2. **Add** `a + (b * c)` — send `a` and the multiplication result to the ALU with the ADD opcode. This takes 1 cycle. The result goes into the register assigned to `x`.

Total: 4 cycles for the entire expression. At 4 GHz, that's 1 nanosecond.

But this is a simplified view. In reality, on a modern out-of-order processor, the multiply might start before the previous instruction has finished. The add has to wait for the multiply's result (this is a **data dependency**), but other independent instructions can execute during those 3 cycles of waiting. We'll explore this in Chapter 10.

Now consider a different expression:

```rust
let y: i64 = a / b + c / d;
```

Two divisions. Each takes 20-90 cycles. But here's the interesting part: `a / b` and `c / d` are **independent** — neither needs the other's result. A modern out-of-order processor can execute both divisions simultaneously on separate execution units. The total time is the latency of one division (not two), plus one cycle for the addition. We'll see exactly how this works in Chapter 10.

What the ALU Tells Us About Performance

This chapter introduced the core computational engine of the processor. Here are the takeaways that will matter throughout the rest of the book:

Not all operations cost the same. Addition is 1 cycle. Multiplication is 3. Division is 20-90. This ratio is baked into the hardware and hasn't fundamentally changed in decades. Compilers know this and optimize aggressively to avoid expensive operations, but there are limits to what a compiler can do.

Integer and floating-point are separate worlds. They use different execution units, different registers, and have different performance profiles. Choosing between `i64` and `f64` is not just a precision decision — it's a hardware decision.

SIMD multiplies throughput. The same ALU principles apply to SIMD, but operating on 2, 4, 8, or more values simultaneously. This is the single biggest performance multiplier available in modern hardware, but it requires the right data layout and access patterns to be effective.

The ALU is almost never the bottleneck. This is perhaps the most important point. The ALU can perform billions of operations per second. What limits real-world performance is almost always *feeding the ALU fast enough* — getting data from memory into registers where the ALU can operate on it. The memory hierarchy, which we'll explore in Chapters 5 and 6, is where most performance is won or lost.

The ALU is the engine. But an engine is useless without fuel. In the next chapter, we'll look at the registers — the tiny, blazing-fast memory cells that sit right next to the ALU and hold the data it's working on.

Chapter 13: Data Layout and Cache Performance

In Part I, we learned that the processor doesn't read individual bytes — it reads 64-byte **cache lines** (Chapter 5), that the L1 cache is 1000x faster than main memory (Chapter 5), and that cache misses stall the entire out-of-order engine (Chapter 10). These aren't abstract facts. They're the most important determinants of real-world performance for data-intensive Rust code.

This chapter is about one thing: **how you arrange your data in memory determines how fast your program runs.** The exact same algorithm, operating on the exact same values, can run 10x faster or slower depending on how the data is laid out. The processor doesn't care about your type system or your abstractions — it cares about which bytes are adjacent in memory and whether the next access hits the cache or misses it.

We'll start with how Rust lays out structs in memory, then move to the fundamental choice between Array of Structs and Struct of Arrays, and finish with practical techniques for restructuring data to make the cache work for you instead of against you.

How Rust Lays Out Structs

Every struct in Rust has three properties that affect its memory footprint: **size**, **alignment**, and **padding**. These are not abstractions — they're physical facts about where bytes go in memory, and they directly determine how many useful bytes fit in each 64-byte cache line.

Alignment

Every type has an **alignment requirement** — the address where it can be placed must be a multiple of its alignment. This isn't arbitrary: the hardware loads data in aligned chunks. An unaligned load may cross a cache line boundary, requiring two cache accesses instead of one and potentially two cycles instead of one.

Type	Size (bytes)	Alignment (bytes)
`u8` / `i8` / `bool`	1	1
`u16` / `i16`	2	2
`u32` / `i32` / `f32`	4	4
`u64` / `i64` / `f64`	8	8
`u128` / `i128`	16	16
`usize` / `isize` / `*const T`	8	8
`&T` (thin reference)	8	8
`&[T]` (slice)	16	8
`&dyn Trait`	16	8

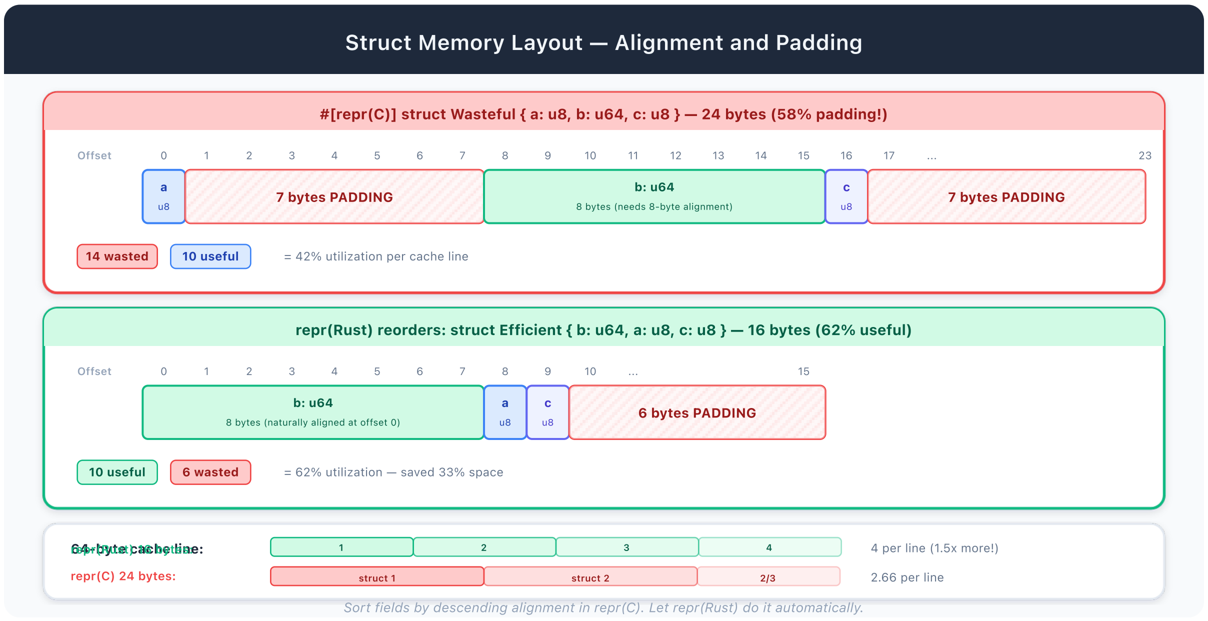
A struct's alignment equals the largest alignment of any of its fields. A struct containing a `u64` field has 8-byte alignment, regardless of what other fields it contains.

Padding

When a struct has fields with different alignments, the compiler inserts **padding bytes** between fields to maintain alignment. These padding bytes occupy space but carry no data — they're wasted cache line capacity.

```
struct Wasteful {
    a: u8,      // 1 byte at offset 0
    // 7 bytes of padding (b needs 8-byte alignment)
    b: u64,     // 8 bytes at offset 8
    c: u8,      // 1 byte at offset 16
    // 7 bytes of padding (struct alignment = 8)
}
// Total: 24 bytes (but only 10 bytes of actual data!)
```

That's 14 bytes of padding — 58% waste. In a `Vec<Wasteful>` with a million elements, you're using 24 MB of memory for 10 MB of data, and every cache line carries 37% less useful information.



Struct Memory Layout — Alignment and Padding
#[repr(C)] struct Wasteful { a: u8, b: u64, c: u8 } — 24 bytes (58% padding!)
Offset
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 ... 23
a
u8
7 bytes PADDING
b: u64
8 bytes (needs 8-byte alignment)
c
u8
7 bytes PADDING
14 wasted
10 useful
= 42% utilization per cache line
repr(Rust) reorders: struct Efficient { b: u64, a: u8, c: u8 } — 16 bytes (62% useful)
Offset
0 1 2 3 4 5 6 7 8 9 10 ... 15
b: u64
8 bytes (naturally aligned at offset 0)
a
u8
c
u8
6 bytes PADDING
10 useful
6 wasted
= 62% utilization — saved 33% space
64-byte cache line:
repr(Rust) 16 bytes:
1 2 3 4
4 per line (1.5x more!)
repr(C) 24 bytes:
struct 1
struct 2
2/3
2.66 per line
Sort fields by descending alignment in repr(C). Let repr(Rust) do it automatically.

Rust's Default Layout: Reordering for Efficiency

Unlike C, Rust's default struct layout (`repr(Rust)`) does **not** guarantee field order in memory. The compiler is free to reorder fields to minimize padding. This is one of Rust's quiet advantages:

```
// You write:
struct Efficient {
    a: u8,
    b: u64,
    c: u8,
}

// Rust may lay this out as:
// b: u64 at offset 0  (8 bytes)
// a: u8  at offset 8  (1 byte)
// c: u8  at offset 9  (1 byte)
// 6 bytes padding to reach alignment of 8
// Total: 16 bytes (not 24!)
```

By placing the `u64` first, the compiler eliminates the 7-byte gap that C would create. The struct is 33% smaller.

You can verify layout with `std::mem`:

```
use std::mem;

println!("size:  {}", mem::size_of::<Efficient>());    // 16
println!("align: {}", mem::align_of::<Efficient>());   // 8
```

repr(C): When You Need C-Compatible Layout

`#[repr(C)]` forces C-compatible layout: fields are laid out in declaration order, with padding inserted as a C compiler would. You need this for FFI, memory-mapped I/O, and any situation where you must control exact byte positions:

```
#[repr(C)]
struct CLayout {
    a: u8,      // offset 0
    // 7 bytes padding
    b: u64,     // offset 8
    c: u8,      // offset 16
    // 7 bytes padding
}
// size = 24, same as C would produce
```

When using `repr(C)`, field order matters. Sort fields by descending alignment to minimize padding:

```
#[repr(C)]
struct CLayoutOptimized {
    b: u64,     // offset 0  (8 bytes, alignment 8)
    a: u8,      // offset 8  (1 byte, alignment 1)
    c: u8,      // offset 9  (1 byte, alignment 1)
    // 6 bytes padding
}
// size = 16 — same data, 33% smaller
```

The rule: **fields with larger alignment go first.** This is the manual version of what `repr(Rust)` does automatically.

Checking Layout at Compile Time

For performance-critical structs, assert the size to catch accidental regressions:

```
const _: () = assert!(std::mem::size_of::<MyStruct>() == 64);
// Compile error if someone adds a field that changes the size
```

You can also use the `#[repr(align(N))]` attribute to force a minimum alignment, typically to ensure a struct occupies exactly one cache line:

```
#[repr(align(64))]
struct CacheAligned {
    data: [u8; 48],
    counter: u64,
}
// Size: 64 bytes, aligned to cache line boundary
// Each element in Vec<CacheAligned> starts at a cache line boundary
```

The Cache Line Budget

A 64-byte cache line is your fundamental budget. Every field that ends up in the same cache line as the data you're accessing comes "for free" — it's already loaded. Every field in a different cache line costs an additional memory access.

Let's make this concrete. Consider iterating over a million structs:

```
struct Player {
    name: String,       // 24 bytes (ptr + len + capacity)
    id: u64,            // 8 bytes
    x: f32,             // 4 bytes
    y: f32,             // 4 bytes
    z: f32,             // 4 bytes
    health: u16,        // 2 bytes
    team: u8,           // 1 byte
    active: bool,       // 1 byte
    // padding to alignment
}
// Total: ~48 bytes (depends on Rust's reordering)
```

If your physics update only uses `x`, `y`, `z`, and `active`, it needs 13 bytes per player. But each cache line load brings in 48 bytes (the entire struct). That's 27% useful data per cache line, and you need at least one cache line per player — likely two when structs straddle boundaries.

With a million players: you're loading 48 MB from memory when you only need 13 MB. The extra 35 MB flows through the cache hierarchy, evicting other useful data and consuming memory bandwidth, all for fields you're not reading.

This is the core problem. Let's fix it.

Array of Structs vs. Struct of Arrays

The most impactful data layout decision in performance-critical code is the choice between **Array of Structs (AoS)** and **Struct of Arrays (SoA)**.

Array of Structs (AoS): The Default

This is how most code is naturally written — a collection of objects, each containing all their fields:

```rust
// AoS: each particle has all its data together
struct Particle {
    x: f32,
    y: f32,
    z: f32,
    vx: f32,
    vy: f32,
    vz: f32,
    mass: f32,
    charge: f32,
}
// Size: 32 bytes per particle

let particles: Vec<Particle> = vec![/* ... */; 1_000_000];
```

Memory layout of `Vec<Particle>`:

```
Cache line 0: [x0 y0 z0 vx0 vy0 vz0 mass0 charge0] [x1 y1 z1 vx1 vy1 vz1 mass1
charge1]
Cache line 1: [x2 y2 z2 vx2 vy2 vz2 mass2 charge2] [x3 y3 z3 vx3 vy3 vz3 mass3
charge3]
...
```

Two particles per cache line (32 bytes each). If your update function reads all 8 fields, this is perfect — 100% of each cache line is useful. But if you only need positions (`x, y, z`):

```rust
// Only need 12 bytes per particle, but load 32
fn total_distance(particles: &[Particle]) -> f32 {
    particles.iter()
        .map(|p| (p.x * p.x + p.y * p.y + p.z * p.z).sqrt())
        .sum()
}
```

Each cache line carries 24 bytes of position data (12 per particle × 2 particles) and 40 bytes of velocity/mass/charge that you don't need. That's 37.5% utilization — the prefetcher and memory bus work 2.7x harder than necessary.

Struct of Arrays (SoA): Fields Stored Separately

SoA groups each field into its own contiguous array:

```rust
// SoA: each field in its own array
struct Particles {
    x: Vec<f32>,
    y: Vec<f32>,
    z: Vec<f32>,
    vx: Vec<f32>,
    vy: Vec<f32>,
    vz: Vec<f32>,
    mass: Vec<f32>,
    charge: Vec<f32>,
}

let particles = Particles {
    x: vec![0.0; 1_000_000],
    y: vec![0.0; 1_000_000],
    z: vec![0.0; 1_000_000],
    // ...
};
```

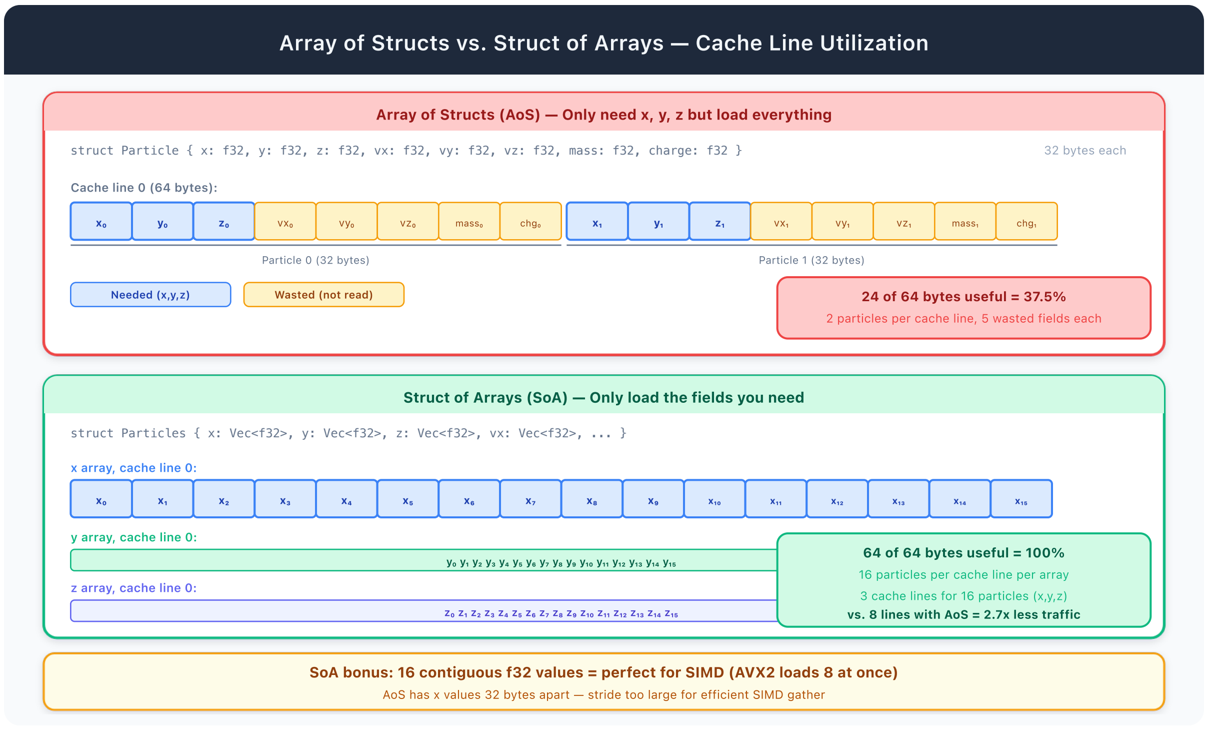
Memory layout:

```
x array:  [x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15] ...
y array:  [y0 y1 y2 y3 y4 y5 y6 y7 y8 y9 y10 y11 y12 y13 y14 y15] ...
z array:  [z0 z1 z2 z3 z4 z5 z6 z7 z8 z9 z10 z11 z12 z13 z14 z15] ...
```

Now the distance computation:

```rust
fn total_distance(p: &Particles) -> f32 {
    p.x.iter().zip(p.y.iter()).zip(p.z.iter())
        .map(|((&x, &y), &z)| (x * x + y * y + z * z).sqrt())
        .sum()
}
```

Each cache line in the `x` array holds 16 `f32` values — all `x` coordinates, all useful. Same for `y` and `z`. You load 3 cache lines to process 16 particles (192 bytes for 192 bytes of useful data). With AoS, you'd load 8 cache lines for the same 16 particles (512 bytes for 192 bytes of useful data).


Array of Structs vs. Struct of Arrays — Cache Line Utilization
Array of Structs (AoS) — Only need x, y, z but load everything
struct Particle { x: f32, y: f32, z: f32, vx: f32, vy: f32, vz: f32, mass: f32, charge: f32 }
32 bytes each
Cache line 0 (64 bytes):
x_0 y_0 z_0 vx_0 vy_0 vz_0 $mass_0$ chg_0
x_1 y_1 z_1 vx_1 vy_1 vz_1 $mass_1$ chg_1
Particle 0 (32 bytes)
Particle 1 (32 bytes)
Needed (x,y,z)
Wasted (not read)
24 of 64 bytes useful = 37.5%
2 particles per cache line, 5 wasted fields each
Struct of Arrays (SoA) — Only load the fields you need
struct Particles { x: Vec<f32>, y: Vec<f32>, z: Vec<f32>, vx: Vec<f32>, ... }
x array, cache line 0:
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13} x_{14} x_{15}
y array, cache line 0:
y_0 y_1 y_2 y_3 y_4 y_5 y_6 y_7 y_8 y_9 y_{10} y_{11} y_{12} y_{13} y_{14} y_{15}
z array, cache line 0:
z_0 z_1 z_2 z_3 z_4 z_5 z_6 z_7 z_8 z_9 z_{10} z_{11} z_{12} z_{13} z_{14} z_{15}
64 of 64 bytes useful = 100%
16 particles per cache line per array
3 cache lines for 16 particles (x,y,z)
vs. 8 lines with AoS = 2.7x less traffic
SoA bonus: 16 contiguous f32 values = perfect for SIMD (AVX2 loads 8 at once)
AoS has x values 32 bytes apart — stride too large for efficient SIMD gather

The speedup from SoA is directly proportional to the fraction of fields you access per operation. If you access all fields, AoS and SoA perform similarly. If you access 2 of 8 fields, SoA can be 3-4x faster.

SoA Enables SIMD

There's a second, equally important advantage: SoA layout is naturally **SIMD-friendly** (Chapter 15). With 16 consecutive `f32` values in a cache line, the compiler can load 4 (SSE), 8 (AVX2), or 16 (AVX-512) values into a SIMD register with a single aligned load instruction and process them all simultaneously.

```
// SoA: the compiler auto-vectorizes this easily
// Processes 8 x-values at once with AVX2
fn scale_positions(p: &mut Particles, factor: f32) {
    for x in p.x.iter_mut() {
        *x *= factor;
    }
}
```

With AoS, the `x` values are 32 bytes apart (the stride of the struct). SIMD gather instructions exist for strided access, but they're 3-5x slower than contiguous loads. The compiler typically won't auto-vectorize strided access at all.

When to Use AoS vs. SoA

Scenario	Better layout	Why
Access all fields per element	AoS	All data for one element in 1-2 cache lines
Access few fields across many elements	SoA	Only load the fields you need
SIMD processing	SoA	Contiguous data for vector loads
Random access by index	AoS	One element = one cache line region
Sorted/shuffled by one field	SoA	Sort one array, rearrange others
Serialization/FFI	AoS (repr(C))	Matches external data formats
Small structs (≤ 16 bytes)	AoS	Struct fits in cache line fraction

The hybrid approach — **Array of Struct of Arrays (AoSoA)** — groups data into small blocks (e.g., 8 or 16 elements), with each block using SoA layout internally:

```rust
// AoSoA: blocks of 8 particles, SoA within each block
struct ParticleBlock {
    x: [f32; 8],
    y: [f32; 8],
    z: [f32; 8],
    vx: [f32; 8],
    vy: [f32; 8],
    vz: [f32; 8],
    mass: [f32; 8],
    charge: [f32; 8],
}
// Size: 256 bytes = 4 cache lines
// x,y,z for 8 particles: 96 bytes (fits in 2 cache lines)
// SIMD-width aligned for AVX2

let blocks: Vec<ParticleBlock> = vec![/* ... */; 125_000]; // 1M / 8
```

AoSoA gives you SIMD-friendly contiguous data within each block, while keeping related data (all fields for 8 particles) close together in memory. This is the layout used by high-performance physics engines and game ECS frameworks.

Practical Patterns for Cache-Friendly Rust

Pattern 1: Hot/Cold Splitting

If some fields are accessed frequently and others rarely, split them into separate structs:

```rust
// Before: one big struct
struct Entity {
    // Hot: used every frame
    x: f32,
    y: f32,
    velocity_x: f32,
    velocity_y: f32,
    // Cold: used occasionally
    name: String,
    description: String,
    creation_time: u64,
    metadata: HashMap<String, String>,
}
// ~120+ bytes per entity

// After: split hot and cold
struct EntityTransform {
    x: f32,
    y: f32,
    velocity_x: f32,
    velocity_y: f32,
}
// 16 bytes — 4 entities per cache line!

struct EntityInfo {
    name: String,
    description: String,
    creation_time: u64,
    metadata: HashMap<String, String>,
}

struct World {
    transforms: Vec<EntityTransform>,   // hot: iterated every frame
    info: Vec<EntityInfo>,              // cold: accessed by index on demand
}
```

The physics update now iterates over a dense `Vec<EntityTransform>` at 4 entities per cache line instead of wading through Strings and HashMaps it doesn't need. Cache utilization goes from ~13% to ~100% for the hot path.

Pattern 2: Indices Instead of Pointers

Pointers (`Box<T>` , `&T`) point to arbitrary heap locations. Following a pointer is a random memory access — the prefetcher can't predict where the next element lives. Indices into a `Vec` maintain contiguity:

```
// Pointer-based: random memory access pattern
struct TreeNode {
    value: i64,
    left: Option<Box<TreeNode>>,
    right: Option<Box<TreeNode>>,
}
// Each node at an unpredictable heap address
// Traversal: pointer chase → cache miss per node

// Index-based: nodes stored contiguously
struct Arena {
    nodes: Vec<ArenaNode>,
}

struct ArenaNode {
    value: i64,
    left: Option<u32>,   // index into arena.nodes
    right: Option<u32>,  // index into arena.nodes
}
// Size: 16 bytes per node (vs 24+ for Box version)
// Nodes are contiguous in memory
// BFS traversal often hits the same cache lines
```

The arena approach stores all nodes in a single `Vec`, so they occupy contiguous memory. A breadth-first traversal accesses nodes roughly sequentially — the prefetcher handles this well. The pointer-based tree sends the processor on a random walk through the heap, missing the cache on nearly every access.

The performance difference is dramatic for large trees:

Tree size	Pointer-based traversal	Arena-based traversal	Speedup
1K nodes	~15 μs	~4 μs	3.7x
100K nodes	~3.8 ms	~0.4 ms	9.5x
10M nodes	~850 ms	~45 ms	19x

The speedup increases with size because the pointer-based tree's random accesses exceed the cache capacity, while the arena's sequential accesses stay prefetcher-friendly.

Pattern 3: Shrink Your Structs

Every byte you remove from a hot struct packs more elements per cache line. Common techniques:

```rust
// Before: 24 bytes
struct GameUnit {
    health: u32,       // 0-1000 range
    mana: u32,         // 0-500 range
    x: f32,
    y: f32,
    unit_type: u64,    // only 12 variants
}

// After: 12 bytes — 2x cache density!
struct GameUnit {
    health: u16,       // 0-65535, plenty for 0-1000
    mana: u16,         // 0-65535, plenty for 0-500
    x: f32,
    y: f32,
    unit_type: u8,     // 256 variants, plenty for 12
    // 3 bytes padding (alignment of f32 = 4)
}
```

Using smaller types where the value range permits halves the struct size. A `Vec<GameUnit>` now fits twice as many units per cache line, and the iteration speed nearly doubles for cache-bound workloads.

Be mindful of the trade-off: `u16` arithmetic on x86-64 requires a `movzx` (zero-extend) instruction when used in expressions that expect `u32` / `u64`. On ARM64, the extension is typically free (wrapped into the next instruction). For hot loops, profile to verify the smaller type is actually faster.

Pattern 4: Iteration Order Matters

For multi-dimensional data, iteration order determines whether you access memory sequentially (cache-friendly) or with large strides (cache-hostile):

```rust
let matrix: Vec<Vec<f64>> = vec![vec![0.0; 1000]; 1000];

// Row-major iteration: sequential access
// Each row is contiguous in memory → prefetcher friendly
fn sum_row_major(matrix: &[Vec<f64>]) -> f64 {
    let mut sum = 0.0;
    for row in matrix {
        for &val in row {
            sum += val;
        }
    }
    sum
}

// Column-major iteration: strided access
// Jumps 8000 bytes between accesses (1000 * 8 bytes per f64)
fn sum_col_major(matrix: &[Vec<f64>]) -> f64 {
    let mut sum = 0.0;
    let cols = matrix[0].len();
    for col in 0..cols {
        for row in matrix {
            sum += row[col];    // stride = 8000 bytes!
        }
    }
    sum
}
```

On a 1000×1000 matrix of `f64`:

Order	Cache behavior	Time
Row-major	Sequential: 8 values per cache line, prefetcher active	~0.3 ms
Column-major	Strided: 1 value per cache line, prefetcher confused	~2.5 ms

The factor of ~8x matches the cache line utilization: row-major uses 8/8 of each loaded `f64` values (100%), while column-major uses 1/8 (12.5%).

Rust's `Vec<Vec<f64>>` stores each inner Vec as a separate heap allocation, so even row-major access has one indirection per row. For maximum performance, use a flat `Vec<f64>` with manual indexing:

```rust
struct Matrix {
    data: Vec<f64>,
    rows: usize,
    cols: usize,
}

impl Matrix {
    fn get(&self, row: usize, col: usize) -> f64 {
        self.data[row * self.cols + col]
    }

    // Iterate in memory order (row-major)
    fn sum(&self) -> f64 {
        self.data.iter().sum()  // one contiguous slice!
    }
}
```

Now the entire matrix is one contiguous allocation. The `sum()` call iterates a single `&[f64]` slice — the simplest possible access pattern for the prefetcher and the auto-vectorizer.

Pattern 5: Filtering Without Allocation

Iterator chains that filter elements avoid materializing intermediate collections — data flows through the pipeline element by element, touching each cache line once:

```
// Good: lazy evaluation, single pass, no allocation
let result: f64 = particles.x.iter()
    .zip(particles.y.iter())
    .zip(particles.mass.iter())
    .filter(|&(_, &m)| m > 0.0)
    .map(|((&x, &y), _)| (x * x + y * y).sqrt())
    .sum();

// Bad: materializes intermediate Vec, touches data twice
let filtered: Vec<(f32, f32)> = particles.x.iter()
    .zip(particles.y.iter())
    .zip(particles.mass.iter())
    .filter(|&(_, &m)| m > 0.0)
    .map(|((&x, &y), _)| (x, y))
    .collect();   // ← heap allocation + copies

let result: f64 = filtered.iter()
    .map(|&(x, y)| ((x * x + y * y) as f64).sqrt())
    .sum();
```

The first version processes each element in L1 cache and never allocates. The second version writes intermediate results to a new `Vec`, which may be large enough to evict the original data from cache. When it then reads the intermediate `Vec`, the original data must be re-fetched for subsequent operations.

Measuring Cache Performance

Knowing the theory is necessary but not sufficient. You must measure to find the actual bottlenecks.

perf stat (Linux)

```
perf stat -e cache-references,cache-misses,L1-dcache-loads,L1-dcache-load-misses \
    ./target/release/my_program
```

Key metrics:

- **L1 cache miss rate**: below 5% is good, above 10% indicates layout problems
- **LLC (Last-Level Cache) miss rate**: below 1% is good for data that fits in L3
- **Instructions per cycle (IPC)**: low IPC (< 1.0) on data-heavy code usually means cache misses stalling the pipeline

cachegrind (Linux/macOS via Valgrind)

```
valgrind --tool=cachegrind ./target/release/my_program
cg_annotate cachegrind.out.*
```

Cachegrind simulates the cache hierarchy and reports miss rates per function and per source line. It's slower than real execution but gives precise per-line attribution.

Instruments (macOS)

Use the **Counters** instrument in Xcode Instruments to record hardware performance counters on Apple Silicon. The key counters are `L1D_CACHE_MISS_LD` and `L1D_CACHE_MISS_ST`.

Simple Benchmarking

For quick comparison, use Criterion:

```rust
use criterion::{black_box, criterion_group, criterion_main, Criterion};

fn bench_layouts(c: &mut Criterion) {
    let aos = create_aos_data(1_000_000);
    let soa = create_soa_data(1_000_000);

    c.bench_function("AoS position sum", |b| {
        b.iter(|| sum_positions_aos(black_box(&aos)))
    });

    c.bench_function("SoA position sum", |b| {
        b.iter(|| sum_positions_soa(black_box(&soa)))
    });
}
```

ARM64 Layout Considerations

The cache line size on Apple Silicon is 128 bytes — double the 64 bytes used by x86-64 (Intel and AMD). This has direct implications:

Larger effective cache lines mean SoA is even more beneficial. A 128-byte cache line holds 32 `f32` values, so SoA access patterns get twice the useful data per line.

But false sharing has a wider blast radius. Two independent atomic variables within 128 bytes of each other cause contention on Apple Silicon, while on x86 they'd only conflict if within 64 bytes. The `CachePadded` type from `crossbeam` uses 128-byte padding on Apple Silicon to avoid this.

Struct size thresholds shift. On x86, a 64-byte struct fits exactly in one cache line. On ARM64/Apple Silicon, two such structs fit in one cache line — accessing the second one is free.

Aspect	x86-64 (Intel/AMD)	ARM64 (Apple Silicon)
Cache line size	64 bytes	128 bytes
`f32` values per line	16	32
`f64` values per line	8	16
Hot struct target size	≤ 64 bytes	≤ 128 bytes
False sharing padding	64 bytes	128 bytes

Note: ARM Cortex cores (server and mobile) typically use 64-byte cache lines, similar to x86-64. The 128-byte line is an Apple Silicon design choice.

Putting It All Together

Data layout optimization follows a clear priority order:

1. **Access only the data you need.** Hot/cold splitting and SoA ensure that every byte loaded from memory is a byte your computation actually uses. This is the highest-impact change — often 2-5x.

2. **Keep data contiguous.** `Vec<T>` over `LinkedList<T>`, arena allocation over scattered `Box<T>`, flat matrices over `Vec<Vec<T>>`. The prefetcher rewards sequential access patterns with effectively zero-latency loads.

3. **Minimize struct size.** Smaller types where the value range permits, field reordering (automatic in `repr(Rust)`, manual in `repr(C)`), and removing unused fields. More elements per cache line = fewer cache lines loaded = faster iteration.

4. **Iterate in memory order.** Row-major for row-major-stored matrices. Sequential iteration of slices. Avoid random-access patterns on large datasets.

5. **Use lazy iterator chains.** Process data in a single pass without intermediate allocations. Let iterator fusion keep data in L1 cache through the entire pipeline.

These aren't premature optimizations. For any code that processes large datasets — game engines, scientific computing, data pipelines, web servers handling request batches — data layout is the single biggest lever you have. The algorithms textbook says your `O(n)` scan is optimal. Cache-aware data layout makes that `O(n)` scan 5-10x faster.

Chapter Summary

1. **Struct layout** in Rust involves size, alignment, and padding. `repr(Rust)` reorders fields to minimize padding; `repr(C)` preserves declaration order (sort by descending alignment to minimize waste).

2. **Padding wastes cache capacity.** A struct with 10 bytes of data and 14 bytes of padding uses only 42% of every cache line. Reordering fields or shrinking types reclaims this space.

3. **The cache line (64 bytes on x86, 128 on Apple Silicon)** is your fundamental budget. Data within the same cache line is free to access; data in a different line may cost 4-200+ cycles.

4. **Array of Structs (AoS)** is optimal when you access all fields of each element. **Struct of Arrays (SoA)** is optimal when you access few fields across many elements — and it enables SIMD auto-vectorization.

5. **Hot/cold splitting** separates frequently-accessed fields from rarely-accessed ones. The hot path iterates over smaller, denser data structures.

6. **Indices beat pointers** for cache performance. Arena-allocated trees with index references can be 10-20x faster than `Box`-based trees due to spatial locality.

7. **Iteration order** on multidimensional data determines cache line utilization. Row-major iteration on row-major storage achieves 100% utilization; column-major achieves ~12%.

8. **Iterator chains** process data lazily — no intermediate allocations, single-pass access, data stays in L1 cache through the pipeline.

9. **Smaller structs = more elements per cache line.** Use `u16` instead of `u32` where the value range permits. The cache doesn't care about type safety — it cares about bytes.

10. **Measure with hardware counters** (`perf stat`, Instruments, cachegrind) to identify cache miss hotspots. Theory guides your changes; measurement proves they work.