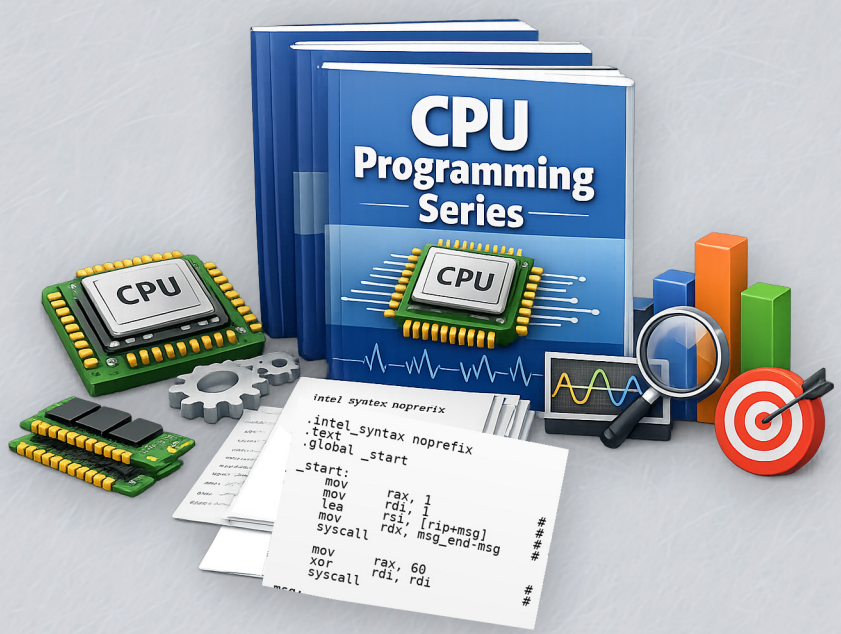
https://simplifycpp.org

CPU Programming Series

Registers, Flags, and Data Representation

Binary Reality for Assembly Programmers



2

Prepared by Ayman Alheraki

CPU Programming Series

Registers, Flags, and Data Representation

Binary Reality for Assembly Programmers

Prepared by Ayman Alheraki

simplifycpp.org

November 2025

Contents

Preface

Why This Booklet Exists

Modern CPUs execute nothing but binary operations on fixed-width registers. Yet many programmers—especially those coming from high-level languages—approach assembly with mental models borrowed from abstractions that do not exist at the hardware level. This booklet exists to correct that mismatch.

Its purpose is not to teach assembly syntax, operating systems, or calling conventions, but to establish a precise and disciplined understanding of how CPUs interpret bits, how arithmetic truly works at the register level, and how status flags expose the consequences of each instruction. Without this foundation, any further exploration of assembly programming becomes fragile, error-prone, and misleading.

This booklet is the first structural pillar in the CPU Programming Series. It defines the mental model required before stacks, memory models, or system-level mechanisms can be understood correctly.

What “Binary Reality” Really Means

“Binary Reality” refers to the fact that the processor has no concept of types, intentions, or abstractions. A register is merely a collection of bits. Whether those bits represent a signed

integer, an unsigned integer, a character, a pointer, or something else entirely is determined solely by the instruction that operates on them and by the programmer's interpretation.
The CPU does not know what is "negative" or "positive". It does not know what is "overflow" in a semantic sense. It only sets flags based on well-defined electrical and logical outcomes of binary operations. Concepts such as signedness, overflow, or comparison are human-imposed interpretations layered on top of this raw behavior.
Understanding binary reality means learning to think the way the processor works: bit by bit, boundary by boundary, without assumptions inherited from high-level languages.

What This Booklet Does Not Cover

To preserve conceptual clarity, this booklet intentionally excludes several important topics that are commonly mixed into early assembly learning:

- Stack behavior and stack frames
- Calling conventions and function mechanics
- Memory hierarchy, caches, and virtual memory
- Instruction pipelines and speculative execution
- Operating system interaction

These topics are not omitted due to lack of importance, but because they rely on a correct understanding of registers, flags, and data representation. Introducing them prematurely often leads to false mental models that are difficult to unlearn.
This booklet focuses exclusively on what happens *inside the register file and status flags* as a direct consequence of instruction execution.

How to Read This Booklet Effectively

This booklet is designed to be read sequentially. Each section builds on assumptions and definitions established earlier. Skipping chapters or treating the material as a quick reference defeats its purpose.
Readers are encouraged to:

- Mentally simulate each operation at the bit level
- Ignore high-level language analogies unless explicitly stated
- Focus on why flags change, not just that they change
- Re-read sections dealing with signed vs unsigned logic and overflow

All assembly examples use Intel syntax with GAS conventions. Comments follow the # style to maintain correctness and consistency across modern toolchains.
Mastery of this booklet does not make one an expert assembly programmer—but without mastering it, expertise is impossible.

Chapter 1

Bits, Registers, and the Machine View

1.1 What a Register Really Is (Beyond the Name)

A register is a fixed-size storage element within the central processing unit (CPU) used to hold binary data that participates directly in arithmetic, logic, and control operations. Physically implemented as flip-flops or static memory cells, registers are the fastest storage available in the computing system. They do not reside in main memory, cache, or any external bus; they are local to the CPU's execution units and are accessed without address translation or memory hierarchy. The name given to a register (e.g., `rax`, `rbx`) is for human convenience. The processor treats a register as a specific bundle of bits wired into its execution logic.

1.2 Bit Width, Lanes, and Hardware Limits

The bit width of a register defines how many binary digits it contains. Common CPU families such as x86-64 have 64-bit general-purpose registers, meaning each register holds 64 individual bits. The width determines the numeric range of values a register can represent

and the amount of data that can be processed in a single instruction. Some registers are conceptually divided into sub-width portions (e.g., the lower 32, 16, or 8 bits of a 64-bit register), commonly referred to as “lanes”. These lanes are not separate registers; they represent different views of the same physical bits. The hardware enforces strict limits: operations that exceed the native bit width either truncate high-order bits or set status flags that reflect the outcome. There is no implicit extension or growth beyond the defined width.

1.3 Logical vs Physical Interpretation of Bits

Bits themselves are the simplest physical representation of information: they can be in one of two electrical states corresponding to logical 0 or 1. This state has no inherent meaning until defined by interpretation rules. Logical interpretation refers to how software or an instruction set architecture assigns semantic weight to bit patterns (e.g., integer, address, boolean). Physical interpretation refers strictly to the presence or absence of electrical charge or transistor state. The CPU hardware operates on the physical level; logical interpretation is a human and software abstraction layered on top of these physical states.

1.4 Registers as Raw Containers (No Types, No Meaning)

At the hardware level, registers contain raw bits without inherent types. The CPU does not enforce signedness, floating point semantics, or any high-level type system. A set of bits loaded into a register is the same whether interpreted as an unsigned integer, a signed integer in two’s complement, or as a bit mask. The meaning is determined exclusively by the operation applied to those bits. For example, adding two registers with an unsigned add instruction versus a signed add instruction uses the same physical bits but produces the same binary result; only the interpretation of status flags changes. A register is a raw container that the instruction set interprets according to defined operational semantics.

1.5 When Meaning Emerges: Instruction Context

Meaning emerges only in the context of instructions. An instruction defines how the binary patterns in registers are to be processed, how the result is formed, and how the CPU flags are affected. For instance, a comparison instruction (`cmp`) evaluates bits according to signed or unsigned rules specified by the instruction semantics and updates flags accordingly. A shift instruction like `sal` (shift arithmetic left) treats bits purely as binary digits to be moved, without any notion of signedness in the act of shifting, although it affects flags in a defined way. The combination of a specific opcode, operand size, and the CPU's defined behavior yields the semantic meaning of otherwise raw bits. Without instruction context, registers hold patterns that are semantically inert.

Chapter 2

Bits, Registers, and the Machine View

2.1 What a Register Really Is (Beyond the Name)

A register is a physically realized storage element inside the CPU core, implemented using high-speed circuitry such as flip-flops or latch-based cells. Unlike memory locations, registers are not addressed through memory buses, caches, or translation mechanisms. They are directly wired into the execution units of the processor and participate in instruction execution without indirection.

From the processor's perspective, a register is nothing more than a fixed-width collection of bits that can be read from and written to by specific instructions. The symbolic names assigned to registers (such as `rax`, `rbx`, or `r0`) exist purely for the benefit of programmers, assemblers, and documentation. Internally, the CPU identifies registers by encoded indices within instruction operands.

Registers do not store metadata. They do not remember how a value was produced, whether it represents a number, an address, or a logical mask. Once written, the previous interpretation of a register's contents is completely irrelevant. Only the current bit pattern matters.

This distinction is critical: registers are not variables, objects, or typed storage. They are

transient, mechanical storage elements optimized for speed and proximity to execution logic.

2.2 Bit Width, Lanes, and Hardware Limits

Every register has a fixed architectural width defined by the processor design. In modern x86-64 systems, general-purpose registers are 64 bits wide. This width determines the maximum amount of information that can be represented and manipulated in a single operation.
Sub-registers, often referred to as lanes, provide partial views of the same physical register. For example, accessing the lower 32 bits of a 64-bit register does not involve a different storage location; it is a restricted view of the same underlying bits. The hardware enforces precise rules regarding how these partial accesses interact with the full register, including zero-extension or preservation of higher bits depending on the instruction.
The hardware imposes strict limits on bit width. When arithmetic or logical operations exceed the representable range of a register, the excess bits are discarded. This is not an error condition; it is normal behavior. The processor records information about such events only through status flags, never by expanding storage or raising exceptions for integer operations.
Understanding these limits is essential. CPUs operate within rigid bit boundaries, and all higher-level notions of numeric range or safety must be constructed explicitly by the programmer.

2.3 Logical vs Physical Interpretation of Bits

At the physical level, a bit corresponds to a stable electrical state within the processor's circuitry. This state is binary and devoid of meaning. The processor's logic gates manipulate these states according to predefined rules, without awareness of any abstract interpretation.
Logical interpretation arises from the instruction set architecture. When an instruction operates on a register, it defines how the bit pattern should be treated: as an unsigned quantity,

as a signed quantity in two's complement form, or as a purely logical pattern. Importantly, this interpretation does not alter the stored bits themselves; it alters only how the CPU evaluates conditions and sets flags.

The same bit pattern can simultaneously be a valid unsigned integer, a valid signed integer, and a meaningful bitmask. The processor does not choose between these interpretations. The responsibility lies entirely with the instruction semantics and the programmer's intent.

This separation between physical state and logical meaning is fundamental to understanding low-level programming.

2.4 Registers as Raw Containers (No Types, No Meaning)

Registers are typeless by design. The CPU does not enforce or track data types for register contents. There is no distinction between an integer register and a pointer register at the hardware level. All registers store bit patterns, and all operations manipulate those patterns according to instruction-defined rules.

For example, the result of an addition instruction is identical at the bit level regardless of whether the operands are considered signed or unsigned. What changes is how the processor sets flags such as Carry or Overflow, which reflect different interpretations of the same binary result.

This lack of inherent meaning is a source of both power and danger. It enables highly efficient and flexible computation but also allows subtle bugs when programmers apply incorrect interpretations to register contents.

Registers are therefore best understood as raw containers whose meaning is ephemeral and entirely contextual.

2.5 When Meaning Emerges: Instruction Context

Meaning emerges only at the moment an instruction is executed. The opcode, operand size, and instruction semantics together define how the processor interprets the bits in a register and how it reacts to the result.

A comparison instruction does not modify registers but evaluates their contents and updates flags based on either signed or unsigned rules. A shift instruction moves bits mechanically, but may interpret the most significant bit differently depending on whether the shift is logical or arithmetic. Rotate instructions treat the register as a closed loop of bits, ignoring numeric interpretation entirely.

In all cases, the register itself remains unchanged in nature. Only the processor's interpretation during instruction execution creates semantic meaning, and that meaning disappears once execution moves on.

Understanding this principle is essential before studying control flow, stack usage, or memory access. Without it, programmers are likely to project abstractions onto hardware that simply do not exist.

Chapter 3

Signed vs Unsigned: Same Bits, Different Truths

3.1 Why the CPU Does Not Know “Signed”

The CPU stores and processes bit patterns. At the hardware level, there is no intrinsic notion of “signed” or “unsigned” values inside a general-purpose register. Integer adders, subtractors, and logic units operate on bits using fixed rules (carry propagation, bitwise logic) and produce a fixed-width result. What changes between signed and unsigned reasoning is not the computed bit pattern, but how software *interprets* that pattern and which status flags are considered meaningful for detecting exceptional conditions.

“Signedness” is therefore an interpretation convention defined by the instruction set architecture (ISA) and by the programmer. The ISA specifies how flags are set and how conditional branches interpret those flags in signed or unsigned contexts, but the underlying datapath does not tag values with types.

3.2 Unsigned Interpretation: Natural Binary Order

Under unsigned interpretation, an N-bit register represents an integer in the range $[0, 2^N - 1]$. The ordering is the natural binary order: higher bits carry greater weight, and the most significant bit (MSB) is simply the highest place value, not a sign indicator. Arithmetic in unsigned interpretation is performed modulo 2^N: results that exceed the maximum wrap around by discarding bits beyond width N.

In this model, the Carry Flag (CF) is the primary indicator for overflow of unsigned addition (a carry out of the MSB) and for borrow behavior in subtraction (as defined by the ISA). Unsigned comparisons also rely on CF and ZF to express the outcome.

3.3 Signed Interpretation: Human-Defined Semantics

Signed interpretation assigns meaning to the same N bits using a representation convention, overwhelmingly two's complement in modern mainstream ISAs. In two's complement, the representable range is $[-2^{N-1}, 2^{N-1} - 1]$. The MSB participates in the numeric value as a negative weight, not as a separate "sign bit" field in the way high-level language diagrams often imply.

Crucially, the bit pattern result of addition/subtraction is identical whether you interpret inputs as signed or unsigned. What differs is which results are considered outside the representable signed range. The Overflow Flag (OF) indicates that a signed operation produced a result that cannot be represented in N-bit two's complement, even though the bit pattern is well-defined modulo 2^N.

3.4 Comparison Instructions and Interpretation Rules

Comparison in assembly is performed by instructions that set flags based on subtraction-like evaluation without necessarily storing a result (e.g., `cmp`). The comparison itself computes relationships by updating status flags (notably ZF, SF, OF, CF) and then a conditional branch instruction interprets those flags according to signed or unsigned rules.
For x86-family semantics in particular:

- Unsigned relations use CF and ZF (e.g., below/above).
- Signed relations use SF and OF (with ZF) (e.g., less/greater).

Thus the ISA provides *two* families of conditional branches: one for unsigned ordering and one for signed ordering, both derived from the same flag-setting operation.

```
# x86-64 GAS, Intel syntax

# unsigned: if (a < b) using CF/ZF
cmp     rax, rbx
jb      .L_unsigned_less        # jump if below (CF=1)

# signed: if (a < b) using SF/OF
cmp     rax, rbx
jl      .L_signed_less          # jump if less (SF != OF)
```

The key discipline is to choose the conditional jump that matches the intended interpretation of the operands, not the visual shape of the bit pattern.

3.5 Signed vs Unsigned in Arithmetic Instructions

Integer arithmetic instructions generally compute the same truncated N-bit result regardless of signedness. The ISA then exposes two distinct diagnostics:

- **CF** indicates carry out of the MSB for addition (unsigned overflow) and is used in unsigned multi-precision arithmetic.
- **OF** indicates signed overflow: the mathematical signed result does not fit in N bits (two's complement range violation).

A concise way to reason about OF in addition is: if two operands have the same signed sign (same MSB) and the result has a different sign, signed overflow occurred. CF is independent: it detects bit carry beyond the width boundary and is meaningful for unsigned wraparound detection.

```
# x86-64 GAS, Intel syntax

# Example: show that CF and OF are different signals
mov     al, 0xFF                # 255 unsigned, -1 signed (8-bit)
add     al, 0x01                # result = 0x00 (wrap)
# CF=1 (unsigned overflow out of 8 bits)
# OF=0 (signed: -1 + 1 = 0 fits)

mov     al, 0x7F                # 127 signed max (8-bit)
add     al, 0x01                # result = 0x80 (-128 signed)
# CF=0 (no carry out of MSB in this case)
# OF=1 (signed overflow: 127 + 1 cannot be represented)
```

The arithmetic datapath is the same; the flags provide separate lenses for correctness depending on interpretation.

3.6 Common Logical Errors in Mixed Interpretation

The most common bugs arise not from wrong instructions, but from mixing interpretation rules:

- **Using signed branches for unsigned data (or vice versa).** A classic error is using `jl/jg` when values are intended as sizes, indices, counters, or bitfields, which are naturally unsigned.
- **Treating the MSB as "the sign" in all contexts.** The MSB is only a sign indicator *under a chosen signed convention*. Under unsigned interpretation it is just the highest weight bit.
- **Detecting unsigned overflow with OF or signed overflow with CF.** CF is for unsigned carry/borrow behavior; OF is for signed range violation.
- **Assuming comparisons are "type aware".** The `cmp` instruction sets flags; the interpretation happens at the conditional jump. The CPU does not remember intent.
- **Implicitly mixing widths.** Extending or truncating values without an explicit rule (zero-extension vs sign-extension) creates silent reinterpretations across widths.
- **Confusing wraparound with error.** Fixed-width arithmetic wraps by design; "error" is a higher-level policy that must be checked using the correct flags and rules.

Correct low-level reasoning requires adopting a single interpretation at a time, choosing the matching extension rule (zero/sign), and using the appropriate conditional branches and overflow checks for that interpretation.

Chapter 4

Two's Complement: The Universal Lie We Agree On

4.1 Why Two's Complement Exists

Two's complement exists because it provides a mathematically consistent way to represent signed integers using fixed-width binary registers while keeping hardware simple. The key design goal is that the same adder circuit used for unsigned addition can also perform signed addition and subtraction without extra "sign handling" circuitry. With two's complement, subtraction can be implemented as addition of a negated operand, enabling a unified arithmetic datapath.

Two's complement also yields a single representation for zero (unlike signed-magnitude and ones' complement, which have both $+0$ and -0), and it makes signed comparisons and sign-extension well-defined and efficient in hardware and instruction sets.

4.2 Encoding Positive and Negative Values

For an N-bit register:

- Non-negative values (0 to $2^{N-1}-1$) are encoded in ordinary binary.
- Negative values (-1 to -2^{N-1}) are encoded such that the most significant bit (MSB) contributes a negative weight.

Formally, an N-bit pattern $b_{N-1}\ldots b_1 b_0$ represents the signed value:

$$-b_{N-1}\cdot 2^{N-1}+\sum_{i=0}^{N-2} b_i\cdot 2^i$$

This is not a "sign bit plus magnitude" model; it is a weighted sum where the MSB has negative weight. The representable range is:

$$-2^{N-1}\leq x\leq 2^{N-1}-1$$

4.3 Two's Complement Arithmetic Rules

Two's complement arithmetic on N-bit registers is modular arithmetic modulo 2^N. The hardware produces the same N-bit result for addition and subtraction regardless of whether operands are interpreted as signed or unsigned; interpretation affects only how you judge correctness (e.g., via overflow detection).

Signed overflow is not "wraparound failure" at the bit level. The bits are always correct modulo 2^N. Signed overflow occurs only when the mathematical signed result lies outside the representable range $[-2^{N-1}, 2^{N-1}-1]$. In many ISAs, this is reflected by the Overflow Flag (OF) for relevant operations.

4.4 Negation, Inversion, and Addition

In two's complement, negation of an N-bit value x is performed by inverting all bits and adding 1:

$$-x \equiv \sim x + 1 \pmod{2^N}$$

This identity is the core reason subtraction becomes addition:

$$a - b \equiv a + (\sim b + 1) \pmod{2^N}$$

At the machine level, this is why CPUs can implement subtraction using the same adder as addition: the only additional operation needed is bitwise inversion and the injection of a carry-in of 1.

```
# x86-64 GAS, Intel syntax

# Compute -x in AL (8-bit) using NOT + ADD
mov     al, 0x2A            # x = 42
not     al                  # ~x
add     al, 1               # ~x + 1  => -x modulo 256

# Subtraction as addition of two's complement:
# a - b == a + (~b + 1)
mov     al, 10
mov     bl, 3
mov     cl, bl
not     cl
add     cl, 1
add     al, cl              # al = 10 - 3 (mod 256)
```

4.5 Edge Cases: Minimum Negative Value

The most important edge case in two's complement is the minimum representable value:

$$\text{MIN} = -2^{N-1}$$

Its bit pattern is a 1 in the MSB and zeros elsewhere:

$$1000\ldots0$$

This value has no positive counterpart representable in the same width, because the positive range ends at $2^{N-1} - 1$. Therefore:

$$-\text{MIN} = \text{MIN} \pmod{2^N}$$

Negating MIN overflows in signed arithmetic. In many ISAs, the negation instruction (or equivalent subtraction from zero) will indicate signed overflow (OF set) when attempting to negate the minimum value.

```
# x86-64 GAS, Intel syntax

# Demonstrate MIN negation behavior for 8-bit
mov     al, 0x80            # -128 in 8-bit two's complement (MIN)
neg     al                  # result is still 0x80; signed overflow
 ↪  occurs (OF=1)
```

This is not a bug in the CPU. It is a consequence of asymmetric range in two's complement.

4.6 Why Two's Complement Never Needs a Sign Bit Instruction

Two's complement does not require a separate instruction to handle a "sign bit" because sign is not a detachable field. The MSB participates in the numeric value by weight, and arithmetic

naturally propagates across all bits through the adder.
Sign extension is likewise mechanical: extending an N-bit signed value to a wider register is performed by replicating the MSB into the new higher bits. No special “convert sign” step is required beyond this replication rule, because it preserves the weighted-sum interpretation.
As a result, the same fundamental instructions (add, sub, inc, dec, cmp) operate uniformly on signed and unsigned interpretations. Only:

- the chosen extension rule (zero-extend vs sign-extend),
- the chosen conditional branches (signed vs unsigned),
- and the chosen overflow interpretation (OF vs CF)

determine whether the programmer is reasoning in signed or unsigned terms. The hardware remains purely bitwise and width-bound.

Chapter 5

Overflow vs Carry: Two Very Different Failures

5.1 Why Arithmetic "Failure" Is Contextual

Fixed-width CPU integer arithmetic is defined modulo 2^N for an N-bit operation: the hardware always produces an N-bit result by discarding any bits beyond the width boundary. In that sense, integer arithmetic on the CPU does not "fail"; it deterministically wraps.

What programmers call "failure" is a higher-level semantic notion: the computed mathematical result (in some interpretation) does not fit the intended representable range. Because the same bit pattern can be interpreted as unsigned or signed (two's complement), the meaning of "out of range" depends on interpretation. The ISA exposes this contextual information through distinct status flags: Carry Flag (CF) for unsigned boundary events, and Overflow Flag (OF) for signed range violations.

5.2 Carry Flag: Unsigned Arithmetic Overflow

The Carry Flag indicates a carry out of the most significant bit (MSB) position in addition, which corresponds to overflow in *unsigned* arithmetic. For an N-bit addition:

$$a + b = r \pmod{2^N}$$

CF is set if the true mathematical sum is $\geq 2^N$, meaning a $(N+1)$-th bit would be needed to represent it. This is exactly the condition required for multi-precision arithmetic: CF becomes the propagated carry into the next higher word.
For subtraction, many ISAs define CF in a complementary way (borrow-related), but the conceptual role remains: CF encodes an unsigned boundary event that matters when operands are interpreted as non-negative integers or as words in a larger integer.

```
# x86-64 GAS, Intel syntax

# Unsigned overflow example (8-bit): 255 + 1 wraps to 0 with CF=1
mov     al, 0xFF            # 255
add     al, 0x01            # al = 0x00, CF=1
```

5.3 Overflow Flag: Signed Arithmetic Violation

The Overflow Flag indicates that a signed two's complement result cannot be represented within the operand width. For signed N-bit integers, the representable range is:

$$-2^{N-1} \leq x \leq 2^{N-1} - 1$$

OF is set for addition when adding two operands with the same sign yields a result with the opposite sign. This is the canonical two's complement overflow condition:

- positive + positive $\rightarrow$ negative

- negative + negative $\rightarrow$ positive

OF is not about carries; it is about *sign consistency under the signed interpretation*. The hardware result remains valid modulo 2^N, but it violates the intended signed range.

```
# x86-64 GAS, Intel syntax

# Signed overflow example (8-bit): 127 + 1 cannot be represented
mov     al, 0x7F            # 127
add     al, 0x01            # al = 0x80 (-128), OF=1
```

5.4 How the Same Operation Triggers Different Flags

The same binary addition can set CF, OF, both, or neither, because each flag encodes a different interpretation:

- CF answers: "Did the addition exceed $2^N - 1$ in unsigned arithmetic?"
- OF answers: "Did the signed result exceed $2^{N-1} - 1$ or go below -2^{N-1}?"

Consider these 8-bit examples:

```
# x86-64 GAS, Intel syntax

# Case A: CF=1, OF=0  (unsigned overflow only)
mov     al, 0xFF            # 255 unsigned, -1 signed
add     al, 0x01            # 0x00, CF=1, OF=0  (-1 + 1 = 0 fits)

# Case B: CF=0, OF=1  (signed overflow only)
mov     al, 0x7F            # 127
add     al, 0x01            # 0x80, CF=0, OF=1
```

```
# Case C: CF=1, OF=1   (both)
mov     al, 0x80            # -128 signed, 128 unsigned
add     al, 0x80            # 0x00, CF=1 (128+128=256), OF=1
 ↪  (-128+-128 out of range)
```

The arithmetic datapath is identical in all cases; the flags provide two different "range lenses" over the same bit-level event.

5.5 Visualizing Overflow in Binary Space

A useful mental model is to treat N-bit results as points on a circle of size 2^N (modular space). Addition moves forward around the circle; subtraction moves backward.

- In **unsigned** interpretation, the valid range is the entire circle labeled 0 to $2^N - 1$. "Overflow" corresponds to crossing the boundary from $2^N - 1$ back to 0, which is exactly what CF reports.
- In **signed** interpretation, the same circle is relabeled into a contiguous signed interval

$$-2^{N-1} \ldots -1, 0, 1 \ldots 2^{N-1} - 1$$

  "Overflow" occurs when an addition step crosses the signed boundary between $2^{N-1} - 1$ and -2^{N-1}, which OF reports.

Thus CF corresponds to wraparound at the unsigned boundary, while OF corresponds to wraparound at the signed boundary under two's complement relabeling.

5.6 Real Bugs Caused by Misreading Flags

Misreading CF and OF produces classic low-level failures:

- **Using OF to validate sizes, indices, or lengths.** These quantities are naturally unsigned; checking OF misses real out-of-range conditions and can accept wrapped values as "valid".
- **Using CF to validate signed computations.** A signed range violation can occur with CF clear (e.g., $127 + 1$ in 8-bit), causing silent corruption when CF is used as the only overflow test.
- **Choosing the wrong conditional jumps after `cmp`.** Signed relations (`jl`, `jg`) depend on SF and OF; unsigned relations (`jb`, `ja`) depend on CF and ZF. Mixing them breaks boundary checks and can introduce security bugs.
- **Multi-precision arithmetic implemented with OF.** Carry propagation between words is an unsigned concept; using OF instead of CF yields incorrect high-word accumulation.
- **Incorrect saturation/clamping.** Clamping to signed bounds requires OF-aware logic; clamping to unsigned bounds requires CF-aware logic. Swapping them clamps at the wrong boundary.

The disciplined rule is simple: use CF for unsigned boundary detection and multi-word carries; use OF for signed range violation. After comparisons, select conditional branches that match the intended signedness of the operands.

Chapter 6

CPU Flags: Reading the Processor's Mind

6.1 Status Flags as Side-Channel Information

Status flags are small pieces of architectural state updated as a by-product of executing many arithmetic, logical, and compare-class instructions. They are not general storage, and they are not "types" or "exceptions". Instead, they are a side-channel: a compact summary of properties of the last operation (e.g., whether the result became zero, whether a carry-out occurred, whether a signed overflow happened).

Flags exist to make control flow and multi-precision arithmetic efficient. A single instruction can compute a result and simultaneously expose conditions needed for branching, bounds checks, and carry propagation. Critically, flags are ephemeral: they describe *the most recent relevant operation*, and most instructions overwrite them. Correct low-level code therefore treats flags as short-lived signals that must be consumed promptly and intentionally.

6.2 Zero Flag (Z): Absence of Value

The Zero Flag (ZF) is set when the result of an operation is exactly zero in the operand width. It does not mean "false" in a high-level sense; it means the computed bit pattern is all zeros. Many instructions update ZF based on their result, including arithmetic (`add`, `sub`), bitwise logic (`and`, `xor`, `or`), and explicit tests (`test`, `cmp`).
ZF is foundational for equality and zero checks because it is width-correct: a 64-bit operation sets ZF based on the full 64-bit result, not an arbitrary subset.

```
# x86-64 GAS, Intel syntax

xor     eax, eax          # eax = 0, ZF=1 (result is zero)
test    rax, rax          # sets ZF based on rax without modifying it
jz      .L_is_zero        # jump if ZF=1
```

6.3 Carry Flag (C): Bit Escaping the Boundary

The Carry Flag (CF) reports a boundary event in fixed-width arithmetic. For addition, CF is set when there is a carry-out from the most significant bit of the operation width, i.e., the true mathematical sum does not fit in N bits for an N-bit add. This makes CF the canonical signal for *unsigned* overflow and for multi-word carry propagation.
For subtraction, many ISAs define CF in relation to borrow in a specified way; the key idea remains that CF encodes an unsigned boundary condition tied to the width limit, not a signed range violation.

```
# x86-64 GAS, Intel syntax

mov     al, 0xFF          # 255
```

```
add     al, 1               # al = 0, CF=1 (carry out of 8-bit
 ↪  boundary)
jc      .L_carry            # jump if CF=1
```

6.4 Sign Flag (S): The Most Significant Bit

The Sign Flag (SF) mirrors the most significant bit of the result (for the chosen operand width). It indicates whether the MSB is 1. Under two's complement signed interpretation, SF corresponds to "negative" results, but SF itself is purely bit-level: it does not know "signedness". It is simply a copy of the MSB.
SF is therefore meaningful when used with signed-interpretation rules (typically in combination with OF for comparisons), and it is also useful as a direct MSB test in bit-manipulation logic.

```
# x86-64 GAS, Intel syntax

mov     al, 0x01
sub     al, 0x02            # al = 0xFF, SF=1 (MSB of 8-bit result is
 ↪  1)
js      .L_msb_set          # jump if SF=1
```

6.5 Overflow Flag (O): Broken Signed Reality

The Overflow Flag (OF) indicates that a signed two's complement result is not representable in the operand width. For addition, OF is set when adding two operands with the same signed sign produces a result with the opposite sign. For subtraction, OF is set when subtracting operands of different signs produces a result whose sign contradicts the expected signed outcome.

OF does not describe carry-out and is not suitable for unsigned bounds checking. It is a signed-range diagnostic: the computation is still correct modulo 2^N, but it violates the mathematical signed range $[-2^{N-1}, 2^{N-1} - 1]$.

```
# x86-64 GAS, Intel syntax

mov     al, 0x7F            # 127 (8-bit)
add     al, 1               # al = 0x80 (-128), OF=1
jo      .L_signed_overflow
```

6.6 Combined Flag Logic in Conditional Jumps

Conditional jumps interpret flags according to signed or unsigned rules. After a `cmp` (conceptually a subtraction that updates flags), the ISA provides two families of relational branches:

- **Unsigned ordering** uses CF and ZF (e.g., below/above).
- **Signed ordering** uses SF and OF (with ZF) (e.g., less/greater).

Equality uses ZF for both interpretations, because equality is bit-exact in a fixed width.

```
# x86-64 GAS, Intel syntax

cmp     rax, rbx
je      .L_equal            # ZF=1

jb      .L_u_less           # unsigned: CF=1
ja      .L_u_greater        # unsigned: CF=0 and ZF=0

jl      .L_s_less           # signed: SF != OF
jg      .L_s_greater        # signed: ZF=0 and SF == OF
```

The discipline is to pick the branch mnemonic that matches the intended interpretation of the operands. The compare sets flags once; the branch chooses the interpretation.

6.7 Flag Dependency and Instruction Ordering

Because flags are overwritten frequently, correct code must treat them as a dependency chain: the consumer of flags (a conditional jump, `adc`/`sbb`, or a setcc instruction) must appear immediately after the producer (an arithmetic, logic, or compare instruction) unless intervening instructions are guaranteed not to modify the relevant flags.
This creates two practical rules:

- **Consume flags as soon as possible.** Do not insert unrelated arithmetic or logic between flag-setting and flag-using operations.
- **Be explicit about flag clobbering.** Many instructions update ZF/SF/OF/CF as part of their normal behavior; assume flags are not preserved unless the ISA guarantees it.

Multi-precision arithmetic illustrates this dependency clearly: `adc` and `sbb` explicitly consume CF from a prior operation. Any instruction that changes CF between the carry-producing add and the carry-consuming `adc` breaks correctness.

```
# x86-64 GAS, Intel syntax

# Correct: consume CF immediately
add     rax, rbx          # sets CF
adc     rcx, rdx          # adds with carry from CF

# Incorrect pattern (conceptual): an intervening instruction may
 ↪  clobber flags
# add     rax, rbx
```

```
# xor     r8d, r8d        # clobbers ZF/SF and often other flags
# adc     rcx, rdx        # now CF may not represent the intended
 ↪  carry
```

Flags are not stable state; they are transient signals. Treat them like one-instruction-wide outputs unless proven otherwise by the instruction semantics.

Chapter 7

Shifts and Rotates: Moving Bits with Consequences

7.1 Logical Shifts vs Arithmetic Shifts

A shift instruction moves bits within a fixed-width operand and discards bits that exit the width boundary. The vacated bit positions are filled according to the shift *kind*:

- **Logical shift** treats the operand as an unsigned bit pattern. Zeros are shifted in from the side opposite the direction of movement.
- **Arithmetic shift** is defined to preserve signed two's complement interpretation for right shifts by replicating the most significant bit (MSB), often called the sign bit under signed interpretation.

Left shifts are typically identical for logical and arithmetic variants on many ISAs because shifting left introduces zeros in the least significant bit (LSB) positions; the distinction becomes critical primarily for right shifts.

```
# x86-64 GAS, Intel syntax

# Logical right shift: zero-fill
shr     eax, 1            # shift right, fill MSB with 0

# Arithmetic right shift: sign-fill
sar     eax, 1            # shift right, replicate previous MSB
```

7.2 Left Shifts: Multiplication or Bit Destruction?

A left shift by k positions (`shl`/`sal`) is equivalent to multiplication by 2^k *only* when the shifted-out bits are all zero and the interpretation matches the intended arithmetic domain. Formally, for an N-bit operand x, the hardware computes:

$$(x \ll k) \bmod 2^N$$

This operation always discards the top k bits that exit the width boundary. Therefore, a left shift is better understood as *bit relocation with truncation*, not inherently as multiplication. The multiplication analogy holds only under a no-overflow condition.

```
# x86-64 GAS, Intel syntax

# Multiplication analogy holds only if no high bits are lost
mov     al, 0x10          # 00010000b (16)
shl     al, 1             # 00100000b (32)  OK

mov     al, 0x90          # 10010000b (144)
shl     al, 1             # 00100000b (32)  high bit destroyed (wrap)
```

7.3 Right Shifts: Sign Preservation vs Zero Fill

Right shifts move bits toward the LSB side; the key question is what fills the MSB side:

- **`shr`** (logical right shift) fills with zeros. This corresponds to division by 2^k for *unsigned* values when no fractional remainder is needed beyond truncation.
- **`sar`** (arithmetic right shift) replicates the original MSB, preserving the sign under two's complement interpretation. This corresponds to signed division by 2^k with truncation behavior defined by the ISA (typically toward zero or toward negative infinity depending on the architecture's exact semantics for division and shift; the shift itself is a bit operation that preserves sign bits).

The essential point: `sar` preserves the sign bit pattern, not a high-level mathematical guarantee for division semantics in every edge case. It is a representation-preserving shift, not a "true signed divide" operator.

```
# x86-64 GAS, Intel syntax

mov     al, 0xF0          # 11110000b (240 unsigned, -16 signed)

mov     bl, al
shr     bl, 1             # 01111000b (120 unsigned) zero-fill

mov     cl, al
sar     cl, 1             # 11111000b (-8 signed) sign-fill
```

7.4 Rotate Instructions: Circular Bit Flow

Rotate instructions do not discard bits; they circulate them within the operand width. A rotate left moves the MSB into the LSB; a rotate right moves the LSB into the MSB. Rotates

preserve the multiset of bits and are therefore fundamentally different from shifts.
Rotates are common in bit manipulation, hashing, cryptographic primitives, checksum logic, and certain normalization workflows. They are also useful for extracting or repositioning bit fields without losing information.

```
# x86-64 GAS, Intel syntax

mov     al, 0b10010001
rol     al, 1             # 00100011b (MSB wraps into LSB)

mov     al, 0b10010001
ror     al, 1             # 11001000b (LSB wraps into MSB)
```

7.5 Flag Effects of Shift and Rotate Operations

Shift and rotate instructions typically update flags based on the result and on the last bit shifted out. While exact flag-update rules are ISA-specific, a disciplined model is:

- **CF** captures the last bit shifted out (or rotated out) for single-bit shifts/rotates; this is often used for bit-serial logic.
- **ZF** reflects whether the post-operation result is zero.
- **SF** reflects the MSB of the post-operation result.
- **OF** for shifts/rotates is defined only for certain counts (commonly count=1) and reflects a sign-related condition derived from MSB transitions; it is not a general “overflow” concept for multi-bit shifts.

Because multi-bit shifts may leave OF undefined or architecturally unspecified depending on ISA rules, robust code avoids relying on OF for shift counts other than those explicitly defined by the architecture.

```
# x86-64 GAS, Intel syntax

# CF as the bit that falls off the edge
mov     al, 0b10000001
shl     al, 1             # result 00000010b, CF=1 (old MSB shifted
 ↪  out)

mov     al, 0b00000001
shr     al, 1             # result 00000000b, CF=1 (old LSB shifted
 ↪  out), ZF=1
```

7.6 When Shifts Break Signed Arithmetic

Shifts easily break signed arithmetic because they are width-bound bit operations, not mathematical operators with infinite precision.
Key failure modes include:

- **Left shift signed overflow.** Shifting a signed value left can change the MSB and therefore change the sign under two's complement interpretation. Even if the bit pattern is well-defined, the signed mathematical value may leave the representable range.
- **Assuming `sar` equals signed division in all cases.** `sar` preserves the sign bit pattern. For negative numbers, the rounding behavior implied by arithmetic shifting can differ from the rounding model expected in higher-level arithmetic unless the ISA and usage are aligned.
- **Mixing signed and unsigned after shifts.** A `shr` on a value later treated as signed can silently destroy sign information. Conversely, a `sar` on data later treated as unsigned injects ones into high bits.

A disciplined approach is:

- Choose `shr` when the value is an unsigned quantity or a bitfield.
- Choose `sar` only when preserving two's complement signed interpretation is explicitly intended.
- Treat left shifts as bit transformations first; only treat them as multiplication when you have proven that no significant bits are lost for the width in use.

Chapter 8

Alignment: When Bit Patterns Meet Hardware Rules

8.1 What Alignment Really Means at the CPU Level

Alignment is a property of a *memory address* relative to a required or preferred boundary. For an access of size k bytes, an address is k*-aligned* (often called *naturally aligned*) when:

$$\text{address} \bmod k = 0$$

This is the simplest and most practical definition: the address is a multiple of the access size. At the CPU level, alignment exists because a load/store instruction is specified to transfer an N-byte quantity beginning at the given address, but the hardware that implements this transfer has internal granularities and boundaries. A core does not fetch "an abstract object"; it fetches *bytes* from an address, and for multi-byte quantities it must gather a contiguous run of bytes and assemble them into a register according to the ISA's rules (including endianness and width).

Alignment is therefore not an attribute of the bits inside a register. A register can hold any bit pattern. Alignment becomes relevant only when the CPU uses a register as an *address* (effective address) for memory access. Two memory addresses that contain the same bytes but start at different offsets may represent the same logical value to a programmer, yet the CPU may have to perform different internal work to retrieve them efficiently and correctly.

Alignment as a boundary contract

Alignment is best understood as a boundary contract between software and hardware:

- Software promises to place certain objects at addresses that meet alignment requirements or preferences.
- Hardware promises that aligned accesses have defined behavior, often better performance, and sometimes stronger guarantees (such as atomicity for specific widths).

Many instruction sets define alignment requirements directly (some accesses may fault if misaligned), while others permit misaligned accesses but may implement them as a slower sequence of internal operations.

Operand width matters

Alignment is always relative to the operand width of the access, not to the register width. A 64-bit register holding an address does not force 8-byte alignment; rather, an instruction that loads 8 bytes from that address tends to be most efficient when the address is 8-byte aligned.

```
# x86-64 GAS, Intel syntax

# Alignment is a property of the address in rdi and the access width.
# 1-byte load: alignment is irrelevant for correctness on most ISAs.
```

```
movzx   eax, byte ptr [rdi]

# 4-byte load: naturally aligned when (rdi % 4 == 0)
mov     eax, dword ptr [rdi]

# 8-byte load: naturally aligned when (rdi % 8 == 0)
mov     rax, qword ptr [rdi]
```

Alignment and internal fetch/merge

Even when an ISA allows misaligned loads, the core may internally perform:

- two smaller aligned loads,
- a merge of the relevant bytes,
- and sometimes additional masking and shifting

to synthesize the architectural result. Alignment reduces the need for such splitting and merging, and it reduces the probability of crossing internal boundaries.

8.2 Natural Alignment and Performance Implications

Natural alignment is the conventional placement rule that gives the hardware its simplest case: an access whose starting address is a multiple of the access size. This convention tends to match the natural granularity of many datapaths and interconnects, and it minimizes boundary-crossing.

Why aligned is usually faster

On typical microarchitectures, aligned accesses tend to be faster for several reasons:

- **Fewer internal micro-operations.** A naturally aligned load is more likely to map to one internal transfer, rather than a split transfer plus merge.
- **Fewer boundary hazards.** Misaligned accesses are more likely to cross important boundaries (word boundaries, cache-line boundaries, page boundaries), each of which can increase internal work or latency.
- **Better throughput.** Even if a single misaligned load is only slightly slower, repeating it in a tight loop can reduce load/store throughput and limit overall performance.

The cost is not uniform

The penalty of misalignment is not a constant. It depends on:

- the access width (2, 4, 8, 16 bytes),
- the address offset (how far from the natural boundary),
- whether the access crosses an internal boundary,
- and which instruction form is used (scalar vs vector, atomic vs non-atomic).

The most important practical distinction is whether an access crosses a large boundary (for example, a cache-line boundary). Crossing such boundaries can force multiple internal transactions, and in the worst case can involve multiple cache lines or even multiple pages, making the access significantly slower and sometimes fault-prone at the page level.

Alignment and atomicity

Alignment is tightly related to architectural atomicity guarantees. Many platforms guarantee that naturally aligned loads/stores of certain sizes (often word-sized) are atomic. Misaligned access may not be atomic, may not be supported for atomic instructions, or may trap. This

matters for lock-free algorithms and for correctness under concurrency. Even when your code is single-threaded, alignment discipline reduces the risk of accidentally relying on non-guaranteed behaviors when code evolves.

8.3 Misaligned Access: Penalties and Exceptions

ISAs fall broadly into two models: permissive and strict. Real systems also sit on a spectrum: they may allow some misaligned widths, but not others; they may allow scalar misalignment, but not vector misalignment; they may allow misalignment for ordinary loads/stores, but forbid it for atomic operations.

Permissive model: allowed but potentially slower

In a permissive model, a misaligned load/store is architecturally defined: the CPU will produce the correct value as if bytes were read consecutively, but it may do so with extra internal work. The extra work can include splitting the transfer, performing multiple aligned sub-transfers, and merging the bytes.
This model improves software portability and reduces the need for explicit byte-by-byte fallback in many common cases, but it still penalizes misalignment and does not guarantee equal performance across different alignments.

Strict model: alignment faults

In a strict model, a misaligned access for certain widths or instructions raises an exception. This simplifies hardware and enforces discipline: software must ensure correct alignment or must handle faults. Strict alignment is common in designs that prioritize predictable behavior, simplicity, or certain guarantees.

Not all instructions behave the same

Even on architectures that allow many misaligned scalar loads/stores, there are common categories where alignment constraints become stricter:

- **Vector/SIMD loads/stores.** Some vector instruction forms historically required alignment, and some still benefit strongly from it even when misalignment is allowed.
- **Atomic read-modify-write instructions.** These often impose alignment requirements because the hardware must lock or coordinate a specific aligned unit.
- **Instructions with special semantics.** Certain gather/scatter patterns, string operations, or platform-specific instructions may have distinct rules.

Therefore, a disciplined rule is: do not infer alignment behavior from one instruction class to another; rely on the ISA definition for each class and maintain alignment where possible.

Misalignment and faults via boundary crossing

Even if an ISA permits misaligned loads, boundary crossing can still introduce exceptional cases:

- If the access crosses a page boundary and the second page is unmapped, a fault occurs even though the first byte range was valid.
- If an architecture implements misaligned loads by multiple sub-loads, each sub-load can fault independently.

This means a misaligned multi-byte access can fault in situations where aligned accesses would not, because it touches additional address ranges.

8.4 Alignment vs Data Representation

Alignment answers: “Is this address positioned on a boundary that makes a multi-byte transfer efficient and well-defined for this instruction?”
Data representation answers: “How do the bytes map to a value (endianness, signedness, field layout)?”
These concerns are orthogonal and must not be conflated.

Alignment does not change meaning; it changes access semantics

If memory contains a sequence of bytes that represent a value under a chosen representation, that meaning is independent of alignment. Alignment affects whether you can *retrieve those bytes as a unit* using a particular instruction form efficiently and safely.
For example:

- Endianness determines which byte becomes the least significant part of a multi-byte integer.
- Two’s complement determines how a bit pattern maps to a signed integer.
- Alignment determines whether the CPU can load the multi-byte value starting at that address in a single efficient operation.

Common pitfall: treating misalignment as “corruption”

Misalignment does not corrupt data. It may:

- slow down access,
- remove atomicity guarantees,
- or trap (fault) depending on the ISA and instruction.

But the bytes in memory remain unchanged. The “problem” is the mismatch between the access width and the boundary expectations of the hardware.

Alignment and struct/record layout

While this booklet avoids high-level language ABI details, one universal principle matters: grouping fields of different sizes often introduces padding so that each field begins at a boundary that satisfies its alignment. This padding is a layout strategy to maintain alignment contracts and improve access efficiency.

Even in pure assembly, the same issue exists: if you design a data record, your chosen offsets determine whether each field can be accessed with naturally aligned loads/stores.

8.5 Why Alignment Is Not a Memory Hierarchy Topic

Alignment is often discussed near caches because misalignment can interact with cache-line boundaries, but alignment is not fundamentally a cache policy topic. It is primarily a load/store semantics and hardware interface topic.

Alignment exists even without caches

Imagine a system with no caches. A load/store unit still transfers bytes between memory and registers. It still has internal granularity (for example, a natural word width). Alignment still matters because hardware must define how multi-byte transfers are performed and whether they can start at arbitrary byte offsets.

Caches can amplify the performance penalty (because crossing a cache line can force touching two lines), but caches do not create alignment requirements. Alignment arises from how the CPU and its memory interface are designed to fetch and assemble multi-byte values.

The core reason: boundary crossing multiplies work

The most general cost model is:

> Aligned accesses tend to touch fewer internal blocks; misaligned accesses are more likely to touch multiple blocks.

Those blocks may be:

- word-aligned internal transfer units,
- cache lines,
- page mappings,
- or bus transaction units.

Memory hierarchy affects *which blocks exist and how expensive they are*, but alignment is the concept of placing accesses so they do not straddle those blocks unnecessarily.

Alignment as a correctness boundary in concurrency

Finally, alignment matters for correctness even outside performance considerations, because many architectures define atomicity and synchronization primitives in terms of aligned accesses. This is not a cache topic; it is an architectural correctness contract for concurrency and for instruction semantics.

Practical discipline summary:

- Treat alignment as an address-width contract: *address modulo size*.
- Prefer natural alignment for multi-byte scalar values and for vector widths when applicable.

- Do not assume misaligned access is always safe: instruction class and ISA rules matter.
- Do not confuse alignment with representation: alignment affects access mechanics, representation affects meaning.
- Remember that alignment is a CPU load/store topic first; memory hierarchy only changes the cost surface.

Chapter 9

Endianness: Ordering the Same Reality Differently

9.1 Byte Order vs Bit Order

Endianness is a convention about *byte order in memory* for values that occupy more than one byte. It answers exactly one architectural question:

> When a multi-byte value is stored in memory, which byte goes at the lowest address?

This is fundamentally different from *bit order*. Bit order concerns how bits are numbered and manipulated *within* a byte or within a register by the ISA (instruction semantics). Endianness does not reorder bits inside a byte; it reorders *bytes* across increasing memory addresses for a multi-byte quantity.

Why this distinction matters

Many practical errors come from mixing these two viewpoints:

- Shifts, rotates, masks, and bit tests operate on bit positions *within a register value* and are not defined by endianness.
- Memory dumps, packet captures, and file hex views show bytes in *address order*. If you interpret those bytes as an integer, you must apply the correct endianness convention to reconstruct the numeric value.

A disciplined low-level mental model is therefore:

- **Bit order** is an instruction-level concept inside registers.
- **Byte order** is a storage/transfer concept at the memory interface.

9.2 Little-Endian vs Big-Endian Explained Precisely

Consider an n-byte integer value V represented by bytes:

$$V = \sum_{i=0}^{n-1} B_i \cdot 256^i$$

where:

- B_0 is the least significant byte (LSB),
- B_{n-1} is the most significant byte (MSB).

Endianness defines how these bytes are mapped to memory addresses:

Little-endian

In little-endian, the lowest address holds the least significant byte:

$$\text{mem}[p + i] = B_i$$

So significance increases with address.

Big-endian

In big-endian, the lowest address holds the most significant byte:

$$\text{mem}[p + i] = B_{n-1-i}$$

So significance decreases with address.

Concrete example (32-bit)

Let $V = \texttt{0x12345678}$ stored at base address $\texttt{p}$. The bytes are:

$$B_0 = \texttt{78},\ B_1 = \texttt{56},\ B_2 = \texttt{34},\ B_3 = \texttt{12}$$

Address offset	+0	+1	+2	+3
Little-endian bytes	`78`	`56`	`34`	`12`
Big-endian bytes	`12`	`34`	`56`	`78`

Both represent the same value, but the memory layout differs.

Endianness is per-architecture convention

Endianness is part of the ISA or the platform configuration. Some ISAs are fixed-endian; others are bi-endian (capable of operating in either mode), sometimes with separate conventions for instruction fetch vs data access depending on the platform.

9.3 Register View vs Memory View

Endianness is not a property of a register as a physical container of bits. Registers hold a bit pattern. The question of "which byte comes first" arises only when mapping between:

- a register value (a fixed-width binary pattern interpreted by the ISA), and
- a sequence of bytes in memory (observed in increasing address order).

The assembly point of view

A load instruction reads bytes from consecutive addresses and assembles them into a register value according to the platform's endianness rules. A store instruction takes a register value and decomposes it into bytes written to consecutive addresses according to those same rules. Thus:

- Memory is naturally viewed as *address order*.
- Registers are naturally viewed as *significance order* (bit positions, numeric value).

This explains a classic debugging confusion: a memory dump appears "reversed" compared to a register printout, because you are viewing the same value under two different coordinate systems.

```
# x86-64 GAS, Intel syntax

# Suppose the bytes in memory at [rdi..rdi+3] in address order are:
#   78 56 34 12
# On a little-endian system, the 32-bit value loaded is 0x12345678.
mov     eax, dword ptr [rdi]     # eax = 0x12345678
```

Byte-wise observation makes endianness visible

If you read the same 32-bit value byte-by-byte, you see the physical layout directly:

```
# x86-64 GAS, Intel syntax

mov     al, byte ptr [rdi]        # lowest address byte
mov     bl, byte ptr [rdi+1]
mov     cl, byte ptr [rdi+2]
mov     dl, byte ptr [rdi+3]
```

This sequence is independent of integer interpretation; it is simply memory address order. Endianness appears only when you *interpret* these four bytes as a 32-bit integer.

9.4 Endianness in Multi-Byte Values

Endianness affects all multi-byte scalar types stored in memory: 16-bit, 32-bit, 64-bit integers, pointers, and any packed binary fields whose meaning depends on byte position.

Single-byte values are endianness-neutral

An 8-bit value occupies one byte. There is no ordering choice, so endianness does not apply.

Multi-byte values require explicit reconstruction rules

Given bytes at `p..p+n-1`, reconstructing the integer value requires endianness:

- Little-endian reconstruction:

$$V = \sum_{i=0}^{n-1} \text{mem}[p+i] \cdot 256^i$$

- Big-endian reconstruction:

$$V = \sum_{i=0}^{n-1} \text{mem}[p+i] \cdot 256^{n-1-i}$$

Endianness does not affect arithmetic

Once a value is in a register, arithmetic and logic operate on the bit pattern according to the ISA. Endianness does not change `add`, `sub`, `and`, shifts, or comparisons on register operands. Endianness only affects the mapping between memory bytes and register values during loads/stores or explicit byte-level assembly/disassembly of values.

Endianness and partial-width access

A common low-level operation is reading part of a value (e.g., lowest byte, lowest word). On little-endian systems, the least significant byte is stored at the lowest address, so `[p]` often corresponds to the low byte of the multi-byte value. On big-endian systems, `[p]` corresponds to the high byte. This affects parsing, checksums, and manual field extraction from raw buffers.

```
# x86-64 GAS, Intel syntax

# On little-endian, this fetches the low byte of the dword at [rdi].
mov     al, byte ptr [rdi]

# This fetches the full dword, assembled by the CPU as little-endian.
mov     eax, dword ptr [rdi]
```

Endianness and bitfields

Bitfields defined in protocols or packed structures are often described in terms of bit positions within an integer, but those integers still have a byte order when serialized into memory. Correct handling requires separating:

- how the integer is serialized into bytes (endianness),
- how bits are extracted from the integer once reconstructed (bit operations).

9.5 Cross-Platform and Cross-Protocol Implications

Endianness becomes critical whenever data crosses boundaries where the producer and consumer may not share the same byte order, or where a specification defines a fixed byte order independent of the host.

Binary file formats and persistence

Any binary file format that stores multi-byte integers must specify byte order, either implicitly by platform assumption (fragile) or explicitly by standard. A file written on a little-endian host and read on a big-endian host will be misinterpreted unless conversion is performed or the format specifies a fixed canonical order.

Network protocols and canonical byte order

Many protocols define a canonical byte order for multi-byte fields so that all participants interpret the stream identically. Historically, big-endian is commonly used as a canonical order (often called “network byte order”). A host must therefore convert between host endianness and protocol endianness when encoding/decoding multi-byte fields.
The essential discipline:

- A protocol field is not “an int”; it is a specified sequence of bytes.
- Conversion is the act of mapping between that byte sequence and a host register value.

Cross-language and ABI boundaries

Within a single machine, endianness is usually consistent across languages, but interoperability still requires agreement on layout and representation for multi-byte fields in memory (structures, packed messages). When data is exchanged across machines (RPC, shared files, distributed systems), endianness must be treated as a first-class compatibility variable.

Debugging implications

When debugging:

- A register display shows a value in numeric significance order.
- A memory view shows bytes in ascending address order.

To avoid mistakes:

- Always label whether you are looking at *address order* (memory dump) or *numeric value* (register/int print).
- When reading a memory dump as an integer, explicitly apply the known byte order.
- When constructing a buffer, write bytes in the protocol/file order, not in the host’s convenient representation.

Practical rules for disciplined low-level code

- Treat endianness as a property of *serialized byte sequences*, not of abstract integers.
- Do not infer protocol/file endianness from the host ISA; assume it must be specified.
- When parsing bytes, reconstruct multi-byte fields explicitly and then apply bit operations.
- When generating bytes, build the specified byte sequence explicitly rather than storing host integers directly.

Endianness is therefore not a philosophical detail; it is a concrete rule that determines whether two systems agree on the meaning of the same bytes.

Chapter 10

Reading Machine State Correctly

10.1 Why Most Assembly Bugs Are Interpretation Bugs

Assembly programming is less about "writing instructions" and more about maintaining a correct *interpretation contract* over raw bit patterns. The CPU executes deterministic operations on fixed-width registers and updates status flags according to the ISA. The hardware does not track types, units, or programmer intent. As a result, many assembly bugs are not caused by an incorrect instruction sequence, but by applying the wrong *meaning* to a correct bit pattern.

1) Typeless registers invite semantic drift

A register is a bit-vector with a name. The CPU does not store metadata that says:

"this is signed" "this is unsigned" "this is a pointer" "this is a length"

If code reuses a register across phases of computation, the human reader often continues to interpret it according to a prior phase. This semantic drift is a primary source of defects. The bit pattern is correct; the interpretation is stale.

2) The same bits represent multiple valid truths

The same N-bit pattern can simultaneously represent:

- an unsigned integer in $[0, 2^N - 1]$,
- a signed two's complement integer in $[-2^{N-1}, 2^{N-1} - 1]$,
- a bitmask with independent boolean flags,
- a set of packed fields,
- an address (under a particular addressing model),
- or part of a wider multi-precision value.

The CPU does not choose among these. Only instruction context and programmer intent do.

3) Width mismatches dominate low-level failures

Many failures come from silently changing the active width:

- performing an operation at 32-bit width and then interpreting the destination as 64-bit,
- truncating a value by using a smaller operand size,
- accidentally zero-extending or sign-extending when moving between widths,
- mixing partial-register operations with full-register expectations.

Fixed-width arithmetic is not "approximate"; it is exact modulo 2^N. Bugs arise when the programmer forgets which N is currently in effect.

4) Signedness mismatches break control flow

A very common class of defects is choosing the wrong relational branch family after a flag-setting instruction such as `cmp`. The compare sets flags once; the *jump* chooses interpretation. Using signed jumps for naturally unsigned quantities (sizes, indices, counts, lengths) can produce catastrophic boundary-check errors.

5) Endianness and memory interpretation errors

Memory is a linear array of bytes. Multi-byte values require a byte-order rule to reconstruct. Bugs appear when developers:

- read bytes in address order but interpret as if they were in numeric significance order,
- treat protocol-serialized data as native host order without conversion,
- interpret partial byte sequences using the wrong endianness assumption.

6) "Overflow" confusion: CF vs OF

Another high-frequency defect is confusing unsigned boundary events (CF) with signed range violations (OF). The arithmetic result is always produced modulo 2^N; the only question is whether that result violates an intended domain constraint. Picking the wrong flag means checking the wrong domain.

Summary principle:

> The CPU is almost always correct about bit patterns. Most assembly bugs come from humans being wrong about meaning.

10.2 Debugging by Flags, Not by Assumptions

Status flags are the CPU's official, architectural evidence about the last relevant computation. They are not "debug symbols" and they are not optional decoration. They exist precisely to make boundary conditions observable without expensive extra computation.

1) Flags are the machine's boundary signals

Treat flags as boundary signals for fixed-width arithmetic and logic:

- **ZF** indicates the result is exactly zero for the active width.
- **CF** indicates an unsigned carry-out (or borrow semantics) at the width boundary.
- **OF** indicates a signed two's complement range violation at the representable boundary.
- **SF** mirrors the MSB of the result (useful in signed relational logic with OF).

2) Consume flags immediately

Flags are ephemeral. Many instructions clobber them. Debugging and correctness require that flag consumers (`jcc`, `setcc`, `cmovcc`, `adc`, `sbb`) appear immediately after the flag producer unless you have proven intervening instructions do not modify the relevant flags.

3) Use `cmp` and `test` as probes

`cmp` is subtraction for flags without storing the difference. `test` is bitwise AND for flags without storing the result. Both exist to observe conditions cheaply.

```
# x86-64 GAS, Intel syntax
```

```
# Probe for zero without modifying rax:
test    rax, rax
jz      .L_zero

# Probe ordering without modifying operands:
cmp     rax, rbx
je      .L_equal
```

4) Branch mnemonic selection is interpretation selection

After `cmp`, equality uses ZF for both signed and unsigned interpretations. Ordering differs:

- Unsigned ordering uses CF and ZF (`jb`, `jbe`, `ja`, `jae`).
- Signed ordering uses SF and OF (with ZF) (`jl`, `jle`, `jg`, `jge`).

```
# x86-64 GAS, Intel syntax

cmp     rax, rbx

# Unsigned:
jb      .L_u_less           # CF=1
ja      .L_u_greater        # CF=0 and ZF=0

# Signed:
jl      .L_s_less           # SF != OF
jg      .L_s_greater        # ZF=0 and SF == OF
```

5) Use CF/OF deliberately for overflow policy

If you want to implement a policy such as “reject values that overflow” you must first define:

- overflow in unsigned space → check CF,
- overflow in signed space → check OF.

```
# x86-64 GAS, Intel syntax

# Unsigned add with overflow detection:
add     rax, rbx
jc      .L_u_overflow      # CF=1 indicates wrap beyond width

# Signed add with overflow detection:
add     rax, rbx
jo      .L_s_overflow      # OF=1 indicates signed range violation
```

6) Debugging discipline: check width, then flags

When debugging unexpected control flow:

1. confirm operand width (8/16/32/64),
2. confirm the compare/operation you think is producing flags is actually the last flag-producing instruction,
3. confirm you are using the correct signed/unsigned branch family,
4. validate with the relevant flags (ZF/CF/OF/SF) rather than intuition.

10.3 Thinking in Binary Space Instead of Values

A reliable low-level mindset treats computation as transformations on a finite set of bit patterns. For an N-bit quantity, the machine operates in the space:

$$\{0,1\}^N$$

All integer arithmetic is performed modulo 2^N:

$$r = f(a, b, \ldots) \bmod 2^N$$

Nothing in the hardware produces "infinite precision" integers. The CPU produces an N-bit result and discards higher bits. That is not a failure; it is the definition.

1) Two labelings of the same space

The same space can be labeled in different ways:

- Unsigned labeling:

$$0, 1, 2, \ldots, 2^N - 1$$

- Signed (two's complement) labeling:

$$-2^{N-1}, \ldots, -1, 0, 1, \ldots, 2^{N-1} - 1$$

These are not different spaces. They are different *interpretations* of the same patterns.

2) "Overflow" is boundary crossing under a labeling

In unsigned labeling, the boundary is $2^N - 1 \to 0$ (CF reports this crossing for addition). In signed labeling, the boundary is $2^{N-1} - 1 \to -2^{N-1}$ (OF reports this crossing for addition). Thus CF and OF are not redundant. They correspond to different boundary crossings in the same binary space.

3) Value-centric thinking hides truncation

High-level language thinking often assumes arithmetic in mathematical integers with automatic range growth or checked overflow. In machine arithmetic:

- truncation is always present,
- wrapping is the default,
- and any “error” policy is software-defined using flags and checks.

4) Debugging method: derive, then interpret

A robust procedure for reasoning about a bug:

1. Fix the width N for the instruction sequence.
2. Treat operands as raw N-bit patterns.
3. Compute the N-bit result under modular arithmetic.
4. Determine which boundary signals (ZF/CF/OF/SF) should be produced.
5. Only then interpret the pattern as signed/unsigned/bitfield and verify the program logic.

This procedure prevents the classic mistake: proving something about mathematical integers while the CPU was computing modulo 2^N.

10.4 Mental Models Used by Real CPU Designers

CPU designers and low-level performance engineers reason about correctness using small, stable models that map directly to hardware reality. Adopting these models makes assembly reasoning more deterministic and less fragile.

1) Bit-vector model (typeless state)

Treat every register as a bit-vector of fixed width:

$$\texttt{reg} \in \{0,1\}^N$$

Instructions are functions on bit-vectors. This eliminates type confusion and forces you to state interpretation explicitly.

2) Width contract (the architecture is width-explicit)

Every instruction has an operand width. Designers treat width as a contract:

- results are truncated to width,
- flags are defined relative to width,
- and extension between widths must be explicit (zero-extend vs sign-extend).

3) Boundary-signal model (flags are detectors)

Designers view flags as detectors attached to arithmetic units:

- ZF is a zero detector on the result bus.
- SF is an MSB tap of the result.
- CF is the carry-out of the adder at bit $N-1$ (unsigned boundary event).
- OF is a signed range detector derived from operand MSBs and result MSB (signed boundary event).

This is why CF and OF can disagree: they detect different boundary events.

4) “Compare is subtract without writeback”

Compare operations exist to drive boundary signals for control flow without consuming a register destination. This supports efficient branching and avoids extra data movement.

5) “Memory is bytes; loads/stores are assembly rules”

Designers treat memory as bytes with addresses and treat loads/stores as defined assembly steps:

- select consecutive bytes,
- assemble/disassemble them according to endianness,
- apply alignment and access semantics.

This prevents the high-level illusion that “memory holds ints.” Memory holds bytes. Integers are reconstructed.

6) “Correctness before cleverness”

A widely used engineering discipline is to enforce correctness with explicit invariants:

- Define the signedness of each quantity at each program point.
- Define the width at each program point.
- State the overflow policy (wrap accepted vs overflow rejected) and implement it with the correct flags.
- Keep flag lifetimes short and local.

Expanded discipline checklist:

- Confirm operand width at every step (8/16/32/64).
- Treat registers as typeless bit-vectors; attach meaning only by context.
- Choose signed vs unsigned before choosing branch mnemonics.
- Use CF for unsigned boundary events; use OF for signed range violations.
- Use `cmp`/`test` as probes; consume flags immediately.
- Interpret memory dumps as byte sequences; reconstruct multi-byte values explicitly.
- Assume flags are clobbered unless you prove otherwise.
- Prefer explicit extension rules when crossing widths (zero vs sign).

Appendices

Appendix A — Binary Truth Tables and Flag Outcomes

This appendix summarizes commonly used flag outcomes as compact reference tables. All outcomes assume fixed-width two's complement arithmetic for the operand width in use. "Carry" refers to a carry out of the most significant bit of the active width. "Overflow" refers to signed two's complement range violation for that width.

Arithmetic Flag Outcome Tables

A.1 Result-Class Flags (Most Common Updates)

For many arithmetic and logical instructions, these flags follow direct properties of the result:

- **ZF (Zero Flag)**: set iff result is all zeros.
- **SF (Sign Flag)**: copy of the result MSB (active width).

Condition on Result (width N)	**ZF**	**SF**
$result = 0$	1	0
$result \neq 0$ and MSB=0	0	0
$result \neq 0$ and MSB=1	0	1

A.2 Unsigned Carry vs Signed Overflow (Addition)

For an N-bit addition $r = (a + b) \bmod 2^N$:

- **CF** is set iff $a + b \geq 2^N$ (carry out of bit $N - 1$).
- **OF** is set iff a and b have the same sign and r has the opposite sign.

Sign(a)	**Sign**(b)	**Sign**(r)	**OF**
0	0	0	0
0	0	1	1
1	1	1	0
1	1	0	1
0	1	0 or 1	0
1	0	0 or 1	0

Note: CF is independent of this sign table. CF depends on the carry out of the MSB in unsigned addition.

A.3 Signed Overflow (Subtraction)

For subtraction $r = (a - b) \bmod 2^N$, signed overflow occurs when operands have different signs and the result sign differs from the sign of a:

Sign(a)	**Sign**(b)	**Sign**(r)	**OF**
0	1	0	0
0	1	1	1
1	0	1	0
1	0	0	1
0	0	0 or 1	0
1	1	0 or 1	0

A.4 Carry/Borrow Intuition for Subtraction

Many ISAs define subtraction in terms of an internal addition:

$$a - b \equiv a + (\sim b + 1) \pmod{2^N}$$

CF for subtraction is therefore tied to the unsigned boundary behavior of this equivalent operation. The disciplined rule remains:

- use **CF** for unsigned boundary checks and multi-precision propagation,
- use **OF** for signed range violations.

Shift and Rotate Flag Behavior

Shift and rotate operations move bits; they do not define numeric overflow in the arithmetic sense. The most reliable and portable way to reason about them is:

- **CF** captures the last bit shifted out (or rotated out) for single-bit count.
- **ZF** reflects whether the result is zero.

- **SF** mirrors the MSB of the result.
- **OF** is defined only for specific counts on some ISAs (commonly count=1); do not rely on OF for multi-bit shifts unless explicitly specified by the architecture.

A.5 Single-Bit Shift-Out Bit to CF (Concept Table)

Let x be an N-bit operand, and r be the result.

Operation (count=1)	**Result bits**	**CF**
`shl/sal`	$r = (x \ll 1) \bmod 2^N$	old MSB of x
`shr`	$r = (x \gg 1)$ with 0-fill	old LSB of x
`sar`	$r = (x \gg 1)$ with sign-fill	old LSB of x
`rol`	rotate left within N bits	old MSB of x
`ror`	rotate right within N bits	old LSB of x

A.6 Minimal Demonstrations

```
# x86-64 GAS, Intel syntax

# CF receives the shifted-out bit (count = 1)
mov     al, 0b10000001
shl     al, 1               # al = 0b00000010, CF = 1 (old MSB)

mov     al, 0b10000001
shr     al, 1               # al = 0b01000000, CF = 1 (old LSB)

mov     al, 0b10000001
rol     al, 1               # al = 0b00000011, CF = 1 (old MSB)
```

Signed vs Unsigned Comparison Matrix

Comparisons are performed by setting flags (typically via `cmp`) and then interpreting them using the correct conditional branch family.

A.7 Core Flag Meanings after `cmp a,b`

Conceptually, `cmp a,b` sets flags as if computing $a - b$ (without storing the result). Then:

- **ZF=1** iff $a = b$ (bit-exact equality).
- **CF** supports unsigned ordering.
- **SF and OF** support signed ordering.

A.8 Matrix of Common Relational Jumps

Relation	Unsigned (after `cmp`)	Signed (after `cmp`)
$a = b$	`je` (ZF=1)	`je` (ZF=1)
$a \neq b$	`jne` (ZF=0)	`jne` (ZF=0)
$a < b$	`jb` (CF=1)	`jl` (SF $\neq$ OF)
$a \leq b$	`jbe` (CF=1 or ZF=1)	`jle` (ZF=1 or SF $\neq$ OF)
$a > b$	`ja` (CF=0 and ZF=0)	`jg` (ZF=0 and SF $=$ OF)
$a \geq b$	`jae` (CF=0)	`jge` (SF $=$ OF)

A.9 Minimal Pattern: Same `cmp`, Different Truths

```
# x86-64 GAS, Intel syntax

cmp     rax, rbx
```

```
# Unsigned interpretation (sizes, indices, lengths)
jb      .L_u_less           # CF=1
ja      .L_u_greater        # CF=0 and ZF=0

# Signed interpretation (two's complement integers)
jl      .L_s_less           # SF != OF
jg      .L_s_greater        # ZF=0 and SF == OF
```

Discipline rule: `cmp` produces one set of flags. The chosen conditional jump defines whether the relation is interpreted as signed or unsigned.

Appendix B — Common Misconceptions and Dangerous Assumptions

This appendix addresses several persistent misconceptions that routinely cause subtle and severe defects in low-level code. Each misconception arises from projecting high-level language semantics onto hardware that operates exclusively on fixed-width bit patterns.

"The CPU Knows This Is Signed"

Reality: The CPU has no concept of "signed" or "unsigned" values stored in registers. Registers contain bit patterns. The arithmetic and logic units operate on those patterns using fixed rules. The distinction between signed and unsigned exists only in:

- how certain status flags are interpreted (primarily CF vs OF),
- which conditional branch instructions are chosen after flag-setting operations,

- how values are extended when moving between widths (zero-extension vs sign-extension).

The same addition instruction produces the same bit pattern regardless of whether the programmer intends a signed or unsigned computation. Only the interpretation of the flags differs.

```
# x86-64 GAS, Intel syntax

# Same bits, same instruction, different truths
mov     al, 0xFF          # 255 unsigned, -1 signed
add     al, 1             # al = 0x00

# CF=1  -> unsigned overflow (255 + 1 wraps)
# OF=0  -> signed result (-1 + 1 = 0) is valid
```

Danger: Assuming the CPU "remembers" signedness leads to:

- wrong branch selection after `cmp`,
- incorrect overflow checks,
- silent acceptance of wrapped values.

Discipline: Signedness is a contract you enforce explicitly by choosing the correct extension rules, flags, and branch mnemonics at every step.

"Overflow and Carry Mean the Same Thing"

Reality: Carry and overflow detect different boundary violations in the same fixed-width arithmetic.

- **Carry Flag (CF)** reports a boundary crossing in *unsigned* arithmetic: a carry out of the most significant bit or an unsigned borrow condition.
- **Overflow Flag (OF)** reports a violation of the representable range in *signed* two's complement arithmetic.

They can be independently set or clear for the same operation.

```
# x86-64 GAS, Intel syntax

# CF=1, OF=0 (unsigned overflow only)
mov     al, 0xFF          # 255 unsigned, -1 signed
add     al, 1             # result 0x00

# CF=0, OF=1 (signed overflow only)
mov     al, 0x7F          # 127
add     al, 1             # result 0x80 (-128)
```

Danger: Using OF to validate sizes or lengths, or using CF to validate signed computations, silently checks the wrong domain.
Discipline:

- Use CF for unsigned bounds, counters, indices, and multi-precision arithmetic.
- Use OF for signed arithmetic range validation.

"Shifts Are Just Fast Multiplication"

Reality: Shifts are bit movement operations with truncation. Any arithmetic interpretation is conditional and fragile.
A left shift by k computes:

$$(x \ll k) \bmod 2^N$$

This equals multiplication by 2^k only if no significant bits are shifted out. Once a bit exits the width boundary, information is destroyed.

```
# x86-64 GAS, Intel syntax

# Safe multiplication analogy
mov     al, 0x10          # 16
shl     al, 1             # 32 (no bits lost)

# Bit destruction
mov     al, 0x90          # 144
shl     al, 1             # 32 (high bit lost)
```

Right shifts introduce an additional trap:

- `shr` fills with zeros (unsigned interpretation).
- `sar` replicates the MSB (signed interpretation).

Neither is a general-purpose division operator; both are representation-preserving bit shifts under specific assumptions.

Danger:

- assuming left shifts are always safe for signed arithmetic,
- assuming `sar` always matches signed division semantics,
- mixing shift results with inconsistent signed/unsigned interpretation.

Discipline:

- Treat shifts as bitwise transformations first, arithmetic shortcuts second.
- Prove that no significant bits are lost before treating shifts as multiplication or division.

- Choose `shr` or `sar` explicitly based on the intended interpretation.

Unifying principle:

> The CPU operates on bit patterns and width boundaries. Every dangerous assumption comes from forgetting this fact.

Appendix C — Minimal Instruction Reference (Concept-Only)

This appendix provides a compact, conceptual reference for the core instruction classes discussed in this booklet. It is not an opcode catalog and intentionally omits encoding details, latency, and microarchitectural variations. The goal is to clarify *what these instructions mean at the machine level*: how they transform fixed-width bit patterns and how they interact with status flags.

Arithmetic Instructions (Conceptual View)

Arithmetic instructions operate on fixed-width operands and always produce a truncated result modulo 2^N, where N is the active operand width. They may also update status flags to expose boundary conditions.

- **`add`** — Adds two operands and writes the N-bit result.
  - Result: $(a + b) \bmod 2^N$
  - CF: unsigned carry-out of bit $N - 1$
  - OF: signed two's complement range violation
- **`sub`** — Subtracts the second operand from the first.

  - Result: $(a - b) \bmod 2^N$
  - CF: unsigned borrow-related boundary signal (ISA-defined)
  - OF: signed two's complement range violation

- **`adc`** — Adds operands plus the current CF.
  - Used for unsigned multi-precision arithmetic
  - CF acts as carry-in from a lower word

- **`sbb`** — Subtracts operands with borrow using CF.
  - Used for unsigned multi-precision subtraction

- **`inc`** / **`dec`** — Increments or decrements by one.
  - Modifies OF/ZF/SF
  - Does not modify CF (architecturally significant)

```
# x86-64 GAS, Intel syntax

add     rax, rbx          # rax = (rax + rbx) mod 2^64
adc     rcx, rdx          # rcx = rcx + rdx + CF

sub     rax, rbx          # rax = (rax - rbx) mod 2^64
sbb     rcx, rdx          # rcx = rcx - rdx - CF
```

Comparison and Flag-Setting Instructions

Comparison instructions exist to *set flags without retaining a result*. They allow control-flow decisions and boundary checks without destroying register contents.

- **`cmp`** — Compares two operands by subtracting them conceptually.
  - Flags set as if computing $(a - b) \bmod 2^N$
  - No result is written back
  - ZF: equality ($a = b$)
  - CF: unsigned ordering support
  - SF/OF: signed ordering support
- **`test`** — Bitwise AND used only to set flags.
  - Equivalent to $a \& b$ for flags
  - Commonly used to test zero or specific bits
  - CF and OF cleared

```
# x86-64 GAS, Intel syntax

cmp     rax, rbx          # set flags for relational checks
je      .L_equal

test    rax, rax          # probe rax == 0 without modifying it
jz      .L_zero
```

Interpretation rule: `cmp` sets one set of flags; the chosen conditional jump defines whether the comparison is signed or unsigned.

Shift and Rotate Instructions

Shift and rotate instructions move bits within a fixed-width operand. Shifts discard bits; rotates preserve all bits by circulating them.

- **`shl` / `sal`** — Shift left (logical/arithmetic identical).
  - Result: $(x \ll k) \bmod 2^N$
  - CF: last bit shifted out of the MSB
  - OF: defined only for specific counts (commonly $k = 1$)
- **`shr`** — Logical right shift (zero-fill).
  - Result: $x \gg k$ with zeros shifted in
  - Used for unsigned interpretation and bitfields
- **`sar`** — Arithmetic right shift (sign-fill).
  - Replicates the original MSB
  - Preserves two's complement signed interpretation
- **`rol` / `ror`** — Rotate left / right.
  - Bits circulate within width
  - CF captures the rotated-out bit
  - No information loss

```
# x86-64 GAS, Intel syntax

mov     al, 0b10000001
shl     al, 1               # al = 00000010b, CF = 1

mov     al, 0b10000001
shr     al, 1               # al = 01000000b, CF = 1
```

```
mov     al, 0b10000001
sar     al, 1               # al = 11000000b, CF = 1

mov     al, 0b10000001
rol     al, 1               # al = 00000011b, CF = 1
```

Conceptual rule:

- Shifts are bit destruction operations with truncation.
- Rotates are bit rearrangement operations without loss.
- Any arithmetic meaning is conditional on interpretation and boundary checks.

Final discipline note: This reference describes *semantic behavior*, not performance. Correctness at the machine level comes from respecting width, interpretation, and flag meaning before any optimization considerations.