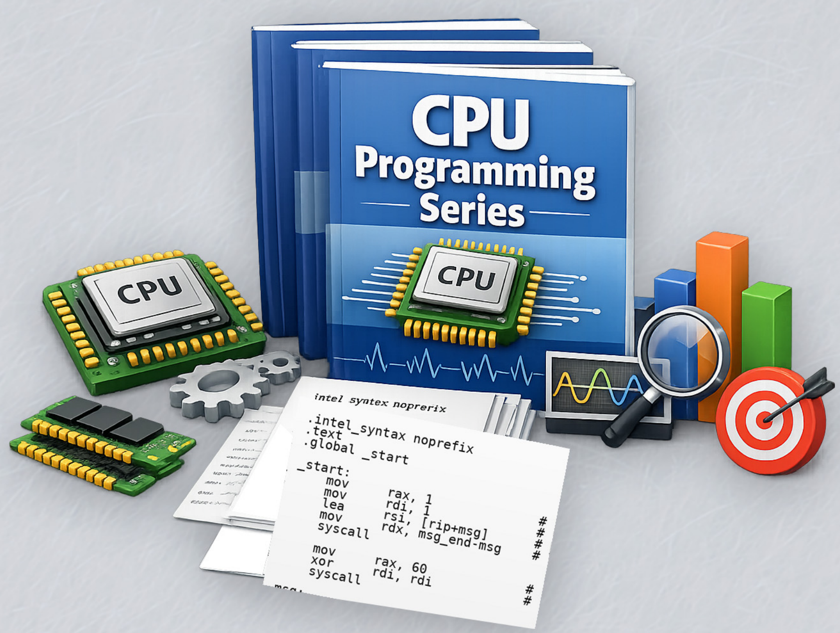
https://simplifycpp.org

CPU Programming Series

x86 Architecture Fundamentals

Registers, Flags, and Addressing Modes



```
intel syntax noprefix
```

```
.intel_syntax noprefix
.text
.global _start
_start:
    mov     rax, 1
    mov     rdi, 1
    lea     rsi, [rip+msg]
    mov     rdx, msg_end-msg
    syscall

    mov     rax, 60
    xor     rdi, rdi
    syscall
```

5

Prepared by Ayman Alheraki

CPU Programming Series

x86 Architecture Fundamentals

Registers, Flags, and Addressing Modes

Prepared by Ayman Alheraki

simplifycpp.org

January 2026

Contents

Preface

P.1 Purpose of This Booklet

This booklet provides a focused and rigorous foundation in *x86 architecture* fundamentals, concentrating on **general-purpose registers**, **FLAGS / RFLAGS**, and **addressing modes**. Its primary purpose is to establish a correct and durable mental model of how x86 instructions operate at the architectural level, independent of operating systems, calling conventions, or compiler-specific behavior.

Rather than teaching assembly as a collection of syntactic rules, this booklet explains *why* x86 behaves as it does: how registers alias, how flags encode execution state, how effective addresses are computed, and how memory operands differ fundamentally from registers. By the end of this booklet, the reader should be able to reason about instruction behavior before writing code, predict side effects, and read compiler-generated assembly with confidence.

P.2 Who This Booklet Is For

This booklet is intended for:

- C and C++ programmers seeking a deeper understanding of generated assembly
- Low-level programmers transitioning from conceptual CPU models to real-world x86

- Systems programmers preparing for ABI, calling convention, or optimization topics
- Learners who already understand basic instruction execution but lack architectural clarity

It assumes prior familiarity with:

- Binary representation and two's complement
- Registers and flags at a conceptual (ISA-independent) level
- Basic instruction flow (fetch, decode, execute, retire)

This booklet does **not** assume prior x86 assembly experience; all x86-specific behavior is introduced from first principles.

P.3 How to Read This Booklet Effectively

This booklet is designed to be read **linearly**. Each chapter builds on assumptions and invariants established earlier. Skipping chapters may result in misunderstanding subtle but critical behaviors such as partial register writes or flag-dependent logic.
Examples should be read slowly and mentally executed. For instance:

```
mov eax, 1         # writes to EAX
mov al, 0xFF       # modifies only the low 8 bits
```

The reader should pause and determine:

- The final value of RAX
- Which bits were preserved
- Why this behavior exists architecturally

Readers are encouraged to treat each instruction as a transformation of architectural state rather than as a textual operation.

P.4 Relationship to the CPU Programming Series

This booklet is the **first x86-specific entry point** in the CPU Programming Series.
It builds directly on the shared foundations established in earlier booklets:

- Instruction execution models
- Register and flag concepts
- Signed vs unsigned arithmetic

It intentionally prepares the reader for subsequent booklets covering:

- The stack and calling conventions
- ABI rules and register roles
- Memory hierarchy, caches, and performance

No ABI rules, system calls, or OS interactions are introduced here; this separation ensures architectural clarity before platform-specific complexity.

P.5 Scope, Assumptions, and Intentional Exclusions

Included in scope:

- General-purpose register structure and aliasing
- FLAGS and RFLAGS behavior
- Address calculation and addressing modes
- Register vs memory operand semantics

- Instruction-side effects visible at the ISA level

Explicitly excluded:

- Calling conventions and ABI rules
- Operating system concepts
- SIMD, floating-point, and vector registers
- Privileged instructions and control registers
- Microarchitectural optimizations and pipelines

These exclusions are intentional. The goal of this booklet is not breadth, but **architectural correctness**. Each excluded topic is addressed in later, dedicated booklets once the reader has mastered the x86 fundamentals presented here.

Chapter 1

Positioning x86 in the CPU Landscape

1.1 What "x86 Architecture" Really Means

The term *x86 architecture* refers to a long-lived and evolving **Instruction Set Architecture (ISA)** originating from the Intel 8086 and extended continuously over decades. It defines:

- The visible programmer model (registers, flags, memory model)
- The instruction encodings and semantics
- The rules for how instructions affect architectural state

x86 is not a single fixed design. Instead, it is a **backward-compatible ISA family** that accumulated features while preserving legacy behavior. This historical continuity explains why modern x86 processors still support 16-bit registers, segmented memory concepts, and partial-register semantics alongside 64-bit execution.
At the architectural level, x86 defines *what the machine must do*, not *how it is implemented*. Multiple processors with radically different internal designs can still correctly execute the same x86 program.

1.2 ISA vs Microarchitecture (Applied to x86)

The **ISA** specifies:

- Instruction behavior
- Register visibility and aliasing
- Flag side effects
- Addressing modes

The **microarchitecture** specifies:

- Pipeline depth
- Instruction decoding strategy
- Execution units
- Caches and internal scheduling

In x86, this distinction is especially important because many instructions that appear complex at the ISA level are internally decomposed into simpler micro-operations.
For example, this instruction is architecturally atomic:

```
add qword ptr [rax], 1
```

Architecturally, it means:

- Read memory at address RAX
- Add 1
- Write back the result

- Update flags

How this is internally executed is a microarchitectural concern and intentionally ignored at the ISA level. This booklet remains strictly on the ISA side.

1.3 Why x86 Feels Complex Compared to RISC

x86 often appears more complex than RISC architectures due to several architectural characteristics:

- Variable-length instructions
- Rich addressing modes
- Memory operands allowed in arithmetic
- Extensive legacy compatibility

Unlike typical RISC designs where instructions operate only on registers, x86 instructions may directly reference memory:

```
add eax, dword ptr [rbx + rcx*4 + 8]
```

This single instruction combines:

- Address calculation
- Memory load
- Arithmetic
- Flag updates

The complexity is not accidental; it reflects a design optimized historically for code density and expressive instructions. Modern processors internally translate these instructions into simpler operations, but the architectural contract remains.

1.4 Execution Model Overview (Without ABI or OS)

At the architectural level, x86 execution follows a simple and deterministic model:

- Instructions are fetched in program order
- Each instruction transforms architectural state
- Side effects are visible through registers, flags, and memory

This booklet intentionally ignores:

- System calls
- Privilege levels
- Interrupts and exceptions
- Calling conventions

For example, this instruction sequence is interpreted purely in terms of architectural state:

```
mov eax, 5
add eax, 3
```

The focus is on:

- How EAX changes
- Which flags are updated
- What guarantees the ISA provides

No assumptions are made about stack usage, function calls, or operating system interaction.

1.5 What This Booklet Covers and What It Deliberately Avoids

This booklet focuses exclusively on **x86 architectural fundamentals** required to understand real assembly behavior.
Covered topics:

- General-purpose register structure and aliasing
- FLAGS and RFLAGS semantics
- Addressing modes and effective address computation
- Register vs memory operand behavior
- Instruction-visible side effects

Deliberately excluded topics:

- ABI and calling conventions
- Stack discipline
- Operating system interaction
- SIMD and floating-point units
- Microarchitectural optimization details

This separation is intentional. A correct understanding of x86 must begin with its architectural rules before layering platform-specific conventions or performance considerations.

Chapter 2

General-Purpose Registers: Legacy to Modern

2.1 The Historical Evolution of x86 Registers

General-purpose registers in the x86 family evolved through successive architectural extensions while preserving backward compatibility. The original 16-bit model provided eight primary general-purpose registers. The 32-bit era extended these to 32-bit views while retaining the 16-bit subregister semantics. The 64-bit era extended them again to 64-bit views and expanded the register file with additional registers, while still keeping legacy naming and aliasing rules.

This historical layering explains why a single architectural register can be accessed through multiple names and widths, and why some legacy subregisters can still influence modern 64-bit code through aliasing effects.

2.2 Register Naming Across Generations (AX → EAX → RAX)

The same physical architectural register is presented through different names depending on operand size. A canonical example is the accumulator register:

- `AX` is the low 16 bits
- `EAX` is the low 32 bits
- `RAX` is the full 64 bits

The naming pattern generalizes across the legacy set:

- `BX/EBX/RBX`, `CX/ECX/RCX`, `DX/EDX/RDX`
- `SI/ESI/RSI`, `DI/EDI/RDI`, `BP/EBP/RBP`, `SP/ESP/RSP`

In 64-bit mode, additional general-purpose registers exist beyond the legacy eight, but the crucial point for correctness is that the legacy ones keep their historical subregister layout and behaviors.

2.3 8-bit, 16-bit, 32-bit, and 64-bit Views

A general-purpose register is not multiple storage locations. It is one architectural register with multiple overlapping views. For `RAX`, the relevant views are:

- `RAX`: bits 63:0
- `EAX`: bits 31:0
- `AX`: bits 15:0

- `AL`: bits 7:0
- `AH`: bits 15:8 (legacy high byte)

Understanding these overlaps is mandatory because writes through one view can preserve, overwrite, or invalidate other bits depending on the rule of the specific subregister width and generation.

2.4 Low and High Byte Registers (AL / AH and Their Caveats)

The low byte register (e.g. `AL`) refers to bits 7:0 of the parent register. The high byte register (e.g. `AH`) refers to bits 15:8, and exists for historical reasons.
High-byte registers have important caveats:

- They overlap with the low 16-bit region and can interact badly with partial updates.
- They complicate instruction encoding in 64-bit mode and are constrained in contexts where certain prefix encodings are used.

For correctness, treat high-byte registers as legacy artifacts and prefer low-byte registers or 32-bit writes when possible.

2.5 Register Aliasing and Partial Register Updates

Because subregisters overlap, a write to a smaller width does not necessarily define the full register. This can lead to stale high bits remaining unchanged.
Example: writing only the low 8 bits does not change the upper bits:

```
mov rax, 0x1122334455667788
mov al, 0xFF              # updates only bits 7:0
# RAX becomes 0x11223344556677FF
```

Similarly, writing only the low 16 bits preserves bits 63:16:

```
mov rbx, 0xAAAAAAAA55555555
mov bx,  0x1234           # updates only bits 15:0
# RBX becomes 0xAAAAAAAA55551234
```

A critical architectural rule in 64-bit mode is that writing a 32-bit subregister (e.g. `EAX`) clears the upper 32 bits of the corresponding 64-bit register (e.g. `RAX`) to zero. This is a correctness and code-generation rule that many compilers use intentionally.

```
mov rax, 0xFFFFFFFF00000000
mov eax, 1                # write to EAX zero-extends into RAX
# RAX becomes 0x0000000000000001
```

This behavior is not sign-extension. It is architectural zeroing of the upper half when writing a 32-bit GPR in 64-bit mode.

2.6 Zero-Extension vs Sign-Extension Behavior

Zero-extension and sign-extension are distinct concepts. Zero-extension fills new high bits with zeros. Sign-extension copies the sign bit (most significant bit of the source width) into the new high bits.

The 32-bit write rule above is zero-extension by definition. Sign-extension is obtained only through specific instructions that explicitly sign-extend a smaller value.

Zero-extension example:

```
mov eax, 0x80000000        # upper 32 bits cleared
# RAX = 0x0000000080000000
```

Sign-extension example using explicit sign-extend instruction forms:

```
mov eax, 0x80000000            # EAX is negative if interpreted as signed
 ↪  32-bit
cdqe                          # sign-extend EAX into RAX
# RAX = 0xFFFFFFFF80000000
```

Zero-extension from a smaller width can also be done explicitly:

```
mov al, 0xFF                   # AL = 255
movzx eax, al                  # zero-extend AL into EAX, then upper half
 ↪  cleared
# RAX = 0x00000000000000FF
```

Sign-extension from a smaller width can be done explicitly:

```
mov al, 0x80                   # AL has sign bit set if treated as int8
movsx eax, al                  # sign-extend AL into EAX, then upper half
 ↪  cleared
# RAX = 0x00000000FFFFFF80
```

To obtain a fully sign-extended 64-bit result from an 8-bit source, combine sign-extension and then extend to 64-bit if needed:

```
mov al, 0x80
movsx eax, al                  # EAX = 0xFFFFFF80
cdqe                          # RAX = 0xFFFFFFFFFFFFFF80
```

2.7 Practical Examples

Same register, different widths

```
mov rax, 0x0123456789ABCDEF
mov ax,  0x1111             # modifies low 16 only
# RAX = 0x0123456789AB1111

mov eax, 0x22222222         # modifies low 32 and clears high 32
# RAX = 0x0000000022222222

mov al,  0x33               # modifies low 8 only
# RAX = 0x0000000022222233
```

This sequence demonstrates three different architectural behaviors for the same register depending on operand width.

Unexpected data corruption

A common source of bugs is assuming a partial write defines the full register. Example: the high bits remain from a previous value, contaminating an address computation.

```
mov rdi, 0x00007FFF00000000
mov di,  0x1234             # updates low 16 only
# RDI = 0x00007FFF00001234  (high bits unchanged)
```

If the programmer expected `RDI = 0x1234`, the code is wrong and may point to an unintended memory region.
Correct approach using a 32-bit write when a clean value is required:

```
mov edi, 0x1234             # zero-extends into RDI in 64-bit mode
# RDI = 0x0000000000001234
```

Performance side effects

Partial register updates can create unnecessary dependencies because the processor must merge old and new bits for the final architectural value. A classic pattern is writing `AL` and later using `RAX` as a full register value.

```
mov rax, 0
mov al,  1                    # partial update
add rax, 2                    # uses full register after partial update
```

A more robust and typically preferable pattern is writing the 32-bit subregister to fully define the register value and clear upper bits:

```
mov eax, 1                    # defines low 32 and clears high 32
add rax, 2
```

Rule of thumb for modern 64-bit code:

- Prefer 32-bit writes when you want a clean known value in a GPR.
- Use 8-bit and 16-bit subregisters only when the narrow operation is intentional and the follow-up usage does not assume a fully defined 64-bit value.

Chapter 3

Special Roles of General-Purpose Registers

3.1 Implicit Register Usage in Instructions

Many x86 instructions allow flexible register selection, but a significant subset uses **implicit operands**: registers that are not written in the assembly text yet are required by the architectural definition of the instruction. These implicit operands are part of the ISA contract and must be understood for correctness.
Common forms of implicit register usage include:

- **Fixed implicit registers** required by the opcode (e.g., accumulator and data registers in multiply/divide families)
- **Implicit address registers** in string instructions (source and destination pointers)
- **Implicit counters** (e.g., repetition count register in repeated string operations)

A practical consequence is that instruction semantics can depend on registers that appear nowhere in the line of assembly. This affects correctness, register allocation strategy, and code reading.

3.2 Accumulator-centric Instructions

Historically, x86 used the accumulator register as a primary operand for many instructions. Modern x86 supports generalized forms for most operations, but accumulator-centric forms still exist and matter for:

- Legacy compatibility
- Specific encodings and special forms
- Multiply/divide semantics that inherently use accumulator-related pairs

The most important accumulator-centric families are the **multiply and divide instructions**, whose architectural definition binds them to accumulator registers.
Unsigned multiply:

```
mul rbx                    # implicit: RAX * RBX -> RDX:RAX
```

Signed multiply:

```
imul rbx                   # implicit: RAX * RBX -> RDX:RAX
↪  (one-operand form)
```

Unsigned divide:

```
div rcx                    # implicit: RDX:RAX / RCX -> quotient in RAX,
↪  remainder in RDX
```

Signed divide:

```
idiv rcx                  # implicit: RDX:RAX / RCX -> quotient in RAX,
↪  remainder in RDX
```

The implicit use of RAX and RDX is not optional. The programmer must arrange inputs and preserve/expect outputs accordingly.

3.3 Index and Base Register Conventions (Conceptual Only)

This booklet does not teach ABI or calling conventions, but some register roles are so widespread in practice that a conceptual model is useful:

- A **base** register commonly holds a stable pointer (e.g., base of an object, base of a data region).
- An **index** register commonly holds a varying offset (e.g., loop index scaled by element size).

At the ISA level, these are not special classes of registers. Any general-purpose register can be used as base or index in addressing forms, with legal addressing constraints defined by the ISA. The point of the convention is readability and reasoning: base tends to be stable, index tends to vary.
Example conceptual pattern:

```
mov rbx, rdi                # RBX as base pointer
mov rcx, 7                  # RCX as index
mov eax, dword ptr [rbx + rcx*4 + 8]
```

Here:

- RBX acts as base
- RCX acts as index scaled by 4 (e.g., array of 32-bit elements)
- 8 acts as displacement (e.g., header offset)

3.4 Registers Commonly Used for Addressing

x86 effective addresses are formed from optional components:

$$\text{EA} = \text{Base} + \text{Index} \times \text{Scale} + \text{Displacement}$$

Not all combinations are legal in all forms, but conceptually:

- **Base** is typically a pointer register used as a starting address.
- **Index** is typically used for scaled traversal.
- **Displacement** is an immediate constant offset encoded in the instruction.

Some registers are frequently used as pointers by convention in real code, but architecturally:

- Any GPR can hold an address value.
- Any legal base and index registers can participate in effective address calculation.

Examples of addressing in increasing richness:
Base only:

```
mov eax, dword ptr [rdi]
```

Base + displacement:

```
mov eax, dword ptr [rdi + 16]
```

Base + index*scale:

```
mov eax, dword ptr [rdi + rcx*4]
```

Base + index*scale + displacement:

```
mov eax, dword ptr [rdi + rcx*4 + 8]
```

3.5 Examples

Instructions that require specific registers

Example 1: 64-bit unsigned multiplication produces a 128-bit result in `RDX:RAX`.

```
mov rax, 0xFFFFFFFFFFFFFFFF
mov rbx, 2
mul rbx                 # RDX:RAX = RAX * RBX
```

Architectural meaning:

- Input multiplicand is implicitly `RAX`
- Input multiplier is explicit operand (`RBX`)
- Output is split across `RDX` (high) and `RAX` (low)

Example 2: Unsigned division consumes `RDX:RAX` as the dividend.

```
mov rax, 100
xor edx, edx            # clear high half of dividend
mov rcx, 9
div rcx                 # quotient -> RAX, remainder -> RDX
```

If `RDX` is not prepared correctly, the dividend is not what the programmer expects, and the divide may raise a fault due to overflow of the quotient.

Example 3: Signed division requires correct sign-extension into `RDX`.

```
mov rax, -100
cqo                     # sign-extend RAX into RDX:RAX
mov rcx, 9
idiv rcx                # quotient -> RAX, remainder -> RDX
```

Here `cqo` is essential: it prepares the implicit high half of the dividend for signed division.

Instructions that silently assume registers

Example 1: String move uses implicit source and destination pointers.

```
mov rsi, rbx              # source pointer
mov rdi, rdx              # destination pointer
mov ecx, 16               # element count
rep movsb                 # copies ECX bytes from [RSI] to [RDI]
```

Architectural meaning:

- Source is implicitly `RSI`
- Destination is implicitly `RDI`
- Count is implicitly `RCX/ECX`
- Pointers are updated implicitly as the instruction proceeds

The instruction line shows only `movsb`, yet the semantics depend on three registers.

Example 2: Compare string bytes uses implicit pointers and updates them.

```
mov rsi, rbx
mov rdi, rdx
mov ecx, 8
repe cmpsb                # repeats while equal and ECX != 0
```

Even without explicit operands, the instruction:

- reads memory at `RSI` and `RDI`
- updates flags based on comparisons
- advances pointers

- decrements the count register

Example 3: Loop uses an implicit counter register.

```
mov ecx, 3
L1:
# work
loop L1                  # decrements ECX and jumps if ECX != 0
```

Here the instruction implicitly depends on `ECX`. Replacing `ECX` with another register is not possible for `loop`; it is architecturally fixed.

Practical rules for reading and writing

- Always consult the implicit operand set of an instruction family when reasoning about correctness.
- Treat multiply/divide as `RAX/RDX`-centric operations that must be staged carefully.
- Treat string instructions as operating on `RSI/RDI/RCX` even when they appear operand-free.
- Prefer explicit forms (`cmp/jcc`, normal load/store loops) unless a string instruction is intentionally chosen and fully understood.

Chapter 4

FLAGS and RFLAGS: The Hidden State Machine

4.1 What FLAGS Actually Represent

`RFLAGS` is an architectural register that records outcomes of many operations and controls certain execution behaviors. It is best understood as a compact **state vector** updated as a side effect of instructions. For most integer code, the most important bits are the **status flags** that summarize properties of a result:

- whether the result is zero
- whether the result is negative in two's complement
- whether an unsigned carry or borrow occurred
- whether a signed overflow occurred

Flags are not computed from "meaning" such as signedness. They are computed from bit-level arithmetic rules and are later interpreted by instructions such as conditional branches and conditional moves.

4.2 Status Flags vs Control Flags (Conceptual Boundary)

Status flags are typically updated by arithmetic, logical, compare, and shift/rotate instructions and are used for decision-making:

- `ZF` (Zero Flag)
- `SF` (Sign Flag)
- `CF` (Carry Flag)
- `OF` (Overflow Flag)

Control flags affect processor behavior (direction of string operations, interrupt enable, trap/step behavior, and others). This booklet focuses on status flags and only treats control flags conceptually to avoid OS, privilege, and debugging topics.
A key boundary rule in this booklet:

- Status flags are about **results and comparisons**.
- Control flags are about **execution mode and control behavior**.

4.3 Core Arithmetic Flags: ZF, SF, CF, OF

The four most important arithmetic status flags:

ZF (Zero Flag)

`ZF=1` if the result is exactly zero, otherwise `ZF=0`.

```
mov eax, 1
sub eax, 1              # EAX = 0
# ZF = 1
```

SF (Sign Flag)

`SF` copies the most significant bit of the result (the sign bit in two's complement). It does not mean "negative" unless the value is interpreted as signed.

```
mov al, 0x7F
add al, 1               # AL = 0x80
# SF = 1 (MSB set)
```

CF (Carry Flag)

`CF` indicates an **unsigned carry out** from the most significant bit on addition, or an **unsigned borrow** on subtraction. CF is the unsigned overflow indicator.
Unsigned carry example:

```
mov al, 0xFF
add al, 1               # AL = 0x00
# CF = 1
```

Unsigned borrow example:

```
mov al, 0x00
sub al, 1               # AL = 0xFF
# CF = 1   (borrow)
```

OF (Overflow Flag)

`OF` indicates **signed overflow** in two's complement arithmetic. It is set when the signed result cannot be represented in the operand width.

Signed overflow example (8-bit):

```
mov al, 0x7F               # +127
add al, 1                  # result 0x80 (-128 in int8)
# OF = 1
```

No signed overflow (even if CF may change):

```
mov al, 0xFF               # -1 in int8
add al, 1                  # 0
# OF = 0
```

4.4 Logical and Shift Instruction Flag Effects

Logical operations (`and`, `or`, `xor`, `test`) compute bitwise results and update some status flags:

- `ZF` and `SF` reflect the logical result.
- `CF` and `OF` are cleared for common logical operations.

Example:

```
mov eax, 0
or eax, 0                  # EAX = 0
# ZF = 1, SF = 0, CF = 0, OF = 0
```

`test` is especially important because it performs an AND for flags only without storing the result:

```
test eax, eax              # checks if EAX is zero without modifying EAX
# ZF = 1 iff EAX == 0
```

Shift instructions update flags based on the shift result and the bit shifted out:

- CF becomes the last bit shifted out.
- ZF and SF reflect the result.
- OF is defined in specific cases (notably for shift-by-1) and should not be assumed for general shift counts.

Example: left shift sets CF from the bit shifted out:

```
mov al, 0x80
shl al, 1                  # AL = 0x00, bit7 shifted out
# CF = 1, ZF = 1, SF = 0
```

Example: right shift sets CF from the bit shifted out:

```
mov al, 0x01
shr al, 1                  # AL = 0x00, bit0 shifted out
# CF = 1, ZF = 1, SF = 0
```

4.5 Flag Preservation and Clobbering

Flags are not stable variables. Any instruction that updates flags can destroy the condition produced by a previous instruction.
Clobber example:

```
cmp eax, ebx               # sets flags based on (EAX - EBX)
add ecx, 1                 # clobbers flags
je  equal                  # now tests flags from ADD, not from CMP
```

Correct pattern: branch immediately after producing flags, or explicitly recompute:

```
cmp eax, ebx
je  equal
add ecx, 1
```

Some instructions do not affect flags, but you must never assume this without knowing the instruction class. A safe mental model:

- Arithmetic, compare, logical, shift, and many bit-manipulation instructions update flags.
- Data movement instructions (`mov`, `lea`) do not update flags.

4.6 Why Flags Are Not "Boolean Results"

Flags are not a single true/false output. They are multiple independent bits that describe different properties of the same operation.
A common misunderstanding is to treat CF or ZF as if they encode the entire meaning of a comparison. In reality:

- `ZF` answers: is the result zero
- `CF` answers: did unsigned arithmetic carry/borrow
- `OF` answers: did signed arithmetic overflow
- `SF` answers: is the MSB of the result set

The CPU does not decide signedness. The programmer chooses whether to interpret the flags as signed or unsigned by selecting different conditional branches.

4.7 Practical Examples

Same operation, different flags

Add `1` to `0xFF` in 8-bit:

```
mov al, 0xFF
add al, 1                # AL = 0x00
# ZF = 1, CF = 1, SF = 0, OF = 0
```

Add `1` to `0x7F` in 8-bit:

```
mov al, 0x7F             # +127
add al, 1                # 0x80 (-128)
# ZF = 0, CF = 0, SF = 1, OF = 1
```

Both are “add 1”, but CF and OF differ because CF tracks unsigned carry while OF tracks signed overflow.

Signed vs unsigned interpretation

The same `cmp` instruction sets the same flags, but different conditional branches interpret them differently.
Example: compare 0xFF with 1 (8-bit views):

```
mov al, 0xFF             # 255 unsigned, -1 signed
cmp al, 1                # flags based on (AL - 1)
```

Unsigned interpretation uses `ja/jb` (above/below). Signed interpretation uses `jg/jl` (greater/less).
Conceptual consequence:

- Unsigned: 255 is above 1
- Signed: -1 is less than 1

This is not two different compares. It is one compare with two interpretations.

Common flag-logic mistakes

Mistake 1: Branching after clobbering flags:

```
cmp eax, 0
add ebx, 1
jne not_zero              # wrong: tests ADD flags, not CMP flags
```

Correct:

```
cmp eax, 0
jne not_zero
add ebx, 1
```

Mistake 2: Using the wrong condition for signed vs unsigned:

```
cmp eax, ebx
jl  less_signed           # signed less-than
```

If values represent sizes or addresses, signed comparisons are usually wrong. Use unsigned conditions (`jb`/`jae`/`ja`) when comparing values that are conceptually non-negative quantities.
Mistake 3: Expecting `mov` to “keep flags meaningful” across sequences:

```
cmp eax, ebx
mov ecx, edx              # does not modify flags
je  equal                 # OK
```

But inserting almost any arithmetic or logical instruction between compare and branch typically breaks the logic:

```
cmp eax, ebx
xor ecx, ecx              # clobbers flags
je  equal                 # wrong
```

Correct approach: branch immediately or recompute flags with another compare or test:

```
cmp eax, ebx
je  equal
xor ecx, ecx
```

Chapter 5

Signed vs Unsigned: Flags in Context

5.1 Why the CPU Does Not Know "Signedness"

At the ISA level, the CPU operates on **bit patterns**. A register contains bits, and arithmetic instructions transform those bits using fixed rules. The hardware does not carry metadata such as "this value is signed" or "this value is unsigned". Signedness is a **human interpretation** of the same bit pattern.
Example: the 8-bit pattern `0xFF` can mean:

- 255 if interpreted as unsigned
- -1 if interpreted as signed two's complement

The CPU performs the same subtraction for a `cmp` regardless of interpretation:

```
mov al, 0xFF                # bits: 11111111
cmp al, 1                   # computes (AL - 1) for flags only
```

What changes is how later instructions (branches, cmov, setcc) interpret the resulting flags.

5.2 How Instructions Interpret Flags Differently

The same flags can be interpreted as **unsigned** or **signed** conditions. x86 provides different condition codes for each interpretation.
After `cmp a, b` (conceptually a subtraction `a - b` for flags):

- Unsigned comparisons use primarily `CF` and `ZF`
- Signed comparisons use relationships among `SF` and `OF`, and also `ZF`

Key unsigned condition meanings:

- `jb` (below): `CF=1`
- `jae` (above or equal): `CF=0`
- `ja` (above): `CF=0 and ZF=0`
- `jbe` (below or equal): `CF=1 or ZF=1`

Key signed condition meanings:

- `jl` (less): `SF != OF`
- `jge` (greater or equal): `SF == OF`
- `jg` (greater): `ZF=0 and SF==OF`
- `jle` (less or equal): `ZF=1 or SF!=OF`

These are not different comparisons. They are different interpretations of one flag state.

5.3 Conditional Logic Based on Flags

Flags become useful only when consumed by conditional instructions. The most common consumers are:

- conditional jumps (`jcc`)
- conditional moves (`cmovcc`)
- conditional sets into a byte (`setcc`)

Example using `setcc` to materialize a boolean:

```
cmp eax, ebx
setl dl                 # DL = 1 if signed less-than (SF != OF), else
↪  0
```

Example using `cmovcc` to avoid branches:

```
mov eax, esi            # candidate A
mov ebx, edi            # candidate B
cmp eax, ebx
cmovg eax, ebx          # if signed greater, EAX = EBX
```

Conditional logic is correct only if the chosen condition matches the intended meaning of the data.

5.4 Overflow vs Carry — Revisited in x86

`CF` and `OF` answer different questions.

CF (Carry Flag) indicates **unsigned overflow** for addition, or **unsigned borrow** for subtraction. It is the correct flag for reasoning about arithmetic on non-negative quantities such as sizes, indices, addresses, and modular arithmetic.

OF (Overflow Flag) indicates **signed overflow** in two's complement. It is relevant when values are intended to represent signed integers.
Unsigned overflow example (8-bit):

```
mov al, 0xFF
add al, 1                 # AL = 0x00
# CF = 1, OF = 0
```

Signed overflow example (8-bit):

```
mov al, 0x7F              # +127
add al, 1                 # 0x80 (-128)
# CF = 0, OF = 1
```

Subtraction borrow vs signed overflow:

```
mov al, 0x00
sub al, 1                 # AL = 0xFF
# CF = 1 (borrow), OF = 0
```

The rule of thumb:

- Use `CF`-based conditions for unsigned comparisons.
- Use `SF/OF`-based conditions for signed comparisons.

5.5 Examples

Comparing signed integers

Suppose `EAX` and `EBX` hold signed 32-bit integers. We want: if `EAX < EBX` then branch.

```
cmp eax, ebx
jl  less_signed           # signed less-than: SF != OF
```

Why this is correct:

- `cmp` sets flags as if computing `eax - ebx`
- `jl` uses `SF != OF` which matches signed ordering in two's complement

Example where signed and unsigned disagree:

```
mov eax, -1             # 0xFFFFFFFF
mov ebx, 1
cmp eax, ebx
jl  signed_less         # taken: -1 < 1
ja  unsigned_above      # also considered: 0xFFFFFFFF > 1 (would be
 ↪  taken if executed)
```

Only one of these is meaningful depending on the intended interpretation of the data.

Comparing unsigned integers

Suppose `EAX` and `EBX` represent sizes or indices. We want: if `EAX < EBX` then branch, unsigned.

```
cmp eax, ebx
jb  below_unsigned      # unsigned below: CF = 1
```

Example where unsigned meaning is essential:

```
mov eax, 0x80000000     # 2147483648 unsigned, -2147483648 signed
mov ebx, 1
cmp eax, ebx
jb  u_below             # not taken: 0x80000000 is not below 1
 ↪  unsigned
jl  s_less              # taken: negative is less than 1 signed
```

If these values represent memory sizes, the signed branch is wrong.

Subtle logic bugs caused by wrong condition

Bug 1: comparing sizes (unsigned) using signed condition:

```
cmp eax, ebx
jl  smaller                 # wrong if EAX/EBX are sizes or indices
```

Correct:

```
cmp eax, ebx
jb  smaller                 # correct for sizes (unsigned below)
```

Bug 2: checking for negative using `CF` instead of `SF`:

```
test eax, eax
jc  negative                # wrong: CF meaning here is not ``negative''
```

Correct:

```
test eax, eax
js  negative                # correct: SF reflects sign bit of result
```

Bug 3: detecting signed overflow using `CF`:

```
add eax, ebx
jc  overflow                # wrong for signed overflow
```

Correct for signed overflow detection:

```
add eax, ebx
jo  overflow                # OF = 1 indicates signed overflow
```

Bug 4: using `jg/jl` for pointers or addresses:

```
cmp rax, rbx
jg  higher_address          # wrong for address ordering
```

Correct (treat addresses as unsigned):

```
cmp rax, rbx
ja  higher_address      # correct unsigned above
```

Practical rule set:

- If the quantity can never be negative (sizes, indices, addresses), use unsigned conditions.
- If the quantity represents a signed integer domain, use signed conditions.
- To detect unsigned overflow use `jc`; to detect signed overflow use `jo`.

Chapter 6

Addressing Modes: The Heart of x86 Power

6.1 What an Addressing Mode Really Is

An **addressing mode** defines how an instruction forms an **effective address** (EA) when it accesses memory. In x86, the effective address is computed from up to four components:

$$\text{EA} = \text{Base} + (\text{Index} \times \text{Scale}) + \text{Displacement}$$

Where:

- **Base** is a general-purpose register holding an address
- **Index** is a general-purpose register holding an offset
- **Scale** is a small integer multiplier (1,2,4,8)
- **Displacement** is an immediate constant encoded in the instruction

This addressing model is one of x86's defining features because it makes common data-structure access patterns expressible in a single memory operand.

6.2 Register-Direct vs Memory-Based Access

x86 instructions can operate on:

- **register operands**: the value is already in a register
- **memory operands**: the value is stored in memory, accessed via an effective address

Register-direct example:

```
add eax, ebx              # EAX = EAX + EBX, no memory access
```

Memory-based example:

```
add eax, dword ptr [rbx]  # EAX = EAX + *(uint32_t*)RBX
```

The second form implies a memory read as part of the instruction's semantics. The addressing mode specifies how the address in brackets is computed.

6.3 Base + Offset Addressing

The simplest useful memory form uses a base register plus an immediate displacement (offset):

$$\text{EA} = \text{Base} + \text{Displacement}$$

Examples:

```
mov eax, dword ptr [rdi]         # EA = RDI
mov eax, dword ptr [rdi + 16]    # EA = RDI + 16
mov byte ptr [rbx + 1], 0x7F     # EA = RBX + 1
```

This pattern naturally models fields within a struct, local variables within a stack frame (conceptually), and fixed offsets from a base pointer.

6.4 Base + Index + Scale + Displacement

The most general x86 addressing form is:

$$\text{EA} = \text{Base} + (\text{Index} \times \text{Scale}) + \text{Displacement}$$

Examples:

```
mov eax, dword ptr [rdi + rcx*4]         # EA = RDI + RCX*4
mov eax, dword ptr [rdi + rcx*4 + 8]     # EA = RDI + RCX*4 + 8
mov rdx, qword ptr [rbx + rsi*8 + 24]    # EA = RBX + RSI*8 + 24
```

This directly expresses:

- array indexing (`base + index*element_size`)
- table lookups
- member access with a header offset (`+ displacement`)

6.5 Legal and Illegal Addressing Combinations

Architecturally, x86 does not allow arbitrary arithmetic expressions in memory operands. Only specific patterns are legal.
Legal:

- `[base]`
- `[base + disp]`

- `[base + index*scale]`
- `[base + index*scale + disp]`
- `[index*scale + disp]` (base omitted)

Not legal (must be rewritten):

- two scaled indexes: `[rdi + rsi*4 + rcx*2]`
- general multiplication by non-scale constants: `[rdi + rcx*3]`
- nested brackets: `[[rdi] + 8]`
- arbitrary expressions: `[rdi + (rcx + rdx)*4]`

When an expression is illegal, compute part of it using `lea` or arithmetic instructions, then use a legal addressing mode.
Example rewrite for a factor of 3:

```
lea rdx, [rcx + rcx*2]           # RDX = RCX*3
mov eax, dword ptr [rdi + rdx*4] # EA = RDI + (RCX*3)*4 = RDI +
↪  RCX*12
```

6.6 Scale Factors and Their Hardware Meaning

The scale factor in x86 addressing is limited to **1, 2, 4, or 8**. This matches common element sizes:

- 1 byte
- 2 bytes
- 4 bytes

- 8 bytes

This is not a general multiplication facility. It is a specialized addressing feature to support indexed addressing efficiently.
Examples:

```
mov al,  byte ptr [rdi + rcx*1]   # byte array
mov ax,  word ptr [rdi + rcx*2]   # 16-bit array
mov eax, dword ptr [rdi + rcx*4]  # 32-bit array
mov rax, qword ptr [rdi + rcx*8]  # 64-bit array
```

If the element size is not 1/2/4/8, you must compute `index * element_size` separately.
Example element size 24:

```
lea rdx, [rcx*8]                # RDX = RCX*8
lea rdx, [rdx + rcx*16]         # RDX = RCX*24  (8 + 16)
mov rax, qword ptr [rdi + rdx]  # EA = base + index*24
```

6.7 Practical Examples

Stack-like access

Without introducing ABI rules, stack-like access can be modeled as using a pointer register that moves and indexing relative to it.
Push-like pattern (conceptual):

```
sub rsp, 8                       # reserve space
mov qword ptr [rsp], rax          # store value at top
```

Pop-like pattern (conceptual):

```
mov rax, qword ptr [rsp]          # load value
add rsp, 8                        # release space
```

Local-slot access pattern (conceptual):

```
mov dword ptr [rsp + 12], 7        # store at fixed offset from a
↪   stack pointer
mov eax, dword ptr [rsp + 12]      # load from same slot
```

Array traversal

Assume `RDI` holds the base of an array of 32-bit integers and `RCX` is an index.
Single element load:

```
mov eax, dword ptr [rdi + rcx*4]   # A[RCX]
```

Sequential traversal using pointer increment:

```
mov rbx, rdi                       # RBX = current pointer
mov ecx, 4                         # count
L1:
mov eax, dword ptr [rbx]           # load *RBX
add rbx, 4                         # advance to next element
dec ecx
jne L1
```

Traversal using index and scale:

```
xor ecx, ecx                       # i = 0
L2:
mov eax, dword ptr [rdi + rcx*4]   # A[i]
inc ecx
cmp ecx, 4
jne L2
```

Struct-like memory layouts

Assume RDI points to a struct-like object with fields at fixed offsets. Example layout (conceptual):

- offset 0: 32-bit id
- offset 4: 32-bit flags
- offset 8: 64-bit pointer
- offset 16: 32-bit length

Access fields:

```
mov eax, dword ptr [rdi + 0]       # id
mov ebx, dword ptr [rdi + 4]       # flags
mov rdx, qword ptr [rdi + 8]       # pointer
mov ecx, dword ptr [rdi + 16]      # length
```

Combining base+index*scale+disp for an array inside a struct: Assume at offset 32 begins an array of 64-bit entries and RCX is an index.

```
mov rax, qword ptr [rdi + rcx*8 + 32]   # obj->entries[RCX]
```

This single addressing mode expresses:

- base pointer to object (RDI)
- index scaling for 8-byte elements (RCX*8)
- member offset within object (+32)

Chapter 7

Memory Operands: Reading and Writing Correctly

7.1 Memory Is Not a Register

A register operand names an architectural register that already contains a value. A memory operand names an **address** and requires memory access to read or write the value stored at that address. Even when written in one instruction, a memory operand implies work that a register operand does not.
Register form:

```
add eax, ebx                    # purely register operation
```

Memory form:

```
add eax, dword ptr [rbx]        # reads memory at address RBX
```

A bracket expression in Intel syntax is not a value. It is a dereference: `[addr]` means "the memory located at addr". Confusing `addr` with `[addr]` is a core source of bugs.

7.2 Operand Size and Explicitness

Every x86 memory access has a size. The CPU must know whether it is reading or writing 1, 2, 4, or 8 bytes (or other sizes for specific instruction families). When the size is not implied by another operand, it must be made explicit.
If the destination is a register, the size is implied:

```
mov eax, [rdi]                  # implies 4-byte load
mov rax, [rdi]                  # implies 8-byte load
```

If the destination is memory, the size is often ambiguous unless specified:

```
mov [rdi], 1                    # ambiguous without size
```

Correct explicit forms:

```
mov byte  ptr [rdi], 1
mov word  ptr [rdi], 1
mov dword ptr [rdi], 1
mov qword ptr [rdi], 1
```

A safe rule:

- If an instruction has a memory operand and no register operand that forces size, specify `byte/word/dword/qword ptr`.

7.3 Why Memory-to-Memory Operations Are Limited

Most integer arithmetic and logic instructions do not permit both operands to be memory. This is a defining characteristic of x86 instruction forms:

- common pattern: **reg, reg** or **reg, mem** or **mem, reg**

- generally illegal: **mem, mem**

Illegal example:

```
add dword ptr [rdi], dword ptr [rsi]    # invalid: memory-to-memory
↪  add
```

Correct rewrite uses a register as a temporary:

```
mov eax, dword ptr [rsi]
add dword ptr [rdi], eax
```

Some specialized instructions do support memory-to-memory-like behavior (string moves, some compare forms), but this is the exception, not the rule.

7.4 Alignment Considerations (Conceptual)

Alignment is the relationship between an address and the size of the data being accessed. For example, a 4-byte integer is naturally aligned when its address is a multiple of 4.
This booklet treats alignment conceptually:

- Misalignment does not necessarily make an access illegal at the ISA level for ordinary integer loads/stores.
- Misalignment can affect performance and may matter more for certain instruction families and platforms.
- Correctness problems arise when code assumes alignment that is not guaranteed by the data layout.

Conceptual demonstration (not ABI-specific):

```
mov eax, dword ptr [rdi]          # expects 4 bytes from address RDI
mov eax, dword ptr [rdi + 1]      # still reads 4 bytes, but at a
 ↪  misaligned address
```

From an ISA viewpoint, both are defined as a 4-byte read at the specified effective address. From a design viewpoint, the second should only be used intentionally with full awareness of the data layout.

7.5 Implicit Memory Access in Instructions

Some instructions access memory implicitly, even if the line does not look like a classic `[addr]` form, or even if registers are not explicitly listed.
Examples of implicit memory access include:

- stack operations: `push`, `pop`
- string operations: `movsb`, `cmpsb` (with `rep` prefixes)
- call/return: `call`, `ret`

Stack-like implicit memory access:

```
push rax                          # writes memory at [RSP-8], updates
 ↪  RSP
pop rax                           # reads memory at [RSP], updates RSP
```

String move implicit memory access:

```
rep movsb                         # reads from [RSI], writes to [RDI],
 ↪  updates RSI/RDI/RCX
```

These instructions do not mention explicit memory operands, but their semantics include memory reads and writes.

7.6 Examples

Correct memory operand usage

Example 1: load a 32-bit value, update it, store it back:

```
mov eax, dword ptr [rdi]
add eax, 10
mov dword ptr [rdi], eax
```

Example 2: increment a memory counter (single memory operand is allowed):

```
inc dword ptr [rdi]
```

Example 3: store an immediate with explicit size:

```
mov qword ptr [rdi + 8], 0
mov dword ptr [rdi + 16], 1
```

Example 4: compute address with `lea` and then access memory:

```
lea rbx, [rdi + rcx*4 + 8]
mov eax, dword ptr [rbx]
```

Ambiguous operand size errors

Ambiguous store of an immediate:

```
mov [rdi], 1                      # ambiguous size, must be specified
```

Correct:

```
mov byte ptr [rdi], 1
```

Ambiguous bit operation on memory without size:

```
and [rdi], 0xFF                  # ambiguous size
```

Correct:

```
and byte ptr [rdi], 0xFF
```

Ambiguous compare against an immediate:

```
cmp [rdi], 0                     # ambiguous size
```

Correct:

```
cmp dword ptr [rdi], 0
```

Hidden memory accesses

Hidden memory access 1: stack operation:

```
push rax                         # implicit write to memory
```

Hidden memory access 2: call and return:

```
call target                      # implicit push of return address to
↪   memory
ret                              # implicit pop of return address
↪   from memory
```

Hidden memory access 3: string copy:

```
mov rsi, rbx
mov rdi, rdx
mov ecx, 8
rep movsb                        # implicit reads and writes to
↪   memory
```

Hidden memory access 4: compare string bytes:

```
mov rsi, rbx
mov rdi, rdx
mov ecx, 8
repe cmpsb                          # implicit reads from memory and
↪  flag updates
```

Practical checklist:

- Always distinguish address (`rdi`) from dereference (`[rdi]`).
- Always ensure memory operand size is known; specify it when not implied.
- Expect most arithmetic instructions to require at least one register operand.
- Treat push/pop/call/ret and string ops as memory operations even when operands are implicit.

Chapter 8

`mov` vs `lea`: The Most Misunderstood Pair

8.1 What `mov` Actually Does

`mov` transfers a value from a source operand to a destination operand. The source can be an immediate, a register, or a memory operand. If the source is memory, `mov` performs a **memory read**. If the destination is memory, `mov` performs a **memory write**.
Register to register:

```
mov rax, rbx                # RAX = RBX
```

Immediate to register:

```
mov eax, 123                # EAX = 123, upper half of RAX cleared in
↪  64-bit mode
```

Memory to register (load):

```
mov eax, dword ptr [rdi]    # EAX = *(uint32_t*)RDI
```

Register to memory (store):

```
mov dword ptr [rdi], eax     # *(uint32_t*)RDI = EAX
```

`mov` does not compute addresses as a result. It either copies a register/immediate value, or it dereferences memory and copies the loaded value.

8.2 What `lea` Actually Does

`lea` means **Load Effective Address**. It computes an address using the x86 addressing-mode formula and writes the computed integer result into a register. It does **not** read memory.

```
lea rax, [rdi + rcx*4 + 8]   # RAX = RDI + RCX*4 + 8
```

The bracket expression in `lea` is not a dereference. It is an arithmetic expression restricted to legal addressing forms:

$$\text{EA} = \text{Base} + (\text{Index} \times \text{Scale}) + \text{Disp}$$

`lea` is an address-calculation instruction that uses the addressing hardware as an integer adder with optional scaled index.

8.3 Address Calculation vs Data Access

A central mental model:

- `mov rax, [addr]` reads data from memory at `addr`
- `lea rax, [addr]` computes the value `addr` as an integer

Example:

```
mov rax, [rdi]               # RAX = memory_at(RDI)
lea rbx, [rdi]               # RBX = RDI
```

Even though both use brackets, only `mov` dereferences.
This difference is crucial when reading assembly:

- `mov` with brackets is a memory access.
- `lea` with brackets is integer arithmetic.

8.4 Why `lea` Is Not a Load Instruction

The term “load” in `lea` is historical and refers to loading the computed address into a register, not loading memory contents.
If `lea` were a load, this would make sense:

```
lea eax, dword ptr [rdi]     # incorrect idea: "load from memory"
```

But architecturally `lea` does this:

```
lea rax, [rdi]               # RAX = RDI
```

It never touches memory. Therefore:

- `lea` cannot fault due to reading invalid memory contents.
- `lea` can still fault if the instruction encoding itself is invalid, but not due to dereferencing.

8.5 Performance and Semantics Differences

Semantics first:

- `mov` with a memory source depends on memory and produces the loaded data.

- `lea` depends only on registers and immediates and produces an address-sized integer result.

Practical implications:

- `lea` is frequently used for pointer arithmetic and index computation without affecting flags.
- `mov` is used to transfer values and to actually load/store memory.

Flags:

- `lea` does not update status flags.
- `mov` does not update status flags.

Because neither updates flags, both are often used between `cmp/test` and a conditional jump without clobbering the condition.
Code-generation reality:

- Compilers commonly use `lea` as a compact way to compute `base + index*scale + disp`.
- Compilers avoid `lea` when they need flags from arithmetic instructions.

A critical caution:

- `lea` can compute many arithmetic forms efficiently, but it is not a general multiply.
- It cannot compute `index*3` directly unless represented through allowed scale combinations (1,2,4,8) and addition.

8.6 Practical Examples

Pointer arithmetic

Compute `p + 32` into `RAX` without touching memory:

```
lea rax, [rdi + 32]          # RAX = RDI + 32
```

Dereferencing the pointer to read the value at `p + 32`:

```
mov eax, dword ptr [rdi + 32]   # EAX = *(uint32_t*)(RDI + 32)
```

Compute `p + i*8` for an array of 64-bit elements:

```
lea rax, [rdi + rcx*8]       # RAX = base + index*8
```

Array indexing

Load `A[i]` where elements are 32-bit:

```
mov eax, dword ptr [rdi + rcx*4]    # EAX = A[i]
```

Compute address of `A[i]` for later use:

```
lea rbx, [rdi + rcx*4]              # RBX = &A[i]
```

Then store through that computed address:

```
mov dword ptr [rbx], 0              # A[i] = 0
```

Compute address and load can be separated cleanly:

```
lea rbx, [rdi + rcx*4 + 8]          # &A[i] plus a header offset
mov eax, dword ptr [rbx]            # load value
```

Accidental bugs from misuse

Bug 1: using `lea` when a load is required:

```
lea eax, [rdi]              # EAX = RDI, not *(uint32_t*)RDI
```

Correct load:

```
mov eax, dword ptr [rdi]    # EAX = memory_at(RDI)
```

Bug 2: using `mov` when the address is required:

```
mov rax, qword ptr [rdi + 16]   # loads value at offset 16
```

If the goal was to compute the address `RDI + 16`:

```
lea rax, [rdi + 16]             # computes address only
```

Bug 3: assuming `lea` performs bounds checking:

```
lea rax, [rdi + rcx*8]          # computes address even if RCX is out
↪  of bounds
```

`lea` does not validate memory. Any later memory access is what may fault:

```
mov rdx, qword ptr [rax]         # this is where invalid address
↪  causes a fault
```

Bug 4: expecting `lea` to compute arbitrary expressions:

```
# desired: RAX = RDI + RCX*12
# illegal as a single addressing mode because scale cannot be 12
```

Correct decomposition:

```
lea rdx, [rcx + rcx*2]           # RDX = RCX*3
lea rax, [rdi + rdx*4]           # RAX = RDI + (RCX*3)*4 = RDI +
↪  RCX*12
```

Practical rules:

- Use `mov reg, [mem]` when you need the value stored in memory.
- Use `lea reg, [expr]` when you need the computed address or scaled index arithmetic.
- Brackets mean dereference for most instructions, but mean arithmetic expression for `lea`.

Chapter 9

Instruction Encoding Implications (Conceptual)

9.1 Why Instruction Form Matters

In x86, the same high-level intent can be expressed using different instruction forms. Instruction **form** includes:

- which operands are registers vs memory
- which addressing mode is used
- whether immediates are embedded
- operand width selection (8/16/32/64)
- whether the encoding requires prefixes and extended fields

Even when two sequences are logically equivalent, they can differ in:

- code size (number of bytes fetched and decoded)
- number and type of implicit operations (loads, stores, merges)
- dependency patterns on registers and memory

This chapter stays conceptual: it teaches **what changes** when form changes, without relying on specific microarchitecture tables.

9.2 Register vs Memory Operand Cost

A register operand is a value already inside the architectural register file. A memory operand implies address generation plus a load or store.
Compare these:
Register form:

```
add eax, ebx                    # EAX = EAX + EBX
```

Memory form:

```
add eax, dword ptr [rbx]        # EAX = EAX + *(uint32_t*)RBX
```

The second form implies:

- effective address computation (base/index/scale/disp)
- a memory read
- then the arithmetic

x86 allows many `reg, mem` forms to reduce code size, but memory operands introduce:

- dependency on memory latency

- sensitivity to cache and alignment
- more work for the front-end (decode, address generation)

A common performance-friendly shape is to separate the load from the arithmetic when reusing data:

```
mov edx, dword ptr [rbx]
add eax, edx
```

This avoids repeating the memory operand if the value is used multiple times.

9.3 Addressing Mode Complexity and Decode Cost

x86 memory operands can encode complex effective addresses:

$$\text{EA} = \text{Base} + (\text{Index} \times \text{Scale}) + \text{Disp}$$

A simple address:

```
mov eax, dword ptr [rdi]
```

A more complex address:

```
mov eax, dword ptr [rdi + rcx*4 + 128]
```

Conceptually, complex addressing can increase:

- instruction length (more bytes for SIB and displacement)
- decode work (more fields to parse)
- address-generation work (more add/scale components)

This does not mean complex addressing is always bad. It means it is not free, and it can become a bottleneck when overused, especially in tight loops.
A common structural improvement is to hoist repeated address computation:

```
lea rbx, [rdi + 128]
mov eax, dword ptr [rbx + rcx*4]
```

Here the constant displacement is paid once, then indexing is simpler and repeated.

9.4 Why Some Instructions "Look Simple but Aren't"

Some single-line x86 instructions can represent multiple conceptual operations.
Example: incrementing a memory location:

```
inc dword ptr [rdi]
```

Conceptually this implies:

- load 32-bit value from memory
- add 1
- store back
- update flags (except CF for `inc`)

Another example: arithmetic directly from memory:

```
add eax, dword ptr [rdi + rcx*4]
```

Conceptually:

- compute effective address
- load from memory

- add to register
- update flags

These instructions may appear compact, but compactness often combines multiple hidden costs.
A separate hidden complexity is partial-register behavior. An instruction that writes an 8-bit subregister may appear small but can create dependencies on the previous full-register value:

```
mov al, 1                       # partial update
add rax, 2                      # later use of full register depends
 ↪  on merge
```

A 32-bit write often defines the value more cleanly:

```
mov eax, 1                      # defines low 32, clears high 32 in
 ↪  64-bit mode
add rax, 2
```

9.5 Examples

Equivalent logic, different encodings

Example 1: adding a constant to a register. Two forms are logically equivalent:

```
add eax, 1
```

```
inc eax
```

They both increment the register by 1, but they differ in flag behavior:

- `add` updates all standard arithmetic flags including CF

- `inc` does not update CF

Thus the forms are not interchangeable if later code depends on CF.
Example 2: comparing against zero.

```
cmp eax, 0
```

```
test eax, eax
```

Both can be used to branch on zero/non-zero, but `test` is purely bitwise AND for flags and is often a preferred idiom for zero checks because it does not need an immediate constant.
Example 3: computing scaled arithmetic with `lea` vs explicit arithmetic.

```
lea rax, [rdi + rcx*8]
```

```
mov rax, rcx
shl rax, 3
add rax, rdi
```

Both compute `RDI + RCX*8`. The first is one instruction that does not change flags. The second is multiple instructions and shifts may update flags, affecting later conditional logic unless planned.

Simpler instruction forms outperforming complex ones

Example 1: avoid repeated complex addressing in a loop.
Complex addressing repeated each iteration:

```
xor ecx, ecx
L1:
mov eax, dword ptr [rdi + rcx*4 + 128]
add ebx, eax
```

```
inc ecx
cmp ecx, edx
jne L1
```

Simplify by hoisting base computation:

```
lea rsi, [rdi + 128]
xor ecx, ecx
L2:
mov eax, dword ptr [rsi + rcx*4]
add ebx, eax
inc ecx
cmp ecx, edx
jne L2
```

Conceptual benefit:

- shorter and simpler memory operand in the loop body
- the constant displacement is paid once

Example 2: reuse a loaded value instead of reloading from memory.
Reloading twice:

```
add eax, dword ptr [rdi]
sub ebx, dword ptr [rdi]
```

Load once, reuse:

```
mov ecx, dword ptr [rdi]
add eax, ecx
sub ebx, ecx
```

Conceptual benefit:

- one memory read instead of two
- clearer dependency structure

Example 3: prefer clean-width writes to avoid partial-register merge effects.
Partial byte write followed by 64-bit use:

```
mov rax, 0
mov al, 1
add rax, 2
```

Cleaner 32-bit write:

```
mov eax, 1
add rax, 2
```

Conceptual benefit:

- the register value is fully defined in the common 64-bit execution model
- fewer hidden dependencies on previous register contents

Practical rule set:

- Choose instruction forms based on correctness first (especially flags).
- Prefer forms that avoid repeated memory operands when values are reused.
- Prefer addressing simplification when the same base and displacement repeat in loops.
- Prefer operand widths that fully define registers when later code uses the full register.

Chapter 10

Common Beginner and Intermediate Mistakes

10.1 Partial Register Writes

A partial register write occurs when code writes only a subregister (`AL`, `AH`, `AX`, `EAX`) while later treating the full register (`RAX`) as fully defined. In 64-bit mode, writing `EAX` clears the upper 32 bits of `RAX`, but writing `AL` or `AX` does not.
Bug pattern: low-byte write leaves stale high bits.

```
mov rax, 0x1122334455667788
mov al,  1                    # only changes bits 7:0
# RAX = 0x1122334455667701
```

Correct pattern: if a clean value is required, prefer a 32-bit write.

```
mov eax, 1                    # defines low 32 and clears high 32
# RAX = 0x0000000000000001
```

Bug pattern: address computation contaminated by stale upper bits.

```
mov rdi, 0x00007FFF00000000
mov di,  0x1234              # only changes low 16 bits
# RDI = 0x00007FFF00001234   # not 0x1234
```

Correct:

```
mov edi, 0x1234              # zero-extends into RDI
```

10.2 Assuming Flags Persist

Flags are overwritten by many common instructions. A comparison is only meaningful until the next flag-clobbering instruction executes.
Bug pattern: clobbering flags between `cmp` and `jcc`.

```
cmp eax, ebx
add ecx, 1                   # clobbers flags
je  equal                    # now tests ADD flags, not CMP flags
```

Correct: branch immediately after producing flags.

```
cmp eax, ebx
je  equal
add ecx, 1
```

Bug pattern: using `xor reg, reg` for zeroing without noticing it also sets flags.

```
cmp eax, 0
xor edx, edx                 # clobbers flags
jne not_zero                 # wrong
```

Correct: either reorder, or re-establish flags.

```
xor edx, edx
cmp eax, 0
jne not_zero
```

10.3 Confusing Address with Value

In Intel syntax, a register like `RDI` is an integer value. `[RDI]` is the memory located at the address stored in `RDI`. Confusing them produces wrong results and hard-to-debug crashes.
Bug pattern: expecting `mov rax, rdi` to load memory.

```
mov rax, rdi                # copies the pointer value, no memory
 ↪  read
```

Correct load:

```
mov rax, qword ptr [rdi]     # loads value from memory
```

Bug pattern: computing an address but accidentally loading from memory.

```
mov rax, qword ptr [rdi + 16]  # loads value at offset 16
```

Correct address computation:

```
lea rax, [rdi + 16]            # computes address only
```

10.4 Misreading Memory Operand Syntax

Intel syntax uses `[base + index*scale + disp]` to express an effective address, not a high-level expression language. The bracket contents are address components, not nested dereferences.
Correct interpretation:

```
mov eax, dword ptr [rdi + rcx*4 + 8]
# EA = RDI + RCX*4 + 8
# EAX = *(uint32_t*)EA
```

Bug pattern: thinking `[rdi]` is the same as `rdi`.

```
lea rax, [rdi]                # RAX = RDI
mov rax, [rdi]                # RAX = memory_at(RDI)
```

Bug pattern: missing operand-size explicitness on memory destinations.

```
mov [rdi], 1                  # ambiguous size
```

Correct:

```
mov byte ptr [rdi], 1
```

Bug pattern: attempting illegal memory-to-memory arithmetic.

```
add dword ptr [rdi], dword ptr [rsi]  # invalid form
```

Correct:

```
mov eax, dword ptr [rsi]
add dword ptr [rdi], eax
```

10.5 Overusing Complex Addressing Modes

Complex addressing is powerful, but overusing it can make code harder to read and can increase front-end work (longer encodings, more address components). The most common mistake is repeating a complex base+index+scale+disp inside a loop when part of it is loop-invariant.

Overused pattern:

```
xor ecx, ecx
L1:
mov eax, dword ptr [rdi + rcx*4 + 128]
add ebx, eax
inc ecx
cmp ecx, edx
jne L1
```

Better pattern: hoist the invariant displacement.

```
lea rsi, [rdi + 128]
xor ecx, ecx
L2:
mov eax, dword ptr [rsi + rcx*4]
add ebx, eax
inc ecx
cmp ecx, edx
jne L2
```

Another overuse is trying to encode expressions that do not match legal forms, leading to incorrect or overly long sequences. A disciplined approach:

- Use addressing modes to express natural data layouts (arrays, structs).
- Hoist invariants out of loops.
- Use `lea` to precompute intermediate addresses when needed.

10.6 Real-World Bug Patterns

Bug pattern 1: wrong signed/unsigned condition

Using signed branches for quantities that should be unsigned (sizes, indices, addresses) causes subtle logic errors.

Wrong:

```
cmp eax, ebx
jl  smaller                 # wrong for sizes
```

Correct:

```
cmp eax, ebx
jb  smaller                 # unsigned below
```

Bug pattern 2: divide without preparing implicit registers

Unsigned division uses `RDX:RAX` as the dividend. Forgetting to clear `RDX` changes the dividend and can raise an exception.

Wrong:

```
mov rax, 100
mov rcx, 9
div rcx                     # wrong if RDX is not known
```

Correct:

```
mov rax, 100
xor edx, edx                # clear high half
mov rcx, 9
div rcx
```

Bug pattern 3: accidental flag dependency

`inc/dec` do not update CF, while `add/sub` do. Mixing them can break code that depends on CF.

Bug example:

```
stc                           # CF = 1
inc eax                       # CF unchanged
jc  carry_path                # still taken, not what "overflow check"
 ↪  intended
```

If carry behavior matters, use explicit `add` and check `jc`.

Bug pattern 4: hidden memory access assumptions

Assuming an instruction is "register-only" when it implicitly accesses memory causes performance surprises and correctness bugs.

Example:

```
push rax                      # implicit memory write at [RSP-8]
pop  rbx                      # implicit memory read at [RSP]
```

Bug pattern 5: inconsistent operand size across loads/stores

Loading 32-bit and storing 64-bit (or vice versa) unintentionally corrupts adjacent fields.

Bug:

```
mov eax, dword ptr [rdi]      # loads 4 bytes
mov qword ptr [rdi], rax      # stores 8 bytes, overwriting next field
```

Correct: match sizes intentionally.

```
mov eax, dword ptr [rdi]
mov dword ptr [rdi], eax
```

Practical checklist:

- Prefer 32-bit writes to fully define GPRs in 64-bit mode.
- Branch immediately after `cmp/test` unless you are sure intervening instructions do not clobber flags.
- Always distinguish address (`rdi`) from dereference (`[rdi]`).
- Always make memory operand size explicit when not implied.
- Hoist loop-invariant address components out of tight loops.
- Treat divide and string/stack instructions as having implicit operands and side effects.

Chapter 11

Mental Models for x86 Assembly

11.1 Think in Data Flow, Not Syntax

The fastest way to gain correctness in x86 is to stop reading instructions as text and start reading them as **data flow transformations**. Each instruction consumes inputs (registers, memory, immediates) and produces outputs (registers, memory, flags).
A practical way to read a line is:

- What values does it **read** (including implicit reads)?
- What state does it **write** (including flags and implicit writes)?
- Does it **define** a full value or only partially update it?

Example: do not read this as “add from memory”. Read it as a flow graph:

```
add eax, dword ptr [rdi + rcx*4]
```

Data flow interpretation:

- input: `RDI`, `RCX`, memory at `RDI + RCX*4`, `EAX`

- output: `EAX`, flags

This framing prevents mistakes such as assuming the address expression is a value, or assuming flags persist.

11.2 Separate Address Computation from Data Access

x86 allows memory operands that combine address calculation with data access. This is expressive but can hide intent and costs. A disciplined mental model separates these phases.
Combined form:

```
mov eax, dword ptr [rdi + rcx*4 + 8]
```

Separated form:

```
lea rbx, [rdi + rcx*4 + 8]       # address computation only
mov eax, dword ptr [rbx]          # data access only
```

Both are correct, but the separated form makes it explicit that:

- `lea` is arithmetic only, no memory access
- `mov` is the operation that touches memory

This separation is especially useful in loops, in debugging, and when validating bounds or pointer correctness.

11.3 Read Instructions as Micro-Operations

Even without microarchitecture details, it is useful to mentally decompose instructions into ISA-visible micro-steps. This improves correctness and helps predict side effects.
Example: memory increment

```
inc dword ptr [rdi]
```

Mental decomposition:

- EA = RDI
- tmp = load32(EA)
- tmp = tmp + 1
- store32(EA, tmp)
- update flags (note: CF not updated by `inc`)

Example: compare is subtraction for flags only

```
cmp eax, ebx
```

Mental decomposition:

- tmp = EAX - EBX
- update flags based on tmp
- EAX and EBX unchanged

Example: division has implicit operands

```
div rcx
```

Mental decomposition:

- dividend = RDX:RAX (128-bit)
- divisor = RCX
- quotient -¿ RAX

- remainder -¿ RDX
- fault if quotient does not fit

This model prevents silent mistakes such as forgetting to clear or sign-extend `RDX` before division.

11.4 How to Predict Side Effects Before Writing Code

Before choosing an instruction, predict its side effects by answering these questions.

Does it write flags?

Many instructions clobber flags. If you need flags from a previous compare, do not place a flag-writing instruction in between.
Wrong:

```
cmp eax, 0
add ebx, 1
jne not_zero                # wrong: tests ADD flags
```

Correct:

```
cmp eax, 0
jne not_zero
add ebx, 1
```

Does it access memory implicitly?

Some instructions access memory without explicit brackets.

```
push rax                   # implicit store to memory
pop  rbx                   # implicit load from memory
call target                # implicit store of return address
ret                        # implicit load of return address
```

Does it fully define a register or partially update it?

Byte and word writes preserve upper bits of the full register. 32-bit writes in 64-bit mode clear the upper 32 bits.
Bug:

```
mov rax, 0x1122334455667788
mov al, 1                  # partial write
```

If the intent is to set `RAX=1`:

```
mov eax, 1                 # full definition in common 64-bit model
```

Is the comparison signed or unsigned?

The CPU does not know signedness. You choose it by selecting conditions.
Unsigned size compare:

```
cmp eax, ebx
jb  smaller                # unsigned below
```

Signed integer compare:

```
cmp eax, ebx
jl  smaller                # signed less-than
```

11.5 Checklist for Correct x86 Instruction Design

Use this checklist before finalizing an instruction sequence.

- **Operand meaning:** Are operands values or addresses? Is `[reg]` intentionally a dereference?
- **Operand size:** Is the memory operand size explicit when not implied by a register?
- **Register definition:** Will any later use depend on upper bits that were not defined due to partial writes?
- **Flags:** Does any instruction between `cmp/test` and `jcc/cmov/setcc` clobber flags?
- **Signedness:** Are you using signed conditions (`jl/jg`) only for signed domains, and unsigned conditions (`jb/ja`) for sizes/addresses?
- **Implicit operands:** Does any instruction silently consume or produce registers (e.g. `mul/div`, string ops, `loop`)?
- **Memory form discipline:** Are you avoiding illegal memory-to-memory arithmetic and using a temporary register when required?
- **Addressing reuse:** In loops, are loop-invariant address components hoisted (e.g. constant displacement moved out)?
- **`mov` vs `lea`:** Are you using `lea` only for address arithmetic and `mov` for actual memory data transfer?
- **Hidden accesses:** Are you accounting for memory touched by `push/pop/call/ret/rep` instructions?

A short discipline that prevents most mistakes:

- Write the intended data-flow in your head.
- Identify every read, every write, and every implicit side effect.
- Then choose the instruction form that matches that model.

Appendices

Appendix A — Minimal Register and Flag Reference

A.1 General-Purpose Register Map

In 64-bit x86, general-purpose registers (GPRs) form a unified architectural register file with multiple overlapping views. Each name refers to a portion of the same physical register storage.
Legacy registers extended to 64-bit:

- `RAX`, `RBX`, `RCX`, `RDX`
- `RSI`, `RDI`, `RBP`, `RSP`

Additional registers available in 64-bit mode:

- `R8`, `R9`, `R10`, `R11`
- `R12`, `R13`, `R14`, `R15`

Architecturally, all general-purpose registers are equivalent. Any register may hold integer data, pointer values, counters, or addresses.
Example usage:

```
mov rax, rbx
lea r10, [rdi + 32]
add r12, r13
```

A.2 Register Width Relationships

Each general-purpose register exposes multiple overlapping width views. For the accumulator register:

- `RAX` : bits 63:0
- `EAX` : bits 31:0
- `AX` : bits 15:0
- `AL` : bits 7:0
- `AH` : bits 15:8

The same structure applies to:

- `RBX / EBX / BX / BL / BH`
- `RCX / ECX / CX / CL / CH`
- `RDX / EDX / DX / DL / DH`

Registers `R8`–`R15` support:

- 64-bit (`R8`)
- 32-bit (`R8D`)
- 16-bit (`R8W`)

- 8-bit low (R8B)

Critical architectural rules:

- Writing a 32-bit subregister clears the upper 32 bits of the 64-bit register.
- Writing an 8-bit or 16-bit subregister preserves upper bits.

Examples:

```
mov rax, 0x1122334455667788
mov eax, 1
# RAX = 0x0000000000000001
```

```
mov rax, 0x1122334455667788
mov al, 1
# RAX = 0x1122334455667701
```

A.3 Core Flags and Their Meanings

The RFLAGS register contains status and control flags. For integer and control-flow logic, the core status flags are:

- ZF (Zero Flag): set if the result is zero
- SF (Sign Flag): copy of the most significant bit of the result
- CF (Carry Flag): unsigned carry or borrow
- OF (Overflow Flag): signed overflow in two's complement

Examples:
Zero result:

```
mov eax, 1
sub eax, 1
# ZF = 1
```

Unsigned carry:

```
mov al, 0xFF
add al, 1
# CF = 1
```

Signed overflow:

```
mov al, 0x7F
add al, 1
# OF = 1
```

Sign flag reflects MSB:

```
mov al, 0x80
test al, al
# SF = 1
```

Comparison interpretation:

Unsigned:

```
cmp eax, ebx
jb  below_unsigned
```

Signed:

```
cmp eax, ebx
jl  less_signed
```

The processor does not track signedness. Signed or unsigned meaning is determined entirely by which conditional instruction consumes the flags.

Appendix B — Addressing Mode Patterns

B.1 Common Addressing Templates

x86 effective addresses follow a fixed template:

$$\text{EA} = \text{Base} + (\text{Index} \times \text{Scale}) + \text{Displacement}$$

where `Scale` is restricted to `1, 2, 4, 8`. The following templates are the patterns you will see constantly in real code.

Base only:

```
mov eax, dword ptr [rdi]                # EA = RDI
```

Base + displacement:

```
mov eax, dword ptr [rdi + 16]           # EA = RDI + 16
```

Index*scale + displacement (no base):

```
mov eax, dword ptr [rcx*4 + 8]          # EA = RCX*4 + 8
```

Base + index*scale:

```
mov eax, dword ptr [rdi + rcx*4]        # EA = RDI + RCX*4
```

Base + index*scale + displacement:

```
mov eax, dword ptr [rdi + rcx*4 + 8]    # EA = RDI + RCX*4 + 8
```

Struct field access (fixed offsets):

```
mov eax, dword ptr [rdi + 0]            # field0
mov ebx, dword ptr [rdi + 4]            # field1
mov rdx, qword ptr [rdi + 8]            # field2
```

Array element access (index scaled by element size):

```
mov eax, dword ptr [rdi + rcx*4]          # int32 A[i]
mov rax, qword ptr [rdi + rcx*8]          # int64 A[i]
```

Two-dimensional array (row-major) common shape:

```
# A[row][col], element size = 4
# offset = row*stride + col*4
lea rbx, [rdi + rsi*stride]               # rsi = row
mov eax, dword ptr [rbx + rcx*4]          # rcx = col
```

When a required multiplier is not 1/2/4/8, compilers synthesize it using `lea` and shifts. Example: index*12 = index*(8+4):

```
lea rdx, [rcx*4]                           # RDX = RCX*4
lea rdx, [rdx + rcx*8]                     # RDX = RCX*12
mov eax, dword ptr [rdi + rdx]             # EA = base + index*12
```

B.2 What Compilers Tend to Generate

Compilers favor patterns that are:

- legal in a single addressing mode
- easy to schedule and reuse
- predictable in register usage

Typical compiler-generated shapes include:

Hoisting constant displacement out of loops:

```
lea rsi, [rdi + 128]                        # base adjusted once
xor ecx, ecx
L1:
mov eax, dword ptr [rsi + rcx*4]            # simpler loop address
add ebx, eax
inc ecx
cmp ecx, edx
jne L1
```

Using `lea` for pointer arithmetic without flags:

```
lea rax, [rdi + rcx*8 + 24]                 # compute &obj->arr[i] with
 ↪  header offset
```

Materializing complex scaling using `lea` chains:

```
# compute base + index*24
lea rdx, [rcx + rcx*2]                      # RDX = RCX*3
lea rax, [rdi + rdx*8]                      # RAX = base + (RCX*3)*8 =
 ↪  base + RCX*24
```

Choosing pointer-increment loops for linear walks:

```
mov rbx, rdi                                 # current = base
mov ecx, edx                                 # count
L2:
mov eax, dword ptr [rbx]
add rbx, 4                                   # advance by element size
dec ecx
jne L2
```

Using 32-bit index registers in 64-bit code when safe, because 32-bit writes/uses are common and zero-extension rules help keep register values well-defined:

```
mov eax, dword ptr [rdi + rcx*4]            # 32-bit load, RCX as index
```

B.3 Patterns You Should Recognize Instantly

These are the addressing shapes that should become automatic to parse.

Pattern 1: **field access** in a struct-like layout:

```
mov eax, dword ptr [rdi + 16]              # obj->field at +16
```

Pattern 2: **array indexing**:

```
mov eax, dword ptr [rdi + rcx*4]           # A[i], 4-byte elements
```

Pattern 3: **array-of-struct** field access:

```
# element_size = 24, field_offset = 8
lea rdx, [rcx + rcx*2]                     # RDX = i*3
mov rax, qword ptr [rdi + rdx*8 + 8]       # base + i*24 + 8
```

Pattern 4: **pointer-chasing** (load pointer, then dereference it):

```
mov rax, qword ptr [rdi + 8]               # load ptr = obj->next
mov eax, dword ptr [rax + 16]              # load ptr->field
```

Pattern 5: **stack-like slots** (conceptual, no ABI assumptions):

```
mov dword ptr [rsp + 12], 7                # local slot at fixed
 ↪  offset
mov eax, dword ptr [rsp + 12]
```

Pattern 6: **address vs value** distinction:

```
lea rax, [rdi + 32]                        # computes address
mov rax, qword ptr [rdi + 32]              # loads value from memory
```

Instant recognition rules:

- `[base + disp]` is almost always “field at fixed offset”.

- `[base + index*4]` or `[base + index*8]` is almost always "array element".
- `lea reg, [..]` is address arithmetic, not a memory load.
- If you see a multiply not in {1,2,4,8}, expect an `lea` chain or shift+add synthesis.

Appendix C — Preparation for Next Booklets

C.1 Readiness for Stack and Calling Conventions

Before studying stacks and calling conventions, the reader must be fluent in the architectural rules that govern registers, flags, addressing modes, and implicit operands. The stack is not a special mechanism; it is memory accessed through disciplined address updates and implicit instruction behavior.
Key readiness points:

- Understanding that `RSP` is an ordinary register whose value is interpreted as an address.
- Recognizing that `push`, `pop`, `call`, and `ret` are memory operations with implicit address updates.
- Being able to reason about partial register writes and their impact on pointer correctness.

Conceptual stack behavior using explicit memory operations:

```
sub rsp, 8
mov qword ptr [rsp], rax      # manual push
```

Equivalent implicit form:

```
push rax                     # implicit store and RSP update
```

Conceptual return address handling:

```
call target                    # implicit push of return address
# ...
ret                            # implicit pop into instruction pointer
```

A reader is ready for calling conventions when they can:

- Predict how `RSP` changes across instructions.
- Identify which registers are read or written implicitly by control-transfer instructions.
- Understand that conventions are rules layered on top of these architectural mechanisms.

C.2 Readiness for Memory Hierarchy and Caches

Understanding caches and memory hierarchy requires a precise mental model of when memory is accessed and how often. This booklet prepares the reader by emphasizing the difference between register operands and memory operands.
Key readiness points:

- Recognizing every instruction that touches memory, explicitly or implicitly.
- Understanding that complex addressing modes still result in a single memory access.
- Knowing that repeated memory operands imply repeated memory accesses unless values are cached in registers.

Example of repeated memory access:

```
add eax, dword ptr [rdi]
add ebx, dword ptr [rdi]
```

Register-cached form:

```
mov ecx, dword ptr [rdi]
add eax, ecx
add ebx, ecx
```

Conceptual cache-relevant pattern: linear traversal

```
mov rbx, rdi
mov ecx, edx
L1:
mov eax, dword ptr [rbx]
add rbx, 4
dec ecx
jne L1
```

The reader is ready for cache discussions when they can:

- Count memory accesses by reading assembly.
- Identify loop bodies with repeated loads and stores.
- Separate address computation cost from memory access cost.

C.3 Mapping These Concepts to ABI-Specific Rules

Application Binary Interfaces (ABIs) impose rules on top of the x86 ISA. They do not change instruction semantics; they restrict how registers and memory are used across function boundaries.
Preparation for ABI study requires mastery of:

- General-purpose register equivalence at the ISA level.
- Flag volatility and the fact that flags are not preserved unless explicitly specified by convention.

- Implicit register usage in instructions such as `mul`, `div`, and string operations.

Example: architectural freedom

```
mov rax, rbx
add rcx, rdx
```

Architecturally valid regardless of ABI, but ABI rules may later require:

- certain registers to be preserved across calls
- certain registers to be used for parameter passing

Conceptual ABI boundary example:

```
# before call
mov rax, 1
mov rbx, 2

call func                 # ABI defines which registers func may
 ↪  overwrite

# after call
# reader must know which registers are safe to use
```

The reader is ready to map ISA knowledge to ABI rules when they can:

- Distinguish architectural behavior from convention-imposed behavior.
- Identify which instructions have implicit side effects that ABIs must account for.
- Understand that ABIs standardize usage patterns without altering x86 semantics.

This appendix marks the transition point from architectural fundamentals to convention-driven system-level programming. All subsequent booklets build on the models established here rather than reintroducing them.

References

R.1 Official Architecture Manuals (x86 / x86-64)

The primary and authoritative sources for x86 and x86-64 behavior are the vendor architecture manuals. These documents define the ISA precisely, including register semantics, flags behavior, addressing modes, instruction encodings, and all architectural corner cases.
Core reference categories:

- Programmer's reference for integer, control-flow, and system instructions
- Architectural description of registers, flags, and memory models
- Precise definitions of instruction side effects and undefined or reserved behavior

These manuals are the ground truth for:

- which instructions update which flags
- which registers are implicit operands
- how effective addresses are computed
- what is architecturally guaranteed versus microarchitectural

All explanations in this booklet are derived from these architectural definitions, not from compiler folklore or platform-specific conventions.

R.2 Compiler Documentation and Generated Code Behavior

Modern compilers provide extensive documentation describing how high-level constructs are lowered into machine code. While compilers do not define the ISA, they reveal common usage patterns and idioms that rely on ISA guarantees.
Relevant documentation domains include:

- instruction selection strategies
- register allocation models
- use of addressing modes for arrays and structures
- lowering of arithmetic and comparisons into flag-based logic

Example of compiler-relevant idioms:

```
test eax, eax
jne  L1
```

This pattern relies on architectural guarantees of `test` flag behavior. Understanding the ISA makes such generated code predictable and readable.
Compiler documentation is used here only to confirm patterns that are already architecturally valid.

R.3 Instruction Set Reference Sources

Instruction set reference material focuses on individual instructions and instruction families. These sources clarify:

- operand forms and legal combinations

- implicit operands and side effects
- flag updates and exceptions
- width-specific behavior across 8/16/32/64-bit forms

Example: understanding implicit operands in division:

```
mov rax, 100
xor edx, edx
mov rcx, 9
div rcx
```

Only an instruction-level reference explains why `RDX:RAX` forms the dividend and why failing to prepare `RDX` causes faults.
Instruction references are essential for:

- writing correct low-level code
- auditing compiler output
- avoiding undefined or faulting instruction sequences

R.4 Academic and Professional CPU Architecture Materials

Academic and professional architecture texts provide the conceptual framework that explains why the ISA is designed as it is. They cover:

- instruction execution models
- separation of ISA and microarchitecture
- memory hierarchies and latency hiding

- trade-offs between instruction complexity and code density

These materials support the mental models emphasized throughout this booklet:

- reading instructions as state transformations
- separating address computation from data access
- understanding flags as a state machine rather than boolean outputs

While these sources often use simplified or generic architectures for teaching, the principles apply directly to real x86 processors.

R.5 Cross-References to Other Booklets in This Series

This booklet is part of a structured CPU Programming Series. Its content depends on concepts introduced earlier and prepares the ground for later volumes.
Relevant backward references:

- execution model and instruction lifecycle
- binary representation and two's complement arithmetic
- conceptual understanding of registers and flags

Forward references enabled by this booklet:

- stack behavior and calling conventions
- ABI rules layered on top of the ISA
- memory hierarchy, caches, and performance analysis
- instruction scheduling and low-level optimization

The separation of concerns across booklets ensures that:

- architectural rules are learned before conventions
- correctness precedes optimization
- ISA knowledge remains valid across operating systems and toolchains

All references listed here are used to reinforce architectural correctness and long-term applicability rather than platform-specific behavior.