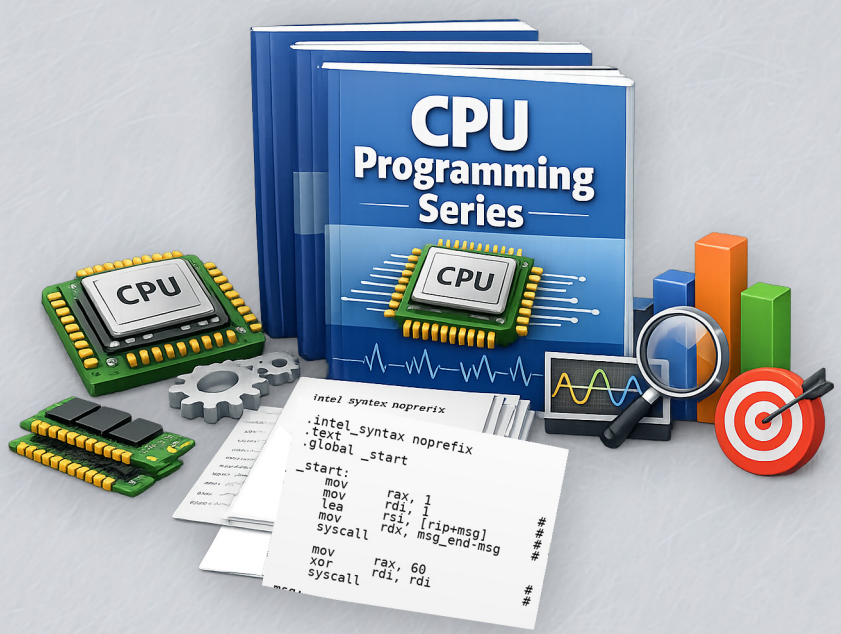
https://simplifycpp.org

CPU Programming Series

How a CPU Executes Instructions

From Instruction Fetch to Retirement



1

Prepared by Ayman Alheraki

CPU Programming Series

How a CPU Executes Instructions

From Instruction Fetch to Retirement

Prepared by Ayman Alheraki

simplifycpp.org

November 2025

Contents

Author's Introduction

CPU programming sits at the foundation of all software, whether you write operating systems, compilers, game engines, databases, or high-level applications. Every abstraction eventually becomes instructions executed by a real processor. Understanding how a CPU executes instructions is therefore not an optional specialization; it is the difference between guessing and knowing.

This series exists because most learning paths approach CPUs backwards: they start with instruction lists, syntax, or isolated optimizations. That approach is slow, fragile, and often misleading. What actually matters is understanding *execution*: how instructions flow through the machine, how state changes become visible, and why performance and correctness are governed by dependencies, memory, and control flow—not by instruction count.

Why CPU programming matters

CPU programming teaches you:

- how software truly runs, beyond language and compiler abstractions,
- why some code scales and other code collapses under real workloads,
- how bugs appear when hidden state (flags, stack, control flow) is misunderstood,
- and why performance problems cannot be solved by intuition alone.

Even when you never write hand-crafted assembly, CPU knowledge sharpens how you write C, C++, Rust, and other systems languages. It changes how you read compiler output, interpret profiles, and design algorithms that cooperate with hardware instead of fighting it.

What this series is designed to do

This series is built for people who are passionate about understanding computers *deeply* and want to learn efficiently:

- It focuses on **core execution principles** instead of long descriptions.
- It teaches **mental models** that transfer across x86, ARM, and RISC-V.
- It removes folklore and replaces it with **stable rules of execution**.
- It builds intuition through **short, precise explanations** and mental tracing.

Every booklet is intentionally compact. The goal is speed of understanding, not volume of text. If a concept does not improve your ability to predict execution, it is excluded.

Who this series is for

This series is for learners who:

- want to understand how CPUs actually work, not just how to use them,
- care about correctness, performance, and system behavior,
- prefer clear models over verbose explanations,
- and want knowledge that remains valid across architectures and decades.

No prior mastery of a specific ISA is required. What matters is curiosity, discipline, and willingness to think in terms of execution rather than syntax.

What you will gain quickly

By following this series, you will rapidly learn to:

- reason about instruction flow without memorizing manuals,
- identify real performance bottlenecks in minutes, not days,
- understand why compilers generate certain patterns,
- and approach advanced topics—ABI, calling conventions, optimization—with confidence.

This first booklet establishes the execution foundation. Everything that follows builds on it. Once you understand how a CPU executes instructions, the rest of low-level programming becomes structured, logical, and predictable.

Ayman Alheraki

Series Preface: CPU Programming Series Booklets

This series is a disciplined introduction to CPU programming: not as a collection of instructions, but as a practical model of *execution*. The objective is to make you fluent in what the hardware actually does with your code: how instruction bytes become work, how state changes become visible, and why performance is dominated by dependencies, control flow, and memory behavior rather than by memorizing mnemonics.
Modern processors are engineered to preserve a simple illusion: instructions execute in the order you wrote them. Internally, they do the opposite whenever allowed: they pipeline, predict, speculate, rename registers, reorder execution, and overlap memory operations. This gap between the *architectural contract* (ISA) and *microarchitectural reality* is where most misunderstandings begin. It is also where professional-level reasoning begins.

What this series is (and is not)

- This series is **execution-first**: it explains the lifecycle of instructions, the state they transform, and the mechanisms that sustain throughput.
- This series is **architecture-aware**: it treats ISA documentation as a contract, then teaches how real CPUs implement that contract.

- This series is **performance-literate**: it replaces folklore with stable cost drivers: critical paths, resource limits, cache/TLB behavior, and branch predictability.

What it is not:

- It is not a reference manual for every instruction form.
- It does not promise universal cycle counts (these depend on microarchitecture and workload).
- It does not teach platform ABIs in the first booklet; ABIs are introduced after the execution model is solid.

Who the series is for

- Systems and performance programmers who want to reason about low-level cost with accuracy.
- Compiler, runtime, and toolchain learners who want a clean mental model before diving into ISA details.
- Engineers writing C/C++/Rust/other low-level code who want assembly literacy as a correctness and optimization tool.

How to read these booklets

Each booklet is designed to be short, focused, and transferable across architectures:

- Start with the mental model; do not rush to optimization tricks.
- Treat registers, flags, PC, and stack as first-class state.

- Always separate: **correctness** (ISA contract) from **performance** (microarchitecture).
- Prefer invariants and mechanisms over implementation trivia.

The progression of the series

The series is structured so that each topic builds on the previous one:

1. **Booklet 1: How a CPU Executes Instructions — From Instruction Fetch to Retirement**
   Builds the baseline execution model: fetch, decode, execute, retire; architectural vs internal state; registers, flags, stack; and the cost of control flow and memory.
2. **Calling conventions and ABI fundamentals**
   Turns execution concepts into cross-function contracts: register preservation, argument passing, stack frames, alignment, and unwind-aware layouts.
3. **Architecture-specific reading (x86, ARM, RISC-V)**
   Applies the same mental model to different ISAs: how encoding, register sets, and condition mechanisms change the surface while the execution constraints remain.
4. **Performance and optimization track**
   Deepens the cost model: caches and TLBs, branch behavior, measurement methodology, loop optimization, vectorization concepts, and compiler interaction.

The discipline you will build

By the end of the series, you should be able to:

- read assembly as dataflow and control-flow, not as a list of instructions,

- trace architectural state precisely and avoid hidden-state bugs,
- identify bottlenecks by category (front-end, back-end, memory, branches),
- design experiments and measurements that explain *why* code is slow,
- and reason across ISAs using the same stable principles.

```
.intel_syntax noprefix

# Series mindset in one fragment:
# 1) Correctness: respect architectural state (regs/flags/PC/stack).
# 2) Performance: find the bottleneck (dependencies, memory, control flow).
# 3) Portability: the ISA is the contract; the microarchitecture is the
↪  cost.

# Dependency chain (latency-bound)
mov     rax, [rdi]
add     rax, 1
imul    rax, rax, 7

# Independent work (throughput opportunity)
mov     r8,  [rsi]
mov     r9,  [rdx]
add     r8,  r9

# Control flow hazard (mispredict risk if unpredictable)
test    eax, 1
jz      .Leven
add     ebx, 3
.Leven:
ret
```

Booklet Preface

This booklet is not an instruction encyclopedia. It is a map from source-level intent to the physical reality of how modern CPUs execute machine code: how bytes become micro-operations, how work overlaps, where latency hides, and why performance is usually decided by data movement and dependencies rather than by instruction mnemonics.
The goal is simple: after reading this booklet, you should be able to look at any assembly listing and ask the only questions that matter: *What limits throughput? What sets the critical path? What stalls the front-end or the back-end? What does the memory hierarchy do to this loop?*

Why understanding what actually happens inside the CPU matters more than memorizing instructions

Memorizing opcodes rarely improves real programs because most performance outcomes are dominated by:

- **Dependencies:** the CPU cannot execute an operation before its inputs are ready. Long dependency chains define the critical path.
- **Front-end limits:** fetching, decoding, and delivering work to the execution engine can become the bottleneck, especially with branches and code-size pressure.

- **Out-of-order execution details:** CPUs reorder work to hide latency, but only when there is independent work available and resources (queues, schedulers, execution ports) are not saturated.
- **Memory behavior:** cache misses, TLB pressure, and unpredictable access patterns routinely dominate cycle cost compared to arithmetic.

If you only learn "what an instruction does" you can still write assembly that is *correct* but slow, because correctness does not guarantee that the CPU can exploit parallelism or avoid stalls. Understanding the pipeline, speculation, dependency tracking, and the memory hierarchy turns assembly into a tool for reasoning about cost.

```
.intel_syntax noprefix

# Two snippets can be "correct", yet differ radically in how well the CPU
↪  can overlap work.

# Example idea: long dependency chain (critical path dominates)
mov     rax, [rdi]          # load
add     rax, 1              # depends on load
imul    rax, rax, 7         # depends on add
add     rax, [rsi]          # depends on previous + another load

# Even if each instruction is valid, the chain limits out-of-order overlap.
# The CPU can't "memorize" its way around dependencies.
```

The difference between correct assembly and effective assembly

Correct assembly:

- Produces the intended results.

- Respects the ABI, calling convention, and required architectural rules.
- Uses valid encodings and legal instruction forms.

Effective assembly:

- Also correct, but structured to fit the CPU's execution model: it exposes independent work, minimizes stalls, and reduces pressure on limited resources.
- Respects the difference between *latency* (time to complete one dependency step) and *throughput* (rate of completing independent operations).
- Treats memory as a first-class cost: layout, alignment, access order, prefetch behavior, and branch predictability often matter more than choosing "faster" instructions.

A common beginner mistake is optimizing instruction selection while ignoring dataflow. Another is chasing micro-optimizations that make code larger or less predictable, hurting the front-end and branch predictor. Effective assembly is not about "clever instructions"; it is about shaping the program so the CPU can keep useful work in flight.

```
.intel_syntax noprefix

# Correctness is not enough: structure matters.

# (A) Correct but fragile: unpredictable branch inside hot loop
#     Branch mispredicts can dominate even when each instruction is cheap.
cmp     eax, 0
je      .Lslow_path        # if unpredictable, speculation fails often
# fast path continues...

# (B) Often more effective: reduce unpredictability / allow steady-state
↪  flow
#     (Exact transformation depends on algorithm and constraints.)
#     The principle: prefer predictable control flow or data flow that
↪  keeps the pipeline fed.
```

How this booklet changes the way you read any ISA documentation

ISA manuals are necessary, but they primarily describe *architectural semantics*: what must happen, not how fast it happens. After this booklet, you will read ISA documentation with a different checklist:

- **Semantics vs. microarchitecture:** the ISA tells you the visible behavior; performance depends on decoding, micro-op expansion, port usage, scheduling, and memory subsystem effects that may not be specified.
- **Hidden costs:** does an instruction imply extra micro-ops, serialization, partial-register penalties, flags dependencies, or special-case handling?
- **Data dependencies:** which operands create true dependencies, and which create false ones (e.g., implicit flags, register merging, or aliasing through memory)?
- **Control flow impact:** how does the instruction sequence interact with prediction, speculation, and recovery?
- **Memory ordering and visibility:** are you relying on ordering guarantees, fences, atomics, or alignment assumptions that restrict reordering and reduce throughput?

In other words, you will stop reading ISA pages as "a list of instructions" and start reading them as *constraints* on an execution engine. This is the mindset that scales across x86-64, ARM64, RISC-V, and any future ISA: you learn the contract, then reason about how real CPUs implement it.

```
.intel_syntax noprefix

# Reading an ISA page after this booklet:
```

```
# - You still learn what the instruction does (architectural contract),
# - but you immediately ask what it implies for execution and memory
↪  behavior.

# Example checklist in code form (conceptual, not a single "rule"):
# 1) Does this create a long dependency chain?
# 2) Does this introduce unpredictable branching?
# 3) Does this increase code size / front-end pressure?
# 4) Does this increase memory traffic or reduce locality?
# 5) Does this require ordering / fencing that limits reordering?
```

Chapter 1

The CPU as an Execution Machine

The Mental Model You Must Build First

A modern CPU is best understood as a *throughput-oriented execution system* constrained by dependencies, limited internal resources, and the memory hierarchy. It is not a single "engine" that finishes one instruction before starting the next; it is a coordinated pipeline of stages that tries to keep many operations in flight, completing them *as if* they executed in program order (architectural order), while internally reordering and speculating whenever correctness rules allow.

This chapter builds the minimum mental model you must carry into every later topic:

- The ISA defines an *architectural contract* (what must be observable).
- The microarchitecture implements that contract using pipelines, queues, speculation, and out-of-order execution.
- Performance is decided by *dataflow* and *memory behavior* more often than by instruction selection.

1.1 What a CPU Really Does

The CPU is not a simple instruction executor

A CPU is an execution machine whose job is to continuously transform a stream of instruction bytes and a stream of data values into an updated architectural state, while maintaining the illusion that each instruction completed in the order specified by the program.
In practice, the CPU is a *resource-scheduled* system:

- It fetches instruction bytes from memory (usually from instruction cache), predicts control flow, and forms a speculative path.
- It decodes instructions and typically translates them into internal operations suitable for scheduling (often micro-operations).
- It tracks dependencies between operations and delays operations whose inputs are not ready.
- It executes ready operations as soon as execution resources are available.
- It retires results in architectural order, making them architecturally visible, and discards speculative work if a prediction was wrong.

The key shift: the CPU is not "doing instructions" one-by-one; it is managing a *graph of dependent operations* under constraints:

- limited decode bandwidth,
- limited execution ports / pipelines,
- limited window size (how many operations can be tracked in flight),
- and limited memory-level parallelism.

Execution as a coordinated process, not isolated instructions

Isolated instruction semantics rarely explain performance. What matters is how a *sequence* behaves when mapped onto:

- the front-end (fetch, predict, decode, deliver),
- the out-of-order core (rename, schedule, execute),
- the memory subsystem (L1/L2/L3 caches, TLBs, prefetchers, coherence),
- and the retirement/commit machinery (precise state, exception boundaries).

This is why instruction tables alone mislead: a “fast” instruction can become slow if it blocks the front-end, extends a dependency chain, or increases cache misses. A “slow” instruction can be harmless if it is off the critical path and overlaps with other work.

```
.intel_syntax noprefix

# The CPU treats this as a dependency chain.
# Even if each instruction is "cheap", the chain limits overlap.

mov     rax, [rdi]        # load -> latency depends on cache/memory
add     rax, 1            # depends on load completing
imul    rax, rax, 7       # depends on add
add     rax, [rsi]        # depends on prior + another load

# Effective thinking starts by identifying:
# - The critical path (true dependencies)
# - Where memory latency can dominate
# - Whether independent work exists to overlap
```

1.2 Instructions, Data, and State

Where instructions live

Instructions are bytes stored in memory like any other data:

- They reside in the program's virtual address space.
- They are fetched through the same memory hierarchy (caches, TLBs) as data, but typically via dedicated instruction-side structures (I-cache and instruction TLB).
- They must be mapped, accessible, and located on a predictable control-flow path for the front-end to deliver them efficiently.

The practical implication: code layout and control flow influence how well the front-end can keep the back-end busy. Code size, alignment, and branch density often matter because they affect fetch/decode efficiency and prediction accuracy.

Where data lives

Data is also stored in memory, accessed through the memory hierarchy:

- Registers hold a small, fast working set.
- Caches and TLBs accelerate access to recently used addresses and translations.
- Main memory provides capacity but much higher latency.

From the CPU's point of view, memory is not a uniform array. The same load instruction can take drastically different time depending on where the data is found:

- L1 hit: low latency.

- L2/L3 hit: higher latency.
- DRAM: very high latency, often dominating execution unless overlapped.

Data access patterns determine whether the CPU can:

- exploit spatial/temporal locality,
- generate memory-level parallelism,
- and keep pipelines busy without stalls.

What “processor state” really means

In architectural terms, the processor state is what the ISA defines as observable:

- general-purpose registers,
- vector/SIMD registers,
- flags/status registers,
- instruction pointer / program counter,
- control registers and privilege state (where applicable),
- and the architecturally visible memory contents.

Modern CPUs also maintain extensive *microarchitectural state* that is not part of the ISA contract but strongly affects performance:

- branch predictor state,
- cache contents and replacement state,
- TLB contents,

- queues, reorder buffers, schedulers,
- and speculation bookkeeping.

A critical rule: architectural state must be precise at instruction boundaries for exceptions and interrupts. Many design choices (like retirement and reorder buffers) exist primarily to preserve this property while allowing out-of-order execution.

```
.intel_syntax noprefix

# Architectural state: what the ISA promises is updated "as if" in order.
# Microarchitectural state: predictors/caches/queues that the ISA does not
↪  expose.

# Example: flags create extra dependencies that are easy to overlook.
add     eax, 1            # updates EAX and flags
adc     ebx, 0            # consumes carry flag -> dependency on flags

# The code is correct, but the dependency can constrain scheduling.
```

1.3 The Illusion of Sequential Execution

Why programs appear linear

At the programming model level, execution appears linear because the ISA defines:

- a single architectural sequence of instructions,
- each with well-defined effects on architectural state,
- and precise rules for exceptions/interrupts that behave as if instructions completed in order.

This illusion is essential: it allows compilers and developers to reason about correctness without needing to model the CPU's internal speculation and reordering.

Why execution is not

Internally, modern CPUs violate the naive linear model to gain performance:

- **Pipelining:** different stages work on different instructions simultaneously.
- **Speculation:** the CPU guesses control flow and executes along the predicted path before certainty.
- **Out-of-order execution:** ready operations execute when resources permit, not strictly by program order.
- **Memory-level parallelism:** multiple cache misses can be in flight at once, subject to limits.

Yet, the CPU must *retire* (commit) results in order so that architectural state changes are coherent and precise. This separation—*execute out of order, retire in order*—is the core mechanism behind the "sequential illusion".

```
.intel_syntax noprefix

# Program order (what you write) vs. execution order (what may happen
↪  internally)

# In program order:
mov     eax, [rdi]          # A: load
mov     ebx, [rsi]          # B: load
add     ecx, eax            # C: depends on A
add     edx, ebx            # D: depends on B
```

```
# Internally, the CPU can overlap A and B, and may execute D before C
# if B completes earlier than A. Retirement still preserves architectural
↪  order.
```

What you should carry forward

From this point onward, treat every performance question as an execution-system question:

- Identify **dependencies** (critical path) versus independent work (available parallelism).
- Separate **front-end** limits (fetch/decode/predict) from **back-end** limits (ports/schedulers) and from **memory** limits (caches/TLB/DRAM).
- Remember the governing principle: the CPU may do a lot of work out of order, but it must always be able to present results *as if* it executed in order.

This mental model is the foundation for the rest of the booklet: instruction fetch, decode, rename, scheduling, execution, speculation recovery, and retirement.

Chapter 2

The Instruction Lifecycle

From Fetch to Retirement

An instruction does not “run” in one step. Modern CPUs implement an instruction lifecycle that turns a byte stream into executed work while preserving the architectural illusion of in-order completion. Conceptually, the lifecycle can be summarized as:

$$\textit{Fetch} \rightarrow \textit{Decode} \rightarrow \textit{(Schedule)} \rightarrow \textit{Execute} \rightarrow \textit{Retire}$$

Real designs include additional internal steps (prediction, renaming, queueing, replay, speculation recovery), but the model above is the stable foundation for understanding any high-performance core.

2.1 Instruction Fetch

Program Counter concept

The **Program Counter** (PC) is the architectural concept that identifies *the next instruction address* in the current control-flow path:

- On x86-64 it is commonly referred to as the instruction pointer (RIP).
- On many ISAs it is explicitly called PC.

Architecturally, the PC advances to the next instruction in sequence, or changes to a branch/jump/call target. Microarchitecturally, fetch is rarely driven by the "true" next PC alone; it is heavily assisted by **branch prediction** and **target prediction** so that instruction bytes can be supplied continuously.

Basic idea of instruction storage

Instructions are stored as bytes in memory within the program's virtual address space. Fetch retrieves those bytes through the memory hierarchy:

- The **instruction cache** (I-cache) supplies recently used instruction bytes quickly.
- The **instruction TLB** supplies cached address translations so virtual addresses can be mapped efficiently.

If instruction bytes are not readily available (I-cache miss, iTLB miss, or heavy code-size pressure), the front-end can starve the rest of the pipeline. For many workloads, front-end behavior (prediction accuracy and fetch bandwidth) is a primary performance limiter.

```
.intel_syntax noprefix

# Conceptual view: control flow changes the PC.
# (The CPU will speculate/predict these changes in real hardware.)
```

```
cmp     eax, 0
je      .Lzero              # if taken, PC becomes .Lzero
add     ebx, 1              # fall-through path (PC continues sequentially)
jmp     .Ldone

.Lzero:
sub     ebx, 1

.Ldone:
ret
```

2.2 Decode

What decoding actually means

Decode is the process of transforming fetched instruction bytes into internal control signals and operations suitable for execution:

- Identify instruction boundaries (especially important for variable-length encodings).
- Determine operation type, operand sources/destinations, and addressing modes.
- Produce one or more internal operations (often micro-operations) that can be scheduled.

Decode is not "understanding" in a human sense; it is translation from a compact binary encoding into a form the core can route to execution resources.

Why instruction format matters

Instruction format directly affects front-end efficiency:

- **Fixed-length** instruction sets simplify boundary detection and can ease fetch/decode scaling.

- **Variable-length** instruction sets can achieve high code density but increase boundary and decode complexity.
- Complex addressing modes and certain instruction classes can expand into multiple internal operations, consuming decode bandwidth and internal queues.

The practical outcome is that "same number of source instructions" does not imply "same decode cost". Code size, alignment, and instruction mix can dominate.

```
.intel_syntax noprefix

# Same semantics can have different front-end implications:
# - Instruction length / encoding density
# - Addressing mode complexity
# - Potential internal expansion

mov     eax, [rdi]              # simple load
add     eax, [rsi+rcx*4+16]      # indexed addressing: still one
↪  instruction, potentially more work internally
```

2.3 Execute

Execution units: ALU, load/store, branch (conceptual view)

After decoding (and typically after internal dependency tracking and scheduling), operations execute on specialized units:

- **ALU / integer units:** arithmetic, logic, shifts, integer multiplies (often different pipelines).
- **Vector/SIMD units:** packed integer/floating-point operations (width and pipelines vary by core).

- **Load/Store units:** compute effective addresses, perform cache accesses, handle store buffering.
- **Branch units:** evaluate branch conditions, resolve targets, and confirm predictions.

Execution is constrained by:

- **Data dependencies:** an operation cannot execute until its inputs are ready.
- **Resource limits:** only so many operations can issue per cycle to each unit/port.
- **Memory latency:** load completion time depends on cache/TLB state; misses introduce long delays.

```
.intel_syntax noprefix

# Conceptual mapping:
# - ALU:      add/xor/and/shl
# - Load:     mov reg, [mem]
# - Store:    mov [mem], reg
# - Branch:   jcc/jmp/call/ret

add     rax, rbx          # ALU work
mov     rcx, [rdi]        # load (can stall on cache miss)
mov     [rsi], rdx        # store (often buffered, becomes visible later)
test    eax, eax
jne     .Lloop            # branch resolution affects speculation

.Lloop:
nop
```

2.4 Retirement

Why instructions must retire

Modern CPUs may execute work out of order and speculatively. **Retirement** (also called commit) is the step that makes results *architecturally visible in program order*. Retirement exists to guarantee:

- **Precise state** at instruction boundaries for exceptions and interrupts.
- Correct handling of **mis-speculation**: if a branch prediction was wrong, the CPU can discard speculative results that have not retired.
- A clear boundary between what is *guaranteed* by the ISA and what is only temporary internal progress.

Architectural state vs internal state

Architectural state is what the ISA defines as the machine state visible to software:

- registers, flags, PC, and architecturally visible memory results.

Internal state is the microarchitectural machinery used to implement performance features:

- predictors, caches/TLB contents, rename maps, queues, reorder buffers, speculative results.

Execution may update internal structures early, but architectural state changes become official only at retirement. This separation is the foundation of "execute out of order, retire in order."

```
.intel_syntax noprefix

# Example: speculative path vs retired path (conceptual)
# If the branch is mispredicted, speculative work is discarded before
↪  retirement.
```

```
cmp     eax, 0
je      .Lbad_guess         # predictor may guess taken/not-taken
# speculative instructions may execute here...
add     ebx, 1

.Lbad_guess:
sub     ebx, 1
```

2.5 Why Pipelines Exist

Throughput vs latency (conceptual understanding)

A pipeline is an engineering strategy to increase **throughput** by overlapping stages:

- **Latency** is how long a single instruction (or operation) takes from start to completion.
- **Throughput** is how many instructions/operations can be completed per unit time in steady state.

Pipelining often does not reduce the latency of a single dependent chain; it increases how much *independent* work can be processed concurrently. The practical rule:

- If your code is dominated by a long dependency chain, latency rules.
- If your code has many independent operations, throughput rules.

Pipelines exist because real workloads contain mixed dependencies and because hardware can overlap fetch, decode, execute, and retire to keep expensive units utilized. Modern designs extend this idea with out-of-order execution: the CPU actively searches for ready work to maximize throughput while retirement preserves architectural correctness.

```
.intel_syntax noprefix

# Latency-bound: dependency chain (limited overlap)
mov     rax, [rdi]
add     rax, 1
add     rax, 1
add     rax, 1

# Throughput opportunity: independent ops (can overlap on wide cores)
mov     r8,  [rdi]
mov     r9,  [rsi]
mov     r10, [rdx]
mov     r11, [rcx]
# Independent loads may overlap (subject to cache/TLB and MLP limits).
```

Chapter 3

ISA vs Microarchitecture

Why the Same Instruction Is Not Always Equal

Two processors can execute the same ISA and still behave very differently in performance, power, and even timing of micro-events. The ISA defines the *architectural contract* (what must be correct and observable). Microarchitecture defines *how* that contract is implemented: pipelines, caches, predictors, execution width, scheduling policies, and the internal decomposition of instructions.

This chapter builds a disciplined rule for assembly thinking:

> *ISA tells you what is legal and correct. Microarchitecture decides how expensive it is.*

3.1 What an ISA Is

The contract between software and hardware

An **Instruction Set Architecture (ISA)** is the specification that software targets and hardware implements. It defines:

- **Instruction semantics:** what each instruction does to architectural state.
- **Architectural state:** registers, flags/status, program counter, and the visible memory effects.
- **Addressing and memory model rules:** how memory is addressed, alignment requirements, atomicity rules, ordering constraints, and exception behavior.
- **Privilege and system features:** where applicable (modes, traps, control registers).

An ISA is intentionally an abstraction layer: it enables compilers, OS kernels, and applications to run across different implementations as long as those implementations preserve the defined observable behavior.

```
.intel_syntax noprefix

# ISA-level view: architectural contract.
# This instruction must add 1 to EAX and update architectural flags
↪  accordingly.

add     eax, 1

# The ISA does not promise how many internal steps are used,
# which execution unit is used, or how many cycles it takes.
```

3.2 What Microarchitecture Is

Implementation details hidden from the programmer

Microarchitecture is the internal design used to implement the ISA. It includes the mechanisms that translate and execute instructions:

- **Front-end:** fetch, branch prediction, decode, instruction queues.
- **Execution core:** register renaming, scheduling, execution ports/units, out-of-order window.
- **Memory subsystem:** caches (L1/L2/L3), TLBs, prefetchers, store buffers, coherence machinery.
- **Speculation and recovery:** speculative execution, misprediction recovery, replay mechanisms.
- **Retirement:** commit logic that makes state precise and in-order architecturally.

These are not optional details; they *are* the CPU's performance model. Two CPUs can implement the same ISA but differ in width, depth, buffers, predictor quality, cache sizes, and execution resources—leading to different outcomes for the same instruction stream.

```
.intel_syntax noprefix

# Same ISA instruction stream, but microarchitecture determines:
# - decode bandwidth (how fast instructions become internal ops)
# - execution width (how many ops can run per cycle)
# - cache/TLB behavior (how fast memory operands arrive)
# - prediction quality (how often speculation is correct)

mov     eax, [rdi]
add     eax, [rsi]
ret
```

3.3 Why Performance Depends on Microarchitecture

Same ISA, different CPUs, different behavior

Performance depends on microarchitecture because the CPU must map the ISA stream onto limited internal resources under dependency constraints. Differences that commonly change performance include:

- **Instruction decode and delivery:** different decoders, different queue sizes, different ability to sustain throughput.
- **Instruction decomposition:** one ISA instruction may become 1 internal operation on one core but multiple operations on another, consuming bandwidth.
- **Execution resources:** number and type of ALUs, vector units, branch units, and load/store pipelines.
- **Scheduling and window size:** how much independent work can be tracked and reordered to hide latency.
- **Memory hierarchy:** cache latency/size/associativity, TLB capacity, prefetch behavior, and memory-level parallelism limits.
- **Branch prediction:** mispredict rate can dominate tight loops with hard-to-predict control flow.

Even when two CPUs have identical instruction semantics, their *bottlenecks* may differ:

- One CPU may be **front-end limited** (cannot fetch/decode enough).
- Another may be **back-end limited** (execution ports saturated).
- Another may be **memory limited** (loads often miss caches).

```
.intel_syntax noprefix

# Same loop, different CPUs:
# - On one CPU: branch predictor handles it well (high throughput)
# - On another: mispredicts more often (pipeline flushes dominate)

.Lloop:
cmp     eax, edx
jl      .Lloop
```

3.4 Why Assembly Programmers Must Care

Why "correct code" can still be slow

Assembly programmers target the ISA for correctness, but real performance is set by microarchitectural realities:

- **Dependency chains:** correct sequences can serialize execution if each instruction depends on the previous.
- **Hidden dependencies:** flags and implicit operands can create constraints that reduce parallelism.
- **Front-end pressure:** correct code can be too large, too branchy, or too complex to decode efficiently.
- **Memory behavior:** correct address computations and loads can still trigger cache/TLB misses that dominate.
- **Speculation cost:** unpredictable branches cause wasted work and recovery penalties.

The disciplined assembly mindset is therefore:

- Write code that is correct with respect to the ISA contract.
- Then evaluate whether the microarchitecture can execute it efficiently:
  - Are there avoidable dependency chains?
  - Is the hot path predictable?
  - Is the code front-end friendly?
  - Does data access have locality and alignment?
  - Are loads/stores arranged to maximize overlap?

```
.intel_syntax noprefix

# Correct but often slower: dependency chain through a single accumulator
xor     eax, eax
add     eax, [rdi]
add     eax, [rdi+4]
add     eax, [rdi+8]
add     eax, [rdi+12]

# Conceptual alternative: create independent work (multiple accumulators)
# The ISA is identical, but the microarchitecture can overlap more
↪  execution.
xor     eax, eax
xor     ecx, ecx
add     eax, [rdi]
add     ecx, [rdi+4]
add     eax, [rdi+8]
add     ecx, [rdi+12]
add     eax, ecx

# Both are correct. The difference is how well the CPU can exploit
↪  parallelism.
```

What this chapter changes in your reading of code

After this chapter, you should stop asking “Is this instruction fast?” and start asking:

- **What is the bottleneck on this CPU?** (front-end, back-end, memory, branch prediction)
- **What prevents overlap?** (true dependencies, hidden dependencies, resource contention)
- **What is the memory story?** (locality, alignment, miss rate, MLP)
- **What is the control-flow story?** (predictability, speculation recovery cost)

This is the only reliable way to write assembly that remains effective across real machines.

Chapter 4

Registers: The Real Working Area

The Fastest Storage in the System

Registers are the CPU's closest and fastest storage. They sit at the boundary between instruction semantics and execution reality: every arithmetic operation consumes register operands, every address calculation uses registers, and effective execution depends on how well values stay in registers instead of repeatedly returning to memory.
This chapter builds a practical mental model:

- Registers are not just "variables"; they are *ports into the execution engine*.
- Architectural registers are the ISA-visible names; physical registers are the real storage used by the core.
- Modern performance depends on *keeping working sets in registers and caches* and *avoiding memory stalls*.

4.1 What Registers Are

Register file concept

A **register** is a small, fast storage location inside the CPU. Collectively, registers are implemented as one or more **register files**: dense arrays of storage with many read/write ports to feed multiple execution units each cycle.
Key properties:

- **Low latency access:** operands can be read and written within core timing budgets.
- **High bandwidth:** multiple reads and writes per cycle must be supported to sustain wide issue execution.
- **Central role:** most executed operations read registers, produce registers, or compute addresses using registers.

From the ISA point of view, registers are named resources (e.g., RAX, RBX, XMM/YMM/ZMM). From the microarchitecture point of view, the register file must support sustained operand delivery without becoming a bottleneck.

```
.intel_syntax noprefix

# Registers are the immediate working area:
# - ALU ops typically read two inputs and write one output register.

mov     eax, 10
mov     ebx, 20
add     eax, ebx          # reads EAX/EBX, writes EAX

# Memory is accessed only through addresses computed from registers.
mov     rcx, [rdi]        # load uses address in RDI
```

4.2 Architectural vs Physical Registers

Register renaming (conceptual overview)

Modern out-of-order CPUs separate **architectural registers** (the ISA-visible names) from **physical registers** (the actual storage locations used internally).
Why this exists:

- The ISA exposes a limited set of named registers.
- The CPU wants many more in-flight values than the ISA naming would allow.
- The CPU also wants to eliminate *false dependencies* created by register reuse.

Register renaming is the mechanism that maps each architectural register write to a new physical register:

- A true dependency (read-after-write) remains real: consumers must wait for the producer value.
- False dependencies are removed:
  - **WAW** (write-after-write): two writes to the same architectural register can be separated.
  - **WAR** (write-after-read): a later write does not block an earlier read if they map to different physical registers.

Conceptually:

- Architectural name: `RAX`
- Physical instances over time: `P17`, then `P42`, then `P08`, ...

This is why a CPU can execute out of order yet still retire in order: speculative physical mappings exist internally, while architectural state is updated precisely at retirement.

```
.intel_syntax noprefix

# Architectural view: both writes target EAX, so it "looks" like a
↪  dependency.
mov     eax, 1            # write EAX
mov     eax, 2            # write EAX again

# Microarchitectural reality with renaming:
# - first write allocates a physical register for EAX (say P10)
# - second write allocates a new physical register for EAX (say P11)
# There is no need for the second to wait for the first unless something
↪  reads the first result.
```

4.3 Why CPUs Have Few Registers

Speed, cost, and physical constraints

Architectural registers are intentionally few because:

- **ISA and ABI stability:** changing register counts impacts calling conventions, binaries, compilers, and OS tooling.
- **Encoding cost:** more architectural registers typically require more bits in instruction encoding, increasing code size and front-end pressure.
- **Hardware complexity:** large multi-ported register files are expensive in area and power; scaling read/write ports is particularly costly.
- **Timing constraints:** delivering operands to multiple units every cycle must fit within tight clock periods.

Modern CPUs compensate with:

- **Renaming and many physical registers** to keep many values in flight.
- **Caching and prediction** to reduce the frequency and cost of memory access.

The key insight: architectural register count is not the same thing as the core's internal ability to track working sets. Performance comes from internal resources (physical registers, reorder window, queues) and memory locality.

4.4 Registers vs Memory

Why memory access dominates execution cost

ALU operations are usually fast relative to memory. Memory access dominates because it involves:

- **Address translation** (TLB lookup, possible page table walk on miss),
- **Cache hierarchy traversal** (L1 $\rightarrow$ L2 $\rightarrow$ L3 $\rightarrow$ DRAM),
- **Coherence and ordering constraints** (especially with shared data or atomics),
- and **variable latency** that depends on locality and contention.

A load that hits in L1 can be serviced quickly; a load that misses into DRAM can take orders of magnitude longer. This is why:

- keeping hot values in registers reduces pressure on caches,
- improving locality reduces cache/TLB misses,
- and restructuring loops to increase independent work helps hide unavoidable latency.

```
.intel_syntax noprefix

# Register work: predictable and fast
add     rax, rbx
xor     rcx, rcx
imul    rdx, rax, 7

# Memory work: latency depends on cache/TLB state
mov     r8,  [rdi]         # could be fast (L1) or very slow (DRAM)
mov     r9,  [rdi+64]       # may touch a different cache line
mov     r10, [rsi]          # another potentially independent miss

# If these loads miss, the core may stall unless it can overlap them (MLP)
# and unless subsequent work is independent enough to execute while
↪  waiting.
```

Practical consequences for assembly

- Treat registers as the **working set** and memory as the **backing store**.
- Favor loop structures that keep frequently used values in registers and minimize reloads.
- Reduce false dependencies so the core can exploit renaming and out-of-order scheduling.
- Assume memory latency is the enemy unless you have strong locality and predictable access.

This is the foundation for later chapters on caches, load/store behavior, and performance reasoning.

Chapter 5

Flags and Status

Invisible State That Controls Everything

Status flags are a compact, implicit communication channel between instructions. They are part of the architectural contract, yet they often behave like *invisible state*: modified as a side effect, consumed implicitly, and capable of creating both performance limits (hidden dependencies) and correctness traps (subtle bugs).
This chapter builds the operational mindset:

- Flags encode outcomes of arithmetic and comparisons.
- Many instructions write flags even when you do not explicitly "use" them.
- Branches and conditional operations typically consume flags.
- Flags can silently couple unrelated code, harming out-of-order overlap and breaking logic during refactoring.

5.1 What Flags Represent

Zero, Carry, Sign, Overflow (conceptual meaning)

Flags summarize properties of a computed result or an arithmetic event. Conceptually:

- **Zero Flag (ZF):** set when the result is zero (used for equality checks).
- **Carry Flag (CF):** indicates an unsigned carry/borrow out of the most significant bit.
  - After addition: CF = unsigned carry out.
  - After subtraction: CF = unsigned borrow (often interpreted as "unsigned less-than").
- **Sign Flag (SF):** reflects the sign bit of the result (most significant bit), used for signed comparisons.
- **Overflow Flag (OF):** indicates signed overflow (result does not fit in signed range).

These flags support both unsigned and signed reasoning:

- Unsigned comparisons commonly interpret CF/ZF.
- Signed comparisons commonly interpret SF/OF (and ZF for equality).

```
.intel_syntax noprefix

# Conceptual examples (x86-like thinking):
# - ZF: result == 0
# - CF: unsigned carry/borrow
# - SF: sign bit of result
# - OF: signed overflow
```

```
mov     eax, 0x7fffffff
add     eax, 1            # signed overflow -> OF becomes 1 (conceptually)
# EAX becomes 0x80000000, SF becomes 1, ZF becomes 0

mov     eax, 0xffffffff
add     eax, 1            # unsigned carry -> CF becomes 1, EAX becomes 0,
↪  ZF becomes 1
```

5.2 Flags as Side Effects

Why many instructions modify flags

Flags exist to make common patterns cheap:

- Arithmetic updates flags so a branch can immediately test the outcome without a separate compare.
- Compare and test instructions update flags without producing a register result, enabling conditional branches and moves.

This design improves code density and expresses "compute + condition" with minimal extra instructions. However, it introduces a cost: flags are an implicit destination of many instructions, even if your program does not intend to use them.
Practically:

- You must treat flags like a *register you did not name*.
- If you insert an instruction between a flag-producing operation and a flag-consuming decision, that inserted instruction may clobber the flags.

```
.intel_syntax noprefix
```

```
# Correct pattern: producer -> consumer
cmp     eax, ebx
jl      .Lless          # consumes flags set by cmp

# Dangerous refactor: inserting a flags-writing instruction breaks the
↪  meaning
cmp     eax, ebx
add     ecx, 1          # modifies flags (side effect)
jl      .Lless          # now reads flags from add, not from cmp
```

5.3 Flags and Control Flow

How decisions are actually made

At the ISA level, control-flow decisions are typically implemented as:

- A **flag-setting** operation (often `cmp`, `test`, or arithmetic),
- followed by a **flag-consuming** conditional branch (`jcc`) or conditional instruction (`cmovcc`, `setcc`).

Conceptually:

- `cmp a,b` performs an internal subtraction ($a - b$) to set flags, without keeping the subtraction result.
- The branch uses a particular interpretation of flags depending on signed vs unsigned intent.

Signed vs unsigned is not "in the data"; it is in the *instruction choice*:

- Unsigned comparisons typically map to conditions using CF/ZF (e.g., below/above).

- Signed comparisons typically map to conditions using SF/OF (e.g., less/greater).

```
.intel_syntax noprefix

# Unsigned decision (conceptual): uses CF/ZF interpretation
cmp     eax, ebx
jb      .Lbelow           # jump if eax < ebx (unsigned)

# Signed decision (conceptual): uses SF/OF interpretation
cmp     eax, ebx
jl      .Lless            # jump if eax < ebx (signed)

# Equality is shared (ZF)
cmp     eax, ebx
je      .Lequal
```

5.4 Why Flags Are Dangerous

Hidden dependencies and subtle bugs

Flags are dangerous for two main reasons.

Correctness hazards

- **Implicit clobbering:** many instructions update flags; inserting one instruction can silently change a later branch decision.
- **Signed/unsigned confusion:** using the wrong conditional (signed vs unsigned) produces correct-looking code that fails on edge values.
- **Over-reliance on "current flags":** code that assumes flags "still mean" what they meant earlier is fragile under maintenance.

```
.intel_syntax noprefix

# Subtle bug example: wrong signed/unsigned condition
# Suppose EAX and EBX should be treated as unsigned.
cmp     eax, ebx
jl      .Lwrong           # signed less-than, incorrect for values with
↪  high bit set

# Correct would be unsigned "below":
cmp     eax, ebx
jb      .Lright
```

Performance hazards (hidden dependencies)

On modern out-of-order cores, flags can become a throughput limiter because they create implicit dependencies:

- A flag-consuming instruction depends on the most recent flag-producing instruction.
- Long sequences that repeatedly produce and consume flags can form a dependency chain, limiting overlap.
- Some patterns overuse a single implicit "flags channel" even when the algorithm could expose more independent work.

A disciplined assembly habit is to treat flags as a scarce dependency resource:

- Keep flag producer and consumer close together.
- Avoid inserting unnecessary flag-writing instructions between them.
- Where appropriate, use patterns that reduce reliance on a single flags chain (for example, independent computations whose results are combined later).

```
.intel_syntax noprefix

# Dependency chain through flags:
# Each iteration produces flags and consumes them immediately.
# This is correct, but can constrain scheduling if it dominates the
↪  critical path.

.Lloop:
add     eax, 1              # produces flags
jne     .Lloop              # consumes flags (ZF), loop-carried dependency
ret

# Key idea: flags are not free; they serialize decisions.
```

Operational rules

- Treat flags like an implicit register: if you did not intentionally set them, do not rely on them.
- Keep compare/test immediately adjacent to the conditional that consumes it.
- Always be explicit about signed vs unsigned intent in conditional choices.
- When optimizing, look for flag-driven dependency chains that limit overlap.

Chapter 6

Control Flow Fundamentals

Branches, Calls, and the Cost of Decisions

Control flow is where the CPU must choose the next instruction address. Straight-line code gives the front-end a predictable stream. Branches, calls, and returns introduce uncertainty, so modern CPUs rely on prediction and speculation to preserve throughput. When predictions fail, performance drops because the machine must discard speculative work and refill the pipeline on the correct path.

6.1 Sequential Execution vs Control Flow

In **sequential execution**, the next instruction address is the fall-through:

- the PC advances to the next instruction in memory,
- fetch and decode can operate steadily,
- the front-end can keep the execution core supplied.

Control flow changes the next PC:

- conditional branches choose between fall-through and a target,
- unconditional jumps set the next PC to a target,
- calls and returns transfer control across code regions and must preserve a return path.

Conceptually, control flow turns the program into a graph of paths. The CPU must select one path early enough to avoid front-end bubbles.

```
.intel_syntax noprefix

# Sequential: next PC is fall-through
add     eax, 1
add     ebx, 2
add     ecx, 3

# Control flow: next PC may change
cmp     eax, edx
jl      .Ltarget
add     ebx, 1            # fall-through path
jmp     .Ldone

.Ltarget:
sub     ebx, 1

.Ldone:
ret
```

6.2 Conditional Branches

Decision-making at the CPU level

A conditional branch is typically a producer-consumer pair:

- a **flag-producing** instruction (often `cmp` or `test`),
- followed by a **flag-consuming** conditional branch (`jcc`).

At runtime, the CPU must determine:

- the branch **direction** (taken vs not-taken),
- and the **target** address (where to fetch next if taken).

Because the front-end must keep feeding the back-end, modern CPUs predict branches and speculatively fetch/execute along the predicted path before the branch is fully resolved.

```
.intel_syntax noprefix

# Producer -> consumer: flags drive the decision
cmp     eax, ebx
jge     .Lge              # uses flags (signed greater-or-equal)
# fall-through
add     ecx, 1
jmp     .Ldone

.Lge:
sub     ecx, 1

.Ldone:
ret
```

6.3 Function Calls as Control Flow

Call and return concepts (no ABI yet)

A **call** is a control-flow transfer that also records a return point:

- control transfers to the callee target,
- a return address (the next instruction after `call`) is saved so execution can resume.

A **return** transfers control back to the saved return address. From the CPU perspective:

- calls/returns are primarily **next-PC selection** problems,
- returns are often harder because the target is dynamic, depending on call history.

```
.intel_syntax noprefix

# Conceptual call/ret control flow (no calling convention details)
call    func
add     eax, 1            # resumes here after ret

func:
add     eax, 10
ret
```

6.4 Why Branches Are Expensive

Introduction to misprediction

A branch is expensive when the CPU guesses wrong about control flow. **Misprediction** means:

- the CPU fetched and possibly executed down the wrong path,
- wrong-path work cannot retire and must be discarded,
- the front-end must restart at the correct address and rebuild the instruction window.

The cost is not the branch instruction itself; it is the **lost throughput** during recovery:

- front-end refill (fetch/decode the correct path),
- back-end rescheduling of correct-path work,
- and lost speculative work that never commits.

Branches tend to be cheap when:

- the outcome is predictable (strong bias or repeating pattern),
- the target is stable and easy to predict,
- the hot path stays in the instruction cache and is decode-friendly.

Branches become expensive when:

- the outcome depends on irregular data (hard-to-predict),
- the code path is large or scattered (front-end pressure),
- indirect targets vary widely (harder target prediction).

```
.intel_syntax noprefix

# Unpredictable branch example (conceptual):
# If the low bit of EAX is effectively random, prediction accuracy drops.
test    eax, 1
jz      .Leven
```

```
add     ebx, 3
jmp     .Ldone

.Leven:
add     ebx, 7

.Ldone:
ret
```

Operational takeaways

- Control flow is a front-end constraint: the CPU must know the next PC early.
- Conditional branches are decisions based on flags; prediction keeps the pipeline busy.
- Calls and returns are control flow; returns are uniquely target-dynamic.
- Branch cost is dominated by misprediction recovery, not by the branch opcode.

Chapter 7

The Stack: A Universal Mechanism

Why Every Architecture Has One

Nearly every general-purpose architecture provides a stack mechanism because it solves a universal problem: managing nested control flow and temporary storage efficiently with simple hardware support. The stack is not primarily a "data structure" for algorithms; it is a control-flow and storage discipline that makes function calls practical, supports recursion, enables re-entrant code, and provides a structured way to allocate short-lived storage.
This chapter builds a clean mental model:

- A stack is LIFO storage with a moving pointer.
- Calls/returns naturally form a nesting pattern that LIFO matches perfectly.
- Stack frames provide a predictable layout for return state and locals.
- The OS and ABI care because stacks interact with exceptions, interrupts, context switches, debugging, and security.

7.1 What the Stack Is

LIFO as a universal solution

A **stack** is a region of memory managed with a **stack pointer** (SP) such that the most recently placed item is the first one removed (**LIFO**).

- **Push:** reserve space and store a value.
- **Pop:** load a value and release space.

Why LIFO is universal:

- Program execution naturally nests (calls within calls, blocks within blocks).
- Temporaries and saved state typically become unnecessary in the reverse order they were created.
- The discipline provides deterministic lifetime without requiring complex allocation metadata.

Even when an ISA does not provide explicit `push`/`pop` instructions, the abstraction exists: adjust SP and store/load memory at `[SP]`.

```
.intel_syntax noprefix

# Conceptual stack operations (x86-64 style naming, Intel syntax):
# push:  SP = SP - 8; [SP] = value
# pop:   value = [SP]; SP = SP + 8

# Not an ABI example; just the basic mechanism.
push    rax
push    rbx
```

```
pop     rbx
pop     rax
ret
```

7.2 The Stack and Function Calls

Return addresses

A function call must preserve a return point so execution can resume:

- A **call** transfers control to the callee.
- The **return address** is the address of the instruction after the call.

Many architectures implement this by saving the return address:

- either on the stack,
- or in a dedicated link register (with spilling to the stack when nesting/recursion requires it).

The key concept is universal: nested calls need a nesting-friendly return mechanism. LIFO fits the call nesting structure.

```
.intel_syntax noprefix

# Conceptual x86-64 style:
# call places the return address on the stack, then jumps to target.
# ret pops that address and jumps back.

call    func
add     eax, 1
ret
```

```
func:
add     eax, 10
ret
```

Local storage

Functions commonly require short-lived storage:

- local variables,
- spilled registers (when registers are insufficient),
- temporary buffers,
- saved state needed to restore the caller's context.

The stack provides a fast, structured allocation model:

- allocate by adjusting SP,
- deallocate by restoring SP.

This is efficient because it is constant-time and naturally scoped to the call lifetime.

```
.intel_syntax noprefix

# Conceptual local allocation by stack pointer adjustment:
# (No ABI details, no specific alignment rules stated here.)
sub     rsp, 32           # reserve 32 bytes of local space
mov     [rsp+0],  eax     # store a local / temporary
mov     [rsp+4],  ebx
add     rsp, 32           # release local space
ret
```

7.3 Stack Frames (Conceptual View)

What every function needs

A **stack frame** is the portion of the stack associated with one active function invocation. Conceptually, a frame exists to support:

- **control flow:** preserving the return path (return address) and any necessary call linkage,
- **state preservation:** saving values that must survive across calls (often registers),
- **local storage:** locals and temporaries with a lifetime bounded by the call.

Many systems also track a stable reference point called a **frame pointer** (FP):

- FP can simplify debugging, stack unwinding, and accessing locals with fixed offsets.
- Some optimizations omit FP and address locals relative to SP to free a register and reduce overhead.

```
.intel_syntax noprefix

# Conceptual prologue/epilogue pattern (illustrative, not a specific ABI):
# - establish a frame reference
# - reserve locals
# - restore on exit

push    rbp
mov     rbp, rsp
sub     rsp, 32           # locals

# ... function body ...
```

```
add     rsp, 32
pop     rbp
ret
```

7.4 Why the Stack Matters to OS and ABI

Preview of upcoming booklets

The stack is a boundary where CPU mechanisms meet system-level rules. Operating systems and ABIs impose requirements because stacks participate in:

- **context switching:** saving and restoring thread execution state includes the stack pointer and stack contents.
- **interrupts and exceptions:** precise recovery requires reliable stack layout and unwind metadata in many environments.
- **debugging and profiling:** call stacks enable backtraces, sampling, and attribution of time to functions.
- **security:** stacks are a common attack surface (e.g., overwriting return addresses), leading to mitigations such as canaries, non-executable memory, and control-flow integrity features.
- **calling conventions:** parameter passing, return values, saved registers, and alignment rules depend on ABI contracts and interact with frame layout.

This booklet treats the stack as a universal mechanism. Future booklets will expand into:

- ABI and calling conventions,
- stack alignment and vector usage implications,

- stack unwinding, exception handling, and debug formats,
- OS thread stacks, guard pages, and security mitigations.

Operational takeaways

- The stack is a memory-managed LIFO discipline centered on a stack pointer.
- It matches the natural nesting structure of calls and scopes.
- Stack frames exist to preserve return linkage, saved state, and locals.
- The stack sits at the intersection of CPU execution, OS management, ABI rules, debugging, and security.

Chapter 8

Temporary Values and Object Lifetimes

Why CPUs Love Short-Lived Data

CPUs execute fastest when values stay close to the execution core, remain reusable for a short window, and do not force repeated memory traffic. “Temporary” values are ideal: they are produced, consumed, and discarded quickly, allowing the machine to keep work in registers, overlap execution, and avoid long-latency memory paths.

This chapter builds a practical execution-centric view of lifetimes:

- Temporaries exist to connect operations in the dataflow graph.
- The best lifetime is one that never leaves registers.
- When values spill to the stack or escape to memory, cost rises and parallelism often shrinks.

8.1 Temporary Results in Execution

During execution, most instructions produce intermediate values that exist only to feed later instructions:

- address calculations feeding loads/stores,
- partial sums in reductions,
- loop induction updates,
- condition values feeding branches.

Microarchitecturally, the CPU prefers to keep these values:

- in registers (architectural and, internally, physical registers),
- available to dependent operations with minimal delay,
- without forcing stores/loads that tie progress to cache/TLB behavior.

```
.intel_syntax noprefix

# Temporaries are the glue of dataflow.
# Ideal: produce -> consume while staying in registers.

mov     eax, [rdi]          # load
imul    eax, eax, 7         # temporary result in EAX
add     eax, [rsi]          # consume temporary, combine with another load
ret
```

8.2 Registers vs Stack vs Memory

Think of storage as concentric rings of cost and capability:

Registers

- Fastest access and highest bandwidth into execution units.
- Best for short-lived values and tight-loop working sets.
- Internal renaming allows many in-flight temporaries beyond ISA names.

Stack

- Still memory, but with structured, contiguous access patterns.
- Used for spills (when registers are insufficient), locals, and call linkage.
- Can be relatively cache-friendly, yet still more expensive than registers.

General memory

- Capacity is high, latency is variable, bandwidth is limited by hierarchy and contention.
- Access cost depends on cache hits/misses, TLB behavior, and locality.
- Unpredictable patterns defeat prefetching and increase stalls.

The key insight: even stack usage is a form of memory traffic. When temporaries spill from registers to stack, you add extra stores/loads and create more opportunities for stalls.

```
.intel_syntax noprefix

# Register-resident temporaries (preferred)
add     eax, ebx
imul    eax, eax, 3

# Spill-like pattern (conceptual): temporary forced to memory
```

```
sub     rsp, 16
mov     [rsp], eax        # store temporary to stack (spill)
# ... later ...
mov     eax, [rsp]        # reload temporary
add     rsp, 16
ret
```

8.3 Why Compilers Avoid Memory When Possible

Optimizing compilers try to keep values in registers because memory introduces:

- **latency risk:** loads can miss caches and stall dependent operations,
- **bandwidth pressure:** extra loads/stores compete for limited load/store throughput,
- **ordering and aliasing constraints:** uncertainty about whether pointers refer to the same memory can block reordering,
- **larger working sets:** more memory traffic increases cache pressure and eviction.

Therefore compilers emphasize:

- **register allocation:** assigning live values to registers to avoid spills,
- **instruction scheduling:** placing independent work to overlap unavoidable latency,
- **common subexpression elimination and redundancy removal:** compute once, reuse in registers,
- **scalar replacement / promotion:** turn memory-resident fields into register temporaries when safe,
- **loop optimizations:** reduce memory traffic, improve locality, and increase reuse.

When compilers cannot prove non-aliasing or must preserve observable memory behavior, they often generate more loads/stores to remain correct. This is why “pointer-heavy” code can limit optimization even if the CPU is fast.

8.4 What Assembly Teaches About High-Level Code

Assembly makes the cost model visible. It teaches disciplined questions that apply directly to high-level programming:

Where do my values live?

- Is a value kept in registers, or does it round-trip to memory?
- Do I reload the same data repeatedly because it was not kept live?

How long do values stay live?

- Long-lived values increase register pressure and trigger spills.
- Shortening lifetimes can reduce spills and increase scheduling freedom.

What forces memory traffic?

- escaping addresses (taking the address of a local),
- aliasing through pointers and references,
- unpredictable access patterns and large working sets.

What is the true bottleneck?

- dependency chains (latency-bound),
- execution resource contention (throughput-bound),
- memory stalls (cache/TLB/DRAM-bound),
- branch misprediction (control-flow-bound).

```
.intel_syntax noprefix

# High-level insight: "temporaries" that stay in registers are cheap;
# temporaries that spill or escape to memory are expensive.

# If a loop repeatedly loads the same value due to aliasing assumptions,
# you often see a pattern of reloads rather than reuse.
mov     eax, [rdi]        # reload
add     ecx, eax
mov     eax, [rdi]        # reload again (could be required if memory might
 ↪  change)
add     edx, eax
ret
```

Operational takeaways

- CPUs love short-lived data because it can remain in registers and feed execution without memory stalls.
- The stack is structured memory; it is still far slower than registers and adds load/store pressure.
- Compilers avoid memory when possible to reduce latency risk, bandwidth pressure, and aliasing constraints.

- Reading assembly trains you to reason about value location, lifetime, and the real bottlenecks of execution.

Chapter 9

Common Execution Pitfalls

Mistakes That Break Performance and Correctness

Execution problems rarely come from a single “slow instruction”. They come from wrong mental models: assuming independence where dependencies exist, forgetting hidden architectural state, feeding the pipeline irregular control flow, and equating fewer instructions with faster code.

This chapter lists the most common pitfalls that repeatedly appear in real low-level code review.

9.1 Assuming Instructions Are Independent

A CPU can overlap work only when there is independent work. Many instruction streams look parallel to humans but are serialized by dependencies.

True dependencies dominate

- **RAW (read-after-write)** dependencies form the critical path: a consumer must wait for its producer.
- Long dependency chains are latency-bound even on wide out-of-order cores.

False dependencies still matter

Even when values are logically independent, code can accidentally create constraints:

- register reuse can create apparent coupling (mitigated by renaming, but not always eliminated for all implicit state),
- flags usage can serialize decisions,
- memory aliasing can force conservative ordering.

```
.intel_syntax noprefix

# Looks like "four adds", but it's a single dependency chain through EAX.
add     eax, 1
add     eax, 1
add     eax, 1
add     eax, 1

# Creating independent work can increase overlap.
# (Conceptual example: two accumulators.)
add     eax, 1
add     ecx, 1
add     eax, 1
add     ecx, 1
add     eax, ecx
```

9.2 Ignoring Hidden State (Flags, PC, Stack)

Assembly correctness depends on architectural state that is easy to forget because it is not explicitly named in the instruction operands.

Flags: implicit inputs/outputs

Many instructions modify flags; many decisions consume flags. Inserting an instruction between a flag producer and consumer can silently change program meaning.

```
.intel_syntax noprefix

# Correct
cmp     eax, ebx
jl      .Lless

# Subtle bug: inserted instruction overwrites flags
cmp     eax, ebx
add     ecx, 1            # modifies flags
jl      .Lless            # now tests flags from add, not cmp
```

PC: control flow is a next-address problem

Branches and returns change the PC. The CPU must predict the next PC early; unpredictable control flow can dominate runtime, and incorrect assumptions about fall-through or jump targets break correctness.

Stack: implicit memory and lifetime

Calls, returns, and local storage depend on stack discipline. Common errors include:

- unbalanced stack pointer adjustments,

- corrupting return addresses or locals,
- assuming stack data remains valid beyond its lifetime.

```
.intel_syntax noprefix

# Incorrect: stack imbalance (conceptual)
sub     rsp, 16           # allocate locals
# ... use locals ...
ret                       # forgot to add rsp, 16 before returning
↪  (corrupts control flow)
```

9.3 Writing Pipeline-Unfriendly Code

Even correct code can be hostile to the pipeline, causing front-end starvation, back-end stalls, or poor overlap.

Common pipeline-unfriendly patterns

- **Unpredictable branches:** frequent mispredictions flush useful work.
- **Front-end pressure:** overly large hot loops, heavy branching, and dense instruction mixes can exceed fetch/decode capacity.
- **Load-dependent control:** branches that depend on loads amplify latency because the decision cannot be resolved until data arrives.
- **Serial dependency chains:** a single accumulator or flag chain can prevent out-of-order overlap.

```
.intel_syntax noprefix
```

```
# Load-dependent branch: decision waits for memory
mov     eax, [rdi]        # if this misses cache, branch resolution is
↪  delayed
test    eax, eax
je      .Lzero            # control flow depends on loaded data
add     ebx, 1
jmp     .Ldone
.Lzero:
sub     ebx, 1
.Ldone:
ret
```

9.4 Confusing Instruction Count with Performance

Fewer instructions does not automatically mean faster execution.

Why instruction count is a weak metric

Performance is shaped by:

- **critical path latency** (dependencies),
- **throughput limits** (issue width, port pressure, load/store bandwidth),
- **front-end limits** (fetch/decode bandwidth, code size),
- **memory behavior** (cache/TLB misses, locality),
- **branch behavior** (prediction accuracy and recovery cost).

A shorter sequence can be slower if it:

- increases cache misses (worse locality),

- introduces unpredictable control flow,
- increases resource contention (e.g., forces more loads),
- or lengthens the dependency chain.

```
.intel_syntax noprefix

# Two correct approaches, but cost depends on microarchitecture and data.
# The "shorter" one may be slower if it increases memory traffic or
↪  branches.

# (A) More instructions but predictable, register-resident work
xor     eax, eax
add     eax, ecx
add     eax, edx

# (B) Fewer instructions but may introduce memory or control-flow costs
↪  elsewhere
# (Shown conceptually as a load; real cases vary.)
add     eax, [rdi]        # can be cheap (L1) or expensive (miss)
```

Operational checklist

When reviewing or writing low-level code, ask:

- **Dependencies:** where is the critical path, and can I create independent work?
- **Hidden state:** am I accidentally relying on flags, stack layout, or PC flow?
- **Pipeline friendliness:** am I feeding the front-end a predictable stream and the back-end enough independent work?

- **Real cost drivers:** is the bottleneck memory, branches, or port pressure rather than instruction count?

Chapter 10

Micro-Labs

Hands-On Mental Execution

These micro-labs train the most valuable skill in low-level work: the ability to *mentally execute* a sequence and predict what must be true about architectural state (registers, memory, PC, flags) at each step. The labs are ISA-independent and use pseudo-assembly focused on universal concepts: loads/stores, arithmetic, branches, calls, and a simplified pipeline.

Rules for these labs

- Treat each instruction as an **architectural state transformer**.
- Track **PC**, **registers**, **flags**, and **stack memory** explicitly.
- When considering performance, separate:
  - **latency** (dependencies) from
  - **throughput** (overlap) and from

– **control-flow disruption** (misprediction).

10.1 Tracing a Simple Instruction Sequence

Tracking state changes step by step

Goal: Practice precise state tracking: register updates, memory writes, and flags.

Setup

Initial architectural state:

- `R1 = 5, R2 = 2, R3 = 0`
- `MEM[100] = 7`
- `ZF = 0, CF = 0, SF = 0, OF = 0`

Pseudo-assembly:

```
.intel_syntax noprefix

# PSEUDO-ASM (ISA-independent): Rn are general registers, MEM[] is memory.

LOAD    R3, [100]      # R3 = MEM[100]
ADD     R3, R1         # R3 = R3 + R1, flags updated
SUB     R3, R2         # R3 = R3 - R2, flags updated
STORE   [100], R3      # MEM[100] = R3
```

Trace table

Fill the table by hand. Record state after each instruction.

Step	Instruction	R1	R2	R3	MEM[100]	ZF/SF/CF/OF
0	Initial	5	2	0	7	0/0/0/0
1	LOAD R3, [100]					
2	ADD R3, R1					
3	SUB R3, R2					
4	STORE [100], R3					

Check yourself

After the final step:

- `MEM[100]` should reflect the arithmetic result.
- `ZF` should be 1 only if the final arithmetic result is zero.

10.2 Simulating a Function Call

Following PC, stack, and registers

Goal: Track control transfer, return address, and temporary local storage.

Setup

Assume:

- `PC = MAIN+0`
- `SP = 1000`
- `R1 = 3, R2 = 4`

Pseudo-assembly:

```
.intel_syntax noprefix

# MAIN:
MAIN+0:  CALL    F           # push return PC, jump to F
MAIN+1:  ADD     R1, 1        # runs after return
MAIN+2:  HALT

# F:
F+0:     PUSH    R2           # save a register (conceptual)
F+1:     SUB     SP, 16       # allocate local storage
F+2:     ADD     R1, R2       # compute
F+3:     ADD     SP, 16       # free locals
F+4:     POP     R2           # restore
F+5:     RET                  # pop return PC into PC
```

Trace what must happen

Fill this checklist while tracing:

- On `CALL F`:
  - what value is saved as the return address?
  - how does `SP` change?
  - what does `PC` become?
- Inside `F`:
  - where is `R2` saved?
  - what address range is used for locals after `SUB SP,16`?
- On `RET`:
  - what value becomes the new `PC`?
  - what is `SP` restored to relative to entry?

Correctness invariants

A correct call/return simulation must satisfy:

- After `RET`, execution resumes at `MAIN+1`.
- `R2` after returning equals its original value (saved/restored).
- `SP` after returning equals the original `SP` value (balanced frame).

10.3 Branch vs Straight-Line Code

Predicting execution stalls

Goal: Understand why unpredictable control flow breaks throughput.
Consider two pseudo-sequences that compute the same result.

A) Straight-line (branchless selection)

```
.intel_syntax noprefix

# Compute: R3 = (R1 != 0) ? (R2 + 1) : (R2 - 1)
# Pseudo-branchless form: uses a predicate mask idea.

CMP     R1, 0
MASKNE  M, R1, 0          # M = all-ones if R1 != 0 else 0 (conceptual)
ADD     T1, R2, 1
SUB     T2, R2, 1
SELECT  R3, M, T1, T2     # R3 = (M ? T1 : T2)
```

B) Branching selection

```
.intel_syntax noprefix
```

```
CMP     R1, 0
JE      LZ
ADD     R3, R2, 1
JMP     LD
LZ:     SUB     R3, R2, 1
LD:     NOP
```

Questions

- If `R1` is random-like, which version tends to suffer more on modern pipelined CPUs, and why?
- If `R1` is strongly biased (almost always nonzero), how does that change the expected outcome?

Performance model to use

- Straight-line code increases steady fetch/decode regularity but may execute extra ops.
- Branching code executes fewer ops on one path but risks misprediction when the condition is hard to predict.

10.4 Execution Timeline Exercise

Drawing a simplified pipeline timeline

Goal: Visualize overlap and stalls using a 5-stage simplified pipeline model:

$$\texttt{F (Fetch)} \rightarrow \texttt{D (Decode)} \rightarrow \texttt{E (Execute)} \rightarrow \texttt{M (Mem)} \rightarrow \texttt{R (Retire)}$$

Assume the following simplifications:

- Each stage takes 1 cycle per instruction in steady state.
- A load has an extra 2-cycle latency in the `M` stage (cache miss penalty in this toy model).
- A taken branch resolves in `E`. If mispredicted, there is a 3-cycle flush/restart penalty.

Sequence to plot:

```
.intel_syntax noprefix

I1: LOAD    R1, [A]
I2: ADD     R2, R2, 1
I3: ADD     R3, R1, R2
I4: CMP     R3, 0
I5: JE      L1
I6: ADD     R4, R4, 1
```

Task A: dependency reasoning

- Identify which instruction depends on the load result `R1`.
- Decide whether `I2` can overlap with the load latency.

Task B: timeline table

Draw a timeline with cycles on top and stages filled per instruction. Start with this template:

Inst	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
I1	F	D	E	M	M	M	R			
I2		F	D	E	M	R				
I3			F	D	E	M	R			
I4				F	D	E	M	R		
I5					F	D	E	M	R	
I6						F	D	E	M	R

Task C: branch penalty

Now assume `I5` was mispredicted as not-taken but is actually taken:

- mark the cycles where wrong-path work would be discarded,
- insert a 3-cycle bubble after branch resolution,
- restart fetch at `L1`.

What you should gain from these labs

- You can trace architectural state changes with discipline (registers, flags, PC, stack).
- You can reason about when control flow disrupts the front-end and why mispredictions hurt.
- You can separate algorithmic correctness from execution cost drivers (dependencies, memory latency, pipeline refill).
- You can draw a simplified timeline that captures overlap, stalls, and recovery costs.

Appendices

Appendix A — Execution Cheat Sheet

Instruction lifecycle summary

- **Architectural model (ISA view):** instructions appear to execute one-by-one in program order, updating architectural state precisely.
- **Microarchitectural reality:** CPUs overlap stages, execute out of order, and speculate; results become architecturally visible only at retirement.

Core lifecycle (conceptual)

Fetch $\rightarrow$ **Decode** $\rightarrow$ **(Rename / Queue)** $\rightarrow$ **Schedule** $\rightarrow$ **Execute** $\rightarrow$ **Retire**

What can go wrong at each step

- **Fetch:** front-end starvation from I-cache/iTLB misses, code size pressure, poor branch prediction.
- **Decode:** limited decode bandwidth, complex/long encodings, internal expansion of instructions.

- **Rename / Queue:** running out of internal tracking resources (physical registers, reorder/queue entries).
- **Schedule:** dependency chains prevent issue; port/resource contention prevents issue.
- **Execute:** long-latency operations (notably cache misses) block dependent work.
- **Retire:** in-order commit can bottleneck if the machine frequently flushes due to misprediction or exceptions, or if retirement bandwidth is saturated.

Minimal mental checklist

- **Critical path:** which operations are on the dependency chain that sets latency?
- **Available parallelism:** how much independent work exists to overlap?
- **Front-end health:** can fetch/decode supply the back-end continuously?
- **Memory health:** are loads/stores hitting in cache and translating efficiently?
- **Control flow:** are branches predictable, and are targets easy to predict?

CPU component map

Architectural state (ISA-visible)

- **PC / instruction pointer:** next architectural instruction address.
- **Registers:** general-purpose, vector/SIMD, flags/status.
- **Memory effects:** architecturally visible loads/stores and atomic rules.

Front-end (builds the instruction stream)

- **Branch predictor:** predicts direction and target to keep fetch continuous.
- **I-cache / iTLB:** instruction bytes and address translation caches.
- **Fetch and decode:** bring bytes, find boundaries, translate into internal operations.
- **Instruction queue:** buffers decoded work for the back-end.

Back-end (executes work)

- **Rename / physical registers:** remove false dependencies, enable many values in flight.
- **Scheduler / issue logic:** selects ready operations for execution resources.
- **Execution units:** integer ALUs, vector units, branch unit, address generation, load/store pipelines.
- **Load/store subsystem:** store buffers, memory disambiguation, cache access.

Memory hierarchy (dominant cost driver)

- **D-cache (L1), L2, L3:** increasing capacity, increasing latency.
- **DTLB:** translation cache for data addresses.
- **DRAM:** high latency; throughput limited by bandwidth and concurrency.
- **Prefetchers:** attempt to fetch data early when patterns are predictable.

Retirement (makes results official)

- **Reorder/retirement logic:** commits results in program order for precise architectural state.
- **Recovery:** discards speculative work after misprediction and restarts fetch.

Execution terminology quick reference

Correctness terms

- **ISA / architectural semantics:** the contract defining observable behavior.
- **Architectural state:** registers/flags/PC and visible memory effects.
- **Precise exceptions:** architectural state corresponds to completion of a prefix of the instruction stream.

Performance terms

- **Latency:** time from producing an input to consuming the result on a dependency chain.
- **Throughput:** steady-state rate of completing independent work.
- **Critical path:** longest dependency chain that limits progress.
- **ILP (instruction-level parallelism):** how much independent work exists.
- **MLP (memory-level parallelism):** how many memory misses can be overlapped.

Dependency terms

- **RAW:** read-after-write (true dependency).
- **WAR:** write-after-read (false dependency, often removed by renaming).
- **WAW:** write-after-write (false dependency, often removed by renaming).
- **Flags dependency:** implicit coupling through status flags (can serialize decisions).

Front-end vs back-end vs memory

- **Front-end bound:** fetch/decode/prediction cannot supply enough work.
- **Back-end bound:** execution resources (ports/units/schedulers) are saturated or dependencies dominate.
- **Memory bound:** stalls from cache/TLB misses or bandwidth pressure dominate.

Speculation and branches

- **Speculative execution:** executing along a predicted path before certainty.
- **Branch prediction:** guessing direction/target to keep fetch running.
- **Misprediction:** wrong guess; wrong-path work is discarded, pipeline refills on correct path.

One-page practical checklist (carry this)

- What is the **bottleneck**? (front-end / back-end / memory / branches)
- What is the **critical path**? (true dependencies)

- Is there enough **independent work**? (ILP/MLP)
- Are **branches predictable**? (avoid mispredict-heavy hot paths)
- Are **loads/stores** cache-friendly and translation-friendly? (locality, alignment, working set)
- Are you accidentally using **hidden state**? (flags, stack, implicit dependencies)

```
.intel_syntax noprefix

# Minimal reminder in code form:
# 1) Producer->consumer distances matter (dependencies).
# 2) Loads can dominate cost if they miss cache/TLB.
# 3) Branches are cheap when predicted, expensive when mispredicted.

# Dependency chain example (latency-bound)
mov     rax, [rdi]
add     rax, 1
imul    rax, rax, 7

# Independent work example (throughput opportunity)
mov     r8,  [rdi]
mov     r9,  [rsi]
add     r8,  r9

# Control flow hazard example (mispredict risk if unpredictable)
test    eax, 1
jz      .Leven
add     ebx, 3
.Leven:
ret
```

Appendix B — What Comes Next

This booklet built the execution-centric mental model that transfers across architectures and toolchains:

- the instruction lifecycle from fetch to retirement,
- the separation between architectural contract (ISA) and implementation (microarchitecture),
- the real cost drivers: dependencies, memory hierarchy, and control flow,
- and the discipline of tracing architectural state (PC, registers, flags, stack).

With this foundation, you can move from “reading assembly” to *reasoning about systems*.

How this booklet prepares you for calling conventions and ABI

An ABI is the operational contract that makes separately compiled code interoperable. It defines what must remain true across function boundaries:

- where arguments and return values live (registers vs stack),
- which registers a function must preserve,
- how stack frames are laid out and aligned,
- how calls/returns interact with unwinding, debugging, and exceptions,
- and what “correct” means for mixed-language and mixed-module systems.

What you already have from this booklet:

- **Control flow mechanics:** calls/returns are next-PC transfers with a return path.
- **Stack discipline:** LIFO frames, locals as stack-resident storage, and why imbalance breaks correctness.
- **Architectural vs internal state:** why retirement and precise state matter for interrupts/exceptions.

In the ABI booklet(s), you will extend this into concrete rules:

- prologue/epilogue structure and frame layout,
- register-save conventions and spill strategies,
- stack alignment and vector/SIMD implications,
- red zones / shadow space concepts (where applicable),
- and call-chain correctness under optimization.

How this booklet prepares you for x86, ARM, and RISC-V

Architectures differ in encoding, register sets, addressing modes, and privileged features—but the execution principles are stable.

What transfers directly

- **Fetch/decode pressure:** code size and control-flow shape matter on every ISA.
- **Dependencies:** critical paths dominate regardless of instruction mnemonics.
- **Memory hierarchy:** caches/TLBs and locality dominate real cost.

- **Speculation:** branch prediction exists because control flow is uncertain everywhere.
- **Retirement/precise state:** architectures enforce architectural correctness; implementations overlap and speculate.

What changes by ISA (and how to approach it)

- **x86:** rich addressing, variable-length encodings, flags-heavy idioms, and frequent implicit-state patterns.
- **ARM:** fixed-length encodings (in AArch64), explicit condition-setting patterns, and a strong emphasis on regular decode/dispatch.
- **RISC-V:** simple, regular encodings and a clean base ISA with optional extensions; performance depends heavily on implementation choices.

Your method going forward:

- Learn the **ISA contract** (semantics and architectural state).
- Then map sequences onto the **execution model**: front-end limits, back-end resources, memory behavior, and branch predictability.

How this booklet prepares you for performance and optimization topics

Performance work is not “assembly tricks”; it is constraint management.

Core optimization questions you can now answer

- **Is this latency-bound?** (long dependency chain, load-to-use delays)
- **Is this throughput-bound?** (port/resource contention, issue limits)
- **Is this memory-bound?** (cache/TLB misses, bandwidth pressure)
- **Is this control-flow-bound?** (mispredictions, indirect target unpredictability)
- **Is this front-end-bound?** (fetch/decode limits, code size, I-cache pressure)

What comes next in the performance track

- **Memory hierarchy deep dive:** cache lines, associativity, prefetch, TLB behavior, page walks.
- **Measuring reality:** profiling, sampling, counters (conceptually), and designing experiments.
- **Loop-level optimization:** unrolling, vectorization, software pipelining, dependency breaking.
- **Branch behavior:** prediction patterns, branchless transforms, indirect control flow costs.
- **Compiler interaction:** how optimization passes reshape lifetimes, register pressure, and memory traffic.

```
.intel_syntax noprefix

# A compact "next steps" reminder:
# - ABI: understand what must be preserved across calls and how frames are
↪  laid out.
```

```
# - ISA: learn the architectural contract; never confuse it with
↪  performance.
# - Performance: find the real bottleneck (front-end, back-end, memory,
↪  branches).

# Dependency vs overlap (what to look for in any ISA)
mov     rax, [rdi]        # potential long latency
add     rbx, 1            # independent work that may overlap
add     rcx, 2            # independent work that may overlap
add     rax, rbx          # depends on load and rbx update
ret
```