# x86-64 Assembly

## A Complete 12-Chapter Course on the Modern, Real-World Architecture

**Topics covered:**

The 8086-to-x86-64 lineage & why AMD created the 64-bit extension

Registers, RFLAGS & the AT&T/Intel syntax divide · Full SIB addressing

Real calling conventions (System V & Microsoft x64) · Branchless CMOV

Privilege rings & paging · SIMD previewed · Syscalls vs. the Windows API

Real toolchains: NASM, GAS, ELF, PE

**Exercises:** 36 hands-on exercises with worked solutions

**Format:** A4 · Dark-theme code examples · the third and final course in the Assembly/Machine Language arc (assembly1 → cpu8bit1 → assembly2)

# Table of Contents

CHAPTER 1 OF 12

# Four Decades of Accretion

## x86-64 Assembly

Chapter 1 · Four Decades of Accretion

`assembly1-1` named x86-64 as the architecture carrying "four decades of backward-compatible accretion," and `cpu8bit1-11` closed its own course by naming the Z80's own richness as a preview, at 1976 scale, of exactly this. This chapter is where that promise gets cashed in — the real lineage from a 1978 chip to the architecture running underneath the device this page is being read on, and the single strangest, most concrete fact about it: the CPU never really let go of anything it used to be.

### The Lineage — 8086 to x86-64

The whole story starts with the **Intel 8086** (1978) — a 16-bit processor, and the direct ancestor of every chip this course covers. It wasn't binary-compatible with the 8080 the way `cpu8bit1-5` 's own Z80 was, but it was deliberately designed so 8080 assembly programs could be mechanically translated into 8086 assembly — the same 8080 lineage `cpu8bit1-5` traced through Zilog, now showing up on Intel's own side of the family tree.

- **80286** (1982) — introduced the first, limited version of **protected mode**, a genuinely new concept this course previews below and covers fully in `assembly2-8`.
- **80386** (1985) — the big one: a full 32-bit extension (often called IA-32), with complete protected mode and paging. This chip anchored the dominant x86 era for the better part of two decades — "386" became informal shorthand for the entire 32-bit generation.
- **Pentium and its successors** (1990s–2000s) — continued the 32-bit era while adding the vector-instruction extensions `assembly2-9` previews (MMX, and later SSE).
- **AMD64 / x86-64** (2003) — the 64-bit extension this entire course is actually about.

### Why AMD, Not Intel

Here's the genuinely surprising part: the 64-bit extension to x86 wasn't Intel's idea. Intel's own answer to 64-bit computing was **Itanium (IA-64)** — a completely new, non-backward-compatible architecture, built from scratch rather than extended from the existing x86 lineage. AMD took the opposite approach: extend the existing, familiar x86 instruction set with 64-bit capability while keeping everything that already worked, working. AMD's approach — **AMD64** — won out commercially, and Intel eventually licensed and adopted AMD's own compatible design, selling it under the name **Intel 64** (formerly branded EM64T). This is one of the rare, well-documented moments where AMD, not Intel, set the direction the entire industry actually followed.

**ONE ARCHITECTURE, SEVERAL NAMES**

AMD64, x86-64, x64, Intel 64, and EM64T all refer to genuinely the same underlying 64-bit architecture — the naming split exists purely because two competing companies each wanted their own branding for a design that, for compatibility's sake, has to behave identically either way. This course uses "x86-64" throughout as the neutral, most common term, but real documentation and real code comments will use all of the above interchangeably.

## Real, Protected, and Long Mode — A First Preview

A modern x86-64 CPU doesn't just run in one mode — it can actually behave like several different, much older CPUs, on demand, because the hardware never removed the old behavior when it added new capability:

- **Real mode** — the original 16-bit, 8086-compatible mode. Every x86-64 PC, including one built this year, literally boots into real mode first, behaving exactly like a 1978 8086, before the operating system switches it into a newer mode.
- **Protected mode** — the 32-bit mode introduced by the 80286/80386, with real memory protection.
- **Long mode** — the 64-bit mode this course is actually about, where the registers, addressing, and instructions covered from `assembly2-2` onward actually apply.

That boot-time detour through real mode isn't a quirky edge case — it's "four decades of accretion" made completely literal. The chip can still pretend to be its own 1978 ancestor, on command, because nothing from that era was ever actually deleted.

| CHIP | YEAR | COMPANY | BIT WIDTH | KEY ADDITION |
|---|---|---|---|---|
| 8086 | 1978 | Intel | 16-bit | The original x86 architecture, assembly-source-compatible with the 8080 |
| 80286 | 1982 | Intel | 16-bit | The first, limited protected mode |
| 80386 | 1985 | Intel | 32-bit | Full protected mode and paging — the IA-32 era begins |
| AMD64 / x86-64 | 2003 | AMD | 64-bit | A backward-compatible 64-bit extension — created by AMD, not Intel |

**THE CLEAREST ARTIFACT IS STILL AHEAD**

This chapter tells the history in words. `assembly2-2` shows the exact same history sitting inside a single register's own name — `RAX`, `EAX`, `AX`, `AH`, and `AL` are all genuinely the same physical storage, viewed at four different historical widths at once. That's the single most concrete artifact of everything this chapter just described.

## Why This Matters for the Rest of the Course

Every register name, every addressing mode, every instruction covered from here forward exists inside layers of decisions this lineage produced. `cpu8bit1-2` and `cpu8bit1-5` traced the 6502 and Z80 back to two clean

founding constraints, each a single, coherent story. x86-64 doesn't have one story — it has five decades of stories, each one built on top of the last without erasing it. That's the shape this entire course is going to keep tracing, one concrete feature at a time.

## Hands-On Exercises

**EXERCISE 1**

Explain why AMD, rather than Intel, is credited with creating the 64-bit x86-64 extension. Name Intel's own competing approach, and explain — using this chapter's own reasoning — why AMD's backward-compatible design won out over it.

View solution

**EXERCISE 2**

Explain what "real mode" is, and explain why the fact that every modern x86-64 PC still boots into it is described in this chapter as "four decades of accretion made completely literal" rather than just an interesting trivia fact.

View solution

**EXERCISE 3**

List at least three different names this chapter gives for the same 64-bit x86 architecture, and explain why this naming confusion exists in the first place.

View solution

### CHAPTER 1 QUICK REFERENCE

- **8086** (1978, Intel) → **80286** (1982, limited protected mode) → **80386** (1985, full 32-bit IA-32) → **AMD64/x86-64** (2003, AMD)
- The 8086 was assembly-source-compatible (not binary-compatible) with the 8080 — the same 8080 lineage cpu8bit1-5 traced through the Z80
- AMD, not Intel, created the 64-bit extension — Intel's own competing Itanium (IA-64) was a non-backward-compatible clean break, and lost out commercially
- AMD64, x86-64, x64, Intel 64, and EM64T all refer to the same architecture — the naming split is purely a branding artifact
- **Real mode** (16-bit, 8086-compatible) → **protected mode** (32-bit) → **long mode** (64-bit, this course's real subject)
- Every modern x86-64 PC boots into real mode first — a literal, ongoing act of backward compatibility, not just a historical footnote

- assembly2-2's own sub-register naming maze is this chapter's history made visible in a single register

CHAPTER 2 OF 12

# Registers, Reimagined

## x86-64 Assembly

Chapter 2 · Registers, Reimagined

`assembly2-1` promised the clearest artifact of x86-64's own history would show up inside a single register's name. Here it is — sixteen general-purpose registers, eight of them carrying four historical layers of naming at once, and eight of them carrying none at all.

### The 16 General-Purpose Registers

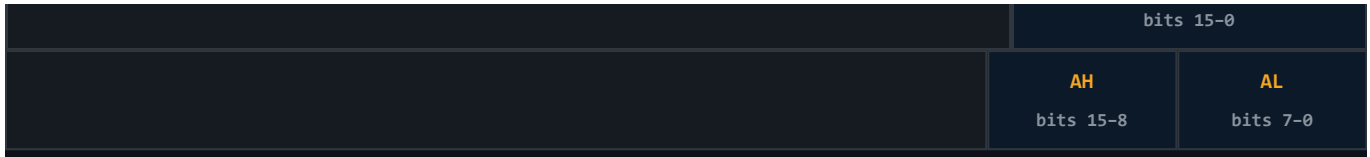
x86-64 has 16 general-purpose 64-bit registers: `RAX`, `RBX`, `RCX`, `RDX`, `RSI`, `RDI`, `RBP`, `RSP`, and `R8`–`R15`. That's comparable in scale to `cpu8bit1-6`'s own Z80 total (14, counting the main and shadow sets together) — but with a real difference: all 16 of x86-64's registers are usable *simultaneously*, with no `EXX`-style swap ever required to reach half of them.

The first eight trace directly back to the original 8086 (`assembly2-1`'s own starting point), each with a historically-loaded name still partly tied to a specific role: **A**ccumulator, **B**ase, **C**ounter, **D**ata, **S**ource **I**ndex, **D**estination **I**ndex, **B**ase **P**ointer, **S**tack **P**ointer. Some of those roles are still real, not just historical trivia — `RCX` is still the implicit loop counter the `LOOP` instruction (`assembly2-6`) automatically decrements, exactly the way its name has promised since 1978.

`R8` through `R15` are entirely new — added specifically for the 64-bit extension `assembly2-1` credited to AMD. They have no inherited name at all, just numbers, because they never needed to carry any legacy forward.

### The Sub-Register Maze — RAX/EAX/AX/AH-AL

One physical 64-bit storage location, addressable at four different widths, under four different names:




`EAX` ("Extended AX") *was* the whole register back when the 80386 was the newest chip in the lineage. `AX` was the whole register back in the 8086's own original 16-bit era. `AH` / `AL` go back that far too — letting even 1978-era code address just the upper or lower half of a 16-bit value. Nothing here was ever removed; every later chip just added a wider name on top of the one that came before.

The same four-level pattern applies to three more of the original eight: `RBX` / `EBX` / `BX` / `BH` - `BL`, `RCX` / `ECX` / `CX` / `CH` - `CL`, and `RDX` / `EDX` / `DX` / `DH` - `DL`.

> **WRITING EAX CLEARS RAX'S UPPER HALF — WRITING AX OR AL DOESN'T**
>
> A genuine, well-documented asymmetry: writing any value to a 32-bit sub-register (like `EAX`) automatically **zeroes** the upper 32 bits of the full 64-bit register. Writing to the 16-bit or 8-bit forms (`AX`, `AH`, `AL`) does **not** — everything above the bits actually written is left completely untouched. The two sub-register sizes behave differently by design, and assuming one behaves like the other is a real, easy mistake.

Historically, `RSI` / `RDI` / `RBP` / `RSP` didn't have their own `H` / `L` 8-bit forms the way `AX` / `BX` / `CX` / `DX` did — only their 16-bit `SI` / `DI` / `BP` / `SP` forms existed classically. x86-64 specifically *added* new 8-bit low-byte access to these — `SIL`, `DIL`, `BPL`, `SPL` — using a special encoding prefix. One more concrete instance of `assembly2-1`'s own theme: a genuinely new capability, layered on without ever touching what already existed.

## R8–R15's Own Cleaner Pattern

Because `R8` – `R15` never had to inherit an old name, they got a uniform, modern naming scheme from day one: `R8` (64-bit), `R8D` (32-bit, "D" for Double word), `R8W` (16-bit, "W" for Word), `R8B` (8-bit, "B" for Byte) — and the exact same pattern for R9 through R15, no exceptions, no historical quirks.

| WIDTH | LEGACY REGISTER (RAX FAMILY) | NEW REGISTER (R8 FAMILY) |
|---|---|---|
| 64-bit | RAX | R8 |
| 32-bit | EAX | R8D |
| 16-bit | AX | R8W |
| 8-bit | AH / AL (two separate sub-names) | R8B |

The new registers' naming is strictly more regular than the legacy ones' — a small, direct illustration that code and conventions never carrying old baggage in the first place are simply cleaner than ones that had to accumulate it.

**SIXTEEN REGISTERS, NO SWAPPING REQUIRED**

Set against `cpu8bit1-6`'s own Z80 register total, x86-64's 16 always-available general-purpose registers are a genuinely more straightforward design than needing an `EXX`-style swap to reach half of a comparable total — one more small way this architecture's real complexity shows up in naming and history rather than in access mechanics.

## Hands-On Exercises

**EXERCISE 1**

Using this chapter's own bit diagrams, identify exactly which bits of RAX are affected by a write to AL, a write to AX, and a write to EAX, respectively.

View solution

**EXERCISE 2**

RAX currently holds 0xFFFFFFFF12345678. The instruction `MOV EAX, 0` executes. Using this chapter's own warn-box, state RAX's full 64-bit value afterward, and explain why it's not 0xFFFFFFFF00000000.

View solution

**EXERCISE 3**

Explain why R8's own sub-register naming (R8/R8D/R8W/R8B) is more regular than RAX's own (RAX/EAX/AX/AH-AL), and connect the reason directly back to assembly2-1's own "four decades of accretion" theme.

View solution

**CHAPTER 2 QUICK REFERENCE**

- 16 general-purpose registers, all simultaneously usable: RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, and R8–R15
- The original 8 trace back to the 8086 and carry historically-loaded names — some (like RCX/LOOP) still have real implicit uses
- **RAX** → **EAX** → **AX** → **AH/AL** — one physical register, four historical widths, four different names
- Writing a 32-bit sub-register (EAX) zeroes the upper 32 bits of the 64-bit register; writing a 16-bit or 8-bit sub-register does not
- SIL/DIL/BPL/SPL are x86-64-only additions — new capability layered on, nothing removed
- **R8–R15** use a clean, uniform R8/R8D/R8W/R8B pattern — no legacy naming to carry, because they're entirely new

- All 16 registers are always available at once — no EXX-style swap needed, unlike cpu8bit1-6's own Z80 shadow set

CHAPTER 3 OF 12

# RFLAGS and Instruction Basics

## x86-64 Assembly

Chapter 3 · RFLAGS and Instruction Basics

Before writing real programs, this chapter covers the two things every one of them will lean on: the flags register, and the handful of instructions that set it. It also resolves a real cliffhanger from `cpu8bit1-12` — and introduces a genuinely new kind of gotcha neither LC-3 nor the 6502/Z80 ever had: two competing, incompatible-looking ways to write the exact same instruction.

## RFLAGS — x86-64's Status Register

`cpu8bit1-3`'s 6502 packed its flags into an 8-bit P register; `cpu8bit1-6`'s Z80 did the same with its 8-bit F register. x86-64's **RFLAGS** is nominally 64 bits wide — but only a small handful of those bits are actually meaningful flags; the rest are reserved, unused, or hold rarely-touched system-level state. Even the flags register itself carries its own quiet layer of accretion.

The flags this course actually uses:

- **CF** (Carry Flag) — the same underlying concept `cpu8bit1-3` and `cpu8bit1-6` both covered.
- **ZF** (Zero Flag) — set when a result is exactly zero, the same idea as every prior chip's own Z flag.
- **SF** (Sign Flag) — the same idea as the 6502's N and the Z80's S: set when a result is negative.
- **OF** (Overflow Flag) — signed overflow, distinct from Carry's unsigned signal, the same distinction `cpu8bit1-3` drew for the 6502's own V flag.

### Resolving cpu8bit1-12's Cliffhanger

`cpu8bit1-12` found that the 6502's `CMP` and the Z80's `CP` set Carry in *opposite* directions for the identical comparison. x86-64 now supplies a third data point — and it sides with the Z80: after a `CMP` or `SUB`, **CF is set if a borrow was needed** (the destination was smaller than the source, unsigned), exactly matching the Z80's own convention, not the 6502's inverted one.

## Basic Instructions — MOV, ADD, SUB, CMP

```
MOV RAX, RBX    ; RAX = RBX (a copy — "MOV" doesn't clear RBX, a slight historical misnomer)
ADD RCX, 5      ; RCX += 5
```

```
SUB RDX, RAX   ; RDX -= RAX — sets CF/ZF/SF/OF
CMP RAX, RBX   ; computes RAX - RBX, sets flags, discards the result — like SUB without storing
```

`CMP` here is the direct x86-64 counterpart to `cpu8bit1-12`'s own 6502 `CMP` and Z80 `CP` — same underlying idea, and, per the resolution above, the same flag convention as the Z80's.

## The AT&T vs. Intel Syntax Divide

Here's the genuinely new kind of gotcha: real-world x86-64 code exists in **two different syntaxes** that look, at a glance, like different instruction sets entirely — even though they describe exactly the same underlying operations.

| | INTEL SYNTAX (THIS COURSE) | AT&T SYNTAX |
|---|---|---|
| Operand order | destination, source | source, destination — **reversed** |
| Register prefix | none | `%` (e.g. `%rax`) |
| Immediate prefix | none | `$` (e.g. `$5`) |
| Size suffix | none (inferred from register name) | b/w/l/q appended to the mnemonic (byte/word/long/quad) |
| Used by | NASM, MASM — this course's own examples | GAS, GCC inline assembly, GDB's default disassembly |

The same instruction, in both syntaxes:

```
; Intel syntax (this course)
MOV RAX, RBX
ADD RCX, 5

; AT&T syntax — same two instructions
movq %rbx, %rax
addq $5, %rcx
```

**THE OPERAND ORDER GENUINELY REVERSES**

This is the single most common source of confusion when reading code in the "other" syntax: `MOV RAX, RBX` (Intel: destination first) and `movq %rbx, %rax` (AT&T: source first) describe the *exact same operation* — copy RBX into RAX — despite RAX and RBX appearing in opposite positions on the line. Misreading one syntax using the other's operand-order rule silently swaps source and destination.

**THIS COURSE USES INTEL SYNTAX THROUGHOUT**

Every example from here forward uses Intel syntax, matching NASM — the toolchain `assembly2-11` covers directly. Real code encountered elsewhere, especially anything touching Linux/GCC inline assembly or a default GDB disassembly listing, will very likely be in AT&T syntax instead (GDB can be told to switch to Intel syntax, but doesn't by

default). Recognizing both, even while primarily writing in one, is a genuinely necessary real-world skill this course's own examples won't otherwise force you to practice.

## Hands-On Exercises

**EXERCISE 1**

Translate this Intel-syntax sequence into AT&T syntax, using this chapter's own table: `MOV RAX, RCX` followed by `SUB RAX, 10`.

View solution

**EXERCISE 2**

Using this chapter's own resolution of cpu8bit1-12's cliffhanger, state whether x86-64's CF convention after CMP matches the 6502's or the Z80's, and explain what CF being SET actually means in x86-64 terms.

View solution

**EXERCISE 3**

Explain why RFLAGS being nominally 64 bits wide, while only using a small handful of those bits for the flags this course actually cares about, counts as a small instance of assembly2-1's own "four decades of accretion" theme.

View solution

**CHAPTER 3 QUICK REFERENCE**

- **RFLAGS** — nominally 64 bits, only a handful of bits actually used as flags (CF, ZF, SF, OF, among others)
- **CF after CMP/SUB matches the Z80's convention** — set if a borrow was needed, the opposite of the 6502's own convention (cpu8bit1-12)
- **MOV/ADD/SUB/CMP** — the basic instruction set this course builds on; CMP computes but discards, only affecting flags
- **Intel syntax** (this course, NASM, MASM) — destination first, no register/immediate prefixes
- **AT&T syntax** (GAS, GCC inline asm, GDB default) — source first, %-prefixed registers, $-prefixed immediates, b/w/l/q size suffixes
- The operand order genuinely reverses between the two syntaxes — the single most common real-world reading mistake

- This course commits to Intel syntax throughout, but recognizing AT&T syntax is a real, necessary skill outside it

CHAPTER 4 OF 12

# Memory Addressing — The Full SIB Addressing Mode

## x86-64 Assembly

Chapter 4 · Memory Addressing — The Full SIB Addressing Mode

Three courses, three addressing stories: LC-3's clean base+offset, the 6502's zero-page workaround, the Z80's displacement-based indexing. x86-64 doesn't pick one — it combines the ideas behind all of them into a single addressing mode genuinely richer than anything either prior course covered, plus one capability neither ever offered at all.

### The Addressing Journey So Far

| ARCHITECTURE | RICHEST ADDRESSING MODE COVERED | AUTOMATIC INDEX SCALING? |
|---|---|---|
| LC-3 (assembly1-3) | Base+offset — a register plus a small constant | No |
| 6502 (cpu8bit1-4) | (zp),Y — a zero-page pointer plus an index register | No |
| Z80 (cpu8bit1-7) | (IX+d) — a register plus an 8-bit displacement | No |
| x86-64 (this chapter) | Base + Index×Scale + Displacement, all in one instruction | **Yes** |

### The Full SIB Addressing Mode

x86-64's richest memory operand computes its effective address as:

```
effective address = Base + (Index × Scale) + Displacement
```

- **Base** — any general-purpose register, holding a starting address, playing the same role as the Z80's own HL or the 6502's zero-page pointer.
- **Index** — any general-purpose register, typically an array index.
- **Scale** — a multiplier applied to Index, restricted to **1, 2, 4, or 8** — matching byte/word/dword/qword element sizes exactly.
- **Displacement** — a constant offset, the same role LC-3's PC-relative offset and the Z80's own `+d` displacement played.

```
MOV RAX, [RBX + RCX*4 + 8]   ; RBX = array base, RCX = index, *4 for 4-byte elements, +8 skips a header
```

## Why This Matters — No Manual Multiply Required

On the 6502 or Z80, indexing into an array of 4-byte elements would mean multiplying the index by 4 *before* the address could be computed — typically two shift-left operations, or repeated addition, as a separate step every single time. x86-64's **Scale** performs that exact multiplication as part of the address computation itself, inside the same instruction that actually accesses memory. This is a genuinely new capability, not just a faster version of something the earlier chips already did.

> **SCALE IS ONLY 1, 2, 4, OR 8 — NOTHING ELSE**
>
> Need to index into an array of 3-byte or 5-byte elements? Scale can't do it directly — those values aren't legal scale factors. The multiply-for-free capability only covers the specific sizes that actually correspond to real data widths (byte/word/dword/qword). Anything else still needs a genuine, separate multiply instruction first, the same real limitation `cpu8bit1-7`'s own warn-box already flagged for the Z80's own richer-but-not-unlimited addressing modes.

## Not Every Component Is Required

Base, Index, Scale, and Displacement are all optional individually — an instruction only pays for the pieces it actually uses:

```
MOV RAX, [RBX]              ; base only — like Z80's own (HL)
MOV RAX, [RBX + 8]          ; base + displacement — like Z80's own (IX+d)
MOV RAX, [RBX + RCX]        ; base + index, scale defaults to 1
MOV RAX, [RBX + RCX*4 + 8]  ; all four components together
```

## RIP-Relative Addressing — PC-Relative, Returned

x86-64 also added a genuinely new mode: `[RIP + offset]`, computing an address relative to the **current instruction pointer**. This is conceptually the exact same idea as `assembly1-3`'s own LC-3 PC-relative `LD` — but for a completely different, distinctly modern reason. LC-3 needed PC-relative addressing because a 16-bit instruction had no room for a full address at all. x86-64 doesn't have that bit-budget problem (`assembly2-1`'s own variable-length instructions solve it) — RIP-relative addressing exists instead to support **position-independent code**: a program whose data references stay correct no matter where in memory the operating system actually loads it, a real security and shared-library requirement neither LC-3 nor the 6502/Z80 ever had to think about. The same underlying technique, reinvented decades later to solve an unrelated problem.

**THIS IS WHAT THE CAPSTONE WILL ACTUALLY USE**

`assembly2-12`'s own capstone leans directly on this chapter's full SIB mode — walking a real array using base+index×scale addressing in a single instruction is exactly the kind of concrete richness neither `assembly1-10` nor `cpu8bit1-12`'s own capstones had available to them.

## Hands-On Exercises

**EXERCISE 1**

Given RBX (base) = 0x1000, RCX (index) = 5, a scale of 8, and a displacement of 16, compute the effective address of `[RBX + RCX*8 + 16]`, showing your work.

View solution

**EXERCISE 2**

Explain specifically what extra instruction(s) a 6502 or Z80 program would need, that an equivalent x86-64 program using Scale wouldn't, when indexing into an array of 8-byte elements.

View solution

**EXERCISE 3**

Explain what RIP-relative addressing and LC-3's own PC-relative addressing (assembly1-3) have in common mechanically, and explain why each architecture actually needed it for a genuinely different reason.

View solution

**CHAPTER 4 QUICK REFERENCE**

- **Effective address = Base + (Index × Scale) + Displacement** — the richest single addressing mode across this whole three-course arc
- Scale must be 1, 2, 4, or 8 — matching byte/word/dword/qword element sizes exactly; nothing else is directly supported
- Scale performs index-multiplication as part of the address computation — the genuinely new capability neither the 6502 nor the Z80 offered
- All four components (Base/Index/Scale/Displacement) are individually optional — an instruction only pays for what it uses
- **RIP-relative addressing** — PC-relative addressing's conceptual return, now for position-independent code rather than a fixed-width instruction's bit budget

- assembly2-12's capstone directly exercises SIB addressing for real array access

CHAPTER 5 OF 12

# The Stack and Calling Conventions

## x86-64 Assembly

Chapter 5 · The Stack and Calling Conventions

`assembly1-7` built one simple, self-invented LC-3 calling convention. `cpu8bit1-12` used one straightforward subroutine convention per chip. This chapter is where that simplicity ends: x86-64 doesn't have a calling convention — it has (at least) two real, competing, mutually incompatible standards, and getting the wrong one is a genuine way to crash a real program.

### The Stack — RSP and RBP

**RSP** is a full 64-bit stack pointer, free to point anywhere — as fully general as the Z80's own SP from `cpu8bit1-7`, with no 6502-style page lock. `PUSH` / `POP` are native instructions, operating on 64-bit values by default in long mode.

**RBP** plays a role neither LC-3 nor the 6502/Z80 ever formally needed: a stable *frame pointer*. As a function runs, RSP itself keeps moving — every push and pop shifts it. RBP is set once, at the start of a function, and held fixed for its entire duration, giving a stable reference point for that function's own parameters and local variables regardless of how RSP moves around them. It's conceptually the same idea as the 6502/Z80's own convention of dedicating a register to a specific job (`cpu8bit1-3`'s own R6-as-stack-pointer precedent from `assembly1-7`) — just applied to a genuinely new problem, deep local-variable frames, that this arc's simpler subroutines never had.

Worth an honest, brief note: modern compiled code often skips the frame pointer entirely ("frame pointer omission"), computing local-variable offsets directly from RSP instead, as a performance optimization. RBP is a real, common convention, not a hardware requirement.

### Register-Based Argument Passing — A Genuine Departure

Neither `assembly1-7`'s LC-3 subroutines nor `cpu8bit1-12`'s own capstone formally passed "arguments" at all — a value was just already sitting in whatever register the subroutine happened to expect, by ad-hoc agreement between caller and callee. Real x86-64 calling conventions formalize this completely: the first several arguments to a function are passed directly in **specific, standardized registers**, not via the stack — the stack is reserved for overflow arguments beyond that fixed count, and for local storage.

## Two Competing Standards

This is the real complexity: which registers hold which arguments depends entirely on which operating system the code targets.

| | SYSTEM V AMD64 ABI (LINUX, MACOS, BSD) | MICROSOFT X64 (WINDOWS) |
|---|---|---|
| First integer/pointer arguments, in order | RDI, RSI, RDX, RCX, R8, R9 (6 registers) | RCX, RDX, R8, R9 (4 registers) |
| Arguments beyond that | Passed on the stack | Passed on the stack |
| Shadow space | None required | Caller must reserve 32 bytes on the stack, even if all arguments fit in registers |
| Integer/pointer return value | RAX | RAX (the one genuine point of agreement) |

A concrete example — calling a hypothetical function with three integer arguments:

```
; System V AMD64 ABI (Linux/macOS)
MOV RDI, 10   ; 1st argument
MOV RSI, 20   ; 2nd argument
MOV RDX, 30   ; 3rd argument
CALL some_function

; Microsoft x64 (Windows) — same call, different registers
MOV RCX, 10   ; 1st argument
MOV RDX, 20   ; 2nd argument
MOV R8, 30    ; 3rd argument
SUB RSP, 32   ; reserve the required 32-byte shadow space first
CALL some_function
```

**SHADOW SPACE IS GENUINELY WINDOWS-ONLY**

Microsoft's convention requires the *caller* to reserve 32 bytes of stack space before every call — space the called function is allowed to use freely, even though the arguments themselves already fit entirely in registers. There's no System V equivalent at all; code written against one convention that ignores this requirement under the other will misbehave in ways that can be genuinely difficult to trace back to the actual cause.

One more real, shared rule worth naming: in *both* conventions, `RBX`, `RBP`, and `R12`–`R15` are **callee-saved** — a called function must preserve their values before returning, exactly the register-preservation discipline `assembly1-7` first established for R7 and `cpu8bit1-12`'s own capstone reinforced by deliberately choosing C over B.

## Why Real Programs Must Actually Care

In every prior course, "the calling convention" was something the programmer invented for their own self-contained program — nothing outside that program ever needed to agree with it. x86-64 calling conventions are a real **contract**: honoring them correctly is what lets hand-written assembly interoperate with the operating system, system libraries, and code compiled by an entirely different compiler. Get it wrong, and the failure isn't a logic bug in your own code — it's a silent violation of an agreement code outside your control was relying on, often surfacing as a crash or corrupted data far from the actual mistake.

**FROM AN INFORMAL HABIT TO A REAL STANDARD**

`assembly1-7` taught calling conventions as good practice within one program. This chapter is the same underlying idea, scaled up into something genuinely external and non-negotiable — the exact kind of real-world complexity a teaching architecture and two 1970s-era chips never needed to force onto the reader.

## Hands-On Exercises

**EXERCISE 1**

A function is called with three integer arguments: 100, 200, 300. State exactly which register holds each argument under System V AMD64 ABI, and separately under Microsoft x64.

View solution

**EXERCISE 2**

Explain what "shadow space" is in the Microsoft x64 calling convention, using this chapter's own example, and explain why System V has no equivalent requirement.

View solution

**EXERCISE 3**

Explain why this chapter describes x86-64's calling conventions as a genuine "contract," and contrast that against assembly1-7's own single, self-invented LC-3 calling convention.

View solution

**CHAPTER 5 QUICK REFERENCE**

- **RSP** — fully flexible 64-bit stack pointer, like the Z80's own SP, with no 6502-style page lock

- **RBP** — a stable frame pointer for a function's own locals, a genuinely new need neither prior course's simpler subroutines had
- Arguments are passed in specific registers first, the stack only for overflow — a real, formalized contract, not an ad-hoc habit
- **System V AMD64 ABI** (Linux/macOS): RDI, RSI, RDX, RCX, R8, R9 — 6 register arguments, no shadow space
- **Microsoft x64** (Windows): RCX, RDX, R8, R9 — 4 register arguments, plus mandatory 32-byte caller-reserved shadow space
- Both conventions agree: the integer/pointer return value goes in RAX, and RBX/RBP/R12–R15 are callee-saved
- These conventions are a real, external contract — violating them can silently break interoperability with the OS or other compiled code

CHAPTER 6 OF 12

# Control Flow at Scale

## x86-64 Assembly

Chapter 6 · Control Flow at Scale

Jumps and loops will feel familiar after two full prior courses of them. This chapter covers that familiar ground quickly, then introduces something genuinely new: a way to make a decision without ever branching the instruction pointer at all.

### Unconditional and Conditional Jumps

`JMP` is unconditional, the same idea as the 6502's own `JMP` or LC-3's `BRnzp`. The conditional **Jcc** family reads RFLAGS the same way `assembly2-3` already covered — including `JC` / `JNC`, a direct parallel to `cpu8bit1-8`'s own 6502 `BCC` and `cpu8bit1-12`'s own Z80 `JR NC`, now with the flag convention already settled in `assembly2-3`.

**JG AND JA ARE NOT THE SAME "GREATER THAN"**

x86-64 deliberately provides two separate families of comparison jumps: **JG**/**JL** (and friends) use *signed* comparison, while **JA**/**JB** (and friends) use *unsigned* comparison — genuinely different conditions, even though both read as "jump if greater/less" in plain English. Using `JG` on data meant to be treated as unsigned (or vice versa) is a real, common bug: the two families can disagree on the exact same bit pattern depending on whether it's interpreted as a negative signed number or a large unsigned one.

### LOOP — A DJNZ-Style Instruction

x86-64 has a native `LOOP` instruction: decrement RCX, and jump if RCX isn't zero — one instruction, doing exactly what `cpu8bit1-8`'s own Z80 `DJNZ` does.

```
MOV RCX, 5
LOOP_START
        ...                 ; loop body
        LOOP LOOP_START  ; RCX-- ; jump to LOOP_START if RCX != 0
```

**MODERN CODE MOSTLY AVOIDS LOOP ANYWAY**

Despite being just as compact and elegant as `DJNZ`, real modern x86-64 code rarely uses `LOOP` — on modern CPU microarchitectures, an explicit `DEC RCX` / `JNZ` pair is typically executed faster than the single `LOOP` instruction,

because modern internal pipelining optimizes the two ordinary instructions better than it optimizes `LOOP` itself. A genuinely elegant, DJNZ-like instruction, deliberately avoided in practice for real performance reasons — one more instance of the "richness doesn't automatically mean faster" theme `cpu8bit1-7` and `cpu8bit1-8` both already established.

## CMOV — A Genuinely New Idea: Branchless Code

Nothing in `assembly1` or `cpu8bit1` offered this: **CMOVcc** conditionally moves a value based on the current flags — but the instruction itself always executes, with no branch taken either way. A decision, without ever redirecting the instruction pointer.

```
CMP  RAX, RBX
CMOVL RAX, RBX   ; if RAX < RBX (signed), RAX becomes RBX — branchless max
```

Compare this directly against `cpu8bit1-12`'s own capstone, which found a maximum using an explicit `CMP`/branch/`STA` sequence on both the 6502 and Z80 — real branches, genuinely taken or not taken. `CMOVL` computes the identical logical result without a single conditional jump anywhere in the sequence.

Why this matters: modern CPUs rely heavily on branch prediction and deep instruction pipelining (concepts this course hasn't needed until now) to run fast. A *mispredicted* branch forces the CPU to discard speculative work and refill its pipeline — a real, measurable cost. For simple "if condition, then set this value" patterns, `CMOV` sidesteps misprediction risk entirely, since there's no branch to mispredict in the first place.

**CMOV ISN'T AUTOMATICALLY THE FASTER CHOICE EITHER**

`CMOV` always does the conditional-move work, every single time, regardless of which way the condition actually goes — where a real branch, correctly predicted (which a modern branch predictor manages the large majority of the time for genuinely predictable patterns), can skip the "not taken" path's cost entirely. For a condition that's rarely true and easy for hardware to predict, an ordinary branch can still outperform `CMOV`. This is exactly the same honest nuance `cpu8bit1-7`'s own cycle-cost table and `cpu8bit1-8`'s own clock-speed caveat already established: more capability is not the same claim as automatically faster.

| TECHNIQUE | LC-3 / 6502 / Z80 | X86-64 |
|---|---|---|
| Simple conditional branch | BR family / Jcc-equivalent branches | Jcc — same idea, richer signed/unsigned condition set |
| Decrement-and-loop | assembly1-6's BR loop, cpu8bit1-8's INX+CPX+BNE, or DJNZ | LOOP — conceptually the same as DJNZ, but often avoided on modern hardware |
| Conditional assignment | Always required an actual branch | CMOV — genuinely branchless, a technique none of the prior three offered |

**A CLEANER TOOL FOR THE CAPSTONE**

`assembly2-12`'s own capstone could implement its own max-finding logic using `CMOV` instead of an explicit branch — the exact same task `cpu8bit1-12`'s capstone solved with branching on two different chips, now solvable with no branch at all.

## Hands-On Exercises

**EXERCISE 1**

Explain the difference between JG and JA, using this chapter's own signed-vs-unsigned distinction, and explain why using the wrong one for a given data type is a real, documented bug source rather than just a style preference.

View solution

**EXERCISE 2**

Trace this chapter's own branchless-max example — `CMP RAX, RBX` then `CMOVL RAX, RBX` — given RAX = 10 and RBX = 25 beforehand. State RAX's final value and explain each step.

View solution

**EXERCISE 3**

Explain why LOOP, despite being conceptually similar to cpu8bit1-8's own celebrated Z80 DJNZ, is often avoided in real modern x86-64 code — and explain why this chapter treats that fact as a further instance of the site's own recurring "richness doesn't automatically mean faster" theme.

View solution

**CHAPTER 6 QUICK REFERENCE**

- **JMP** — unconditional; **Jcc** family — conditional, reading RFLAGS per assembly2-3
- **JG/JL vs. JA/JB** — signed vs. unsigned comparison, a real and common source of bugs when mismatched to the data
- **LOOP** — decrement RCX and jump if nonzero, one instruction, conceptually matching cpu8bit1-8's own DJNZ
- Modern code often prefers explicit DEC+JNZ over LOOP for real microarchitectural performance reasons
- **CMOVcc** — conditionally moves a value with no branch at all — a genuinely new technique, sidestepping branch-misprediction cost
- CMOV isn't automatically faster than a real branch — it always does its own work regardless of the condition, unlike a correctly-predicted skipped branch

- The capstone (assembly2-12) can revisit cpu8bit1-12's own max-finding task using CMOV instead of branching

CHAPTER 7 OF 12

# The Sheer Scale of the Instruction Set

## x86-64 Assembly

Chapter 7 · The Sheer Scale of the Instruction Set

Step back from individual instructions for a moment. `cpu8bit1-11` formalized RISC vs. CISC using a 56-instruction chip and a ~158-instruction chip as its two data points. This chapter adds the third, modern one — and finds the clean binary that comparison suggested is, at this scale, more complicated than it first looked.

### Counting the Instructions — A Genuinely Hard Question

`assembly1-5`'s LC-3 had a small, exact, countable opcode set. `cpu8bit1-2` counted 56 6502 mnemonics; `cpu8bit1-5` counted roughly 158 for the Z80. x86-64 doesn't have an equally clean number to cite. Base integer instructions alone already number in the hundreds; once every SIMD/vector extension family (`assembly2-9`'s own preview) is counted as its own set of distinct mnemonic-and-operand-form combinations, the real total runs into the **thousands**. This chapter deliberately doesn't cite one precise figure — pinning down an exact count depends entirely on what's being counted, and a false-precision number would be less honest than admitting the real answer is "a lot, by any reasonable measure, and the exact figure depends on your counting method."

### Why So Many — Extending cpu8bit1-5's Own Trick

`cpu8bit1-5` explained the Z80's CB/DD/ED/FD prefix bytes as a way to unlock additional opcode spaces beyond a single byte's own 256-value ceiling. x86-64 uses exactly the same underlying idea, stacked several layers deeper: a `0x0F` escape byte (in use since the 80386 era) unlocks an entire second opcode table; the **REX prefix** (already named in `assembly2-2` as the mechanism behind `SIL`/`DIL`/`BPL`/`SPL` and R8–R15 access) unlocks 64-bit operand sizes and the extended register set; and further escape sequences layer entire SIMD instruction families (`assembly2-9`) on top of all of that. It's the identical mechanism the Z80 pioneered at a small, four-prefix scale — just applied many more times over.

### A Brief, Honest Tour of Instruction Categories

Cataloging the full instruction set is explicitly out of scope for this course — instead, a categorized overview:

- **Data movement** — `MOV` and its relatives, including `MOVZX` / `MOVSX` (zero-extend / sign-extend a smaller value into a larger register), directly relevant to `assembly2-2`'s own sub-register material.
- **Arithmetic and logic** — `ADD` / `SUB` / `MUL` / `DIV` / `AND` / `OR` / `XOR` / `NOT` / `NEG`, plus specialized variants.
- **Control flow** — covered in full in `assembly2-6`.
- **String/memory-block instructions** — `MOVS` / `CMPS` / `SCAS` / `STOS`, combined with a `REP` prefix to repeat an operation across an entire block of memory as one conceptual instruction. A genuinely distinctive x86 family with no real equivalent in LC-3 or the 8-bit chips this arc covered.
- **Bit manipulation** — instructions like `BT` / `BTS` / `BTR` and `POPCNT` for testing, setting, and counting individual bits directly.
- **System/privileged instructions** — previewed fully in `assembly2-8`.
- **SIMD/vector instructions** — previewed fully in `assembly2-9`.

## RISC vs. CISC, Confirmed — and Complicated — at Modern Scale

By raw instruction count, x86-64 is exactly what `cpu8bit1-11` predicted the Z80 previewed: the CISC trajectory, taken to modern scale.

| ARCHITECTURE | INSTRUCTION COUNT | RISC/CISC LEAN |
|---|---|---|
| LC-3 (assembly1) | A small, fixed teaching-ISA set | Predates the framing entirely |
| 6502 (cpu8bit1-2) | 56 mnemonics | RISC precedent |
| Z80 (cpu8bit1-5) | ~158 mnemonics | CISC precedent |
| x86-64 (this chapter) | Hundreds to thousands, depending on counting method | CISC realized at modern scale |

But the real, honest ending to this comparison is more nuanced than "x86-64 proves CISC won." Modern x86-64 CPUs internally translate their own complex, CISC-style instructions into simpler internal **micro-ops**, executed on hardware that itself behaves in a genuinely RISC-like way — uniform, pipeline-friendly operations under the hood. The instruction set a programmer writes is CISC; the microarchitecture actually running it, underneath, borrows heavily from RISC's own design lessons. `cpu8bit1-11`'s own clean binary — a chip is either RISC or CISC — was accurate for two 1970s designs studied at the instruction-set level. At this modern scale, the two philosophies don't compete anymore; they coexist inside the same chip, at different layers.

> **TWO PREVIEWS STILL AHEAD**
>
> `assembly2-8` covers the system/privileged instructions this chapter only named; `assembly2-9` gives SIMD its own deliberately light-touch treatment, honestly scoped as a topic large enough to be its own course.

## Hands-On Exercises

**EXERCISE 1**

Explain, using this chapter's own reasoning, why stating an exact x86-64 instruction count is genuinely harder than it was for the 6502 or Z80 — what specifically makes "how many instructions" an ambiguous question here that it wasn't for the earlier chips?

View solution

**EXERCISE 2**

Explain how x86-64's own prefix mechanism (the 0x0F escape byte, the REX prefix) is the same underlying idea as cpu8bit1-5's own Z80 CB/DD/ED/FD prefixes, just applied at greater scale.

View solution

**EXERCISE 3**

Explain what this chapter's own "CISC instructions on a RISC-like microarchitecture" finding complicates about cpu8bit1-11's own clean RISC-vs-CISC binary, and explain why this chapter treats that complication as an honest conclusion rather than an oversimplification to avoid.

View solution

## CHAPTER 7 QUICK REFERENCE

- x86-64's real instruction count is genuinely hard to pin down — hundreds of base instructions, thousands once every SIMD extension is counted
- The 0x0F escape byte and REX prefix extend cpu8bit1-5's own Z80 CB/DD/ED/FD prefix-byte mechanism, at greater scale
- Instruction categories: data movement, arithmetic/logic, control flow, string/block operations (REP-prefixed), bit manipulation, system/privileged, SIMD/vector
- REP-prefixed string instructions (MOVS/CMPS/SCAS/STOS) are a genuinely distinctive x86 family with no LC-3/6502/Z80 equivalent
- By raw instruction count, x86-64 confirms cpu8bit1-11's own CISC-trajectory prediction at modern scale
- Modern CPUs translate CISC instructions into RISC-like internal micro-ops — the RISC/CISC binary from cpu8bit1-11 coexists inside one chip rather than one philosophy "winning"

CHAPTER 8 OF 12

# Privilege Levels & Protected/Long Mode

## x86-64 Assembly

Chapter 8 · Privilege Levels & Protected/Long Mode

Every chapter so far has extended something the prior arc already covered — richer registers, richer addressing, richer instructions. This chapter is different: it's genuinely new ground. Nothing in `assembly1` or `cpu8bit1` has an equivalent for what's about to be covered, because neither LC-3 nor the 6502/Z80 enforce any concept of trust at all.

### A Genuinely New Kind of Chapter

Every instruction covered across both prior courses could be executed by literally any code running on those chips. LC-3 has no privileged instructions; the 6502 and Z80 have none either — no rings, no memory protection, no hardware-enforced OS/user distinction anywhere in either design. x86-64 is fundamentally different: the CPU itself enforces a hierarchy of trust, and certain things simply cannot be done from the wrong level, no matter what the program tries.

### Protection Rings — 0 Through 3

- **Ring 0** — kernel/OS mode. Full access to every instruction, every memory location, every piece of hardware. This is where the operating system kernel itself runs.
- **Ring 3** — user mode. Restricted. Ordinary application programs run here, and a specific category of instructions — ones that reconfigure paging, change privilege level, or touch hardware directly — are simply **forbidden**. Attempting one doesn't fail silently; it triggers a CPU exception.

> **RINGS 1 AND 2 EXIST, BUT ARE RARELY USED**
>
> x86-64's design theoretically supports four rings, but real modern operating systems — Linux and Windows both — effectively use only rings 0 and 3. The full four-ring design is more a historical/theoretical capability than something real-world software actually exploits.

This is the real hardware mechanism behind `assembly1-8`'s own OS/hardware boundary concept — LC-3's `TRAP` was a clean, simplified stand-in for exactly this idea: user code cannot directly do privileged things, and has to go through a controlled channel instead. `assembly2-10` covers that channel's real, modern form.

## Virtual Memory via Paging

Every running process sees its own private, seemingly complete address space — even though many processes actually share the same physical RAM underneath. **Paging** is the mechanism: memory is divided into fixed-size chunks (pages, typically 4KB), and the CPU's own hardware, guided by OS-managed page tables, translates every *virtual* address a program's instructions reference into a real *physical* address, on every single memory access.

If a virtual address has no valid mapping in a process's own page table, any attempt to touch it raises a **page fault** — a real, controlled CPU exception.

> **THE SAME PROBLEM, TWO VERY DIFFERENT ANSWERS**
>
> `cpu8bit1-3`'s own warn-box described a real 6502 hazard: a stack pushed past its limit doesn't error at all — SP silently wraps around and quietly corrupts whatever data used to be there. That's "a program touching memory it shouldn't" with *zero* protection. x86-64's page fault is the exact same category of problem — a program reaching memory it has no business touching — handled with real, hardware-enforced detection instead of silent corruption. This is genuine architectural progress on a problem this arc named concretely, several chapters ago.

## Closing assembly2-1's Own Loop

Recall `assembly2-1`'s own three modes: real, protected, and long. Now their real difference can be stated precisely:

- **Real mode** has no paging and no protection at all — a program running in real mode has exactly the same total, unguarded access to memory that every LC-3, 6502, and Z80 program in this entire arc has always had. Booting into real mode isn't just "acting like an old chip" in spirit — it's a genuine, temporary return to zero memory protection.
- **Protected mode** (introduced by the 80386, `assembly2-1`'s own lineage) is where paging and rings first appear.
- **Long mode** — this course's real subject — uses an extended, deeper paging structure to address vastly more memory than 32-bit protected mode ever could.

An instruction attempted from the wrong ring doesn't just quietly fail — it raises a specific, named, well-documented exception: the **General Protection Fault** (`#GP`), a real CPU-level event any programmer doing low-level debugging eventually meets in person.

| CONCEPT | LC-3 (ASSEMBLY1) | 6502 / Z80 (CPU8BIT1) | X86-64 (THIS CHAPTER) |
|---|---|---|---|
| Privilege levels | None | None | Rings 0–3 (practically 0 and 3) |
| Memory protection | None — any address accessible | None — cpu8bit1-3's own unprotected stack wraparound | Paging + page faults, hardware-enforced |

| CONCEPT | LC-3 (ASSEMBLY1) | 6502 / Z80 (CPU8BIT1) | X86-64 (THIS CHAPTER) |
|---|---|---|---|
| Who can run any instruction | Any code | Any code | Only ring 0 for privileged instructions — ring 3 gets a #GP fault |

## Hands-On Exercises

**EXERCISE 1**

Explain why LC-3 and the 6502/Z80 have no possible equivalent of "an instruction is forbidden in ring 3" at all — what would each of those architectures need to add before such a concept could even exist?

View solution

**EXERCISE 2**

Using this chapter's own tip-box, explain how a page fault is a genuine architectural improvement over cpu8bit1-3's own silent 6502 stack-wraparound hazard — both are the same underlying problem, handled completely differently.

View solution

**EXERCISE 3**

Using this chapter's own explanation of real mode, state exactly what memory protection a program running in real mode actually has, and compare that directly to the protection LC-3 and the 6502/Z80 provide.

View solution

### CHAPTER 8 QUICK REFERENCE

- Genuinely new ground: LC-3 and the 6502/Z80 have no privilege concept at all — any code can execute any instruction
- **Ring 0** (kernel) has full access; **Ring 3** (user) is forbidden from privileged instructions — rings 1/2 exist but are rarely used by real OSes
- **Paging** — fixed-size pages, hardware-translated virtual-to-physical addresses, enforced per process
- A **page fault** is the hardware-enforced answer to exactly the kind of hazard cpu8bit1-3's own unprotected 6502 stack wraparound left completely undetected
- Real mode = zero protection, matching LC-3/6502/Z80 exactly; protected mode introduces rings/paging; long mode extends paging further
- A forbidden instruction from ring 3 raises a **General Protection Fault (#GP)** — a real, named, commonly-encountered CPU exception

- assembly2-10 covers the sanctioned channel (syscalls) user-mode code actually uses to ask the kernel for privileged work

CHAPTER 9 OF 12

# A First Look at SIMD

## x86-64 Assembly

Chapter 9 · A First Look at SIMD

Every instruction covered since `assembly1-1` — across all three courses — has operated on one piece of data at a time. This chapter previews something genuinely different: instructions that operate on *several* pieces of data at once. Deliberately light-touch, on purpose — this is honestly a topic large enough to be its own course.

### Why Vector Instructions Exist — SIMD in One Idea

**SIMD** — Single Instruction, Multiple Data. Every ordinary instruction this entire arc has covered works on one value: one register, one memory operand. Adding two 4-element arrays together the ordinary way takes four separate `ADD` instructions, one pair at a time. A SIMD instruction packs several values into one wide register and performs the *same operation* on all of them **simultaneously**, inside a single instruction's own execution — not four instructions doing one thing each, but one instruction doing four things at once.

### The Real Lineage — MMX → SSE → AVX

| EXTENSION | YEAR | REGISTERS | WIDTH | REAL NOTE |
|---|---|---|---|---|
| MMX | 1996 | MM0–MM7 | 64-bit | Physically shared the existing x87 floating-point register space — a real, genuinely awkward compromise |
| SSE | 1999+ | XMM0–XMM15 | 128-bit | A genuinely separate register set, resolving MMX's own sharing problem |
| AVX | 2011+ | YMM0–YMM15, ZMM (AVX-512) | 256/512-bit | Extends the same register numbering further and wider |

MMX's own register-sharing compromise meant a program couldn't freely mix MMX instructions with ordinary floating-point math without real overhead switching between the two uses of the same physical registers. SSE fixed this by giving vector instructions their own dedicated register file entirely. This whole lineage is one more concrete instance of `assembly2-1`'s own "four decades of accretion" theme — SIMD has its own internal history of layered extensions, each one adding capability on top of the last, mirroring `assembly2-2`'s own RAX/EAX/AX nesting, just for an entirely different feature.

## One Worked Example

Adding two 4-element arrays of single-precision floats, the SIMD way:

```
MOVUPS XMM0, [array1]   ; load 4 floats from array1 into XMM0
MOVUPS XMM1, [array2]   ; load 4 floats from array2 into XMM1
ADDPS  XMM0, XMM1       ; add all 4 pairs simultaneously — ONE instruction
MOVUPS [result], XMM0   ; store all 4 results back at once
```

`ADDPS` — **ADD P**acked **S**ingle-precision — the mnemonic itself describes exactly what it does. The equivalent ordinary, scalar version of this same task would need four separate `ADD`-family instructions, one value pair at a time; `ADDPS` does all four in one.

**NOT EVERY CPU SUPPORTS EVERY EXTENSION**

A real program can't just assume AVX-512, or even AVX, is available — SIMD support varies by CPU model and generation. Real code checks which extensions the running CPU actually supports (via the `CPUID` instruction) before using anything beyond the most basic, universally-present SSE instructions. Using an unsupported instruction doesn't degrade gracefully; it faults.

## Honestly Scoping This Topic

This is genuinely as far as this course goes. A real, complete treatment of SIMD would need to cover the many packed integer/float data-type variants, each instruction set's own feature-detection requirements, real memory-alignment rules (some SIMD instructions fault on unaligned memory operands), and AVX-512's own further masking and broadcasting capabilities. All of that is large enough to be a genuinely separate course on its own — this chapter deliberately stops here rather than pretending a shallow tour is complete coverage.

**A BIG SHARE OF ASSEMBLY2-7'S OWN INSTRUCTION COUNT**

`assembly2-7` already named SIMD extensions as a major reason x86-64's own honest instruction count runs into the thousands rather than the hundreds — this chapter is the concrete look at exactly which family of instructions is doing most of that counting.

## Hands-On Exercises

**EXERCISE 1**

Using this chapter's own ADDPS example, explain what "packed" means in this context, and explain specifically why one ADDPS instruction accomplishes what would otherwise take four separate scalar ADD instructions.

View solution

**EXERCISE 2**

Explain the real historical MMX register-sharing problem described in this chapter, and explain specifically why SSE's own separate XMM register set was a genuine improvement rather than just a wider version of the same idea.

View solution

**EXERCISE 3**

List at least two specific things this chapter names as out of scope, and explain why the chapter treats naming them explicitly as more honest than simply not mentioning them at all.

View solution

### CHAPTER 9 QUICK REFERENCE

- **SIMD** — Single Instruction, Multiple Data: one instruction operates on several packed values at once
- **MMX** (1996) — shared registers with the x87 FPU, a real design compromise
- **SSE** (1999+) — a genuinely separate 128-bit XMM register file
- **AVX** (2011+) — extends the same registers to 256-bit (YMM) and 512-bit (ZMM, AVX-512)
- `ADDPS` — ADD Packed Single-precision — adds 4 float pairs in one instruction instead of 4 separate ADDs
- Real programs must check CPU support (via CPUID) before using advanced SIMD extensions — nothing degrades gracefully
- Deliberately scoped light — data type variants, alignment rules, and AVX-512 masking/broadcasting are all honestly out of scope

CHAPTER 10 OF 12

# Talking to the OS

## x86-64 Assembly

Chapter 10 · Talking to the OS

`assembly2-8` established that ring-3 code can't do privileged things directly — it has to ask. This chapter covers how it actually asks, and the answer genuinely splits in two depending on which operating system is listening.

### Recap — assembly1-8's Clean Abstraction

LC-3's `TRAP` was one unified mechanism: a single instruction, one vector table, a small fixed set of OS services (`GETC` / `OUT` / `PUTS` / `HALT`) defined by the LC-3 specification itself, identical on every LC-3 system that ever exists. Real x86-64 has no equivalent single mechanism — what "asking the OS" even looks like depends entirely on which operating system the code is running under.

### Linux — The SYSCALL Instruction

x86-64 Linux provides a dedicated `SYSCALL` instruction — a fast, purpose-built replacement for the older, slower software-interrupt-based mechanism (`INT 0x80`) that 32-bit x86 Linux used. The convention: the syscall number goes in RAX, and arguments go in RDI, RSI, RDX, **R10**, R8, R9.

```
MOV RAX, 1      ; syscall number for write()
MOV RDI, 1      ; file descriptor 1 = stdout
MOV RSI, msg    ; pointer to the string
MOV RDX, 5      ; length
SYSCALL           ; make the call — return value comes back in RAX
```

**R10, NOT RCX — A REAL, EASY MISTAKE**

`assembly2-5`'s own System V AMD64 ABI uses **RCX** as the 4th argument register for ordinary function calls. Syscalls use **R10** in that same 4th argument position instead — because the `SYSCALL` instruction itself internally clobbers RCX as part of how it works. Assuming syscall arguments follow the exact same register order as a normal function call is a real, understandable mistake that silently passes the wrong value.

The return value comes back in RAX — the one point of real consistency with `assembly2-5`'s own ordinary function-call convention.

## Windows — No Direct Syscalls for Applications

Windows takes a genuinely different approach. It does not expose a stable, documented, directly-invokable syscall interface to ordinary application code the way Linux does — Windows's own internal syscall numbers are considered private implementation detail, and can (and do) change between versions and even updates. Instead, applications call into the **Windows API** (Win32 API): a large collection of ordinary functions, living in system DLLs like `kernel32.dll`, which themselves make the real, internal, unstable syscalls on the application's behalf.

The practical consequence: an x86-64 Windows assembly program calling something like `WriteFile` or `ExitProcess` does so with an ordinary `CALL` instruction, using `assembly2-5`'s own Microsoft x64 calling convention, to a function imported from a DLL — not via any direct syscall-style instruction the programmer writes.

## A Genuine, Important Divergence

| | LC-3 (assembly1-8) | Linux x86-64 | Windows x86-64 |
|---|---|---|---|
| Mechanism | TRAP + vector table | SYSCALL instruction | CALL into a Windows API DLL function |
| Stability | Fixed by the LC-3 spec itself | A stable, documented, directly-invokable interface | Internal syscalls unstable — only the Windows API surface is the real contract |
| Argument passing | Fixed register (R0) | RDI, RSI, RDX, **R10**, R8, R9 | Ordinary Microsoft x64 calling convention (assembly2-5) — RCX, RDX, R8, R9 |

This is the real point: LC-3 had one portable OS abstraction, identical everywhere. x86-64 genuinely has two different models depending on target OS — code written to make Linux syscalls directly has no Windows equivalent to fall back on, and vice versa, even though the underlying CPU instructions covered in every prior chapter of this course work identically on both.

**REAL TOOLING, COMING NEXT**

`assembly2-11` covers the real toolchains (NASM, GAS, ELF, PE) needed to actually assemble and link a program that uses either of these mechanisms — the OS divergence covered here shows up again at the linking stage, not just the instruction level.

## Hands-On Exercises

**EXERCISE 1**

Explain specifically why the Linux syscall convention uses R10 instead of RCX for the 4th argument, even though assembly2-5's own regular System V calling convention uses RCX in that exact position.

View solution

**EXERCISE 2**

Explain why a Windows assembly program calling a Windows API function uses an ordinary CALL instruction rather than anything resembling Linux's own SYSCALL — what's fundamentally different about how each OS exposes its own services to application code?

View solution

**EXERCISE 3**

Explain why this chapter describes x86-64's OS interface situation as "genuinely two different models" rather than "one interface with two syntaxes," using assembly1-8's own single unified TRAP mechanism as the contrast point.

View solution

### CHAPTER 10 QUICK REFERENCE

- LC-3's TRAP was one unified, spec-defined OS abstraction — x86-64 has no equivalent single mechanism
- **Linux**: the SYSCALL instruction, syscall number in RAX, arguments in RDI/RSI/RDX/R10/R8/R9, return value in RAX
- Syscalls use R10, not RCX, for the 4th argument — because SYSCALL itself clobbers RCX internally
- **Windows**: no stable, documented direct syscall interface for applications — internal syscall numbers can change between versions
- Windows applications call the Windows API (DLL functions like kernel32.dll) via an ordinary CALL, using assembly2-5's own Microsoft x64 convention
- This is a genuine divergence, not just a syntax difference — code targeting one OS's mechanism has no equivalent on the other
- assembly2-11 covers the real, OS-specific tooling (NASM/GAS, ELF/PE) needed to actually build a working program using either mechanism

CHAPTER 11 OF 12

# Assemblers & Toolchains in Practice

## x86-64 Assembly

Chapter 11 · Assemblers & Toolchains in Practice

`assembly1-9` taught the two-pass assembler algorithm, symbol tables, object files, and linking using LC-3 as the example. This chapter is the confirmation that none of that was simplified for teaching purposes — real, modern tools do exactly the same thing, just at real-world scale.

### NASM and GAS — Two Real Assemblers

**NASM** (Netwide Assembler) uses Intel syntax — matching this course's own committed choice from `assembly2-3` — and is widely used for standalone, hand-written assembly projects with a simple, self-contained command-line workflow. **GAS** (GNU Assembler, part of binutils) uses AT&T syntax by default and is the assembler GCC itself uses internally to emit its own generated assembly — it's what shows up when reading compiler output or inline assembly inside C code, deeply embedded in the Linux/GNU toolchain.

For hand-writing assembly directly — what every example in this course has been doing — NASM's Intel syntax and simpler standalone workflow are typically the more approachable starting point; GAS becomes unavoidable the moment the goal shifts to reading or embedding assembly inside compiler-generated code.

### assembly1-9's Own Two-Pass Assembler, For Real

Real assemblers use the exact same two-pass approach `assembly1-9` taught, for the exact same reason: Pass 1 builds a symbol table by scanning the whole file first; Pass 2 generates real machine code, resolving forward references against that already-complete table. This isn't a simplified teaching model real tools have since moved past — it's genuinely how NASM and GAS work today.

The directives differ in spelling but not in job — `assembly1-9`'s own "instructions to the assembler, not the CPU" idea still applies exactly:

| JOB | LC-3 (ASSEMBLY1-9) | NASM | GAS |
|---|---|---|---|
| Set the origin address | .ORIG | ORG | (handled by linker script / section placement) |
| Reserve a byte value | .FILL | DB | .byte |
| Reserve a wider value | .FILL (word-sized by default) | DW / DQ | .word / .quad |

| JOB | LC-3 (ASSEMBLY1-9) | NASM | GAS |
|---|---|---|---|
| A null-terminated string | .STRINGZ | DB "text", 0 | .asciz "text" |

## Object Files and Real Linking

Real x86-64 toolchains produce real object files (`.o` on Linux, `.obj` on Windows), and a real linker — `ld` on Linux, `link.exe` on Windows — does exactly the job `assembly1-9` described: resolving cross-file references, combining multiple object files, and (for real programs) linking against system libraries, into one final executable.

What's genuinely new here: a real executable isn't just raw bits sitting at a fixed address the way LC-3's own simple model was. It's a structured, documented file format.

## ELF vs. PE — Real Executable Formats

Linux (and most Unix-like systems) use **ELF** (Executable and Linkable Format); Windows uses **PE** (Portable Executable). Both exist to carry information a flat block of bits never could: where the program's entry point is, which external libraries need to be dynamically linked in — and, directly connecting back to `assembly2-8`'s own paging material, **which memory permissions each section of the program needs**. A file format explicitly marks which parts are code (executable), which are data (writable), and which are read-only constants — and it's precisely this information the OS loader uses to set up the actual per-page protections `assembly2-8` covered. The file format isn't just packaging; it's the literal source of the information ring-3 memory protection is built from.

## A Minimal Real Build Pipeline

Conceptually — this chapter illustrates the shape of the real pipeline rather than serving as a full setup tutorial:

```
# Linux, via NASM + ld
nasm -f elf64 program.asm -o program.o
ld program.o -o program

# Windows, via NASM + a Windows linker
nasm -f win64 program.asm -o program.obj
; (linked with link.exe or an equivalent)
```

**ILLUSTRATIVE, NOT A FULL SETUP TUTORIAL**

This chapter shows the shape of a real build pipeline, matching `cpu8bit1-12`'s and `assembly1-10`'s own scope notes about not walking through full hardware/emulator setup. Real toolchain installation and environment

configuration genuinely varies by system and is deliberately left outside this course's own scope.

| CONCEPT | ASSEMBLY1-9 (LC-3) | THIS CHAPTER (X86-64) |
|---|---|---|
| Two-pass assembly | Yes | Yes — genuinely the same real mechanism |
| Object files | Described conceptually | Real .o/.obj files, produced by real tools |
| Linker | Described conceptually | Real: ld (Linux), link.exe (Windows) |
| Final output | A simple, flat LC-3 executable image | A structured format carrying real protection metadata: ELF or PE |

**EVERYTHING CONVERGES IN THE CAPSTONE**

`assembly2-12` puts this entire chapter's own toolchain to real use — a genuine NASM program, assembled and linked for real, making a real Linux syscall.

## Hands-On Exercises

**EXERCISE 1**

Explain why real assemblers like NASM and GAS still use the exact two-pass approach assembly1-9 taught for LC-3, rather than some more advanced modern technique — what specific problem does the two-pass approach solve that hasn't changed between a teaching ISA and a real modern assembler?

View solution

**EXERCISE 2**

Explain what genuinely new information a structured executable format (ELF or PE) needs to carry that assembly1-9's own simple LC-3 executable model never needed to represent, tying your answer directly to assembly2-8's own paging/protection material.

View solution

**EXERCISE 3**

Using this chapter's own reasoning, explain which assembler — NASM or GAS — a beginner writing x86-64 assembly by hand would likely find more approachable, and why.

View solution

### CHAPTER 11 QUICK REFERENCE

- **NASM** — Intel syntax, this course's own choice, approachable for hand-written assembly
- **GAS** — AT&T syntax, GCC's own internal assembler, unavoidable when reading compiler output or inline asm
- Real assemblers use assembly1-9's exact two-pass algorithm — not a simplified teaching model, genuinely how they work
- Directives differ in spelling (ORG/DB/DW vs. .section/.byte/.word) but do the same job as LC-3's own .ORIG/.FILL
- Real object files (.o/.obj) and real linkers (ld, link.exe) do exactly the job assembly1-9 described conceptually
- **ELF** (Linux) and **PE** (Windows) — structured executable formats carrying the exact per-section permission data assembly2-8's own paging protection is built from
- This chapter is illustrative of the real pipeline's shape, not a full environment-setup tutorial

CHAPTER 12 OF 12

# Capstone: A Real x86-64 Program

## x86-64 Assembly

Chapter 12 · Capstone — A Real x86-64 Program

One real, working NASM program, closing not just this course but the entire three-course Assembly/Machine Language arc: a function call using a genuine calling convention, an array walked with full SIB addressing, and output produced through a real Linux syscall.

### The Program

Sums a small array of five integers using a real function (not inlined code), then prints the single-digit result.

```
section .data
    array    dq 1, 2, 1, 3, 1      ; 5 qwords — kept small so the sum stays a single digit
    outbuf   db 0, 10              ; byte 0: the digit (filled in below); byte 1: newline

section .text
global _start

; --- Function: sum_array ---
; Input:  RDI = array pointer, RSI = element count (System V AMD64 ABI, assembly2-5)
; Output: RAX = sum
sum_array:
        PUSH RBP
        MOV  RBP, RSP        ; a real stable frame pointer (assembly2-5)
        XOR  RAX, RAX        ; RAX = running sum = 0
        XOR  RCX, RCX        ; RCX = index = 0 (caller-saved — safe to clobber freely)
.loop:
        CMP  RCX, RSI
        JE   .done
        ADD  RAX, [RDI + RCX*8]   ; full SIB addressing (assembly2-4): Base + Index*Scale
        INC  RCX
        JMP  .loop
.done:
        POP  RBP
        RET

_start:
        LEA  RDI, [rel array]    ; 1st argument — RIP-relative addressing (assembly2-4)
        MOV  RSI, 5              ; 2nd argument — element count
        CALL sum_array           ; a real function call, System V convention (assembly2-5)

        ADD  AL, '0'             ; digit -> ASCII, assembly1-10's own single-digit trick, echoed here
        MOV  [rel outbuf], AL
```

```
        MOV  RAX, 1            ; syscall number for write() (assembly2-10)
        MOV  RDI, 1             ; fd = stdout
        LEA  RSI, [rel outbuf]
        MOV  RDX, 2             ; length — digit + newline
        SYSCALL

        MOV  RAX, 60            ; syscall number for exit()
        XOR  RDI, RDI          ; exit code 0
        SYSCALL
```

Assembled and linked exactly the way `assembly2-11` described: `nasm -f elf64 program.asm -o program.o`, then `ld program.o -o program`. Running it prints `8` followed by a newline.

## Chapter Attribution — All Three Courses

| CAPSTONE PIECE | CONCEPT | FROM |
|---|---|---|
| The fetch-decode-execute cycle running the whole program | The universal execution model every chip in this arc shares | assembly1-2 |
| outbuf's single-digit ASCII trick (ADD AL, '0') | Converting a small integer into a printable character | assembly1-10 |
| [RDI + RCX*8] — full SIB addressing | Base + Index×Scale, the richest addressing mode this whole arc covered | assembly2-4, previewed by cpu8bit1-4's own zero-page,X and cpu8bit1-7's own (IX+d) |
| PUSH RBP / MOV RBP, RSP / POP RBP | A real stack frame — the modern descendant of assembly1-7's manually-built LC-3 stack and cpu8bit1-3/cpu8bit1-7's own real hardware stacks | assembly2-5 |
| RDI/RSI argument passing, CALL/RET | A real, external calling-convention contract, not a self-invented one | assembly2-5, contrasted with assembly1-7's and cpu8bit1-12's own single informal conventions |
| CMP/JE inside the loop | Condition-code-driven branching, unchanged in spirit since Chapter 6 of assembly1 | assembly1-6, assembly2-3, assembly2-6 |
| SYSCALL, syscall numbers in RAX, arguments in RDI/RSI/RDX | The real, OS-specific modern replacement for assembly1-8's own unified TRAP | assembly2-10 |
| [rel array] / [rel outbuf] | RIP-relative addressing — PC-relative addressing's own conceptual return | assembly2-4, tracing back to assembly1-3's own LC-3 PC-relative LD |

**HONEST SCOPE NOTE**

This capstone deliberately stays within what this course actually taught. Left out, on purpose: any Windows-side build of this same program (assembly2-10's own Windows API path would need an entirely different, non-syscall-based ending); any use of SIMD in the capstone itself (assembly2-9 was deliberately scoped light, and nothing here needed it); any kernel-mode or privileged code (everything here runs entirely in ring 3, per assembly2-8); and no multi-threading of any kind. None of these are gaps in what was taught — they're deliberate boundaries of a single, focused capstone, the same honest-scoping precedent assembly1-10 and cpu8bit1-12 both set before it.

## Closing the Full Arc

Three courses, one throughline, stated first in `assembly1-1`: strip away real-hardware history and teach the universal concepts cleanly (LC-3), then meet two real, historically important 1970s chips shaped by genuinely opposite founding constraints (the 6502 and Z80), then meet the real, modern architecture carrying every one of those forces forward, compounded, for four more decades (x86-64). Registers grew from LC-3's uniform eight, to the 6502's cost-starved three and the Z80's shadow-doubled fourteen, to x86-64's sixteen — each carrying its own real history in its own names. Addressing grew from a single PC-relative formula, to zero page and (IX+d), to a single instruction computing Base+Index×Scale+Displacement. Stacks grew from entirely hand-built, to real-but-page-locked, to real-and-fully-flexible, to a modern frame pointer inside a real function. And the RISC-vs-CISC question `cpu8bit1-1` only previewed got a full, honest answer: confirmed at modern scale, and then honestly complicated by the discovery that a CISC instruction set can run on a genuinely RISC-like microarchitecture underneath. This capstone is the last, concrete proof that every one of those threads was real — not just described, but written, and run.

## Hands-On Exercises

**EXERCISE 1**

Trace sum_array's own loop for the array [1, 2, 1, 3, 1], stating RAX's value after each iteration, and state the final ASCII character written to outbuf.

View solution

**EXERCISE 2**

Using this chapter's own `[RDI + RCX*8]` instruction, identify exactly which register or value plays the role of Base, Index, Scale, and Displacement in assembly2-4's own SIB formula.

View solution

**EXERCISE 3**

Pick three rows from this chapter's own chapter-attribution table and explain, in one or two sentences each, exactly which piece of this capstone's code draws on that chapter's material and why it was needed here.

View solution

### CHAPTER 12 QUICK REFERENCE — COURSE & ARC RECAP

- **assembly2-1 to -3** — the 8086-to-x86-64 lineage, register naming as history made visible, RFLAGS and the two real syntaxes
- **assembly2-4 to -5** — full SIB addressing, RIP-relative addressing's return, real calling conventions as a genuine external contract
- **assembly2-6 to -7** — Jcc/LOOP/CMOV, and the instruction set's real scale confirming (and complicating) cpu8bit1-11's RISC/CISC framing
- **assembly2-8** — genuinely new ground: rings, paging, and page faults
- **assembly2-9 to -11** — SIMD previewed honestly, syscalls vs. the Windows API, real assemblers and linkers
- **assembly2-12** — all of the above, combined into one real, working program
- This closes the full Assembly/Machine Language arc: assembly1 (LC-3) → cpu8bit1 (6502/Z80) → assembly2 (x86-64)