https://simplifycpp.org

DRAFT

Low-Level Programming and Processor Architecture

VOLUME 1
MODERN ASSEMBLY FOUNDATIONS



Prepared by Ayman Alheraki

Low-Level Programming and Processor Architecture Encyclopedia

VOLUME 1 — MODERN ASSEMBLY FOUNDATIONS

Some drafting assistance and idea exploration were supported by modern AI tools, with full supervision, verification, correction, and authorship.

Prepared by Ayman Alheraki

February 2026

Contents

I MACHINE REPRESENTATION **41**

1 Binary, Hex, and Bit-Level Thinking **42**

1.1 Why This Chapter Exists 42

1.2 Bit Operations as Algebra 43

1.2.1 The Three Core Operators: AND, OR, XOR 43

1.2.2 Identity and Annihilator Elements 43

1.2.3 Involution and Reversibility 44

1.2.4 NOT and De Morgan Laws 44

1.2.5 The Practical "Bitset" View 45

1.2.6 Extensive Rust Example (Windows-Friendly) 45

1.3 Masking, Shifting, Rotation 47

1.3.1 Mask Design: The Canonical Patterns 48

1.3.2 Extensive Rust Example: Extract and Insert Bitfields 48

1.3.3 Shifts: Logical vs Arithmetic 49

1.3.4 Rotation: When Shifts Lose Information 50

1.3.5 GAS Assembly (Intel Syntax): Mask, Shift, Rotate 50

1.3.6 Rust Calling These Routines (Windows-Compatible Style) 52

1.4 Common Signed/Unsigned Traps 52
1.4.1 Trap 1: Comparing Signed Values as Unsigned (and Vice Versa) 53
1.4.2 Trap 2: Right Shift on Signed Values 54
1.4.3 Trap 3: Mixed-Width Operations and Implicit Extension 54
1.4.4 Trap 4: Shifting by the Word Size or More 55
1.4.5 Trap 5: Using Signed Types for Bitfields 56
1.5 Hex Thinking: The Engineer's Compression Format 56
1.5.1 Nibble-to-Bit Mapping 57
1.5.2 Hex Dump Discipline 57
1.5.3 Rust Example: Endian-Explicit Decoding 57
1.6 Practical Exercises (Foundation Level) 58
1.7 Professional Summary 58

2 Two's Complement and Integer Behavior 60
2.1 Scope and Engineering Goal 60
2.2 Overflow 61
2.2.1 Two Kinds of Overflow Engineers Must Separate 61
2.2.2 Unsigned Addition Overflow 61
2.2.3 Signed Addition Overflow 62
2.2.4 Subtraction Overflow and Borrow 64
2.2.5 Boundary-Value Discipline 64
2.3 Sign Extension 65
2.3.1 What Sign Extension Means 65
2.3.2 Zero Extension vs Sign Extension 65
2.3.3 x86-64 Sign Extension Instructions (Foundation Set) 65
2.3.4 GAS Assembly (Intel Syntax): Zero vs Sign Extension Demonstration . 66
2.3.5 Rust Example: Explicit Widening Correctness 67
2.3.6 Why Sign Extension Appears Everywhere in Assembly 67

2.4 Flag Interaction 68
2.4.1 x86 Status Flags (Foundation Set) 68
2.4.2 Flag Semantics After ADD and SUB 68
2.4.3 GAS Assembly (Intel Syntax): Capturing CF and OF 68
2.4.4 Rust Example: Comparing CPU Flags vs Language Overflow APIs . . 70
2.4.5 Flag Pitfalls That Break Real Programs 71
2.4.6 ARM and RISC-V Flag Model (Foundation Comparison) 73
2.5 Practical Exercises (Boundary and Flag Correctness) 74
2.6 Professional Summary 74

3 Floating-Point Basics for Assembly Authors 76
3.1 Scope and Engineering Goal 76
3.2 IEEE-754 Essentials 77
3.2.1 The Two Workhorse Formats: Binary32 and Binary64 77
3.2.2 Special Encodings and Their Meaning 78
3.2.3 Rounding: Why the Last Bit Often Lies 79
3.2.4 Exceptions and Status Flags (Conceptual) 79
3.2.5 Rust (Windows-Friendly) Example: Bit-Exact Inspection and Classification 80
3.3 ABI Implications 81
3.3.1 Core Principle: Do Not Guess the ABI 82
3.3.2 Common 64-bit ABIs (Foundations View) 82
3.3.3 The Precision Trap: x87 vs SSE vs Strictly-64-bit Semantics 84
3.4 Passing Floats Between Languages 84
3.4.1 FFI Rule Set for Floating Values 84
3.4.2 Minimal ABI-Safe Patterns 85
3.4.3 Pattern B: Using Float Bit Patterns Across the Boundary 86
3.4.4 Pattern C: Comparing Floats and Handling NaNs Correctly 88
3.4.5 ABI Pitfall Catalogue for Floating Boundaries 89

3.5 Practical Exercises (Foundation Level) . . . 91
3.6 Professional Summary . . . 91

4 Endianness and Data Interpretation 93
4.1 Scope and Engineering Goal . . . 93
4.2 Little vs Big . . . 94
4.2.1 Definition . . . 94
4.2.2 The Engineer's Rule: Addresses Increase, Significance May Not . . . 94
4.2.3 A Simple Table You Must Memorize . . . 95
4.2.4 Rust (Windows-Friendly) Example: Endianness by Construction . . . 95
4.2.5 Assembly Reality: Byte Order Appears in Loads and Stores . . . 96
4.2.6 GAS Assembly (Intel Syntax): Showing the Byte Layout . . . 96
4.3 Network Order . . . 98
4.3.1 What "Network Byte Order" Means . . . 98
4.3.2 Conversion Discipline: Host $\leftrightarrow$ Network . . . 98
4.3.3 Rust Implementation: Network Order Encode/Decode . . . 99
4.3.4 Assembly View: Byte Swap as a Primitive . . . 100
4.3.5 Rust Calling Assembly Conversions (Windows-Friendly) . . . 101
4.4 Cross-Platform Data Pitfalls . . . 102
4.4.1 Pitfall 1: Reinterpreting Struct Memory as a Wire Format . . . 102
4.4.2 Pitfall 2: Confusing "Byte Order" with "Bit Order" . . . 103
4.4.3 Pitfall 3: Mixed-Endian File Formats . . . 103
4.4.4 Pitfall 4: Using Native Types in On-Disk Formats . . . 103
4.4.5 Pitfall 5: Casting Bytes to Integers with Unsafe Reinterpretation . . . 104
4.4.6 Rust Example: A Safe, Extensible Reader for BE/LE . . . 105
4.4.7 Pitfall 6: Endianness and Floating-Point Reinterpretation . . . 107
4.4.8 Rust Example: Encoding f32/f64 as Network-Order Bytes . . . 108
4.5 Practical Exercises (Foundation Level) . . . 109

4.6 Professional Summary 110

5 Alignment and Data Layout **111**

5.1 Scope and Engineering Goal 111
5.2 Natural Alignment 112
5.2.1 Definition 112
5.2.2 What Determines Alignment 112
5.2.3 Typical Alignment Expectations (64-bit Systems) 112
5.2.4 Rust (Windows-Friendly) Example: Observing Size and Alignment . . 113
5.2.5 Alignment in Assembly Terms 114
5.3 Structure Padding 115
5.3.1 Why Padding Exists 115
5.3.2 Hand-Computing Layout: A Canonical Example 115
5.3.3 Field Ordering Rule: Reduce Padding by Reordering 116
5.3.4 Rust Layout Contracts: `repr(C)` vs Default 117
5.3.5 Packed Structures: When "Fixing Size" Breaks Correctness 117
5.3.6 Rust Example: `repr(C)` vs `repr(packed)` and Safe Unaligned Reads . 118
5.4 Misaligned Penalties 119
5.4.1 What "Misaligned" Means 119
5.4.2 The Three Possible Outcomes Across Architectures 119
5.4.3 x86-64 Reality (Practical) 119
5.4.4 AArch64 and RISC-V Reality (Practical) 120
5.4.5 SIMD Alignment Fault Demonstration (x86-64) 120
5.4.6 Scalar Unaligned Loads: Correctness vs Performance 123
5.4.7 Rust Example: `read_unaligned` for Portable Correctness 123
5.4.8 ABI Boundary Warning: Alignment is Part of the Contract 124
5.5 Professional Checklist 125
5.6 Exercises (Foundation Level) 125

5.7 Professional Summary 126

II INSTRUCTION SET FOUNDATIONS 127

6 Instruction Encoding Concepts 128

6.1 Variable vs Fixed Length 128
6.1.1 The Two Families 128
6.1.2 Why This Choice Matters 129
6.1.3 The Practical Tradeoff 129
6.2 x86 vs ARM vs RISC-V Encoding Philosophy 130
6.2.1 x86-64 Philosophy: Backward Compatibility and Density Through Prefix Layers 130
6.2.2 AArch64 (ARM64) Philosophy: Fixed 32-bit Words, Regular Fields . . 133
6.2.3 RISC-V Philosophy: Simple Base 32-bit Encoding With Optional Compression 134
6.3 Encoding Examples and Field Extraction 135
6.3.1 RISC-V: R-Type and I-Type With Real Machine Words 135
6.3.2 AArch64: Fixed 32-bit Words and Byte Order 136
6.3.3 x86-64: Variable-Length Components and Practical Decoding Mindset 137
6.4 Rust (Windows-Friendly) Laboratory: Decoding Real Encodings 137
6.4.1 Rust: Helper Functions for Bitfield Extraction 137
6.4.2 Rust: Decode RISC-V R-Type and I-Type 138
6.4.3 Rust: Show AArch64 Word and Its Bytes 139
6.5 Philosophy Summary: What the Encodings Optimize For 140
6.5.1 x86-64 140
6.5.2 AArch64 140
6.5.3 RISC-V 140

6.6 Exercises (Foundation Level) . . . 141
6.7 Professional Summary . . . 141

7 Register Architectures Compared 143
7.1 Scope and Engineering Goal . . . 143
7.2 Concepts: Architectural vs ABI Roles . . . 144
7.2.1 Architectural Registers . . . 144
7.2.2 ABI Roles: Volatile vs Non-Volatile . . . 144
7.3 x86-64 Registers . . . 144
7.3.1 General-Purpose Integer Registers (GPRs) . . . 145
7.3.2 Instruction Pointer and Flags . . . 145
7.3.3 Vector/SIMD Registers . . . 146
7.3.4 x86-64 ABI Volatile vs Non-Volatile (Two Dominant 64-bit ABIs) . . . 146
7.3.5 x86-64 Windows Example: Callee-Saved Discipline in Assembly . . . 147
7.3.6 Rust (Windows) Calling the Assembly Function . . . 148
7.4 AArch64 Registers . . . 148
7.4.1 Integer Registers . . . 149
7.4.2 SIMD/Floating Registers . . . 149
7.4.3 Condition Flags . . . 149
7.4.4 AArch64 ABI Volatile vs Non-Volatile (AAPCS64 Foundation Summary) 149
7.4.5 AArch64 Example: Saving Callee-Saved Registers . . . 150
7.5 RV64 Registers (RISC-V 64-bit) . . . 151
7.5.1 Integer Registers: x0–x31 . . . 151
7.5.2 ABI Register Names (Must-Know Table) . . . 151
7.5.3 Floating-Point Registers (If F/D Extensions Are Present) . . . 152
7.5.4 RV64 ABI Volatile vs Non-Volatile (Foundation Summary) . . . 152
7.5.5 RV64 Example: Using s0 as Frame Pointer and Preserving It . . . 153
7.6 Volatile vs Non-Volatile: Practical Rules That Prevent Real Bugs . . . 153

7.6.1 Rule 1: Treat Calls as Clobbering All Volatile Registers 153
7.6.2 Rule 2: Only Save Non-Volatiles If You Actually Use Them 154
7.6.3 Rule 3: ABI Defines Register Preservation, Not the ISA 154
7.7 Extended x86-64 Example: Caller-Saved vs Callee-Saved in One Scenario . . . 155
7.8 Professional Checklist 156
7.9 Exercises (Foundation Level) 157
7.10 Professional Summary 157

8 Addressing Modes and Effective Address Calculation 159
8.1 Scope and Engineering Goal 159
8.2 Memory Operand Formats 160
8.2.1 Core Concept: "Memory Is Bytes, Addressing Selects a Byte Location" 160
8.2.2 x86-64 Memory Operand Format (GAS Intel Syntax) 160
8.2.3 AArch64 Memory Operand Formats 161
8.2.4 RV64 Memory Operand Formats 162
8.3 Address Computation Rules 163
8.3.1 Effective Address vs Loaded Value 163
8.3.2 x86-64: Formal EA Model for the Common Addressing Form 163
8.3.3 AArch64: Structured Addressing Rules 166
8.3.4 RV64: Explicit Address Arithmetic Philosophy 167
8.4 Cross-Architecture Comparison: What You Must Internalize 167
8.4.1 Addressing Complexity Placement 167
8.4.2 Element Indexing Example: `base + i*sizeof(T)` 168
8.5 Cross-Platform Data Pitfalls (Addressing Edition) 168
8.5.1 Pitfall 1: Mistaking Index Units 168
8.5.2 Pitfall 2: Sign Extension of Displacements and Immediates 168
8.5.3 Pitfall 3: Alignment Assumptions Hidden in Addressing 169
8.5.4 Pitfall 4: Pointer Size and Integer Truncation 169

8.6 Rust (Windows-Friendly) Laboratory: Verifying EA Computation Against High-Level Math . . . 169
8.7 Practical Patterns for Assembly Authors . . . 171
8.7.1 Pattern 1: Struct Field Address and Load . . . 171
8.7.2 Pattern 2: Bounds-Safe Indexing Discipline . . . 171
8.7.3 Pattern 3: Use Address Computation Instructions Intentionally . . . 171
8.8 Exercises (Foundation Level) . . . 172
8.9 Professional Summary . . . 173

9 Arithmetic and Logic Operations 174
9.1 Flags Behavior . . . 174
9.1.1 Flags Are Side Effects of Arithmetic . . . 174
9.1.2 x86-64 Flags (RFLAGS) That Matter for Comparisons . . . 174
9.1.3 How x86-64 Sets Key Flags . . . 175
9.1.4 AArch64 Flags (NZCV) and Flag-Setting Instructions . . . 176
9.1.5 RV64 Compare Model: Flags-Free, Branch-on-Compare Style . . . 176
9.2 Compare Patterns . . . 177
9.2.1 Compare Is Often Just Subtraction . . . 177
9.2.2 x86-64 Comparison Conditions You Must Know . . . 177
9.2.3 AArch64 Compare Patterns . . . 178
9.2.4 RV64 Compare Patterns . . . 179
9.3 Conditional Execution . . . 179
9.3.1 x86-64 Conditional Branches and Conditional Moves . . . 179
9.3.2 AArch64 Conditional Branch and Conditional Select . . . 180
9.3.3 RV64 Conditional Control . . . 180
9.4 x86-64 (Windows) Extensive Examples With Rust Verification . . . 181
9.4.1 Example 1: Return Flags After Addition (Carry and Overflow) . . . 181
9.4.2 Example 2: Unsigned Compare Pattern (a < b) Without Branch . . . 184

9.4.3 Example 3: Signed Compare Pattern (a < b) Without Branch 185
9.4.4 Example 4: Conditional Select (min) With Branchless CMOV 186
9.4.5 Example 5: Conditional Execution by Branch (abs) With Correct Signed Handling 187
9.5 Cross-Architecture Pattern Cookbook 188
9.5.1 Pattern: Compare for Equality 188
9.5.2 Pattern: Signed Less-Than Branch 189
9.5.3 Pattern: Unsigned Less-Than Branch 189
9.5.4 Pattern: Branchless Select (min/max) Concept 190
9.6 Professional Checklist 190
9.7 Exercises (Foundation Level) 191
9.8 Professional Summary 192

10 Control Flow and Branching 193
10.1 Scope and Engineering Goal 193
10.2 Conditional Jumps 194
10.2.1 Control Flow as “PC Update Rules” 194
10.2.2 x86-64: Condition Codes Are Evaluations of Flags 194
10.2.3 AArch64: Compare Sets NZCV, Branch Tests NZCV 195
10.2.4 RV64: Branches Compare Registers Directly 196
10.3 Loops 196
10.3.1 A Loop is a State Update + Conditional Branch 196
10.3.2 x86-64: Do Not Use the `loop` Instruction in Modern Code 197
10.3.3 Loop Pattern 1: Counted Sum of u64 Array (x86-64 Windows) 197
10.3.4 Loop Pattern 2: Find First Match (x86-64 Windows) 199
10.3.5 AArch64 and RV64: Equivalent Loop Intents 200
10.4 Call/Ret Model 202
10.4.1 The Universal Function Call Contract 202

10.4.2 x86-64: Call/Ret Mechanism 202
10.4.3 AArch64: Link Register Call/Return Model 203
10.4.4 RV64: Link Register Call/Return Model 203
10.5 x86-64 Windows: Extensive Call/Ret Examples 204
10.5.1 Example 1: Leaf Function (No Calls) With Minimal Stack Usage . . . 204
10.5.2 Example 2: Non-Leaf Function (Calls Another) With Volatile Preservation 205
10.5.3 Example 3: Building a Simple Loop With a Call Inside 206
10.6 Conditional Jumps: Pattern Cookbook 208
10.6.1 If/Else (x86-64) 208
10.6.2 While Loop (x86-64) 209
10.6.3 Do-While Loop (x86-64) 209
10.6.4 Switch-Like Jump Table (Conceptual) 209
10.7 Professional Checklist 210
10.8 Exercises (Foundation Level) 211
10.9 Professional Summary 211

III ABI and Calling Conventions 212

11 What Is an ABI? 213
11.1 API vs ABI 213
11.1.1 What an ABI includes (practical checklist) 213
11.1.2 Why an assembler engineer must care 214
11.2 Binary compatibility 215
11.2.1 Binary compatibility vs source compatibility 215
11.2.2 The three most common causes of ABI breaks 215
11.2.3 Example 1: same function signature, broken calling convention (Windows x64) 215

11.2.4 Example 2: same API type, broken layout 216
11.2.5 Binary compatibility design patterns (what actually works) 216
11.3 Why ABIs matter more than syntax 217
11.3.1 A mental model 218
11.3.2 Where ABI dominates engineering reality 218
11.3.3 Windows reality: one platform, multiple ABIs 218
11.4 Windows x64 ABI essentials (the rules you must not violate) 219
11.4.1 Argument passing (Win64) 219
11.4.2 Shadow space 219
11.4.3 Stack alignment 220
11.4.4 Volatile vs non-volatile registers (Win64) 220
11.5 Extensive Windows examples (Rust + C + GAS Intel syntax) 220
11.5.1 Example A: A Win64 ABI assembly function called from Rust 220
11.5.2 Example B: Exporting a stable C ABI from Rust as a Windows DLL . . 222
11.5.3 Example C: The classic ABI pitfall (struct layout across boundary) . . . 226
11.6 ABI discipline rules for a professional toolchain 227
11.7 Summary 228

12 Stack Frames and Prolog/Epilog Patterns 229
12.1 Stack Growth 229
12.1.1 What “stack growth” means in real machines 229
12.1.2 x86 / x86-64: downward-growing stack 230
12.1.3 Why stack direction matters to low-level engineers 230
12.2 Frame Pointer Usage 231
12.2.1 The two “pointers” in a stack frame 231
12.2.2 What a frame pointer gives you 231
12.2.3 Frame-pointer prolog/epilog: classic pattern 232
12.2.4 Frame-pointer omission (FPO) and “RSP-relative frames” 232

12.2.5 Windows x64 nuance: unwinding and prolog constraints 233
12.3 Red Zone vs Shadow Space 233
12.3.1 Red zone (System V AMD64 ABI concept) 233
12.3.2 Shadow space (Windows x64 ABI concept) 234
12.3.3 Concrete comparison (what you can touch without moving RSP) 234
12.3.4 Why this matters for Rust on Windows 235
12.4 Prolog/Epilog Patterns on Windows x64 (practical and correct) 235
12.4.1 Pattern 1: leaf function, no locals, no calls 235
12.4.2 Pattern 2: non-leaf function (makes a call) with shadow space and alignment 236
12.4.3 Pattern 3: frame pointer based (debug-friendly) function with locals . . 237
12.4.4 Pattern 4: saving non-volatile registers 238
12.5 Rust on Windows: calling these patterns safely 239
12.5.1 Rust declarations (Windows x64) 239
12.5.2 Rust + global_asm! (Windows-friendly packaging) 240
12.6 Common mistakes and how to avoid them 241
12.6.1 Mistake 1: treating Windows like SysV (red zone assumption) 241
12.6.2 Mistake 2: forgetting shadow space before a call 241
12.6.3 Mistake 3: clobbering non-volatile registers 241
12.6.4 Mistake 4: breaking stack alignment 242
12.7 Summary 242

13 System V AMD64 ABI **243**
13.1 Overview and Scope 243
13.2 Parameter Passing 244
13.2.1 Register classes and the two parallel channels 244
13.2.2 Integer and pointer parameters (GPR order) 244
13.2.3 Floating-point parameters (XMM order) 244
13.2.4 Mixed argument lists (two-channel allocation) 245

13.2.5 Stack-passed arguments (when registers overflow) 246
13.2.6 Varargs (C-style . . .) and the AL rule 246
13.3 Return Values 246
13.3.1 Integer and pointer returns 246
13.3.2 Floating-point returns 247
13.3.3 Small struct returns (in registers) vs large struct returns (sret) 247
13.4 Stack Alignment Rules 247
13.4.1 The 16-byte alignment contract 247
13.4.2 Red zone (SysV) versus Windows shadow space 248
13.4.3 Canonical SysV leaf function using red zone 248
13.4.4 Non-leaf SysV function: align stack before call 248
13.5 Struct Passing 249
13.5.1 The SysV “classification” idea (eightbytes) 249
13.5.2 Example 1: struct of two u64 passed in registers 250
13.5.3 Example 2: struct with two doubles passed in XMM regs 251
13.5.4 Example 3: small mixed struct passed across channels 251
13.5.5 Example 4: large struct passed by memory (stack) 252
13.6 Extensive Rust Examples (written on Windows, targeting SysV) 254
13.6.1 Rust ABI declarations 254
13.6.2 Rust #[repr(C)] structs for SysV interop 254
13.6.3 Assembly implementations matching SysV (GAS Intel syntax) 255
13.6.4 Cross-compilation notes (Windows host) 256
13.7 Practical ABI Summary for SysV AMD64 256
13.7.1 Parameter passing 256
13.7.2 Return values 257
13.7.3 Stack 257
13.7.4 Struct passing 257

14 Microsoft x64 ABI **258**

14.1 Purpose and Scope . . . 258
14.2 Shadow Space . . . 258
14.2.1 Definition . . . 258
14.2.2 Why shadow space exists (engineering rationale) . . . 259
14.2.3 Who allocates and who frees . . . 259
14.2.4 Shadow space and stack arguments . . . 260
14.2.5 Minimal correct call sequence from assembly (Win64) . . . 260
14.2.6 Leaf functions and shadow space . . . 261
14.3 Register Differences . . . 261
14.3.1 Argument registers: Windows x64 vs System V AMD64 . . . 261
14.3.2 Caller-saved vs callee-saved (non-volatile) register sets . . . 262
14.3.3 Stack alignment expectations . . . 263
14.4 Interop Dangers . . . 263
14.4.1 Danger 1: Shadow space vs red zone confusion . . . 263
14.4.2 Danger 2: Wrong ABI name in Rust (`extern "C"` vs `extern "system"`) 263
14.4.3 Danger 3: Non-volatile XMM registers on Windows . . . 264
14.4.4 Danger 4: Stack alignment violations that crash far away . . . 266
14.4.5 Danger 5: Mixing object formats or unwinding expectations . . . 266
14.5 Extensive Windows Examples (Rust + GAS Intel Syntax) . . . 267
14.5.1 Example A: Rust calling a Win64 ABI assembly function . . . 267
14.5.2 Example B: Assembly calls a C function (must provide shadow space) . 268
14.5.3 Example C: Exporting a stable ABI from Rust for other languages . . . 269
14.6 Practical Summary . . . 270
14.6.1 Shadow space . . . 270
14.6.2 Register differences . . . 270
14.6.3 Interop dangers . . . 271

15 AArch64 Procedure Call Standard **272**
15.1 Scope and Naming . . . 272
15.2 Register Mapping . . . 273
15.2.1 General-purpose registers and their roles . . . 273
15.2.2 SIMD and floating-point registers . . . 274
15.2.3 Argument mapping summary tables . . . 274
15.2.4 Register preservation (what must survive a call) . . . 275
15.3 Stack Alignment . . . 275
15.3.1 The non-negotiable rule: 16-byte alignment . . . 275
15.3.2 Canonical prolog/epilog patterns . . . 276
15.3.3 Allocating local space correctly . . . 277
15.3.4 Why alignment matters (real failure modes) . . . 277
15.4 Float Handling . . . 278
15.4.1 Scalar float and double passing/returns . . . 278
15.4.2 Mixed integer and float arguments . . . 278
15.4.3 Homogeneous Floating-point Aggregates (HFA idea) . . . 279
15.5 Struct Passing and Returning (practical rules you can rely on) . . . 280
15.5.1 Small structs: often split across registers . . . 280
15.5.2 Larger structs: passed by memory . . . 281
15.5.3 Returning large structs: hidden sret pointer in `x8` . . . 281
15.6 Rust Examples Written on Windows (Cross-Targeting AAPCS64) . . . 282
15.6.1 Rust declarations and `#[repr(C)]` . . . 282
15.6.2 Build mindset (host vs target) . . . 284
15.7 Summary . . . 284

16 RISC-V Calling Convention **285**
16.1 Scope and Terminology . . . 285
16.2 a*, t*, s* Registers . . . 286

16.2.1 Register naming overview 286
16.2.2 Caller-saved vs callee-saved: the real contract 286
16.2.3 Floating-point register families (when hard-float ABI is used) 287
16.3 Return Rules 287
16.3.1 Integer and pointer returns 287
16.3.2 Floating-point returns (hard-float ABIs) 288
16.3.3 Returning aggregates (struct/union) 288
16.3.4 Example: returning a 128-bit value on RV64 (two-word return) 288
16.4 Stack Discipline 289
16.4.1 Stack growth direction and alignment 289
16.4.2 What must be true at a call boundary 289
16.4.3 Frame pointer usage 289
16.4.4 Canonical prolog/epilog patterns 290
16.4.5 Passing more than 8 integer arguments (stack arguments) 291
16.5 Floating-Point Handling (Soft-Float vs Hard-Float) 293
16.5.1 Hard-float register passing 293
16.5.2 Soft-float behavior 294
16.6 Extensive Rust Examples (Written on Windows, Targeting RISC-V) 294
16.6.1 Rust FFI declarations using the C ABI 294
16.6.2 Rust + embedded assembly via `global_asm!` (portable for teaching) . . 295
16.7 Interop-Safe Discipline Checklist (RISC-V) 296
16.8 Summary 297

17 Variadic Functions and Edge Cases **298**
17.1 Why Variadics Are an ABI Stress-Test 298
17.2 Hidden Parameters and Hidden State 299
17.2.1 Hidden parameters are not “mystery magic” 299
17.2.2 Hidden return pointer (sret) for large aggregates 299

17.2.3 Hidden state in SysV AMD64 varargs: the AL rule 300
17.2.4 Hidden state in Windows x64 varargs: FP duplication into integer regs . 301
17.2.5 Other common hidden parameters you must recognize 303
17.3 Default Promotions 304
17.3.1 C default argument promotions (the law of variadics) 304
17.3.2 Consequences for ABI and assembly 304
17.3.3 C example: promotions in practice 304
17.3.4 Rust perspective on default promotions 305
17.4 ABI Corner Cases 306
17.4.1 Corner case 1: SysV vs Win64 varargs are fundamentally different . . . 306
17.4.2 Corner case 2: va_list is ABI-defined and not portable 307
17.4.3 Corner case 3: passing structs through ... is a trap 307
17.4.4 Corner case 4: long double is ABI-sensitive 308
17.4.5 Corner case 5: alignment and "over-aligned" types 308
17.4.6 Corner case 6: Windows x64 shadow space interactions 309
17.4.7 Corner case 7: mixed calling conventions in one process 309
17.5 Extensive Worked Examples 310
17.5.1 Example A: Implementing a tiny variadic logger in C (portable structure, ABI-managed internals) 310
17.5.2 Example B: Win64 assembly calling a C variadic function correctly . . 310
17.5.3 Example C: Rust on Windows calling printf with correct promotions . 311
17.6 Engineering Rules You Can Enforce in Your Toolchain 312
17.7 Summary 313

18 Unwinding and Exception Boundaries (Foundational Only) 314
18.1 Stack Traces 314
18.1.1 What a "stack trace" really is 314
18.1.2 Why unwinding is harder than "follow the frame pointer" 315

18.1.3 Two high-level strategies . . . 315
18.2 Metadata Overview . . . 316
18.2.1 What unwind metadata must describe . . . 316
18.2.2 Windows x64 model: table-driven unwinding . . . 316
18.2.3 ELF/SysV model: DWARF CFI and `.eh_frame` . . . 317
18.3 Safe Boundaries . . . 317
18.3.1 The number one rule: unwinding must match the declared ABI boundary 317
18.3.2 Rust on Windows: the safe default boundary . . . 318
18.3.3 When you intentionally want unwinding across FFI . . . 319
18.3.4 C++ exceptions across boundaries on Windows . . . 320
18.4 Stack Discipline and Unwinding Implications . . . 320
18.4.1 What unwinders assume about well-formed frames . . . 320
18.4.2 Windows x64: strict epilog/prolog patterns for reliable unwinding . . . 321
18.5 Extensive Examples (Rust for Windows OS, plus Intel-syntax x86-64 assembly) 321
18.5.1 Example A: leaf assembly function (typically unwind-trivial) . . . 321
18.5.2 Example B: non-leaf assembly that calls a function (unwind-sensitive) . 322
18.5.3 Example C: Rust boundary wrapper that prevents unwinding into foreign callers . . . 323
18.5.4 Example D: “panic=abort” as an explicit boundary policy . . . 324
18.6 ABI Corner Cases at Unwind Boundaries . . . 324
18.6.1 Corner case 1: crossing runtimes . . . 324
18.6.2 Corner case 2: missing or incorrect metadata in hand-written assembly 325
18.6.3 Corner case 3: non-volatile SIMD registers on Windows . . . 325
18.7 Safe Boundaries Checklist (Practical Rules) . . . 325
18.8 Summary . . . 326

IV Memory Model and Atomics 327

19 Why Memory Ordering Exists 328

19.1 Foundational Goal . . . 328
19.2 Reordering Sources . . . 328
19.2.1 Reordering comes from three independent layers . . . 328
19.2.2 Compiler reordering . . . 329
19.2.3 CPU reordering . . . 330
19.2.4 x86 vs weaker architectures . . . 331
19.2.5 Memory subsystem reordering . . . 332
19.3 Data Race Reality . . . 332
19.3.1 What is a data race? . . . 332
19.3.2 The dangerous misconception . . . 333
19.3.3 Classic broken spin loop . . . 333
19.4 Why Memory Ordering Exists . . . 334
19.5 Acquire and Release (Foundational View) . . . 334
19.5.1 Release: publishing data . . . 334
19.5.2 Acquire: consuming data . . . 335
19.5.3 Correct publication pattern in Rust (Windows-compatible) . . . 335
19.6 Assembly Perspective (x86-64 Windows) . . . 336
19.6.1 x86-64 implicit ordering . . . 336
19.6.2 Why locked instructions matter . . . 337
19.7 Data Race Reality in Assembly . . . 337
19.7.1 Broken lock implementation . . . 337
19.8 Practical Guidelines . . . 338
19.9 Foundational Summary . . . 338

20 C11/C++20 Memory Model (Assembler Perspective) 339
20.1 Purpose of This Chapter . . . 339
20.2 Happens-before . . . 340
20.2.1 The one concept that replaces “guessing” . . . 340
20.2.2 Three building blocks (practical form) . . . 340
20.2.3 Operational interpretation for assembly authors . . . 341
20.2.4 Worked pattern: publish data with release/acquire (portable mental model) 342
20.3 Atomic Categories . . . 344
20.3.1 Category 1: Atomic loads and stores . . . 344
20.3.2 Category 2: Read-Modify-Write (RMW) operations . . . 345
20.3.3 Category 3: Compare-and-exchange (CAS) . . . 347
20.3.4 Category 4: Fences (barriers) . . . 348
20.4 Assembler-Centric Summary of Ordering Strengths . . . 349
20.4.1 Relaxed . . . 349
20.4.2 Acquire . . . 349
20.4.3 Release . . . 350
20.4.4 Acq_Rel . . . 350
20.4.5 Seq_Cst . . . 350
20.5 Practical FFI Safety (Rust on Windows) . . . 350
20.5.1 Rule: do not mix atomic and non-atomic access . . . 350
20.5.2 Rule: prefer high-level sync unless you are building primitives . . . 351
20.6 Extensive Example: A Minimal Spin Lock (Assembler Perspective) . . . 351
20.6.1 Rust spin lock using atomic exchange (Windows-ready) . . . 351
20.6.2 x86-64 implementation intuition . . . 352
20.6.3 Win64 external assembly for `swap` primitive (educational) . . . 353
20.7 Summary . . . 353

21 x86 Memory Ordering Model **355**

21.1 Scope and Practical Goal 355

21.2 Strong Ordering (What x86 Actually Guarantees) 355

21.2.1 The mental model: TSO (Total Store Order) 355

21.2.2 What reorderings are forbidden (foundational form) 356

21.2.3 The one reordering you must not ignore: Store → Load 356

21.2.4 Atomicity vs ordering 357

21.2.5 Why compilers still matter on x86 358

21.3 The LOCK Prefix (The Hardware Ordering Workhorse) 358

21.3.1 What `LOCK` does in practical terms 358

21.3.2 Which instructions commonly use `LOCK` 359

21.3.3 Atomic exchange is the simplest lock primitive 359

21.3.4 Atomic compare-exchange is the foundation of lock-free algorithms . . 360

21.3.5 A locked add for atomic counters 360

21.3.6 Where `mfence` fits (and why locked ops often replace it) 361

21.4 Strong Ordering in Practice (Rust on Windows) 361

21.4.1 Why Rust orderings often compile to "plain" instructions on x86 361

21.4.2 Worked example: release/acquire flag 362

21.4.3 Spin lock example that relies on `xchg` 363

21.5 Common Pitfalls (Assembler View) 364

21.5.1 Pitfall 1: confusing atomicity with ordering 364

21.5.2 Pitfall 2: missing `LOCK` on RMW instructions 364

21.5.3 Pitfall 3: mixing atomic and non-atomic access to the same variable . . 365

21.5.4 Pitfall 4: writing "portable" assembly without acknowledging ISA differences 365

21.6 Summary 365

22 ARM Acquire/Release Model **366**
22.1 Practical Goal . . . 366
22.2 Acquire/Release in ARM Terms (Foundational) . . . 367
22.2.1 The engineering meaning . . . 367
22.2.2 Why AArch64 needs this more than x86 . . . 367
22.3 LDAXR/STXR . . . 368
22.3.1 Exclusive access instructions in one sentence . . . 368
22.3.2 Why the `A` and `L` variants matter . . . 368
22.3.3 Example 1: atomic increment with `LDAXR/STLXR` . . . 369
22.3.4 Example 2: CAS (compare-and-swap) with `LDAXR/STLXR` . . . 369
22.3.5 Example 3: atomic swap (exchange) with `LDAXR/STLXR` . . . 370
22.4 DMB/DSB Basics . . . 371
22.4.1 Barrier instructions in practical terms . . . 371
22.4.2 Barrier scopes and domains (keep it simple) . . . 372
22.4.3 When acquire/release is not enough . . . 372
22.4.4 Example: explicit full barrier around a critical point . . . 373
22.4.5 DSB example: stronger completion guarantee . . . 373
22.5 Mapping to Language Atomics (Rust Perspective) . . . 374
22.5.1 Acquire/Release in Rust . . . 374
22.5.2 Rust example: publish/consume . . . 374
22.6 Extensive Worked Example: A Spin Lock Using `LDAXR/STLXR` . . . 376
22.6.1 AArch64 assembly lock acquire/release . . . 376
22.6.2 Rust wrapper interface (written on Windows, cross-targeting AArch64) 377
22.7 Summary . . . 378

23 RISC-V AMOs and `FENCE` **379**
23.1 Practical Goal . . . 379
23.2 aq/rl Bits . . . 380

23.2.1 What aq and rl are . . . 380
23.2.2 Foundational operational meaning . . . 380
23.2.3 What aq/rl do *not* guarantee . . . 381
23.2.4 Example: publish/consume with aq/rl (flag publication) . . . 382
23.3 LR/SC Loops . . . 383
23.3.1 Why LR/SC exists . . . 383
23.3.2 The reservation reality . . . 384
23.3.3 Example 1: CAS with `lr.d.aq` / `sc.d.rl` . . . 384
23.3.4 Example 2: fetch-add implemented with LR/SC (RV64) . . . 385
23.3.5 Example 3: spin lock using LR/SC . . . 386
23.4 `FENCE` (Foundational) . . . 387
23.4.1 What `FENCE` does . . . 387
23.4.2 Common simple form . . . 387
23.4.3 When you typically need fences . . . 387
23.4.4 Minimal illustrative publish/consume using fences + plain ops . . . 388
23.5 Rust Atomic Orderings (Windows Host, RISC-V Target) . . . 389
23.5.1 Rust code that relies on Acquire/Release . . . 389
23.6 Practical Checklist . . . 390
23.7 Summary . . . 390

24 Correct Locking Patterns **392**
24.1 Purpose and Boundary of This Chapter . . . 392
24.2 Minimal Spinlock . . . 393
24.2.1 What a minimal lock must guarantee . . . 393
24.2.2 Minimal spinlock in Rust (Windows-ready) . . . 393
24.2.3 Minimal x86-64 Win64 assembly intuition (LOCK / implicit locked `xchg`) 395
24.3 Retry Loops . . . 396
24.3.1 Why retry loops exist . . . 396

24.3.2 Retry loop failure is normal . . . 396
24.3.3 AArch64 minimal lock with retry loop (LDAXR/STLXR) . . . 397
24.3.4 RISC-V minimal lock with retry loop (LR/SC) . . . 397
24.4 What Breaks Without Fences (or Acquire/Release) . . . 398
24.4.1 The simplest misconception . . . 398
24.4.2 Broken lock: no acquire/release ordering (conceptual) . . . 399
24.4.3 Concrete failure pattern: "publish under lock" is not published . . . 399
24.4.4 Demonstration in Rust: how to break it on purpose . . . 400
24.4.5 Fix: Acquire on lock, Release on unlock . . . 401
24.4.6 What fences look like at the assembly level (foundational intuition) . . . 402
24.4.7 Why fences matter even if you think you have a lock . . . 402
24.5 Extensive Worked Example: Correct Locking with a Protected Invariant . . . 403
24.5.1 Rust demonstration with correct spinlock . . . 403
24.6 Practical Rules (Assembler-Level Discipline) . . . 405
24.7 Summary . . . 405

V Common Engineering Failures 407

25 Stack Corruption 408
25.1 What Stack Corruption Really Means . . . 408
25.2 Typical Patterns . . . 409
25.2.1 Pattern 1: wrong stack pointer adjustment (frame size bug) . . . 409
25.2.2 Pattern 2: violating stack alignment rules . . . 410
25.2.3 Pattern 3: writing past a local buffer (classic overflow) . . . 411
25.2.4 Pattern 4: calling convention mismatch (SysV vs Win64, or wrong prototype) . . . 412
25.2.5 Pattern 5: Win64 shadow space violation . . . 413

25.2.6 Pattern 6: variadic calling rule mistakes . . . 414
25.2.7 Pattern 7: clobbering non-volatile registers . . . 414
25.2.8 Pattern 8: use-after-scope with stack addresses (escaping pointers) . . . 415
25.3 Diagnosis Workflow . . . 416
25.3.1 Step 1: classify the failure signature . . . 416
25.3.2 Step 2: verify ABI invariants for the failing function . . . 417
25.3.3 Step 3: locate the first frame that looks wrong . . . 417
25.3.4 Step 4: inspect stack memory around RSP and saved return address . . . 417
25.3.5 Step 5: check for call boundary mistakes . . . 418
25.3.6 Step 6: isolate by reducing the function . . . 418
25.3.7 Step 7: guard and canary logic (conceptual) . . . 419
25.3.8 Step 8: confirm unwind metadata assumptions (Windows) . . . 419
25.4 Extensive Examples: Failure and Fix (Rust on Windows + x86-64 assembly) . . . 420
25.4.1 Example A: Win64 call-site stack corruption by missing shadow space . 420
25.4.2 Example B: Rust calling into a function that clobbers a non-volatile register421
25.4.3 Example C: pointer-to-stack escape (unsafe Rust) . . . 422
25.5 Practical Prevention Rules . . . 423
25.6 Summary . . . 423

26 Calling Convention Violations 425
26.1 What a Calling Convention Really Guarantees . . . 425
26.2 Windows x64 ABI: The Contract You Must Obey . . . 426
26.2.1 Correct Windows x64 call sequence in GAS (Intel syntax) . . . 427
26.2.2 The classic broken call: missing shadow space . . . 428
26.3 Real Crash Signatures . . . 428
26.3.1 Signature A: Crash at RET or immediately after it . . . 429
26.3.2 Signature B: Access violation in a totally unrelated function . . . 429
26.3.3 Signature C: SIMD/alignment fault (often optimization-dependent) . . . 430

26.3.4 Signature D: Exception/unwind chaos when mixing toolchains 430
26.4 Cross-Compiler Disasters 431
26.4.1 Disaster 1: MSVC vs MinGW runtime boundary 431
26.4.2 Disaster 2: SysV assumptions accidentally used on Windows x64 . . . 432
26.4.3 Disaster 3: 32-bit calling convention mismatches (cdecl/stdcall/thiscall) 432
26.5 Rust on Windows: ABI Strings Are Not Documentation, They Are Instructions 433
26.5.1 Example: Correct Rust export callable from GAS assembly (Windows x64) 433
26.5.2 Example: Wrong ABI in Rust declaration (can crash instantly) 433
26.6 Violations You Can Reproduce and Learn From 434
26.6.1 Violation 1: Stack corruption via 32-bit stdcall vs cdecl mismatch . . . 434
26.6.2 Violation 2: Non-volatile register clobber (Windows x64) 436
26.6.3 Violation 3: Missing shadow space when calling into Rust from assembly (Windows x64) 437
26.6.4 Violation 4: Struct layout mismatch (repr(C) is not optional) 439
26.6.5 Violation 5: Variadic boundary misuse 439
26.7 Checklist: How to Prevent Calling Convention Violations 440
26.7.1 Boundary design rules 440
26.7.2 Assembly call rules for Windows x64 441
26.8 Laboratory: Diagnose a Convention Violation Like an Engineer 441
26.8.1 Step 1: Verify argument registers at the call boundary 441
26.8.2 Step 2: Verify stack alignment and shadow space 441
26.8.3 Step 3: Verify non-volatile registers across the call 442
26.8.4 Step 4: Reduce boundary complexity 442
26.9 Summary 442

27 Alignment Mistakes **443**
27.1 Why Alignment Bugs Are Uniquely Dangerous 443
27.2 Alignment Fundamentals You Must Treat as Invariants 444

27.2.1 Natural alignment vs required alignment 444
27.2.2 Stack alignment is an ABI rule, not an optimization hint 444
27.3 SIMD Crashes 445
27.3.1 Crash Pattern 1: Aligned vector load/store from an unaligned pointer . 445
27.3.2 Crash Pattern 2: Stack misalignment triggers SIMD faults in callees . . 446
27.3.3 Crash Pattern 3: Over-alignment assumptions (AVX/AVX-512 locals and spills) 448
27.3.4 Rust on Windows: Intrinsics and alignment requirements 448
27.3.5 Crash Pattern 4: Packed structs and misaligned fields 450
27.4 ABI Violations 451
27.4.1 Violation 1: Call-boundary stack alignment (Windows x64) 451
27.4.2 Violation 2: Over-aligned types across FFI without matching declarations 452
27.4.3 Violation 3: Returning or passing structs with mismatched layout/alignment 454
27.4.4 Violation 4: Rust `extern` ABI mismatch for Windows API calls 454
27.5 Alignment Debug Playbook (Academy Workflow) 455
27.5.1 Playbook A: Prove pointer alignment at the fault site 455
27.5.2 Playbook B: Prove stack alignment at every external call 456
27.5.3 Playbook C: Find the first boundary that breaks alignment 456
27.6 Engineering Patterns That Prevent Alignment Failures 456
27.6.1 Pattern 1: Always offer an unaligned-safe API at module boundaries . . 456
27.6.2 Pattern 2: Validate alignment in debug builds 457
27.6.3 Pattern 3: Do not trust implicit alignment across FFI 457
27.6.4 Pattern 4: Never write Windows x64 assembly without a fixed template 457
27.7 Laboratory Exercises 458
27.7.1 Lab 1: Trigger a `movaps` fault with an intentionally misaligned buffer . 458
27.7.2 Lab 2: Fix Lab 1 by switching to unaligned loads 458
27.7.3 Lab 3: Demonstrate Windows x64 stack alignment bug in assembly . . 459

27.8 Summary 459

28 Inline Assembly Pitfalls **460**

28.1 Why Inline Assembly Is a High-Risk Boundary 460

28.2 Inline Assembly in Rust on Windows: The Model 461

28.2.1 Rust `asm!` operands (core patterns) 461

28.2.2 Rust `asm!` options (side-effect modeling) 461

28.3 Clobbers 462

28.3.1 The core idea 462

28.3.2 Clobber mistake 1: Writing a register without declaring it 463

28.3.3 Clobber mistake 2: Lying about flags with `preserves_flags` 464

28.3.4 Clobber mistake 3: Touching memory while claiming `nomem` or `readonly` 465

28.3.5 Clobber mistake 4: Modifying the stack or calling from inline asm on Windows x64 466

28.4 Compiler Reordering 467

28.4.1 The core idea 467

28.4.2 Reordering mistake 1: Missing dependency between asm and memory . 467

28.4.3 Reordering mistake 2: Assuming `volatile` implies a full memory barrier 468

28.4.4 Reordering mistake 3: Incorrect use of `pure` and `readonly` 470

28.5 Constraint Errors 471

28.5.1 The core idea 471

28.5.2 Constraint error 1: Wrong width (32-bit vs 64-bit) on x86-64 472

28.5.3 Constraint error 2: Using `out` instead of `lateout` when overlap is possible 472

28.5.4 Constraint error 3: Forgetting that some instructions require specific registers 474

28.5.5 Constraint error 4: Confusing pointer operands and memory operands . 475

28.5.6 Constraint error 5: Mixing Windows x64 ABI rules with fixed registers 476

28.6 GAS Intel Syntax: Inline-Asm Thinking Without Inline-Asm Risk 476

28.7 Practical Rules (Academy Checklist) 478
28.7.1 Clobbers 478
28.7.2 Compiler reordering 478
28.7.3 Constraint discipline 478
28.8 Summary 479

29 Undefined Behavior at the Boundary **480**
29.1 What "Undefined Behavior at the Boundary" Means 480
29.2 Boundary Contracts You Must Treat as Invariants 481
29.2.1 Windows x64 ABI invariants 481
29.2.2 Rust FFI invariants 481
29.3 C/C++ + asm Traps 482
29.3.1 Trap 1: The compiler assumes more than the CPU requires 482
29.3.2 Trap 2: Missing ABI obligations in assembly callers (shadow space + alignment) 482
29.3.3 Trap 3: Flags, registers, and memory clobbers not modeled at the boundary 484
29.3.4 Trap 4: Layout and alignment mismatches across the boundary 486
29.3.5 Trap 5: Uninitialized and "trap" representations 487
29.3.6 Trap 6: Unwinding across the boundary 487
29.4 Lifetime Violations 488
29.4.1 Violation 1: Returning a pointer to a stack local (dangling pointer) . . . 488
29.4.2 Violation 2: Storing a borrowed pointer beyond its valid lifetime 490
29.4.3 Violation 3: Passing Rust references across the boundary 492
29.4.4 Violation 4: Creating slices from raw parts with wrong length 493
29.4.5 Violation 5: Callback lifetimes and re-entrancy 494
29.5 Boundary-Safe Design Patterns 495
29.5.1 Pattern 1: Use a C-style ABI surface even if your internals are advanced 495
29.5.2 Pattern 2: Make ownership explicit 495

29.5.3 Pattern 3: Treat alignment and layout as contracts, not assumptions . . 496
29.5.4 Pattern 4: Contain unwinding 496
29.6 Summary 496

VI PART VI — TOOLING AND DEBUGGING 498

30 Reading Compiler-Generated Assembly 499
30.1 Why Reading Compiler Assembly Is a Core Skill 499
30.2 Toolchain Setup: Producing Assembly You Can Study 499
30.2.1 Emit assembly from Rust (Windows) 500
30.2.2 Decide what assembly you want 500
30.3 Recognizing Patterns 501
30.3.1 Pattern 1: Windows x64 calling convention 501
30.3.2 Pattern 2: Prolog/epilog (stack frame setup) 502
30.3.3 Pattern 3: Non-volatile register preservation 503
30.3.4 Pattern 4: Stack alignment fix-ups 503
30.3.5 Pattern 5: Branchless conditionals (cmov, setcc, sbb trick) 504
30.3.6 Pattern 6: Bounds check + panic path 504
30.4 Optimization Artifacts 506
30.4.1 Artifact 1: Inlining and the disappearance of function boundaries . . . 506
30.4.2 Artifact 2: Dead code elimination and “impossible” missing work . . . 507
30.4.3 Artifact 3: Strength reduction and LEA-based arithmetic 508
30.4.4 Artifact 4: Loop transformations (unrolling, vectorization, and peeled iterations) 509
30.4.5 Artifact 5: Cold/hot splitting and “huge” error blocks 510
30.4.6 Artifact 6: Stack spills and register allocation illusions 511
30.4.7 Artifact 7: Security and platform scaffolding 511

30.5 A Practical Reading Workflow (Academy Method) . . . 512
30.5.1 Step 1: Determine the compilation regime . . . 512
30.5.2 Step 2: Locate the true entry and the true hot path . . . 512
30.5.3 Step 3: Identify ABI scaffolding first . . . 513
30.5.4 Step 4: Map registers to source-level meaning . . . 513
30.5.5 Step 5: Translate basic blocks, not lines . . . 513
30.5.6 Step 6: Only then interpret optimizations . . . 514
30.6 Extended Examples: From Source to "Typical" Assembly Shapes . . . 514
30.6.1 Example A: Copy loop that may become memset/memcpy-like . . . 514
30.6.2 Example B: Checked arithmetic vs wrapping arithmetic . . . 515
30.6.3 Example C: Match and jump tables . . . 516
30.7 Common Misreads (And How to Avoid Them) . . . 517
30.7.1 Misread 1: "The compiler added useless work" . . . 517
30.7.2 Misread 2: "This LEA means pointers" . . . 517
30.7.3 Misread 3: "The loop vanished, so it doesn't execute" . . . 517
30.7.4 Misread 4: "This function is wrong because it has no stack frame" . . . 518
30.8 Summary . . . 518
30.8.1 Recognizing patterns . . . 518
30.8.2 Recognizing optimization artifacts . . . 518

31 GDB and LLDB Essentials 520
31.1 Purpose of a Debugger in Assembly-Level Engineering . . . 520
31.2 Registers: Viewing and Interpreting . . . 521
31.2.1 Inspecting Registers in GDB . . . 521
31.2.2 Inspecting Registers in LLDB . . . 522
31.3 Stack Inspection . . . 523
31.3.1 Viewing Stack Contents . . . 523
31.3.2 Backtrace and Frame Navigation . . . 524

31.4 Breakpoints: Setting, Using, and Controlling Execution 525
31.4.1 Basic Breakpoints 525
31.4.2 Conditional Breakpoints 525
31.4.3 Stepping and Continuing Execution 526
31.4.4 Watching Memory and Registers 526
31.4.5 Breakpoints in Assembly-Level Debugging 527
31.5 Debugger Commands for Stack and Register Control 527
31.5.1 Examining a Frame in Detail 527
31.5.2 Setting the Instruction Pointer 528
31.5.3 Examining Floating and SIMD Registers 528
31.6 Range-Based and Pattern-Based Breakpoints 529
31.7 Debugging Rust Binaries Without Source (Assembly Focus) 529
31.8 Summary 530

32 WinDbg Essentials **531**
32.1 Why WinDbg Matters for Modern Assembly Work 531
32.2 Minimal Rust Setup for WinDbg-Friendly Builds 532
32.2.1 Cargo profiles for debuginfo on Windows 532
32.2.2 Build commands 532
32.2.3 A crash target to practice on 533
32.3 Symbol Awareness 533
32.3.1 The symbol path model in WinDbg 533
32.3.2 Essential symbol commands 534
32.3.3 Module listing and symbol status 534
32.3.4 Common symbol failure patterns (and how to recognize them) 535
32.3.5 Symbol-based navigation commands 536
32.4 Stack Analysis 537
32.4.1 Start from the fault context 537

32.4.2 Registers relevant to stack reading on Windows x64 538
32.4.3 Basic stack traces: k, kb, kp, kv 538
32.4.4 Stack memory inspection: proving whether the stack is sane 539
32.4.5 Disassembly at the fault site 540
32.4.6 A concrete crash walk-through (null dereference) 540
32.4.7 Diagnosing stack corruption versus missing unwind info 541
32.5 Symbol Awareness for Rust-Specific Realities 542
32.5.1 Name mangling and exported boundaries 542
32.5.2 Optimization effects on local variables 542
32.6 A Minimal WinDbg Command Card (Stack + Symbols) 543
32.6.1 Symbols 543
32.6.2 Crash anchor + stack 543
32.7 Summary 544
32.7.1 Stack analysis 544
32.7.2 Symbol awareness 544

33 Crash Analysis Workflow **545**
33.1 Introduction: What Is Crash Analysis? 545
33.2 Access Violation Analysis 546
33.2.1 Immediate Diagnostics Upon Crash 546
33.2.2 Disassemble Around the Fault 547
33.2.3 Common Root Causes for Access Violations 548
33.2.4 Memory Inspection at Fault Site 548
33.2.5 Correlation with Source Code 549
33.3 Stack Trace Reliability 549
33.3.1 Unwind Metadata and ABI Compliance 549
33.3.2 Getting a Stack Trace in WinDbg 550
33.3.3 When Stack Trace Is Not Reliable 551

33.3.4 Validating Stack Trace Frames 551
33.3.5 Handling Missing Unwind Info (Assembly Frames) 552
33.4 Crash Analysis Case Study: Rust Null Dereference 552
33.4.1 Step 1: Exception Context 553
33.4.2 Step 2: Disassemble Faulting Instruction 553
33.4.3 Step 3: Stack Backtrace 553
33.4.4 Step 4: Correlate With Source 554
33.5 Summary: A Repeatable Crash Workflow 554
33.5.1 Access Violation Analysis 554
33.5.2 Stack Trace Reliability 555

VII PRACTICAL CROSS-ARCHITECTURE EXAMPLES 556

34 Hello World — Linux x86-64 557
34.1 Why this example matters 557
34.2 Minimal prerequisites 558
34.3 Linux x86-64 system call ABI (the essentials) 558
34.4 Hello World (no libc): pure syscalls 559
34.5 Hello World (with libc): normal `main` 561
34.6 A precise comparison: syscall vs libc 563
34.7 Inspection and verification (engineering habits) 563
34.8 Cross-architecture takeaways (kept small, but comparable) 564
34.9 Practical pitfalls (what breaks beginners) 564
34.10Windows-focused Rust examples (x86-64) — pure Rust, no C runtime required 565
34.11Chapter summary 568

35 Hello World — Windows x64 570
35.1 Toolchain and file format realities on Windows x64 570

35.2 Windows x64 calling convention essentials . . . 571
35.3 Win32 API mapping for “Hello World” . . . 572
35.4 Hello World in pure assembly (no C runtime) . . . 572
35.5 Hello World with a normal `main` (CRT-managed startup) . . . 576
35.6 Precise comparison to Linux x86-64 . . . 578
35.7 Practical pitfalls (what breaks real code) . . . 578
35.8 Extensive Rust examples for Windows (no external crates) . . . 579
35.9 Build instructions for Rust examples (Windows x64) . . . 584
35.10 Chapter summary . . . 584

36 Hello World — AArch64 Linux **586**
36.1 Toolchain assumptions . . . 586
36.2 AArch64 Linux syscall ABI (the essentials) . . . 587
36.3 Hello World (no libc): pure syscalls . . . 587
36.4 Hello World (with libc): normal `main` . . . 589
36.5 A precise comparison: Linux x86-64 vs Linux AArch64 . . . 591
36.6 Practical pitfalls (common failure modes) . . . 592
36.7 Cross-architecture takeaway . . . 592
36.8 Rust examples (Windows OS only, as requested) . . . 593
36.9 Chapter summary . . . 596

37 Hello World — RISC-V Linux **597**
37.1 Toolchain assumptions . . . 597
37.2 Linux RISC-V syscall ABI (the essentials) . . . 598
37.3 Hello World (no libc): pure syscalls . . . 599
37.4 Hello World (with libc): normal `main` . . . 600
37.5 A precise comparison: Linux x86-64 vs AArch64 vs RISC-V . . . 602
37.6 Practical pitfalls (real failure modes) . . . 603

37.7 Cross-architecture takeaway . . . 603
37.8 Rust examples (Windows OS only, as requested) . . . 604
37.9 Chapter summary . . . 607

38 Function Call Comparisons — Same Function on 3 ISAs 608
38.1 Specification of the test function . . . 609
38.2 x86-64 System V ABI implementation . . . 609
38.3 AArch64 (ARM64 Linux ABI) implementation . . . 611
38.4 RISC-V 64 Linux ABI implementation . . . 613
38.5 Comparison tables: passing ints, pointers, floats . . . 614
38.6 Calling from Rust on Windows (MS x64 ABI) . . . 615
38.7 Practical pitfalls . . . 616
38.8 Summary . . . 617

39 Minimal Atomic Increment Example 618
39.1 What exactly are we implementing? . . . 618
39.2 Ordering model used in this chapter . . . 619
39.3 x86: `LOCK` atomic increment . . . 619
39.4 AArch64: `LDAXR/STLXR` exclusive loop . . . 621
39.5 RISC-V: `AMO` atomic increment . . . 622
39.6 A compact cross-ISA comparison . . . 624
39.7 Build notes (minimal, practical) . . . 624
39.8 Common pitfalls (what breaks correctness) . . . 625
39.9 Windows-only Rust examples (as requested) . . . 626
39.10 Chapter summary . . . 628

VIII ENGINEER REFERENCE 629

40 ABI Quick Tables 630

40.1 Legend and conventions . . . 631
40.2 x86-64 System V ABI (Linux): quick tables . . . 631
40.3 x86-64 Microsoft ABI (Windows): quick tables . . . 632
40.4 AArch64 ABI (Linux): quick tables . . . 633
40.5 RISC-V 64 ABI (Linux): quick tables . . . 635
40.6 Quick reference: argument passing examples (by type) . . . 636
40.7 Engineer checklist: writing correct ABI boundaries . . . 637
40.8 Windows-only Rust reference (as requested) . . . 637
40.9 Chapter summary . . . 639

41 Register Quick Reference 641

41.1 x86-64 register set (GPR) . . . 642
41.2 x86-64 SIMD/FP registers (XMM/YMM/ZMM) . . . 643
41.3 AArch64 register set (GPR) . . . 644
41.4 AArch64 SIMD/FP registers (V registers) . . . 645
41.5 RISC-V 64 register set (GPR) . . . 646
41.6 RISC-V floating-point registers (FPR) . . . 647
41.7 Quick cross-architecture mapping tables . . . 648
41.8 Windows-only Rust examples (as requested) . . . 649
41.9 Chapter summary . . . 651

42 Memory Ordering Quick Reference 652

42.1 Two independent reorderers: compiler and CPU . . . 652
42.2 A minimal vocabulary: atomicity vs ordering . . . 653
42.3 x86/x86-64 quick reference (TSO-style behavior) . . . 654
42.4 AArch64 quick reference (weakly ordered, explicit tools) . . . 656

42.5 RISC-V quick reference (RVWMO, explicit ordering) . . . 659
42.6 Quick tables: intent to instruction choices . . . 661
42.7 Common engineer pitfalls . . . 662
42.8 Windows-only Rust reference (as requested) . . . 663
42.9 Chapter summary . . . 665

43 Debugger Command Cheat Sheet 666
43.1 Symbols and builds that debuggers can use . . . 666
43.2 GDB essentials (Linux) . . . 667
43.3 LLDB essentials (Linux/macOS style, minimal) . . . 670
43.4 WinDbg / CDB essentials (Windows) . . . 671
43.5 Visual Studio debugger essentials (Windows) . . . 674
43.6 Cross-architecture register display patterns . . . 674
43.7 Minimal Rust debugging on Windows: practical workflow . . . 675
43.8 Engineer checklists . . . 677
43.9 Chapter summary . . . 678

44 Professional Assembly Style Guide 679
44.1 Core principles . . . 679
44.2 File layout conventions (GAS Intel syntax) . . . 681
44.3 Function header template (required in professional code) . . . 683
44.4 Naming conventions . . . 684
44.5 Indentation and formatting . . . 684
44.6 Comments: rules and quality . . . 685
44.7 Register discipline . . . 685
44.8 Stack discipline . . . 686
44.9 Data movement rules . . . 687
44.10 Control flow rules . . . 687

44.11Atomic and concurrency style 688
44.12Error handling and return conventions 689
44.13Interoperability rules (C/C++/Rust) 689
44.14Documentation and verification checklist 691
44.15Chapter summary 692

Part I

MACHINE REPRESENTATION

Binary, Hex, and Bit-Level Thinking

1.1 Why This Chapter Exists

Assembly is the art of controlling *representations*. At the CPU level, everything becomes:

- bit patterns stored in registers and memory,
- operations that transform bit patterns deterministically,
- conventions (ABIs) that interpret some bit patterns as arguments, return values, addresses, or flags.

If you cannot *think in bits*, you will write code that looks correct, compiles, and still fails in production. This chapter trains the mental model required to:

- treat bitwise operations as a compact algebra,
- build correct masks, shifts, and rotations,
- avoid the most dangerous signed/unsigned traps,
- translate high-level intent to correct low-level instructions (x86-64, AArch64, RISC-V),
- validate behavior with small, deterministic tests.

1.2 Bit Operations as Algebra

Bitwise operations are not "low-level hacks"; they form a practical algebra over fixed-width words. For a fixed width w (e.g., 8, 32, 64), values live in $\{0,1\}^w$ and operations are applied bit-by-bit.

1.2.1 The Three Core Operators: AND, OR, XOR

Let x and y be w-bit words.

- **AND** ($x \,\&\, y$) keeps a bit only if both have 1:

$$(x \,\&\, y)_i = x_i \cdot y_i$$

- **OR** ($x \mid y$) sets a bit if any has 1:

$$(x \mid y)_i = \max(x_i, y_i)$$

- **XOR** ($x \oplus y$) sets a bit if they differ:

$$(x \oplus y)_i = x_i + y_i \pmod 2$$

1.2.2 Identity and Annihilator Elements

These are the rules engineers use daily to simplify and verify bit logic:

- $x \,\&\, 0 = 0$ (AND annihilator)
- $x \,\&\, \texttt{all_ones} = x$ (AND identity)
- $x \mid 0 = x$ (OR identity)
- $x \mid \texttt{all_ones} = \texttt{all_ones}$ (OR annihilator)

- $x \oplus 0 = x$ (XOR identity)
- $x \oplus x = 0$ (XOR self-cancel)

1.2.3 Involution and Reversibility

XOR is *reversible*:

$$(x \oplus y) \oplus y = x$$

This single property powers:

- bit toggling,
- simple stream obfuscation,
- parity calculations,
- set symmetric-difference,
- hash mixing primitives (at foundation level).

1.2.4 NOT and De Morgan Laws

Bitwise NOT flips every bit:

$$\sim x = \texttt{all_ones} \oplus x$$

De Morgan laws are essential for correct mask algebra:

$$\sim (x \,\&\, y) = (\sim x) \,|\, (\sim y)$$

$$\sim (x \,|\, y) = (\sim x) \,\&\, (\sim y)$$

1.2.5 The Practical "Bitset" View

Treat a word as a set of bit positions:

- AND = intersection
- OR = union
- XOR = symmetric difference
- NOT = complement (within word width)

This viewpoint makes correctness proofs easier:

- **Test membership:** $(x \,\&\, (1 \ll k)) \neq 0$
- **Add element:** $x \leftarrow x \mid (1 \ll k)$
- **Remove element:** $x \leftarrow x \,\&\, \sim(1 \ll k)$
- **Toggle element:** $x \leftarrow x \oplus (1 \ll k)$

1.2.6 Extensive Rust Example (Windows-Friendly)

The following is a practical "bitset algebra" toolkit in Rust. It is deterministic, safe, and portable. It also teaches the exact mental model you should carry into assembly.

```rust
use std::fmt;

#[derive(Copy, Clone)]
struct Bits64(u64);

impl Bits64 {
    fn test(self, k: u32) -> bool {
```

```
    // k in [0,63] expected; shifting by >= 64 is a runtime panic in Rust.
     (self.0 & (1u64 << k)) != 0
}

fn set(mut self, k: u32) -> Self {
    self.0 |= 1u64 << k;
    self
}

fn clear(mut self, k: u32) -> Self {
    self.0 &= !(1u64 << k);
    self
}

fn toggle(mut self, k: u32) -> Self {
    self.0 ^= 1u64 << k;
    self
}

fn keep_only(self, mask: u64) -> Self {
    Bits64(self.0 & mask)
}

fn union(self, other: Self) -> Self {
    Bits64(self.0 | other.0)
}

fn inter(self, other: Self) -> Self {
    Bits64(self.0 & other.0)
```

```
    }

    fn symdiff(self, other: Self) -> Self {
        Bits64(self.0 ^ other.0)
    }
}

impl fmt::Debug for Bits64 {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "0x{:016X}", self.0)
    }
}

fn main() {
    let a = Bits64(0).set(1).set(3).set(60);
    let b = Bits64(0).set(3).set(4).set(60);

    println!("a = {:?}, b = {:?}", a, b);
    println!("a inter b  = {:?}", a.inter(b));
    println!("a union b  = {:?}", a.union(b));
    println!("a symdiff b= {:?}", a.symdiff(b));

    let a2 = a.toggle(3).clear(1).set(2);
    println!("a2 = {:?}, test bit 2 = {}", a2, a2.test(2));
}
```

1.3 Masking, Shifting, Rotation

Masking and shifting are the core of bit extraction, packing, encoding, and low-level protocols.

1.3.1 Mask Design: The Canonical Patterns

Single-Bit Mask

To isolate bit k:

$$m = 1 \ll k$$

$$x_k = (x \,\&\, m) \neq 0$$

Field Mask (n Bits Wide)

A field of width n starting at bit offset s:

$$\text{mask}(n, s) = ((1 \ll n) - 1) \ll s$$

Then:

$$\text{extract}(x, n, s) = (x \gg s) \,\&\, ((1 \ll n) - 1)$$

$$\text{insert}(x, v, n, s) = (x \,\&\, \sim \text{mask}(n, s)) \,|\, ((v \,\&\, ((1 \ll n) - 1)) \ll s)$$

1.3.2 Extensive Rust Example: Extract and Insert Bitfields

This is "academy-grade" bitfield manipulation suitable for systems code and testing.

```rust
fn mask_u64(width: u32, shift: u32) -> u64 {
    assert!(width > 0 && width <= 64);
    assert!(shift < 64);
    // Special-case width=64 to avoid shifting by 64 (invalid).
    if width == 64 {
        !0u64
    } else {
        ((1u64 << width) - 1) << shift
    }
}
```

```rust
fn extract_u64(x: u64, width: u32, shift: u32) -> u64 {
    assert!(width > 0 && width <= 64);
    assert!(shift < 64);
    if width == 64 {
        x
    } else {
        (x >> shift) & ((1u64 << width) - 1)
    }
}

fn insert_u64(x: u64, v: u64, width: u32, shift: u32) -> u64 {
    let m = mask_u64(width, shift);
    let v_trunc = if width == 64 { v } else { v & ((1u64 << width) - 1) };
    (x & !m) | (v_trunc << shift)
}

fn main() {
    // Example: pack a 3-bit field into bits [8..10] and extract it back.
    let mut x: u64 = 0x1234_5678_9ABC_DEF0;
    x = insert_u64(x, 0b101, 3, 8);
    let f = extract_u64(x, 3, 8);

    println!("x = 0x{:016X}", x);
    println!("field = 0b{:03b}", f);
}
```

1.3.3 Shifts: Logical vs Arithmetic

Shifts are not all equivalent. The *meaning* depends on whether the value is treated as signed.

- **Logical shift right** fills zeros from the left. Used for unsigned values and bitfields.

- **Arithmetic shift right** replicates the sign bit. Used for signed division-by-powers-of-two patterns and sign-preserving scaling.
- **Shift left** is the same bit movement for both signed and unsigned, but can overflow in high-level languages; in assembly it is well-defined bit movement with flags side effects.

1.3.4 Rotation: When Shifts Lose Information

Shifts discard bits; rotations preserve them by wrapping around.

- Rotate-left: $\text{rol}(x, k)$
- Rotate-right: $\text{ror}(x, k)$

Rotations are essential for:

- bit permutation and mixing,
- checksums and hash primitives,
- cryptographic construction foundations (without diving into full crypto yet),
- efficient circular buffers and ring indexing tricks (carefully).

1.3.5 GAS Assembly (Intel Syntax): Mask, Shift, Rotate

Below is a minimal x86-64 example in `gas` Intel syntax. **Note:** Comments are written as `/* ... */` as requested.

```
.intel_syntax noprefix
.text

/* u64 extract field: (x >> shift) & ((1<<width)-1)
```

```
   Inputs:
     rdi = x
     esi = shift (0..63)
     edx = width (1..63)   ; width=64 handled separately in high-level
   Output:
     rax = extracted
*/
.global extract_u64_asm
extract_u64_asm:
    mov     rax, rdi          /* rax = x */
    mov     ecx, esi          /* cl = shift */
    shr     rax, cl           /* logical shift right */
    mov     ecx, edx          /* cl = width */
    mov     r8, 1
    shl     r8, cl            /* r8 = 1 << width */
    dec     r8                /* r8 = (1<<width)-1 */
    and     rax, r8           /* rax &= mask */
    ret

/* rotate-left 64-bit
   Inputs:
     rdi = x
     esi = k (0..63)
   Output:
     rax = rol(x,k)
*/
.global rol64_asm
rol64_asm:
    mov     rax, rdi
    mov     ecx, esi
    rol     rax, cl
    ret
```

1.3.6 Rust Calling These Routines (Windows-Compatible Style)

On Windows, Rust commonly targets `x86_64-pc-windows-msvc`. The FFI boundary remains standard:

- use `extern "C"` to stabilize symbol ABI on the Rust side,
- use fixed-width integers (`u64`, `u32`) for clarity,
- keep parameter types simple (no Rust references across FFI).

```
#[link(name = "myasm")]
extern "C" {
    fn extract_u64_asm(x: u64, shift: u32, width: u32) -> u64;
    fn rol64_asm(x: u64, k: u32) -> u64;
}

fn main() {
    let x = 0x1234_5678_9ABC_DEF0u64;

    unsafe {
        let f = extract_u64_asm(x, 8, 12);
        let r = rol64_asm(x, 17);
        println!("extract[8..19] = 0x{:X}", f);
        println!("rol17          = 0x{:016X}", r);
    }
}
```

1.4 Common Signed/Unsigned Traps

Signedness errors are among the most expensive low-level bugs because they:

- pass tests with “normal” values,

- fail only at boundary values,
- appear as random memory corruption or wrong branches in assembly.

1.4.1 Trap 1: Comparing Signed Values as Unsigned (and Vice Versa)

A classic failure: a negative signed integer becomes a huge unsigned integer after reinterpretation.

Rust Demonstration

Rust makes conversions explicit, which is good, but explicit does not mean safe. The conversion is still a reinterpretation modulo 2^w.

```rust
fn main() {
    let a: i32 = -1;
    let b: u32 = a as u32; // reinterpret bits: 0xFFFF_FFFF = 4_294_967_295

    println!("a (i32) = {}", a);
    println!("b (u32) = {}", b);

    // A signed check:
    if a < 10 {
        println!("a < 10 is true (signed)");
    }

    // An unsigned check:
    if b < 10 {
        println!("b < 10 is true (unsigned)");
    } else {
        println!("b < 10 is false (unsigned), because b is huge");
    }
}
```

1.4.2 Trap 2: Right Shift on Signed Values

In many systems languages, right shift of signed negatives can be arithmetic (sign-propagating). In assembly, you must choose the correct instruction:

- x86: `shr` (logical), `sar` (arithmetic)

GAS Intel Syntax Example: shr vs sar

```
.intel_syntax noprefix
.text

/* Demonstrate difference:
   Input: edi = signed value
   Output: eax = (value >> 1) arithmetic, edx = (value >> 1) logical
*/
.global shr_sar_demo
shr_sar_demo:
    mov     eax, edi      /* eax = value */
    sar     eax, 1        /* arithmetic shift right */
    mov     edx, edi      /* edx = value */
    shr     edx, 1        /* logical shift right */
    ret
```

1.4.3 Trap 3: Mixed-Width Operations and Implicit Extension

A smaller integer (like 8-bit) must be extended to a larger register. The extension choice matters:

- **Zero-extend** preserves unsigned meaning.
- **Sign-extend** preserves signed meaning.

x86-64: movzx vs movsx

```
.intel_syntax noprefix
.text

/* Inputs:
   dil = byte value
   Output:
   eax = zero-extended (unsigned)
   edx = sign-extended (signed)
*/
.global extend_demo
extend_demo:
    movzx   eax, dil      /* eax = (u8)dil */
    movsx   edx, dil      /* edx = (i8)dil */
    ret
```

1.4.4 Trap 4: Shifting by the Word Size or More

At the machine level, many ISAs mask the shift count (or define special behavior). At the language level, shifting by >= `bitwidth` is often invalid or traps.
In Rust, shifting by an amount >= the number of bits is a runtime panic in safe code. This is a *feature*: it prevents silent wrong results.

Safe Rust Pattern for Variable Shifts

```
fn shl_checked_u64(x: u64, k: u32) -> Option<u64> {
    if k < 64 { Some(x << k) } else { None }
}

fn main() {
    let x = 1u64;
    for &k in &[0u32, 1, 63, 64, 100] {
```

```
        match shl_checked_u64(x, k) {
            Some(v) => println!("1 << {} = 0x{:016X}", k, v),
            None => println!("1 << {} is invalid for u64", k),
        }
    }
}
```

1.4.5 Trap 5: Using Signed Types for Bitfields

Bitfields are representation logic; representation logic should almost always use **unsigned** types. If you store masks in signed types:

- comparisons change meaning,
- right shifts may sign-propagate,
- formatting and debugging become confusing,
- boundary cases become dangerous.

Correct Rule

- Use `u32`/`u64` for masks and bitfields.
- Convert to signed only when you have a semantic signed quantity.
- At assembly boundaries, document whether each register is carrying signed or unsigned meaning.

1.5 Hex Thinking: The Engineer's Compression Format

Hex is not a convenience; it is the natural representation of bytes.

1.5.1 Nibble-to-Bit Mapping

A single hex digit represents 4 bits. This means:

- two hex digits = one byte,
- eight hex digits = 32-bit value,
- sixteen hex digits = 64-bit value.

1.5.2 Hex Dump Discipline

When you see memory:

- group by bytes first,
- then interpret by endianness,
- then interpret by type and alignment.

1.5.3 Rust Example: Endian-Explicit Decoding

This is a correct pattern for decoding byte arrays into integers without ambiguity.

```rust
fn main() {
    let bytes: [u8; 8] = [0xF0, 0xDE, 0xBC, 0x9A, 0x78, 0x56, 0x34, 0x12];

    let le = u64::from_le_bytes(bytes);
    let be = u64::from_be_bytes(bytes);

    println!("bytes = {:02X?}", bytes);
    println!("as LE = 0x{:016X}", le);
    println!("as BE = 0x{:016X}", be);
}
```

1.6 Practical Exercises (Foundation Level)

1. Implement a function that packs:
   - 6-bit field at offset 0,
   - 10-bit field at offset 6,
   - 16-bit field at offset 16,

   into a single `u64`. Verify by extracting back.
2. Write an x86-64 assembly routine (GAS Intel syntax) that:
   - extracts a field,
   - inserts a field,
   - rotates a word,

   and call it from Rust on Windows via `extern "C"`.
3. Create test vectors that include edge cases:
   - all zeros,
   - all ones,
   - only MSB set,
   - only LSB set,
   - alternating patterns: `0xAAAA...` and `0x5555...`.

1.7 Professional Summary

Bit-level thinking is the foundation beneath every ABI, instruction, and optimization. Mastery requires:

- treating bit ops as algebra (simplify and prove),
- designing masks with canonical formulas,
- choosing correct shift and rotation semantics,
- enforcing unsigned discipline for representation tasks,
- recognizing signed/unsigned traps before they reach the debugger.

Once you can perform these transformations confidently on paper, you can implement them correctly in any ISA. The next chapters will extend these skills into instruction formats, register models, and ABI-correct function calling.

Two's Complement and Integer Behavior

2.1 Scope and Engineering Goal

This chapter establishes the exact, machine-precise behavior of signed integers in two's complement systems. You will learn to:

- reason about **overflow** as modular arithmetic versus *signed overflow*,
- apply **sign extension** correctly across widths (8/16/32/64),
- understand **flag interaction** (x86 flags, ARM condition flags, and the flagless RISC-V model),
- write tests that prove behavior at boundary values.

Throughout this chapter:

- "bit pattern" means the raw bits in a register or memory,
- "signed meaning" means interpreting that pattern in two's complement,
- "unsigned meaning" means interpreting it as a natural number modulo 2^w.

2.2 Overflow

2.2.1 Two Kinds of Overflow Engineers Must Separate

In fixed-width arithmetic with word size w (e.g., $w = 32$):

- **Unsigned overflow** is not an error; results are computed modulo 2^w.
- **Signed overflow** is a *semantic* event: the mathematical result cannot be represented in $[-2^{w-1}, 2^{w-1} - 1]$.

CPUs generally compute the same low w bits for addition/subtraction regardless of signedness. Signedness affects:

- how you interpret the result,
- which flag you consult to detect overflow (CF for unsigned carry/borrow; OF for signed overflow on x86),
- which branch condition you use.

2.2.2 Unsigned Addition Overflow

For unsigned addition in w bits:

$$r = (a + b) \bmod 2^w$$

Unsigned overflow occurs when $a + b \geq 2^w$. On x86, this condition sets the **Carry Flag (CF)**.

Rust Example: Unsigned Overflow Detection (Windows-Friendly)

Rust provides explicit, correctness-friendly APIs for overflow behavior.

```rust
fn main() {
    let a: u32 = 0xFFFF_FFF0;
    let b: u32 = 0x0000_0030;

    let (r, carry) = a.overflowing_add(b);
    println!("a       = 0x{:08X}", a);
    println!("b       = 0x{:08X}", b);
    println!("r       = 0x{:08X}", r);
    println!("carry   = {}", carry);

    let checked = a.checked_add(b);
    println!("checked = {:?}", checked);

    let sat = a.saturating_add(b);
    println!("sat     = 0x{:08X}", sat);
}
```

2.2.3 Signed Addition Overflow

For signed w-bit two's complement, representable range is:

$$\min = -2^{w-1}, \quad \max = 2^{w-1} - 1$$

Signed overflow occurs when the mathematical sum is outside this range.

The Practical Rule (Two's Complement)

Signed overflow on addition happens when:

- the inputs have the **same sign**,
- but the result has a **different sign**.

Equivalently (in bit terms):

- adding two positives yields a negative,
- adding two negatives yields a positive.

On x86, this condition sets the **Overflow Flag (OF)**.

Rust Example: Signed Overflow Behavior

Rust offers:

- `overflowing_add` for modular result + flag,
- `checked_add` for `Option`,
- `saturating_add` for clamping,
- `wrapping_add` for explicit modulo arithmetic.

```rust
fn main() {
    let max: i32 = i32::MAX;
    let one: i32 = 1;

    let (r, of) = max.overflowing_add(one);
    println!("max      = {}", max);
    println!("one      = {}", one);
    println!("r        = {}", r);
    println!("overflow= {}", of);

    println!("checked = {:?}", max.checked_add(one));
    println!("wrap    = {}", max.wrapping_add(one));
    println!("sat     = {}", max.saturating_add(one));

    let min: i32 = i32::MIN;
    let (r2, of2) = min.overflowing_sub(one);
    println!("min      = {}", min);
```

```
    println!("min-1   = {}", r2);
    println!("overflow= {}", of2);
}
```

2.2.4 Subtraction Overflow and Borrow

Subtraction can be viewed as:

$$a - b = a + (\sim b + 1)$$

But engineers must still distinguish:

- **Unsigned borrow**: on x86, CF indicates a borrow for subtraction (CF=1 after `sub` when unsigned underflow occurs).
- **Signed overflow**: on x86, OF indicates signed overflow for subtraction.

Signed overflow on subtraction happens when:

- the operands have **different signs**,
- and the result sign differs from the sign of a.

2.2.5 Boundary-Value Discipline

To prove integer logic, always test the boundary patterns:

- 0
- 1
- -1 (all ones)
- `MIN` ($0x8000...$)
- `MAX` ($0x7FFF...$)
- alternating patterns: `0xAAAA...`, `0x5555...`

2.3 Sign Extension

2.3.1 What Sign Extension Means

Sign extension is the operation of widening a signed integer while preserving its mathematical value.
If an n-bit signed value has most-significant bit (MSB) s:

- if $s = 0$ (non-negative), extend with zeros,
- if $s = 1$ (negative), extend with ones.

This is not a "CPU trick"; it is required by the mathematics of two's complement.

2.3.2 Zero Extension vs Sign Extension

When widening:

- **Zero extension** preserves *unsigned meaning*.
- **Sign extension** preserves *signed meaning*.

If you apply the wrong extension, your program can remain stable but compute consistently wrong results.

2.3.3 x86-64 Sign Extension Instructions (Foundation Set)

Common widening instructions:

- `movzx` : zero-extend (e.g., byte to 32-bit)
- `movsx` : sign-extend (e.g., byte to 32-bit)

- `movsxd` : sign-extend 32-bit to 64-bit

Implicit extension rules also matter:

- writing to a 32-bit register (e.g., `eax`) zero-extends into the full 64-bit register (`rax`).

2.3.4 GAS Assembly (Intel Syntax): Zero vs Sign Extension Demonstration

```
.intel_syntax noprefix
.text

/* Input:
     dil = 8-bit value
   Output:
     eax = zero-extended value (u8 -> u32)
     edx = sign-extended value (i8 -> i32)
*/
.global ext_u8_i8_demo
ext_u8_i8_demo:
    movzx   eax, dil          /* eax = (u8)dil */
    movsx   edx, dil          /* edx = (i8)dil */
    ret

/* Input:
     edi = 32-bit signed value
   Output:
     rax = sign-extended to 64-bit
*/
.global ext_i32_to_i64_demo
ext_i32_to_i64_demo:
    movsxd  rax, edi          /* rax = (i64)(i32)edi */
    ret
```

2.3.5 Rust Example: Explicit Widening Correctness

Rust requires explicit casts; the engineer must choose correctly.

```rust
fn main() {
    let b: u8 = 0xF0;          // 240
    let sb: i8 = -16;          // 0xF0 interpreted as signed

    let z1: u32 = b as u32;    // zero-extension semantics
    let s1: i32 = sb as i32;   // sign-extension semantics

    println!("b  (u8)  = {}", b);
    println!("sb (i8)  = {}", sb);
    println!("z1 (u32) = {}", z1);
    println!("s1 (i32) = {}", s1);

    // The dangerous mistake:
    let wrong: i32 = (b as i8) as i32; // reinterpret u8 bits as i8, then extend signed
    println!("wrong    = {}", wrong);
}
```

2.3.6 Why Sign Extension Appears Everywhere in Assembly

You will use sign extension when:

- reading smaller signed values from memory (e.g., int8, int16),
- implementing arithmetic that mixes widths,
- preparing arguments for ABI registers that expect a larger width,
- implementing branch logic based on signed comparisons.

2.4 Flag Interaction

Different architectures handle integer condition evaluation differently. Understanding flags is essential for writing correct comparisons, overflow checks, and multiword arithmetic.

2.4.1 x86 Status Flags (Foundation Set)

The most important x86 flags for integer work:

- **CF** (Carry Flag): unsigned carry/borrow for add/sub and multiword arithmetic
- **OF** (Overflow Flag): signed overflow for add/sub
- **ZF** (Zero Flag): result is zero
- **SF** (Sign Flag): MSB of result (sign bit)

2.4.2 Flag Semantics After ADD and SUB

- `add` affects CF, OF, ZF, SF (and others).
- `sub` affects CF, OF, ZF, SF.

Key engineer rule:

- Consult **CF** for *unsigned* overflow/borrow.
- Consult **OF** for *signed* overflow.

2.4.3 GAS Assembly (Intel Syntax): Capturing CF and OF

This routine adds two 32-bit integers and returns:

- the 32-bit result,
- whether CF was set,
- whether OF was set.

```
.intel_syntax noprefix
.text

/* u32 add_flags_u32(u32 a, u32 b, u32* out_cf, u32* out_of)
   Windows x64 / SysV note:
     This is a low-level example of flag capture; integrate via correct calling convention
     ↪  in your build.
   Inputs:
     edi = a
     esi = b
     rdx = out_cf pointer
     rcx = out_of pointer
   Output:
     eax = (a + b) low 32 bits
*/
.global add_flags_u32
add_flags_u32:
    mov     eax, edi          /* eax = a */
    add     eax, esi          /* eax = a + b, flags updated */

    setc    r8b               /* r8b = 1 if CF=1 else 0 */
    seto    r9b               /* r9b = 1 if OF=1 else 0 */

    movzx   r8d, r8b
    movzx   r9d, r9b

    mov     dword ptr [rdx], r8d   /* *out_cf = CF */
    mov     dword ptr [rcx], r9d   /* *out_of = OF */
```

```
    ret
```

2.4.4 Rust Example: Comparing CPU Flags vs Language Overflow APIs

In high-level code, you should typically use language APIs (`overflowing_add`, `checked_add`). But when integrating assembly, it is valuable to confirm equivalence with deterministic tests.

```
#[repr(C)]
struct FlagsOut {
    cf: u32,
    of: u32,
}

#[link(name = "myasm")]
extern "C" {
    fn add_flags_u32(a: u32, b: u32, out_cf: *mut u32, out_of: *mut u32) -> u32;
}

fn main() {
    let tests: &[(u32, u32)] = &[
        (0, 0),
        (1, 2),
        (0xFFFF_FFFF, 1),
        (0x7FFF_FFFF, 1),
        (0x8000_0000, 0x8000_0000),
    ];

    for &(a, b) in tests {
        let mut f = FlagsOut { cf: 0, of: 0 };
        let r_asm = unsafe { add_flags_u32(a, b, &mut f.cf as *mut u32, &mut f.of as *mut
        ↪  u32) };

        let (r_rust, carry_rust) = a.overflowing_add(b);
```

```
        // Signed overflow check in Rust:
        let a_s = a as i32;
        let b_s = b as i32;
        let (r_s, of_rust) = a_s.overflowing_add(b_s);

        println!("a=0x{:08X} b=0x{:08X}", a, b);
        println!(" asm r=0x{:08X} CF={} OF={}", r_asm, f.cf, f.of);
        println!("rust r=0x{:08X} CF={}  signed_r={} OF={}", r_rust, carry_rust, r_s,
        ↪  of_rust);
        println!();
    }
}
```

2.4.5 Flag Pitfalls That Break Real Programs

Pitfall 1: Using the Wrong Condition for Signed vs Unsigned

On x86:

- unsigned comparisons use CF/ZF relations (e.g., `jb`, `ja`)
- signed comparisons use SF/OF/ZF relations (e.g., `jl`, `jg`)

If you accidentally use a signed jump for an unsigned quantity, your logic can silently fail for values where the MSB is 1.

Pitfall 2: Assuming Flags Survive Instructions That Do Not Preserve Them

Many instructions modify flags; some do not. In correct low-level code:

- assume flags are ephemeral,
- capture them immediately if needed (`setc`, `seto`, conditional jumps),

- never insert unrelated arithmetic between a flag-setting instruction and its consumer.

Pitfall 3: Confusing `INC/DEC` vs `ADD/SUB`

A critical x86 detail:

- `inc` and `dec` do *not* update CF,
- `add` and `sub` do update CF.

When implementing multiword arithmetic or carry-based algorithms, using `inc/dec` can introduce hard-to-find bugs.

GAS Intel Syntax: Demonstrating `inc` vs `add`

```
.intel_syntax noprefix
.text

/* Input:
     eax = initial value
   Output:
     eax = eax + 1 using INC (CF unchanged)
     edx = CF after INC captured (may be old CF)
*/
.global inc_cf_demo
inc_cf_demo:
    stc                     /* set CF=1 intentionally */
    inc     eax             /* CF is not modified by INC */
    setc    dl              /* captures CF (still 1) */
    movzx   edx, dl
    ret

/* Input:
     eax = initial value
```

```
   Output:
     eax = eax + 1 using ADD (CF updated)
     edx = CF after ADD captured
*/
.global add_cf_demo
add_cf_demo:
    stc                     /* set CF=1 intentionally */
    add     eax, 1          /* CF updated by ADD */
    setc    dl
    movzx   edx, dl
    ret
```

2.4.6 ARM and RISC-V Flag Model (Foundation Comparison)

AArch64 (Condition Flags)

AArch64 commonly uses NZCV flags for conditional branches. Key foundation points:

- many arithmetic instructions have flag-setting variants,
- conditional branches consult those flags.

RISC-V (No Global Flags)

RISC-V does not use global condition flags like x86. Instead, comparisons produce results reminding you:

- “comparison is an instruction that writes a register”,
- branches compare registers directly.

This difference changes how you translate code:

- x86 often does: `cmp` then `jcc`
- RISC-V often does: `slt`/`sltu` or branch comparisons directly.

2.5 Practical Exercises (Boundary and Flag Correctness)

1. Write Rust tests that validate:
    - `checked_add` fails for `i32::MAX + 1`,
    - `overflowing_add` returns overflow flag for same case,
    - `wrapping_add` matches modular arithmetic.
2. In x86-64 assembly, implement:
    - `add_flags_u64`: returns result and stores CF/OF in memory,
    - `sub_flags_u64`: returns result and stores CF/OF.
3. Demonstrate sign extension correctness:
    - load byte `0x80` (which is `-128` as `i8`),
    - show `movsx` yields `0xFFFF...FF80`,
    - show `movzx` yields `0x0000...0080`.
4. Prove the `INC` carry pitfall:
    - set CF to 1,
    - run `inc`,
    - read CF (unchanged),
    - repeat using `add` and observe CF updates correctly.

2.6 Professional Summary

Two's complement is best understood as:

- a fixed-width modular arithmetic system for bit patterns,
- with a signed interpretation range that introduces the concept of signed overflow.

Professional correctness requires three disciplines:

- **Overflow discipline:** consult CF for unsigned carry/borrow and OF for signed overflow; use explicit overflow APIs in high-level code.
- **Sign-extension discipline:** always widen signed quantities with sign extension and unsigned quantities with zero extension; document the intended meaning per register.
- **Flag discipline:** treat flags as short-lived; capture them immediately never assume they survive unrelated instructions; avoid `INC`/`DEC` when carry correctness matters.

Floating-Point Basics for Assembly Authors

3.1 Scope and Engineering Goal

Floating-point is not “real numbers on hardware”; it is a standardized approximation system with:

- finite precision,
- special values (NaN, infinities, signed zeros),
- defined rounding,
- well-defined bit layouts for common formats.

For assembly authors, floating-point correctness is mainly about:

- understanding IEEE-754 representations and corner cases,
- respecting ABI rules for passing and returning floating values,
- ensuring cross-language boundaries do not silently change meaning, precision, or calling convention.

This chapter focuses on three sections:

- IEEE-754 essentials,

- ABI implications,
- passing floats between languages (with Windows-friendly Rust examples).

3.2 IEEE-754 Essentials

3.2.1 The Two Workhorse Formats: Binary32 and Binary64

Most systems software uses:

- **Binary32** (commonly `float` / `f32`): 32-bit
- **Binary64** (commonly `double` / `f64`): 64-bit

Binary32 (f32) Layout

- 1 sign bit: s
- 8 exponent bits: e (bias 127)
- 23 fraction bits: f

For normalized numbers:

$$\text{value} = (-1)^s \times (1.f) \times 2^{(e-127)}$$

The leading `1` before the binary point is *implicit* (not stored) for normalized values.

Binary64 (f64) Layout

- 1 sign bit: s
- 11 exponent bits: e (bias 1023)
- 52 fraction bits: f

For normalized numbers:

$$\text{value} = (-1)^s \times (1.f) \times 2^{(e-1023)}$$

3.2.2 Special Encodings and Their Meaning

IEEE-754 defines special categories based on exponent field:

Zeros

- $e = 0$, $f = 0$ gives **zero**
- sign bit distinguishes **+0** and **-0**

Both compare equal in standard comparisons, but the sign can matter in:

- division: $1/(+0) = +\infty$ vs $1/(-0) = -\infty$
- some functions and sign-sensitive transformations

Subnormals (Denormals)

- $e = 0$, $f \neq 0$ gives **subnormal**

Subnormals represent very small magnitudes with reduced precision:

$$\text{value} = (-1)^s \times (0.f) \times 2^{(1-\text{bias})}$$

Many high-performance systems choose modes that flush subnormals to zero in certain contexts. As an assembly author, you must assume:

- subnormals can exist in input data,
- execution environments may vary in how they treat them unless explicitly configured.

Infinities

- $e =$ all ones, $f = 0$ gives $+\infty$ **or** $-\infty$

Produced by overflow or division by zero (depending on operation and environment).

NaNs (Not a Number)

- $e =$ all ones, $f \neq 0$ gives **NaN**

Key properties:

- NaN is *unordered*: comparisons like ==, <, > are false (except `!=` is true).
- NaNs propagate through many arithmetic operations.

3.2.3 Rounding: Why the Last Bit Often Lies

IEEE-754 requires rounding when results cannot be represented exactly. Common default rounding mode is *round to nearest, ties to even*. This matters because:

- floating addition is not associative: $(a + b) + c$ may differ from $a + (b + c)$,
- reordering by compiler or CPU can change the least significant bits,
- vectorization or fused operations can produce different results than scalar step-by-step code.

3.2.4 Exceptions and Status Flags (Conceptual)

IEEE-754 defines exception conditions such as:

- invalid operation (e.g., `0/0`),

- division by zero,
- overflow,
- underflow,
- inexact.

Hardware typically tracks these in a floating-point status/control environment. As a foundations rule:

- do not assume exceptions raise traps by default,
- do assume the status flags may be set and later observed by runtime libraries or other code.

3.2.5 Rust (Windows-Friendly) Example: Bit-Exact Inspection and Classification

When validating assembly, you frequently need to inspect raw float bits. Rust provides `to_bits` and `from_bits` for exact bit-level work.

```
fn show_f32(x: f32) {
    let bits = x.to_bits();
    let sign = bits >> 31;
    let exp  = (bits >> 23) & 0xFF;
    let frac = bits & 0x7F_FFFF;

    println!("f32 = {:e}", x);
    println!("bits= 0x{:08X}", bits);
    println!("s={} e=0x{:02X} f=0x{:06X}", sign, exp, frac);

    if exp == 0 && frac == 0 {
        println!("class: zero ({})", if sign == 1 { "-0" } else { "+0" });
```

```
    } else if exp == 0 {
        println!("class: subnormal");
    } else if exp == 0xFF && frac == 0 {
        println!("class: infinity ({})", if sign == 1 { "-inf" } else { "+inf" });
    } else if exp == 0xFF {
        println!("class: NaN");
    } else {
        println!("class: normal");
    }
    println!();
}

fn main() {
    show_f32(0.0);
    show_f32(-0.0);
    show_f32(1.0);
    show_f32(f32::INFINITY);
    show_f32(f32::NEG_INFINITY);
    show_f32(f32::NAN);

    let tiny = f32::from_bits(0x0000_0001); // smallest positive subnormal
    show_f32(tiny);
}
```

3.3 ABI Implications

Floating-point is not only a numeric topic; it is an ABI topic. ABIs define:

- which registers carry floating arguments,
- where return values appear,
- how mixed integer/float arguments are assigned,

- how aggregates containing floats are passed,
- alignment rules for stack-spilled floats and vector values.

3.3.1 Core Principle: Do Not Guess the ABI

Assembly that crosses language boundaries must follow the platform ABI exactly. If you guess incorrectly, typical symptoms include:

- correct results in trivial tests but failures in real calls,
- corrupted stack or incorrect return values,
- values appearing shifted, truncated, or NaN unexpectedly.

3.3.2 Common 64-bit ABIs (Foundations View)

This foundations chapter gives the essential mental model. Detailed ABI tables appear later in the volume.

System V AMD64 (Common on Linux/Unix-like)

- floating args: XMM0–XMM7
- float return: XMM0
- integer/pointer args: RDI, RSI, RDX, RCX, R8, R9
- stack alignment: 16-byte at call boundary

Microsoft x64 (Windows)

- floating args: XMM0–XMM3
- float return: XMM0
- integer/pointer args: RCX, RDX, R8, R9
- 32-byte shadow space reserved by caller
- stack alignment: 16-byte at call boundary

AArch64 (Linux/Windows on ARM64 environments)

- floating args: V0–V7
- float return: V0
- integer/pointer args: X0–X7
- stack alignment: typically 16-byte

RISC-V (RV64, typical calling convention)

- floating args: fa0–fa7
- float return: fa0
- integer/pointer args: a0–a7
- stack alignment: commonly 16-byte

3.3.3 The Precision Trap: x87 vs SSE vs Strictly-64-bit Semantics

On modern x86-64 toolchains:

- scalar float work is typically done using SSE2+ (XMM registers),
- legacy x87 extended precision can still exist in some environments, but it is usually not the default model for x86-64 ABI floating returns/args.

Engineering rule:

- when interfacing with modern C/Rust on x86-64, use XMM registers for scalar float ABI compliance.

3.4 Passing Floats Between Languages

3.4.1 FFI Rule Set for Floating Values

Across C and Rust (and other languages), to make float FFI reliable:

- use `extern "C"` on the high-level side to force C ABI,
- use fixed, explicit types: `f32`, `f64`,
- avoid passing platform-dependent types (notably `long double`) unless you fully control both sides and ABI,
- keep aggregates simple: prefer passing floats as separate parameters rather than embedding them in complex structs at the boundary,
- never pass Rust references or non-`repr(C)` aggregates across FFI.

3.4.2 Minimal ABI-Safe Patterns

Pattern A: Pure Scalar f64

We define an assembly function:

- inputs: two `f64`
- output: `f64`

GAS Assembly (Intel Syntax): `add_f64`

This code uses XMM registers and works for the common x86-64 scalar float model.

```
.intel_syntax noprefix
.text

/* f64 add_f64(f64 a, f64 b)
   ABI note (x86-64):
     a in xmm0, b in xmm1, return in xmm0
*/
.global add_f64
add_f64:
    addsd   xmm0, xmm1        /* xmm0 = xmm0 + xmm1 */
    ret
```

Rust (Windows) Calling the Assembly Function

On Windows, Rust typically uses the Microsoft x64 ABI by default on `x86_64-pc-windows-msvc`. Using `extern "C"` aligns with the platform C ABI.

```
#[link(name = "myasm")]
extern "C" {
    fn add_f64(a: f64, b: f64) -> f64;
```

```
}

fn main() {
    let a = 1.25f64;
    let b = 2.75f64;
    let r = unsafe { add_f64(a, b) };
    println!("{} + {} = {}", a, b, r);
}
```

3.4.3 Pattern B: Using Float Bit Patterns Across the Boundary

Sometimes you need to pass floats as raw bits to avoid any conversion ambiguity. This is common in:

- serialization formats,
- testing NaN payload propagation,
- implementing bit-exact libraries.

Rust Example: Pass f64 as u64 Bits

```
#[link(name = "myasm")]
extern "C" {
    fn negate_sign_f64_bits(x_bits: u64) -> u64;
}

fn main() {
    let x = -13.5f64;
    let bits = x.to_bits();
    let out_bits = unsafe { negate_sign_f64_bits(bits) };
    let y = f64::from_bits(out_bits);
```

```
    println!("x      = {}", x);
    println!("x_bits = 0x{:016X}", bits);
    println!("y      = {}", y);
    println!("y_bits = 0x{:016X}", out_bits);
}
```

GAS Assembly (Intel Syntax): Flip Sign Bit in f64 Bit Pattern

This routine flips the sign bit of an IEEE-754 binary64 bit pattern. It is purely integer logic, independent of floating-point execution modes.

```
.intel_syntax noprefix
.text

/* u64 negate_sign_f64_bits(u64 x_bits)
   Input:
     x_bits in rcx (Windows) or rdi (SysV)
   Output:
     rax = x_bits ^ (1<<63)
   Note:
     This function assumes a C-like ABI; adjust register mapping by platform
     at integration time. The operation itself is invariant.
*/
.global negate_sign_f64_bits
negate_sign_f64_bits:
    mov     rax, rcx          /* Windows x64: first integer arg in rcx */
    mov     rdx, 0x8000000000000000
    xor     rax, rdx
    ret
```

3.4.4 Pattern C: Comparing Floats and Handling NaNs Correctly

In IEEE-754, NaN breaks ordinary comparisons. Assembly authors must know how the ISA reports unordered comparisons.

x86-64 Scalar Comparison: Ordered vs Unordered

For scalar double comparisons using SSE:

- `ucomisd` sets flags based on comparison and also marks unordered when NaN is involved.
- conditional jumps must be chosen carefully when NaNs can appear.

GAS Assembly (Intel Syntax): `is_unordered_f64`

This routine returns 1 if either input is NaN, else 0.

```
.intel_syntax noprefix
.text

/* u32 is_unordered_f64(f64 a, f64 b)
   Inputs:
     a in xmm0, b in xmm1
   Output:
     eax = 1 if unordered (NaN involved), else 0
*/
.global is_unordered_f64
is_unordered_f64:
    ucomisd xmm0, xmm1          /* sets flags; unordered if NaN */
    jp      .unordered          /* JP checks parity flag set: unordered */
    xor     eax, eax
    ret
.unordered:
    mov     eax, 1
    ret
```

Rust Test Harness for NaN Behavior

```
#[link(name = "myasm")]
extern "C" {
    fn is_unordered_f64(a: f64, b: f64) -> u32;
}

fn main() {
    let a = 1.0f64;
    let b = 2.0f64;
    let nan = f64::NAN;

    unsafe {
        println!("unordered(a,b)   = {}", is_unordered_f64(a, b));
        println!("unordered(nan,b) = {}", is_unordered_f64(nan, b));
        println!("unordered(a,nan) = {}", is_unordered_f64(a, nan));
    }

    // Demonstrate IEEE-754 NaN comparison properties in Rust:
    println!("nan == nan ? {}", nan == nan);
    println!("nan != nan ? {}", nan != nan);
    println!("nan <  1.0 ? {}", nan < 1.0);
    println!("nan >  1.0 ? {}", nan > 1.0);
}
```

3.4.5 ABI Pitfall Catalogue for Floating Boundaries

These are the most common real-world failure modes when passing floats across languages:

Pitfall 1: Wrong Calling Convention

Symptoms:

- returned value is NaN or a strange constant,

- parameters appear swapped or uninitialized,
- code works in one build mode and fails in another.

Cause:

- treating Windows x64 like System V,
- assuming more XMM argument registers than the ABI provides.

Pitfall 2: Mixing Integer and Float Registers Incorrectly

ABIs assign integer parameters and float parameters through separate register classes. If you incorrectly place float arguments in integer registers (or vice versa), the callee reads garbage.

Pitfall 3: Passing Aggregates with Floats Without `repr(C)`

Even if the bit layout seems identical, the ABI can choose different passing rules. Foundation rule:

- prefer passing scalar floats directly,
- if you must pass a struct, use `repr(C)` and keep it simple, and validate by disassembly.

Pitfall 4: Using `long double` Across the Boundary

Across platforms, `long double` can differ in size and calling convention. Foundation rule:

- avoid `long double` in cross-language APIs unless you fully control platform and toolchain.

3.5 Practical Exercises (Foundation Level)

1. Build a Rust test that prints `to_bits` for:

   - `+0.0` and `-0.0`
   - `1.0`, `0.5`, `2.0`
   - `INFINITY`, `NEG_INFINITY`
   - a quiet NaN created by `from_bits`

   Confirm sign, exponent, and fraction fields.

2. Implement in assembly:

   - `add_f64`
   - `mul_f64`
   - `is_unordered_f64`

   Validate from Rust on Windows using deterministic inputs including NaNs.

3. Demonstrate signed zero behavior:

   - compute `1.0 / +0.0` and `1.0 / -0.0` in Rust,
   - print results and bit patterns.

3.6 Professional Summary

IEEE-754 floating-point is a standardized, finite-precision system with special values and defined rounding. For assembly authors, correctness depends on:

- understanding layouts of binary32/binary64 and their special encodings,

- respecting ABI rules for float register passing and return values,
- treating NaN as unordered and handling comparisons accordingly,
- validating cross-language calls via bit inspection and deterministic tests.

This foundation is required before moving to:

- SIMD vector formats and instructions,
- floating exception environment control,
- performance and numerical stability engineering.

Endianness and Data Interpretation

4.1 Scope and Engineering Goal

Endianness is the byte ordering rule that defines how multi-byte values are stored in memory and transmitted over protocols. For assembly authors, endianness is not a "theory topic"; it is a primary source of real-world defects in:

- binary file parsers,
- network protocol implementations,
- serialization formats,
- cross-platform libraries,
- cryptographic and hash pipelines (bit-exact inputs),
- ABI boundaries when doing raw memory reinterpretation.

This chapter trains you to:

- reason precisely about little-endian vs big-endian layouts,
- implement correct host/network conversions,
- avoid the most common cross-platform data pitfalls,

- validate correctness with bit-exact Rust tests on Windows and with assembly inspection.

4.2 Little vs Big

4.2.1 Definition

Endianness describes the order of bytes of a multi-byte integer (or a multi-byte field) when stored at increasing addresses.
Let a 32-bit integer be:

$$X = \texttt{0x11223344}$$

This integer occupies 4 bytes. The question is: which byte appears at the lowest address?

Big-Endian (BE)

- Most significant byte stored at the lowest address.

Memory layout:

$$[\texttt{0x11}, \texttt{0x22}, \texttt{0x33}, \texttt{0x44}]$$

Little-Endian (LE)

- Least significant byte stored at the lowest address.

Memory layout:

$$[\texttt{0x44}, \texttt{0x33}, \texttt{0x22}, \texttt{0x11}]$$

4.2.2 The Engineer's Rule: Addresses Increase, Significance May Not

Many conceptual errors happen because programmers confuse:

- **byte significance** (MSB vs LSB),

- **address order** (low address vs high address),
- **human notation** (writing `0x11223344` left to right).

Endianness only tells you how the *bytes* are arranged in memory or on the wire. It does not change the meaning of arithmetic operations inside registers, where the CPU interprets a fixed-width binary value.

4.2.3 A Simple Table You Must Memorize

Value	Little-endian bytes in memory	Big-endian bytes in memory
`0x1234`	`34 12`	`12 34`
`0x11223344`	`44 33 22 11`	`11 22 33 44`
`0x0102030405060708`	`08 07 06 05 04 03 02 01`	`01 02 03 04 05 06 07 08`

4.2.4 Rust (Windows-Friendly) Example: Endianness by Construction

Rust provides exact conversions for integers:

- `to_le_bytes` / `from_le_bytes`
- `to_be_bytes` / `from_be_bytes`
- `to_ne_bytes` / `from_ne_bytes` (native-endian)

```
fn main() {
    let x: u32 = 0x1122_3344;

    let le = x.to_le_bytes();
    let be = x.to_be_bytes();
    let ne = x.to_ne_bytes();
```

```
    println!("x  = 0x{:08X}", x);
    println!("LE = {:02X?}", le);
    println!("BE = {:02X?}", be);
    println!("NE = {:02X?}", ne);

    let x1 = u32::from_le_bytes(le);
    let x2 = u32::from_be_bytes(be);
    let x3 = u32::from_ne_bytes(ne);

    println!("from LE = 0x{:08X}", x1);
    println!("from BE = 0x{:08X}", x2);
    println!("from NE = 0x{:08X}", x3);
}
```

4.2.5 Assembly Reality: Byte Order Appears in Loads and Stores

When you load a multi-byte value from memory into a register, the CPU uses the machine's endian rules to assemble bytes into the register value. In most modern deployments, x86-64 systems are little-endian in practice.
The correct mental model:

- **Registers hold numbers.**
- **Memory holds bytes.**
- **Endianness is the mapping between them.**

4.2.6 GAS Assembly (Intel Syntax): Showing the Byte Layout

The following routine stores a 32-bit value to memory and then reads back each byte separately. This demonstrates the byte order you will observe in memory.

```
.intel_syntax noprefix
.text

/* void store_u32_and_read_bytes(u32 x, u8* out4)
   Inputs (Windows x64 style for integers/pointers):
     ecx = x
     rdx = out4 pointer (must have space for 4 bytes)
   Behavior:
     Stores x into [rdx..rdx+3] and then reads bytes back explicitly.
*/
.global store_u32_and_read_bytes
store_u32_and_read_bytes:
    mov     dword ptr [rdx], ecx     /* store x as a 32-bit word */

    mov     al, byte ptr [rdx+0]     /* read byte0 */
    mov     byte ptr [rdx+10], al

    mov     al, byte ptr [rdx+1]     /* read byte1 */
    mov     byte ptr [rdx+1], al

    mov     al, byte ptr [rdx+2]     /* read byte2 */
    mov     byte ptr [rdx+2], al

    mov     al, byte ptr [rdx+3]     /* read byte3 */
    mov     byte ptr [rdx+3], al

    ret
```

4.3 Network Order

4.3.1 What "Network Byte Order" Means

In the Internet protocol ecosystem, "network byte order" for multi-byte integers is defined as **big-endian**. This means:

- when a protocol field is a 16-bit or 32-bit integer on the wire, the most significant byte is transmitted first,
- the receiver must convert from network order to host order.

Network order is a protocol contract, not a CPU property. Therefore:

- a little-endian host must swap multi-byte fields when reading/writing network values,
- a big-endian host can often treat network fields as native, but should still use conversion functions for clarity and portability.

4.3.2 Conversion Discipline: Host $\leftrightarrow$ Network

Classic socket APIs provide conversion functions (historically for 16- and 32-bit types). The conceptual mapping is:

Direction	Meaning
Host to Network	Convert native-endian integer to big-endian byte order
Network to Host	Convert big-endian integer to native-endian

Even if your platform is almost always little-endian today, do not hardcode assumptions into protocol code.

4.3.3 Rust Implementation: Network Order Encode/Decode

This is a strict and portable pattern for encoding and decoding protocol fields.

```rust
fn write_u16_be(buf: &mut Vec<u8>, x: u16) {
    buf.extend_from_slice(&x.to_be_bytes());
}

fn write_u32_be(buf: &mut Vec<u8>, x: u32) {
    buf.extend_from_slice(&x.to_be_bytes());
}

fn read_u16_be(buf: &[u8], off: usize) -> Option<u16> {
    if off + 2 <= buf.len() {
        Some(u16::from_be_bytes([buf[off], buf[off + 1]]))
    } else {
        None
    }
}

fn read_u32_be(buf: &[u8], off: usize) -> Option<u32> {
    if off + 4 <= buf.len() {
        Some(u32::from_be_bytes([buf[off], buf[off + 1], buf[off + 2], buf[off + 3]]))
    } else {
        None
    }
}

fn main() {
    let mut packet = Vec::new();

    // Example protocol header:
    //   u16 magic
    //   u16 version
```

```
    //   u32 payload_len
    write_u16_be(&mut packet, 0xABCD);
    write_u16_be(&mut packet, 0x0001);
    write_u32_be(&mut packet, 0x0000_0010);

    println!("packet bytes = {:02X?}", packet);

    let magic = read_u16_be(&packet, 0).unwrap();
    let ver   = read_u16_be(&packet, 2).unwrap();
    let len   = read_u32_be(&packet, 4).unwrap();

    println!("magic=0x{:04X} ver={} len={}", magic, ver, len);
}
```

4.3.4 Assembly View: Byte Swap as a Primitive

When implementing network protocols in assembly on little-endian machines, you often need byte swapping. On x86-64, the bswap instruction swaps byte order in a register.

GAS Assembly (Intel Syntax): Network Conversions by `bswap`

```
.intel_syntax noprefix
.text

/* u32 ntohl_u32(u32 x_net)
   Input:
     ecx = x_net (big-endian bytes interpreted as a u32 in memory order)
   Output:
     eax = host u32 (native-endian), assuming host is little-endian
*/
.global ntohl_u32
ntohl_u32:
    mov     eax, ecx
```

```
    bswap   eax                 /* reverse byte order: BE <-> LE */
    ret

/* u16 ntohs_u16(u16 x_net)
   Input:
     cx = x_net
   Output:
     ax = host u16
*/
.global ntohs_u16
ntohs_u16:
    mov     ax, cx
    xchg    al, ah              /* swap two bytes */
    ret
```

4.3.5 Rust Calling Assembly Conversions (Windows-Friendly)

```
#[link(name = "myasm")]
extern "C" {
    fn ntohl_u32(x: u32) -> u32;
    fn ntohs_u16(x: u16) -> u16;
}

fn main() {
    // Simulate "network order integers" represented as numeric values constructed from BE
    ↪  bytes.
    let net_u32 = u32::from_be_bytes([0x11, 0x22, 0x33, 0x44]);
    let net_u16 = u16::from_be_bytes([0x12, 0x34]);

    let host_u32 = unsafe { ntohl_u32(net_u32) };
    let host_u16 = unsafe { ntohs_u16(net_u16) };

    println!("net_u32 (as number) = 0x{:08X}", net_u32);
```

```
    println!("host_u32            = 0x{:08X}", host_u32);

    println!("net_u16 (as number) = 0x{:04X}", net_u16);
    println!("host_u16            = 0x{:04X}", host_u16);

    // Cross-check with Rust conversions:
    let host_u32_rust = u32::from_be_bytes(net_u32.to_ne_bytes()); // illustrates
    ↪  conceptually; prefer direct be_bytes in real code
    let host_u16_rust = u16::from_be_bytes(net_u16.to_ne_bytes());
    println!("host_u32_rust       = 0x{:08X}", host_u32_rust);
    println!("host_u16_rust       = 0x{:04X}", host_u16_rust);
}
```

4.4 Cross-Platform Data Pitfalls

Endianness problems usually appear in the following engineering scenarios.

4.4.1 Pitfall 1: Reinterpreting Struct Memory as a Wire Format

A common mistake is to send a struct by copying its raw bytes, assuming the receiver will interpret it the same way. This fails because:

- endianness differs across machines,
- alignment and padding differ across ABIs and compilers,
- integer sizes can differ (`usize`, `long`),
- floating-point edge cases (NaN payloads, signaling NaNs) are not guaranteed stable across environments,
- some fields require network order (big-endian) regardless of host.

Correct rule:

- Define the wire format explicitly in bytes.
- Serialize each field using explicit endianness conversions.

4.4.2 Pitfall 2: Confusing "Byte Order" with "Bit Order"

Some low-level protocols define bit-level transmission order (LSB-first vs MSB-first) within each byte. This is a different concept than byte endianness. You must treat them separately:

- **byte order** = ordering of bytes in multi-byte integers,
- **bit order** = ordering of bits in serialized transmission.

4.4.3 Pitfall 3: Mixed-Endian File Formats

Some file formats contain fields in different endianness (or an endianness marker). Engineers must:

- detect the file's declared endianness,
- decode each field accordingly,
- never assume the host endianness matches the file's.

4.4.4 Pitfall 4: Using Native Types in On-Disk Formats

Avoid storing:

- `usize` (varies by 32/64-bit),
- `isize`,

- platform `long`,
- raw pointers,
- platform-dependent `bool` representations.

Correct rule:

- use fixed-width integers (`u16`, `u32`, `u64`),
- specify endianness explicitly (BE or LE),
- include versioning fields in the format.

4.4.5 Pitfall 5: Casting Bytes to Integers with Unsafe Reinterpretation

Reinterpreting a byte slice as an integer pointer (or using transmute-like operations) is fragile because:

- it assumes alignment,
- it assumes endianness,
- it risks undefined behavior in many languages,
- it breaks under different compiler/CPU constraints.

Correct rule:

- decode using `from_le_bytes`/`from_be_bytes`,
- validate bounds before reading.

4.4.6 Rust Example: A Safe, Extensible Reader for BE/LE

This reader demonstrates production-grade parsing discipline:

- explicit endianness,
- bounds checking,
- predictable behavior across platforms and Windows builds.

```rust
#[derive(Copy, Clone)]
enum Endian {
    Little,
    Big,
}

struct Reader<'a> {
    data: &'a [u8],
    pos: usize,
    e: Endian,
}

impl<'a> Reader<'a> {
    fn new(data: &'a [u8], e: Endian) -> Self {
        Self { data, pos: 0, e }
    }

    fn remaining(&self) -> usize {
        self.data.len().saturating_sub(self.pos)
    }

    fn take(&mut self, n: usize) -> Option<&'a [u8]> {
        if self.pos + n <= self.data.len() {
            let s = &self.data[self.pos..self.pos + n];
```

```
            self.pos += n;
            Some(s)
        } else {
            None
        }
    }

    fn u16(&mut self) -> Option<u16> {
        let b = self.take(2)?;
        let x = match self.e {
            Endian::Little => u16::from_le_bytes([b[0], b[1]]),
            Endian::Big => u16::from_be_bytes([b[0], b[1]]),
        };
        Some(x)
    }

    fn u32(&mut self) -> Option<u32> {
        let b = self.take(4)?;
        let x = match self.e {
            Endian::Little => u32::from_le_bytes([b[0], b[1], b[2], b[3]]),
            Endian::Big => u32::from_be_bytes([b[0], b[1], b[2], b[3]]),
        };
        Some(x)
    }

    fn u64(&mut self) -> Option<u64> {
        let b = self.take(8)?;
        let x = match self.e {
            Endian::Little => u64::from_le_bytes([b[0], b[1], b[2], b[3], b[4], b[5], b[6],
            ↪  b[7]]),
            Endian::Big => u64::from_be_bytes([b[0], b[1], b[2], b[3], b[4], b[5], b[6],
            ↪  b[7]]),
        };
```

```
        Some(x)
    }
}

fn main() {
    // Example binary record (big-endian):
    // magic: u16 (0xCAFE)
    // version: u16 (2)
    // size: u32 (16)
    // id: u64 (0x0102030405060708)
    let record = [
        0xCA, 0xFE,
        0x00, 0x02,
        0x00, 0x00, 0x00, 0x10,
        0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08
    ];

    let mut r = Reader::new(&record, Endian::Big);
    let magic = r.u16().unwrap();
    let ver = r.u16().unwrap();
    let size = r.u32().unwrap();
    let id = r.u64().unwrap();

    println!("magic=0x{:04X} ver={} size={} id=0x{:016X}", magic, ver, size, id);
    println!("remaining bytes = {}", r.remaining());
}
```

4.4.7 Pitfall 6: Endianness and Floating-Point Reinterpretation

IEEE-754 defines the meaning of bit patterns for floats, but when you store floats to bytes, endianness controls byte order. If you transmit float values in a protocol:

- you must define a canonical byte order (commonly big-endian network order or explicitly

little-endian),

- you should transmit floats as integer bit patterns with explicit endianness conversion, then reconstruct on the other side.

4.4.8 Rust Example: Encoding f32/f64 as Network-Order Bytes

```rust
fn f32_to_be_bytes(x: f32) -> [u8; 4] {
    let bits = x.to_bits();
    bits.to_be_bytes()
}

fn f32_from_be_bytes(b: [u8; 4]) -> f32 {
    let bits = u32::from_be_bytes(b);
    f32::from_bits(bits)
}

fn f64_to_be_bytes(x: f64) -> [u8; 8] {
    let bits = x.to_bits();
    bits.to_be_bytes()
}

fn f64_from_be_bytes(b: [u8; 8]) -> f64 {
    let bits = u64::from_be_bytes(b);
    f64::from_bits(bits)
}

fn main() {
    let x = -13.5f32;
    let b = f32_to_be_bytes(x);
    let y = f32_from_be_bytes(b);

    println!("x = {} bytes={:02X?} y = {}", x, b, y);
```

```
    let a = f64::NAN;
    let bb = f64_to_be_bytes(a);
    let c = f64_from_be_bytes(bb);

    println!("a is NaN? {}  bytes={:02X?}  c is NaN? {}", a.is_nan(), bb, c.is_nan());
}
```

4.5 Practical Exercises (Foundation Level)

1. Confirm your host endianness observationally:
    - store `0x11223344` to memory,
    - print bytes in increasing addresses,
    - verify the observed order.
2. Implement a mini-protocol encoder/decoder:
    - header fields: `u16`, `u16`, `u32` in network order,
    - payload length must match actual payload size,
    - reject packets that are too short.
3. Build cross-check tests:
    - decode using your Reader,
    - decode using direct `from_be_bytes` calls,
    - assert identical results for multiple test vectors.
4. Extend the Reader with signed integers:
    - decode `i16`, `i32`, `i64` via unsigned decode plus two's complement reinterpretation,

- verify negative values decode correctly under both endian modes.

4.6 Professional Summary

Endianness is the byte-order contract between registers, memory, and protocols. Professional correctness requires:

- explicit byte-order conversions for every multi-byte field in external formats,
- never serializing raw structs as wire formats,
- avoiding platform-dependent types and pointer-based formats,
- decoding bytes using explicit BE/LE functions with bounds checking,
- treating network order as big-endian by protocol definition.

Mastering endianness eliminates an entire class of cross-platform bugs that appear only after deployment.

Alignment and Data Layout

5.1 Scope and Engineering Goal

Alignment and data layout are among the most common root causes of:

- rare crashes that appear only under optimization,
- wrong values when crossing ABI boundaries,
- unexpected slowdowns in tight loops,
- faults triggered by certain SIMD instructions,
- incorrect atomic behavior or torn reads on some architectures.

This chapter provides a disciplined, cross-platform mental model centered on three topics:

- natural alignment,
- structure padding,
- misaligned penalties.

The practical objective is simple:

- If you can compute object layout by hand and predict alignment requirements, you can write correct assembly that interoperates safely with C and Rust and that does not break when the compiler or OS changes optimization or ABI details.

5.2 Natural Alignment

5.2.1 Definition

An object of size N bytes typically has a **natural alignment** of A bytes, where A is commonly a power of two and usually:

$$A \leq N$$

Natural alignment means the object should reside at an address divisible by A:

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

This is not a stylistic preference. It is a performance and, on some architectures, correctness requirement.

5.2.2 What Determines Alignment

Alignment constraints come from:

- the ISA (how loads/stores are implemented),
- the ABI (how stack frames and aggregates are laid out),
- the compiler (layout strategy and optimization),
- the instruction class (scalar vs SIMD vs atomic).

5.2.3 Typical Alignment Expectations (64-bit Systems)

The exact values are ABI/toolchain-defined, but on mainstream 64-bit environments, typical expectations are:

Type	Size (bytes)	Typical alignment (bytes)
u8/i8	1	1
u16/i16	2	2
u32/i32/f32	4	4
u64/i64/f64	8	8
Pointer (usize)	8	8
u128	16	16 (often)
SIMD 128-bit vector	16	16 (commonly required by some instructions)

Engineering rule:

- **Never assume** a type's alignment by memory. Measure it in your toolchain environment and design your ABI boundary using stable, explicit layouts.

5.2.4 Rust (Windows-Friendly) Example: Observing Size and Alignment

Rust provides stable introspection using `std::mem::size_of` and `std::mem::align_of`. This is ideal for verifying your assumptions before writing assembly offsets.

```rust
use std::mem::{size_of, align_of};

#[repr(C)]
struct S1 {
    a: u8,
    b: u32,
    c: u16,
}

fn main() {
    println!("u8   : size={} align={}", size_of::<u8>(), align_of::<u8>());
```

```
    println!("u16  : size={} align={}", size_of::<u16>(), align_of::<u16>());
    println!("u32  : size={} align={}", size_of::<u32>(), align_of::<u32>());
    println!("u64  : size={} align={}", size_of::<u64>(), align_of::<u64>());
    println!("f32  : size={} align={}", size_of::<f32>(), align_of::<f32>());
    println!("f64  : size={} align={}", size_of::<f64>(), align_of::<f64>());

    println!("S1   : size={} align={}", size_of::<S1>(), align_of::<S1>());

    // Print field offsets without unstable features:
    let x = S1 { a: 1, b: 2, c: 3 };
    let base = (&x as *const S1 as usize);
    let off_a = (&x.a as *const u8 as usize) - base;
    let off_b = (&x.b as *const u32 as usize) - base;
    let off_c = (&x.c as *const u16 as usize) - base;

    println!("offsets: a={} b={} c={}", off_a, off_b, off_c);
}
```

5.2.5 Alignment in Assembly Terms

In assembly, alignment is primarily about whether you are allowed to do:

- a wide load/store at some address,
- a SIMD load/store using an instruction that assumes alignment,
- an atomic read-modify-write spanning cache-line boundaries.

Correctness rule:

- If you write an assembly routine that reads a `u64` from a pointer, you must assume that pointer is aligned to at least 8 unless the API explicitly says otherwise. If the API allows unaligned pointers, you must use unaligned-safe logic or instructions.

5.3 Structure Padding

5.3.1 Why Padding Exists

Compilers insert **padding bytes** inside and after structures to satisfy alignment constraints for each field. Padding is not wasted; it is the cost of enabling efficient aligned access for members. A structure has:

- an alignment equal to the maximum alignment of its fields (plus any explicit alignment attributes),
- a size that is a multiple of its alignment (often requiring tail padding).

5.3.2 Hand-Computing Layout: A Canonical Example

Consider:

```
#[repr(C)]
struct A {
    a: u8,
    b: u32,
    c: u16,
}
```

Assume typical alignments: `u8`=1, `u32`=4, `u16`=2.

Step-by-step layout

- `a` at offset 0 (size 1)
- next field `b` needs 4-byte alignment, so compiler inserts 3 padding bytes: offsets 1–3
- `b` at offset 4 (size 4): occupies 4–7

- next field c needs 2-byte alignment; offset 8 is already divisible by 2
- c at offset 8 (size 2): occupies 8–9
- structure alignment is 4 (max of fields)
- size must be multiple of 4: currently 10, so add tail padding to reach 12

Result:

- **size** = 12
- **alignment** = 4
- **offsets:** a=0, b=4, c=8

5.3.3 Field Ordering Rule: Reduce Padding by Reordering

Padding depends on field order. Sorting fields by decreasing alignment often reduces size. Example:

```
#[repr(C)]
struct B {
    b: u32,
    c: u16,
    a: u8,
}
```

Typical layout:

- b at 0..3
- c at 4..5
- a at 6

- tail padding to multiple of 4: size becomes 8

This is a dramatic reduction from 12 to 8 on common ABIs.

5.3.4 Rust Layout Contracts: `repr(C)` vs Default

In Rust:

- **Default** (`repr(Rust)`): layout is not a stable ABI contract across compilers/versions.
- **`repr(C)`**: layout matches C ABI rules for structs on the target platform and is the correct choice for FFI and assembly offsets.

Engineering rule:

- If assembly code uses fixed offsets into a Rust struct, the struct must be `repr(C)` (or an explicitly documented stable layout) and must not be changed without updating assembly.

5.3.5 Packed Structures: When "Fixing Size" Breaks Correctness

Rust supports `repr(packed)`. This removes padding and can reduce size, but it also:

- can produce misaligned fields,
- can make taking references to fields unsafe or invalid,
- can introduce faults on architectures that do not tolerate misaligned accesses,
- can degrade performance substantially on architectures that emulate misaligned loads/stores.

Packed is appropriate only when:

- you are mapping a byte-exact external format (wire protocol, file format),

- you access fields using unaligned-safe loads (`read_unaligned`) or byte decoding,
- you treat the structure as a byte container, not a normal in-memory object model.

5.3.6 Rust Example: `repr(C)` vs `repr(packed)` and Safe Unaligned Reads

```rust
use std::mem::{size_of, align_of};
use std::ptr;

#[repr(C)]
struct CNormal {
    a: u8,
    b: u32,
}

#[repr(C, packed)]
struct CPacked {
    a: u8,
    b: u32, // likely unaligned
}

fn main() {
    println!("CNormal: size={} align={}", size_of::<CNormal>(), align_of::<CNormal>());
    println!("CPacked: size={} align={}", size_of::<CPacked>(), align_of::<CPacked>());

    let p = CPacked { a: 0x11, b: 0xAABB_CCDD };

    // Direct reference to p.b is not safe in general because it may be unaligned.
    // Correct approach: read_unaligned from a raw pointer.
    let b_ptr = (&p as *const CPacked as *const u8).wrapping_add(1) as *const u32;
    let b_val = unsafe { ptr::read_unaligned(b_ptr) };

    println!("packed b (read_unaligned) = 0x{:08X}", b_val);
```

```
}
```

5.4 Misaligned Penalties

5.4.1 What “Misaligned” Means

A load/store is misaligned if the address is not a multiple of the natural alignment the operation expects. Example: reading a 64-bit value from an address not divisible by 8.

5.4.2 The Three Possible Outcomes Across Architectures

Depending on ISA, instruction, and memory type, misaligned access can:

1. succeed but run slower,
2. succeed but require multiple micro-operations or special handling,
3. raise a fault/trap (immediate crash unless handled).

5.4.3 x86-64 Reality (Practical)

On x86-64:

- most scalar integer loads/stores can tolerate misalignment, but may incur penalties (especially when crossing cache-line boundaries),
- some SIMD instructions assume alignment and can fault if misaligned (example: aligned-move instructions),
- atomicity guarantees become subtle if an access crosses certain boundaries; do not build lock-free logic on misaligned objects.

5.4.4 AArch64 and RISC-V Reality (Practical)

On AArch64 and RISC-V:

- misaligned access behavior is more configuration- and instruction-dependent,
- some environments trap misaligned accesses,
- some emulate misaligned access in software at significant cost,
- many OS kernels and low-level runtimes assume correct alignment for performance and correctness.

Engineering rule:

- If your library targets multiple architectures, design APIs that either require alignment or explicitly perform unaligned-safe decoding.

5.4.5 SIMD Alignment Fault Demonstration (x86-64)

Some instructions require aligned memory operands. A classic example is using an aligned 128-bit load. If the pointer is not 16-byte aligned, the instruction can fault.
The safe alternative is using an unaligned load instruction when alignment is not guaranteed.

GAS Assembly (Intel Syntax): Aligned vs Unaligned Loads

```
.intel_syntax noprefix
.text

/* void load128_aligned(const void* p)
   Input:
     rcx = p (Windows x64 first pointer arg)
   Behavior:
```

```
    Uses an aligned 128-bit load. If p is not 16-byte aligned, this can fault.
*/
.global load128_aligned
load128_aligned:
    movaps  xmm0, xmmword ptr [rcx]   /* requires 16-byte alignment */
    ret

/* void load128_unaligned(const void* p)
   Input:
     rcx = p
   Behavior:
     Uses an unaligned 128-bit load. Works with any alignment, potentially slower.
*/
.global load128_unaligned
load128_unaligned:
    movups  xmm0, xmmword ptr [rcx]   /* works for unaligned addresses */
    ret
```

Rust (Windows) Test Harness: Creating Aligned and Misaligned Pointers

This harness constructs:

- a 16-byte aligned pointer,
- a deliberately misaligned pointer (offset by 1),

then calls the two routines.

```
use std::alloc::{alloc, dealloc, Layout};
use std::ptr;

#[link(name = "myasm")]
extern "C" {
    fn load128_aligned(p: *const u8);
```

```
    fn load128_unaligned(p: *const u8);
}

fn main() {
    unsafe {
        // Allocate 64 bytes aligned to 16.
        let layout = Layout::from_size_align(64, 16).unwrap();
        let p = alloc(layout);
        if p.is_null() {
            panic!("alloc failed");
        }

        // Fill with data.
        for i in 0..64 {
            ptr::write(p.add(i), i as u8);
        }

        let aligned = p as *const u8;         // 16-byte aligned by construction
        let misaligned = p.add(1) as *const u8; // intentionally misaligned

        // Unaligned load should always work.
        load128_unaligned(aligned);
        load128_unaligned(misaligned);

        // Aligned load should work only for aligned pointer.
        load128_aligned(aligned);

        // Calling aligned load with misaligned pointer may fault.
        // Uncomment to observe the fault behavior in a controlled environment.
        // load128_aligned(misaligned);

        dealloc(p, layout);
    }
```

```
}
```

5.4.6 Scalar Unaligned Loads: Correctness vs Performance

Even when scalar unaligned loads succeed, they can be slower due to:

- extra internal micro-operations,
- crossing cache-line boundaries,
- crossing page boundaries (worst case),
- interference with vectorization and prefetch patterns.

Engineering guideline:

- Avoid designing hot data structures that force frequent misaligned loads.
- When parsing external byte streams, decode using byte-wise or explicit endian conversions, not by casting to misaligned typed pointers.

5.4.7 Rust Example: `read_unaligned` for Portable Correctness

When you must read a numeric value from an arbitrary byte address (e.g., protocol parsing), use unaligned-safe reads or byte decoding.

```
use std::ptr;

fn read_u32_le_unaligned(p: *const u8) -> u32 {
    unsafe {
        let raw = ptr::read_unaligned(p as *const u32);
        u32::from_le(raw)
    }
}
```

```
fn main() {
    let buf = [0x44u8, 0x33, 0x22, 0x11, 0xAA, 0xBB];
    let p0 = buf.as_ptr();       // aligned is not guaranteed for u32, but safe
    ↪  read_unaligned handles it
    let p1 = unsafe { p0.add(1) }; // definitely misaligned for u32 in most cases

    let a = read_u32_le_unaligned(p0);
    let b = read_u32_le_unaligned(p1);

    println!("read@0 = 0x{:08X}", a); // expects 0x11223344 on little-endian interpretation
    println!("read@1 = 0x{:08X}", b); // reads next 4 bytes starting at buf[1]
}
```

5.4.8 ABI Boundary Warning: Alignment is Part of the Contract

At ABI boundaries:

- stack alignment is mandatory (commonly 16-byte aligned at call boundaries on 64-bit ABIs),
- parameters passed by pointer often have implied alignment requirements,
- aggregates passed by value assume standard alignment and padding rules.

If you violate alignment assumptions:

- you may crash inside unrelated code (because the misalignment is detected later),
- you may break SIMD code that assumes alignment,
- you may silently degrade performance in a way that is hard to diagnose.

5.5 Professional Checklist

Before writing assembly that touches memory layouts:

- Confirm sizes and alignments in your target toolchain.
- Use `repr(C)` for any struct whose layout is used by assembly or FFI.
- Compute field offsets explicitly and validate them with a small harness.
- Avoid `repr(packed)` unless mapping an external byte format; then use `read_unaligned` or byte decoding.
- When alignment is not guaranteed, use unaligned-safe instructions (`movups` vs `movaps`) or decode bytes manually.
- Keep stack alignment correct at every call boundary.

5.6 Exercises (Foundation Level)

1. Define two structs with the same fields in different orders. Measure size, alignment, and offsets. Explain padding bytes.
2. Create a packed struct representing a protocol header and implement decoding using explicit BE/LE byte functions. Do not take references to packed fields.
3. Implement two assembly functions:
   - one that loads 128-bit from memory with aligned requirement,
   - one that loads 128-bit from memory without alignment requirement.

   Call them from Rust on Windows with aligned and misaligned pointers and observe behavior.

4. Build a parsing routine that reads `u32` values from a byte buffer using both:
   - unsafe pointer cast (incorrect approach),
   - `read_unaligned` or `from_le_bytes` (correct approach).

   Explain which is portable and why.

5.7 Professional Summary

Alignment and padding are not compiler trivia; they are machine contracts. Professional assembly authors must treat:

- natural alignment as a correctness/performance requirement,
- structure padding as a predictable consequence of field alignment,
- misaligned access as an operation that may be slower or may fault depending on instruction and architecture.

With the discipline in this chapter, you can safely compute layouts, write ABI-correct assembly, and design cross-platform binary formats without hidden assumptions.

Part II

INSTRUCTION SET FOUNDATIONS

Instruction Encoding Concepts

6.1 Variable vs Fixed Length

6.1.1 The Two Families

Modern mainstream ISAs fall into two broad encoding families:

Variable-Length Encodings

- Instruction length varies (within an ISA-defined limit).
- Decoder must find boundaries dynamically.
- Backward compatibility and code density are often major goals.

Canonical example: x86/x86-64 (legacy + modern prefixes).

Fixed-Length Encodings

- Instruction length is constant (usually 32 bits).
- Instruction boundaries are trivial: every 4 bytes is an instruction.
- Decoder can be simpler and more regular.

Canonical example: AArch64 (ARM64) uses fixed 32-bit instruction words aligned on 4-byte boundaries.

6.1.2 Why This Choice Matters

Encoding style affects:

- **Decode complexity:** finding boundaries and extracting fields,
- **Code density:** how many bytes represent typical work,
- **Pipeline front-end:** fetch bandwidth and decode throughput,
- **Tooling complexity:** disassembly correctness, patching, relocation,
- **Security analysis:** gadget discovery, instruction boundary ambiguity,
- **Performance**: code size affects i-cache footprint; decode affects front-end bottlenecks.

6.1.3 The Practical Tradeoff

Variable-Length Strengths

- high code density for common operations,
- rich addressing modes encoded compactly,
- backward compatibility through layered prefix schemes.

Variable-Length Costs

- complicated decoding logic,
- instruction boundary ambiguity without decoding,

- multiple encoding schemes (legacy prefixes, REX, VEX, EVEX),
- patching requires careful length preservation or rewriting.

Fixed-Length Strengths

- simple boundaries,
- regular field extraction,
- predictable alignment,
- typically easier for assemblers/disassemblers and JIT emitters.

Fixed-Length Costs

- code density can be worse for small operations unless mitigated,
- some immediates and addressing forms require multiple instructions (materializing constants),
- extensions can consume opcode space (requires careful planning).

6.2 x86 vs ARM vs RISC-V Encoding Philosophy

6.2.1 x86-64 Philosophy: Backward Compatibility and Density Through Prefix Layers

x86-64 encoding is historically layered. A practical mental model:

High-Level Structure (Conceptual)

An x86-64 instruction is composed of components in order (some optional):

- legacy prefixes (0–4 bytes),
- optional REX prefix in 64-bit mode (1 byte),
- opcode bytes (1–3+),
- optional ModR/M (1 byte) for many register/memory forms,
- optional SIB (1 byte) for certain addressing forms,
- optional displacement (1/2/4/8 bytes),
- optional immediate (1/2/4/8 bytes).

There is an ISA-defined maximum instruction length (15 bytes). This cap exists to bound decode work and prevent pathological sequences.

Why ModR/M and SIB Exist

Instead of dedicating opcodes for every addressing mode, x86 uses:

- ModR/M to encode register-vs-memory and select registers/opcode extensions,
- SIB to encode scaled index addressing (base + index*scale + displacement).

This yields compact encodings for complex addressing, but forces the decoder to interpret multiple interacting fields.

Concrete x86-64 Encodings You Should Memorize

These examples illustrate variable-length reality. Bytes shown in increasing memory order.

```
.intel_syntax noprefix
/* Example 1: mov eax, 0x11223344
   Encoding: B8 + imm32
   Bytes: B8 44 33 22 11
*/
mov eax, 0x11223344

/* Example 2: add rax, rbx
   Encoding: REX.W + 01 /r
   Bytes: 48 01 D8
   Breakdown:
     48 = REX.W
     01 = opcode for add r/m64, r64
     D8 = ModR/M (mod=11 reg=011 r/m=000) => rax = rax + rbx
*/
add rax, rbx

/* Example 3: mov rax, [rcx+8]
   Encoding: REX.W + 8B /r + ModR/M + disp8
   Bytes (common form): 48 8B 41 08
   Breakdown:
     48 = REX.W
     8B = opcode mov r64, r/m64
     41 = ModR/M (mod=01 reg=000 r/m=001) => rax, [rcx + disp8]
     08 = disp8
*/
mov rax, qword ptr [rcx+8]
```

Modern Prefix Families: REX, VEX, EVEX

In 64-bit mode, REX extends register encodings and selects 64-bit operand size. VEX/EVEX introduce compact ways to encode vector instructions, operand count expansion, and opcode-map selection. Engineering consequence:

- decoding must understand multiple prefix regimes,
- binary rewriting must preserve correct prefix semantics,
- tools must not treat prefixes as "optional decoration".

6.2.2 AArch64 (ARM64) Philosophy: Fixed 32-bit Words, Regular Fields

AArch64 uses fixed-size 32-bit instruction words aligned on 4-byte boundaries. Practical consequences:

- instruction boundaries are trivial,
- decoding is uniform bitfield extraction,
- many patterns encode class + operation + registers + immediate fields in stable locations.

Example: AArch64 `ADD X0, X1, X2`

One common encoding of:

```
ADD X0, X1, X2
```

is the 32-bit word:

```
0x8B020020
```

In memory on a little-endian machine, the bytes appear as:

```
20 00 02 8B
```

AArch64 Constant Materialization as an Encoding Strategy

Because the instruction is only 32 bits, not every 64-bit constant can fit directly as an immediate. AArch64 provides a family of `MOVZ/MOVK/MOVN` that build constants in 16-bit chunks with shifts. A known byte sequence:

```
80 46 C2 D2
```

corresponds to a 32-bit word `0xD2C24680` (little-endian in memory) and is commonly disassembled as a move-wide immediate form that constructs a shifted 16-bit constant.
Engineering consequence:

- fixed encoding pushes complexity into *instruction sequences*,
- assemblers and compilers choose canonical encodings,
- disassemblers may choose different but equivalent textual forms.

6.2.3 RISC-V Philosophy: Simple Base 32-bit Encoding With Optional Compression

RISC-V base integer instructions are 32 bits and use a small set of instruction formats (R/I/S/B/U/J). To improve code density, an optional compressed extension introduces 16-bit encodings for common operations. RISC-V also permits longer instruction lengths in the overall design space (future or specialized extensions).
Engineering consequence:

- base decode is simple and regular,
- compressed instructions introduce mixed 16/32 boundaries (still designed to be decodable),
- the encoding philosophy strongly favors simplicity and orthogonality over dense “one instruction does everything” addressing.

6.3 Encoding Examples and Field Extraction

This section gives extensive, bit-exact examples for RISC-V and AArch64 and practical component breakdown for x86-64.

6.3.1 RISC-V: R-Type and I-Type With Real Machine Words

R-Type Example: `add x5, x6, x7`

RISC-V R-type fields:

- `funct7` (7 bits)
- `rs2` (5 bits)
- `rs1` (5 bits)
- `funct3` (3 bits)
- `rd` (5 bits)
- `opcode` (7 bits)

A standard encoding for `add x5, x6, x7` is:

```
0x007302B3
```

Bytes in little-endian memory:

```
B3 02 73 00
```

I-Type Example: `addi x1, x0, 1`

I-type fields:

- `imm[11:0]` (12-bit signed immediate)
- `rs1`
- `funct3`
- `rd`
- `opcode`

A standard encoding for `addi x1, x0, 1` is:

```
0x00100093
```

Bytes in little-endian memory:

```
93 00 10 00
```

6.3.2 AArch64: Fixed 32-bit Words and Byte Order

AArch64 instructions are 32-bit words. Endianness affects memory byte order, not the logical bit positions in the word when you decode it as a 32-bit integer.
Example word:

```
0x8B020020
```

Memory bytes on little-endian:

```
20 00 02 8B
```

6.3.3 x86-64: Variable-Length Components and Practical Decoding Mindset

Do not attempt to decode x86 by guessing length from opcode alone. Instead:

- consume prefixes (legacy/REX or VEX/EVEX),
- read opcode map and opcode,
- decide whether ModR/M is required,
- if ModR/M indicates SIB, parse it,
- read displacement and immediate as required by the form.

Professional rule for tooling and patching:

- never change instruction bytes without re-decoding from the start of the instruction boundary,
- never assume “this instruction is 2 bytes” unless you decoded it.

6.4 Rust (Windows-Friendly) Laboratory: Decoding Real Encodings

This laboratory is designed for Windows builds, does not require external tooling, and gives deterministic outputs.

6.4.1 Rust: Helper Functions for Bitfield Extraction

```rust
fn bits(x: u32, lo: u32, hi: u32) -> u32 {
    // Extract inclusive bit range [hi:lo], 0 <= lo <= hi < 32
    let width = hi - lo + 1;
```

```
    (x >> lo) & ((1u32 << width) - 1)
}

fn sign_extend(value: u32, bits_width: u32) -> i32 {
    // Interpret 'value' as signed with bits_width bits, return i32
    let shift = 32 - bits_width;
    ((value << shift) as i32) >> shift
}
```

6.4.2 Rust: Decode RISC-V R-Type and I-Type

```
fn decode_riscv_r(inst: u32) {
    let opcode = bits(inst, 0, 6);
    let rd     = bits(inst, 7, 11);
    let funct3 = bits(inst, 12, 14);
    let rs1    = bits(inst, 15, 19);
    let rs2    = bits(inst, 20, 24);
    let funct7 = bits(inst, 25, 31);

    println!("RISC-V R-type inst=0x{inst:08X}");

    ↪  println!(" opcode=0b{opcode:07b} rd=x{rd} funct3=0b{funct3:03b} rs1=x{rs1} rs2=x{rs2} funct
    println!();
}

fn decode_riscv_i(inst: u32) {
    let opcode = bits(inst, 0, 6);
    let rd     = bits(inst, 7, 11);
    let funct3 = bits(inst, 12, 14);
    let rs1    = bits(inst, 15, 19);
    let imm12  = bits(inst, 20, 31);
    let imm_s  = sign_extend(imm12, 12);
```

```
    println!("RISC-V I-type inst=0x{inst:08X}");
    println!(" opcode=0b{opcode:07b} rd=x{rd} funct3=0b{funct3:03b}
    rs1=x{rs1} imm12=0x{imm12:03X} imm_signed={imm_s}");
    println!();
}

fn main() {
    // add x5, x6, x7  -> 0x007302B3
    decode_riscv_r(0x0073_02B3);

    // addi x1, x0, 1  -> 0x00100093
    decode_riscv_i(0x0010_0093);
}
```

6.4.3 Rust: Show AArch64 Word and Its Bytes

```
fn show_aarch64_word(word: u32) {
    let b = word.to_le_bytes();
    println!("AArch64 inst word = 0x{word:08X}");
    println!("bytes in memory (LE) = {:02X?}", b);
    println!();
}

fn main() {
    // Example: ADD X0, X1, X2 often encoded as 0x8B020020
    show_aarch64_word(0x8B02_0020);

    // Example word seen in move-wide immediate family
    show_aarch64_word(0xD2C2_4680);
}
```

6.5 Philosophy Summary: What the Encodings Optimize For

6.5.1 x86-64

- prioritizes backward compatibility and dense encodings for common operations,
- supports complex addressing forms compactly via ModR/M and SIB,
- accumulates complexity in decode logic and multiple prefix schemes,
- requires disciplined decoding in tooling and patching.

6.5.2 AArch64

- prioritizes regularity and decode simplicity through fixed 32-bit instruction words,
- places complexity into instruction selection and sequences (especially immediates),
- produces stable, uniform boundaries that simplify disassembly and rewriting.

6.5.3 RISC-V

- prioritizes a small, regular base encoding and clean format families,
- uses optional compression to recover code density while keeping base decode simple,
- keeps addressing and operations orthogonal (simplicity over “do-everything” opcodes),
- scales via extensions rather than ever-growing prefix stacks.

6.6 Exercises (Foundation Level)

1. For x86-64, take three instructions and write their byte encodings and component breakdown:
    - one immediate move,
    - one reg-reg ALU op,
    - one reg-memory op with displacement.
2. For RISC-V, compute by hand the machine code for `add x5, x6, x7` using the R-type field placements, then verify with the Rust decoder output.
3. For AArch64, take one known 32-bit instruction word and:
    - print its bytes in memory order,
    - confirm how endianness changes byte order but not the meaning of the decoded word.
4. Write a minimal "instruction stream scanner":
    - for AArch64: step by 4 bytes,
    - for RISC-V with compression: step by 2 or 4 based on low bits,
    - for x86: explain why you cannot reliably step without a full decoder.

6.7 Professional Summary

Instruction encoding concepts determine how you debug, patch, and build low-level tooling. Remember the engineering truths:

- Fixed-length ISAs make boundaries trivial and decoding regular.

- Variable-length ISAs trade decode complexity for density and historical compatibility.
- RISC-V explicitly separates a simple base from optional density extensions.
- For x86, never reason about instruction length without decoding.

This foundation prepares you for the next chapters: addressing modes, immediate encoding strategies, relocation constraints, and code-generation patterns across architectures.

Register Architectures Compared

7.1 Scope and Engineering Goal

Registers are the CPU's *working set*. If you master the register model of an architecture, you can:

- read and write assembly with confidence,
- understand compiler output,
- implement ABI-correct function calls,
- write correct prolog/epilog sequences,
- avoid catastrophic calling-convention bugs that look like random memory corruption.

This chapter compares:

- x86-64 registers,
- AArch64 registers,
- RV64 registers,
- volatile vs non-volatile registers (caller-saved vs callee-saved) as an ABI contract.

7.2 Concepts: Architectural vs ABI Roles

7.2.1 Architectural Registers

These are defined by the ISA:

- integer registers,
- floating/SIMD registers,
- special registers (PC/IP, SP, flags/status),
- system registers (privileged; out of scope here).

7.2.2 ABI Roles: Volatile vs Non-Volatile

ABIs classify registers by who must preserve them across a function call:

- **Volatile (caller-saved):** may be clobbered by the callee. If the caller needs them after the call, the caller must save them.
- **Non-volatile (callee-saved):** must be preserved by the callee if used. If the callee changes them, it must restore them before returning.

Engineering rule:

- The ISA defines *what* registers exist; the ABI defines *how* they must be used at function boundaries.

7.3 x86-64 Registers

x86-64 provides a dense register ecosystem with historical layering.

7.3.1 General-Purpose Integer Registers (GPRs)

In 64-bit mode, the core GPRs are:

- `RAX, RBX, RCX, RDX`
- `RSI, RDI`
- `RBP` (often frame pointer by convention)
- `RSP` (stack pointer)
- `R8–R15`

Subregister Views (Width Aliases)

Each GPR can be accessed with different widths. Example for `RAX`:

- `RAX` (64-bit)
- `EAX` (32-bit)
- `AX` (16-bit)
- `AL` (8-bit low)
- `AH` (8-bit high of `AX`, legacy; avoid mixing with certain encodings)

Critical x86-64 rule (foundational):

- Writing a 32-bit subregister (e.g., `EAX`) **zero-extends** into the full 64-bit register (`RAX`).

7.3.2 Instruction Pointer and Flags

- `RIP`: instruction pointer (not a general-purpose register; used implicitly by control flow).
- `RFLAGS`: status flags (CF, ZF, SF, OF, etc.).

7.3.3 Vector/SIMD Registers

Modern x86-64 uses XMM/YMM/ZMM families depending on enabled extensions:

- `XMM0–XMM15` (128-bit) are universally relevant for scalar floating ABI on x86-64.
- Wider `YMM/ZMM` expand the same logical register files (when supported), but ABI fundamentals still name XMM registers for argument passing.

7.3.4 x86-64 ABI Volatile vs Non-Volatile (Two Dominant 64-bit ABIs)

Because your encyclopedia is cross-platform, you must treat Windows x64 and System V AMD64 as distinct.

Windows x64 (Microsoft ABI) Summary

Non-volatile (callee-saved) GPRs:

- `RBX, RBP, RSI, RDI, R12, R13, R14, R15`
- `RSP` must be restored (stack pointer discipline)

Volatile (caller-saved) GPRs:

- `RAX, RCX, RDX, R8, R9, R10, R11`

Non-volatile XMM:

- `XMM6–XMM15`

Volatile XMM:

- `XMM0–XMM5`

System V AMD64 Summary

Non-volatile (callee-saved) GPRs:

- `RBX, RBP, R12, R13, R14, R15`
- `RSP` must be restored

Volatile (caller-saved) GPRs:

- `RAX, RCX, RDX, RSI, RDI, R8, R9, R10, R11`

Vector registers (common baseline view):

- `XMM0–XMM15` are caller-saved in typical System V usage.

7.3.5 x86-64 Windows Example: Callee-Saved Discipline in Assembly

This example follows Windows x64 register preservation rules. It uses `RBX` and `XMM6` (non-volatile), therefore it saves/restores them.

```
.intel_syntax noprefix
.text

/* u64 demo_win64_preserve(u64 a, u64 b)
   Windows x64:
     a in rcx, b in rdx, return in rax
   Non-volatile used here: rbx, xmm6
*/
.global demo_win64_preserve
demo_win64_preserve:
    push    rbx                     /* save non-volatile GPR */
    sub     rsp, 32                 /* local space; keep stack aligned */
    movdqu  xmmword ptr [rsp], xmm6  /* save non-volatile XMM6 */
```

```
    mov     rbx, rcx                /* rbx = a (using callee-saved register) */
    add     rbx, rdx                /* rbx = a + b */

    pxor    xmm6, xmm6              /* xmm6 clobbered intentionally */
    mov     rax, rbx                /* return */

    movdqu  xmm6, xmmword ptr [rsp] /* restore XMM6 */
    add     rsp, 32
    pop     rbx                     /* restore RBX */
    ret
```

7.3.6 Rust (Windows) Calling the Assembly Function

```
#[link(name = "myasm")]
extern "C" {
    fn demo_win64_preserve(a: u64, b: u64) -> u64;
}

fn main() {
    let a = 10u64;
    let b = 32u64;
    let r = unsafe { demo_win64_preserve(a, b) };
    println!("{} + {} = {}", a, b, r);
}
```

7.4 AArch64 Registers

AArch64 is designed for regularity: fixed instruction width and a uniform register file.

7.4.1 Integer Registers

- `X0–X30`: 64-bit integer registers
- `W0–W30`: lower 32-bit views of `X0–X30`

Special roles by convention:

- `X29`: frame pointer (`FP`) in many ABIs and compiler conventions
- `X30`: link register (`LR`), holds return address after `BL`
- `SP`: stack pointer (separate name; not just `X31` in assembly usage)

AArch64 also has a zero/register-alias concept for certain encodings:

- `XZR/WZR` as a zero register in many instruction forms.

7.4.2 SIMD/Floating Registers

- `V0–V31`: 128-bit vector registers
- scalar floating types (`S` for 32-bit float, `D` for 64-bit float) are views into `Vn`.

7.4.3 Condition Flags

AArch64 uses condition flags (NZCV) that are set by flag-setting variants of instructions and tested by conditional branches.

7.4.4 AArch64 ABI Volatile vs Non-Volatile (AAPCS64 Foundation Summary)

A common foundational view used by mainstream toolchains:

- **Volatile (caller-saved) integer:** `X0–X18` are typically treated as caller-saved in general usage, with special conventions for `X16/X17` (intra-procedure-call scratch) and `X18` (platform register).
- **Non-volatile (callee-saved) integer:** `X19–X28`
- **Frame and link:** `X29` (FP) and `X30` (LR) are saved/restored as needed by the function.

SIMD/Floating (foundation summary):

- **Volatile:** `V0–V7` are commonly used for arguments/returns and are volatile.
- **Non-volatile:** `V8–V15` are commonly treated as callee-saved in ABI practice.
- **Volatile:** `V16–V31` are typically volatile.

7.4.5 AArch64 Example: Saving Callee-Saved Registers

This is a minimal AArch64-style prolog/epilog pattern using callee-saved `X19` and the frame/link pair.

```
/* AArch64 pseudo-ABI demonstration (GNU as syntax style, C comments used)
   u64 demo_a64(u64 a, u64 b)
   Typical conventions:
     a in x0, b in x1, return in x0
*/
.global demo_a64
demo_a64:
    stp     x29, x30, [sp, -16]!     /* save FP/LR, update SP */
    mov     x29, sp                  /* set frame pointer */
    stp     x19, x20, [sp, -16]!     /* save callee-saved regs if used */

    mov     x19, x0                  /* x19 = a (callee-saved) */
    add     x19, x19, x1             /* x19 = a + b */
```

```
    mov     x0, x19                 /* return */

    ldp     x19, x20, [sp], 16      /* restore callee-saved regs */
    ldp     x29, x30, [sp], 16      /* restore FP/LR */
    ret
```

7.5 RV64 Registers (RISC-V 64-bit)

RISC-V is deliberately minimal and orthogonal. Registers have fixed roles under the ABI naming convention.

7.5.1 Integer Registers: x0–x31

RISC-V defines 32 integer registers:

- `x0` is hard-wired to zero (reads as 0; writes are ignored).
- `x1−x31` are general-purpose.

7.5.2 ABI Register Names (Must-Know Table)

The standard ABI names:

Register	ABI name	Typical role
x0	zero	constant 0
x1	ra	return address
x2	sp	stack pointer
x3	gp	global pointer
x4	tp	thread pointer
x5–x7	t0–t2	temporaries (caller-saved)
x8	s0/fp	callee-saved / frame pointer
x9	s1	callee-saved
x10–x17	a0–a7	arguments / return values (caller-saved)
x18–x27	s2–s11	callee-saved
x28–x31	t3–t6	temporaries (caller-saved)

7.5.3 Floating-Point Registers (If F/D Extensions Are Present)

When the floating-point ABI is in use:

- `f0–f31` exist, with ABI names `ft0..`, `fs0..`, `fa0..` etc.

In this foundations chapter, the critical idea is:

- integer ABI and floating ABI can be configured; do not assume floating registers are used unless the ABI and compilation mode specify it.

7.5.4 RV64 ABI Volatile vs Non-Volatile (Foundation Summary)

Common RISC-V calling convention discipline:

- **Callee-saved:** `s0–s11` (x8–x9, x18–x27), and `sp` (x2)
- **Caller-saved:** `t0–t6` (x5–x7, x28–x31), `a0–a7` (x10–x17), and `ra` (x1)

7.5.5 RV64 Example: Using s0 as Frame Pointer and Preserving It

```
/* RV64 pseudo-ABI demonstration (GNU as style, C comments used)
   u64 demo_rv64(u64 a, u64 b)
   Typical conventions:
     a in a0, b in a1, return in a0
*/
.global demo_rv64
demo_rv64:
    addi    sp, sp, -16             /* allocate stack */
    sd      ra, 8(sp)               /* save return address */
    sd      s0, 0(sp)               /* save s0/fp (callee-saved) */
    addi    s0, sp, 16              /* set frame pointer if desired */

    add     a0, a0, a1              /* a0 = a + b */

    ld      s0, 0(sp)               /* restore s0 */
    ld      ra, 8(sp)               /* restore ra */
    addi    sp, sp, 16              /* deallocate */
    ret
```

7.6 Volatile vs Non-Volatile: Practical Rules That Prevent Real Bugs

7.6.1 Rule 1: Treat Calls as Clobbering All Volatile Registers

When you write assembly and you call another function:

- assume every volatile register may be destroyed,
- save what you need before the call,
- restore after the call if needed.

7.6.2 Rule 2: Only Save Non-Volatiles If You Actually Use Them

Non-volatile registers are expensive to save/restore in leaf functions. Optimize correctness first, then cost:

- leaf function: prefer volatile temporaries when possible,
- non-leaf or heavy computation: using callee-saved registers can reduce caller pressure, but you must preserve them properly.

7.6.3 Rule 3: ABI Defines Register Preservation, Not the ISA

Example:

- On x86-64, `RBX` exists everywhere, but whether it must be preserved depends on the ABI.
- On Windows x64, `XMM6-XMM15` are non-volatile; on System V, XMM registers are typically volatile.

If you write portable assembly:

- either target one ABI per file,
- or isolate ABI differences behind thin wrappers.

7.7 Extended x86-64 Example: Caller-Saved vs Callee-Saved in One Scenario

This example shows a caller that needs a volatile value preserved across a call. We will:

- compute a value into `R10` (volatile),
- call another function that may clobber it,
- preserve `R10` by saving it on the stack.

```
.intel_syntax noprefix
.text

/* u64 callee_clobber(u64 x)
   Intentionally clobbers volatile registers.
   Windows x64: x in rcx, return in rax
*/
.global callee_clobber
callee_clobber:
    mov     rax, rcx
    xor     r10, r10                /* clobber volatile register */
    add     rax, 5
    ret

/* u64 caller_preserve(u64 a)
   Computes temp in r10, calls callee, then uses r10 again.
*/
.global caller_preserve
caller_preserve:
    sub     rsp, 32                 /* keep stack aligned; local space */
    mov     r10, rcx                /* r10 = a (volatile temp) */
    imul    r10, r10, 3             /* r10 = a*3 */
```

```
    mov     qword ptr [rsp], r10     /* save volatile across call */
    call    callee_clobber
    mov     r10, qword ptr [rsp]     /* restore r10 */

    add     rax, r10                 /* rax = callee_result + (a*3) */

    add     rsp, 32
    ret
```

```
#[link(name = "myasm")]
extern "C" {
    fn caller_preserve(a: u64) -> u64;
}

fn main() {
    let a = 7u64;
    let r = unsafe { caller_preserve(a) };
    // r = (a + 5) + (a*3) = 4a + 5
    println!("a={} r={}", a, r);
}
```

7.8 Professional Checklist

Before you publish an assembly routine intended for real use:

- Decide the target ABI (Windows x64 vs System V AMD64 vs AArch64 AAPCS64 vs RV64 psABI).
- Write down:
  - which registers carry arguments/returns,
  - which registers you plan to use as temporaries,

  - which non-volatile registers you will touch.

- If you touch any non-volatile register, save/restore it correctly (including SIMD non-volatiles where applicable).
- Treat all volatile registers as destroyed across any call.
- Keep stack pointer discipline (alignment and restoration) at every call boundary.
- Validate by:
  - disassembling produced code,
  - running deterministic tests that call your routine repeatedly under optimization.

7.9 Exercises (Foundation Level)

1. Write a leaf function on x86-64 Windows that uses only volatile registers. Confirm it has no pushes/pops beyond stack alignment needs.
2. Modify the function to use `RBX` and `XMM6`. Add correct save/restore and confirm results remain correct.
3. For AArch64, write a function that uses `X19` and `V8`. Add correct preservation.
4. For RV64, write a function that uses `s2` and calls another function. Preserve `s2` and `ra` and verify correctness.

7.10 Professional Summary

Register architecture mastery is not memorizing names; it is mastering contracts:

- x86-64 provides layered register views and a distinct flags model; ABIs differ significantly between Windows and System V.
- AArch64 provides a uniform integer file `X0–X30` and vector file `V0–V31`, with clear call-boundary expectations.
- RV64 provides a minimal 32-register model with explicit ABI naming that makes caller/callee responsibilities visible.
- Volatile vs non-volatile is the difference between reliable assembly and intermittent disasters.

Addressing Modes and Effective Address Calculation

8.1 Scope and Engineering Goal

Every load/store, every stack access, every field read in a struct, every array index, and many control-flow patterns depend on one core idea:

- **Effective Address (EA)** = the address produced by the instruction's addressing mode.

Assembly authors must be able to:

- read memory operand syntax and immediately infer the address computation,
- design ABI-safe memory accesses (alignment, width, volatility assumptions),
- implement correct indexing for arrays, structs, and tables,
- recognize which architectures support complex address computation directly and which require explicit arithmetic.

This chapter focuses on:

- memory operand formats,

- address computation rules,

with a comparative view across x86-64, AArch64, and RV64.

8.2 Memory Operand Formats

8.2.1 Core Concept: "Memory Is Bytes, Addressing Selects a Byte Location"

A memory operand is not a value; it is a **rule to compute an address**. Then the instruction loads/stores at that address with a certain width:

- byte (8-bit), word (16-bit), dword (32-bit), qword (64-bit),
- vector widths for SIMD (128/256/512 bits),
- special widths for atomics or paired loads/stores on some ISAs.

Professional rule:

- Always separate **EA calculation** from **data interpretation** (signed/unsigned, float/int, endianness).

8.2.2 x86-64 Memory Operand Format (GAS Intel Syntax)

In Intel syntax, the canonical mental model is:

```
[base + index*scale + displacement]
```

Where:

- `base` is a 64-bit register (or absent),
- `index` is a 64-bit register (or absent),

- `scale` is one of {1,2,4,8},
- `displacement` is an immediate (commonly 8 or 32-bit sign-extended by encoding rules).

Common x86-64 Memory Operand Shapes

Form	Example	EA meaning
Base only	`[rcx]`	EA = RCX
Base + disp	`[rcx+16]`	EA = RCX + 16
Base + index*scale	`[rcx+rdx*8]`	EA = RCX + RDX*8
Base + index*scale + disp	`[rcx+rdx*4+12]`	EA = RCX + RDX*4 + 12
RIP-relative	`[rip+disp32]`	EA = RIP(next) + disp32

Width Qualifiers

In Intel syntax you often specify width explicitly:

- `byte ptr [..]`, `word ptr [..]`, `dword ptr [..]`, `qword ptr [..]`

Professional rule:

- In machine code, the width is determined by the opcode and operand-size rules. The assembler's `qword ptr` is how you disambiguate the intended form.

8.2.3 AArch64 Memory Operand Formats

AArch64 is more uniform. Most loads/stores are based on a **base register** plus an **offset**:

- base + immediate offset (often scaled by access size),
- base + register offset (optionally scaled/extended),

- pre-index and post-index forms (update base as part of the addressing).

Conceptually:

```
[Xn, #imm]  [Xn, Xm]  [Xn, #imm]!  [Xn], #imm
```

Where `!` indicates pre-index update, and post-index updates after the access.
Professional rule:

- AArch64 does not expose x86-style `base + index*scale + disp` as one universal pattern. It provides a family of structured forms designed for predictable decoding and safety.

8.2.4 RV64 Memory Operand Formats

RV64 base integer load/store formats are intentionally simple:

```
offset(base)
```

Example:

- `ld a0, 24(sp)` means EA = SP + 24

There is no general "scaled index" addressing in the load/store instruction itself. Instead:

- compute address in a register using `add`, shifts, etc.,
- then use base+imm load/store.

Professional rule:

- On RV64, addressing complexity moves into explicit arithmetic instructions. This improves ISA regularity and simplifies decoding.

8.3 Address Computation Rules

8.3.1 Effective Address vs Loaded Value

Given an instruction like:

```
mov rax, qword ptr [rcx+rdx*8+16]
```

Two distinct computations exist:

- EA computation: `EA = RCX + RDX*8 + 16`
- load semantics: read 8 bytes from memory at `EA`, interpret as a 64-bit value (endianness applies), place into `RAX`

If you confuse these two layers, you will mis-debug and mis-design low-level code.

8.3.2 x86-64: Formal EA Model for the Common Addressing Form

For x86-64, the common addressing form can be summarized as:

$$\text{EA} = \text{base} + \text{index} \times \text{scale} + \text{disp}$$

Constraints and rules:

- scale is restricted to 1,2,4,8 (chosen to match byte sizes of common element widths),
- displacement is encoded as an immediate (often 8-bit or 32-bit) and is sign-extended when applicable,
- some register combinations force a SIB byte in encoding, but the *EA rule remains the same*.

LEA: Address Computation Without a Memory Access

`LEA` computes an EA and writes the result into a register *without reading memory*. It is one of the most important x86-64 primitives for:

- pointer arithmetic,
- array indexing,
- struct field address calculation,
- efficient multiply-add patterns using scale factors.

GAS Intel Syntax: x86-64 EA Examples Using LEA

```
.intel_syntax noprefix
.text

/* u64 ea_base(u64* p)
   Windows x64:
     p in rcx, return in rax
   Returns EA = p
*/
.global ea_base
ea_base:
    lea     rax, [rcx]
    ret

/* u64 ea_base_disp(u8* p)
   Returns EA = p + 24
*/
.global ea_base_disp
ea_base_disp:
    lea     rax, [rcx+24]
```

```
    ret

/* u64 ea_base_index_scale_disp(u64* base, u64 index)
   Returns EA = base + index*8 + 16
   Note: scale=8 matches sizeof(u64)
*/
.global ea_base_index_scale_disp
ea_base_index_scale_disp:
    lea     rax, [rcx+rdx*8+16]
    ret

/* u64 load_u64_indexed(u64* base, u64 index)
   Loads *(base + index) using EA = base + index*8
   Returns loaded value in rax
*/
.global load_u64_indexed
load_u64_indexed:
    mov     rax, qword ptr [rcx+rdx*8]
    ret
```

RIP-Relative Addressing: Position-Independent Access

In x86-64, many code models access static data using RIP-relative addressing:

```
EA = RIP(next_instruction) + disp32
```

Engineering consequences:

- code can be position-independent,
- the displacement is 32-bit signed in encoding, limiting direct reach per instruction,
- relocation/loader can adjust targets without rewriting absolute addresses.

8.3.3 AArch64: Structured Addressing Rules

AArch64 load/store addressing forms are regular and tied to access size.

Base + Immediate Offset (often scaled)

For many loads/stores, the immediate is scaled by the data size. Example conceptual pattern:

- load 64-bit from `[Xn, #imm]` where `imm` represents multiples of 8 bytes (architecture-defined by instruction form).

Professional rule:

- When reading AArch64 disassembly, always determine whether the offset is **scaled** or **unscaled**. The printed assembly usually reflects the effective byte offset, but understanding scaling matters for encoding and range limits.

Base + Register Offset (optional extension/shift)

AArch64 can use a register offset optionally shifted or extended, enabling efficient indexing patterns while keeping instruction format fixed-width.

Pre-Index and Post-Index

These forms update the base register:

- pre-index: compute EA, update base, then access,
- post-index: access, then update base.

This is heavily used for stack operations and pointer walks.

8.3.4 RV64: Explicit Address Arithmetic Philosophy

RV64 load/store addressing is base+imm only:

```
EA = base + sign_extended_imm
```

To do array indexing:

- compute `index * element_size` using shifts/mul,
- add to base,
- load/store using offset 0 (or a small constant offset).

This results in:

- fewer addressing-mode corner cases,
- straightforward disassembly and verification,
- predictable decode and execution behavior.

8.4 Cross-Architecture Comparison: What You Must Internalize

8.4.1 Addressing Complexity Placement

ISA	Where complexity lives	Typical result
x86-64	inside a single memory operand (SIB + disp)	fewer instructions, heavier decode

ISA	Where complexity lives	Typical result
AArch64	in structured forms (imm scaled, reg offset, pre/post index)	predictable patterns, fixed width
RV64	in explicit arithmetic instructions before load/store	simplest loads/stores, more explicit steps

8.4.2 Element Indexing Example: `base + i*sizeof(T)`

For an array of 64-bit elements:

- x86-64: `[base + i*8]` is native
- AArch64: typically use base+reg offset with shift or base+scaled immediate if i is constant
- RV64: compute `i«3` then add to base, then `ld` with offset 0

8.5 Cross-Platform Data Pitfalls (Addressing Edition)

8.5.1 Pitfall 1: Mistaking Index Units

Some programmers treat an index as bytes when it is elements, or vice versa. Correct discipline:

- if index is element index, multiply by element size before adding to base,
- if offset is already bytes, do not scale again.

8.5.2 Pitfall 2: Sign Extension of Displacements and Immediates

Many ISAs sign-extend certain immediates during decoding. If you treat an encoded displacement as unsigned in your mental model, negative offsets will surprise you.
Professional rule:

- Always know whether the immediate/displacement is sign-extended and what its bit-width is in encoding.

8.5.3 Pitfall 3: Alignment Assumptions Hidden in Addressing

Even if an ISA tolerates unaligned scalar loads, your ABI and performance targets may not. If your addressing produces potentially unaligned addresses:

- either document alignment requirements in the API,
- or use unaligned-safe access patterns and accept the cost.

8.5.4 Pitfall 4: Pointer Size and Integer Truncation

On 64-bit targets:

- addresses are 64-bit,
- truncating to 32-bit breaks above the 4GiB range.

Discipline:

- keep address arithmetic in pointer-width registers/types (`u64`/`usize`).

8.6 Rust (Windows-Friendly) Laboratory: Verifying EA Computation Against High-Level Math

This laboratory calls the x86-64 assembly routines above and cross-checks their results with Rust pointer arithmetic. It is designed for Windows x64 builds and uses only stable language features.

```
use std::mem::size_of;

#[link(name = "myasm")]
extern "C" {
    fn ea_base(p: *const u8) -> u64;
    fn ea_base_disp(p: *const u8) -> u64;
    fn ea_base_index_scale_disp(base: *const u64, index: u64) -> u64;
    fn load_u64_indexed(base: *const u64, index: u64) -> u64;
}

fn main() {
    let data_u8: [u8; 64] = [0u8; 64];
    let p = data_u8.as_ptr();

    let a0 = unsafe { ea_base(p) };
    let a1 = unsafe { ea_base_disp(p) };

    let r0 = p as u64;
    let r1 = unsafe { p.add(24) as u64 };

    println!("ea_base      asm=0x{:016X} rust=0x{:016X}", a0, r0);
    println!("ea_base_disp asm=0x{:016X} rust=0x{:016X}", a1, r1);

    let arr: [u64; 8] = [10, 20, 30, 40, 50, 60, 70, 80];
    let base = arr.as_ptr();

    let i: u64 = 3;

    let ea_asm = unsafe { ea_base_index_scale_disp(base, i) };
    let ea_rust = unsafe {
        (base as *const u8).add((i as usize) * size_of::<u64>() + 16) as u64
    };
```

```
    println!("EA(base+i*8+16) asm=0x{:016X} rust=0x{:016X}", ea_asm, ea_rust);

    let v_asm = unsafe { load_u64_indexed(base, i) };
    let v_rust = arr[i as usize];

    println!("load_u64_indexed asm={} rust={}", v_asm, v_rust);
}
```

8.7 Practical Patterns for Assembly Authors

8.7.1 Pattern 1: Struct Field Address and Load

For a C/Rust `repr(C)` struct, field offset is fixed by layout rules. Your assembly should:

- compute address = base + field_offset,
- load with correct width.

8.7.2 Pattern 2: Bounds-Safe Indexing Discipline

Assembly has no automatic bounds checks. If your routine receives `base` and `count`, enforce:

- index < count,
- address does not overflow pointer range (when relevant),
- alignment requirements (when required by ABI/instruction choice).

8.7.3 Pattern 3: Use Address Computation Instructions Intentionally

- x86-64: use `lea` for pointer math (no load), then `mov` to load/store.

- AArch64: use `add` (possibly with shift/extend) for pointer math, or use pre/post-indexed addressing when it matches the algorithm.
- RV64: compute address with `add`/shift, then use base+imm load/store.

8.8 Exercises (Foundation Level)

1. Write an x86-64 Windows assembly function that returns the address of `arr[i]` for:
   - `u8`, `u16`, `u32`, `u64` element sizes,

   using `lea` and scale factors when possible. Validate from Rust.
2. Write a parser-style routine in Rust that reads a field at a fixed offset from a byte buffer and compare:
   - computing pointer with `add(offset)` and reading bytes,
   - calling an assembly helper that computes the EA for the field.
3. For AArch64 and RV64:
   - write pseudo-assembly sequences that compute `base + i*8 + 16` and load a 64-bit value,
   - explain which part is done by addressing mode vs explicit arithmetic.
4. Design a small ABI-stable API for an array accessor:
   - inputs: base pointer and index,
   - output: loaded value,
   - document alignment and bounds assumptions.

8.9 Professional Summary

Addressing modes are the **math of memory access**. Professional assembly authors must be able to:

- translate every memory operand into an explicit EA equation,
- separate EA from value interpretation,
- respect ABI-driven alignment and register usage rules,
- understand where each ISA places complexity:
  - x86-64: complex EA inside the instruction,
  - AArch64: structured and predictable addressing forms,
  - RV64: explicit arithmetic outside the load/store.

Master this chapter and you can reliably implement arrays, structs, stacks, and protocol parsers in assembly without hidden assumptions.

Arithmetic and Logic Operations

9.1 Flags Behavior

9.1.1 Flags Are Side Effects of Arithmetic

Most ISAs provide status flags (or a status register) that record facts about the *last* operation. Those facts are then consumed by conditional branches, conditional moves/selects, and some arithmetic sequences.
The key principle:

- **Flags do not store the result; they store properties of the result and/or the operation.**

9.1.2 x86-64 Flags (RFLAGS) That Matter for Comparisons

For integer arithmetic and comparisons, the most important flags are:

- **ZF** (Zero Flag): result equals zero
- **SF** (Sign Flag): most significant bit of the result (sign bit for two's complement)
- **CF** (Carry Flag): carry out from addition, or borrow from subtraction (unsigned overflow/underflow indicator)
- **OF** (Overflow Flag): signed overflow indicator

Two other flags appear in specialized contexts:

- **PF** (Parity Flag): parity of low 8 bits; rarely used in modern high-level logic
- **AF** (Adjust Flag): BCD-era artifact; rarely used in modern systems programming

9.1.3 How x86-64 Sets Key Flags

Addition

For `add dst, src` (or `adc`):

- ZF: set if result is zero
- SF: set if result MSB is 1
- CF: set if unsigned carry out occurred
- OF: set if signed overflow occurred

Subtraction / Compare

`sub dst, src` sets flags like `dst - src`. `cmp a, b` is the same as `sub a, b` *without storing the result*. It sets flags as if computing:

$$a - b$$

Key interpretation:

- CF is meaningful for **unsigned** comparisons
- SF and OF together are meaningful for **signed** comparisons

9.1.4 AArch64 Flags (NZCV) and Flag-Setting Instructions

AArch64 uses four primary condition flags (NZCV):

- N: negative (result MSB set)
- Z: zero
- C: carry (in addition) / NOT-borrow (in subtraction)
- V: signed overflow

AArch64 does not automatically set flags for every arithmetic instruction; it has flag-setting variants:

- `ADDS`, `SUBS` set NZCV
- plain `ADD`, `SUB` do not set flags

This design forces discipline:

- you explicitly choose when flags are produced,
- you avoid unintended flag clobbering.

9.1.5 RV64 Compare Model: Flags-Free, Branch-on-Compare Style

RISC-V does not have a general flags register for integer comparisons. Instead, it uses:

- comparison instructions that produce a boolean integer (`slt`, `sltu`, `slti`, `sltiu`),
- conditional branches that compare registers (`beq`, `bne`, `blt`, `bltu`, `bge`, `bgeu`).

Engineering consequence:

- the dependency chain is explicit in registers, not implicit in flags,
- reordering is often simpler, and side effects are more visible.

9.2 Compare Patterns

9.2.1 Compare Is Often Just Subtraction

Across architectures, comparison is conceptually based on subtraction:

$$a - b$$

and then checking properties:

- is result zero?
- is it negative (signed)?
- did unsigned underflow occur?
- did signed overflow occur?

9.2.2 x86-64 Comparison Conditions You Must Know

For `cmp a, b` (sets flags as `a - b`):

Equality

- equal: ZF = 1 (`je`)
- not equal: ZF = 0 (`jne`)

Unsigned Comparisons

- `a < b` (unsigned): CF = 1 (`jb`)
- `a >= b` (unsigned): CF = 0 (`jae`)

- `a > b` (unsigned): CF = 0 and ZF = 0 (`ja`)
- `a <= b` (unsigned): CF = 1 or ZF = 1 (`jbe`)

Signed Comparisons

Signed relations use SF and OF:

- `a < b` (signed): SF $\neq$ OF (`jl`)
- `a >= b` (signed): SF = OF (`jge`)
- `a > b` (signed): ZF = 0 and SF = OF (`jg`)
- `a <= b` (signed): ZF = 1 or SF $\neq$ OF (`jle`)

9.2.3 AArch64 Compare Patterns

AArch64 compare is typically:

- `CMP Xn, Xm` which is an alias for `SUBS XZR, Xn, Xm`
- `CMN Xn, Xm` which is an alias for `ADDS XZR, Xn, Xm`

Conditions are tested with `B.<cond>` where `<cond>` refers to NZCV. Conceptually:

- equality uses Z
- unsigned uses C/Z combinations
- signed uses N/V combinations

9.2.4 RV64 Compare Patterns

RISC-V uses explicit compare-and-branch:

- `blt a0, a1, label` signed less-than
- `bltu a0, a1, label` unsigned less-than

And for building boolean results:

- `slt rd, rs1, rs2` sets rd=1 if rs1 < rs2 (signed), else 0
- `sltu rd, rs1, rs2` sets rd=1 if rs1 < rs2 (unsigned), else 0

9.3 Conditional Execution

Conditional execution means: do something only if a condition holds, without necessarily using a high-level `if`.
There are three main families:

- conditional branches (control-flow changes),
- conditional moves/selects (data-flow selection without branching),
- predication (some instruction sets can predicate operations, AArch64 has conditional select family and conditional branches; older AArch32 had full predication).

9.3.1 x86-64 Conditional Branches and Conditional Moves

x86-64 provides:

- a rich family of conditional jumps (`jcc`) based on flags,
- conditional moves (`cmovcc`) that select between registers based on flags.

Branch vs CMOV: Engineering Tradeoff

- Branch is cheap when well-predicted; expensive when mispredicted.
- CMOV avoids branches but may execute extra work unconditionally and can have its own scheduling costs.

Foundations rule:

- Use branches for control flow and large blocks.
- Use CMOV/select patterns for tiny selections in hot paths where branch misprediction is likely.

9.3.2 AArch64 Conditional Branch and Conditional Select

AArch64 provides:

- conditional branches `B.<cond>`,
- conditional select `CSEL` and related instructions (`CSINC`, `CSINV`, `CSNEG`) that select or transform based on condition flags.

9.3.3 RV64 Conditional Control

RV64 uses:

- branch instructions that compare registers,
- conditional selection by explicit branches or by arithmetic/bit tricks producing masks.

There is no direct equivalent to x86 `cmovcc` in the base ISA; selection patterns are built from:

- branches,

- `slt/sltu` plus arithmetic,
- bitwise mask composition.

9.4 x86-64 (Windows) Extensive Examples With Rust Verification

All assembly examples in this section assume Windows x64 calling convention:

- first integer arg: RCX
- second integer arg: RDX
- return: RAX

9.4.1 Example 1: Return Flags After Addition (Carry and Overflow)

This routine adds two 64-bit integers and returns a bitmask describing CF and OF. Bit assignment:

- bit0 = CF (unsigned carry)
- bit1 = OF (signed overflow)
- bit2 = ZF (result zero)
- bit3 = SF (result negative)

```
.intel_syntax noprefix
.text

/* u64 add_flags(u64 a, u64 b)
   Windows x64: a in rcx, b in rdx, return in rax
```

```
   Returns:
     rax bits:
       0: CF
       1: OF
       2: ZF
       3: SF
*/
.global add_flags
add_flags:
    mov     rax, rcx
    add     rax, rdx

    xor     r8d, r8d                /* r8d will hold the mask */

    setc    r9b                     /* CF into r9b */
    movzx   r9d, r9b
    or      r8d, r9d                /* bit0 */

    seto    r9b                     /* OF into r9b */
    movzx   r9d, r9b
    shl     r9d, 1
    or      r8d, r9d                /* bit1 */

    sete    r9b                     /* ZF into r9b */
    movzx   r9d, r9b
    shl     r9d, 2
    or      r8d, r9d                /* bit2 */

    sets    r9b                     /* SF into r9b */
    movzx   r9d, r9b
    shl     r9d, 3
    or      r8d, r9d                /* bit3 */
```

```
    mov     eax, r8d                /* return mask in rax */
    ret
```

```
#[link(name = "myasm")]
extern "C" {
    fn add_flags(a: u64, b: u64) -> u64;
}

fn decode(mask: u64) -> (bool, bool, bool, bool) {
    let cf = (mask & 1) != 0;
    let of = (mask & 2) != 0;
    let zf = (mask & 4) != 0;
    let sf = (mask & 8) != 0;
    (cf, of, zf, sf)
}

fn main() {
    let tests = [
        (0u64, 0u64),
        (1u64, 2u64),
        (u64::MAX, 1u64),                   // CF expected
        (0x7FFF_FFFF_FFFF_FFFF, 1u64),      // signed overflow expected (OF)
    ];

    for (a, b) in tests {
        let m = unsafe { add_flags(a, b) };
        let (cf, of, zf, sf) = decode(m);
        let r = a.wrapping_add(b);
        println!("a=0x{a:016X} b=0x{b:016X} r=0x{r:016X}  CF={} OF={} ZF={} SF={}", cf, of,
        ↪  zf, sf);
    }
}
```

9.4.2 Example 2: Unsigned Compare Pattern (a < b) Without Branch

We compute a boolean (0/1) using flags.

```
.intel_syntax noprefix
.text

/* u64 u64_lt(u64 a, u64 b)
   Returns 1 if a < b (unsigned), else 0
*/
.global u64_lt
u64_lt:
    cmp     rcx, rdx
    setb    al                  /* CF=1 means a<b unsigned */
    movzx   eax, al
    ret
```

```
#[link(name = "myasm")]
extern "C" {
    fn u64_lt(a: u64, b: u64) -> u64;
}

fn main() {
    let cases = [
        (0u64, 1u64),
        (5u64, 5u64),
        (10u64, 2u64),
        (u64::MAX, 0u64),
        (0u64, u64::MAX),
    ];

    for (a, b) in cases {
        let r = unsafe { u64_lt(a, b) };
        println!("a={} b={} asm={} rust={}", a, b, r, (a < b) as u64);
```

```
    }
}
```

9.4.3 Example 3: Signed Compare Pattern (a < b) Without Branch

Signed less-than uses `SF != OF` after `cmp`. We can use `setl`.

```
.intel_syntax noprefix
.text

/* u64 i64_lt(i64 a, i64 b)
   Windows x64 passes i64 same as u64 in registers.
   Returns 1 if a < b (signed), else 0
*/
.global i64_lt
i64_lt:
    cmp     rcx, rdx
    setl    al                      /* signed: SF != OF */
    movzx   eax, al
    ret
```

```
#[link(name = "myasm")]
extern "C" {
    fn i64_lt(a: i64, b: i64) -> u64;
}

fn main() {
    let cases = [
        (-5i64, 0i64),
        (0i64, -1i64),
        (7i64, 7i64),
        (10i64, -2i64),
        (-10i64, -2i64),
```

```
    ];
    for (a, b) in cases {
        let r = unsafe { i64_lt(a, b) };
        println!("a={} b={} asm={} rust={}", a, b, r, (a < b) as u64);
    }
}
```

9.4.4 Example 4: Conditional Select (min) With Branchless CMOV

Compute `min(a,b)` for unsigned integers using `cmovb`.

```
.intel_syntax noprefix
.text

/* u64 u64_min(u64 a, u64 b)
   Returns min(a,b) under unsigned comparison
*/
.global u64_min
u64_min:
    mov     rax, rcx
    cmp     rcx, rdx
    cmova   rax, rdx              /* if a>b (unsigned), select b into rax */
    ret
```

```
#[link(name = "myasm")]
extern "C" {
    fn u64_min(a: u64, b: u64) -> u64;
}

fn main() {
    let cases = [
        (10u64, 20u64),
        (20u64, 10u64),
```

```
        (5u64, 5u64),
        (0u64, u64::MAX),
    ];
    for (a, b) in cases {
        let r = unsafe { u64_min(a, b) };
        println!("min({}, {}) = {}", a, b, r);
    }
}
```

9.4.5 Example 5: Conditional Execution by Branch (abs) With Correct Signed Handling

`abs(x)` for signed 64-bit requires care around the most negative number, but at the ISA level we demonstrate the conditional logic.
This function returns:

- `x` if `x >= 0`
- `-x` if `x < 0`

```
.intel_syntax noprefix
.text

/* i64 i64_abs(i64 x)
   Windows x64: x in rcx, return in rax
*/
.global i64_abs
i64_abs:
    mov     rax, rcx
    test    rax, rax            /* set ZF/SF based on rax */
    jns     .done               /* if SF=0 => non-negative */
    neg     rax                 /* rax = -rax, sets CF unless rax was 0 */
.done:
```

```
    ret
```

```
#[link(name = "myasm")]
extern "C" {
    fn i64_abs(x: i64) -> i64;
}

fn main() {
    let cases = [0i64, 1i64, -1i64, 123i64, -123i64, i64::MIN];
    for x in cases {
        let r = unsafe { i64_abs(x) };
        println!("x={} abs(asm)={}", x, r);
    }
}
```

9.5 Cross-Architecture Pattern Cookbook

This section provides “equivalent intent” patterns across the three architectures.

9.5.1 Pattern: Compare for Equality

x86-64

```
cmp     rax, rbx
je      equal_label
```

AArch64

```
/* cmp is subs xzr, xN, xM */
cmp     x0, x1
b.eq    equal_label
```

RV64

```
beq     a0, a1, equal_label
```

9.5.2 Pattern: Signed Less-Than Branch

x86-64

```
cmp     rax, rbx
jl      less_label
```

AArch64

```
cmp     x0, x1
b.lt    less_label
```

RV64

```
blt     a0, a1, less_label
```

9.5.3 Pattern: Unsigned Less-Than Branch

x86-64

```
cmp     rax, rbx
jb      below_label
```

AArch64

```
cmp     x0, x1
b.lo    below_label           /* unsigned lower */
```

RV64

```
bltu    a0, a1, below_label
```

9.5.4 Pattern: Branchless Select (min/max) Concept

x86-64 (cmov)

```
mov     rax, a
cmp     a, b
cmova   rax, b
```

AArch64 (csel)

```
/* x2 = (a > b)? b : a  (unsigned example) */
cmp     x0, x1
csel    x2, x1, x0, hi          /* if a>b unsigned (hi), choose x1 else x0 */
```

RV64 (mask/select idea)

RV64 often builds selects using:

- compare into a boolean register,
- negate into a mask,
- use bitwise selection:

$$\text{select} = (mask \& b) | (\sim mask \& a)$$

9.6 Professional Checklist

Before using flags/compare logic in real assembly:

- Decide signed vs unsigned *first*; then choose the correct condition codes.

- Treat flags as volatile state: any arithmetic/logical instruction may clobber them.
- Use `test reg, reg` for zero/sign checks when you do not need a subtraction compare.
- For micro hot-path selection:
  - consider `cmov` (x86-64) or `csel` (AArch64),
  - avoid unpredictable branches when possible.
- Validate with exhaustive edge tests:
  - 0, 1, max, min,
  - signed extremes (`i64::MIN`, `i64::MAX`),
  - unsigned extremes (`u64::MAX`).

9.7 Exercises (Foundation Level)

1. Implement `u64_gt`, `u64_le`, `i64_ge` on x86-64 using `setcc`. Verify from Rust.
2. Implement branchless `max(a,b)` for unsigned using `cmov`. Verify from Rust.
3. Implement a "clamp" function:
   - clamp(x, lo, hi) returns:
   $$\max(lo, \min(x, hi))$$

   Provide an x86-64 version using compares and conditional moves.
4. Write equivalent pseudo-assembly for AArch64 using `cmp` + `csel`, and for RV64 using branches or mask selection.

9.8 Professional Summary

Arithmetic and logic operations are not only about the numeric result. They are about **control**. A professional assembly author must master:

- how flags/NZCV record properties of results,
- how comparisons reduce to subtraction and flag interpretation,
- how signed/unsigned meanings differ using the same machine bits,
- how conditional execution is implemented:
  - x86-64: flags + jcc + cmov,
  - AArch64: NZCV + b.cond + csel family,
  - RV64: explicit compare/branch and boolean registers.

Control Flow and Branching

10.1 Scope and Engineering Goal

Control flow is how programs *decide*, *repeat*, and *compose* work. At the assembly level, control flow is built from three mechanisms:

- conditional jumps (and conditional branches),
- loops (which are conditional jumps plus state updates),
- the call/ret model (function composition and ABI boundaries).

This chapter provides an engineering-grade view:

- how conditions are formed and consumed,
- how loops are lowered by compilers and written by hand,
- how call/ret interacts with the stack, return addresses, and register preservation rules,
- how these patterns differ across x86-64, AArch64, and RV64.

All runnable code and interoperability examples use Rust on Windows and x86-64 assembly in GAS Intel syntax. AArch64 and RV64 are presented as equivalent intent patterns and pseudo-assembly to build cross-ISA literacy.

10.2 Conditional Jumps

10.2.1 Control Flow as "PC Update Rules"

A conditional jump updates the program counter depending on a condition:

- if condition is true, jump to target
- otherwise, continue to fall-through (next instruction)

The internal CPU representation differs by ISA, but the observable behavior is consistent.

10.2.2 x86-64: Condition Codes Are Evaluations of Flags

x86-64 conditional jumps (`jcc`) consult flags in `RFLAGS` that were set by a prior instruction:

- arithmetic (`add/sub`),
- compare (`cmp`),
- bit-test (`test`),
- increments/decrements and shifts (many update flags).

Engineering rule:

- Treat flags as a volatile hidden dependency. Any instruction that sets flags can destroy the condition you intended to use.

Forming Conditions: `cmp` and `test`

Two foundational idioms:

- `cmp a, b` sets flags as if `a - b` was computed

- `test x, x` sets ZF/SF based on `x & x` (useful for zero and sign checks without changing the value)

A Minimal Set of Conditional Jump Semantics

Equality:

- `je` / `jne`: ZF = 1 / ZF = 0

Unsigned relations (after `cmp a, b`):

- `jb` (below): a < b when CF = 1
- `jae` (above/equal): a >= b when CF = 0
- `ja` (above): a > b when CF = 0 and ZF = 0
- `jbe` (below/equal): a <= b when CF = 1 or ZF = 1

Signed relations:

- `jl` (less): a < b when SF $\neq$ OF
- `jge` (greater/equal): a >= b when SF = OF
- `jg` (greater): a > b when ZF = 0 and SF = OF
- `jle` (less/equal): a <= b when ZF = 1 or SF $\neq$ OF

10.2.3 AArch64: Compare Sets NZCV, Branch Tests NZCV

AArch64 forms conditions using:

- `CMP` (alias of `SUBS` to set flags)

- `TST` (bit test)

Then branches with:

- `B.<cond> label` (branch on condition)

Engineering advantage:

- flags are set only when you use flag-setting variants, reducing accidental flag clobbers.

10.2.4 RV64: Branches Compare Registers Directly

RV64 avoids a general flags register. Branch instructions compare registers:

- `beq`, `bne`
- `blt`, `bge` (signed)
- `bltu`, `bgeu` (unsigned)

Engineering advantage:

- compare dependencies are explicit: operands appear directly in the branch instruction.

10.3 Loops

10.3.1 A Loop is a State Update + Conditional Branch

At the assembly level, all loops reduce to:

- initialize loop variables
- check condition (either at top or bottom)

- body
- update
- branch back if condition holds

Two canonical shapes:

- **Top-tested loop** (like `while`)
- **Bottom-tested loop** (like `do-while`)

10.3.2 x86-64: Do Not Use the `loop` Instruction in Modern Code

x86 provides a historical `loop` instruction that decrements `RCX/ECX` and branches if not zero. In modern optimized code, compilers typically prefer:

- `dec` / `cmp` + `jcc`,
- or pointer end comparisons,

because they integrate better with modern branch prediction and decoding heuristics.
Foundations rule:

- Understand `loop` for reading legacy code, but write explicit compare/branch loops for performance and clarity.

10.3.3 Loop Pattern 1: Counted Sum of u64 Array (x86-64 Windows)

We implement:

$$\texttt{sum(arr, n)} = \sum_{i=0}^{n-1} \texttt{arr[i]}$$

Windows x64 calling convention:

- RCX = pointer to u64 array
- RDX = n
- RAX = return

```
.intel_syntax noprefix
.text

/* u64 sum_u64(const u64* p, u64 n)
   RCX = p, RDX = n
   Returns sum in RAX
*/
.global sum_u64
sum_u64:
    xor     rax, rax                /* sum = 0 */
    test    rdx, rdx                /* n == 0 ? */
    je      .done

.loop:
    add     rax, qword ptr [rcx]    /* sum += *p */
    add     rcx, 8                  /* p++ */
    dec     rdx                     /* n-- */
    jne     .loop

.done:
    ret
```

```
#[link(name = "myasm")]
extern "C" {
    fn sum_u64(p: *const u64, n: u64) -> u64;
}

fn main() {
```

```
    let a: [u64; 8] = [1,2,3,4,5,6,7,8];
    let r = unsafe { sum_u64(a.as_ptr(), a.len() as u64) };
    let rust_sum: u64 = a.iter().copied().sum();
    println!("asm sum={} rust sum={}", r, rust_sum);
}
```

10.3.4 Loop Pattern 2: Find First Match (x86-64 Windows)

We implement:

- return index of first element equal to target, or `u64::MAX` if not found.

Inputs:

- RCX = pointer to u32 array
- RDX = count
- R8D = target

```
.intel_syntax noprefix
.text

/* u64 find_u32(const u32* p, u64 n, u32 target)
   RCX=p, RDX=n, R8D=target
   Returns index in RAX or u64::MAX if not found
*/
.global find_u32
find_u32:
    xor     rax, rax                /* index = 0 */
    test    rdx, rdx
    je      .not_found

.loop:
```

```
    cmp     dword ptr [rcx], r8d
    je      .found
    add     rcx, 4                  /* p++ */
    inc     rax                     /* index++ */
    dec     rdx
    jne     .loop

.not_found:
    mov     rax, -1                 /* u64::MAX */
    ret

.found:
    ret
```

```
#[link(name = "myasm")]
extern "C" {
    fn find_u32(p: *const u32, n: u64, target: u32) -> u64;
}

fn main() {
    let a: [u32; 6] = [10, 20, 30, 20, 50, 60];

    let i1 = unsafe { find_u32(a.as_ptr(), a.len() as u64, 20) };
    let i2 = unsafe { find_u32(a.as_ptr(), a.len() as u64, 99) };

    println!("find 20 => {}", i1);
    println!("find 99 => {}", i2);
}
```

10.3.5 AArch64 and RV64: Equivalent Loop Intents

AArch64 Pseudo-Assembly for Sum Loop

```
/* x0 = p, x1 = n, return x0 */
```

```
sum_u64_a64:
    mov     x2, 0
    cbz     x1, done
loop:
    ldr     x3, [x0], 8
    add     x2, x2, x3
    subs    x1, x1, 1
    b.ne    loop
done:
    mov     x0, x2
    ret
```

RV64 Pseudo-Assembly for Sum Loop

```
/* a0 = p, a1 = n, return a0 */
sum_u64_rv64:
    li      t0, 0
    beq     a1, zero, done
loop:
    ld      t1, 0(a0)
    add     t0, t0, t1
    addi    a0, a0, 8
    addi    a1, a1, -1
    bne     a1, zero, loop
done:
    mv      a0, t0
    ret
```

10.4 Call/Ret Model

10.4.1 The Universal Function Call Contract

A function call establishes:

- argument passing convention (registers/stack),
- return value convention,
- preservation rules (volatile vs non-volatile registers),
- stack alignment requirements,
- return address mechanism,
- unwind/debug metadata expectations (in real systems).

10.4.2 x86-64: Call/Ret Mechanism

What `call` Does

`call target`:

- pushes the address of the next instruction (return address) onto the stack,
- jumps to the target.

What `ret` Does

`ret`:

- pops an address from the stack,

- jumps to it.

Therefore, correctness requires:

- the stack pointer points to a valid return address at `ret`,
- the return address remains uncorrupted,
- stack alignment is correct at call boundaries.

10.4.3 AArch64: Link Register Call/Return Model

AArch64 uses a link register:

- `BL target` saves the return address in `LR` (`X30`) and branches,
- `RET` returns by branching to `LR` (or specified register form).

Functions that make nested calls typically save `LR` to the stack (often paired with frame pointer) because another `BL` would overwrite it.

10.4.4 RV64: Link Register Call/Return Model

RV64 also uses a link register mechanism:

- `jal` writes return address into `ra` and jumps,
- `ret` is an assembler pseudo-instruction (commonly `jalr x0, ra, 0`).

Nested calls require saving `ra` if it must be preserved across further calls.

10.5 x86-64 Windows: Extensive Call/Ret Examples

10.5.1 Example 1: Leaf Function (No Calls) With Minimal Stack Usage

A leaf function does not call other functions, so it often:

- uses only volatile registers,
- avoids saving LR/RA because x86 uses stack-return address automatically,
- may avoid stack allocation if not needed.

```
.intel_syntax noprefix
.text

/* u64 leaf_mul_add(u64 a, u64 b, u64 c)
   RCX=a, RDX=b, R8=c
   return a*b + c
*/
.global leaf_mul_add
leaf_mul_add:
    mov     rax, rcx
    imul    rax, rdx
    add     rax, r8
    ret
```

```
#[link(name = "myasm")]
extern "C" {
    fn leaf_mul_add(a: u64, b: u64, c: u64) -> u64;
}

fn main() {
    let r = unsafe { leaf_mul_add(3, 7, 5) };
    println!("3*7+5 = {}", r);
}
```

10.5.2 Example 2: Non-Leaf Function (Calls Another) With Volatile Preservation

This example demonstrates:

- a caller keeps a value in a volatile register across a call,
- so it saves it explicitly.

We implement:

```
f(a,b) = g(a) + (b*3)
```

and assume g clobbers volatile registers.

```
.intel_syntax noprefix
.text

/* u64 g(u64 x): returns x+10, clobbers volatile regs */
.global g
g:
    mov     rax, rcx
    xor     r10, r10                /* clobber a volatile reg */
    add     rax, 10
    ret

/* u64 f(u64 a, u64 b): returns g(a) + b*3 */
.global f
f:
    sub     rsp, 32                 /* maintain alignment; local space */

    mov     r10, rdx                /* r10 = b (volatile) */
    imul    r10, r10, 3             /* r10 = b*3 */
    mov     qword ptr [rsp], r10    /* save across call */
```

```
    call    g                       /* g(a), uses rcx as input, returns rax */

    mov     r10, qword ptr [rsp]    /* restore b*3 */
    add     rax, r10

    add     rsp, 32
    ret
```

```
#[link(name = "myasm")]
extern "C" {
    fn f(a: u64, b: u64) -> u64;
}

fn main() {
    let r = unsafe { f(5, 7) }; // g(5)=15, b*3=21, total=36
    println!("f(5,7) = {}", r);
}
```

10.5.3 Example 3: Building a Simple Loop With a Call Inside

We implement:

$$\texttt{sum_g(arr,n)} = \sum_{i=0}^{n-1} g(arr[i])$$

This demonstrates:

- loop control,
- call inside loop,
- preserving registers that hold loop state across the call.

We use:

- RCX = pointer

- RDX = count

The call uses RCX as argument to g, so we must preserve pointer and count. We will keep:

- pointer in R8 (volatile, but we control saving)
- count in R9 (volatile)
- sum in RAX

```
.intel_syntax noprefix
.text

/* u64 sum_g(const u64* p, u64 n)
   RCX=p, RDX=n
*/
.global sum_g
sum_g:
    sub     rsp, 32                 /* local space and alignment */
    xor     rax, rax                /* sum = 0 */
    mov     r8, rcx                 /* r8 = p */
    mov     r9, rdx                 /* r9 = n */
    test    r9, r9
    je      .done

.loop:
    mov     rcx, qword ptr [r8]     /* rcx = *p (argument to g) */

    /* Save loop state across call */
    mov     qword ptr [rsp+0], r8
    mov     qword ptr [rsp+8], r9
    mov     qword ptr [rsp+16], rax

    call    g                       /* rax = g(*p) */
```

```
    /* Restore loop state */
    mov     r8, qword ptr [rsp+0]
    mov     r9, qword ptr [rsp+8]
    mov     r10, qword ptr [rsp+16]  /* previous sum */
    add     rax, r10                 /* new sum */

    add     r8, 8                    /* p++ */
    dec     r9                       /* n-- */
    jne     .loop

.done:
    add     rsp, 32
    ret
```

```
#[link(name = "myasm")]
extern "C" {
    fn sum_g(p: *const u64, n: u64) -> u64;
}

fn main() {
    let a: [u64; 4] = [1, 2, 3, 4];
    let r = unsafe { sum_g(a.as_ptr(), a.len() as u64) };
    // g(x) = x+10 => sum = (1+10)+(2+10)+(3+10)+(4+10)=50
    println!("sum_g = {}", r);
}
```

10.6 Conditional Jumps: Pattern Cookbook

10.6.1 If/Else (x86-64)

```
cmp     rcx, rdx
jb      .less
```

```
/* else-part */
jmp     .done
.less:
/* then-part */
.done:
```

10.6.2 While Loop (x86-64)

```
test    rdx, rdx
je      .done
.loop:
    /* body */
    dec     rdx
    jne     .loop
.done:
```

10.6.3 Do-While Loop (x86-64)

```
.loop:
    /* body */
    dec     rdx
    jne     .loop
```

10.6.4 Switch-Like Jump Table (Conceptual)

Many compilers lower dense switches into jump tables:

- bounds check index,
- load target address from table,
- indirect jump.

x86-64 uses `jmp [table + index*8]` patterns. AArch64 uses `adrp/add/ldr/br` patterns. RV64 uses `auipc/add/ld/jalr` patterns.

10.7 Professional Checklist

Before shipping assembly that uses control flow:

- Keep flag lifetimes short: generate condition and consume it immediately.
- Use the correct signed/unsigned condition codes.
- For loops:
  - define loop invariant and update clearly,
  - avoid hidden state clobbered by calls.
- For calls:
  - preserve non-volatile registers you touch,
  - assume volatile registers are destroyed by callees,
  - maintain stack alignment at every call boundary,
  - never corrupt the return address or stack frame.
- Validate with stress tests:
  - repeated calls,
  - random inputs,
  - edge cases: zero length, large length, max/min integers.

10.8 Exercises (Foundation Level)

1. Implement `count_nonzero` for a u64 array using a loop and conditional branch.
2. Implement `clamp(x, lo, hi)` using branches; then implement a branchless version using conditional moves (x86-64) and compare results.
3. Implement a function that calls another function inside a loop and prove that it preserves loop state correctly by stress testing from Rust.
4. Write equivalent pseudo-assembly for AArch64 and RV64 for the sum loop and the find loop.

10.9 Professional Summary

Control flow is the assembly-level language of decisions. A professional model includes:

- conditional jumps built on flags (x86-64) or NZCV (AArch64) or explicit compare branches (RV64),
- loops as structured state updates plus conditional branches,
- call/ret as an ABI contract: return address mechanism, register preservation rules, and stack alignment discipline.

With these patterns, you can implement safe and efficient decisions, loops, and function composition in assembly and verify behavior using Rust on Windows.

Part III

ABI and Calling Conventions

What Is an ABI?

11.1 API vs ABI

An **API** (*Application Programming Interface*) is the interface you consume at the **source-code level**: function names, parameters, types, constants, macros, headers, and documented behavior. If you can compile against it, you are using an API.
An **ABI** (*Application Binary Interface*) is the interface you consume at the **machine-code level**: how arguments are passed, how return values are produced, how the stack is managed, how symbols are named in object files, how structures are laid out in memory, how alignment and padding work, what the object file format is, and what the dynamic loader expects.
In low-level engineering, you can change an API and still keep the ABI compatible (sometimes), and you can keep the same API but break the ABI easily.

11.1.1 What an ABI includes (practical checklist)

An ABI is not a single rule; it is a **bundle of contracts**. On modern systems it typically includes:

- **Calling convention**: which registers carry arguments, where extra arguments go (stack), which registers must be preserved by the callee, stack alignment, shadow space (Windows x64), red zone (SysV), etc.

- **Data layout**: size and alignment of fundamental types; struct field ordering; padding rules; bitfield layout; enum size rules; vector types rules.
- **Name mangling and symbol visibility**: how function/variable names become linker symbols; export/import rules.
- **Object format**: COFF/PE on Windows; ELF on most UNIX-like systems; relocation models.
- **Exception and unwinding model**: how stack unwinding works, which personality routines are used, whether exceptions may cross module boundaries safely.
- **Thread-local storage model**: how TLS variables are accessed.
- **Floating-point and SIMD rules**: which registers are used, what must be preserved, how aggregates containing floats are passed.
- **Dynamic linking rules**: calling into DLLs, import libraries, delay-load patterns, symbol resolution.

11.1.2 Why an assembler engineer must care

Assemblers and linkers sit **below** syntax-level APIs. They must output binaries that obey the platform ABI; otherwise:

- calls silently pass the wrong values;
- stack frames become misaligned (crashing inside unrelated code);
- registers are clobbered (corrupting state far away from the call site);
- dynamic linking fails (symbol mismatch);
- structs become incompatible (data corruption).

11.2 Binary compatibility

Binary compatibility means that a program (or plugin, or DLL consumer) built against an *older* binary interface continues to work with a *newer* binary implementation **without recompilation**. Binary compatibility is always an ABI question.

11.2.1 Binary compatibility vs source compatibility

- **Source compatibility**: old code still compiles (API compatible).
- **Binary compatibility**: old binaries still run (ABI compatible).

You can keep source compatibility but break binary compatibility, for example by changing a struct layout while keeping the same field names in headers.

11.2.2 The three most common causes of ABI breaks

1. **Calling convention mismatch** (registers/stack rules differ).
2. **Type layout drift** (struct size/alignment/packing changes).
3. **Symbol identity changes** (name mangling, exports, ordinals).

11.2.3 Example 1: same function signature, broken calling convention (Windows x64)

On **Windows x86-64**, the **Win64** ABI passes the first 4 integer/pointer arguments in: `RCX, RDX, R8, R9`. The caller also reserves **32 bytes of shadow space** on the stack.
If you accidentally assemble/call as if you were on SysV (Linux/macOS), you might put args in `RDI, RSI, RDX, RCX` and *not* reserve shadow space. The callee may still run, but it will read the wrong registers and may overwrite stack memory the caller did not reserve.

11.2.4 Example 2: same API type, broken layout

Consider a public struct in a DLL boundary:

```
typedef struct Vec3 {
    float x;
    float y;
    float z;
} Vec3;
```

If one side compiles with a packing change, or adds a field, or changes alignment assumptions, binaries will disagree on: **sizeof(Vec3)** and field offsets. The API text looks similar, but the ABI is broken.

11.2.5 Binary compatibility design patterns (what actually works)

In serious binary interfaces, avoid exposing layout-sensitive or compiler-sensitive constructs across boundaries.

Pattern A: C ABI + opaque handles

Expose only stable C-like functions and hide all internal types:

```
typedef struct ForgeVM_Context ForgeVM_Context;

ForgeVM_Context* forgevm_create(void);
void             forgevm_destroy(ForgeVM_Context* ctx);
int              forgevm_eval(ForgeVM_Context* ctx, const char* code_utf8);
```

Here the consumer never needs `sizeof(ForgeVM_Context)`. The layout can change freely without breaking the ABI.

Pattern B: versioned structs with size field

When you must expose a struct, add a **size** (and sometimes **version**) so the callee can validate:

```
typedef struct ForgeVM_Config {
    uint32_t size;       /* must be sizeof(ForgeVM_Config) */
    uint32_t version;    /* ABI version */
    uint32_t flags;
    uint32_t reserved;
} ForgeVM_Config;
```

Pattern C: function tables (stable vtable-like ABI)

A single exported function returns a table of function pointers:

```
typedef struct ForgeVM_API {
    uint32_t size;
    uint32_t abi_version;
    void*    (*create)(void);
    void     (*destroy)(void*);
    int      (*eval)(void*, const char*);
} ForgeVM_API;

const ForgeVM_API* forgevm_get_api(void);
```

Adding new functions can be done by extending the table *at the end* while keeping the front stable.

11.3 Why ABIs matter more than syntax

Syntax is what programmers write. ABI is what **machines must agree on**.
At low levels (assemblers, linkers, kernels, drivers, runtimes, plugins, language FFI), the ABI is the real contract. If the ABI contract is correct, very different languages can interoperate. If it is wrong, even the same language can fail when mixing compilers or settings.

11.3.1 A mental model

- **API answers**: "How do I call this functionality in code?"
- **ABI answers**: "What binary sequence must exist so the call works?"

11.3.2 Where ABI dominates engineering reality

- **Cross-language calls** (C $\leftrightarrow$ Rust, C $\leftrightarrow$ assembly).
- **Plugins and SDKs** distributed as binary libraries.
- **OS boundaries**: system calls and kernel ABIs.
- **Runtime interoperability**: embedding, JITs, VM toolchains.
- **Long-lived ecosystems** where recompilation is not guaranteed.

11.3.3 Windows reality: one platform, multiple ABIs

Windows has multiple important ABI surfaces:

- **Win64 ABI** (x86-64 Windows calling convention, shadow space, register set).
- **Win32 API ABI** (x86 uses stdcall historically, x64 uses the unified Win64 convention).
- **Arm64EC ABI** (Windows-on-Arm interoperability constraints).

A modern assembly engineer must treat ABI knowledge as **core literacy**, not as an optional detail.

11.4 Windows x64 ABI essentials (the rules you must not violate)

This section focuses on Windows x86-64 because your toolchain examples here are **Rust on Windows** and **GAS Intel syntax**.

11.4.1 Argument passing (Win64)

For integer/pointer arguments:

- arg1 → `RCX`
- arg2 → `RDX`
- arg3 → `R8`
- arg4 → `R9`
- arg5+ → stack (above the 32-byte shadow space)

Floating-point arguments typically use XMM registers (XMM0..XMM3 for the first four FP args), but you must also respect how mixed integer/FP signatures are classified by the ABI.

11.4.2 Shadow space

The caller allocates **32 bytes** of stack space before `call`, even if the callee uses fewer than four arguments. Many callees assume it exists and may spill registers into it.

11.4.3 Stack alignment

On Win64, the stack must be **16-byte aligned** at a call boundary (with the ABI-defined effect of the pushed return address). In practice: before `call`, arrange the stack so the callee sees the expected alignment.

11.4.4 Volatile vs non-volatile registers (Win64)

Caller-saved (volatile): `RAX, RCX, RDX, R8, R9, R10, R11` (and XMM0..XMM5 commonly treated as volatile).
Callee-saved (non-volatile): `RBX, RBP, RDI, RSI, R12, R13, R14, R15` (and XMM6..XMM15 are non-volatile in Win64).
If your assembly function modifies a non-volatile register, it must restore it.

11.5 Extensive Windows examples (Rust + C + GAS Intel syntax)

All examples below target **Windows x86-64** and intentionally use **C ABI / Win64 ABI** so that: Rust, C, and assembly can interoperate reliably.

11.5.1 Example A: A Win64 ABI assembly function called from Rust

A.1 Assembly file (GAS, Intel syntax)

Save as `add5_win64.S`:

```
.intel_syntax noprefix
.text
.globl add5_win64
add5_win64:
```

```
    /* Win64 ABI:
       a,b,c,d are in RCX,RDX,R8,R9
       e is on stack at [rsp + 40] on entry:
         [rsp+0]  = return address
         [rsp+8..+39] = shadow space (32 bytes)
         [rsp+40] = 5th argument (e)
    */

    mov rax, rcx
    add rax, rdx
    add rax, r8
    add rax, r9
    add rax, qword ptr [rsp + 40]
    ret
```

A.2 Rust caller (Windows)

Key ABI choice: On Windows x86-64, `extern "C"` uses the Win64 ABI.

```
#[link(name = "add5lib")]
extern "C" {
    fn add5_win64(a: i64, b: i64, c: i64, d: i64, e: i64) -> i64;
}

fn main() {
    unsafe {
        let r = add5_win64(10, 20, 30, 40, 50);
        println!("add5_win64 = {}", r);
    }
}
```

A.3 Building on Windows (MinGW-w64 toolchain)

You can build the assembly into a static library and link from Rust.

```
gcc -c -o add5_win64.o add5_win64.S
ar rcs libadd5lib.a add5_win64.o
```

Then in Rust:

```
# Cargo.toml
[package]
name = "abi_demo"
version = "0.1.0"
edition = "2021"

[build-dependencies]
```

And provide linker search path (one simple approach is to place the library where the linker can find it or use RUSTFLAGS):

```
set RUSTFLAGS=-L .
cargo run
```

11.5.2 Example B: Exporting a stable C ABI from Rust as a Windows DLL

B.1 Why this matters

If you ship a DLL to third parties, **they cannot recompile** when you change internals. Your exported surface must be ABI-stable.
We will export a **C ABI** API from Rust:

- avoid Rust-native layouts across the boundary;
- use `#[repr(C)]` for structs that cross the boundary;
- use `#[no_mangle]` to keep symbol names stable;
- return error codes, not Rust panics.

B.2 Cargo configuration

```
# Cargo.toml
[package]
name = "forgeabi"
version = "0.1.0"
edition = "2021"

[lib]
crate-type = ["cdylib"]
```

B.3 Rust DLL exports (Windows-safe ABI surface)

```
use core::ffi::c_void;
use core::ptr;

#[repr(C)]
pub struct ForgeCtx {
    magic: u32,
    counter: u64,
}

/* Exported: create/destroy via opaque pointer */
#[no_mangle]
pub extern "C" fn forge_create() -> *mut c_void {
    let ctx = Box::new(ForgeCtx { magic: 0xF0RGE001, counter: 0 });
    Box::into_raw(ctx) as *mut c_void
}

#[no_mangle]
pub extern "C" fn forge_destroy(p: *mut c_void) {
    if p.is_null() { return; }
    unsafe {
        let _ = Box::from_raw(p as *mut ForgeCtx);
```

```
    }
}

/* Exported: stable function using only fixed-size primitives */
#[no_mangle]
pub extern "C" fn forge_add_u64(p: *mut c_void, x: u64, y: u64, out: *mut u64) -> i32 {
    if p.is_null() || out.is_null() {
        return -1;
    }
    unsafe {
        let ctx = &mut *(p as *mut ForgeCtx);
        if ctx.magic != 0xF0RGE001 {
            return -2;
        }
        ctx.counter = ctx.counter.wrapping_add(1);
        *out = x.wrapping_add(y);
    }
    0
}

/* Exported: UTF-8 input contract (Windows-friendly) */
#[no_mangle]
pub extern "C" fn forge_strlen_utf8(s: *const u8, len: usize, out: *mut usize) -> i32 {
    if s.is_null() || out.is_null() { return -1; }
    let bytes = unsafe { core::slice::from_raw_parts(s, len) };
    /* Validate UTF-8 without allocating */
    if core::str::from_utf8(bytes).is_err() { return -2; }
    unsafe { *out = bytes.len(); }
    0
}
```

B.4 C consumer (Windows)

```c
#include <stdint.h>
#include <stdio.h>

__declspec(dllimport) void* forge_create(void);
__declspec(dllimport) void  forge_destroy(void*);

__declspec(dllimport) int32_t forge_add_u64(void* ctx, uint64_t x, uint64_t y, uint64_t*
↪  out);
__declspec(dllimport) int32_t forge_strlen_utf8(const uint8_t* s, size_t len, size_t* out);

int main(void) {
    void* ctx = forge_create();
    uint64_t r = 0;

    if (forge_add_u64(ctx, 40, 2, &r) == 0) {
        printf("40 + 2 = %llu\n", (unsigned long long)r);
    }

    const char* msg = "hello";
    size_t n = 0;
    if (forge_strlen_utf8((const uint8_t*)msg, 5, &n) == 0) {
        printf("utf8 len = %zu\n", n);
    }

    forge_destroy(ctx);
    return 0;
}
```

B.5 ABI rules enforced by this design

- The consumer never relies on `sizeof(ForgeCtx)` because it is hidden behind `void*`.

- All exported functions use **fixed-size primitives** and pointers.
- The exported names are stable (`#[no_mangle]`).
- No exceptions/panics are allowed to cross the boundary (return codes are used).

11.5.3 Example C: The classic ABI pitfall (struct layout across boundary)

C.1 Wrong approach (layout exposed)

```
#[repr(C)]
pub struct Bad {
    pub a: u8,
    pub b: u64,
}
```

This looks safe because of `repr(C)`, but it still bakes in: alignment rules, padding, and field offsets. If you later reorder fields, insert a new field, or change types, you break binary compatibility.

C.2 Better approach (versioned layout)

```
#[repr(C)]
pub struct GoodV1 {
    pub size: u32,
    pub version: u32,
    pub a: u8,
    pub reserved0: [u8; 7],
    pub b: u64,
}

#[no_mangle]
pub extern "C" fn good_init_v1(p: *mut GoodV1) -> i32 {
```

```
    if p.is_null() { return -1; }
    unsafe {
        if (*p).size as usize != core::mem::size_of::<GoodV1>() { return -2; }
        (*p).version = 1;
    }
    0
}
```

Now the callee can validate the struct size. Future versions can add fields at the end while still recognizing older layouts.

11.6 ABI discipline rules for a professional toolchain

When you build assembler/linker ecosystems (and especially when you want your components to be callable from many languages), adopt these rules:

1. Export a **C ABI** surface for interoperability.
2. Do not export C++ classes, templates, references, exceptions, or STL types across a stable boundary.
3. Do not export Rust-native types across a stable boundary (use `repr(C)` and opaque handles).
4. Define ownership rules: who allocates, who frees. Never free memory across module boundaries unless you guarantee the same allocator contract.
5. Avoid passing `bool` and `enum` unless you fix their size explicitly (e.g., `uint8_t`, `int32_t`).
6. Make all exported structs versioned or opaque.
7. Keep calling convention explicit when needed (on Rust you can use `extern "C"` / `extern "win64"`).

8. Freeze symbol names (`no_mangle`) and define stable exports for DLL builds.
9. Treat ABI changes as **breaking changes** even if the API text looks similar.

11.7 Summary

- **API** is a source-level contract; **ABI** is a binary-level contract.
- Binary compatibility is fundamentally an ABI problem.
- ABIs matter more than syntax whenever prebuilt binaries must interoperate.
- On Windows x64, the Win64 calling convention rules (register arguments, shadow space, stack alignment, preserved registers) are non-negotiable.
- A robust ABI strategy for professional toolchains is: **C ABI + opaque handles + versioned data + explicit ownership + stable symbol exports**.

Stack Frames and Prolog/Epilog Patterns

12.1 Stack Growth

12.1.1 What “stack growth” means in real machines

On almost all mainstream CPU/OS ABIs you will meet in practice, the stack is a contiguous region of virtual memory used for:

- return addresses (control-flow linkage),
- spilled registers (temporary preservation),
- local variables (automatic storage),
- outgoing arguments (when registers are insufficient),
- stack-unwinding metadata (directly or indirectly, depending on platform),
- alignment padding and ABI-mandated areas (for example, shadow space on Windows x64).

The key operational fact is: **the stack pointer changes in a consistent direction as frames are allocated and freed.**

12.1.2 x86 / x86-64: downward-growing stack

On x86 and x86-64, the stack **grows downward** toward lower addresses:

- allocating stack space means **subtracting** from `RSP` (or `ESP` in 32-bit),
- freeing stack space means **adding** to `RSP`.

Canonical picture (x86-64)

Assume a function is called and the CPU pushes the return address. The stack looks like:

```
High addresses
    ...
    [caller locals / spills]
    [caller outgoing args ...]
    [32-byte shadow space]     (Windows x64 only; allocated by caller)
RSP -> return address          (pushed by CALL)
Low addresses
```

If the callee needs locals/spills, it will allocate by moving `RSP` downward.

12.1.3 Why stack direction matters to low-level engineers

Stack direction affects:

- how you compute argument locations on the stack,
- where you can place temporaries,
- what memory is safe to touch without moving `RSP`,
- whether special ABI areas exist *below* or *above* `RSP`.

A correct mental model is essential when you write assembly that interoperates with Rust on Windows, because Rust will rely on the platform ABI rules for stack alignment, register volatility, and stack usage around calls.

12.2 Frame Pointer Usage

12.2.1 The two "pointers" in a stack frame

In x86-64 code generation you will see two common registers involved:

- **Stack Pointer (SP)**: `RSP`. It marks the current top of stack.
- **Frame Pointer (FP)**: usually `RBP` (on x86-64). It can be used as a stable base to reference locals/args.

12.2.2 What a frame pointer gives you

Using a dedicated frame pointer typically provides:

- **Stable addressing**: locals at constant offsets from `RBP`.
- **Simpler debugging**: stack traces can be reconstructed reliably when frame pointers are present.
- **Conservative unwinding**: some tooling benefits (or requires) predictable frame chains.

But it costs:

- one general-purpose register (`RBP`) becomes unavailable for allocation,
- slightly larger prolog/epilog,
- potentially reduced optimization freedom (though modern unwind metadata can remove the need for FP on many targets).

12.2.3 Frame-pointer prolog/epilog: classic pattern

The classic pattern on x86-64 looks like:

```
.intel_syntax noprefix
.text
.globl fp_function
fp_function:
    /* Prolog */
    push rbp
    mov  rbp, rsp
    sub  rsp, 32          /* locals (example) */

    /* body ... locals at [rbp - offset] */

    /* Epilog */
    add  rsp, 32
    pop  rbp
    ret
```

12.2.4 Frame-pointer omission (FPO) and “RSP-relative frames”

Optimizing compilers frequently omit the frame pointer and address locals relative to RSP directly:

```
.intel_syntax noprefix
.text
.globl fpo_function
fpo_function:
    sub  rsp, 40          /* locals + alignment padding (example) */

    /* body ... locals at [rsp + offset] */
```

```
    add  rsp, 40
    ret
```

This is valid and often faster/denser, but offsets depend on RSP movement (and can be harder for humans to follow). On modern platforms, correct unwinding is usually achieved via unwind metadata rather than frame-pointer chains.

12.2.5 Windows x64 nuance: unwinding and prolog constraints

On Windows x64, stack unwinding is not "best effort"; it is part of the platform contract. Real production code needs unwindability for exceptions, debugging, and asynchronous events. Practically:

- many Windows toolchains generate structured unwind info,
- hand-written assembly must be careful: a trivial leaf function can be safe, but non-trivial frames may require proper unwind directives depending on how you build and link.

In this chapter we focus on **ABI-correct prolog/epilog shapes** and concrete stack rules; if you write larger Windows assembly routines meant to interoperate with exceptions/unwinding, you must also provide correct unwind metadata in the format expected by your toolchain.

12.3 Red Zone vs Shadow Space

12.3.1 Red zone (System V AMD64 ABI concept)

On many UNIX-like systems using the **System V AMD64 ABI** (common on Linux and other ELF-based environments), there is a concept called the **red zone**:

- it is a small region **below** the current stack pointer (RSP) that leaf functions may use **without adjusting** RSP,

- it exists to enable small, fast leaf functions to use stack storage without a prolog/epilog that modifies `RSP`.

The red zone is **not** a universal x86-64 rule. It is an ABI choice.

12.3.2 Shadow space (Windows x64 ABI concept)

On **Windows x64** there is **no red zone**. Instead, Windows defines **shadow space**:

- the **caller** reserves **32 bytes** of stack space *before every call*,
- the callee may use this area to spill the first four register arguments if needed,
- this space exists even if the function takes fewer than four parameters.

This is a major reason Windows x64 assembly that was written with SysV assumptions often breaks: it may not allocate what the callee expects, and it may incorrectly assume a red zone is available.

12.3.3 Concrete comparison (what you can touch without moving RSP)

System V AMD64 (typical Linux)

- leaf function may use a red-zone region below `RSP` without changing `RSP` (ABI-defined),
- no caller-mandated shadow space concept like Windows x64.

Windows x64

- **no red zone** (do not touch memory below `RSP` unless you allocated it),
- caller allocates **32-byte shadow space** for the callee *above* the return address area in the call frame,
- stack must satisfy the ABI alignment rules at call boundaries.

12.3.4 Why this matters for Rust on Windows

Rust uses the platform ABI for `extern "C"` calls and for its own code generation on Windows. So:

- when Rust calls your assembly, it will expect **Win64 shadow space rules** to hold for calls *you make* inside assembly,
- your assembly must preserve non-volatile registers and maintain stack alignment,
- you must not assume a red zone exists on Windows x64.

12.4 Prolog/Epilog Patterns on Windows x64 (practical and correct)

In the following examples, we use **GAS Intel syntax** and target **Windows x86-64** (MinGW-w64 style toolchains). Comments are written using `/* ... */`.

12.4.1 Pattern 1: leaf function, no locals, no calls

If a function:

- uses only volatile registers (or preserves non-volatiles it touches),
- does not call other functions,
- does not need stack locals,

then it may not need to adjust `RSP` at all.

```
.intel_syntax noprefix
.text
```

```
.globl leaf_add_u64
leaf_add_u64:
    /* extern "C" fn(a: u64, b: u64) -> u64
       Win64: a in RCX, b in RDX, return in RAX */
    mov rax, rcx
    add rax, rdx
    ret
```

12.4.2 Pattern 2: non-leaf function (makes a call) with shadow space and alignment

When your assembly function calls another function, you must:

- ensure stack alignment is correct at the call boundary,
- allocate required locals (if any),
- ensure the **32-byte shadow space** exists for the callee you are calling.

A robust and easy-to-reason approach is to allocate:

- 32 bytes shadow space (for the call you are about to make),
- plus any local space you need,
- plus optional padding to keep alignment correct.

```
.intel_syntax noprefix
.text
.extern puts
.globl call_puts_demo
call_puts_demo:
    /* extern "C" fn() -> i32
       Calls puts(const char*) */
```

```
    /* Prolog: allocate 32-byte shadow space + 8 bytes padding to keep alignment simple */
    sub rsp, 40

    /* RCX = pointer to string (1st arg in Win64) */
    lea rcx, [rip + msg]
    call puts

    /* Epilog */
    add rsp, 40
    ret

.section .rdata,"dr"
msg:
    .asciz "Hello from Win64 ABI assembly"
```

Why `sub rsp, 40`? The extra 8 bytes are commonly used as padding so that the stack alignment at the call boundary remains correct given the pushed return address. The precise arithmetic depends on the ABI alignment rule and the state of RSP at entry, but this pattern is widely used because it keeps a simple invariant: allocate shadow space and keep alignment consistent for calls.

12.4.3 Pattern 3: frame pointer based (debug-friendly) function with locals

When you want a stable base register, you can use RBP as a frame pointer:

```
.intel_syntax noprefix
.text
.globl fp_locals_demo
fp_locals_demo:
    /* Prolog */
    push rbp
```

```
    mov  rbp, rsp

    /* Allocate locals + shadow space for potential calls.
       Example: 32 shadow + 32 locals = 64 bytes. */
    sub  rsp, 64

    /* Example local store: [rbp - 8] */
    mov  qword ptr [rbp - 8], 1234

    /* If you call another function, RCX/RDX/R8/R9 hold args, shadow space is present */
    /* ... */

    /* Epilog */
    add  rsp, 64
    pop  rbp
    ret
```

12.4.4 Pattern 4: saving non-volatile registers

On Windows x64, if you modify non-volatile registers (`RBX, RBP, RDI, RSI, R12-R15`), you must restore them.

```
.intel_syntax noprefix
.text
.globl save_nv_demo
save_nv_demo:
    /* Prolog */
    push rbx              /* RBX is non-volatile */
    sub  rsp, 40          /* shadow + padding for calls (if needed) */

    /* body */
    mov  rbx, rcx         /* safe now: we will restore RBX */
    /* ... maybe call other functions ... */
```

```
    /* Epilog */
    add  rsp, 40
    pop  rbx
    ret
```

12.5 Rust on Windows: calling these patterns safely

12.5.1 Rust declarations (Windows x64)

On Windows x86-64, `extern "C"` uses the Win64 ABI.

```
extern "C" {
    fn leaf_add_u64(a: u64, b: u64) -> u64;
    fn call_puts_demo() -> i32;
    fn fp_locals_demo();
    fn save_nv_demo();
}

fn main() {
    unsafe {
        let r = leaf_add_u64(10, 32);
        println!("leaf_add_u64(10,32) = {}", r);

        let _ = call_puts_demo();

        fp_locals_demo();
        save_nv_demo();
    }
}
```

12.5.2 Rust + global_asm! (Windows-friendly packaging)

If you want to keep assembly inside the Rust crate (still using Intel syntax), you can embed assembly with `global_asm!`. This approach keeps everything in one build system and is often practical for toolchain components.

```
#![allow(non_snake_case)]

use core::arch::global_asm;

global_asm!(r#"
.intel_syntax noprefix
.text
.globl leaf_add_u64
leaf_add_u64:
    mov rax, rcx
    add rax, rdx
    ret
"#);

extern "C" {
    fn leaf_add_u64(a: u64, b: u64) -> u64;
}

fn main() {
    unsafe {
        let r = leaf_add_u64(7, 5);
        println!("7 + 5 = {}", r);
    }
}
```

Engineering note (Windows ABI discipline). Even when embedding assembly, the ABI rules do not change:

- argument registers are `RCX, RDX, R8, R9`,
- return value in `RAX`,
- preserve non-volatile registers,
- do not assume a red zone,
- maintain correct stack alignment and provide shadow space before making calls.

12.6 Common mistakes and how to avoid them

12.6.1 Mistake 1: treating Windows like SysV (red zone assumption)

On Windows x64:

- **do not** write to memory below `RSP` unless you allocated it,
- do not assume a compiler will keep red-zone memory intact (it is not part of the Windows ABI).

12.6.2 Mistake 2: forgetting shadow space before a call

If your assembly function calls another function and you do not reserve the 32-byte shadow space, you may corrupt the stack frame or crash unpredictably.

12.6.3 Mistake 3: clobbering non-volatile registers

A function may appear to work in small tests but fail when called from optimized Rust code that assumes ABI preservation.

12.6.4 Mistake 4: breaking stack alignment

Misalignment can crash inside SIMD instructions or inside library code that assumes alignment. Always treat call-boundary alignment as mandatory.

12.7 Summary

- On x86-64, the stack grows downward: allocate by subtracting from `RSP`.
- The frame pointer (`RBP`) can simplify debugging and stable addressing but costs a register.
- SysV and Windows differ sharply: SysV may provide a red zone; Windows x64 provides caller-allocated 32-byte shadow space and **no red zone**.
- For Rust on Windows, `extern "C"` uses Win64 ABI: respect register arguments, shadow space, non-volatile registers, and stack alignment.
- Correct prolog/epilog patterns are the foundation of reliable interop between Rust, C, and hand-written assembly.

System V AMD64 ABI

13.1 Overview and Scope

This chapter describes the **System V AMD64 ABI** (often called **SysV x86-64 ABI**), which is the dominant ABI for **x86-64 on UNIX-like systems** (most notably Linux). It defines how compiled code must behave at the binary boundary:

- how parameters are passed,
- how return values are produced,
- how the stack is aligned and used at call boundaries,
- how aggregates (structs/unions) are classified and passed/returned.

Important note for a Windows-based Rust workflow. Rust code can be written and built on Windows while **targeting a SysV platform** via cross-compilation (e.g., `x86_64-unknown-linux-gnu`). The ABI described here is **not the Windows x64 ABI**. If you compile for Windows targets, Rust uses the Win64 calling convention instead.

13.2 Parameter Passing

13.2.1 Register classes and the two parallel channels

SysV AMD64 ABI effectively uses two parallel argument channels:

- **Integer / pointer channel** using general-purpose registers (GPRs).
- **Floating-point / vector channel** using SIMD registers (XMM).

Arguments are assigned left-to-right to the appropriate channel.

13.2.2 Integer and pointer parameters (GPR order)

For integer-like arguments (integers, pointers, addresses, many enums, and integer parts of aggregates classified as INTEGER), the first six are passed in:

Arg #	Register
1	`RDI`
2	`RSI`
3	`RDX`
4	`RCX`
5	`R8`
6	`R9`

If there are more than six integer-like arguments, the remaining ones are passed on the stack.

13.2.3 Floating-point parameters (XMM order)

For scalar floating-point arguments (`float`, `double`) and many vector types, the first eight are passed in:

FP Arg #	Register
1	XMM0
2	XMM1
3	XMM2
4	XMM3
5	XMM4
6	XMM5
7	XMM6
8	XMM7

Extra floating-point arguments go on the stack.

13.2.4 Mixed argument lists (two-channel allocation)

A mixed signature uses both channels in parallel. Example:

```
long f(int a, double b, long c, float d, void* p);
```

Allocation:

- a (int) → EDI (in RDI)
- b (double) → XMM0
- c (long) → RSI
- d (float) → XMM1
- p (pointer) → RDX

13.2.5 Stack-passed arguments (when registers overflow)

Once a channel's register budget is exhausted:

- further integer-like args spill to the stack,
- further FP-like args spill to the stack.

Placement. Stack arguments are placed by the caller so that the callee can address them at fixed offsets from `RSP` after the call. The exact offsets depend on how much the callee subtracts from `RSP` in its prolog and whether it uses a frame pointer.

13.2.6 Varargs (C-style . . .) and the `AL` rule

For SysV varargs calls, a critical rule exists:

- The caller sets `AL` to the number of vector (XMM) registers used to pass arguments to the callee.

This enables the callee to correctly find register-passed floating arguments when using `va_start`/`va_arg`.

13.3 Return Values

13.3.1 Integer and pointer returns

- Up to 64-bit integer/pointer return values are in `RAX`.
- Some wider integer returns (e.g., 128-bit) may use `RDX:RAX` (high:low), depending on language and type.

13.3.2 Floating-point returns

- `float` and `double` return values are in `XMM0`.
- Some two-register FP returns may use `XMM0` and `XMM1` (e.g., certain small aggregates classified as SSE).

13.3.3 Small struct returns (in registers) vs large struct returns (sret)

SysV can return some small aggregates in registers, but larger or more complex aggregates are returned indirectly:

- **Register return**: many aggregates of total size up to 16 bytes (subject to classification rules) can be returned in `RAX/RDX` or `XMM0/XMM1`.
- **Indirect return (sret)**: if the aggregate is classified as `MEMORY`, the caller allocates space and passes a hidden pointer (the **sret pointer**) in `RDI`; the function returns that pointer in `RAX` (common convention) or just returns normally while writing through the pointer.

13.4 Stack Alignment Rules

13.4.1 The 16-byte alignment contract

SysV AMD64 requires that the stack is **16-byte aligned at call boundaries** in the following practical sense:

- The caller arranges the stack so that `RSP` is 16-byte aligned *before* executing `CALL`.
- Since `CALL` pushes an 8-byte return address, the callee entry typically sees `RSP` misaligned by 8 relative to 16.

- Therefore, if the callee wants 16-byte alignment for local aligned data or to call other functions, it typically subtracts an amount that restores a 16-byte aligned RSP before making further calls.

13.4.2 Red zone (SysV) versus Windows shadow space

SysV provides a **red zone**:

- A 128-byte area **below** the current RSP that leaf functions may use without adjusting RSP.

SysV **does not** define Windows-style shadow space. If you write portable assembly, treat **red zone as SysV-only** and **shadow space as Win64-only**.

13.4.3 Canonical SysV leaf function using red zone

This function uses 32 bytes of temporary space below RSP without modifying RSP:

```
.intel_syntax noprefix
.text
.globl leaf_redzone_demo
leaf_redzone_demo:
    /* SysV: leaf function may use 128-byte red zone below RSP */
    /* Store temporary data at [rsp - 8], [rsp - 16], ... */
    mov qword ptr [rsp - 8], rdi     /* save arg1 temporarily */
    mov rax, qword ptr [rsp - 8]
    add rax, 1
    ret
```

Warning. Do not use this pattern on Windows x64. Windows has no red zone.

13.4.4 Non-leaf SysV function: align stack before call

When calling another function, align and allocate your locals:

```
.intel_syntax noprefix
.text
.extern puts
.globl sysv_call_puts
sysv_call_puts:
    /* SysV: first arg in RDI */
    /* Ensure 16-byte alignment before CALL.
       On entry, RSP is typically 8 mod 16 (because return address was pushed).
       Subtract 8 to make it 0 mod 16, then reserve local space if desired. */
    sub rsp, 8

    lea rdi, [rip + msg]
    call puts

    add rsp, 8
    xor eax, eax
    ret

.section .rodata
msg:
    .asciz "Hello from SysV AMD64 ABI"
```

13.5 Struct Passing

13.5.1 The SysV “classification” idea (eightbytes)

SysV uses a precise classification algorithm for aggregates:

- The aggregate is conceptually split into 8-byte chunks called **eightbytes**.
- Each eightbyte is classified into classes such as `INTEGER`, `SSE`, or `MEMORY`, among others.
- If the result is `MEMORY`, the aggregate is passed on the stack (or indirectly for return).

- Otherwise, the aggregate may be passed in registers by assigning each eightbyte to a GPR or XMM register depending on its class.

For practical engineering, most everyday rules you need can be summarized safely as:

- small plain aggregates (up to 16 bytes) of integer-like fields often go in GPRs,
- small aggregates of floating fields often go in XMM regs,
- anything large, oddly aligned, containing unsupported members, or exceeding the register-return constraints becomes `MEMORY`.

13.5.2 Example 1: struct of two u64 passed in registers

C-like type

```
typedef struct PairU64 { unsigned long a; unsigned long b; } PairU64;
unsigned long sum_pair(PairU64 p);
```

SysV argument mapping

- `p.a` → `RDI`
- `p.b` → `RSI`

Assembly implementation

```
.intel_syntax noprefix
.text
.globl sum_pair
sum_pair:
    /* SysV: struct {u64,u64} often passed as two INTEGER eightbytes in RDI,RSI */
    mov rax, rdi
    add rax, rsi
    ret
```

13.5.3 Example 2: struct with two doubles passed in XMM regs

Type

```
typedef struct PairF64 { double x; double y; } PairF64;
double sum_pairf(PairF64 p);
```

SysV mapping

Often:

- `p.x` → `XMM0`
- `p.y` → `XMM1`

Assembly implementation

```
.intel_syntax noprefix
.text
.globl sum_pairf
sum_pairf:
    /* SysV: two doubles often in XMM0, XMM1 */
    addsd xmm0, xmm1
    ret
```

13.5.4 Example 3: small mixed struct passed across channels

Type

```
typedef struct Mix { int a; double b; } Mix;
long mix_f(Mix m);
```

Depending on classification, parts may land in a GPR and an XMM register. A common outcome:

- integer part → `RDI` (lower bits carry `a`)
- FP part → `XMM0` (carries `b`)

Assembly implementation (one robust interpretation)

```
.intel_syntax noprefix
.text
.globl mix_f
mix_f:
    /* Interpret a from EDI, b from XMM0 */
    mov eax, edi            /* eax = a (sign/zero extension depends on type rules) */
    cvtsi2sd xmm1, eax      /* xmm1 = (double)a */
    addsd xmm0, xmm1        /* xmm0 = b + (double)a */
    /* Return long: convert double to signed integer in RAX */
    cvttsd2si rax, xmm0
    ret
```

Engineering note. Exact integer sign-extension and classification details depend on the C type widths and signedness. The ABI defines how the bytes are delivered; the language ABI and compiler define how to interpret signedness.

13.5.5 Example 4: large struct passed by memory (stack)

Type

```
typedef struct Big {
    unsigned long a;
    unsigned long b;
    unsigned long c;
} Big; /* 24 bytes */
unsigned long big_sum(Big x);
```

Since 24 bytes exceeds the common 16-byte register passing window, it is typically passed in memory:

- caller places x on the stack,
- callee reads it from the stack frame.

Assembly sketch (offsets depend on frame)

```
.intel_syntax noprefix
.text
.globl big_sum
big_sum:
    /* Typical non-leaf frame with RBP for stable offsets */
    push rbp
    mov  rbp, rsp

    /* Stack layout at entry (conceptual):
       [rbp+8]  return address
       [rbp+16] first stack arg begins (Big.a)
       [rbp+24] Big.b
       [rbp+32] Big.c
    */

    mov rax, qword ptr [rbp + 16]
    add rax, qword ptr [rbp + 24]
    add rax, qword ptr [rbp + 32]

    pop rbp
    ret
```

13.6 Extensive Rust Examples (written on Windows, targeting SysV)

13.6.1 Rust ABI declarations

On a SysV target, `extern "C"` follows SysV AMD64 calling convention.

```
extern "C" {
    fn sum_pair(a: u64, b: u64) -> u64;
    fn sum_pairf(x: f64, y: f64) -> f64;
}

fn main() {
    unsafe {
        let s = sum_pair(10, 20);
        println!("sum_pair = {}", s);

        let d = sum_pairf(1.25, 2.75);
        println!("sum_pairf = {}", d);
    }
}
```

13.6.2 Rust `#[repr(C)]` structs for SysV interop

```
#[repr(C)]
#[derive(Copy, Clone)]
pub struct PairU64 {
    pub a: u64,
    pub b: u64,
}

#[repr(C)]
```

```
#[derive(Copy, Clone)]
pub struct PairF64 {
    pub x: f64,
    pub y: f64,
}

extern "C" {
    fn sum_pair_struct(p: PairU64) -> u64;
    fn sum_pairf_struct(p: PairF64) -> f64;
}

fn main() {
    unsafe {
        let p = PairU64 { a: 7, b: 9 };
        let r = sum_pair_struct(p);
        println!("sum_pair_struct = {}", r);

        let q = PairF64 { x: 0.5, y: 0.25 };
        let z = sum_pairf_struct(q);
        println!("sum_pairf_struct = {}", z);
    }
}
```

13.6.3 Assembly implementations matching SysV (GAS Intel syntax)

```
.intel_syntax noprefix
.text

.globl sum_pair_struct
sum_pair_struct:
    /* SysV: PairU64 likely arrives with a in RDI, b in RSI */
    mov rax, rdi
    add rax, rsi
```

```
    ret

.globl sum_pairf_struct
sum_pairf_struct:
    /* SysV: PairF64 likely arrives with x in XMM0, y in XMM1 */
    addsd xmm0, xmm1
    ret
```

13.6.4 Cross-compilation notes (Windows host)

A common Windows-host workflow is:

- write Rust and assembly on Windows,
- build for a SysV target using a cross toolchain,
- run under a Linux environment (VM, container, or remote machine).

The crucial engineering rule is: **the ABI is determined by the target triple, not by the host OS.**

13.7 Practical ABI Summary for SysV AMD64

13.7.1 Parameter passing

- Integer/pointer args: `RDI, RSI, RDX, RCX, R8, R9`, then stack.
- FP/vector args: `XMM0..XMM7`, then stack.
- Varargs: caller sets `AL` = number of XMM regs used for args.

13.7.2 Return values

- Integer/pointer: `RAX`.
- FP: `XMM0`.
- Small aggregates may return in `RAX/RDX` or `XMM0/XMM1` depending on classification.
- Large/complex aggregates: indirect return via hidden sret pointer (commonly in `RDI`).

13.7.3 Stack

- Maintain 16-byte alignment at call boundaries (caller responsibility).
- Red zone: 128 bytes below `RSP` usable by leaf functions (SysV-only).
- No shadow space concept (Win64-only).

13.7.4 Struct passing

- Up to 16 bytes often passed in registers, but exact behavior is determined by SysV classification rules.
- Larger aggregates typically passed in memory (stack).
- Prefer `#[repr(C)]` and simple fields for cross-language stability.

Microsoft x64 ABI

14.1 Purpose and Scope

The **Microsoft x64 ABI** (also called the **Win64 calling convention**) is the binary contract used by 64-bit Windows for user-mode code generation across toolchains (MSVC, Clang/LLVM targeting Windows, MinGW-w64, Rust targets such as `x86_64-pc-windows-msvc` and `x86_64-pc-windows-gnu`). It defines:

- how parameters and return values flow through registers and stack,
- which registers must be preserved by the callee,
- the **32-byte shadow space** rule,
- stack alignment rules at call boundaries,
- practical interop constraints for mixed-language systems (Rust, C, and assembly).

14.2 Shadow Space

14.2.1 Definition

The Microsoft x64 ABI requires that the **caller** reserves **32 bytes** of stack space *for every call*. This 32-byte area is commonly called:

- **shadow space**, or
- **home space**, or
- **parameter home area**.

It conceptually provides storage for the first four register arguments:

- `RCX` home at `[rsp+0x20]` relative to the *callee entry* view (conceptually),
- `RDX` home,
- `R8` home,
- `R9` home,

but you must treat the exact offsets carefully because they depend on where `RSP` points in the specific prolog/epilog shape. The key invariant is simpler and is the one you must enforce as an assembly author:

> **Before executing `call`, the caller must have at least 32 bytes reserved above the return address for the callee.**

14.2.2 Why shadow space exists (engineering rationale)

Shadow space exists because the ABI passes the first four integer/pointer arguments in registers: `RCX, RDX, R8, R9`. The ABI allows the callee to *spill* these register arguments into their home slots without needing to allocate its own stack space first. This makes prolog generation simpler and makes debugging/profiling conventions more uniform.

14.2.3 Who allocates and who frees

- The **caller allocates** shadow space (by decrementing `RSP` before `call`).
- The **caller frees** shadow space (by incrementing `RSP` after the call returns).

14.2.4 Shadow space and stack arguments

If a function has more than four integer/pointer parameters, the 5th and later are placed on the stack by the caller. Critically, those stack arguments sit **above** the 32-byte shadow space.

14.2.5 Minimal correct call sequence from assembly (Win64)

The following pattern is the simplest safe shape for calling another function in Win64 assembly:

```
.intel_syntax noprefix
.text
.extern some_callee
.globl call_some_callee
call_some_callee:
    /* Reserve 32 bytes shadow space + 8 bytes padding for alignment simplicity */
    sub rsp, 40

    /* Setup args: RCX, RDX, R8, R9 */
    mov rcx, 1
    mov rdx, 2
    mov r8,  3
    mov r9,  4
    call some_callee

    add rsp, 40
    ret
```

Why `sub rsp, 40`? The ABI requires 32 bytes shadow space, and practical call-site alignment constraints are easiest to satisfy by also reserving an extra 8 bytes. This keeps the call-site stack alignment correct under the common entry alignment state.

14.2.6 Leaf functions and shadow space

A **leaf function** (one that makes no calls) is not required to allocate shadow space for itself. However:

- If a function *does* call another function, it must reserve shadow space before that call.
- If you later modify a leaf function to make a call, the shadow-space requirement becomes mandatory.

14.3 Register Differences

14.3.1 Argument registers: Windows x64 vs System V AMD64

The most immediately dangerous difference between the Microsoft x64 ABI and the System V AMD64 ABI is the register assignment for parameters.

Windows x64 (Microsoft ABI)

- Integer / pointer args 1..4: `RCX, RDX, R8, R9`
- Floating args 1..4: `XMM0, XMM1, XMM2, XMM3`
- Additional args: stack (above shadow space)

System V AMD64 (typical Linux ABI)

- Integer / pointer args 1..6: `RDI, RSI, RDX, RCX, R8, R9`
- Floating args 1..8: `XMM0..XMM7`
- Additional args: stack

Immediate consequence. Code that assumes SysV and runs under Windows will read arguments from the wrong registers and will frequently crash or compute nonsense.

14.3.2 Caller-saved vs callee-saved (non-volatile) register sets

Windows x64 non-volatile registers

The callee must preserve (save/restore if used):

- GPRs: `RBX, RBP, RDI, RSI, R12, R13, R14, R15`
- Special: `RSP` must be restored to its original value at return
- XMM: `XMM6..XMM15` are non-volatile

Windows x64 volatile registers

Caller-saved (may be clobbered by a call):

- GPRs: `RAX, RCX, RDX, R8, R9, R10, R11`
- XMM: `XMM0..XMM5` are volatile

The critical difference: `RDI` and `RSI`

A classic interop pitfall:

- In **SysV**, `RDI` and `RSI` are used for arguments and are treated as volatile (caller-saved).
- In **Windows x64**, `RDI` and `RSI` are **non-volatile** (callee-saved).

If you port SysV-style assembly that freely clobbers `RDI`/`RSI` into Windows without saving them, you will corrupt the caller state.

14.3.3 Stack alignment expectations

Windows x64 requires disciplined stack alignment at call boundaries. In practice for assembly authors:

- ensure your call sites maintain the required alignment (commonly handled by reserving `40` bytes: 32 shadow + 8 padding),
- never return with a different `RSP` than you had at entry (except for well-formed prolog/epilog adjustments that net to zero),
- do not assume any **red zone** exists below `RSP` (Windows has none).

14.4 Interop Dangers

14.4.1 Danger 1: Shadow space vs red zone confusion

- SysV environments may provide a **red zone** (temporary scratch below `RSP` for leaf functions).
- Windows x64 provides **no red zone**.
- Windows x64 instead requires **shadow space** above `RSP` for callees.

Consequence. Any code that writes to `[rsp - N]` without first allocating space is non-conforming on Windows and may corrupt memory.

14.4.2 Danger 2: Wrong ABI name in Rust (`extern "C"` vs `extern "system"`)

On 64-bit Windows, `extern "C"` typically maps to the Win64 ABI. However, `extern "system"` is the correct default choice when binding Windows APIs because it tracks

Windows-specific ABI selections across architectures (especially relevant historically on 32-bit Windows, and still a best practice for correctness and clarity).

Rust: calling a Windows API safely

```rust
#![allow(non_snake_case)]
use core::ffi::c_void;

type HWND = *mut c_void;
type HINSTANCE = *mut c_void;
type LPCSTR = *const u8;
type UINT = u32;

extern "system" {
    fn MessageBoxA(hWnd: HWND, lpText: LPCSTR, lpCaption: LPCSTR, uType: UINT) -> i32;
}

fn main() {
    let text = b"Hello from Rust on Windows\0";
    let cap  = b"Win64 ABI\0";
    unsafe {
        MessageBoxA(core::ptr::null_mut(), text.as_ptr(), cap.as_ptr(), 0);
    }
}
```

What the ABI guarantees here. The call will use: `RCX`, `RDX`, `R8`, `R9` for the four pointer/integer arguments and will reserve the required call-frame space as generated by Rust.

14.4.3 Danger 3: Non-volatile XMM registers on Windows

On Windows x64, `XMM6..XMM15` are non-volatile. This is a major difference from many developers' intuition formed on SysV code where most XMM regs are treated as volatile.

If your hand-written assembly modifies `XMM6` without saving/restoring it, you can corrupt the caller state in subtle, data-dependent ways.

Correct preservation example (Windows x64)

```
.intel_syntax noprefix
.text
.globl preserve_xmm6_demo
preserve_xmm6_demo:
    /* Preserve RBX (non-volatile) and XMM6 (non-volatile) */
    push rbx

    /* Allocate 32 bytes shadow space + 16 bytes for saving XMM6 = 48 bytes total.
       Add 8 bytes padding to simplify alignment: 56 bytes (0x38). */
    sub rsp, 0x38

    /* Save XMM6 to stack (unaligned-safe store) */
    movdqu xmmword ptr [rsp + 0x20], xmm6  /* place after 32-byte shadow area */

    /* Body: clobber XMM6 intentionally */
    pxor xmm6, xmm6

    /* Restore XMM6 */
    movdqu xmm6, xmmword ptr [rsp + 0x20]

    add rsp, 0x38
    pop rbx
    ret
```

Notes.

- We placed the XMM save area at `[rsp+0x20]` to avoid overlapping the 32-byte shadow space.

- We used `movdqu` to avoid assuming 16-byte alignment for the save slot.

14.4.4 Danger 4: Stack alignment violations that crash far away

If you break stack alignment and then call into system libraries or compiled Rust/C code, failures may occur:

- inside unrelated functions,
- only under optimization,
- only when vector instructions are used,
- only on some machines.

This is why ABI compliance must be treated as a correctness constraint, not a performance hint.

14.4.5 Danger 5: Mixing object formats or unwinding expectations

On Windows, the ABI contract is entangled with:

- PE/COFF object conventions,
- structured exception handling and unwinding metadata requirements for non-trivial functions,
- toolchain-specific assembler directives when generating unwind info.

Practical rule for professional-grade assembly:

- keep hand-written assembly functions small and leaf when possible,
- when you must build non-leaf, ensure your toolchain produces correct unwind metadata or explicitly provide it in the expected form.

14.5 Extensive Windows Examples (Rust + GAS Intel Syntax)

14.5.1 Example A: Rust calling a Win64 ABI assembly function

This is the cleanest interop unit: Rust → assembly, no WinAPI, no special libraries.

Assembly (GAS Intel syntax), `add4_win64.S`

```
.intel_syntax noprefix
.text
.globl add4_win64
add4_win64:
    /* extern "C" fn(a: u64, b: u64, c: u64, d: u64) -> u64
       Win64: a,b,c,d in RCX,RDX,R8,R9; return in RAX */
    mov rax, rcx
    add rax, rdx
    add rax, r8
    add rax, r9
    ret
```

Rust caller (Windows)

```
extern "C" {
    fn add4_win64(a: u64, b: u64, c: u64, d: u64) -> u64;
}

fn main() {
    unsafe {
        let r = add4_win64(1, 2, 3, 4);
        println!("add4_win64 = {}", r);
    }
}
```

14.5.2 Example B: Assembly calls a C function (must provide shadow space)

This demonstrates why shadow space is a **caller obligation**.

C callee

```c
#include <stdint.h>

uint64_t mul_add(uint64_t a, uint64_t b, uint64_t c) {
    return a * b + c;
}
```

Assembly caller (Win64 correct)

```
.intel_syntax noprefix
.text
.extern mul_add
.globl asm_calls_c
asm_calls_c:
    /* extern "C" fn(x: u64, y: u64, z: u64) -> u64
       Win64: x in RCX, y in RDX, z in R8 */
    sub rsp, 40           /* 32 shadow + 8 padding for alignment */
    /* RCX,RDX,R8 already set by our own caller, forward directly */
    call mul_add
    add rsp, 40
    ret
```

Rust caller

```rust
extern "C" {
    fn asm_calls_c(x: u64, y: u64, z: u64) -> u64;
}

fn main() {
```

```
    unsafe {
        let r = asm_calls_c(6, 7, 1);
        println!("6*7+1 = {}", r);
    }
}
```

14.5.3 Example C: Exporting a stable ABI from Rust for other languages

If you ship a DLL, your exported surface should be intentionally ABI-stable.

Rust DLL export pattern (Windows)

```
use core::ffi::c_void;

#[repr(C)]
pub struct Handle {
    magic: u32,
    value: u64,
}

#[no_mangle]
pub extern "C" fn handle_create(init: u64) -> *mut c_void {
    let h = Box::new(Handle { magic: 0xAB1C0DEF, value: init });
    Box::into_raw(h) as *mut c_void
}

#[no_mangle]
pub extern "C" fn handle_destroy(p: *mut c_void) {
    if p.is_null() { return; }
    unsafe { let _ = Box::from_raw(p as *mut Handle); }
}

#[no_mangle]
```

```
pub extern "C" fn handle_add(p: *mut c_void, x: u64, out: *mut u64) -> i32 {
    if p.is_null() || out.is_null() { return -1; }
    unsafe {
        let h = &mut *(p as *mut Handle);
        if h.magic != 0xAB1C0DEF { return -2; }
        h.value = h.value.wrapping_add(x);
        *out = h.value;
    }
    0
}
```

Why this design is ABI-stable.

- The consumer never relies on the internal layout because it sees only `void*`.
- Only fixed-size primitives cross the boundary.
- The name is stable due to `#[no_mangle]`.

14.6 Practical Summary

14.6.1 Shadow space

- Caller reserves 32 bytes for every call.
- Extra stack arguments are placed above that area.
- Leaf functions do not need to allocate shadow space unless they make calls.

14.6.2 Register differences

- Windows x64 arg regs: `RCX, RDX, R8, R9` (GPR) and `XMM0..XMM3` (FP).

- Non-volatile regs include `RDI, RSI` and `XMM6..XMM15`, which is a major interop trap for SysV-trained assembly authors.

14.6.3 Interop dangers

- Confusing red zone (SysV) with shadow space (Win64).
- Forgetting to reserve shadow space before calling from assembly.
- Clobbering non-volatile registers (especially `RDI, RSI, XMM6..XMM15`).
- Breaking stack alignment and triggering crashes in unrelated library code.
- Ignoring unwind/metadata expectations for larger non-leaf assembly routines on Windows.

AArch64 Procedure Call Standard

15.1 Scope and Naming

The **AArch64 Procedure Call Standard** is commonly referred to as **AAPCS64**. It defines the binary-level contract for function calls on 64-bit Arm (AArch64): which registers carry arguments and return values, which registers must be preserved, how the stack is aligned, and how floating-point/SIMD values and aggregates are passed.

Important note about syntax. **GAS "Intel syntax" is an x86 feature.** AArch64 uses its own assembly syntax. In this chapter, all assembly examples are **GNU assembler (GAS) AArch64 syntax**.

Important note about "Rust on Windows".

- On **Windows on Arm64**, Rust targeting `aarch64-pc-windows-msvc` follows the **Windows ARM64 ABI** (Windows-specific).
- AAPCS64 is most commonly encountered on **Linux/ELF** targets such as `aarch64-unknown-linux-gnu`.
- You can still write the code on Windows and **cross-compile** to an AAPCS64 target for correctness testing and education.

15.2 Register Mapping

15.2.1 General-purpose registers and their roles

AArch64 provides 31 general-purpose registers `x0..x30` (64-bit), plus `SP`. Key roles defined by the procedure call standard:

- `x0..x7`: **argument registers** (integer/pointer and many aggregate pieces).
- `x0`: **return value register** for integer/pointer returns (primary).
- `x8`: **indirect result location register** (used when returning large/complex aggregates by hidden pointer).
- `x9..x15`: temporary registers (caller-saved).
- `x16, x17`: intra-procedure-call scratch (commonly used by linkers/PLT veneers; treat as caller-saved temporaries).
- `x18`: platform register (platform-specific usage; treat as reserved unless you control the whole platform ABI).
- `x19..x28`: **callee-saved** registers (must be preserved by the callee if used).
- `x29`: frame pointer (FP) when used.
- `x30`: link register (LR), holds return address on `bl`.
- `SP`: stack pointer, must obey alignment rules.

15.2.2 SIMD and floating-point registers

AArch64 has SIMD/FP registers `v0..v31` (128-bit each). Subregister views: `sN` for 32-bit float, `dN` for 64-bit double, `qN` for 128-bit vectors.

- `v0..v7`: **FP/SIMD argument registers** (for floats/vectors and FP-classified aggregate parts).
- `v0`: **FP return register** (primary).
- `v8..v15`: typically callee-saved in many AAPCS64 environments (treat as non-volatile for ABI-safe code; preserve if you use them in hand-written assembly intended to interoperate broadly).
- `v16..v31`: caller-saved temporaries.

15.2.3 Argument mapping summary tables

Integer/pointer arguments

Arg #	Register
1	`x0`
2	`x1`
3	`x2`
4	`x3`
5	`x4`
6	`x5`
7	`x6`
8	`x7`

Floating-point / SIMD arguments

FP Arg #	Register
1	`v0` (`s0`/`d0`)
2	`v1` (`s1`/`d1`)
3	`v2`
4	`v3`
5	`v4`
6	`v5`
7	`v6`
8	`v7`

15.2.4 Register preservation (what must survive a call)

For reliable interop, assume:

- **Callee-saved**: `x19..x28`, `x29` (if used), and `SP`.
- **Caller-saved**: `x0..x18`, `x30` (LR is overwritten by calls), and most `v` registers except the commonly non-volatile set.

15.3 Stack Alignment

15.3.1 The non-negotiable rule: 16-byte alignment

AAPCS64 requires that:

- `SP` is **16-byte aligned** at all times where the ABI requires it, and in particular at **public interfaces** and **call boundaries**.

- A conforming function must preserve this alignment when it calls another function.

15.3.2 Canonical prolog/epilog patterns

Leaf function (no stack frame)

A leaf that uses only argument registers and temporaries can be frameless:

```
.text
.global add_u64_aapcs64
add_u64_aapcs64:
    /* u64 add_u64_aapcs64(u64 a, u64 b)
       a in x0, b in x1, return in x0 */
    add x0, x0, x1
    ret
```

Non-leaf function with a frame (FP/LR saved)

A common AArch64 frame saves `x29` (FP) and `x30` (LR) together using paired stores:

```
.text
.global framed_demo
framed_demo:
    /* Prolog: allocate 16 bytes and save FP/LR */
    stp x29, x30, [sp, -16]!
    mov x29, sp

    /* body ... you may allocate more space with sub sp, sp, #N (N multiple of 16) */

    /* Epilog */
    ldp x29, x30, [sp], 16
    ret
```

15.3.3 Allocating local space correctly

Local allocation must preserve alignment. Always allocate in multiples of 16:

```
.text
.global locals_32bytes
locals_32bytes:
    stp x29, x30, [sp, -16]!
    mov x29, sp

    /* Allocate 32 bytes locals (multiple of 16) */
    sub sp, sp, 32

    /* ... locals live at [sp .. sp+31] or via [x29, #-offset] if you choose */

    add sp, sp, 32
    ldp x29, x30, [sp], 16
    ret
```

15.3.4 Why alignment matters (real failure modes)

If stack alignment is broken and your function calls into code using:

- vector loads/stores,
- ABI-assumed aligned spills,
- compiler-generated call sequences,

you can observe:

- unpredictable crashes,
- incorrect data movement,
- failures far from the actual bug site.

15.4 Float Handling

15.4.1 Scalar float and double passing/returns

- `float` parameters are passed in `s0..s7` (within `v0..v7`).
- `double` parameters are passed in `d0..d7`.
- `float` returns use `s0`; `double` returns use `d0`.

Example: `double f(double,double)`

```
.text
.global add_f64_aapcs64
add_f64_aapcs64:
    /* double add_f64_aapcs64(double a, double b)
       a in d0, b in d1, return in d0 */
    fadd d0, d0, d1
    ret
```

15.4.2 Mixed integer and float arguments

Integer and FP channels are independent. Example:

```
u64 g(u64 a, double b, u64 c, float d)
```

A typical mapping:

- a in `x0`
- b in `d0`
- c in `x1`
- d in `s1`

Assembly example (illustrative)

```
.text
.global mix_demo
mix_demo:
    /* u64 mix_demo(u64 a, double b, u64 c, float d)
       a in x0, c in x1, b in d0, d in s1
       returns u64 in x0 */

    /* Convert a and c to double, add with b, then add (double)d, then convert back */
    scvtf d2, x0          /* d2 = (double)a */
    scvtf d3, x1          /* d3 = (double)c */
    fadd  d0, d0, d2      /* b += a */
    fadd  d0, d0, d3      /* b += c */
    fcvt  d1, s1          /* d1 = (double)d */
    fadd  d0, d0, d1
    fcvtzu x0, d0         /* x0 = (u64)trunc(b) */
    ret
```

15.4.3 Homogeneous Floating-point Aggregates (HFA idea)

A key performance feature of AAPCS64 is that certain small aggregates made entirely of floating-point elements of the same type (often called **homogeneous floating-point aggregates**) can be passed in `v` registers rather than memory.
Consider:

```
typedef struct Vec3f { float x, y, z; } Vec3f;
```

A typical efficient mapping is:

- `x` in `s0`, `y` in `s1`, `z` in `s2` (all within `v0..v2`)
- return similarly uses `s0..` depending on classification

Example: sum components of `Vec3f`

```
.text
.global sum_vec3f
sum_vec3f:
    /* float sum_vec3f(Vec3f v)
       Typical HFA mapping: v.x in s0, v.y in s1, v.z in s2
       return float in s0 */

    fadd s0, s0, s1
    fadd s0, s0, s2
    ret
```

Engineering note. Exact aggregate classification depends on the standard's rules. The safe interop practice in mixed-language systems is:

- use `#[repr(C)]` on Rust structs,
- keep aggregates simple and homogeneous when you want register passing,
- otherwise assume "may pass by memory" and design ABI-safe interfaces accordingly.

15.5 Struct Passing and Returning (practical rules you can rely on)

15.5.1 Small structs: often split across registers

Many small aggregates are decomposed into 8-byte chunks and placed in `x` registers or `v` registers depending on content. A common and very practical case:

Two u64 values

```
struct PairU64 { u64 a; u64 b; };
```

Often:

- a in x0, b in x1

```
.text
.global sum_pair_u64
sum_pair_u64:
    /* u64 sum_pair_u64(PairU64 p)  with p split into x0/x1 */
    add x0, x0, x1
    ret
```

15.5.2 Larger structs: passed by memory

When an aggregate is too large or does not fit the register passing classification, it is passed in memory on the stack.

15.5.3 Returning large structs: hidden sret pointer in x8

For aggregates returned indirectly:

- caller allocates space for the return object,
- caller passes a hidden pointer to that space in x8,
- callee stores into [x8] and returns normally.

Illustrative assembly: return a 24-byte struct by sret

Assume:

```
struct Big { u64 a; u64 b; u64 c; };  /* 24 bytes */
Big make_big(u64 x, u64 y, u64 z);
```

One common shape:

- hidden return pointer in `x8`
- explicit args in `x0`, `x1`, `x2`

```
.text
.global make_big
make_big:
    /* Big make_big(u64 x, u64 y, u64 z)
       x in x0, y in x1, z in x2
       hidden sret pointer in x8 */

    str x0, [x8, 0]
    str x1, [x8, 8]
    str x2, [x8, 16]
    ret
```

15.6 Rust Examples Written on Windows (Cross-Targeting AAPCS64)

15.6.1 Rust declarations and `#[repr(C)]`

These are ABI-stable patterns for AArch64 targets:

```
#[repr(C)]
#[derive(Copy, Clone)]
pub struct PairU64 {
    pub a: u64,
```

```
    pub b: u64,
}

#[repr(C)]
#[derive(Copy, Clone)]
pub struct Vec3f {
    pub x: f32,
    pub y: f32,
    pub z: f32,
}

extern "C" {
    fn add_u64_aapcs64(a: u64, b: u64) -> u64;
    fn add_f64_aapcs64(a: f64, b: f64) -> f64;

    fn sum_pair_u64(p: PairU64) -> u64;
    fn sum_vec3f(v: Vec3f) -> f32;
}

fn main() {
    unsafe {
        let r = add_u64_aapcs64(10, 20);
        let d = add_f64_aapcs64(1.5, 2.25);

        let p = PairU64 { a: 7, b: 9 };
        let ps = sum_pair_u64(p);

        let v = Vec3f { x: 1.0, y: 2.0, z: 3.0 };
        let vs = sum_vec3f(v);

        println!("add_u64_aapcs64 = {}", r);
        println!("add_f64_aapcs64 = {}", d);
        println!("sum_pair_u64    = {}", ps);
```

```
        println!("sum_vec3f       = {}", vs);
    }
}
```

15.6.2 Build mindset (host vs target)

- The **host OS** (Windows) does not determine the ABI.
- The **target triple** determines the ABI (AAPCS64 on an AArch64 Linux target).
- For Windows Arm64 targets, you must follow the Windows ARM64 ABI instead of AAPCS64.

15.7 Summary

- **Register mapping:** integer/pointer args in `x0..x7`, FP/SIMD args in `v0..v7`; returns in `x0` or `v0`.
- **Preservation:** treat `x19..x28` (and FP/LR when used) as callee-saved; preserve non-volatile `v` registers when writing interop-grade assembly.
- **Stack alignment:** maintain `SP` 16-byte alignment, allocate locals in multiples of 16, and keep call-boundary alignment correct.
- **Float handling:** scalar floats/doubles in `s/d` views of `v` registers; homogeneous FP aggregates often stay in `v` registers for efficiency.
- **Struct passing/returning:** small structs often split across registers; large structs pass by memory; large returns commonly use a hidden sret pointer in `x8`.

RISC-V Calling Convention

16.1 Scope and Terminology

The standard RISC-V calling convention is specified by the RISC-V ELF psABI. It defines the register roles, which registers are preserved across calls, how return values are delivered, and the stack discipline (including the 16-byte alignment rule). It applies to both RV32 and RV64 ABIs (with specific type-size differences), and it has variants for floating-point (soft-float vs hard-float).

About assembly syntax. GAS "Intel syntax" is an x86 feature. RISC-V assembly uses the standard GNU assembler RISC-V syntax. All assembly examples in this chapter are **RISC-V GAS syntax**.

About Rust on Windows. You can write and build Rust on Windows while targeting RISC-V via cross-compilation (for example, `riscv64gc-unknown-linux-gnu` or `riscv64gc-unknown-none-elf`). The calling convention described here is determined by the **target** ABI (psABI), not by the Windows host.

16.2 a*, t*, s* Registers

16.2.1 Register naming overview

RISC-V general-purpose registers are `x0..x31`. The ABI provides conventional names:

- **Argument/return registers**: `a0..a7` (`x10..x17`)
- **Temporary (caller-saved) registers**: `t0..t6` (`x5..x7, x28..x31`)
- **Saved (callee-saved) registers**: `s0..s11` (`x8..x9, x18..x27`)

Other important conventional names:

- `zero` (`x0`): constant 0
- `ra` (`x1`): return address (written by `jal`/`jalr`)
- `sp` (`x2`): stack pointer
- `gp` (`x3`): global pointer (platform/runtime use)
- `tp` (`x4`): thread pointer (TLS/runtime use)
- `s0` may also be used as **frame pointer** (`fp`)

16.2.2 Caller-saved vs callee-saved: the real contract

The central interop rule is:

- **Caller-saved (volatile)**: the caller must assume these can be clobbered by a call.
  - `ra`, `a0..a7`, `t0..t6`

- **Callee-saved (non-volatile)**: if a callee uses these registers, it must restore them before returning.
  - `s0..s11` (and always restore `sp` to its entry value)

Practical consequence. If your assembly function modifies `s1` and does not restore it, you break the ABI and will corrupt the caller state in ways that may only appear under optimization or in unrelated code paths.

16.2.3 Floating-point register families (when hard-float ABI is used)

If the ABI includes hardware floating-point calling convention:

- **FP argument/return registers**: `fa0..fa7` (`f10..f17`)
- **FP temporaries**: `ft0..ft11`
- **FP saved**: `fs0..fs11`

If the ABI is **soft-float**, floating arguments and returns are passed using integer registers and/or memory as defined by the psABI rules for that ABI variant.

16.3 Return Rules

16.3.1 Integer and pointer returns

The standard rule:

- Return scalars up to one XLEN word in `a0`.
- Return scalars up to two XLEN words in `a0` and `a1`.

Here XLEN is 32 on RV32 and 64 on RV64.

16.3.2 Floating-point returns (hard-float ABIs)

If the hard-float calling convention is in effect:

- Single-precision and double-precision floating results are returned in `fa0`.
- Two-register FP returns (when applicable) use `fa0` and `fa1`.

16.3.3 Returning aggregates (struct/union)

A core ABI rule:

- Small aggregates that fit within two XLEN words are typically returned in `a0/a1`.
- Larger aggregates are returned **indirectly**: the caller allocates the result object and passes a hidden pointer to it as an implicit first parameter; the callee writes the result into that memory.

16.3.4 Example: returning a 128-bit value on RV64 (two-word return)

On RV64, a 128-bit quantity is two 64-bit words:

- low 64 bits in `a0`, high 64 bits in `a1` (common two-word scalar return pattern).

```
    .text
    .globl ret_u128_parts
ret_u128_parts:
    /* Returns (lo=a0, hi=a1). Demonstration only. */
    li a0, 0x1122334455667788
    li a1, 0x99aabbccddeeff00
    ret
```

16.4 Stack Discipline

16.4.1 Stack growth direction and alignment

In the standard RISC-V calling convention:

- The stack grows **downward** (toward lower addresses).
- The stack pointer `sp` is kept **16-byte aligned**.

16.4.2 What must be true at a call boundary

When your code calls another function:

- `sp` must be 16-byte aligned at the point of the call.
- Any stack arguments (beyond the register argument capacity) must be placed at the correct stack locations for the callee to load.
- If you want values in caller-saved registers after the call, **you must save them** before the call.

16.4.3 Frame pointer usage

Using `s0` as a frame pointer (`fp`) is optional. If used:

- `s0` becomes non-volatile by convention (it already is callee-saved), and
- typical prologs save `ra` and `s0` and then set `s0` to a stable reference point.

16.4.4 Canonical prolog/epilog patterns

Leaf function: no frame, no saves

```
    .text
    .globl add2_rv
add2_rv:
    /* u64 add2_rv(u64 a, u64 b)
       a in a0, b in a1, return in a0 */
    add a0, a0, a1
    ret
```

Non-leaf function: save ra and s0, allocate aligned frame

This is a robust interop-grade pattern:

```
    .text
    .globl framed_sum3_rv
framed_sum3_rv:
    /* u64 framed_sum3_rv(u64 a, u64 b, u64 c)
       a0=a, a1=b, a2=c; return a0 */

    /* Allocate 16 bytes (aligned) and save ra/s0 */
    addi sp, sp, -16
    sd   ra, 8(sp)
    sd   s0, 0(sp)
    addi s0, sp, 16          /* optional: establish a frame pointer base */

    add a0, a0, a1
    add a0, a0, a2

    /* Restore and return */
    ld   s0, 0(sp)
    ld   ra, 8(sp)
```

```
    addi sp, sp, 16
    ret
```

Why 16 bytes even if you save only two registers? Because the ABI requires 16-byte stack alignment. Frame sizes should be a multiple of 16 to preserve alignment across calls.

16.4.5 Passing more than 8 integer arguments (stack arguments)

RISC-V passes up to 8 integer/pointer arguments in `a0..a7`. Additional integer/pointer arguments go on the stack.

C-like signature

```
u64 f(u64 a0,u64 a1,u64 a2,u64 a3,u64 a4,u64 a5,u64 a6,u64 a7,u64 a8,u64 a9);
```

Mapping:

- `a0..a7` → registers `a0..a7`
- `a8, a9` → stack (placed by caller)

Assembly callee sketch (loads stack arguments)

Exact offsets depend on the frame you build. A common approach: load extra args from the incoming stack area relative to `sp` at entry, or relative to a frame pointer after prolog.

```
    .text
    .globl sum10_rv
sum10_rv:
    /* u64 sum10_rv(u64 p0..p9)
       p0..p7 in a0..a7, p8/p9 on stack */

    /* Build a small frame (aligned) */
```

```
addi sp, sp, -16
sd   ra, 8(sp)
sd   s0, 0(sp)
addi s0, sp, 16

/* The caller placed p8 and p9 in the argument stack area.
   With this 16-byte frame, their location relative to s0 is stable here:
   p8 at 0(s0), p9 at 8(s0) is a common teaching model when s0 points to the caller's sp.
   (Exact layout is ABI-defined by the caller's placement; this pattern assumes the
   ↪ standard convention.) */
ld   t0, 0(s0)       /* p8 */
ld   t1, 8(s0)       /* p9 */

add  a0, a0, a1
add  a0, a0, a2
add  a0, a0, a3
add  a0, a0, a4
add  a0, a0, a5
add  a0, a0, a6
add  a0, a0, a7
add  a0, a0, t0
add  a0, a0, t1

ld   s0, 0(sp)
ld   ra, 8(sp)
addi sp, sp, 16
ret
```

Engineering note. For production-quality hand-written assembly, you should verify the exact incoming stack argument layout for your specific ABI variant and toolchain expectations, especially when mixing ILP32/LP64 or when using a nonstandard prolog. The invariant you must keep is 16-byte alignment and correct save/restore discipline.

16.5 Floating-Point Handling (Soft-Float vs Hard-Float)

16.5.1 Hard-float register passing

When the hard-float ABI is used:

- FP arguments are passed in `fa0..fa7` when possible.
- FP return values are returned in `fa0` (and `fa1` when a two-register FP return is used).
- `fs0..fs11` are callee-saved FP registers (preserve if you use them).

Example: `double add(double,double)`

```
    .text
    .globl add_f64_rv
add_f64_rv:
    /* double add_f64_rv(double x, double y)
       x in fa0, y in fa1, return in fa0 */
    fadd.d fa0, fa0, fa1
    ret
```

Example: mixed integer + double

```
    .text
    .globl mix_i64_f64_rv
mix_i64_f64_rv:
    /* u64 mix_i64_f64_rv(u64 a, double b, u64 c)
       a in a0, b in fa0, c in a1
       return u64 in a0 */

    /* Convert a and c to double, add with b, then convert back */
    fcvt.d.l  ft0, a0        /* ft0 = (double)a */
```

```
    fcvt.d.l  ft1, a1        /* ft1 = (double)c */
    fadd.d    fa0, fa0, ft0  /* b += a */
    fadd.d    fa0, fa0, ft1  /* b += c */
    fcvt.l.d  a0, fa0        /* a0 = (long)b  (rounding mode applies) */
    ret
```

16.5.2 Soft-float behavior

In soft-float ABIs, floating arguments and returns are handled without using the FP calling convention. Practically, this means:

- floats may be passed in integer registers (`a*`) or via memory according to psABI rules,
- you must not assume `fa*` registers carry arguments or returns in soft-float mode,
- interop requires that all translation units agree on the ABI variant (soft vs hard float).

16.6 Extensive Rust Examples (Written on Windows, Targeting RISC-V)

16.6.1 Rust FFI declarations using the C ABI

RISC-V Rust targets use the platform C ABI for `extern "C"`.

```
#[cfg(target_pointer_width = "64")]
type Ux = u64;

#[cfg(target_pointer_width = "32")]
type Ux = u32;

extern "C" {
    fn add2_rv(a: Ux, b: Ux) -> Ux;
```

```
    fn framed_sum3_rv(a: Ux, b: Ux, c: Ux) -> Ux;
}

fn main() {
    unsafe {
        let r1 = add2_rv(10, 20);
        let r2 = framed_sum3_rv(1, 2, 3);
        println!("add2_rv = {}", r1);
        println!("framed_sum3_rv = {}", r2);
    }
}
```

16.6.2 Rust + embedded assembly via `global_asm!` (portable for teaching)

This keeps the assembly inside the Rust crate. It is useful when your Windows host environment is the primary editor/build driver.

```
use core::arch::global_asm;

global_asm!(r#"
    .text
    .globl add2_rv
add2_rv:
    add a0, a0, a1
    ret
"#);

extern "C" {
    fn add2_rv(a: u64, b: u64) -> u64;
}

fn main() {
    unsafe {
```

```
        let r = add2_rv(7, 5);
        println!("7 + 5 = {}", r);
    }
}
```

Cross-compilation mindset.

- Host: Windows (editor/build orchestration).
- Target: a RISC-V triple (determines the calling convention and codegen).
- The ABI rules in this chapter apply when the target uses the RISC-V psABI calling convention.

16.7 Interop-Safe Discipline Checklist (RISC-V)

For assembly intended to interoperate reliably with Rust/C across toolchains:

1. Treat `a*` and `t*` as caller-saved; save them in the caller if needed across calls.
2. Preserve `s0..s11` if you use them in a callee.
3. Preserve `fs0..fs11` if you use them and the hard-float ABI is in effect.
4. Keep `sp` 16-byte aligned; allocate frame sizes as multiples of 16.
5. Save/restore `ra` if your function is non-leaf (it calls other functions).
6. For large struct returns, expect an implicit pointer argument (sret) and write into that memory.
7. Ensure all linked objects agree on ABI variant (RV32/RV64, ILP32/LP64, soft-float vs hard-float).

16.8 Summary

- **a* registers** (`a0..a7`) carry up to 8 integer/pointer arguments; `a0/a1` also carry return values.
- **t* registers** are temporaries: volatile across calls.
- **s* registers** are saved: callee must restore them if used.
- Stack grows downward and **`sp` is kept 16-byte aligned**.
- Hard-float ABIs use `fa0..fa7` for FP args and `fa0` for FP returns; `fs0..fs11` are preserved if used.
- Large aggregates are returned indirectly via a hidden pointer provided by the caller.

Variadic Functions and Edge Cases

17.1 Why Variadics Are an ABI Stress-Test

Variadic functions (`...`) are where **language rules** (C default promotions, prototypes, `va_list`) and **ABI rules** (register allocation, stack layout, hidden state for FP arguments) collide.
For a non-variadic function, the compiler knows the full signature at compile time and can follow a deterministic register/stack mapping. For a variadic function, the callee must be able to **recover** the additional arguments at runtime using only ABI conventions and `va_list` machinery. That is why variadic calls contain special rules, extra hidden state, and corner cases that do not exist elsewhere.
This chapter focuses on three areas:

- **Hidden parameters and hidden state** that exist even when you do not see them in source.
- **Default promotions** (C language rules) and their ABI consequences.
- **ABI corner cases** that commonly break interop between Rust/C/assembly, and between SysV/Win64.

17.2 Hidden Parameters and Hidden State

17.2.1 Hidden parameters are not “mystery magic”

A hidden parameter is an extra argument the compiler passes implicitly to satisfy ABI or language semantics. It does not appear in the source signature but does appear at the machine interface.

17.2.2 Hidden return pointer (sret) for large aggregates

Many ABIs return large or complex structs **indirectly**:

- the caller allocates space for the return object,
- passes a hidden pointer to that space,
- the callee writes the result into that memory.

Even if you never write a variadic function, this is a major ABI edge case. In variadic contexts, it becomes more dangerous because:

- the position of arguments changes if an sret pointer is inserted,
- mismatched declarations across translation units can silently corrupt argument decoding.

Example (C): large struct return becomes hidden pointer

```c
#include <stdint.h>

typedef struct Big {
    uint64_t a, b, c, d; /* 32 bytes */
} Big;
```

```
Big make_big(uint64_t x, uint64_t y) {
    Big r = { x, y, x + y, x ^ y };
    return r;
}
```

At the binary boundary, many ABIs will transform this into something like:

```
void make_big(Big* out /* hidden */, uint64_t x, uint64_t y);
```

Interop danger. If one module believes `make_big` returns in registers and another module believes it returns via hidden pointer, the call will compile, link, and then produce garbage at runtime.

17.2.3 Hidden state in SysV AMD64 varargs: the `AL` rule

On SysV AMD64, a variadic call carries a critical hidden piece of state:

- the caller sets `AL` to the number of vector (XMM) registers used to pass arguments to the callee.

This is not a separate stack argument; it is an implicit register contract that allows the callee to implement `va_start`/`va_arg` correctly when floating-point values are present.

Illustrative SysV call sequence (Intel-syntax x86-64 GAS)

```
.intel_syntax noprefix
.text
.globl sysv_call_varargs_demo
sysv_call_varargs_demo:
    /* SysV AMD64 demonstration only:
       call printf("x=%d y=%f\n", 7, 1.25);

       SysV arg regs:
```

```
        RDI, RSI, RDX, RCX, R8, R9 for integer/pointer
        XMM0..XMM7 for FP
      Hidden varargs state:
        AL = number of XMM regs used for arguments
   */

   /* Align stack for SysV before call (typical: sub rsp, 8) */
   sub rsp, 8

   lea rdi, [rip + fmt]     /* fmt in RDI */
   mov esi, 7               /* int 7 in ESI (2nd arg) */
   movsd xmm0, qword ptr [rip + val] /* double 1.25 in XMM0 (3rd arg) */

   mov al, 1                /* one XMM argument register used */
   call printf

   add rsp, 8
   ret

.section .rodata
fmt: .asciz "x=%d y=%f\n"
val: .double 1.25
```

Important. This SysV rule is **not** the Windows x64 rule. Confusing these is one of the most common interop failures in cross-platform assembly.

17.2.4 Hidden state in Windows x64 varargs: FP duplication into integer regs

Windows x64 has a different strategy for variadic calls:

- the first four arguments are assigned to `RCX`, `RDX`, `R8`, `R9` (integer/pointer channel) and

`XMM0..XMM3` (FP channel),

- the caller reserves **32 bytes of shadow space** for every call,
- for **variadic** calls, floating-point arguments must be provided in a way that `va_arg` can recover reliably; a common ABI rule is that the caller **duplicates** floating values into the corresponding integer register slot as well.

Illustrative Win64 call sequence (Intel-syntax x86-64 GAS)

```
.intel_syntax noprefix
.text
.extern printf
.globl win64_call_varargs_demo
win64_call_varargs_demo:
    /* Win64 demonstration:
       call printf("x=%d y=%f\n", 7, 1.25);

       Win64 arg regs:
         RCX, RDX, R8, R9  (integer/pointer)
         XMM0..XMM3        (float/double)
       Shadow space:
         caller must reserve 32 bytes for every call
       Variadic nuance:
         FP values are typically also made available through the integer arg path
         so va_arg can fetch them uniformly.
    */

    sub rsp, 40               /* 32 shadow + 8 padding */

    lea rcx, [rip + fmt]      /* fmt */
    mov edx, 7                /* int 7 */
```

```
    /* Put double into XMM2? No: it's the 3rd argument, so XMM1 is the FP slot.
       (The first FP argument position corresponds to the argument index among parameters.)
       ↪ */
    movsd xmm1, qword ptr [rip + val]

    /* Duplicate the double bits into the integer register that corresponds to arg3 (R8).
       This is a key varargs rule on Win64. */
    mov r8, qword ptr [rip + val]

    call printf

    add rsp, 40
    ret

.section .rdata,"dr"
fmt: .asciz "x=%d y=%f\n"
val: .double 1.25
```

Why this matters. If you pass a floating vararg in XMM only (without the ABI-required duplication strategy), the callee's `va_arg` decoding may read the wrong location and produce nonsense.

17.2.5 Other common hidden parameters you must recognize

Even outside variadics, these frequently appear and interact badly with ABI boundaries:

- **C++ `this` pointer**: instance methods receive an implicit object pointer (register varies by ABI).
- **Closure/environment pointers**: languages with nested functions or closures often pass a hidden context pointer.

- **Thread-local access base**: some platforms reserve a register for TLS (e.g., `GS`/`FS`-based models on x86-64, `x18` on some AArch64 platforms).

17.3 Default Promotions

17.3.1 C default argument promotions (the law of variadics)

In C, for arguments that correspond to `...`, the compiler applies default promotions:

- `float` is promoted to `double`,
- integer promotions: types narrower than `int` are promoted to `int` (or `unsigned int` when necessary).

This is not optional. It is required so that the callee can retrieve values with `va_arg` based on a known set of promoted types.

17.3.2 Consequences for ABI and assembly

- If you pass `float` as a variadic argument, it will be passed as a `double` at the ABI boundary.
- If you pass `char`, `signed char`, `unsigned char`, `short`, or `unsigned short`, the actual value is passed as an `int` or `unsigned int`.

So when you implement a variadic function in assembly, you must decode the **promoted** types, not the source-level narrow types.

17.3.3 C example: promotions in practice

```
#include <stdarg.h>
```

```c
#include <stdio.h>

static void demo_promotions(const char* tag, ...) {
    va_list ap;
    va_start(ap, tag);

    /* If caller passes 'float', you must read it as double */
    double x = va_arg(ap, double);

    /* If caller passes 'char' or 'short', you must read as int */
    int y = va_arg(ap, int);

    printf("%s: x=%f y=%d\n", tag, x, y);

    va_end(ap);
}

int main(void) {
    float f = 1.25f;
    unsigned char c = 200;

    demo_promotions("case", f, c);
    return 0;
}
```

17.3.4 Rust perspective on default promotions

Rust does **not** have C-style variadic functions in stable Rust for user-defined functions (defining them is restricted), but Rust can **call** foreign C variadic functions through FFI declarations. The critical rule remains: when calling C variadics, the caller must obey C default promotions.

Rust on Windows: calling `printf` correctly

```
use core::ffi::c_char;

#[link(name = "msvcrt")]
extern "C" {
    fn printf(fmt: *const c_char, ...) -> i32;
}

fn main() {
    let fmt = b"x=%d y=%f\n\0";

    let x: i32 = 7;

    /* float must be promoted to double for C variadics */
    let y_f32: f32 = 1.25;
    let y_f64: f64 = y_f32 as f64;

    unsafe {
        printf(fmt.as_ptr() as *const c_char, x, y_f64);
    }
}
```

Common bug. Passing `f32` directly as a C variadic argument is incorrect; you must pass `f64`.

17.4 ABI Corner Cases

17.4.1 Corner case 1: SysV vs Win64 varargs are fundamentally different

Even when the source code is identical, the hidden state differs:

- SysV uses the `AL` register to communicate how many XMM registers were used for arguments.

- Win64 uses shadow space and varargs FP handling that is compatible with its `va_list` strategy (commonly requiring duplication of FP into integer argument slots).

Practical rule. Never reuse varargs assembly sequences between SysV and Win64 without rewriting the call setup.

17.4.2 Corner case 2: `va_list` is ABI-defined and not portable

`va_list` is not a universal structure across platforms:

- On some ABIs it is a simple pointer into a stack argument area.
- On others it is a structure tracking multiple sources (register save area + overflow stack area).

Therefore:

- you must not assume you can "hand-craft" `va_list` in portable C,
- you must not pass `va_list` objects across ABI boundaries unless the boundary guarantees the same ABI.

17.4.3 Corner case 3: passing structs through `...` is a trap

In C, passing a struct as a variadic argument is legal only if you pass it by value and retrieve it with the exact correct type. But in practice it is fragile because:

- struct passing rules are complex and ABI-dependent (register vs memory, classification),
- default promotions do **not** "normalize" structs the way they normalize scalar types,
- a mismatch in the callee's expected type produces silent memory/register misinterpretation.

Safer design

If you need to pass a compound object through `...`:

- pass a pointer to a `repr(C)` struct, or
- pass flattened scalars (e.g., three `double`s instead of a vector struct).

17.4.4 Corner case 4: `long double` is ABI-sensitive

The size and calling convention for `long double` varies across ecosystems:

- some environments treat it as 80-bit extended in 16 bytes,
- others treat it as 128-bit quad,
- some Windows toolchains treat `long double` the same as `double`.

Variadic decoding of `long double` is therefore especially dangerous across toolchains. If you control an ABI boundary, avoid `long double` in exported varargs APIs.

17.4.5 Corner case 5: alignment and "over-aligned" types

Some types require alignment greater than their size (or greater than default stack alignment for certain ABIs). If you pass such types in varargs, the ABI may impose padding and alignment rules that differ from non-variadic contexts.
If you are writing assembly:

- always maintain the ABI-required stack alignment at call boundaries,
- do not assume packed layouts for arguments on the stack,
- avoid passing over-aligned vector types through ....

17.4.6 Corner case 6: Windows x64 shadow space interactions

On Win64 every call site allocates 32 bytes shadow space. Consequences:

- "stack arguments" start **above** that home area,
- hand-written assembly that reads stack arguments must account for the home space and any local allocations,
- forgetting shadow space can corrupt adjacent stack data even if the call appears to work in small tests.

17.4.7 Corner case 7: mixed calling conventions in one process

On Windows you can accidentally mix:

- MSVC ABI expectations (`x86_64-pc-windows-msvc`),
- GNU/MinGW conventions for objects and libraries (`x86_64-pc-windows-gnu`),
- different CRTs (`msvcrt`, UCRT, static runtimes).

For variadic functions, mixing CRT boundaries is particularly risky because:

- formatting functions (`printf` family) are variadic,
- `va_list` handling is tightly coupled to the compiler+CRT ABI,
- misconfiguration can produce failures that look like "formatting bugs" but are actually ABI mismatches.

17.5 Extensive Worked Examples

17.5.1 Example A: Implementing a tiny variadic logger in C (portable structure, ABI-managed internals)

```c
#include <stdarg.h>
#include <stdio.h>

/* A minimal logger that forwards varargs to vprintf */
void logf(const char* tag, const char* fmt, ...) {
    va_list ap;
    va_start(ap, fmt);

    printf("[%s] ", tag);
    vprintf(fmt, ap);

    va_end(ap);
}

int main(void) {
    logf("INFO", "x=%d y=%f name=%s\n", 7, 1.25, "forge");
    return 0;
}
```

Why this is ABI-safe. The ABI-sensitive work is delegated to `vprintf`, and `va_list` is constructed by `va_start` as required by the platform.

17.5.2 Example B: Win64 assembly calling a C variadic function correctly

This is a condensed pattern showing the three mandatory pieces:

- shadow space allocation,

- correct register mapping,
- FP duplication into integer register slot for varargs.

```
.intel_syntax noprefix
.text
.extern printf
.globl win64_printf_x_y
win64_printf_x_y:
    /* void win64_printf_x_y(void);
       Calls: printf("x=%d y=%f\n", 7, 1.25) */

    sub rsp, 40                 /* shadow + padding */

    lea rcx, [rip + fmt]        /* arg1: fmt */
    mov edx, 7                  /* arg2: int */

    movsd xmm1, qword ptr [rip + val]  /* arg3: double in XMM1 */
    mov r8,   qword ptr [rip + val]    /* duplicate arg3 bits in R8 for varargs */

    call printf

    add rsp, 40
    ret

.section .rdata,"dr"
fmt: .asciz "x=%d y=%f\n"
val: .double 1.25
```

17.5.3 Example C: Rust on Windows calling `printf` with correct promotions

```
use core::ffi::c_char;
```

```
#[link(name = "msvcrt")]
extern "C" {
    fn printf(fmt: *const c_char, ...) -> i32;
}

fn main() {
    let fmt = b"x=%d y=%f c=%d\n\0";

    let x: i32 = 7;

    /* float promoted to double */
    let y_f32: f32 = 1.25;
    let y: f64 = y_f32 as f64;

    /* unsigned char promoted to int */
    let c_u8: u8 = 200;
    let c_promoted: i32 = c_u8 as i32;

    unsafe {
        printf(fmt.as_ptr() as *const c_char, x, y, c_promoted);
    }
}
```

17.6 Engineering Rules You Can Enforce in Your Toolchain

1. For cross-language ABIs, avoid exporting variadic functions. Prefer fixed signatures or a v-style API: `foo(fmt, va_list)` with controlled construction.

2. If you must expose varargs, expose them only inside a single well-defined ABI ecosystem (same compiler family and CRT).

3. In Win64 assembly, always allocate shadow space for any call and follow varargs FP

handling rules.

4. In SysV assembly, always set `AL` correctly for variadic calls that pass FP arguments.
5. In Rust, when calling C variadics, enforce C default promotions explicitly (notably `f32` → `f64`).
6. Avoid structs, `long double`, and over-aligned vector types in `...` across boundaries; pass pointers or flatten.

17.7 Summary

- Variadic functions require ABI-visible hidden state, not just "extra stack arguments".
- SysV AMD64 uses the hidden `AL` count of XMM arguments; Win64 relies on shadow space and a varargs strategy that makes FP recoverable by `va_arg`.
- C default promotions are mandatory and directly affect ABI-level values (`float`→`double`, narrow ints →`int`).
- The most common interop failures are caused by mixing SysV/Win64 assumptions, forgetting shadow space, failing FP varargs rules, and mismatching hidden parameters like sret.

Unwinding and Exception Boundaries (Foundational Only)

18.1 Stack Traces

18.1.1 What a “stack trace” really is

A stack trace is a **reconstruction of the dynamic call chain** that led to a point in execution. At the machine level, it is produced by repeatedly moving from the *current frame* to the *caller frame*, collecting:

- the **return address** for each call frame (the instruction pointer to resume in the caller),
- sometimes the **symbol / function name** corresponding to each address (if symbols are available),
- sometimes **source line info** (if debug information is present).

A stack trace is **not** inherently about exceptions. You can produce stack traces for:

- debugging crashes (access violations),
- profiling,

- telemetry,
- cooperative error reporting (Rust panic backtrace, C++ terminate handler, etc.).

18.1.2 Why unwinding is harder than “follow the frame pointer”

There is a classic idea: if every function uses a frame pointer (for example `RBP` on x86-64), then a debugger can walk a linked list of frames by reading saved `RBP` values.
Modern optimized code breaks that assumption:

- compilers frequently omit the frame pointer (FPO) and use `RSP`-relative addressing for locals,
- functions may have multiple entry/exit paths and mid-prolog/mid-epilog states,
- tail calls can remove frames entirely,
- dynamic stack allocations (e.g., `alloca`) change offsets at runtime,
- vector registers and non-volatile registers must be restored correctly for accurate recovery.

Therefore, accurate stack tracing in production relies on **metadata-driven unwinding** rather than guessing patterns from raw instructions.

18.1.3 Two high-level strategies

1. **Frame-pointer chaining** (simple, but costs a register and may reduce optimization).
2. **Metadata-based unwinding** (more complex, works with optimized code, required on some platforms).

Windows x64 strongly emphasizes metadata-based unwinding. ELF/SysV ecosystems commonly use DWARF Call Frame Information (CFI) metadata.

18.2 Metadata Overview

18.2.1 What unwind metadata must describe

Unwind metadata provides a **map** from an instruction address inside a function to rules that let the unwinder reconstruct the caller state:

- where the return address is,
- how to restore the stack pointer,
- how to restore preserved registers,
- where non-volatile SIMD registers were saved (if applicable),
- how to handle special regions (prolog/epilog, multiple epilogs).

18.2.2 Windows x64 model: table-driven unwinding

On Windows x64, **stack unwinding is a platform contract**, not just a debugging feature. It is used for:

- C++ exceptions,
- structured exception handling (SEH),
- asynchronous events and system-level stack walks,
- debugging and crash reporting.

The key design principle: **unwinding must succeed even in the middle of a prolog or epilog**. To achieve this, Windows relies on **function table entries** and compact **unwind codes** that describe how the prolog established the frame and where registers were saved.

Practical takeaway for assembly authors on Windows. If you write non-trivial x64 assembly (non-leaf, uses stack, saves non-volatiles, uses XMM6–XMM15, calls other functions), you must ensure your object contains correct unwind metadata in the form your toolchain expects. Otherwise, you may get:

- broken stack traces,
- crashes during exception unwinding,
- failures in debuggers/profilers,
- process termination when an exception crosses your frame.

18.2.3 ELF/SysV model: DWARF CFI and `.eh_frame`

On many non-Windows platforms, metadata for unwinding is carried using **DWARF Call Frame Information (CFI)**. It typically appears in `.eh_frame` (runtime unwinding) and/or `.debug_frame` (debugging). The same conceptual goal applies: restore the caller state reliably for any PC location inside a function.

Foundational-only boundary. This booklet section is not a full implementation guide for producing DWARF CFI or Windows unwind tables. Instead, it explains what they represent and how to design safe boundaries so you can keep your assembly and FFI surfaces robust.

18.3 Safe Boundaries

18.3.1 The number one rule: unwinding must match the declared ABI boundary

Unwinding across a boundary that does not permit unwinding is **undefined behavior** in many ecosystems. This is especially important when mixing:

- Rust panics (unwind vs abort),
- C++ exceptions,
- C code (which typically does not expect foreign exceptions),
- hand-written assembly (which may lack unwind metadata).

18.3.2 Rust on Windows: the safe default boundary

In Rust, panics can be configured as:

- **unwind** (panic unwinds the stack),
- **abort** (panic terminates the process immediately).

For **FFI exports** meant to be called from C/C++/other languages, the safest default is:

- **do not allow a Rust panic to cross an `extern "C"` boundary**.

A robust pattern is: **catch the panic inside Rust and translate it to an error code**.

Rust: safe FFI export that never unwinds past the boundary (Windows-ready)

```
use core::ffi::c_void;

#[no_mangle]
pub extern "C" fn forgevm_eval_safe(ctx: *mut c_void, code_ptr: *const u8, code_len: usize)
↪ -> i32 {
    /* Validate inputs first (never trust foreign callers) */
    if ctx.is_null() || code_ptr.is_null() {
        return -1;
    }

    /* Convert to slice safely */
```

```
    let code = unsafe { core::slice::from_raw_parts(code_ptr, code_len) };

    /* Never let a panic escape the FFI boundary */
    let result = std::panic::catch_unwind(|| {
        /* Your real work here. This example just checks UTF-8. */
        let _s = core::str::from_utf8(code).map_err(|_| -2)?;
        Ok::<(), i32>(())
    });

    match result {
        Ok(Ok(())) => 0,
        Ok(Err(e)) => e,      /* your domain error */
        Err(_) => -999,       /* panic occurred */
    }
}
```

Engineering consequence. Even if Rust is compiled with panic=unwind, this exported function behaves as an ABI-safe boundary: it will not unwind into foreign code.

18.3.3 When you intentionally want unwinding across FFI

Some advanced integrations intentionally allow unwinding across language boundaries (for example, integrating with C++ exception-aware frames or language runtimes). In Rust, that is done by using an ABI that explicitly permits unwinding, such as `extern "C-unwind"`.

Rust: an explicit unwind-permitting boundary

```
#[no_mangle]
pub unsafe extern "C-unwind" fn forgevm_may_unwind() {
    /* If compiled with panic=unwind, this may unwind into a compatible caller */
    panic!("demo panic");
}
```

Warning. This is only safe if the **entire call chain is unwind-capable and configured consistently**. If the caller (or any frame you cross) was built without unwind support (or expects different unwinding rules), behavior can be undefined or the process may abort.

18.3.4 C++ exceptions across boundaries on Windows

On Windows, C++ exceptions and SEH depend on correct unwind metadata for every frame that might be unwound through. Safe boundary rules for toolchain-quality code:

- Do not let C++ exceptions escape into C callers (translate to error codes at the boundary).
- Do not unwind across hand-written assembly unless you have correct unwind metadata and you understand the runtime rules.
- Keep exception ownership within one runtime domain (same compiler/runtime configuration across modules).

18.4 Stack Discipline and Unwinding Implications

18.4.1 What unwinders assume about well-formed frames

At a minimum, unwinders assume:

- the stack pointer was adjusted according to ABI rules,
- non-volatile registers were saved/restored properly if modified,
- the return address is recoverable (directly or via metadata),
- epilogs follow recognized patterns (Windows has strict rules),
- the metadata accurately matches the actual prolog/epilog behavior.

18.4.2 Windows x64: strict epilog/prolog patterns for reliable unwinding

Windows unwind metadata is compact partly because the platform constrains how prologs and epilogs are written:

- prolog operations must be describable by a limited set of unwind opcodes,
- epilogs must match patterns the unwinder can recognize (often by scanning instruction streams).

Design recommendation for hand-written assembly on Windows. Prefer:

- leaf functions (no stack usage, no calls) when possible,
- small frames with simple save/restore patterns,
- avoiding non-volatile XMM usage unless necessary,
- minimizing multiple epilog variants.

18.5 Extensive Examples (Rust for Windows OS, plus Intel-syntax x86-64 assembly)

18.5.1 Example A: leaf assembly function (typically unwind-trivial)

Leaf functions often do not need complex unwind metadata because they do not establish a frame and do not modify non-volatiles.

```
.intel_syntax noprefix
.text
.globl leaf_mul_add_win64
leaf_mul_add_win64:
```

```
    /* Win64 ABI leaf:
       u64 leaf_mul_add_win64(u64 a, u64 b, u64 c)
       a=RCX, b=RDX, c=R8, return=RAX */
    mov rax, rcx
    imul rax, rdx
    add rax, r8
    ret
```

Rust caller (Windows)

```
extern "C" {
    fn leaf_mul_add_win64(a: u64, b: u64, c: u64) -> u64;
}

fn main() {
    unsafe {
        let r = leaf_mul_add_win64(6, 7, 1);
        println!("6*7+1 = {}", r);
    }
}
```

18.5.2 Example B: non-leaf assembly that calls a function (unwind-sensitive)

Once you call out, you must follow Win64 stack rules (shadow space, alignment) and preserve non-volatiles. If an exception or stack walk occurs through this frame, correct unwind info becomes important.

```
.intel_syntax noprefix
.text
.extern puts
.globl nonleaf_calls_puts_win64
nonleaf_calls_puts_win64:
    /* Prolog: reserve 32 bytes shadow + 8 padding */
```

```
    sub rsp, 40

    lea rcx, [rip + msg]
    call puts

    add rsp, 40
    xor eax, eax
    ret

.section .rdata,"dr"
msg:
    .asciz "nonleaf call (Win64) - unwind metadata matters in real systems"
```

Foundational-only note. This example demonstrates correct calling convention discipline. In production Windows binaries, non-leaf assembly should also be accompanied by correct unwind metadata in the object format expected by your toolchain so that unwinding and stack walking are reliable through this function.

18.5.3 Example C: Rust boundary wrapper that prevents unwinding into foreign callers

This is the recommended safe boundary when exporting from Rust to other languages.

```
use core::ffi::c_void;

#[no_mangle]
pub extern "C" fn forgevm_do_work_no_unwind(_ctx: *mut c_void) -> i32 {
    let r = std::panic::catch_unwind(|| {
        /* Simulate a failure path */
        panic!("panic must not cross FFI");
    });
```

```
    match r {
        Ok(_) => 0,
        Err(_) => -1,
    }
}
```

18.5.4 Example D: “panic=abort” as an explicit boundary policy

If you choose process-abort on panic, you remove unwinding complexity across FFI at the cost of terminating on unexpected panics.

```
# Cargo.toml (conceptual policy snippet)
[profile.release]
panic = "abort"
```

Interpretation. With abort policy, a panic will not unwind through frames at all; it terminates. This can be a valid design choice for low-level components and toolchains where integrity is more important than recovery.

18.6 ABI Corner Cases at Unwind Boundaries

18.6.1 Corner case 1: crossing runtimes

Unwinding is not just an ABI rule; it is a **runtime protocol**. Mixing:

- Rust panic unwinding,
- C++ exceptions,
- SEH,

requires the entire chain to agree on unwind semantics. If one segment is not compatible, you get aborts or undefined behavior.

18.6.2 Corner case 2: missing or incorrect metadata in hand-written assembly

If your assembly establishes a frame, saves registers, and then an exception tries to unwind through it:

- incorrect metadata can restore the wrong registers,
- the unwinder may compute the wrong caller `RSP` and return address,
- the process may crash during unwind or produce corrupted state.

18.6.3 Corner case 3: non-volatile SIMD registers on Windows

On Win64, `XMM6..XMM15` are non-volatile. If your function touches them, you must preserve them. If you do not, unwinding is not the only risk: normal execution can be corrupted even without exceptions.

18.7 Safe Boundaries Checklist (Practical Rules)

1. At FFI boundaries, prefer **error codes** over cross-language exceptions/panics.
2. In Rust exports, use `catch_unwind` to prevent panics from escaping `extern "C"`.
3. Only use `extern "C-unwind"` when you fully control and validate unwind compatibility across the whole chain.
4. For Windows x64 assembly that is non-trivial (non-leaf, stack frame, preserved registers, calls), ensure your build emits correct unwind metadata.
5. Keep stack and register discipline perfect: shadow space, alignment, and non-volatile preservation are prerequisites for trustworthy stack traces and unwind behavior.

18.8 Summary

- Stack traces require reliable recovery of caller state; modern code relies on metadata rather than frame pointers.
- Windows x64 uses table-driven unwinding with strict prolog/epilog constraints; correct unwind metadata is essential for non-trivial assembly.
- Unwinding across boundaries is safe only when the boundary ABI explicitly permits it and all involved runtimes agree.
- For Rust on Windows, the safest export boundary is: **never let panics cross `extern "C"`**; catch and translate to error codes.

Part IV

Memory Model and Atomics

Why Memory Ordering Exists

19.1 Foundational Goal

Memory ordering rules exist because modern processors and optimizing compilers are allowed to **reorder operations** in ways that preserve single-threaded semantics but break naive multi-threaded assumptions.
This chapter is foundational and practical:

- Why reordering happens.
- What a data race actually means in real systems.
- Why "it works on my machine" is not a correctness argument.
- How Rust (and underlying hardware) expose ordering controls.

We focus on reality, not abstract proofs.

19.2 Reordering Sources

19.2.1 Reordering comes from three independent layers

When two threads share memory, the observed behavior depends on three independent agents:

1. The **compiler**.
2. The **CPU core**.
3. The **memory subsystem (caches + interconnect)**.

Each of these may reorder memory operations.

19.2.2 Compiler reordering

Compilers are allowed to:

- move loads earlier,
- delay stores,
- eliminate redundant loads,
- keep values in registers,
- merge stores,
- hoist code out of loops.

As long as single-threaded behavior is preserved, these transformations are legal.

Example: naive flag synchronization (broken)

```
int data = 0;
int ready = 0;

/* Thread A */
data = 42;
ready = 1;
```

```
/* Thread B */
while (ready == 0) { }
printf("%d\n", data);
```

In source order, it appears correct.
However, a compiler may reorder stores:

```
/* Optimized possibility */
ready = 1;
data = 42;
```

Now Thread B may observe:

- ready == 1
- data still old (0)

No hardware bug occurred. The compiler legally reordered instructions.

19.2.3 CPU reordering

Modern CPUs use:

- out-of-order execution,
- speculative execution,
- store buffers,
- load-store queues.

Even if the compiler emits instructions in program order, the CPU may execute them differently.

Store buffering reality

Consider:

```
Thread A:              Thread B:
x = 1;                 y = 1;
r1 = y;                r2 = x;
```

Both threads start with x = 0, y = 0.
It is possible on many architectures that:

- r1 == 0
- r2 == 0

Because each core:

- places its store into a store buffer,
- executes the load before the store becomes globally visible.

This is not theoretical. This is real hardware behavior.

19.2.4 x86 vs weaker architectures

- x86-64 has relatively strong ordering (TSO), but still allows store-to-load reordering via store buffers.
- ARM and RISC-V allow more aggressive reordering unless explicit barriers are used.

Therefore:

> If your code is correct only on x86 without explicit ordering, it is likely incorrect in principle.

19.2.5 Memory subsystem reordering

Even when a core commits operations in order:

- cache coherence protocols,
- write combining,
- inter-core communication latency

can delay visibility of stores to other cores.
Memory ordering rules exist to constrain when writes become visible.

19.3 Data Race Reality

19.3.1 What is a data race?

In modern language memory models (C, C++, Rust):
A data race occurs when:

- Two threads access the same memory location,
- At least one access is a write,
- There is no synchronization ordering between them.

In such a case:

> Behavior is undefined.

Undefined does not mean “sometimes wrong”. It means the compiler may assume it never happens and optimize accordingly.

19.3.2 The dangerous misconception

Many developers believe:

> "If the hardware eventually makes the value visible, it will work."

This is false because:

- Compilers assume no data race exists unless atomics or synchronization are used.
- Therefore, compilers may cache values in registers forever.
- Loads may be removed entirely.

19.3.3 Classic broken spin loop

```c
int ready = 0;

/* Thread A */
ready = 1;

/* Thread B */
while (ready == 0) { }
```

Without atomic semantics, Thread B may:

- Load ready once,
- Keep it in a register,
- Never reload from memory.

The loop may never terminate.

19.4 Why Memory Ordering Exists

Memory ordering exists to:

1. Constrain compiler reordering.
2. Constrain CPU reordering.
3. Establish visibility guarantees across threads.

It allows you to express:

- "This write must be visible before that flag is observed."
- "This read must observe prior writes from another thread."
- "This operation must not move across this boundary."

Without ordering, shared-memory concurrency is unreliable.

19.5 Acquire and Release (Foundational View)

The most important practical concept is the **Acquire-Release pair**.

19.5.1 Release: publishing data

A **release** store guarantees:

- All prior writes in this thread become visible before the release becomes visible.

19.5.2 Acquire: consuming data

An **acquire** load guarantees:

- No subsequent reads/writes move before the acquire.
- If it observes a release store, it also observes all writes that happened-before that release.

19.5.3 Correct publication pattern in Rust (Windows-compatible)

```rust
use std::sync::atomic::{AtomicBool, Ordering};
use std::thread;

static READY: AtomicBool = AtomicBool::new(false);
static mut DATA: u64 = 0;

fn producer() {
    unsafe {
        DATA = 42;
    }
    READY.store(true, Ordering::Release);
}

fn consumer() {
    while !READY.load(Ordering::Acquire) {
        std::hint::spin_loop();
    }
    let value = unsafe { DATA };
    println!("Observed value: {}", value);
}

fn main() {
    let t1 = thread::spawn(producer);
```

```
    let t2 = thread::spawn(consumer);

    t1.join().unwrap();
    t2.join().unwrap();
}
```

This establishes:

- DATA write happens-before READY.store(Release)
- READY.load(Acquire) synchronizes-with the store
- Therefore DATA == 42 is guaranteed visible

No theoretical reasoning needed — this is the real minimal safe pattern.

19.6 Assembly Perspective (x86-64 Windows)

19.6.1 x86-64 implicit ordering

On x86-64:

- Loads are not reordered with other loads.
- Stores are not reordered with other stores.
- But store→load reordering is allowed via store buffers.

Acquire loads are usually just normal loads. Release stores are usually just normal stores. However, **full fences** use:

```
mfence
```

Or locked instructions:

```
lock xchg qword ptr [rcx], rax
```

19.6.2 Why locked instructions matter

A locked instruction:

- Flushes store buffers,
- Enforces global visibility ordering,
- Acts as a full memory barrier.

19.7 Data Race Reality in Assembly

19.7.1 Broken lock implementation

```
.intel_syntax noprefix
.text
.globl naive_lock
naive_lock:
    /* rcx points to lock variable */
spin:
    mov rax, 1
    xchg rax, qword ptr [rcx]   /* missing LOCK prefix is fatal */
    test rax, rax
    jne spin
    ret
```

Without proper atomic semantics, two threads may both think they acquired the lock.
Correct version must use atomic instruction semantics:

```
.intel_syntax noprefix
.text
.globl atomic_lock
atomic_lock:
```

```
spin:
    mov rax, 1
    lock xchg qword ptr [rcx], rax
    test rax, rax
    jne spin
    ret
```

19.8 Practical Guidelines

1. Never share mutable data between threads without atomics or synchronization.
2. Do not rely on “x86 is strong” as a correctness argument.
3. Use Acquire/Release for one-way synchronization.
4. Use stronger ordering only when necessary.
5. Avoid mixing atomic and non-atomic access to the same location.
6. In Rust, prefer `Atomic*` types over raw pointers for shared state.

19.9 Foundational Summary

- Memory reordering comes from compiler, CPU, and memory system.
- Data races are undefined behavior at the language level.
- Memory ordering primitives exist to constrain reordering and establish visibility.
- Acquire-Release is the foundational synchronization pair.
- Assembly-level atomic instructions enforce ordering when used correctly.
- Correct concurrency requires discipline at every layer of the stack.

C11/C++20 Memory Model (Assembler Perspective)

20.1 Purpose of This Chapter

The C11 and C++20 memory models exist to provide one coherent contract between:

- **high-level code** (C/C++ source),
- **optimizing compilers** (reordering, caching, elimination),
- **real CPUs** (out-of-order execution, store buffers, weak ordering),
- **assembly-level reality** (instructions, fences, locked ops).

This chapter is foundational and practical. You will learn:

- what **happens-before** means in operational terms,
- the **atomic categories** you must recognize as an assembly author,
- how compilers typically map these semantics to x86-64 instructions (especially Windows),
- the patterns that keep FFI boundaries safe (Rust/C/C++ and assembly).

20.2 Happens-before

20.2.1 The one concept that replaces “guessing”

In concurrent programming, you do not want to rely on:

- source order,
- intuition,
- “it worked in testing”,
- “x86 is strong”.

You want one simple rule:

> If write A **happens-before** read B, then B must observe A (or some later write), and all effects before A become visible before B.

Everything else is a way of constructing happens-before edges.

20.2.2 Three building blocks (practical form)

1) Sequenced-before (single-thread order)

Inside one thread, the language defines a well-formed order of evaluation called **sequenced-before**. Compilers may reorder instructions, but not in a way that changes the observable single-thread behavior.

2) Synchronizes-with (cross-thread bridge)

A cross-thread bridge is created when:

- one thread performs a **release** operation on an atomic object,

- another thread performs an **acquire** operation on the same atomic object,
- and the acquire reads the value written (or a value in the release sequence).

This is the fundamental “publish/consume” link.

3) Happens-before (transitive closure)

Happens-before is the transitive closure of:

- sequenced-before edges, and
- synchronizes-with edges.

If:

```
A sequenced-before R   (in thread 1)
R synchronizes-with A' (release/acquire bridge)
A' sequenced-before B  (in thread 2)
```

Then:

```
A happens-before B
```

20.2.3 Operational interpretation for assembly authors

When a compiler emits code for atomics, it must ensure:

- the compiler itself does not reorder memory operations across the atomic boundary in a forbidden way,
- the CPU is forced (when needed) to respect visibility constraints using fences or locked instructions.

So happens-before is not only a high-level idea. It directly determines:

- where the compiler places barriers,
- which instructions become `lock`-prefixed,
- whether fences are inserted.

20.2.4 Worked pattern: publish data with release/acquire (portable mental model)

C11 version

```c
#include <stdatomic.h>
#include <stdint.h>

static _Atomic int ready = 0;
static uint64_t data_value = 0;

void producer(void) {
    data_value = 42;                        /* ordinary store */
    atomic_store_explicit(&ready, 1, memory_order_release);
}

uint64_t consumer(void) {
    while (atomic_load_explicit(&ready, memory_order_acquire) == 0) {
        /* spin */
    }
    return data_value;                      /* guaranteed to observe 42 */
}
```

What happens-before is achieved

- The store to `data_value` is sequenced-before the release-store to `ready`.

- The acquire-load from `ready` synchronizes-with that release-store (if it observes the value).
- Therefore the store to `data_value` happens-before the read of `data_value`.

Rust (Windows-friendly, same concept)

```rust
use std::sync::atomic::{AtomicBool, Ordering};
use std::thread;

static READY: AtomicBool = AtomicBool::new(false);
static mut DATA: u64 = 0;

fn producer() {
    unsafe { DATA = 42; }
    READY.store(true, Ordering::Release);
}

fn consumer() -> u64 {
    while !READY.load(Ordering::Acquire) {
        std::hint::spin_loop();
    }
    unsafe { DATA }
}

fn main() {
    let t1 = thread::spawn(producer);
    let t2 = thread::spawn(|| {
        let v = consumer();
        println!("DATA={}", v);
    });

    t1.join().unwrap();
```

```
    t2.join().unwrap();
}
```

Assembler takeaway.

The compiler must:

- Prevent moving `DATA = 42` after the release-store.
- Prevent moving the read of `DATA` before the acquire-load.
- Emit appropriate machine instructions so the CPU also respects the ordering.

20.3 Atomic Categories

From an assembler perspective, you must distinguish atomics by two axes:

- **operation type** (load/store/RMW/fence),
- **ordering strength** (relaxed, acquire, release, acq_rel, seq_cst).

20.3.1 Category 1: Atomic loads and stores

Relaxed

Relaxed atomics guarantee:

- atomicity (no torn reads/writes),
- a modification order for that single atomic object,

but **no ordering** with other memory operations.
Use cases:

- counters/statistics,
- reference counting components (with additional rules),
- flags where ordering is provided elsewhere.

Acquire load

Acquire loads prevent subsequent operations from moving above the load. If the acquire reads from a release store, it creates a happens-before edge.

Release store

Release stores prevent prior operations from moving below the store. When observed by an acquire, it publishes preceding writes.

Sequentially consistent load/store

Seq_Cst adds a global total order constraint across all seq_cst operations. It is the simplest to reason about but can be more expensive on weak memory systems.

20.3.2 Category 2: Read-Modify-Write (RMW) operations

RMW operations are atomic operations that:

- read a value,
- compute a new value,
- write it back,
- all as one atomic action.

Examples:

- atomic exchange,
- fetch-add,
- compare-exchange.

RMW is important because it typically maps to:

- **locked instructions** on x86-64,
- **LDXR/STXR loops** on AArch64,
- **LR/SC loops** on RISC-V.

C11 example: fetch-add

```c
#include <stdatomic.h>

static _Atomic int counter = 0;

int next_id(void) {
    return atomic_fetch_add_explicit(&counter, 1, memory_order_relaxed);
}
```

Assembler perspective on x86-64

A typical implementation uses a locked instruction:

```
.intel_syntax noprefix
.text
.globl atomic_fetch_add_i32
atomic_fetch_add_i32:
    /* Windows ABI example:
       rcx = pointer to i32 counter
       edx = addend
```

```
       returns previous value in eax */

    mov eax, edx
    lock xadd dword ptr [rcx], eax
    /* after xadd: eax holds old value */
    ret
```

Ordering note. On x86-64, `lock` instructions act as full barriers for ordering purposes, which is why they are used to implement stronger atomic operations.

20.3.3 Category 3: Compare-and-exchange (CAS)

CAS is the foundation of many lock-free algorithms.

C11 CAS

```
#include <stdatomic.h>
#include <stdbool.h>

static _Atomic int state = 0;

bool try_set_state(int expected, int desired) {
    return atomic_compare_exchange_strong_explicit(
        &state, &expected, desired,
        memory_order_acq_rel,
        memory_order_acquire
    );
}
```

x86-64 CAS mapping (Windows ABI assembly)

```
.intel_syntax noprefix
.text
```

```
.globl cas_i32
cas_i32:
    /* bool cas_i32(int* p, int expected, int desired)
       Win64:
         rcx = p
         edx = expected
         r8d = desired
       Returns:
         al = 1 on success, 0 on failure */

    mov eax, edx                    /* expected in eax for cmpxchg */
    lock cmpxchg dword ptr [rcx], r8d
    sete al                         /* al=1 if ZF=1 (success) */
    ret
```

Assembler takeaway. `cmpxchg` uses `EAX/RAX` as an implicit operand. This is a classic "hidden register" rule you must remember when mixing inline assembly, Rust `asm!`, and external assembly.

20.3.4 Category 4: Fences (barriers)

Fences enforce ordering constraints independent of a particular atomic object. In C/C++ you see:

- `atomic_thread_fence(order)`
- `atomic_signal_fence(order)` (compiler-only barrier, not hardware)

C11 fence

```
#include <stdatomic.h>

void full_barrier(void) {
    atomic_thread_fence(memory_order_seq_cst);
```

```
}
```

x86-64 mapping

A typical strong fence maps to `mfence` or a locked operation depending on the compiler strategy and order strength:

```
mfence
```

Foundational note. On x86-64, acquire/release often require no explicit fence instructions because the memory model is relatively strong, but the compiler barrier aspect still matters. On weaker architectures, fences or barrier instructions are used much more often.

20.4 Assembler-Centric Summary of Ordering Strengths

20.4.1 Relaxed

- Guarantees atomicity of the object itself.
- No ordering with other memory.
- Assembly: often the same instruction as non-atomic (but compiler must treat it specially).

20.4.2 Acquire

- Prevents later operations from moving before the load.
- Assembly: may be a plain load on x86-64; may require barrier semantics on weak architectures.

20.4.3 Release

- Prevents earlier operations from moving after the store.
- Assembly: may be a plain store on x86-64; may require barrier semantics on weak architectures.

20.4.4 Acq_Rel

- Used for RMW operations when you need both properties.
- Assembly: usually a locked op on x86-64; LR/SC with barriers on weaker ISAs.

20.4.5 Seq_Cst

- Adds a global total order across all seq_cst operations.
- Assembly: strongest constraints; may require fences on weak ISAs.

20.5 Practical FFI Safety (Rust on Windows)

20.5.1 Rule: do not mix atomic and non-atomic access

If one thread accesses `DATA` non-atomically while another thread accesses it atomically without synchronization, you have a data race.

Safe pattern: atomic pointer publication

```
use std::sync::atomic::{AtomicPtr, Ordering};

static PTR: AtomicPtr<u8> = AtomicPtr::new(core::ptr::null_mut());
```

```
fn publish(p: *mut u8) {
    PTR.store(p, Ordering::Release);
}

fn consume() -> *mut u8 {
    PTR.load(Ordering::Acquire)
}
```

20.5.2 Rule: prefer high-level sync unless you are building primitives

For most application-level code, use:

- mutexes,
- channels,
- atomics only for small flags/counters.

When you implement a primitive (spin lock, lock-free queue), verify:

- your operations create the intended happens-before edges,
- your assembly emission matches the ABI and ISA requirements.

20.6 Extensive Example: A Minimal Spin Lock (Assembler Perspective)

20.6.1 Rust spin lock using atomic exchange (Windows-ready)

```
use std::sync::atomic::{AtomicU32, Ordering};

pub struct SpinLock {
```

```
    state: AtomicU32, /* 0 = unlocked, 1 = locked */
}

impl SpinLock {
    pub const fn new() -> Self {
        Self { state: AtomicU32::new(0) }
    }

    pub fn lock(&self) {
        while self.state.swap(1, Ordering::Acquire) == 1 {
            while self.state.load(Ordering::Relaxed) == 1 {
                std::hint::spin_loop();
            }
        }
    }

    pub fn unlock(&self) {
        self.state.store(0, Ordering::Release);
    }
}

static LOCK: SpinLock = SpinLock::new();
static mut DATA: u64 = 0;

fn main() {
    LOCK.lock();
    unsafe { DATA = 123; }
    LOCK.unlock();
}
```

20.6.2 x86-64 implementation intuition

- swap maps to xchg (implicitly locked for memory operands on x86).

- acquire semantics are naturally satisfied by this locked behavior.
- release store is typically a normal store on x86-64 plus compiler constraints.

20.6.3 Win64 external assembly for swap primitive (educational)

```
.intel_syntax noprefix
.text
.globl atomic_swap_u32_win64
atomic_swap_u32_win64:
    /* u32 atomic_swap_u32_win64(u32* p, u32 v)
       Win64:
         rcx = p
         edx = v
       returns old value in eax */

    mov eax, edx
    xchg dword ptr [rcx], eax
    ret
```

Note. On x86, `xchg` with a memory operand is atomic and acts as a strong ordering point. This is why it is used for simple locks.

20.7 Summary

- Happens-before is the operational rule that defines cross-thread visibility.
- You build happens-before using sequenced-before (within a thread) plus synchronizes-with (release/acquire bridge).
- Atomic categories that matter to assembly:

  - loads/stores (relaxed/acquire/release/seq_cst),
  - RMW operations (xchg/xadd/cmpxchg or LR/SC loops),
  - fences (compiler-only vs hardware).

- On x86-64, many acquire/release operations are plain loads/stores at the instruction level, but the compiler must still enforce ordering.
- For Windows Rust interop, locked instructions (`lock` prefix, `xchg`) are the workhorses that implement strong atomic categories.

x86 Memory Ordering Model

21.1 Scope and Practical Goal

The x86-64 memory ordering model is often described as "strong" compared to many other architectures. In practice, this means:

- many reorders that are allowed on weaker architectures do **not** occur on x86,
- acquire/release synchronization is often achieved without explicit fence instructions,
- but the system is still **not** sequentially consistent by default.

This chapter stays foundational and assembler-oriented:

- what "strong ordering" really guarantees (and what it does not),
- what the `LOCK` prefix guarantees and why it matters,
- how Windows-friendly Rust/C/C++ atomics typically map to x86 instructions.

21.2 Strong Ordering (What x86 Actually Guarantees)

21.2.1 The mental model: TSO (Total Store Order)

For most practical concurrency work, x86 behaves like a **TSO-like model**:

- the core executes loads and stores in a disciplined way,
- stores become globally visible in a coherent order,
- each core has a store buffer that can delay visibility of its own stores to other cores.

Key point. x86 is strong because it forbids many reorderings, but store buffers mean one specific reordering still matters a lot.

21.2.2 What reorderings are forbidden (foundational form)

As a practical engineering summary:

- **Load → Load reordering:** generally not observed.
- **Store → Store reordering:** generally not observed.
- **Load → Store reordering:** generally not observed (the core preserves this ordering).

This is why many code patterns that fail on ARM/RISC-V appear to work on x86.

21.2.3 The one reordering you must not ignore: Store → Load

x86 allows **Store → Load** reordering in the following real hardware sense:

- a core may perform a later load before its earlier store becomes globally visible,
- because the store sits in a store buffer while the load can retire.

Classic demonstration: store buffering outcome

Two threads, initially `x=0`, `y=0`:

```
Thread A:              Thread B:
x = 1;                 y = 1;
r1 = y;                r2 = x;
```

On x86, it is possible to see:

```
r1 = 0 and r2 = 0
```

Because each thread's store can remain buffered when the load executes.

Assembler takeaway. If you need to prevent that outcome, you need a stronger ordering primitive (typically a locked operation or a fence).

21.2.4 Atomicity vs ordering

x86 provides strong **atomicity** for aligned naturally sized loads/stores:

- aligned 1, 2, 4, and 8-byte loads/stores are atomic (no tearing) on x86-64,
- but **atomicity alone does not provide ordering guarantees between distinct locations**.

So you can have:

- no tearing,
- but still observe reordering effects across different addresses.

21.2.5 Why compilers still matter on x86

Even if the CPU forbids most reorderings, the compiler can still:

- cache values in registers,
- move loads/stores across code boundaries,
- remove “useless” loads in spin loops,
- reorder independent memory operations.

So for correctness you still need language-level atomics or explicit compiler barriers.

21.3 The LOCK Prefix (The Hardware Ordering Workhorse)

21.3.1 What `LOCK` does in practical terms

The `LOCK` prefix turns certain read-modify-write memory instructions into:

- **atomic** operations with respect to other cores, and
- a **strong ordering point** (acts like a full barrier for memory ordering purposes in common engineering usage).

A locked operation ensures that:

- the read and write happen as one indivisible action at the memory-coherence level,
- the core cannot let surrounding memory operations float across the locked region in a way that breaks the ordering contract,
- store buffer effects are constrained so other cores observe a consistent ordering around the locked operation.

21.3.2 Which instructions commonly use `LOCK`

Typical locked primitives:

- `lock xchg` (note: `xchg` with a memory operand is implicitly locked even without the prefix),
- `lock xadd`,
- `lock add`, `lock inc`, `lock dec` (depending on usage),
- `lock cmpxchg`,
- `lock bts/btr/btc` for bit test and set operations.

21.3.3 Atomic exchange is the simplest lock primitive

A minimal spin-lock acquisition uses atomic exchange:

```
.intel_syntax noprefix
.text
.globl spin_lock_acquire
spin_lock_acquire:
    /* rcx = pointer to u32 lock variable (0 unlocked, 1 locked) */
.L_try:
    mov eax, 1
    xchg dword ptr [rcx], eax  /* atomic; implicit LOCK for mem operand */
    test eax, eax
    jne .L_try
    ret
```

Why this works. `xchg [mem], reg` is an atomic RMW. It forces exclusive ownership for the update and provides strong ordering so that:

- after you acquire the lock, later loads in the critical section do not float above the acquisition,
- before you release the lock, prior stores in the critical section are not delayed past the release store (compiler + hardware rules together).

21.3.4 Atomic compare-exchange is the foundation of lock-free algorithms

CAS uses `cmpxchg` with `LOCK`:

```
.intel_syntax noprefix
.text
.globl cas_u32_win64
cas_u32_win64:
    /* bool cas_u32_win64(u32* p, u32 expected, u32 desired)
       rcx=p, edx=expected, r8d=desired
       returns al=1 success, al=0 fail */

    mov eax, edx
    lock cmpxchg dword ptr [rcx], r8d
    sete al
    ret
```

Hidden operand rule. `cmpxchg` always uses `EAX/RAX` as the implicit "expected" register. If you forget that in hand-written assembly, the algorithm fails.

21.3.5 A locked add for atomic counters

```
.intel_syntax noprefix
.text
.globl atomic_fetch_add_u64
atomic_fetch_add_u64:
```

```
/* u64 atomic_fetch_add_u64(u64* p, u64 add)
   rcx=p, rdx=add
   returns old value in rax */

mov rax, rdx
lock xadd qword ptr [rcx], rax
ret
```

21.3.6 Where `mfence` fits (and why locked ops often replace it)

`mfence` is a full memory fence instruction:

```
mfence
```

In practice:

- many implementations prefer using locked RMW operations to provide both atomicity and ordering,
- explicit fences are used when you need ordering without modifying a particular memory location.

For foundational assembly design, the key is:

> If you need a strong global ordering point on x86, a locked instruction is the most common primitive; `mfence` is the explicit fence option.

21.4 Strong Ordering in Practice (Rust on Windows)

21.4.1 Why Rust orderings often compile to “plain” instructions on x86

On x86-64:

- acquire loads often compile to a plain `mov` load (plus compiler constraints),
- release stores often compile to a plain `mov` store (plus compiler constraints),
- seq_cst operations or RMW operations often use locked instructions.

This is why x86 is called "strong" in practice: it needs fewer explicit fences for common acquire/release patterns.

21.4.2 Worked example: release/acquire flag

```
use std::sync::atomic::{AtomicBool, Ordering};
use std::thread;

static READY: AtomicBool = AtomicBool::new(false);
static mut DATA: u64 = 0;

fn producer() {
    unsafe { DATA = 0x1122334455667788; }
    READY.store(true, Ordering::Release);
}

fn consumer() -> u64 {
    while !READY.load(Ordering::Acquire) {
        std::hint::spin_loop();
    }
    unsafe { DATA }
}

fn main() {
    let t1 = thread::spawn(producer);
    let t2 = thread::spawn(|| {
        let v = consumer();
```

```
        println!("DATA = 0x{:016x}", v);
    });

    t1.join().unwrap();
    t2.join().unwrap();
}
```

Assembler perspective. Even if the generated machine code uses plain `mov` for the acquire/release operations, the compiler is required to maintain the ordering constraints:

- it must not move the `DATA` store after the release store,
- it must not move the `DATA` load before the acquire load.

21.4.3 Spin lock example that relies on xchg

```
use std::sync::atomic::{AtomicU32, Ordering};

pub struct SpinLock {
    state: AtomicU32,
}

impl SpinLock {
    pub const fn new() -> Self {
        Self { state: AtomicU32::new(0) }
    }

    pub fn lock(&self) {
        while self.state.swap(1, Ordering::Acquire) == 1 {
            while self.state.load(Ordering::Relaxed) == 1 {
                std::hint::spin_loop();
            }
        }
```

```
    }

    pub fn unlock(&self) {
        self.state.store(0, Ordering::Release);
    }
}

static LOCK: SpinLock = SpinLock::new();
static mut SHARED: u64 = 0;

fn main() {
    LOCK.lock();
    unsafe { SHARED = 123; }
    LOCK.unlock();
}
```

Assembler mapping intuition. `swap` maps to `xchg [mem], reg` (implicitly locked) which provides a strong ordering point and atomicity in one instruction.

21.5 Common Pitfalls (Assembler View)

21.5.1 Pitfall 1: confusing atomicity with ordering

A plain aligned store is atomic, but it does not establish cross-location ordering guarantees. This is why "write data; set flag" must use release/acquire or a stronger primitive.

21.5.2 Pitfall 2: missing `LOCK` on RMW instructions

If you implement a lock or counter using RMW but forget `LOCK` (or use a non-atomic instruction sequence), it can fail under contention.

21.5.3 Pitfall 3: mixing atomic and non-atomic access to the same variable

Even on x86, mixing atomic and non-atomic accesses creates a data race in C/C++/Rust rules and enables compiler optimizations that break your program.

21.5.4 Pitfall 4: writing "portable" assembly without acknowledging ISA differences

x86 tends to hide ordering bugs that explode on ARM/RISC-V. If your algorithm requires ordering, express it in the language model (atomics) or use explicit barriers.

21.6 Summary

- x86-64 is "strong" because it forbids many reorderings, but store buffering still allows important store→load effects.
- The compiler is still a reordering source; atomics constrain compiler transformations.
- The `LOCK` prefix (and `xchg` with memory) provides atomic RMW and a strong ordering point used to implement many atomic categories.
- Common atomic building blocks on x86:
  - `xchg` for swap/lock acquire,
  - `lock xadd` for counters,
  - `lock cmpxchg` for CAS-based algorithms,
  - `mfence` for explicit global fences when needed.
- Rust on Windows typically maps acquire/release to plain loads/stores plus compiler ordering, while seq_cst and RMW rely on locked instructions.

ARM Acquire/Release Model

22.1 Practical Goal

AArch64 (64-bit ARM) is a **weakly ordered** architecture compared to x86. That does not mean it is "random"; it means that loads and stores may be observed in different orders by different cores unless you use:

- acquire/release memory operations, and
- explicit barriers (DMB/DSB) when stronger ordering is required.

This chapter stays foundational and assembler-oriented:

- what **LDAXR/STXR** are and why they exist,
- what **DMB/DSB** do at a practical level,
- how Rust atomic orderings map conceptually to these primitives,
- how to build safe low-level patterns without turning the chapter into theory.

About assembly syntax. AArch64 uses its own GAS syntax. "Intel syntax" is x86-only. All assembly examples here are AArch64 GAS syntax.

About Rust on Windows. You can write Rust on Windows and cross-compile to an AArch64 target (for example `aarch64-unknown-linux-gnu`) to study AArch64 atomics. Windows on Arm64 uses Windows ARM64 ABI, but the instruction-level atomic and barrier concepts remain the same.

22.2 Acquire/Release in ARM Terms (Foundational)

22.2.1 The engineering meaning

Acquire/release exists to let you express:

- **Release:** publish data, then publish a flag.
- **Acquire:** read the flag, then safely consume the published data.

On AArch64, acquire and release are often expressed directly at the instruction level using:

- acquire loads (for example `ldar` or `ldaxr`),
- release stores (for example `stlr` or `stlxr` in RMW loops).

22.2.2 Why AArch64 needs this more than x86

On ARM, the hardware may reorder:

- loads with later loads,
- stores with later stores,
- loads with later stores,

in more aggressive ways than x86. Acquire/release gives you a low-cost way to constrain that reordering only where you need it.

22.3 LDAXR/STXR

22.3.1 Exclusive access instructions in one sentence

Exclusive load/store instructions implement atomic read-modify-write sequences:

- `LDXR` / `STXR`: exclusive load and exclusive store (no implicit acquire/release).
- `LDAXR` / `STLXR`: exclusive load acquire and exclusive store release.

They work with an **exclusive monitor** mechanism:

- `LD*XR` marks an address as "exclusively observed" by the core.
- `ST*XR` succeeds only if no conflicting write has occurred since the exclusive load.
- The store returns a status in a register: 0 means success, nonzero means failure and you must retry.

22.3.2 Why the `A` and `L` variants matter

- `LDAXR` is an **acquire** operation: subsequent loads/stores in the core cannot move above it.
- `STLXR` is a **release** operation: prior loads/stores in the core cannot move below it.

This is the low-level foundation of:

- atomic swap,
- compare-and-swap (CAS),
- fetch-add,
- lock implementations,
- many lock-free algorithms.

22.3.3 Example 1: atomic increment with `LDAXR/STLXR`

This returns the old value and increments memory by 1.

```
.text
.global atomic_fetch_add_1_u64
atomic_fetch_add_1_u64:
    /* u64 atomic_fetch_add_1_u64(u64* p)
       x0 = p
       returns old value in x0 */

.L_retry:
    ldaxr x1, [x0]        /* x1 = old, acquire exclusive load */
    add   x2, x1, 1
    stlxr w3, x2, [x0]    /* attempt store-release; w3=0 success */
    cbnz  w3, .L_retry

    mov x0, x1
    ret
```

Foundational interpretation.

- The load is exclusive and acquire-ordered.
- The store is exclusive and release-ordered.
- The loop retries on contention.

22.3.4 Example 2: CAS (compare-and-swap) with `LDAXR/STLXR`

This function returns 1 on success, 0 on failure.

```
.text
.global cas_u64_ldaxr_stlxr
```

```
cas_u64_ldaxr_stlxr:
    /* int cas_u64_ldaxr_stlxr(u64* p, u64 expected, u64 desired)
       x0 = p
       x1 = expected
       x2 = desired
       returns w0 = 1 success, 0 fail */

.L_try:
    ldaxr x3, [x0]          /* x3 = *p (acquire) */
    cmp   x3, x1
    b.ne  .L_fail

    stlxr w4, x2, [x0]      /* store desired (release) */
    cbnz  w4, .L_try        /* retry if lost exclusivity */

    mov w0, 1
    ret

.L_fail:
    clrex                    /* clear exclusive state (polite) */
    mov w0, 0
    ret
```

Notes.

- `clrex` clears the exclusive monitor if you abandon the sequence.
- In tight loops, `yield` (or platform-specific pause hints) may be used for fairness, but this chapter remains foundational.

22.3.5 Example 3: atomic swap (exchange) with `LDAXR/STLXR`

```
.text
```

```
.global atomic_swap_u64
atomic_swap_u64:
    /* u64 atomic_swap_u64(u64* p, u64 v)
       x0 = p
       x1 = v
       returns old value in x0 */

.L_loop:
    ldaxr x2, [x0]
    stlxr w3, x1, [x0]
    cbnz  w3, .L_loop

    mov x0, x2
    ret
```

22.4 DMB/DSB Basics

22.4.1 Barrier instructions in practical terms

ARM provides barrier instructions that constrain ordering and completion in ways stronger than acquire/release.
Two of the most important:

- **DMB** (Data Memory Barrier): orders memory accesses (visibility ordering).
- **DSB** (Data Synchronization Barrier): orders and also waits for completion of effects (stronger).

A foundational way to remember them:

- **DMB:** "Do not let memory accesses cross this point in an illegal order."
- **DSB:** "Do not proceed until prior memory operations are complete to the required scope."

22.4.2 Barrier scopes and domains (keep it simple)

You will see barrier variants such as:

- `dmb ish` (inner shareable domain),
- `dsb ish` (inner shareable domain),
- `dmb sy` (full system).

For foundational user-mode multicore code, `ish` is a common default domain because it targets the typical coherent inner-shareable region.

22.4.3 When acquire/release is not enough

Acquire/release typically suffices for:

- publishing data with a flag,
- locks and lock-free algorithms built from atomic RMW.

You need stronger barriers when:

- interacting with device memory or memory-mapped I/O,
- requiring ordering across specific non-atomic sequences that cannot be expressed with acquire/release alone,
- building low-level runtime primitives (context switching, signal-like mechanisms),
- implementing sequences where completion (not just order) matters.

22.4.4 Example: explicit full barrier around a critical point

This example is illustrative for cases where you want a strong barrier:

```
.text
.global barrier_demo
barrier_demo:
    /* This is a conceptual pattern, not a recommended default for all code */
    dmb ish
    /* critical sequence of loads/stores */
    dmb ish
    ret
```

22.4.5 DSB example: stronger completion guarantee

```
.text
.global dsb_demo
dsb_demo:
    /* DSB is stronger: it ensures completion to the specified domain */
    dsb ish
    isb
    ret
```

About ISB. `ISB` (Instruction Synchronization Barrier) is used to synchronize instruction fetching/execution with changes in system state. It is often paired with `DSB` in system-level code. This chapter stays foundational, so treat this as awareness rather than a full system-programming guide.

22.5 Mapping to Language Atomics (Rust Perspective)

22.5.1 Acquire/Release in Rust

Rust provides ordering variants analogous to C/C++:

- `Ordering::Relaxed`
- `Ordering::Acquire`
- `Ordering::Release`
- `Ordering::AcqRel`
- `Ordering::SeqCst`

On AArch64 targets, these typically map to:

- `Acquire` load → `ldar` (or acquire semantics in an exclusive load for RMW),
- `Release` store → `stlr` (or release semantics in an exclusive store for RMW),
- RMW operations → `ldaxr/stlxr` loops (or similar),
- `SeqCst` → stronger barriers in addition to acquire/release semantics (implementation-dependent).

22.5.2 Rust example: publish/consume

```rust
use std::sync::atomic::{AtomicBool, Ordering};
use std::thread;

static READY: AtomicBool = AtomicBool::new(false);
static mut DATA: u64 = 0;
```

```
fn producer() {
    unsafe { DATA = 0xAABBCCDD11223344; }
    READY.store(true, Ordering::Release);
}

fn consumer() -> u64 {
    while !READY.load(Ordering::Acquire) {
        std::hint::spin_loop();
    }
    unsafe { DATA }
}

fn main() {
    let t1 = thread::spawn(producer);
    let t2 = thread::spawn(|| {
        let v = consumer();
        println!("DATA=0x{:016x}", v);
    });

    t1.join().unwrap();
    t2.join().unwrap();
}
```

Assembler takeaway. On AArch64, this pattern typically compiles to instructions with acquire/release semantics rather than relying on the stronger default ordering x86 provides.

22.6 Extensive Worked Example: A Spin Lock Using LDAXR/STLXR

22.6.1 AArch64 assembly lock acquire/release

Lock variable: `0` = unlocked, `1` = locked.

```
.text
.global spin_lock_acquire_a64
spin_lock_acquire_a64:
    /* void spin_lock_acquire_a64(u32* lock)
       x0 = lock pointer */

.L_acq:
    ldaxr w1, [x0]          /* acquire exclusive load */
    cbnz  w1, .L_acq        /* if already locked, keep spinning */

    mov   w2, 1
    stlxr w3, w2, [x0]      /* attempt store-release of 1 */
    cbnz  w3, .L_acq        /* retry on failure */
    ret

.text
.global spin_lock_release_a64
spin_lock_release_a64:
    /* void spin_lock_release_a64(u32* lock)
       x0 = lock pointer */

    /* Release store is enough to publish prior critical-section writes */
    stlr wzr, [x0]          /* store 0 with release semantics */
    ret
```

22.6.2 Rust wrapper interface (written on Windows, cross-targeting AArch64)

```
use core::arch::global_asm;

global_asm!(r#"
    .text
    .global spin_lock_acquire_a64
spin_lock_acquire_a64:
.L_acq:
    ldaxr w1, [x0]
    cbnz  w1, .L_acq
    mov   w2, 1
    stlxr w3, w2, [x0]
    cbnz  w3, .L_acq
    ret

    .text
    .global spin_lock_release_a64
spin_lock_release_a64:
    stlr wzr, [x0]
    ret
"#);

extern "C" {
    fn spin_lock_acquire_a64(p: *mut u32);
    fn spin_lock_release_a64(p: *mut u32);
}

static mut LOCK: u32 = 0;
static mut DATA: u64 = 0;

fn main() {
```

```
    unsafe {
        spin_lock_acquire_a64(&mut LOCK as *mut u32);
        DATA = 123;
        spin_lock_release_a64(&mut LOCK as *mut u32);
    }
}
```

Foundational note. This lock is a minimal teaching primitive. Production lock design also considers fairness, backoff, power, priority inversion, and integration with OS primitives.

22.7 Summary

- AArch64 is weakly ordered; acquire/release semantics matter for correctness.
- `LDAXR/STLXR` implement atomic RMW with acquire/release semantics and retry-on-contention behavior.
- `DMB` orders memory accesses; `DSB` orders and waits for completion (stronger).
- Rust (and C/C++) express these semantics via atomic orderings; compilers map them to `ldar/stlr` and `ldaxr/stlxr` loops plus barriers for stronger orders.
- For foundational correctness: use acquire/release for publish/consume and lock primitives; use explicit barriers when interacting with device memory or when completion guarantees are required.

RISC-V AMOs and FENCE

23.1 Practical Goal

RISC-V is designed to be clean and explicit about memory ordering. Unlike x86, you cannot rely on "strong default ordering". Instead, ordering is expressed using:

- **aq/rl bits** on atomic instructions (acquire/release semantics at the instruction),
- **LR/SC loops** for implementing read-modify-write sequences,
- explicit **FENCE** instructions for ordering between memory operations.

This chapter stays foundational and assembler-oriented:

- what **aq** and **rl** mean operationally,
- how LR/SC loops work and what can make them fail,
- when and why FENCE is needed,
- how Rust atomic orderings map conceptually to these primitives.

Assembly syntax note. RISC-V does not have "Intel syntax". All assembly examples here use standard RISC-V GAS syntax.

Rust on Windows note. You can write Rust on Windows and cross-compile to a RISC-V target. The ordering model described here applies to the RISC-V target ABI/ISA, not to the Windows host.

23.2 aq/rl Bits

23.2.1 What aq and rl are

RISC-V atomic instructions can carry two ordering qualifiers:

- **aq** (acquire)
- **rl** (release)

They are written as suffixes on atomic operations, for example:

- `amoswap.w.aq`
- `amoadd.d.rl`
- `amoand.w.aqrl`
- `lr.d.aq`
- `sc.d.rl`

23.2.2 Foundational operational meaning

Acquire (`aq`)

Acquire means:

- memory operations **after** the atomic in program order must not be observed to happen **before** it.

- in practice: later loads/stores cannot be reordered above an acquire atomic.

Release (`rl`)

Release means:

- memory operations **before** the atomic in program order must not be observed to happen **after** it.
- in practice: earlier loads/stores cannot be reordered below a release atomic.

Acquire+Release (`aqrl`)

`aqrl` provides both constraints:

- prior operations do not float below it (release),
- subsequent operations do not float above it (acquire).

Assembler perspective. This is the RISC-V direct equivalent of the AArch64 `LDAXR/STLXR` model: the semantics are encoded at the instruction, rather than always requiring separate fences.

23.2.3 What aq/rl do *not* guarantee

Foundational caution:

- aq/rl constrain ordering, but they do not "make all memory globally synchronized" in an unlimited way.
- they also do not replace atomicity rules: you still must use atomic instructions for shared locations that require atomic behavior.
- for some global total ordering requirements, you may need stronger mechanisms (for example patterns implemented by compilers for sequential consistency).

23.2.4 Example: publish/consume with aq/rl (flag publication)

This is the minimal pattern you want in low-level code:

- Producer: write data, then release-store the flag.
- Consumer: acquire-load the flag, then read data.

In RISC-V assembly, this is often written using AMOs or explicit fences plus normal stores/loads depending on exact constraints. A simple teaching version uses an AMO swap for the flag.

```
    .text
    .globl publish_flag_rv64
publish_flag_rv64:
    /* void publish_flag_rv64(u64* data, u32* flag)
       a0=data pointer, a1=flag pointer */

    li   t0, 42
    sd   t0, 0(a0)                /* write data first */

    li   t1, 1
    amoswap.w.rl x0, t1, 0(a1)    /* release publish flag */
    ret

    .text
    .globl consume_flag_rv64
consume_flag_rv64:
    /* u64 consume_flag_rv64(u64* data, u32* flag)
       a0=data pointer, a1=flag pointer
       returns data in a0 */

.L_spin:
    /* Acquire load of flag: use AMO as an acquire read */
    li   t1, 0
```

```
    amoswap.w.aq t0, t1, 0(a1)     /* t0 gets old flag, store 0 back (teaching only) */
    bnez t0, .L_got
    j    .L_spin

.L_got:
    ld   a0, 0(a0)
    ret
```

Teaching note. Using `amoswap` to read a flag is not always what you want in production because it writes as well. A production pattern often uses an acquire load mechanism or LR/SC with no net update, or uses fences with normal loads depending on the chosen ABI/library approach. The purpose here is to show the ordering semantics (`aq`/`rl`) clearly in pure assembly.

23.3 LR/SC Loops

23.3.1 Why LR/SC exists

RISC-V uses **Load-Reserved / Store-Conditional** (LR/SC) as the core primitive for implementing atomic read-modify-write when AMOs are not used or when you need CAS-like logic:

- `lr.w` / `lr.d`: load-reserved
- `sc.w` / `sc.d`: store-conditional (writes only if reservation still valid)

These can carry `aq`/`rl` bits:

- `lr.d.aq`
- `sc.d.rl`
- `lr.w.aqrl`

23.3.2 The reservation reality

An LR/SC sequence succeeds only if:

- no conflicting store (from another core or certain events) invalidated the reservation,
- the implementation-specific conditions remain valid.

Reasons `sc` may fail even without obvious contention:

- another core writes the same cache line,
- interrupts/context switches (implementation-dependent effects),
- the program takes too long between LR and SC,
- the core performs operations that break the reservation (implementation-dependent).

Therefore, LR/SC must always be coded as a **retry loop**.

23.3.3 Example 1: CAS with `lr.d.aq` / `sc.d.rl`

This is the foundational lock-free primitive.

```
    .text
    .globl cas_u64_lrsc
cas_u64_lrsc:
    /* int cas_u64_lrsc(u64* p, u64 expected, u64 desired)
       a0=p, a1=expected, a2=desired
       return a0=1 success, a0=0 fail */

.L_retry:
    lr.d.aq t0, 0(a0)         /* t0 = *p (acquire) */
    bne     t0, a1, .L_fail
```

```
    sc.d.rl t1, a2, 0(a0)     /* attempt store (release), t1=0 success */
    bnez    t1, .L_retry

    li      a0, 1
    ret

.L_fail:
    li      a0, 0
    ret
```

Foundational interpretation.

- The acquire on `lr` prevents later operations from moving before it.
- The release on `sc` prevents earlier operations from moving after it.
- Together they form an acquire/release RMW operation.

23.3.4 Example 2: fetch-add implemented with LR/SC (RV64)

```
    .text
    .globl fetch_add_u64_lrsc
fetch_add_u64_lrsc:
    /* u64 fetch_add_u64_lrsc(u64* p, u64 add)
       a0=p, a1=add
       returns old value in a0 */

.L_loop:
    lr.d.aq t0, 0(a0)         /* t0 = old */
    add     t1, t0, a1        /* t1 = new */
    sc.d.rl t2, t1, 0(a0)     /* try store, t2=0 success */
    bnez    t2, .L_loop
```

```
    mv      a0, t0
    ret
```

23.3.5 Example 3: spin lock using LR/SC

Lock variable: 0 = unlocked, 1 = locked.

```
    .text
    .globl lock_acquire_lrsc
lock_acquire_lrsc:
    /* void lock_acquire_lrsc(u32* lock)
       a0=lock pointer */

.L_spin:
    lr.w.aq t0, 0(a0)         /* acquire read lock */
    bnez    t0, .L_spin       /* if locked, keep spinning */

    li      t1, 1
    sc.w.rl t2, t1, 0(a0)     /* attempt to set to 1 with release */
    bnez    t2, .L_spin
    ret

    .text
    .globl lock_release_store
lock_release_store:
    /* void lock_release_store(u32* lock)
       a0=lock pointer */

    /* A release store can be implemented by using an atomic op with rl,
       or by a fence+store pattern depending on environment. Teaching version: */
    li      t0, 0
    amoswap.w.rl x0, t0, 0(a0)  /* release unlock */
    ret
```

Teaching note. Some environments implement unlock as a simple store plus an ordering primitive; others use AMO with `rl`. The key is: unlock must have **release semantics** so that critical-section stores become visible before another core sees the lock as free.

23.4 FENCE (Foundational)

23.4.1 What FENCE does

FENCE is an explicit barrier instruction that orders memory operations with respect to other memory operations.
At a foundational level:

- FENCE constrains reordering and visibility across a boundary in program order.
- It is used when aq/rl on an atomic instruction is not sufficient or when you need to order non-atomic sequences.

23.4.2 Common simple form

```
    fence rw, rw
```

This is a strong general fence ordering reads/writes before the fence with reads/writes after it.

23.4.3 When you typically need fences

- when you implement an atomic operation using plain loads/stores and need ordering,
- when you interact with device memory (MMIO) and must enforce ordering,
- when you require a strong barrier for correctness at a particular boundary.

23.4.4 Minimal illustrative publish/consume using fences + plain ops

This illustrates why fences exist even if you are not using AMOs.

```
    .text
    .globl publish_with_fence
publish_with_fence:
    /* a0=data pointer, a1=flag pointer */
    li  t0, 42
    sd  t0, 0(a0)
    fence w, w              /* ensure prior writes are ordered before flag store */
    li  t1, 1
    sw  t1, 0(a1)
    ret

    .text
    .globl consume_with_fence
consume_with_fence:
    /* a0=data pointer, a1=flag pointer */
.L_wait:
    lw  t0, 0(a1)
    beqz t0, .L_wait
    fence r, r              /* ensure subsequent reads see published data */
    ld  a0, 0(a0)
    ret
```

Foundational interpretation.

- The producer fence prevents the data store from being observed after the flag store.
- The consumer fence prevents the data load from being observed before the flag load.

23.5 Rust Atomic Orderings (Windows Host, RISC-V Target)

23.5.1 Rust code that relies on Acquire/Release

```
use std::sync::atomic::{AtomicBool, Ordering};
use std::thread;

static READY: AtomicBool = AtomicBool::new(false);
static mut DATA: u64 = 0;

fn producer() {
    unsafe { DATA = 42; }
    READY.store(true, Ordering::Release);
}

fn consumer() -> u64 {
    while !READY.load(Ordering::Acquire) {
        std::hint::spin_loop();
    }
    unsafe { DATA }
}

fn main() {
    let t1 = thread::spawn(producer);
    let t2 = thread::spawn(|| {
        let v = consumer();
        println!("DATA={}", v);
    });
    t1.join().unwrap();
    t2.join().unwrap();
}
```

Assembler perspective. On a RISC-V target, the compiler typically uses:

- `aq`/`rl` annotated atomics,
- or fences,
- or LR/SC loops,

to implement `Acquire/Release` semantics consistently with the architecture.

23.6 Practical Checklist

1. Use `aq` on loads when you need acquire semantics.
2. Use `rl` on stores when you need release semantics.
3. Use `aqrl` on RMW operations that must both acquire and release.
4. Always code LR/SC as a retry loop; `sc` can fail for many reasons.
5. Use `FENCE` when ordering must apply to plain (non-atomic) loads/stores or when a stronger barrier is needed.
6. Do not mix atomic and non-atomic accesses to the same location across threads.

23.7 Summary

- RISC-V expresses ordering explicitly: `aq` and `rl` bits encode acquire/release at the instruction.
- LR/SC loops are the fundamental building block for CAS, fetch-add, and many lock-free algorithms.

- `FENCE` orders memory operations across a boundary and is essential for ordering plain loads/stores and device interactions.
- Rust atomic orderings map to these mechanisms on RISC-V targets; the Windows host environment does not change the target memory model.

Correct Locking Patterns

24.1 Purpose and Boundary of This Chapter

This chapter explains locking patterns the way an assembly-oriented engineer must understand them:

- a lock is not just “atomicity”; it is **atomicity + ordering**,
- the ordering must hold across compilers and CPUs,
- the lock must be correct under contention, preemption, and optimization.

We keep it foundational and practical:

- build a minimal spinlock,
- understand retry loops and why failures happen,
- see what breaks when fences (or acquire/release semantics) are missing,
- implement safe patterns in Rust for Windows, and show the assembly intuition.

24.2 Minimal Spinlock

24.2.1 What a minimal lock must guarantee

A correct lock must provide these properties:

Mutual exclusion

Only one thread at a time may enter the critical section.

Visibility and ordering

When a thread releases the lock:

- all stores performed in the critical section must become visible to a thread that later acquires the lock,
- a thread that acquires the lock must not execute critical-section loads/stores before the lock acquisition in a way that breaks correctness.

This is the core reason memory ordering exists in lock design.

24.2.2 Minimal spinlock in Rust (Windows-ready)

This is a minimal teaching lock for short critical sections.

```
use std::sync::atomic::{AtomicU32, Ordering};

pub struct SpinLock {
    state: AtomicU32, /* 0 = unlocked, 1 = locked */
}

impl SpinLock {
```

```
    pub const fn new() -> Self {
        Self { state: AtomicU32::new(0) }
    }

    pub fn lock(&self) {
        /* Acquire semantics: critical-section reads/writes must not float above lock
        ↪  acquisition */
        while self.state.swap(1, Ordering::Acquire) == 1 {
            /* polite spinning: if it's locked, keep reading until it looks free */
            while self.state.load(Ordering::Relaxed) == 1 {
                std::hint::spin_loop();
            }
        }
    }

    pub fn unlock(&self) {
        /* Release semantics: critical-section writes must become visible before unlock
        ↪  becomes visible */
        self.state.store(0, Ordering::Release);
    }
}

static LOCK: SpinLock = SpinLock::new();
static mut SHARED: u64 = 0;

fn main() {
    LOCK.lock();
    unsafe { SHARED = 123; }
    LOCK.unlock();
}
```

Why this is correct (foundational).

- `swap(Acquire)` prevents subsequent operations from moving before acquisition.
- `store(Release)` prevents prior operations from moving after unlock.
- Together they establish the fundamental lock ordering: writes in the critical section become visible to the next acquirer.

24.2.3 Minimal x86-64 Win64 assembly intuition (LOCK / implicit locked xchg)

On x86, xchg with a memory operand is atomic and acts as a strong ordering point.

```
.intel_syntax noprefix
.text
.globl spin_lock_acquire_win64
spin_lock_acquire_win64:
    /* void spin_lock_acquire_win64(u32* lock)
       rcx = lock pointer */

.L_try:
    mov eax, 1
    xchg dword ptr [rcx], eax     /* atomic; implicit LOCK */
    test eax, eax
    jne .L_try
    ret

.globl spin_lock_release_win64
spin_lock_release_win64:
    /* void spin_lock_release_win64(u32* lock)
       rcx = lock pointer */

    mov dword ptr [rcx], 0        /* release store in high-level terms */
    ret
```

Assembler note. On x86-64, the ordering strength is often provided by the implicit locked behavior of `xchg` and by the architecture's ordering rules. However, correctness still relies on using atomic operations for the lock variable and on compiler-enforced constraints.

24.3 Retry Loops

24.3.1 Why retry loops exist

On many architectures, atomic read-modify-write is implemented using a **retry loop**:

- AArch64 uses `LDAXR/STLXR` (exclusive monitor).
- RISC-V uses `LR/SC` (reservation + conditional store).

In both cases, the store can fail, so the operation must retry.

24.3.2 Retry loop failure is normal

Retry failure does not mean a bug. It can occur when:

- another core modifies the location,
- the reservation/monitor is cleared by system events (implementation dependent),
- the core loses exclusivity due to coherence traffic,
- the loop is too long between load-exclusive and store-exclusive.

Therefore:

> Exclusive / conditional store failure is a routine control-flow path, not an exceptional case.

24.3.3 AArch64 minimal lock with retry loop (LDAXR/STLXR)

```
.text
.global spin_lock_acquire_a64
spin_lock_acquire_a64:
    /* void spin_lock_acquire_a64(u32* lock)
       x0 = lock pointer */

.L_spin:
    ldaxr w1, [x0]            /* acquire exclusive load */
    cbnz  w1, .L_spin         /* locked: spin */

    mov   w2, 1
    stlxr w3, w2, [x0]        /* attempt store-release of 1 */
    cbnz  w3, .L_spin         /* failed: retry */
    ret

.text
.global spin_lock_release_a64
spin_lock_release_a64:
    /* void spin_lock_release_a64(u32* lock)
       x0 = lock pointer */

    stlr wzr, [x0]            /* release store 0 */
    ret
```

24.3.4 RISC-V minimal lock with retry loop (LR/SC)

```
    .text
    .globl lock_acquire_lrsc
lock_acquire_lrsc:
    /* void lock_acquire_lrsc(u32* lock)
       a0 = lock pointer */
```

```
.L_retry:
    lr.w.aq t0, 0(a0)         /* acquire read */
    bnez    t0, .L_retry      /* locked: spin */

    li      t1, 1
    sc.w.rl t2, t1, 0(a0)     /* try store-release; t2=0 success */
    bnez    t2, .L_retry
    ret

    .text
    .globl lock_release_amo
lock_release_amo:
    /* void lock_release_amo(u32* lock)
       a0 = lock pointer */

    li      t0, 0
    amoswap.w.rl x0, t0, 0(a0) /* release unlock */
    ret
```

Assembler takeaway. In both AArch64 and RISC-V, the retry loop is part of the atomic operation semantics. Designing lock algorithms requires that you treat retry as the steady state under contention.

24.4 What Breaks Without Fences (or Acquire/Release)

24.4.1 The simplest misconception

Many developers believe:

> "Atomic test-and-set gives mutual exclusion, so the data inside is safe automatically."

Mutual exclusion alone is not enough. The lock must also provide:

- acquire semantics on lock acquisition,
- release semantics on unlock,

so that data written by one thread becomes visible to the next.

24.4.2 Broken lock: no acquire/release ordering (conceptual)

Suppose you implement:

- lock acquisition using an atomic RMW but treat it as **relaxed**,
- unlock using a plain store with no release semantics.

On weakly ordered CPUs, this allows reordering such as:

- critical-section writes being delayed past unlock,
- critical-section reads happening before lock acquisition in the execution window.

24.4.3 Concrete failure pattern: “publish under lock” is not published

Thread A:

- acquires lock,
- writes `DATA=42`,
- releases lock.

Thread B:

- acquires lock,

- reads DATA.

If unlock does not have release semantics, Thread B can acquire the lock and still see old DATA due to store buffering / reordering.
This is precisely what lock release must prevent.

24.4.4 Demonstration in Rust: how to break it on purpose

The following code compiles, but it is not a correct lock. It is a teaching example of what *not* to do.

```
use std::sync::atomic::{AtomicU32, Ordering};

pub struct BrokenLock {
    state: AtomicU32,
}

impl BrokenLock {
    pub const fn new() -> Self {
        Self { state: AtomicU32::new(0) }
    }

    pub fn lock(&self) {
        /* WRONG: relaxed does not provide acquire semantics */
        while self.state.swap(1, Ordering::Relaxed) == 1 {
            while self.state.load(Ordering::Relaxed) == 1 {
                std::hint::spin_loop();
            }
        }
    }

    pub fn unlock(&self) {
        /* WRONG: relaxed store is not a release */
```

```
        self.state.store(0, Ordering::Relaxed);
    }
}
```

Why it breaks. On weakly ordered architectures, this does not establish the required ordering edges between:

- writes in the critical section,
- visibility to the next lock holder.

Even on x86-64, relying on this is incorrect in principle because the language memory model requires ordering to be expressed, not assumed.

24.4.5 Fix: Acquire on lock, Release on unlock

```
use std::sync::atomic::{AtomicU32, Ordering};

pub struct SpinLock {
    state: AtomicU32,
}

impl SpinLock {
    pub const fn new() -> Self {
        Self { state: AtomicU32::new(0) }
    }

    pub fn lock(&self) {
        while self.state.swap(1, Ordering::Acquire) == 1 {
            while self.state.load(Ordering::Relaxed) == 1 {
                std::hint::spin_loop();
            }
        }
```

```
    }

    pub fn unlock(&self) {
        self.state.store(0, Ordering::Release);
    }
}
```

24.4.6 What fences look like at the assembly level (foundational intuition)

On x86-64:

- locked instructions provide strong ordering,
- explicit fence exists (`mfence`) but is often avoided unless needed.

On AArch64:

- acquire/release is encoded in instructions (`ldaxr/stlxr`, `ldar/stlr`),
- explicit barriers exist (`dmb ish`) when stronger ordering is needed.

On RISC-V:

- acquire/release is encoded with `aq/rl` bits,
- explicit `fence` orders memory operations when required.

24.4.7 Why fences matter even if you think you have a lock

If you implement the lock variable using atomics but access protected data using:

- non-atomic loads/stores without proper acquire/release in the lock operations,

then the compiler and CPU can move operations across the boundary in ways that violate the lock contract.
The lock must be a real boundary:

- **compiler boundary** (prevents reordering across),
- **hardware boundary** (prevents visibility/reordering across).

24.5 Extensive Worked Example: Correct Locking with a Protected Invariant

We now implement a small protected invariant:

- `A + B = 100` must always hold.

24.5.1 Rust demonstration with correct spinlock

```rust
use std::sync::atomic::{AtomicU32, Ordering};
use std::thread;

pub struct SpinLock {
    state: AtomicU32,
}

impl SpinLock {
    pub const fn new() -> Self {
        Self { state: AtomicU32::new(0) }
    }

    pub fn lock(&self) {
        while self.state.swap(1, Ordering::Acquire) == 1 {
            while self.state.load(Ordering::Relaxed) == 1 {
                std::hint::spin_loop();
            }
        }
    }
```

```
    pub fn unlock(&self) {
        self.state.store(0, Ordering::Release);
    }
}

static LOCK: SpinLock = SpinLock::new();
static mut A: u64 = 50;
static mut B: u64 = 50;

fn worker(iter: usize) {
    for _ in 0..iter {
        LOCK.lock();
        unsafe {
            A += 1;
            B -= 1;
            if A + B != 100 {
                /* If this prints, your locking is broken */
                println!("Invariant broken: A={} B={}", A, B);
                std::process::abort();
            }
        }
        LOCK.unlock();
    }
}

fn main() {
    let t1 = thread::spawn(|| worker(1_000_00));
    let t2 = thread::spawn(|| worker(1_000_00));
    t1.join().unwrap();
    t2.join().unwrap();

    unsafe { println!("OK: A={} B={}", A, B); }
```

```
}
```

Why this matters. The invariant check detects:

- failed mutual exclusion (two writers inside at once),
- failed ordering/visibility (one thread sees torn or stale state across what should be a lock boundary).

24.6 Practical Rules (Assembler-Level Discipline)

1. Lock acquisition must be an **acquire** operation.
2. Lock release must be a **release** operation.
3. If you use LR/SC or LDAXR/STLXR, implement as a retry loop and assume failure is normal.
4. Use polite spinning (`spin_loop` hint / pause-like hint) to reduce contention impact.
5. Keep critical sections short; spinlocks are not for blocking work.
6. Do not mix atomic and non-atomic access to the lock variable.
7. If you write assembly locks for Windows x64, obey shadow space and preserve non-volatile registers.

24.7 Summary

- A correct lock is atomicity plus ordering: acquire on lock, release on unlock.

- Retry loops are fundamental for exclusive/conditional-store based atomics (AArch64 and RISC-V).
- Missing fences (or missing acquire/release) breaks the lock contract on weak memory systems and violates the language memory model even on x86.
- Rust on Windows can express correct locking using `AtomicU32` with `Acquire/Release`, and assembly implementations map to locked ops on x86 or retry loops + barriers on weak ISAs.

Part V

Common Engineering Failures

Stack Corruption

25.1 What Stack Corruption Really Means

Stack corruption means that the memory region used for call frames has been overwritten or reconstructed incorrectly, so that one or more of the following are no longer trustworthy:

- return address (the address popped into the instruction pointer on return),
- saved non-volatile registers (frame-preserved registers),
- stack pointer value and alignment,
- local variables and spilled temporaries,
- exception/unwind metadata assumptions about prolog/epilog behavior.

From an assembly perspective, stack corruption is not a single bug type. It is a **failure class** caused by breaking ABI invariants.

25.2 Typical Patterns

25.2.1 Pattern 1: wrong stack pointer adjustment (frame size bug)

How it happens

- You allocate N bytes but deallocate a different N.
- You forget that Win64 requires 32 bytes shadow space for every call.
- You allocate locals but forget alignment padding.

x86-64 Win64 example: mismatched `sub rsp` / `add rsp`

```
.intel_syntax noprefix
.text
.globl bad_frame_mismatch
bad_frame_mismatch:
    /* BAD: allocates 40, deallocates 32 -> stack pointer drifts */
    sub rsp, 40
    /* ... */
    add rsp, 32
    ret
```

Correct version

```
.intel_syntax noprefix
.text
.globl good_frame_match
good_frame_match:
    sub rsp, 40
    /* ... */
    add rsp, 40
    ret
```

Reality. A mismatched adjustment might appear to work in small tests but will eventually:

- crash on return,
- smash a caller's locals,
- break unwinding and stack tracing.

25.2.2 Pattern 2: violating stack alignment rules

Why it matters

Many ABIs require stack alignment at call boundaries. Violating it can break:

- code that uses vector instructions with alignment assumptions,
- ABI-mandated spill slots,
- runtime helpers and prologs inserted by compilers.

x86-64 SysV: 16-byte alignment before `call`

A classic failure is forgetting the alignment correction in a leaf-to-call conversion.

```
.intel_syntax noprefix
.text
.extern puts
.globl bad_sysv_alignment
bad_sysv_alignment:
    /* BAD on SysV if rsp isn't aligned before call */
    lea rdi, [rip !+! msg]
    call puts
    ret
.section .rodata
msg: .asciz "hello"
```

Fix strategy. Make sure `rsp` satisfies the ABI constraint at the call point (often by subtracting 8 when entering a function on SysV x86-64, depending on entry alignment and saved return address).

25.2.3 Pattern 3: writing past a local buffer (classic overflow)

C example that corrupts the stack

```c
#include <string.h>
#include <stdio.h>

void overflow_demo(const char* s) {
    char buf[8];
    /* BAD: no bounds checking */
    strcpy(buf, s);
    printf("buf=%s\n", buf);
}
```

Why it becomes stack corruption. `buf` is stored in the stack frame. Writing beyond it overwrites:

- saved registers,
- saved frame pointer,
- return address,
- stack cookies/guards (if enabled),
- adjacent locals.

25.2.4 Pattern 4: calling convention mismatch (SysV vs Win64, or wrong prototype)

How it happens

- You declare a function with the wrong calling convention in C/C++.
- You link objects built for different ABIs (Windows MSVC vs MinGW-gnu).
- You hand-write assembly expecting SysV registers but call from Win64 code.

Symptom

The callee reads the wrong registers as arguments and may:

- interpret a pointer as an integer,
- treat an integer as an address and crash,
- store through a garbage pointer and corrupt the caller stack.

Concrete x86-64 example: argument register mismatch

- SysV first argument: `RDI`
- Win64 first argument: `RCX`

If you compile a Win64 caller but write a SysV callee, the callee will read garbage from `RDI`, possibly corrupting memory.

25.2.5 Pattern 5: Win64 shadow space violation

The rule

On Win64, the caller must reserve **32 bytes shadow space** for every call. Many callees assume it exists.

BAD: calling without shadow space

```
.intel_syntax noprefix
.text
.extern printf
.globl bad_win64_no_shadow
bad_win64_no_shadow:
    /* BAD: no shadow space, call may smash stack */
    lea rcx, [rip !+! fmt]
    call printf
    ret
.section .rdata,"dr"
fmt: .asciz "hello\n"
```

Correct call frame

```
.intel_syntax noprefix
.text
.extern printf
.globl good_win64_shadow
good_win64_shadow:
    sub rsp, 40              /* 32 shadow + 8 padding */
    lea rcx, [rip !+! fmt]
    call printf
    add rsp, 40
    ret
```

```
.section .rdata,"dr"
fmt: .asciz "hello\n"
```

25.2.6 Pattern 6: variadic calling rule mistakes

Variadic calls add hidden rules (SysV uses `AL` for XMM count; Win64 has varargs FP handling and shadow space). Mistakes often manifest as:

- incorrect stack unwinds after the call,
- overwriting home space,
- corrupted return address due to wrong stack adjustment,
- reading wrong arguments leading to out-of-bounds writes.

25.2.7 Pattern 7: clobbering non-volatile registers

If you modify a callee-saved register and do not restore it, you corrupt the caller state.

Win64: examples of non-volatile registers

- `RBX, RBP, RSI, RDI, R12–R15` must be preserved.
- `XMM6–XMM15` must be preserved.

BAD: clobbering `RBX`

```
.intel_syntax noprefix
.text
.globl bad_clobber_rbx
bad_clobber_rbx:
    /* BAD: RBX is non-volatile on Win64 */
```

```
    mov rbx, 123
    ret
```

Correct: save/restore

```
.intel_syntax noprefix
.text
.globl good_preserve_rbx
good_preserve_rbx:
    push rbx
    mov  rbx, 123
    pop  rbx
    ret
```

25.2.8 Pattern 8: use-after-scope with stack addresses (escaping pointers)

C example

```c
int* bad_return_stack_ptr(void) {
    int x = 7;
    return &x; /* BAD: returning pointer to stack local */
}
```

The caller later writes through this pointer and corrupts arbitrary stack frames.

Rust analogue (prevented in safe Rust, possible with unsafe)

Safe Rust prevents returning references to stack locals that outlive the function. But with raw pointers and `unsafe`, you can still create the same failure.

```rust
fn bad_stack_ptr() -> *const i32 {
    let x: i32 = 7;
    &x as *const i32 /* BAD: pointer becomes dangling after return */
```

```
}

fn main() {
    let p = bad_stack_ptr();
    unsafe {
        /* Reading may appear to work; using it later is undefined behavior */
        println!("{}", *p);
    }
}
```

25.3 Diagnosis Workflow

This workflow is designed for real engineering practice. It assumes you are working with x86-64 Windows frequently (as your environment indicates), but it generalizes.

25.3.1 Step 1: classify the failure signature

Look at the symptom:

- crash on `ret` or near return address,
- crash in a function epilog,
- crash inside an unrelated function after a call,
- stack trace is garbage or stops at your function,
- exception unwinding fails or terminates the process.

These are strong indicators of stack corruption rather than a pure logic bug.

25.3.2 Step 2: verify ABI invariants for the failing function

For the function where corruption likely begins, verify:

1. Stack pointer adjustment: entry allocation equals exit deallocation.
2. Alignment rules: stack aligned at each call boundary.
3. Win64: shadow space reserved for every call.
4. Non-volatile registers preserved (GPRs and XMMs as required).
5. No writes beyond allocated local stack space.

25.3.3 Step 3: locate the first frame that looks wrong

Stack corruption often appears **after** the first bad write. The key is to find:

- the last frame with a valid return address sequence,
- the next frame where return address is nonsense.

That boundary is where corruption likely started.

25.3.4 Step 4: inspect stack memory around `RSP` and saved return address

In x86-64:

- return address is stored at `[RSP]` at function entry (after the call),
- in a framed function, saved non-volatiles are near the bottom of the frame.

Inspect memory near:

- the saved return address location,

- the saved frame pointer (if used),
- saved registers.

If these values look overwritten by pattern bytes (for example repeating ASCII), you likely have a buffer overflow.

25.3.5 Step 5: check for call boundary mistakes

If the crash occurs immediately after calling a function, verify:

- correct calling convention and prototype,
- shadow space for Win64 calls,
- stack alignment at the call site,
- varargs rules (if calling `printf`-like functions).

Many stack corruptions begin at call sites, not inside the callee.

25.3.6 Step 6: isolate by reducing the function

A very effective technique:

- temporarily remove code from the function body until the crash disappears,
- then add sections back until it reappears.

The moment it reappears, you have identified the exact write or call sequence that triggers corruption.

25.3.7 Step 7: guard and canary logic (conceptual)

Even without relying on special tooling, you can use a manual canary in debug builds:

- place a known constant in stack local storage,
- check it before returning.

Rust demonstration (conceptual canary for unsafe code paths)

```rust
fn unsafe_block_with_manual_canary(buf: &mut [u8]) {
    let canary: u64 = 0xfeed_face_dead_beef;
    let mut local_canary = canary;

    unsafe {
        /* Simulate a buggy write */
        let p = buf.as_mut_ptr();
        for i in 0..(buf.len() + 16) {
            *p.add(i) = 0x41; /* BAD: writes past end */
        }
    }

    if local_canary != canary {
        /* In a real debug build you would abort here */
        println!("Stack corruption detected by canary");
    }
}
```

This is not a replacement for real stack protectors, but it helps explain how overwrites are detected.

25.3.8 Step 8: confirm unwind metadata assumptions (Windows)

If:

- stack traces stop at your assembly function, or
- exceptions crossing your function terminate,

then check whether the function is:

- leaf (no frame, no calls): often unwind-trivial,
- non-leaf or uses non-volatiles: likely needs correct unwind metadata under Windows x64.

A mismatch between actual prolog/epilog and unwind expectations can look like stack corruption.

25.4 Extensive Examples: Failure and Fix (Rust on Windows + x86-64 assembly)

25.4.1 Example A: Win64 call-site stack corruption by missing shadow space

Bad call wrapper

```
.intel_syntax noprefix
.text
.extern puts
.globl bad_wrapper_calls_puts
bad_wrapper_calls_puts:
    /* BAD: missing 32-byte shadow space */
    lea rcx, [rip !+! msg]
    call puts
    ret
.section .rdata,"dr"
msg: .asciz "hello"
```

Fixed call wrapper

```
.intel_syntax noprefix
.text
.extern puts
.globl good_wrapper_calls_puts
good_wrapper_calls_puts:
    sub rsp, 40
    lea rcx, [rip !+! msg]
    call puts
    add rsp, 40
    ret
.section .rdata,"dr"
msg: .asciz "hello"
```

25.4.2 Example B: Rust calling into a function that clobbers a non-volatile register

Bad assembly

```
.intel_syntax noprefix
.text
.globl bad_clobber_r12
bad_clobber_r12:
    /* BAD: R12 is non-volatile on Win64 */
    mov r12, 0x1122334455667788
    ret
```

Rust harness

```
extern "C" {
    fn bad_clobber_r12();
}
```

```
fn main() {
    unsafe {
        /* This may break unrelated code in optimized builds */
        bad_clobber_r12();
    }
    println!("If this prints reliably, you were just lucky.");
}
```

Fixed assembly

```
.intel_syntax noprefix
.text
.globl good_preserve_r12
good_preserve_r12:
    push r12
    mov  r12, 0x1122334455667788
    pop  r12
    ret
```

25.4.3 Example C: pointer-to-stack escape (unsafe Rust)

```
fn bad_stack_ptr() -> *const u64 {
    let x: u64 = 0xDEADBEEF;
    &x as *const u64
}

fn main() {
    let p = bad_stack_ptr();
    unsafe {
        /* UB: may print garbage or crash later */
        println!("value=0x{:x}", *p);
    }
```

```
}
```

Fix strategy. Return owned memory (`Box`, `Vec`) or store into caller-provided output buffer.

25.5 Practical Prevention Rules

1. Follow ABI stack rules: alignment, shadow space (Win64), and consistent frame size.
2. Save/restore non-volatile registers exactly as required by the ABI.
3. Avoid unsafe writes; validate lengths; never write past local buffers.
4. Treat calling convention and prototype mismatches as fatal.
5. For Windows x64 assembly that is non-trivial, ensure unwind behavior matches prolog/epilog patterns expected by the platform.
6. In Rust, keep unsafe blocks small; do not let stack pointers escape; validate raw pointer inputs at FFI boundaries.

25.6 Summary

- Stack corruption is usually an ABI invariant violation: wrong `rsp` adjustment, wrong alignment, missing shadow space, overflow, or wrong calling convention.
- Typical patterns repeat across projects: mismatched frames, buffer overruns, non-volatile clobbers, and pointer escape.
- Diagnosis is systematic: confirm ABI invariants, find the first bad frame, inspect call sites, isolate by reduction, and validate unwind assumptions.

- Rust reduces many categories of stack corruption, but unsafe code and FFI can reintroduce them; treat boundaries defensively.

Calling Convention Violations

26.1 What a Calling Convention Really Guarantees

A calling convention is not a stylistic choice. It is a binary contract between two independently compiled pieces of code (your code, the OS, the compiler runtime, a library, a foreign language, an injected hook, or even your own hand-written assembly).
At minimum, the contract defines:

- Where arguments are placed (registers vs stack, and in what order).
- Who owns stack cleanup (caller vs callee) on ABIs that require it.
- Which registers must be preserved by the callee (non-volatile) and which may be clobbered (volatile).
- Stack alignment requirements at call boundaries.
- Special required stack areas (e.g., Windows x64 **shadow space**).
- How return values are produced (register(s), hidden sret pointer, XMM registers, memory).
- How variadic arguments are passed, promoted, and discovered.
- How unwinding and exception metadata is produced (critical when mixing toolchains).

If **any** side violates the contract, the CPU still executes instructions correctly; what breaks is your assumptions about *where data is* and *what state the machine must be in*. The resulting failures are among the hardest to debug because they often crash **far away** from the real mistake.

26.2 Windows x64 ABI: The Contract You Must Obey

In modern Windows (x86-64), most languages and compilers converge to the Microsoft x64 calling convention for non-variadic functions.
Key rules that matter for low-level engineers:

- Integer/pointer arguments 1–4 are in `RCX, RDX, R8, R9`.
- Floating arguments 1–4 are in `XMM0–XMM3`.
- The caller must reserve **32 bytes of shadow space** on the stack *before every call*, even if the callee does not use it.
- The stack pointer must be **16-byte aligned at the point of the CALL instruction**.
- Non-volatile (callee-saved) registers: `RBX, RBP, RSI, RDI, R12–R15` and `XMM6–XMM15`.
- The callee may freely clobber volatile registers: `RAX, RCX, RDX, R8–R11` and `XMM0–XMM5`.
- There is **no red zone**. Do not write below `RSP` without allocating space.

Violating shadow space and alignment is one of the most common causes of crashes when mixing hand-written assembly, Rust FFI, and different compilers.

26.2.1 Correct Windows x64 call sequence in GAS (Intel syntax)

```
.intel_syntax noprefix
.text

/* extern "C" function we will call:
   int add4(int a, int b, int c, int d);
*/

.extern add4
.global call_add4
call_add4:
    /* Prologue */
    push rbp
    mov rbp, rsp

    /* Allocate shadow space (32 bytes) and keep 16-byte alignment.
       After "push rbp", RSP is 8 bytes misaligned.
       Subtracting 40 (32 shadow + 8 alignment fix) makes RSP 16-aligned. */
    sub rsp, 40

    /* Prepare args: RCX, RDX, R8, R9 */
    mov ecx, 1
    mov edx, 2
    mov r8d, 3
    mov r9d, 4

    call add4

    /* Epilogue */
    add rsp, 40
    pop rbp
    ret
```

26.2.2 The classic broken call: missing shadow space

```
.intel_syntax noprefix
.text

.extern add4
.global broken_call_add4
broken_call_add4:
    /* WRONG: no shadow space; callee may spill into [rsp+0..31] */
    mov ecx, 1
    mov edx, 2
    mov r8d, 3
    mov r9d, 4
    call add4
    ret
```

This can crash in multiple ways:

- The callee stores register arguments into the shadow area and overwrites your return address or other stack data.
- A deeper callee assumes aligned stack for XMM operations and faults on aligned loads/stores.
- It may appear to work until optimization changes the callee's prologue or register allocation.

26.3 Real Crash Signatures

Calling convention violations do not have a single signature; they create patterns. The key is to recognize *machine-state breakage* rather than application logic failure.

26.3.1 Signature A: Crash at RET or immediately after it

Common symptoms:

- Instruction pointer becomes an implausible address (`RIP=0x0000000000000001`, `0xCCCCCCCCCCCCCCCC`, `0xFEEEFEEE...`, or a pointer into stack/heap).
- Stack trace becomes garbage: frames with nonsense return addresses, missing module names, or impossible call nesting.
- The failure happens in the function epilogue: `pop rbp; ret` or `ret`.

Root causes:

- Stack pointer (`RSP`) was not restored to the expected value.
- The return address was overwritten because shadow space was missing or stack cleanup mismatched.
- A callee clobbered a non-volatile register that the caller later uses as a pointer.

26.3.2 Signature B: Access violation in a totally unrelated function

Common symptoms:

- Crash happens inside a standard library routine, allocator, logging function, or OS API.
- It looks like memory corruption, but sanitizers do not always catch it because the corruption occurs at ABI boundaries.

Root causes:

- Wrong argument register mapping: callee reads a pointer from the wrong register and dereferences it.

- Wrong type width: 32-bit truncation of a 64-bit pointer or size, leading to invalid memory access.
- Wrong struct layout or alignment: callee reads fields at incorrect offsets, producing nonsense pointers.

26.3.3 Signature C: SIMD/alignment fault (often optimization-dependent)

Common symptoms:

- The crash appears only in Release builds, not Debug.
- Faulting instruction is an aligned SIMD load/store or an instruction sequence assuming stack alignment.
- The crash disappears if you add logging or change unrelated code (heisenbug).

Root cause:

- Stack misalignment at call boundary (violating the ABI rule that stack must be aligned at `call`).

26.3.4 Signature D: Exception/unwind chaos when mixing toolchains

Common symptoms:

- A thrown exception or panic causes termination instead of unwinding.
- Stack traces are truncated or incorrect.
- The process terminates during unwinding or at a personality routine.

Root causes:

- Mixing incompatible exception models (notably across different Windows toolchains and runtimes).
- Letting C++ exceptions cross C/Rust boundaries, or letting Rust panics cross into foreign code.

26.4 Cross-Compiler Disasters

Even if two compilers target the same CPU, they may not be compatible at the binary boundary unless you deliberately constrain the interface.
The most common disaster patterns on Windows are:

26.4.1 Disaster 1: MSVC vs MinGW runtime boundary

It is possible to make object code link, and still be wrong at runtime.
Typical hazards:

- Different C runtime (CRT) allocation domains: memory allocated in one runtime is freed in another.
- Different unwind/exception metadata expectations: unwinding across the boundary is undefined in practice.
- Subtle structure layout differences when not using `repr(C)` on the Rust side or when using non-standard packing on the C/C++ side.

Engineering rule:

- If you must mix toolchains, keep the boundary **C-only**, **no exceptions**, **no cross-runtime allocation ownership**, **fixed-width types**, **repr(C) structs**, and **explicit ownership APIs**.

26.4.2 Disaster 2: SysV assumptions accidentally used on Windows x64

Developers who learned x86-64 ABI on Linux often bring SysV habits to Windows:

- SysV uses `RDI, RSI, RDX, RCX, R8, R9` for integer args; Windows uses `RCX, RDX, R8, R9`.
- SysV has a red zone; Windows does not.
- SysV does not require Windows-style shadow space.

If you write assembly on Windows as if it were SysV, your program can crash while appearing superficially correct.

26.4.3 Disaster 3: 32-bit calling convention mismatches (cdecl/stdcall/thiscall)

On 32-bit x86 Windows, calling conventions differ by:

- stack cleanup responsibility,
- argument order and placement,
- name decoration,
- register usage for `this` pointer in methods.

A mismatch can corrupt the stack **immediately** (often crashing on return) or **later** (stack gradually drifts).
Even if your main target is x86-64, this still matters when:

- you maintain legacy components,
- you deal with injected hooks,
- you build a 32-bit test harness, loader, or compatibility tool.

26.5 Rust on Windows: ABI Strings Are Not Documentation, They Are Instructions

Rust provides explicit ABI selectors for FFI. The choice changes how arguments are passed and how the symbol is callable.
Key choices:

- `extern "C"`: the platform's C ABI (Windows x64 uses the MS x64 convention).
- `extern "system"`: the platform's system ABI (on Windows, often matches Windows API calling convention).
- `extern "stdcall"`, `extern "fastcall"`: relevant mainly to 32-bit x86 Windows.

26.5.1 Example: Correct Rust export callable from GAS assembly (Windows x64)

```
#![allow(non_snake_case)]

#[no_mangle]
pub extern "C" fn add4(a: i32, b: i32, c: i32, d: i32) -> i32 {
    a + b + c + d
}
```

With the earlier `call_add4` assembly, this is a stable boundary.

26.5.2 Example: Wrong ABI in Rust declaration (can crash instantly)

Assume a foreign library provides a function with the Windows x64 ABI. If you declare it incorrectly, Rust will place arguments where the callee does not expect them.

```rust
use core::ffi::c_void;

#[link(name = "mylib")]
extern "C" {
    /* Suppose the real function expects (ptr, len) but we declared them swapped. */
    fn process_buffer(len: usize, ptr: *mut c_void) -> i32;
}

pub fn demo(ptr: *mut c_void, len: usize) -> i32 {
    unsafe { process_buffer(len, ptr) }
}
```

This compiles and links, but at runtime:

- the callee interprets the length as a pointer and dereferences it, or
- interprets the pointer as a length and reads far beyond bounds.

The crash often appears in the callee (or deeper), far from the declaration site.

26.6 Violations You Can Reproduce and Learn From

The following failures are valuable because they teach you to recognize ABI breakage with deterministic examples.

26.6.1 Violation 1: Stack corruption via 32-bit stdcall vs cdecl mismatch

This example is for 32-bit x86 Windows. It demonstrates the classic stack cleanup mismatch.

```rust
#![cfg(target_arch = "x86")]

#[repr(C)]
pub struct Pair {
```

```
    pub a: i32,
    pub b: i32,
}

/* Exported as stdcall (callee cleans stack) */
#[no_mangle]
pub extern "stdcall" fn sum_pair(p: Pair) -> i32 {
    p.a + p.b
}
```

Now imagine a C caller (or assembly) calls it as `cdecl` (caller cleans stack). Both sides clean the stack: **stack pointer drifts**. The crash may occur on the next `ret` in an unrelated function.
A 32-bit GAS caller demonstrating the mismatch:

```
.intel_syntax noprefix
.text
.global call_sum_pair_as_cdecl
.extern _sum_pair@8

call_sum_pair_as_cdecl:
    /* Push args (Pair by value: two 32-bit ints) */
    push 2
    push 1

    /* Call stdcall function but treat it like cdecl */
    call _sum_pair@8

    /* WRONG: caller also cleans stack (stdcall already did) */
    add esp, 8

    ret
```

This is not a logic bug. It is a contract violation.

26.6.2 Violation 2: Non-volatile register clobber (Windows x64)

If you write assembly and clobber a non-volatile register without restoring it, the caller may later use it as a pointer or loop index.

Broken function:

```
.intel_syntax noprefix
.text
.global broken_clobber_rbx
broken_clobber_rbx:
    /* WRONG: RBX is non-volatile on Windows x64 */
    mov rbx, 0x1122334455667788
    ret
```

Correct function:

```
.intel_syntax noprefix
.text
.global good_clobber_rbx
good_clobber_rbx:
    push rbx
    mov rbx, 0x1122334455667788
    /* ... work ... */
    pop rbx
    ret
```

Rust caller demonstrating the symptom:

```
extern "C" {
    fn broken_clobber_rbx();
    fn good_clobber_rbx();
}

#[no_mangle]
```

```
pub extern "C" fn rust_calls_asm(flag: i32) -> i64 {
    let mut x: i64 = 10;
    let mut y: i64 = 20;

    unsafe {
        if flag != 0 {
            broken_clobber_rbx();
        } else {
            good_clobber_rbx();
        }
    }

    /* If the optimizer kept something important in RBX across the call,
       the broken version can corrupt program state in ways that look random. */
    x += 1;
    y += x;
    y
}
```

In Release builds, this class of bug can manifest as memory corruption rather than an immediate crash.

26.6.3 Violation 3: Missing shadow space when calling into Rust from assembly (Windows x64)

Rust `extern "C"` on Windows x64 expects the caller to honor shadow space rules, because Rust code is generated by LLVM to obey the platform ABI.

Rust export:

```
#[no_mangle]
pub extern "C" fn mul_add(a: i64, b: i64, c: i64) -> i64 {
    a * b + c
}
```

Broken assembly caller:

```
.intel_syntax noprefix
.text
.extern mul_add
.global broken_call_mul_add

broken_call_mul_add:
    /* WRONG: no shadow space, stack alignment unknown */
    mov rcx, 6
    mov rdx, 7
    mov r8,  8
    call mul_add
    ret
```

Correct assembly caller:

```
.intel_syntax noprefix
.text
.extern mul_add
.global good_call_mul_add

good_call_mul_add:
    push rbp
    mov rbp, rsp
    sub rsp, 40             /* 32 shadow + 8 align fix */
    mov rcx, 6
    mov rdx, 7
    mov r8,  8
    call mul_add
    add rsp, 40
    pop rbp
    ret
```

26.6.4 Violation 4: Struct layout mismatch (repr(C) is not optional)

If you pass structs across the ABI boundary without `repr(C)`, Rust is free to reorder fields (and alignment may differ). The result is silent corruption.
Wrong Rust definition (do not do this for FFI):

```
pub struct Header {
    pub len: u32,
    pub flags: u16,
    pub kind: u8,
    pub pad: u8,
}
```

Correct FFI-safe definition:

```
#[repr(C)]
pub struct Header {
    pub len: u32,
    pub flags: u16,
    pub kind: u8,
    pub pad: u8,
}
```

A C side expecting `len` at offset 0 will read nonsense if the Rust layout differs. The crash may occur later, when the misread length becomes a huge value and drives an out-of-bounds access.

26.6.5 Violation 5: Variadic boundary misuse

Variadic functions (`...`) are ABI-sensitive and are a common source of disasters when declared incorrectly.
Engineering rule on Windows:

- Prefer non-variadic wrappers at the ABI boundary.

- If you must use varargs, isolate them behind a C-only shim and expose a fixed signature to Rust/assembly.

26.7 Checklist: How to Prevent Calling Convention Violations

Use this checklist as a pre-flight gate before integrating any low-level boundary.

26.7.1 Boundary design rules

- Export/import only **global functions** with explicit ABI: `extern "C"` or `extern "system"` as required.
- Use fixed-width types: `i32/u32/i64/u64`, and explicit pointer types. Avoid ambiguous types at boundaries.
- For sizes, be explicit about `usize` vs `u32`. If the other side is C, use `size_t` equivalence carefully and document 32/64-bit expectations.
- Use `#[repr(C)]` for every struct passed across the boundary.
- Never let Rust panics cross the boundary. Wrap exported Rust functions: catch unwind, return error codes, and keep unwinding inside Rust.
- Never let C++ exceptions cross the boundary. Catch them in C++ and translate to error codes.
- Do not share ownership of memory across different runtimes unless you provide paired alloc/free APIs in the same module.

26.7.2 Assembly call rules for Windows x64

- Always allocate 32-byte shadow space before calling any external function.
- Maintain 16-byte alignment at the CALL instruction.
- Preserve all non-volatile registers you touch (`RBX, RBP, RSI, RDI, R12–R15, XMM6–XMM15`).
- Do not use red zone assumptions.

26.8 Laboratory: Diagnose a Convention Violation Like an Engineer

When you suspect a calling convention violation, do not start with source-level logic. Start with invariants:

26.8.1 Step 1: Verify argument registers at the call boundary

- On Windows x64, confirm values in `RCX, RDX, R8, R9` (and `XMM0–XMM3` if floats).
- Confirm the called symbol is the one you think it is (wrong import library or wrong decoration can redirect calls).

26.8.2 Step 2: Verify stack alignment and shadow space

- Confirm `RSP` alignment at call sites.
- Confirm shadow space is reserved and not overwritten by your own code.

26.8.3 Step 3: Verify non-volatile registers across the call

- If a pointer in `RBX/RDI/RSI/R12–R15` suddenly becomes invalid, suspect clobber.

26.8.4 Step 4: Reduce boundary complexity

- Replace structs with pointers + explicit sizes.
- Replace callbacks with integer handles and explicit trampoline functions.
- Replace varargs with fixed wrappers.

26.9 Summary

Calling convention violations are not rare. They are the default outcome whenever engineers treat ABI boundaries as a detail.
Real systems fail because:

- one side used SysV assumptions on Windows,
- shadow space was forgotten,
- stack alignment was accidentally broken,
- a non-volatile register was clobbered,
- a struct lacked `repr(C)`,
- or two toolchains were mixed without designing a safe boundary.

If your booklet teaches anything at a foundational level, let it teach this: **Correctness at the ABI boundary is not optional. It is the difference between software and superstition.**

Alignment Mistakes

27.1 Why Alignment Bugs Are Uniquely Dangerous

Alignment mistakes are not just performance problems. In low-level engineering, alignment is a **correctness contract** enforced by:

- the CPU (some instructions **fault** on misaligned addresses),
- the ABI (stack alignment and call-boundary rules),
- the compiler (it will generate aligned moves when it believes alignment is guaranteed),
- and SIMD intrinsics (many are specified to require aligned pointers unless you choose the unaligned variant).

What makes alignment failures hard:

- They often appear only in optimized builds.
- They can crash far from the origin of the bug.
- A single missing `sub rsp, ...` or a single `packed` struct can turn safe code into undefined behavior across a whole module.

This chapter focuses on two high-impact failure classes:

- **SIMD crashes** (faulting aligned vector loads/stores, stack misalignment causing SIMD prolog faults).
- **ABI violations** (call-boundary alignment and required stack layout rules, especially on Windows x64).

27.2 Alignment Fundamentals You Must Treat as Invariants

27.2.1 Natural alignment vs required alignment

For ordinary scalar types, most CPUs can access misaligned data (sometimes slower). SIMD is different:

- Aligned SIMD moves (e.g., `movaps`, `movdqa`, `vmovdqa*`) may **fault** if the memory operand is not aligned to the vector width they require.
- Unaligned SIMD moves (e.g., `movups`, `movdqu`, `vmovdqu*`) are designed to work with any alignment (still potentially slower).

27.2.2 Stack alignment is an ABI rule, not an optimization hint

On x86-64, mainstream ABIs require a specific stack alignment at call boundaries. Compilers assume it and will generate instructions that **depend on it**:

- Stack-aligned local storage for XMM spills and vectorized locals.
- Aligned stack-based temporaries for library functions.
- Aligned moves for function prolog/epilog code in optimized builds.

Breaking stack alignment is one of the fastest ways to trigger **SIMD faults inside unrelated code** such as formatting, parsing, math, or crypto routines.

27.3 SIMD Crashes

27.3.1 Crash Pattern 1: Aligned vector load/store from an unaligned pointer

The classic failure: your code (or the compiler) uses an aligned instruction but the pointer is not sufficiently aligned.

Minimal assembly reproducer (faulting `movaps`)

```
.intel_syntax noprefix
.text
.global simd_fault_movaps
simd_fault_movaps:
    /* RDI points to data (SysV style for demonstration). On Windows x64, the first arg is
    ↪  in RCX. */
    movaps xmm0, XMMWORD PTR [rdi]   /* Requires 16-byte alignment; may fault if misaligned
    ↪  */
    ret
```

Safe alternative (works regardless of alignment):

```
.intel_syntax noprefix
.text
.global simd_safe_movups
simd_safe_movups:
    movups xmm0, XMMWORD PTR [rdi]   /* Unaligned load */
    ret
```

What the crash looks like

Common symptoms:

- Faulting instruction: `movaps` / `movdqa` / `vmovdqa*`.

- The fault occurs at the first instruction of a function or in a library routine.
- Debug builds may not reproduce because the compiler uses different moves or avoids vectorization.

27.3.2 Crash Pattern 2: Stack misalignment triggers SIMD faults in callees

Even if your own code never uses SIMD instructions, **your callees might**. If you violate stack alignment rules, a callee may execute an aligned load/store to the stack and fault.
This is extremely common on Windows x64 when:

- shadow space is missing,
- stack alignment at the `call` instruction is wrong,
- or hand-written assembly forgets the ABI-mandated layout.

Windows x64: correct call setup for SIMD-safe callees

On Windows x64:

- Caller must reserve **32 bytes shadow space** before every call.
- Stack must be **16-byte aligned at the CALL instruction**.

Correct call sequence in GAS (Intel syntax):

```
.intel_syntax noprefix
.text
.extern callee_uses_simd
.global win64_call_correct
win64_call_correct:
    push rbp
    mov rbp, rsp
```

```
    /* After push, RSP is misaligned by 8.
       Reserve 32 bytes shadow + 8 bytes alignment fix = 40 bytes. */
    sub rsp, 40

    /* Windows x64 arg registers: RCX, RDX, R8, R9 */
    xor ecx, ecx
    call callee_uses_simd

    add rsp, 40
    pop rbp
    ret
```

Broken call sequence (common real-world bug):

```
.intel_syntax noprefix
.text
.extern callee_uses_simd
.global win64_call_broken
win64_call_broken:
    /* WRONG: no shadow space, alignment unknown */
    xor ecx, ecx
    call callee_uses_simd
    ret
```

A callee that may fault if stack is misaligned

```
.intel_syntax noprefix
.text
.global callee_uses_simd
callee_uses_simd:
    /* Typical prolog that uses aligned stores to stack for XMM regs */
    sub rsp, 32
    movaps XMMWORD PTR [rsp], xmm6   /* May fault if stack not aligned as ABI expects */
```

```
    movaps xmm6, xmm6
    movaps xmm6, XMMWORD PTR [rsp]
    add rsp, 32
    ret
```

27.3.3 Crash Pattern 3: Over-alignment assumptions (AVX/AVX-512 locals and spills)

Even though the ABI might only require 16-byte stack alignment, compilers can generate code that benefits from larger alignments:

- 32-byte for YMM (AVX) spills and local arrays.
- 64-byte for ZMM (AVX-512) related data in some strategies.

Important rule:

- If the compiler believes an object is 32- or 64-aligned, it may use aligned moves of that width.
- If you break that belief (by mis-declaring alignment across FFI, by casting, or by creating misaligned views), you can get faults or silent data corruption.

27.3.4 Rust on Windows: Intrinsics and alignment requirements

Rust (on Windows) uses the platform ABI for `extern "C"` and will rely on correct stack alignment at call boundaries.
Additionally, SIMD intrinsics enforce alignment expectations:

- Some intrinsics are defined as aligned loads/stores and require aligned pointers.
- Unaligned variants exist and must be used when alignment cannot be guaranteed.

Rust example: aligned vs unaligned loads

```
#![cfg(target_arch = "x86_64")]
use core::arch::x86_64::*;

#[no_mangle]
pub extern "C" fn sum4_aligned(ptr: *const f32) -> f32 {
    unsafe {
        /* This is conceptually an aligned load; calling it with misaligned ptr is a bug. */
        let v = _mm_load_ps(ptr);     /* expects 16-byte alignment */
        let shuf = _mm_shuffle_ps(v, v, 0b_11_10_01_00);
        let add1 = _mm_add_ps(v, shuf);
        let shuf2 = _mm_movehl_ps(shuf, add1);
        let add2 = _mm_add_ss(add1, shuf2);
        _mm_cvtss_f32(add2)
    }
}

#[no_mangle]
pub extern "C" fn sum4_unaligned(ptr: *const f32) -> f32 {
    unsafe {
        let v = _mm_loadu_ps(ptr);    /* works with any alignment */
        let shuf = _mm_shuffle_ps(v, v, 0b_11_10_01_00);
        let add1 = _mm_add_ps(v, shuf);
        let shuf2 = _mm_movehl_ps(shuf, add1);
        let add2 = _mm_add_ss(add1, shuf2);
        _mm_cvtss_f32(add2)
    }
}
```

Engineering rule for Rust FFI:

- If the caller cannot guarantee alignment, **only export the unaligned variant**, or export both and document the contract in your API design.

Rust example: creating correctly aligned data

```
#[repr(align(16))]
pub struct Align16<T>(pub T);

#[no_mangle]
pub extern "C" fn aligned_buffer_demo() -> f32 {
    let a = Align16([1.0f32, 2.0, 3.0, 4.0]);
    let ptr = a.0.as_ptr();
    unsafe { core::arch::x86_64::_mm_cvtss_f32(core::arch::x86_64::_mm_load_ps(ptr)) }
}
```

27.3.5 Crash Pattern 4: Packed structs and misaligned fields

Packed structs are a frequent source of alignment faults and UB:

- A packed field may be misaligned inside the struct.
- Taking a reference to a misaligned field is invalid for many languages and compilers.
- Even if the CPU can load it, the compiler may assume alignment and generate faulting moves or reorder accesses incorrectly.

Rust hazard demonstration:

```
#[repr(C, packed)]
pub struct Packed {
    pub tag: u8,
    pub x: u32,   /* likely misaligned (offset 1) */
}

#[no_mangle]
pub extern "C" fn read_packed_x(p: *const Packed) -> u32 {
    unsafe {
```

```
        /* This is a classic bug pattern: reading a possibly misaligned u32 through a
        ↪  reference-like access. */
        /* Correct handling uses ptr::read_unaligned. */
        core::ptr::read_unaligned(core::ptr::addr_of!((*p).x))
    }
}
```

27.4 ABI Violations

Alignment mistakes become ABI violations when they break the calling convention or object layout guarantees at binary boundaries.

27.4.1 Violation 1: Call-boundary stack alignment (Windows x64)

Windows x64 requires:

- 16-byte aligned stack at `call`.
- 32-byte shadow space reserved by caller.

If you violate either rule, you can trigger:

- SIMD faults in callee prolog/epilog,
- corrupted return addresses,
- corrupted home space spills,
- crashes in unrelated library routines.

Correct Windows x64 prolog/epilog template in GAS

```
.intel_syntax noprefix
.text
.global win64_template
win64_template:
    /* Preserve non-volatile regs you modify (RBX, RBP, RSI, RDI, R12-R15, XMM6-XMM15) */
    push rbp
    mov rbp, rsp

    /* Reserve local space and shadow space for calls.
       Use multiples of 16 and account for the pushed return address and pushes you do. */
    sub rsp, 40

    /* ... body ... may call other functions safely ... */

    add rsp, 40
    pop rbp
    ret
```

27.4.2 Violation 2: Over-aligned types across FFI without matching declarations

Over-alignment above 16 bytes is a common trap:

- One side declares `alignas(32)` or `#[repr(align(32))]` for performance.
- The other side assumes default alignment and allocates the object normally.
- The callee uses aligned AVX moves (32-byte) and faults or reads wrong data.

Rust side (over-aligned type):

```
#[repr(C, align(32))]
```

```rust
pub struct Vec8f32 {
    pub v: [f32; 8],
}

#[no_mangle]
pub extern "C" fn sum8_aligned(p: *const Vec8f32) -> f32 {
    unsafe {
        use core::arch::x86_64::*;
        let ptr = (*p).v.as_ptr();
        /* If p is not actually 32-byte aligned as promised, an aligned load may fault or be
        ↪ UB via assumptions. */
        let x = _mm256_load_ps(ptr); /* 32-byte alignment contract */
        let hi = _mm256_extractf128_ps(x, 1);
        let lo = _mm256_castps256_ps128(x);
        let add = _mm_add_ps(lo, hi);
        let shuf = _mm_shuffle_ps(add, add, 0b_11_10_01_00);
        let add2 = _mm_add_ps(add, shuf);
        let shuf2 = _mm_movehl_ps(shuf, add2);
        let add3 = _mm_add_ss(add2, shuf2);
        _mm_cvtss_f32(add3)
    }
}
```

Engineering rule:

- If you export an over-aligned type across FFI, the allocator and all constructors on the other side must honor that alignment. Otherwise, redesign the boundary to use `loadu` variants or pass raw pointers with explicit alignment checks.

27.4.3 Violation 3: Returning or passing structs with mismatched layout/alignment

Even with `repr(C)` on Rust, the boundary can still break if the other side uses different packing pragmas or nonstandard ABI flags.
Rust safe declaration:

```
#[repr(C)]
pub struct Header {
    pub len: u32,
    pub flags: u16,
    pub kind: u16,
}
```

If the C/C++ side uses packing that changes alignment, offsets change and the ABI contract breaks. The resulting failures often look like:

- huge lengths,
- invalid pointers,
- impossible flags,
- crashes in loops that trust these fields.

Engineering rule:

- Never use packed layout across a public ABI unless you **also** force unaligned-safe access patterns and treat every field load/store as potentially unaligned.

27.4.4 Violation 4: Rust `extern` ABI mismatch for Windows API calls

On Windows, calling conventions differ historically on 32-bit; on 64-bit most conventions collapse into the Windows x64 convention, but the `extern` string is still a real instruction to the compiler.

Correct pattern for Windows API bindings is typically `extern "system"` (especially relevant for 32-bit compatibility and correctness-by-intent):

```
use core::ffi::c_void;

type BOOL = i32;
type HANDLE = *mut c_void;
type DWORD = u32;

extern "system" {
    fn CloseHandle(h: HANDLE) -> BOOL;
}

#[no_mangle]
pub extern "C" fn close_handle_safe(h: HANDLE) -> BOOL {
    unsafe { CloseHandle(h) }
}
```

Engineering rule:

- If you bind OS APIs, use the ABI that matches the platform's system calling convention, not assumptions.

27.5 Alignment Debug Playbook (Academy Workflow)

When you suspect alignment is involved, do not start by rewriting logic. Prove or disprove invariants.

27.5.1 Playbook A: Prove pointer alignment at the fault site

- If the faulting instruction is an aligned SIMD move, inspect the effective address and compute address modulo required alignment (16/32/64).

- If it is stack-based, inspect `RSP` at the call boundary and at the faulting instruction.

27.5.2 Playbook B: Prove stack alignment at every external call

For Windows x64 assembly:

- Verify you reserved 32-byte shadow space.
- Verify `RSP` is 16-byte aligned at `call`.
- Verify you did not write into the shadow space as if it were your locals.

27.5.3 Playbook C: Find the first boundary that breaks alignment

- Replace aligned loads with unaligned loads temporarily (`movaps` → `movups`, `vmovdqa` → `vmovdqu`) to confirm alignment causality.
- Replace vectorized code with scalar code temporarily to confirm the crash is SIMD-triggered.
- Reduce the ABI boundary: pass raw pointers and sizes; avoid structs until stable.

27.6 Engineering Patterns That Prevent Alignment Failures

27.6.1 Pattern 1: Always offer an unaligned-safe API at module boundaries

Expose:

- `foo_unaligned(ptr)` for general callers,
- optional `foo_aligned(ptr)` for high-performance callers who can guarantee alignment.

27.6.2 Pattern 2: Validate alignment in debug builds

You can add a cheap debug-only guard in Rust before using aligned intrinsics:

```rust
#[inline]
fn is_aligned(ptr: *const u8, align: usize) -> bool {
    (ptr as usize) & (align - 1) == 0
}

#[no_mangle]
pub extern "C" fn sum4_checked(ptr: *const f32) -> f32 {
    let p = ptr as *const u8;
    debug_assert!(is_aligned(p, 16));
    unsafe { core::arch::x86_64::_mm_cvtss_f32(core::arch::x86_64::_mm_load_ps(ptr)) }
}
```

27.6.3 Pattern 3: Do not trust implicit alignment across FFI

At boundaries:

- Prefer explicit allocation functions that guarantee alignment.
- Or accept unaligned pointers and internally choose unaligned loads.

27.6.4 Pattern 4: Never write Windows x64 assembly without a fixed template

Treat the Windows x64 call template as mandatory:

- shadow space,
- 16-byte stack alignment,
- non-volatile register preservation,
- no red-zone assumptions.

27.7 Laboratory Exercises

27.7.1 Lab 1: Trigger a `movaps` fault with an intentionally misaligned buffer

```
#![cfg(target_arch = "x86_64")]
use core::arch::x86_64::*;

#[no_mangle]
pub extern "C" fn lab_movaps_fault() -> f32 {
    /* Create a buffer with extra bytes so we can offset it to misalign. */
    let mut buf = [0u8; 32];
    let base = buf.as_mut_ptr();
    let misaligned = unsafe { base.add(4) } as *const f32; /* 4-byte offset breaks 16-byte
    ↪  alignment */

    unsafe {
        /* This may crash/fault because misaligned is not 16-byte aligned. */
        let v = _mm_load_ps(misaligned);
        _mm_cvtss_f32(v)
    }
}
```

27.7.2 Lab 2: Fix Lab 1 by switching to unaligned loads

```
#![cfg(target_arch = "x86_64")]
use core::arch::x86_64::*;

#[no_mangle]
pub extern "C" fn lab_movups_safe() -> f32 {
    let mut buf = [0u8; 32];
    let base = buf.as_mut_ptr();
    let misaligned = unsafe { base.add(4) } as *const f32;
```

```
    unsafe {
        let v = _mm_loadu_ps(misaligned);
        _mm_cvtss_f32(v)
    }
}
```

27.7.3 Lab 3: Demonstrate Windows x64 stack alignment bug in assembly

Use the broken caller from this chapter and call any callee that uses aligned XMM stack spills. Observe:

- it may crash immediately in the callee prolog,
- or it may corrupt state and crash later.

27.8 Summary

Alignment mistakes are one of the most expensive classes of low-level bugs because they combine:

- CPU fault behavior (aligned SIMD moves),
- ABI call-boundary requirements (stack alignment and required stack layout),
- compiler assumptions (vectorization and aligned spills),
- and FFI layout constraints (over-aligned types, packed fields).

If you want a single rule to enforce across your entire toolchain:

- Make alignment a **declared contract** at boundaries, and default to **unaligned-safe** operations unless you can prove alignment end-to-end.

Inline Assembly Pitfalls

28.1 Why Inline Assembly Is a High-Risk Boundary

Inline assembly is not just "assembly inside a high-level language". It is a **three-way contract** between:

- your assembly instructions,
- the compiler optimizer and register allocator,
- and the platform ABI (especially on Windows x64).

If any side is misinformed, the program can miscompile without warnings. The most common failures cluster into:

- **Clobbers**: the compiler assumes registers/flags/memory are unchanged when they are not.
- **Compiler reordering**: the compiler moves loads/stores across inline asm because you did not model side effects.
- **Constraint errors**: you asked for the wrong registers, wrong widths, wrong aliasing rules, or wrong input/output relationships.

This chapter uses **Rust on Windows** and **GAS Intel syntax** examples. The principles apply to any LLVM-backed toolchain.

28.2 Inline Assembly in Rust on Windows: The Model

Rust provides `asm!` for inline assembly. The key design principle is:

- You must fully describe **inputs**, **outputs**, and **side effects** so the compiler can optimize safely.

Inline assembly correctness depends more on the operand and options specification than on the instructions.

28.2.1 Rust `asm!` operands (core patterns)

- `in(reg) x`: input operand in a general register chosen by compiler.
- `out(reg) y`: output operand written by asm (value is produced).
- `lateout(reg) y`: output written *after* all inputs are consumed; allows input/output register reuse safely.
- `inout(reg) x => y`: same register is both input and output (read then written).
- Fixed registers: `in("rcx")`, `out("rax")`, etc. (use sparingly).

28.2.2 Rust `asm!` options (side-effect modeling)

- `volatile`: prevents the compiler from deleting or merging the asm block; still does not automatically create a full memory barrier.
- `nomem`: promises the asm does not read or write memory.
- `readonly`: promises the asm may read memory but does not write memory.

- `pure`: promises deterministic outputs depend only on inputs and memory reads; can be CSE'd/hoisted; must not have hidden side effects.
- `nostack`: promises the asm does not touch the stack pointer or use the stack.
- `preserves_flags`: promises the asm does not change CPU flags.

If any promise is false, you create undefined behavior at the optimizer level.

28.3 Clobbers

28.3.1 The core idea

A clobber is any machine state your asm modifies that the compiler might rely on later. In practice:

- **General registers** (RAX, RCX, etc.)
- **Flags** (EFLAGS/RFLAGS)
- **Memory** (reads/writes not visible through operands)
- **Stack pointer / call frame** (RSP, alignment, Windows x64 shadow space if you call)

Rust inline asm does not use textual "clobber lists" like GCC; you express clobbers by:

- using outputs/inout operands for registers you write,
- not claiming `preserves_flags` if you modify flags,
- not claiming `nomem/readonly/pure` if you touch memory,
- not claiming `nostack` if you touch the stack.

28.3.2 Clobber mistake 1: Writing a register without declaring it

Broken example (writes `rax` but does not declare it as output). The compiler may keep a live value in `rax` across the asm and later use it.

```
use core::arch::asm;

#[no_mangle]
pub extern "C" fn broken_hidden_rax(mut x: u64) -> u64 {
    unsafe {
        asm!(
            "mov rax, 1234",
            /* WRONG: no outputs; compiler assumes asm has no register outputs */
            options(nomem, nostack)
        );
    }
    /* Compiler might have placed x in rax and now it is destroyed. */
    x.wrapping_add(1)
}
```

Correct example: declare `rax` as an output (or better: let the compiler choose a register).

```
use core::arch::asm;

#[no_mangle]
pub extern "C" fn good_declared_output(x: u64) -> u64 {
    let mut out: u64;
    unsafe {
        asm!(
            "mov {o}, 1234",
            o = lateout(reg) out,
            options(nomem, nostack)
        );
    }
```

```
    out ^ x
}
```

28.3.3 Clobber mistake 2: Lying about flags with `preserves_flags`

Many instructions modify flags: `add`/`sub`/`inc`/`dec`/`cmp`/`test`/`shl`/`ror` and others. If you claim `preserves_flags` but change flags, the compiler may schedule flag-dependent code assuming flags are intact.
Broken example:

```
use core::arch::asm;

#[no_mangle]
pub extern "C" fn broken_preserves_flags(a: u64, b: u64) -> u64 {
    let mut x = a;
    unsafe {
        asm!(
            "add {x}, {b}",
            x = inout(reg) x,
            b = in(reg) b,
            /* WRONG: add modifies flags */
            options(nomem, nostack, preserves_flags)
        );
    }
    x
}
```

Correct example: simply remove `preserves_flags` when flags may change.

```
use core::arch::asm;

#[no_mangle]
pub extern "C" fn good_flags_modeled(a: u64, b: u64) -> u64 {
```

```
    let mut x = a;
    unsafe {
        asm!(
            "add {x}, {b}",
            x = inout(reg) x,
            b = in(reg) b,
            options(nomem, nostack) /* flags are allowed to be clobbered */
        );
    }
    x
}
```

28.3.4 Clobber mistake 3: Touching memory while claiming `nomem` or `readonly`

If your asm reads or writes memory that is not expressed via explicit pointer operands in a way the compiler understands, you must not claim `nomem`. If it writes, do not claim `readonly`. If it has side effects, do not claim `pure`.

Broken example: writes to a memory-mapped register but claims `nomem`.

```
use core::arch::asm;

#[no_mangle]
pub extern "C" fn broken_mmio_write(mmio: *mut u32, v: u32) {
    unsafe {
        asm!(
            "mov DWORD PTR [{p}], {v:e}",
            p = in(reg) mmio,
            v = in(reg) v,
            /* WRONG: this writes memory */
            options(nomem, nostack, volatile)
        );
```

```
    }
}
```

Correct example: do not claim `nomem`. Also avoid `pure`. Keep `volatile` because MMIO writes must not be removed.

```
use core::arch::asm;

#[no_mangle]
pub extern "C" fn good_mmio_write(mmio: *mut u32, v: u32) {
    unsafe {
        asm!(
            "mov DWORD PTR [{p}], {v:e}",
            p = in(reg) mmio,
            v = in(reg) v,
            options(nostack, volatile)
        );
    }
}
```

28.3.5 Clobber mistake 4: Modifying the stack or calling from inline asm on Windows x64

On Windows x64, every call requires:

- 32-byte shadow space reserved by the caller,
- 16-byte stack alignment at the `call` instruction.

Inline asm that performs `call` is fragile because the compiler also manages the stack. You can easily violate alignment, shadow-space, unwind expectations, and register preservation. Engineering rule:

- Do not perform function calls from Rust inline asm on Windows. Put such code in a dedicated `.S` file and obey the ABI with a full prolog/epilog.

28.4 Compiler Reordering

28.4.1 The core idea

The compiler is allowed to reorder instructions as long as it preserves the **observable behavior** under the language memory model.
Inline asm creates reordering hazards because:

- the compiler may treat asm as a pure computation if you promised `nomem/readonly/pure`,
- the compiler may move memory operations across asm unless it sees a dependency or side effect,
- even `volatile` does not automatically become a full memory barrier.

28.4.2 Reordering mistake 1: Missing dependency between asm and memory

Broken example: the asm reads from the pointer, but the compiler does not see the memory dependence if you hide it behind claims.

```
use core::arch::asm;

#[no_mangle]
pub extern "C" fn broken_hidden_load(p: *const u64) -> u64 {
    let out: u64;
    unsafe {
        asm!(
            "mov {o}, QWORD PTR [{p}]",
            o = lateout(reg) out,
            p = in(reg) p,
            /* WRONG: asm reads memory, but nomem claims it does not */
```

```
            options(nomem, nostack)
        );
    }
    out
}
```

Correct example: do not claim `nomem`. If you truly only read memory, `readonly` is appropriate.

```
use core::arch::asm;

#[no_mangle]
pub extern "C" fn good_modeled_load(p: *const u64) -> u64 {
    let out: u64;
    unsafe {
        asm!(
            "mov {o}, QWORD PTR [{p}]",
            o = lateout(reg) out,
            p = in(reg) p,
            options(nostack, readonly)
        );
    }
    out
}
```

28.4.3 Reordering mistake 2: Assuming `volatile` implies a full memory barrier

`volatile` prevents deletion/merging of the asm block, but it does not automatically imply the strongest ordering constraints across threads or devices.

If you need ordering for shared-memory concurrency, use the language's atomic fences (and only use CPU fences when you truly need hardware ordering beyond the language fence).

Rust pattern:

```
use core::arch::asm;
use core::sync::atomic::{compiler_fence, fence, Ordering};

#[no_mangle]
pub extern "C" fn publish_then_notify(flag: *mut u32, data: *mut u32, v: u32) {
    unsafe {
        /* Write data */
        core::ptr::write_volatile(data, v);

        /* Prevent compiler reordering around the publication point */
        compiler_fence(Ordering::Release);

        /* Optionally: hardware fence if required by your protocol */
        fence(Ordering::Release);

        /* Notify via MMIO-like write using inline asm */
        asm!(
            "mov DWORD PTR [{p}], {x:e}",
            p = in(reg) flag,
            x = in(reg) 1u32,
            options(nostack, volatile)
        );
    }
}
```

Engineering rule:

- Use `compiler_fence` to stop compiler reordering.
- Use `fence` for cross-core ordering semantics.
- Use inline asm fences (`mfence/lfence/sfence`) only when you have a precise hardware-level requirement.

28.4.4 Reordering mistake 3: Incorrect use of `pure` and `readonly`

`pure` allows the compiler to treat asm like a referentially transparent computation. This is only valid when:

- outputs depend only on inputs and allowed memory reads,
- there are no hidden side effects,
- there is no dependence on time, IO, device state, or flags beyond modeled outputs.

Broken example: reading a timestamp counter and claiming `pure` is wrong because it changes every time.

```
use core::arch::asm;

#[no_mangle]
pub extern "C" fn broken_rdtsc_pure() -> u64 {
    let lo: u32;
    let hi: u32;
    unsafe {
        asm!(
            "rdtsc",
            out("eax") lo,
            out("edx") hi,
            /* WRONG: rdtsc is not pure */
            options(nomem, nostack, pure, readonly)
        );
    }
    ((hi as u64) << 32) | (lo as u64)
}
```

Correct example: keep it `volatile` and avoid `pure`.

```
use core::arch::asm;

#[no_mangle]
pub extern "C" fn good_rdtsc() -> u64 {
    let lo: u32;
    let hi: u32;
    unsafe {
        asm!(
            "rdtsc",
            out("eax") lo,
            out("edx") hi,
            options(nomem, nostack, volatile)
        );
    }
    ((hi as u64) << 32) | (lo as u64)
}
```

28.5 Constraint Errors

28.5.1 The core idea

Constraints are where most inline asm breaks silently. A constraint error means:

- the compiler put your operand in a register/class/width you did not intend,
- or you forced a register that violates the ABI/optimizer assumptions,
- or you expressed an incorrect input/output dependency, allowing the compiler to reuse registers unsafely.

28.5.2 Constraint error 1: Wrong width (32-bit vs 64-bit) on x86-64

On x86-64, writing to a 32-bit register zero-extends into the full 64-bit register. If you unintentionally use 32-bit operands, you can destroy upper bits of pointers or values. Broken example: accidentally truncating a pointer-like value.

```
use core::arch::asm;

#[no_mangle]
pub extern "C" fn broken_truncate(mut x: u64) -> u64 {
    unsafe {
        /* WRONG: using {x:e} forces 32-bit subregister, zero-extending into 64-bit */
        asm!("add {x:e}, 1", x = inout(reg) x, options(nomem, nostack));
    }
    x
}
```

Correct example: use 64-bit form.

```
use core::arch::asm;

#[no_mangle]
pub extern "C" fn good_add64(mut x: u64) -> u64 {
    unsafe {
        asm!("add {x}, 1", x = inout(reg) x, options(nomem, nostack));
    }
    x
}
```

28.5.3 Constraint error 2: Using `out` instead of `lateout` when overlap is possible

If an output register can overlap an input register, the compiler may allocate the same register for both. If your asm writes the output early, it can destroy the input before it is used.

Broken example:

```
use core::arch::asm;

#[no_mangle]
pub extern "C" fn broken_overlap(a: u64, b: u64) -> u64 {
    let out: u64;
    unsafe {
        asm!(
            "mov {o}, {a}",
            "add {o}, {b}",
            a = in(reg) a,
            b = in(reg) b,
            o = out(reg) out, /* can overlap with a or b */
            options(nomem, nostack)
        );
    }
    out
}
```

Correct example: use `lateout` or explicitly model `inout`.

```
use core::arch::asm;

#[no_mangle]
pub extern "C" fn good_overlap(a: u64, b: u64) -> u64 {
    let out: u64;
    unsafe {
        asm!(
            "mov {o}, {a}",
            "add {o}, {b}",
            a = in(reg) a,
            b = in(reg) b,
            o = lateout(reg) out,
```

```
            options(nomem, nostack)
        );
    }
    out
}
```

28.5.4 Constraint error 3: Forgetting that some instructions require specific registers

Some instructions implicitly use fixed registers (e.g., division uses `RDX:RAX` as numerator). If you do not declare those registers correctly, the compiler may place live values there and your asm will clobber them.

Correct division example (unsigned) using explicit registers:

```
use core::arch::asm;

#[no_mangle]
pub extern "C" fn udiv64(n: u64, d: u64) -> u64 {
    let q: u64;
    unsafe {
        asm!(
            /* Dividend in RDX:RAX. For 64-bit unsigned div, clear RDX. */
            "xor edx, edx",
            "div {div}",
            div = in(reg) d,
            in("rax") n,
            lateout("rax") q,
            /* RDX is clobbered by div (remainder), so declare it. */
            lateout("rdx") _,
            options(nomem, nostack)
        );
    }
```

```
    q
}
```

Common incorrect variants:

- Not clearing RDX for unsigned division.
- Not declaring RDX as clobbered/output.
- Not pinning the dividend to RAX.

28.5.5 Constraint error 4: Confusing pointer operands and memory operands

If you pass a pointer in a register and then dereference it inside the template, you must model memory effects correctly (do not lie with nomem). Also, you must avoid creating "hidden" memory dependencies.
Correct atomic-like load with explicit modeling (still not a full atomic by itself):

```
use core::arch::asm;

#[no_mangle]
pub extern "C" fn load_u32_relaxed(p: *const u32) -> u32 {
    let out: u32;
    unsafe {
        asm!(
            "mov {o:e}, DWORD PTR [{p}]",
            o = lateout(reg) out,
            p = in(reg) p,
            options(nostack, readonly)
        );
    }
    out
}
```

28.5.6 Constraint error 5: Mixing Windows x64 ABI rules with fixed registers

On Windows x64, the first integer argument is `RCX`. If you force inputs into `RCX/RDX/R8/R9` without reason, you reduce allocator freedom and increase the chance of conflicts. Prefer `in(reg)` unless:

- the instruction requires a fixed register,
- you are implementing a specific ABI boundary intentionally.

Example: a simple add that *does not* need fixed registers.

```
use core::arch::asm;

#[no_mangle]
pub extern "C" fn add_generic(a: u64, b: u64) -> u64 {
    let mut x = a;
    unsafe {
        asm!(
            "add {x}, {b}",
            x = inout(reg) x,
            b = in(reg) b,
            options(nomem, nostack)
        );
    }
    x
}
```

28.6 GAS Intel Syntax: Inline-Asm Thinking Without Inline-Asm Risk

Many “inline asm” use-cases on Windows are safer as out-of-line assembly in a `.S` file, because you can:

- write a correct Windows x64 prolog/epilog,
- reserve shadow space correctly for any calls,
- preserve non-volatile registers explicitly,
- avoid lying to the compiler about side effects.

A safe out-of-line function template (Windows x64) in GAS Intel syntax:

```
.intel_syntax noprefix
.text

/* Windows x64: RCX, RDX, R8, R9 are arg regs.
   Non-volatile regs: RBX, RBP, RSI, RDI, R12-R15, XMM6-XMM15.
   Caller reserves 32-byte shadow space before calls; callee does not. */

.global add4_win64
add4_win64:
    /* a in ecx, b in edx, c in r8d, d in r9d */
    mov eax, ecx
    add eax, edx
    add eax, r8d
    add eax, r9d
    ret
```

Rust declaration for calling it:

```
extern "C" {
    fn add4_win64(a: i32, b: i32, c: i32, d: i32) -> i32;
}

#[no_mangle]
pub extern "C" fn call_add4_win64(a: i32, b: i32, c: i32, d: i32) -> i32 {
    unsafe { add4_win64(a, b, c, d) }
}
```

28.7 Practical Rules (Academy Checklist)

28.7.1 Clobbers

- Every register you write must be an `out`/`lateout`/`inout` operand or an explicitly fixed output.
- Never claim `preserves_flags` unless you are certain no instruction changes flags.
- Never claim `nomem` if you touch memory. Never claim `readonly` if you write memory.
- Never claim `nostack` if you push/pop, call, or otherwise use the stack.

28.7.2 Compiler reordering

- If the asm has side effects, use `volatile`.
- If you need ordering, use `compiler_fence`/`fence` in Rust; do not assume `volatile` is a full barrier.
- Do not use `pure` unless the asm is truly referentially transparent under the language model.

28.7.3 Constraint discipline

- Prefer `in(reg)` over fixed registers unless required.
- Use `lateout` when overlap with inputs is possible.
- Be precise about widths: 32-bit subregister writes zero-extend on x86-64.
- For instructions with implicit registers (`mul`/`div`, some string ops), declare those registers explicitly.
- Keep inline asm blocks as small as possible; one conceptual operation per block.

28.8 Summary

Inline assembly bugs are usually not “wrong instructions”. They are wrong **contracts**:

- wrong clobber modeling,
- wrong memory/ordering modeling,
- wrong constraints (widths, overlaps, implicit registers).

If you cannot express the full contract accurately, the correct engineering move on Windows is often:

- write the routine in a dedicated `.S` file with a real ABI-correct prolog/epilog,
- expose a small `extern "C"` boundary to Rust,
- and keep inline asm only for tiny, strictly modeled instruction sequences.

Undefined Behavior at the Boundary

29.1 What "Undefined Behavior at the Boundary" Means

At the boundary between a high-level language and assembly, you stop being protected by:

- the compiler's internal correctness assumptions,
- the language's memory model,
- the ABI's hidden invariants (stack alignment, shadow space, preserved registers),
- and the type system's lifetime/aliasing rules.

A boundary bug is often not "wrong assembly". It is a mismatch in **contracts**:

- The compiler assumes one set of invariants.
- Your inline/out-of-line assembly assumes another.
- The ABI enforces a third.

When these disagree, the optimizer is allowed to do transformations that appear "impossible" to a low-level engineer reading machine code linearly. That is the practical meaning of undefined behavior at the boundary: **the compiler is no longer obliged to preserve the behavior you thought you wrote**.
This chapter focuses on two sections:

- **C/C++ + asm traps** (as engineering patterns that still apply when your boundary partner is any C-ABI consumer).
- **Lifetime violations** (the most common UB class in FFI + assembly work, especially in Rust).

All examples are written for **Rust on Windows** and **GAS Intel syntax**.

29.2 Boundary Contracts You Must Treat as Invariants

29.2.1 Windows x64 ABI invariants

When your boundary is callable via the Windows C ABI (Rust `extern "C"`), these rules matter immediately:

- Integer/pointer args 1–4: `RCX, RDX, R8, R9`.
- Floating args 1–4: `XMM0–XMM3`.
- Stack must be **16-byte aligned at the CALL instruction**.
- Caller must reserve **32 bytes of shadow space** before calling any function.
- Callee-saved (non-volatile) regs: `RBX, RBP, RSI, RDI, R12–R15` and `XMM6–XMM15`.

Violating these does not merely "crash sometimes". It can turn safe code into UB because the compiler-generated code assumes the ABI contract is honored.

29.2.2 Rust FFI invariants

Rust gives you raw power with `unsafe`, but the cost is that you must uphold:

- **Validity**: pointers must be valid for the accessed type, alignment, and size.
- **Aliasing**: creating references implies exclusivity and non-aliasing rules that the optimizer relies on.
- **Initialization**: reading uninitialized bytes as a typed value is UB.
- **Lifetimes**: pointers/references must outlive their use on the other side.
- **Unwinding**: panics must not cross into foreign code unless you explicitly define and support that model.

29.3 C/C++ + asm Traps

This section describes classic boundary traps traditionally seen in C/C++ with inline/out-of-line assembly. The same traps apply to any C ABI boundary, including Rust `extern "C"` exports called from assembly, injectors, or foreign runtimes.

29.3.1 Trap 1: The compiler assumes more than the CPU requires

A CPU may happily load misaligned data, but the compiler may assume alignment and generate aligned SIMD moves. A CPU may allow aliasing, but the compiler may assume strict aliasing rules and reorder loads/stores accordingly. A CPU may preserve some registers across a call only by convention, but the compiler treats it as a guarantee.
Boundary UB occurs when your assembly invalidates compiler assumptions.

29.3.2 Trap 2: Missing ABI obligations in assembly callers (shadow space + alignment)

A correct Windows x64 assembly caller to a Rust `extern "C"` function must reserve shadow space and preserve alignment.

Rust function to be called

```
#[no_mangle]
pub extern "C" fn add4(a: i32, b: i32, c: i32, d: i32) -> i32 {
    a + b + c + d
}
```

Correct Windows x64 caller (GAS Intel syntax)

```
.intel_syntax noprefix
.text
.extern add4
.global call_add4_ok

call_add4_ok:
    push rbp
    mov rbp, rsp

    /* After push rbp, RSP is misaligned by 8.
       Reserve 32 bytes shadow space + 8 bytes align fix = 40. */
    sub rsp, 40

    mov ecx, 1
    mov edx, 2
    mov r8d, 3
    mov r9d, 4
    call add4

    add rsp, 40
    pop rbp
    ret
```

Broken caller (missing shadow space and unknown alignment)

```
.intel_syntax noprefix
.text
.extern add4
.global call_add4_broken

call_add4_broken:
    /* WRONG: does not reserve 32-byte shadow space.
       WRONG: does not ensure stack is aligned at CALL. */
    mov ecx, 1
    mov edx, 2
    mov r8d, 3
    mov r9d, 4
    call add4
    ret
```

Why this becomes UB in practice:

- The callee may store argument registers into its home area (shadow space) and overwrite your stack data.
- The callee may assume alignment and use aligned XMM stores to stack, faulting or corrupting memory.
- Optimizations change prolog behavior; your program "works" until it doesn't.

29.3.3 Trap 3: Flags, registers, and memory clobbers not modeled at the boundary

Out-of-line assembly can clobber non-volatile registers; inline assembly can clobber flags or memory. If the compiler believes they are preserved, it may keep live values in them and reorder code.

Even without inline asm, the trap still exists if:

- you call a hand-written asm function that violates non-volatile register preservation,
- or you hook/patch code with a trampoline that fails to preserve state.

Broken callee: clobbers a non-volatile register (RBX) and returns

```
.intel_syntax noprefix
.text
.global clobber_rbx_bad
clobber_rbx_bad:
    /* WRONG: RBX is non-volatile (callee-saved) on Windows x64 */
    mov rbx, 0x1122334455667788
    ret
```

Correct callee: preserves RBX

```
.intel_syntax noprefix
.text
.global clobber_rbx_ok
clobber_rbx_ok:
    push rbx
    mov rbx, 0x1122334455667788
    pop rbx
    ret
```

Rust caller that can be miscompiled if the callee breaks the contract

```
extern "C" {
    fn clobber_rbx_bad();
    fn clobber_rbx_ok();
}
```

```
#[no_mangle]
pub extern "C" fn use_after_call(flag: i32, p: *const u8) -> u8 {
    /* In optimized builds, the compiler may keep p or derived values in non-volatile regs.
    ↪ */
    unsafe {
        if flag != 0 { clobber_rbx_bad(); } else { clobber_rbx_ok(); }
    }
    unsafe { *p }
}
```

When the ABI is violated, the compiler's assumptions are invalid, and the behavior is not constrained.

29.3.4 Trap 4: Layout and alignment mismatches across the boundary

If one side assumes a struct layout that the other side does not guarantee, field accesses become UB.
In Rust, always use `repr(C)` for any FFI-visible struct.

FFI-safe struct

```
#[repr(C)]
pub struct Header {
    pub len: u32,
    pub flags: u16,
    pub kind: u16,
}
```

A boundary trap appears when a foreign consumer uses packing or different alignment assumptions. The correct engineering response is to redesign the boundary to avoid implicit layout dependency:

- pass pointers + explicit sizes,

- provide accessor functions,
- avoid packed layouts for public ABI unless all accesses are explicitly unaligned-safe.

29.3.5 Trap 5: Uninitialized and “trap” representations

At boundaries, it is common to:

- allocate raw bytes,
- cast them to a typed pointer,
- and read a typed value.

If the bytes are uninitialized or not a valid representation of the type, reading them is UB.
Safe pattern: use `MaybeUninit` and initialize before reading.

```
use core::mem::MaybeUninit;

#[no_mangle]
pub extern "C" fn init_then_read() -> u64 {
    let mut x = MaybeUninit::<u64>::uninit();
    unsafe {
        x.as_mut_ptr().write(0xAABBCCDDEEFF0011u64);
        x.assume_init()
    }
}
```

29.3.6 Trap 6: Unwinding across the boundary

Letting a Rust panic cross into assembly or foreign code is a boundary UB trap in real systems.
Engineering rule:

- Do not unwind across FFI boundaries.

- Catch panics inside Rust and convert to error codes, or configure panic abort for boundary modules.

Example wrapper that prevents unwinding from escaping:

```
use core::panic::{self, AssertUnwindSafe};

#[no_mangle]
pub extern "C" fn boundary_entry(x: i32) -> i32 {
    let r = panic::catch_unwind(AssertUnwindSafe(|| {
        if x == 0 { panic!("bad"); }
        x + 1
    }));
    match r {
        Ok(v) => v,
        Err(_) => -1, /* error code */
    }
}
```

29.4 Lifetime Violations

Lifetime violations are the most common UB at the boundary because lifetimes are not a CPU concept. The CPU only sees addresses; the language and optimizer attach meaning to *whether those addresses remain valid and exclusive over time*.

29.4.1 Violation 1: Returning a pointer to a stack local (dangling pointer)

This is a classic boundary failure: the address becomes invalid immediately after return.

Broken Rust export

```
#[no_mangle]
```

```rust
pub extern "C" fn get_ptr_to_stack_bad() -> *const u8 {
    let local = [1u8, 2, 3, 4];
    local.as_ptr() /* UB: returns pointer to stack memory that will go out of scope */
}
```

Correct pattern: heap allocation with explicit free

Design the ABI to make ownership explicit.

```rust
use core::ptr;

#[no_mangle]
pub extern "C" fn alloc_buffer(len: usize) -> *mut u8 {
    if len == 0 { return ptr::null_mut(); }
    let mut v = Vec::<u8>::with_capacity(len);
    v.resize(len, 0);
    let p = v.as_mut_ptr();
    core::mem::forget(v);
    p
}

#[no_mangle]
pub extern "C" fn free_buffer(p: *mut u8, len: usize) {
    if p.is_null() || len == 0 { return; }
    unsafe {
        /* Reconstruct the Vec so Rust frees it with the same allocator. */
        let _ = Vec::from_raw_parts(p, len, len);
    }
}
```

Boundary rule:

- The side that allocates must also define the deallocator.
- Never free memory in a different runtime/allocator domain.

29.4.2 Violation 2: Storing a borrowed pointer beyond its valid lifetime

A frequent bug pattern:

- A boundary function receives a pointer (or reference-like input).
- It stores it in a global/static for later use.
- The caller frees or invalidates it.
- Later, the program uses the stored pointer: use-after-free.

Broken pattern: global raw pointer with no ownership

```
use core::sync::atomic::{AtomicPtr, Ordering};

static STORED: AtomicPtr<u8> = AtomicPtr::new(core::ptr::null_mut());

#[no_mangle]
pub extern "C" fn store_ptr_bad(p: *mut u8) {
    /* UB risk: no lifetime/ownership contract. */
    STORED.store(p, Ordering::Relaxed);
}

#[no_mangle]
pub extern "C" fn use_stored_bad() -> u8 {
    let p = STORED.load(Ordering::Relaxed);
    if p.is_null() { return 0; }
    unsafe { *p } /* may be dangling */
}
```

Correct engineering response: redesign the ABI to store **handles** or enforce explicit ownership transfer.

Correct pattern: store owned heap allocation via handle

```
use core::sync::atomic::{AtomicU64, Ordering};

static NEXT_ID: AtomicU64 = AtomicU64::new(1);

/* Very small demo handle table (fixed size) */
const MAX: usize = 64;
static mut TABLE: [*mut u8; MAX] = [core::ptr::null_mut(); MAX];

#[no_mangle]
pub extern "C" fn create_owned_byte(v: u8) -> u64 {
    let id = NEXT_ID.fetch_add(1, Ordering::Relaxed);
    let slot = (id as usize) % MAX;

    let b = Box::new(v);
    let p = Box::into_raw(b) as *mut u8;

    unsafe { TABLE[slot] = p; }
    id
}

#[no_mangle]
pub extern "C" fn read_owned_byte(id: u64) -> u8 {
    let slot = (id as usize) % MAX;
    let p = unsafe { TABLE[slot] };
    if p.is_null() { return 0; }
    unsafe { *p }
}

#[no_mangle]
pub extern "C" fn destroy_owned_byte(id: u64) {
    let slot = (id as usize) % MAX;
```

```
    let p = unsafe { TABLE[slot] };
    if p.is_null() { return; }
    unsafe {
        TABLE[slot] = core::ptr::null_mut();
        let _ = Box::from_raw(p as *mut u8);
    }
}
```

This is not a production table, but it demonstrates the principle:

- do not store borrowed pointers without lifetime enforcement,
- store owned objects or use explicit copy-in/copy-out designs,
- use handles when foreign code cannot model lifetimes safely.

29.4.3 Violation 3: Passing Rust references across the boundary

A Rust reference (`&T` or `&mut T`) carries strong guarantees:

- `&T` implies the data is valid and properly aligned for `T`.
- `&mut T` implies unique access (no aliases) for the duration of the borrow.

Foreign code (including assembly) cannot enforce these guarantees. Therefore:

- Do not expose Rust references in FFI function signatures.
- Use raw pointers (`*const T`, `*mut T`) and document the contract explicitly.

Correct boundary signature pattern:

```
#[no_mangle]
pub extern "C" fn inc_u64(p: *mut u64) -> i32 {
    if p.is_null() { return -1; }
```

```
    unsafe {
        *p = (*p).wrapping_add(1);
    }
    0
}
```

29.4.4 Violation 4: Creating slices from raw parts with wrong length

A common boundary pattern is to accept `(ptr, len)` and treat it as a slice.
This becomes UB if:

- the pointer is null with nonzero length,
- the pointer is not valid for `len` elements,
- the memory overlaps in illegal ways for `&mut [T]`,
- the lifetime does not cover the slice usage.

Safe-by-contract pattern with explicit checks:

```
#[no_mangle]
pub extern "C" fn sum_bytes(ptr: *const u8, len: usize) -> u64 {
    if ptr.is_null() && len != 0 { return 0; }
    let mut s: u64 = 0;
    unsafe {
        let slice = core::slice::from_raw_parts(ptr, len);
        for &b in slice {
            s += b as u64;
        }
    }
    s
}
```

29.4.5 Violation 5: Callback lifetimes and re-entrancy

Boundary UB often appears when foreign code provides a callback pointer and context pointer. Traps:

- callback outlives captured context,
- callback is called from a different thread than expected,
- callback re-enters the caller and violates aliasing expectations,
- callback panics across the boundary.

Robust boundary design:

- callback type is `extern "C" fn(ctx: *mut core::ffi::c_void, ...)`.
- context ownership is explicit (owned handle or borrowed with documented lifetime).
- panics are contained and converted to error codes.

Example pattern:

```
use core::ffi::c_void;

pub type Callback = extern "C" fn(ctx: *mut c_void, value: u32) -> i32;

#[no_mangle]
pub extern "C" fn run_callback(cb: Option<Callback>, ctx: *mut c_void) -> i32 {
    let cb = match cb {
        Some(f) => f,
        None => return -1,
    };
    /* The contract must specify ctx validity and thread rules. */
    cb(ctx, 123)
}
```

29.5 Boundary-Safe Design Patterns

29.5.1 Pattern 1: Use a C-style ABI surface even if your internals are advanced

Expose only:

- functions,
- fixed-width integers,
- pointers,
- explicit lengths,
- explicit ownership functions.

29.5.2 Pattern 2: Make ownership explicit

Preferred ABI shapes:

- `create() -> handle`, `destroy(handle)`.
- `alloc(len) -> ptr`, `free(ptr, len)`.
- `process(ptr, len) -> status`.

Avoid:

- returning borrowed pointers to temporaries,
- accepting references,
- sharing allocator ownership implicitly.

29.5.3 Pattern 3: Treat alignment and layout as contracts, not assumptions

- `repr(C)` for structs.
- avoid packed public ABI unless every access is unaligned-safe.
- if alignment is required for performance (SIMD), provide both aligned and unaligned entry points.

29.5.4 Pattern 4: Contain unwinding

- catch panics in boundary functions, return error codes.
- do not unwind across assembly/foreign frames.

29.6 Summary

Undefined behavior at the boundary is an engineering failure mode where:

- ABI invariants (alignment, shadow space, preserved registers) are violated by assembly or hooks,
- layout/alignment assumptions differ across modules,
- uninitialized or invalid representations are read as typed values,
- references/borrows are smuggled across FFI and the optimizer exploits their guarantees,
- lifetimes are violated by returning stack pointers, storing borrowed pointers, or misusing callbacks,
- unwinding crosses boundaries and corrupts control flow and state.

The boundary is where high-level intent meets low-level reality. The only stable solution is to design the boundary as a strict contract:

- explicit ownership,
- explicit sizes,
- explicit layout,
- explicit alignment,
- explicit unwind policy,
- and ABI-correct assembly templates on Windows x64.

Part VI

PART VI — TOOLING AND DEBUGGING

Reading Compiler-Generated Assembly

30.1 Why Reading Compiler Assembly Is a Core Skill

Compiler-generated assembly is the most honest representation of what your program *will actually do* on the CPU, under a specific toolchain, target, ABI, and optimization level. The goal of this chapter is not to memorize instruction sequences, but to learn how to:

- recognize recurring patterns quickly,
- separate *logic* from *ABI and runtime scaffolding*,
- identify optimization artifacts that mislead readers,
- and build a reliable mental mapping between source constructs and low-level effects.

All examples are written for **Rust on Windows x86-64** and shown in **Intel syntax** assembly (GAS style). The exact instruction choices vary by compiler version, target CPU features, LTO settings, and codegen flags; the *patterns* remain stable.

30.2 Toolchain Setup: Producing Assembly You Can Study

The first mistake is studying assembly without controlling the conditions that produce it. You need reproducible builds with explicit flags.

30.2.1 Emit assembly from Rust (Windows)

```
# Debug-like codegen with full symbols (less optimization)
rustc -C opt-level=0 -C debuginfo=2 -C target-cpu=x86-64 `
  --emit=asm -o out_O0.s main.rs

# Release-like codegen (aggressive optimization)
rustc -C opt-level=3 -C debuginfo=2 -C target-cpu=x86-64 `
  --emit=asm -o out_O3.s main.rs

# LTO can radically change code shape (cross-crate inlining, dedup)
rustc -C opt-level=3 -C lto -C debuginfo=2 -C target-cpu=x86-64 `
  --emit=asm -o out_O3_lto.s main.rs

# Force a specific code model choice: disable frame pointer omission (helps reading)
rustc -C opt-level=3 -C force-frame-pointers=yes -C debuginfo=2 `
  --emit=asm -o out_fp.s main.rs
```

Engineering notes:

- Compare `opt-level=0` vs `opt-level=3`. Most "weird" assembly is optimization scaffolding.
- Keep `debuginfo=2` while learning; it makes correlating source $\leftrightarrow$ assembly much easier.
- Use `force-frame-pointers=yes` when doing stack reasoning; optimized builds often omit them.

30.2.2 Decide what assembly you want

You typically need three views:

- **compiler .s output**: easiest to read, includes labels and (often) directives.

- **disassembly of the final binary**: shows reality after linking, ICF, hot/cold splitting, thunks.
- **annotated assembly with source mapping**: best for education and debugging.

30.3 Recognizing Patterns

This section builds a pattern vocabulary. When you can name a pattern, you can stop reading it as "mystery instructions" and treat it as a known block.

30.3.1 Pattern 1: Windows x64 calling convention

On Windows x64, integer/pointer arguments arrive in `RCX, RDX, R8, R9`. Return value usually in `RAX`. The caller reserves 32 bytes of shadow space before a call, and the stack must be 16-byte aligned at the `call` instruction.
A typical tiny leaf function compiled from Rust:

```
#[no_mangle]
pub extern "C" fn add3(a: i64, b: i64, c: i64) -> i64 {
    a + b + c
}
```

A typical code shape (leaf, no stack use needed) may look like:

```
.intel_syntax noprefix
.text
add3:
    /* a in rcx, b in rdx, c in r8 */
    lea rax, [rcx + rdx]
    add rax, r8
    ret
```

What to recognize:

- No prolog/epilog: leaf function, no spills, no local stack.
- Uses LEA for arithmetic: common optimization artifact (LEA is a fast add-like instruction).
- Registers correspond to Windows x64 ABI arg registers.

30.3.2 Pattern 2: Prolog/epilog (stack frame setup)

When a function needs stack locals, spills, or calls other functions, you will usually see a frame setup.
Typical shape:

```
.intel_syntax noprefix
.text
fn_has_frame:
    push rbp
    mov rbp, rsp
    sub rsp, 64              /* local space, alignment decisions happen here */
    /* ... body ... */
    add rsp, 64
    pop rbp
    ret
```

What to recognize:

- `push rbp; mov rbp, rsp` establishes a base pointer (if not omitted).
- `sub rsp, N` allocates locals and spill slots.
- In optimized builds, `rbp` may disappear and offsets are from `rsp` directly.

30.3.3 Pattern 3: Non-volatile register preservation

On Windows x64, registers like `RBX, RSI, RDI, R12–R15` are callee-saved. If the compiler uses them, it must save/restore them.
Typical pattern:

```
.intel_syntax noprefix
.text
fn_uses_rbx:
    push rbx
    sub rsp, 32
    /* ... rbx used as long-lived value ... */
    add rsp, 32
    pop rbx
    ret
```

What to recognize:

- If you see `push rbx` and later `pop rbx`, that is ABI compliance.
- Missing restore is not an optimization; it is a bug (usually only in hand-written asm).

30.3.4 Pattern 4: Stack alignment fix-ups

Because calls require alignment, compilers often choose allocation sizes that preserve 16-byte alignment at call boundaries. You may see apparently odd values like 40, 72, 104.
Recognize that:

- The number is not arbitrary: it is "locals + spills + alignment correction".
- On Windows x64, call sites additionally assume the caller provided shadow space, but the compiler still arranges local stack correctly.

30.3.5 Pattern 5: Branchless conditionals (cmov, setcc, sbb trick)

Source:

```
#[no_mangle]
pub extern "C" fn max_u64(a: u64, b: u64) -> u64 {
    if a > b { a } else { b }
}
```

Typical optimized pattern:

```
.intel_syntax noprefix
.text
max_u64:
    mov rax, rcx
    cmp rax, rdx
    cmovb rax, rdx      /* if a < b, move b */
    ret
```

What to recognize:

- cmov removes a branch (avoids branch misprediction).
- Condition codes may invert depending on operand order; do not memorize one form.

30.3.6 Pattern 6: Bounds check + panic path

Rust inserts bounds checks in safe indexing unless it can prove safety.

Source:

```
#[no_mangle]
pub extern "C" fn get_i32(a: *const i32, len: usize, idx: usize) -> i32 {
    if a.is_null() { return 0; }
    unsafe {
```

```
        let s = core::slice::from_raw_parts(a, len);
        s[idx]
    }
}
```

Typical shape:

- compare `idx` with `len`,
- branch to cold panic/abort path on failure,
- compute address and load value on success.

Representative assembly outline:

```
.intel_syntax noprefix
.text
get_i32:
    test rcx, rcx
    je .ret0

    /* idx in r8, len in rdx, ptr in rcx (example mapping may vary by signature order) */
    cmp r8, rdx
    jae .panic_oob

    mov eax, DWORD PTR [rcx + r8*4]
    ret

.ret0:
    xor eax, eax
    ret

.panic_oob:
    /* cold path: calls panic handler or abort */
```

```
    /* ... */
    int3   /* placeholder: real code will call into runtime */
```

What to recognize:

- The “panic path” is a separate block, often placed in a cold section and called rarely.
- In `opt-level=3`, the hot path becomes very tight; the cold path can look enormous.

30.4 Optimization Artifacts

Optimization artifacts are transformations that preserve semantics but distort the intuitive relationship between source and assembly. If you do not recognize them, you misdiagnose correct code as wrong.

30.4.1 Artifact 1: Inlining and the disappearance of function boundaries

Inlining replaces a call with the callee body. Consequences:

- Functions may vanish entirely from the assembly output.
- A “single function” in assembly may contain logic from many source functions.
- Debug stepping becomes confusing unless you use debuginfo and proper disassembly views.

Source:

```
#[inline(always)]
fn square(x: u64) -> u64 { x * x }

#[no_mangle]
pub extern "C" fn f(x: u64) -> u64 {
```

```
    square(x) + 1
}
```

Typical optimized shape:

```
.intel_syntax noprefix
.text
f:
    mov rax, rcx
    imul rax, rax
    add rax, 1
    ret
```

Artifact recognition:

- No call instruction. The helper function is gone.
- `#[inline(always)]` increases the likelihood but optimization may inline anyway.

30.4.2 Artifact 2: Dead code elimination and “impossible” missing work

If an operation has no observable effect, it can be removed.
Source:

```
#[no_mangle]
pub extern "C" fn do_nothing(mut x: u64) -> u64 {
    x ^= 0x1234;
    x ^= 0x1234;
    x
}
```

Optimized result may reduce to:

```
.intel_syntax noprefix
.text
do_nothing:
    mov rax, rcx
    ret
```

Recognize:

- Two XORs cancel. The compiler collapses them.
- Do not assume missing instructions means the compiler “ignored” your code; it proved equivalence.

30.4.3 Artifact 3: Strength reduction and LEA-based arithmetic

Compilers often replace multiplications or address computations with shifts and LEA.
Source:

```
#[no_mangle]
pub extern "C" fn mul_by_9(x: u64) -> u64 {
    x * 9
}
```

Typical pattern:

```
.intel_syntax noprefix
.text
mul_by_9:
    lea rax, [rcx + rcx*8]
    ret
```

Recognize:

- `lea` is used as arithmetic, not only address calculation.
- This is an optimization, not a sign of pointer usage.

30.4.4 Artifact 4: Loop transformations (unrolling, vectorization, and peeled iterations)

Loops may be:

- unrolled (multiple iterations per loop body),
- vectorized (SIMD registers and wide loads/stores),
- peeled (a few initial iterations handled separately to align pointers),
- or reduced to a library call (memcpy/memset-like patterns).

Source:

```
#[no_mangle]
pub extern "C" fn sum_u8(ptr: *const u8, len: usize) -> u64 {
    if ptr.is_null() { return 0; }
    let mut s: u64 = 0;
    unsafe {
        for i in 0..len {
            s += *ptr.add(i) as u64;
        }
    }
    s
}
```

Optimized assembly may show:

- a scalar loop for small `len`,
- a vectorized loop for large `len`,
- tail handling for remaining bytes.

Recognizing vectorization:

- XMM/YMM registers used heavily.
- instructions like `pxor`, `punpck`, `pmaddubsw`, `psadbw`, or sequences that widen bytes to words/dwords.
- alignment checks and dual paths (aligned vs unaligned loads).

30.4.5 Artifact 5: Cold/hot splitting and "huge" error blocks

Rust error paths (panic, allocation failure) often appear as cold blocks. The hot path becomes minimal, and the cold path becomes very large.
Recognize:

- cold labels (often grouped away from the hot path),
- calls to runtime helpers,
- setup of panic messages, formatting, and metadata.

When reading assembly for performance, you usually focus on:

- the hot path instructions,
- branch predictability,
- memory access patterns,
- and whether bounds checks or aliasing prevent vectorization.

30.4.6 Artifact 6: Stack spills and register allocation illusions

In optimized builds, the compiler assigns values to registers aggressively. When registers run out, it spills to the stack.
What looks like “unnecessary loads/stores” may be required due to:

- register pressure,
- ABI-mandated call-clobbered registers,
- or scheduling constraints.

Recognize spill slots:

- repeated `mov [rsp+offset], reg` and `mov reg, [rsp+offset]` patterns,
- offsets that match the function’s stack allocation size.

30.4.7 Artifact 7: Security and platform scaffolding

Depending on build settings and environment, you may see additional scaffolding:

- stack probes for large stack allocations,
- control-flow protection related sequences,
- unwind metadata-related patterns (not always visible in plain disassembly),
- debug checks or instrumentation if enabled.

Engineering approach:

- Learn to identify and ignore scaffolding blocks when your goal is logic tracing.
- Learn to identify and respect scaffolding when your goal is ABI correctness and exploit resistance.

30.5 A Practical Reading Workflow (Academy Method)

This workflow reduces misinterpretation.

30.5.1 Step 1: Determine the compilation regime

Record:

- optimization level,
- LTO status,
- debug info level,
- target CPU and enabled features,
- panic strategy (unwind vs abort),
- and whether you are reading .s or final binary.

30.5.2 Step 2: Locate the true entry and the true hot path

In optimized builds:

- the “function” may begin with a small dispatch to hot/cold blocks,
- the body may be split into multiple basic blocks,
- inlined code may introduce multiple logically separate regions.

Identify:

- the blocks executed for common inputs,
- the blocks executed only for errors.

30.5.3 Step 3: Identify ABI scaffolding first

Before reading logic, recognize and mentally remove:

- prolog/epilog,
- non-volatile saves/restores,
- stack allocation and alignment decisions,
- call-site preparation and shadow space concerns.

30.5.4 Step 4: Map registers to source-level meaning

Write a small table:

- which register holds which argument,
- which register holds loop indices,
- which register holds pointers and base addresses,
- which stack slots represent spills or locals.

30.5.5 Step 5: Translate basic blocks, not lines

Assembly must be understood as control-flow graphs:

- conditional branches connect blocks,
- fallthrough indicates likely hot paths,
- indirect jumps often indicate jump tables or virtual dispatch.

30.5.6 Step 6: Only then interpret optimizations

Once you know the logic, interpret artifacts:

- why bounds checks stayed or disappeared,
- why vectorization happened or failed,
- why a loop is split into multiple loops,
- why a call was inlined or preserved.

30.6 Extended Examples: From Source to "Typical" Assembly Shapes

These examples are designed to teach recognition rather than to match a single compiler build byte-for-byte.

30.6.1 Example A: Copy loop that may become memset/memcpy-like

```
#[no_mangle]
pub extern "C" fn fill(ptr: *mut u8, len: usize, v: u8) {
    if ptr.is_null() { return; }
    unsafe {
        for i in 0..len {
            *ptr.add(i) = v;
        }
    }
}
```

Typical optimized shapes you may see:

- a small scalar loop for tiny `len`,
- a call or inlined sequence that resembles a bulk fill,
- vectorized stores with tail handling.

Recognition cues:

- repeated stores with increasing pointer or index,
- alignment checks and two store loops,
- tail loop for the last 1–15 bytes.

30.6.2 Example B: Checked arithmetic vs wrapping arithmetic

```
#[no_mangle]
pub extern "C" fn add_checked(a: u64, b: u64) -> u64 {
    match a.checked_add(b) {
        Some(x) => x,
        None => 0,
    }
}

#[no_mangle]
pub extern "C" fn add_wrapping(a: u64, b: u64) -> u64 {
    a.wrapping_add(b)
}
```

Typical recognition:

- `add_wrapping` is usually a single `add`.
- `add_checked` often uses: `add` + flag test (`jc` / `setc`) or `sbb` trick, then a conditional move/select to produce 0 on overflow.

Representative outline:

```
.intel_syntax noprefix
.text
add_wrapping:
    lea rax, [rcx + rdx]
    ret

add_checked:
    mov rax, rcx
    add rax, rdx
    jc .overflow
    ret
.overflow:
    xor eax, eax
    ret
```

30.6.3 Example C: Match and jump tables

```
#[no_mangle]
pub extern "C" fn classify(x: u32) -> u32 {
    match x {
        0 => 10,
        1 => 11,
        2 => 12,
        3 => 13,
        _ => 99,
    }
}
```

Recognition:

- small ranges often compile to a chain of compares,

- denser ranges compile to a jump table (indirect jump),
- or to arithmetic + table lookup.

30.7 Common Misreads (And How to Avoid Them)

30.7.1 Misread 1: "The compiler added useless work"

Often that "work" is:

- ABI compliance (saving non-volatile regs),
- stack alignment maintenance,
- bounds checks for safety,
- cold error paths that almost never run.

30.7.2 Misread 2: "This LEA means pointers"

LEA is frequently arithmetic. Check whether the value is later used as an address or as a scalar.

30.7.3 Misread 3: "The loop vanished, so it doesn't execute"

The loop may have been:

- unrolled,
- vectorized,
- replaced by a bulk operation,
- or proven unnecessary and removed.

30.7.4 Misread 4: “This function is wrong because it has no stack frame”

Leaf functions commonly have no frame. Lack of `rbp` is normal in optimized builds.

30.8 Summary

To read compiler-generated assembly reliably, you must learn two complementary skills:

30.8.1 Recognizing patterns

- ABI register usage and return conventions (Windows x64).
- Prolog/epilog, non-volatile saves/restores, alignment fix-ups.
- Branchless conditionals, bounds checks, panic/cold paths.
- Spills and stack slots as register-pressure artifacts.

30.8.2 Recognizing optimization artifacts

- Inlining (missing calls, merged logic).
- Dead code elimination (missing operations that cancel or have no effect).
- Strength reduction (LEA, shifts, address-mode arithmetic).
- Loop transformations (unrolling, vectorization, peeled iterations, tail loops).
- Hot/cold splitting (small hot path, huge cold error blocks).

The core discipline is this:

- First identify ABI and scaffolding.

- Then reconstruct control flow at the basic-block level.
- Only then interpret the optimizer's transformations.

GDB and LLDB Essentials

31.1 Purpose of a Debugger in Assembly-Level Engineering

Debuggers like **GDB** and **LLDB** are essential for observing the actual state of a running program. When working with compiler-generated assembly or hand-written assembly, you must inspect:

- The contents of CPU registers at breakpoints and faults.
- The call stack and how the stack pointer and frame pointer evolve.
- Memory contents at specific addresses.
- Execution control via breakpoints, stepping, and conditional stops.
- How high-level semantics map to low-level state transitions.

Although both GDB and LLDB support multiple platforms, on Windows most workflows involve either:

- Using the GCC toolchain with `gdb` (MinGW/MSYS2), or
- Using `lldb` with MSVC toolchains via LLVM/Clang.

This chapter focuses on patterns and commands useful for debugging Rust-produced binaries on Windows from a command-line terminal.

31.2 Registers: Viewing and Interpreting

Registers are the core state of a CPU. On x86-64 Windows:

- **General-purpose registers**: `RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8–R15`.
- **Instruction pointer**: `RIP`.
- **Flags register**: `RFLAGS`.
- **Floating/SIMD registers**: `XMM0–XMM15`, `YMM/ZMM` if AVX/AVX-512 enabled.

A key skill is to differentiate between:

- argument registers (Windows x64 ABI: `RCX, RDX, R8, R9`),
- return value register (`RAX`),
- volatile (caller-saved) vs non-volatile (callee-saved) registers.

31.2.1 Inspecting Registers in GDB

Start your debug session by loading the compiled binary:

```
gdb ./my_program.exe
```

Common register inspection commands:

```
(gdb) info registers
/* Displays all integer and pointer registers at once, including RIP and RSP */
(gdb) info registers rax rbx rcx
/* Displays only specific registers of interest */
(gdb) x/4gx $rsp
/* Dump 4 8-byte values from the stack pointer */
```

Example when stopped at a breakpoint:

```
Breakpoint 1, main () at main.rs:10
10        let x = compute(5);
(gdb) info registers
RAX            0x0                 0
RBX            0x0                 0
RCX            0x5                 5
RDX            0x0                 0
RSP            0x000000000012fef0  0x12fef0
RIP            0x00007ff6abcd1234  0x7ff6abcd1234 <compute+4>
...
```

Interpreting the snapshot:

- `RIP` shows where execution is paused.
- `RSP` shows the current stack pointer.
- `RCX`, `RDX`, etc., contain argument values according to the Windows x64 ABI.

31.2.2 Inspecting Registers in LLDB

Launch:

```
lldb ./my_program.exe
```

Break and inspect:

```
(lldb) breakpoint set --name main
(lldb) run
(lldb) register read
/* Shows all registers */
(lldb) register read rax rcx
/* Shows specific registers */
```

LLDB uses the same register naming as GDB; on x86-64 the register set is identical for the architectures relevant to this booklet.

31.3 Stack Inspection

Understanding the stack is critical for both:

- verifying the ABI (correct shadow space and alignment),
- diagnosing stack corruption,
- unwinding stack traces,
- confirming local variable locations and spill slots.

31.3.1 Viewing Stack Contents

A typical stack memory dump uses the debugger's memory examine command.
In GDB:

```
(gdb) x/16gx $rsp
/* Shows 16 8-byte words starting at RSP */
(gdb) x/8gx $rbp-32
/* Shows stack frame around RBP */
```

In LLDB:

```
(lldb) memory read --format hex --count 16 --size 8 $rsp
```

Interpreting the stack:

- The first value at `RSP` usually corresponds to the saved return address after a call.

- If a frame pointer (`RBP`) is present, `[RBP]` often holds the previous `RBP`.
- Between `RBP-...` and `RSP+...` you find local variables and spill slots allocated by the compiler.

Example at a breakpoint:

```
(gdb) info registers rbp rsp
RBP 0x000000000012ff00
RSP 0x000000000012fef0

(gdb) x/8gx $rbp-16
0x12fef0: 0x0000000000000005 0x0000000000000000
0x12fef0+0x10: 0x00007ff6abcd1234 0x0000000000000010
/* First QWORD is local variable, second is saved RBP, third is return RIP */
```

31.3.2 Backtrace and Frame Navigation

Stack inspection is often achieved through backtraces.
GDB:

```
(gdb) backtrace
#0 compute (x=5) at main.rs:25
#1 main () at main.rs:10
#2 0x00007ff6abcd0000 in __scrt_common_main_seh ()
```

LLDB:

```
(lldb) thread backtrace
```

Backtrace interpretation:

- Each frame shows the function and line of source that corresponds to the saved instruction pointer.
- Registers and stack contents can be examined independently per frame.

31.4 Breakpoints: Setting, Using, and Controlling Execution

Breakpoints halt execution when a condition is met and allow inspection of state.

31.4.1 Basic Breakpoints

In GDB:

```
(gdb) break main
/* Sets a breakpoint at the entry of main */
(gdb) break compute
/* Breaks at all compute function entries */
(gdb) break *0x7ff6abcd1234
/* Break at a specific address */
```

In LLDB:

```
(lldb) breakpoint set --name main
(lldb) breakpoint set --address 0x7ff6abcd1234
```

31.4.2 Conditional Breakpoints

You often want to break only when specific conditions hold.
GDB:

```
(gdb) break compute if x == 5
/* Only break when register/variable x equals 5 */
```

LLDB:

```
(lldb) breakpoint set --name compute --condition "x == 5"
```

Conditions can refer to:

- local variables when source-level debuginfo is present,
- registers directly (e.g., `$rcx == 5`),
- memory contents at addresses (e.g., `*(int*)0x12345678 == 0`).

31.4.3 Stepping and Continuing Execution

Once at a breakpoint:
GDB:

```
(gdb) step
/* Step into functions, obeys source line association */
(gdb) next
/* Step over function calls */
(gdb) continue
/* Continue until next breakpoint */
```

LLDB uses similar commands:

```
(lldb) step
(lldb) next
(lldb) process continue
```

31.4.4 Watching Memory and Registers

You can set watch expressions for memory or registers:
GDB:

```
(gdb) watch *ptr
/* Break when the memory pointed to by ptr changes */
(gdb) watch $rax
/* Break if RAX changes */
```

LLDB:

```
(lldb) watchpoint set expression -- *ptr
```

31.4.5 Breakpoints in Assembly-Level Debugging

When source-level symbols are sparse (e.g., optimized builds), break at specific addresses:
GDB:

```
(gdb) break *0x7ff6abcd1234
```

LLDB:

```
(lldb) breakpoint set --address 0x7ff6abcd1234
```

Use `disassemble` to find the address of interest:
GDB:

```
(gdb) disassemble compute
```

LLDB:

```
(lldb) disassemble --name compute
```

31.5 Debugger Commands for Stack and Register Control

31.5.1 Examining a Frame in Detail

In GDB:

```
(gdb) frame 1
/* Select frame 1 for focused inspection */
(gdb) info locals
/* Show local variables if available */
(gdb) info args
/* Show arguments for the frame */
```

LLDB:

```
(lldb) frame select 1
(lldb) frame info
(lldb) frame variable
```

31.5.2 Setting the Instruction Pointer

On rare occasions (hot patching, rerunning logic), you may set RIP:
GDB:

```
(gdb) set $rip = 0x7ff6abcd1280
```

LLDB:

```
(lldb) register write rip 0x7ff6abcd1280
```

This forces the next instruction fetched from the new address.

31.5.3 Examining Floating and SIMD Registers

In GDB:

```
(gdb) info registers xmm0 xmm1 xmm2
```

LLDB:

```
(lldb) register read xmm0 xmm1 xmm2
```

For AVX/AVX-512 wide registers (YMM/ZMM), symbol support depends on the debugger build.

31.6 Range-Based and Pattern-Based Breakpoints

Breakpoints can be placed over a range of addresses or by name patterns.
GDB:

```
(gdb) rbreak compute
/* Break on any function matching regex "compute" */
```

LLDB:

```
(lldb) breakpoint set --name "compute" --regex
```

31.7 Debugging Rust Binaries Without Source (Assembly Focus)

Sometimes you must debug optimized Rust binaries where symbols are stripped or missing. The strategy:

- Use `disassemble` to explore code regions.
- Use `info registers` and `memory read` to inspect state.
- Set breakpoints at suspect instruction addresses.
- Step over and into loops manually, watching registers.

Assembly context commands:
GDB:

```
(gdb) disassemble /r $rip-32, $rip+32
/* Show 32 bytes before/after RIP */
```

LLDB:

```
(lldb) disassemble --pc-range $rip-32 $rip+32
```

31.8 Summary

Understanding GDB and LLDB essentials is foundational for:

- inspecting **register state** that reflects ABI, arguments, and computed results,
- performing **stack inspection** to trace calls, frames, and spill slots,
- using **breakpoints** to control execution and isolate failure points.

The academy workflow emphasizes:

- producing reproducible builds with controlled optimization,
- correlating source (when available) to machine state,
- inspecting registers and stack contents regularly,
- and using conditional and watch breakpoints to observe subtle state changes.

Mastering these tools dissolves the barrier between source-level intent and the actual machine behavior you are building your assembly foundations upon.

WinDbg Essentials

32.1 Why WinDbg Matters for Modern Assembly Work

WinDbg is the Windows-native debugger designed to diagnose real failures in real binaries: access violations, stack corruption, calling-convention mistakes, SEH/unwind issues, and optimizer-shaped artifacts that are hard to interpret in source-level debuggers.
For low-level engineers, WinDbg is most valuable when:

- you must trust the platform's symbol model (PDBs, public symbols, private symbols),
- you must interpret stacks produced by Windows x64 unwinding rules,
- you must reason about register state and memory at the exact fault site,
- you must debug optimized code paths where source-level stepping becomes misleading.

This chapter focuses on two essential skill sets:

- **Stack analysis**: getting from a crash to a correct call chain and the true fault mechanism.
- **Symbol awareness**: making WinDbg load the right symbols, diagnosing symbol mismatches, and understanding what symbols can and cannot tell you.

All examples are written for **Rust on Windows x86-64** and typical Windows user-mode debugging.

32.2 Minimal Rust Setup for WinDbg-Friendly Builds

WinDbg is only as good as your symbols. For Rust on Windows, you typically want:

- a `.pdb` generated for your executable and relevant crates,
- stable function names for exported entry points when needed (`#[no_mangle]` + `extern "C"`),
- enough debuginfo to map frames and local variables (when possible).

32.2.1 Cargo profiles for debuginfo on Windows

```
# Cargo.toml

[profile.dev]
debug = 2

[profile.release]
debug = 1
# For deeper stack and variable info in release debugging, use:
# debug = 2
```

32.2.2 Build commands

```
# Debug build with PDB
cargo build

# Release build with PDB (if profile.release.debug is nonzero)
cargo build --release
```

32.2.3 A crash target to practice on

Use a controlled crash so you can practice stack and symbol workflows without guessing.

```
#[no_mangle]
pub extern "C" fn crash_null_deref(p: *const u64) -> u64 {
    unsafe { *p }
}

fn main() {
    let p: *const u64 = core::ptr::null();
    let _ = crash_null_deref(p);
}
```

32.3 Symbol Awareness

Symbol awareness is the discipline of always knowing:

- *what symbols are loaded*,
- *where they were loaded from*,
- *whether they match the exact binary you are debugging*,
- *what level of detail they contain* (public vs private, types vs no types).

32.3.1 The symbol path model in WinDbg

WinDbg searches for symbols in a symbol path. In modern Windows debugging, you typically combine:

- a local cache directory,
- Microsoft's public symbol server (for OS and system DLL public symbols),

- your build output directories containing your private PDBs.

32.3.2 Essential symbol commands

```
.symfix
/* Sets a reasonable default symbol path (commonly includes the Microsoft symbol server). */

.sympath
/* Display current symbol path. */

.sympath+ C:\path\to\my\pdbs
/* Append a directory containing your private symbols. */

.reload
/* Reload symbols for modules already loaded. */

.reload /f
/* Force-reload symbols (useful when symbols are wrong or partially loaded). */

!sym noisy
/* Turn on symbol loading diagnostics (critical for troubleshooting). */

!sym quiet
/* Turn off noisy diagnostics when done. */
```

32.3.3 Module listing and symbol status

You must be able to confirm which modules are loaded and whether symbols are loaded for each one.

```
lm
/* List loaded modules. */
```

```
lmv
/* Verbose module listing; shows symbol status and paths. */

lm m my_program
/* Filter modules by name pattern. */
```

What to look for in practice:

- Your executable module should show private symbols loaded (your PDB), not just exports.
- System DLLs will often have public symbols (enough for stack names but not full private type data).
- If a module says symbols are deferred or mismatched, your stack trace quality will degrade sharply.

32.3.4 Common symbol failure patterns (and how to recognize them)

Pattern A: No symbols loaded

Symptoms:

- stack frames show only raw addresses,
- `ln` (list nearest symbol) fails to resolve meaningful names,
- breakpoints by function name do not bind.

Response workflow:

```
!sym noisy
.sympath
.symfix
.sympath+ C:\path\to\my\pdbs
.reload /f
lmv m my_program
```

Pattern B: Wrong symbols (mismatched PDB)

Symptoms:

- function names look wrong or inconsistent with your build,
- source mapping points to unrelated files/lines,
- local variables appear nonsensical, or frames do not unwind correctly.

Response workflow:

```
!sym noisy
.reload /f my_program.exe
lmv m my_program
/* Verify the loaded PDB matches the exact build output you are debugging. */
```

Pattern C: Public symbols only

This is normal for many Windows system binaries. Expect:

- function names for stack traces,
- but limited type and local-variable inspection.

For your own binaries, insist on private symbols (your PDB).

32.3.5 Symbol-based navigation commands

Once symbols are correct, these become your primary navigation tools:

```
x my_program!*crash*
/* Search for symbols matching a wildcard pattern. */

ln @rip
```

```
/* Show nearest symbol to the instruction pointer. */

uf my_program!crash_null_deref
/* Unassemble a function by name (requires symbols). */
```

32.4 Stack Analysis

Stack analysis is not just printing a backtrace. It is the skill of determining:

- the **true faulting context**,
- whether the stack is **trustworthy** (not corrupted),
- where **unwinding** failed (if it did),
- and how ABI, calling convention, and optimization shaped the frames you see.

32.4.1 Start from the fault context

In crashes, always anchor your analysis at the exception context record.

```
!analyze -v
/* High-level analysis; often identifies access violations, stack corruption, and likely
↪  faulting module. */

.ecxr
/* Switch to the exception context record (registers at the fault site). */

r
/* Show registers in the current context. */
```

Typical workflow for a null dereference (practice target above):

```
!analyze -v
.ecxr
r
ln @rip
u @rip-20 L40
```

32.4.2 Registers relevant to stack reading on Windows x64

The minimum set you read constantly:

- `RIP`: current instruction pointer (faulting instruction).
- `RSP`: stack pointer (top of stack, return addresses, spills).
- `RBP`: frame pointer (may be omitted in optimized code).
- `RCX, RDX, R8, R9`: integer/pointer argument registers.
- `RAX`: common return-value register and scratch.

Register inspection:

```
r
r rip rsp rbp rcx rdx r8 r9 rax
```

32.4.3 Basic stack traces: k, kb, kp, kv

WinDbg provides several stack trace formats. Your goal is to choose the one that gives you the most signal.

```
k
/* Basic stack trace. */

kb
```

```
/* Stack trace with parameters (when available). */

kp
/* Stack trace with parameters and more detail; often useful for x64. */

kv
/* Verbose stack trace; may include frame pointers, return addresses, and more metadata. */
```

Interpretation discipline:

- If frames look nonsensical (repeating addresses, impossible modules, abrupt stops), suspect stack corruption or missing unwind info.
- If symbols are missing, k output may be almost useless until you fix symbols.

32.4.4 Stack memory inspection: proving whether the stack is sane

When you do not trust unwinding, you must look at raw stack memory.

Dump stack pointers and return addresses

```
r rsp
dps @rsp L40
/* Display pointers from RSP; often reveals return-address chain and saved values. */

dq @rsp L40
/* Dump 8-byte values; raw view, less interpretation than dps. */
```

Heuristics:

- Return addresses usually point into executable code regions (module text sections).
- If you see many values that point into non-executable memory or random heap regions, stack may be corrupted.

- If you see plausible code pointers but frames still do not unwind, you may have missing unwind metadata or nonstandard prolog/epilog (common in hand-written assembly).

Mapping return addresses to symbols

```
ln <address>
/* Resolve an address to the nearest symbol+offset. */

ln poi(@rsp)
/* Resolve the pointer stored at the top of the stack. */
```

32.4.5 Disassembly at the fault site

When reading compiler-generated assembly, you must confirm what the faulting instruction actually did.

```
u @rip-20 L40
/* Disassemble around the instruction pointer. */

ub @rip
/* Disassemble backwards from RIP (useful when the crash is at a branch target). */

uf my_program!crash_null_deref
/* Disassemble the whole function when symbols are available. */
```

32.4.6 A concrete crash walk-through (null dereference)

Assume you launched WinDbg and hit an access violation in `crash_null_deref`.
Core workflow:

```
!analyze -v
.ecxr
r
```

```
ln @rip
kp
u @rip-20 L40
```

What you prove:

- `RIP` is inside your function and points to a load instruction.
- The address being dereferenced is `0` (or near `0`).
- The stack shows the call chain: `main` → `crash_null_deref`.

32.4.7 Diagnosing stack corruption versus missing unwind info

These two failure classes produce similar symptoms (bad backtraces), but they have different fingerprints.

Stack corruption fingerprints

- `!analyze -v` mentions stack corruption, or the stack trace stops abruptly.
- `dps @rsp` shows return addresses that do not map into code sections.
- `u` at suspected return addresses disassembles garbage or faults.

Missing/unsupported unwind fingerprints

- Return addresses look plausible in `dps`, but `k` cannot unwind reliably.
- This occurs with hand-written assembly lacking correct Windows x64 unwind metadata.
- Optimized code without frame pointers can be harder to unwind if unwind info is missing or damaged.

Practical response:

- For hand-written assembly intended to be unwound correctly on Windows x64, you must follow Windows x64 unwind requirements (beyond the scope of this chapter).
- For investigation, rely more on raw stack inspection (`dps`, `ln`) and disassembly around call sites.

32.5 Symbol Awareness for Rust-Specific Realities

Rust introduces patterns that affect what you see in WinDbg:

32.5.1 Name mangling and exported boundaries

If you want stable names visible to WinDbg without relying on Rust-mangled internal symbols, use explicit exports for boundary entry points:

```
#[no_mangle]
pub extern "C" fn api_entry(x: u64) -> u64 {
    x.wrapping_mul(3).wrapping_add(1)
}
```

Then you can search and disassemble easily:

```
x my_program!api_entry*
uf my_program!api_entry
```

32.5.2 Optimization effects on local variables

Even with good PDBs:

- optimized builds may keep variables in registers, eliminate them, or merge lifetimes,

- some variables will appear as optimized-out or unavailable,
- stack frames may omit `RBP` and use `RSP`-relative addressing aggressively.

Correct mindset:

- Trust registers and memory at the current context more than source-level locals in heavy optimization.
- Use disassembly and data-flow reasoning when variables are optimized away.

32.6 A Minimal WinDbg Command Card (Stack + Symbols)

Keep this short card near your keyboard.

32.6.1 Symbols

```
!sym noisy
.symfix
.sympath
.sympath+ C:\path\to\my\pdbs
.reload /f
lmv m my_program
```

32.6.2 Crash anchor + stack

```
!analyze -v
.ecxr
r
kp
dps @rsp L40
ln @rip
u @rip-20 L40
```

32.7 Summary

WinDbg becomes powerful for assembly-level engineering once you master two things:

32.7.1 Stack analysis

- Anchor on the exception context (`.ecxr`), read registers (`r`), then read the stack (`k`, `kp`, `dps`).
- Use raw stack inspection to validate whether unwinding is trustworthy.
- Use disassembly around `RIP` to prove what instruction faulted and what address it touched.

32.7.2 Symbol awareness

- Treat symbols as a first-class dependency: verify them before trusting any stack trace.
- Use `!sym noisy`, `.symfix`, `.sympath`, and `.reload /f` as your standard recovery workflow.
- For Rust, export stable boundary entry points when you want predictable symbol names for debugging and integration.

With these essentials, you can reliably move from a Windows crash to a verified fault mechanism and a correct low-level explanation—even in optimized binaries.

Crash Analysis Workflow

33.1 Introduction: What Is Crash Analysis?

Crash analysis is the disciplined methodology for turning a program failure (access violation, stack corruption, illegal instruction, divide-by-zero, assertion abort) into a reproducible explanation of root cause. At low levels—especially with Rust, C/C++, and assembly—crashes often manifest far from their true origin due to optimizer transformations, calling convention violations, stack misalignment, and undefined behavior. This chapter codifies a reliable workflow that professional reverse engineers, OS support teams, and systems programmers use to analyze crashes end-to-end on Windows.
Two key themes:

- **Access Violation Analysis**: how to interpret and diagnose a memory access fault based on registers, addresses, instruction context, and memory state.
- **Stack Trace Reliability**: how to assess whether a stack backtrace can be trusted, how to validate it, and how to recover meaningful call chains even when unwind info is missing or stack is corrupted.

All examples assume:

- Windows x86-64 targets,

- Rust binaries (but the workflow applies equally to C/C++),
- debugging in WinDbg (but many concepts translate to GDB/LLDB),
- standard Windows calling convention (`RCX, RDX, R8, R9` argument registers, 32-byte shadow space, 16-byte stack alignment).

33.2 Access Violation Analysis

An access violation occurs when the CPU attempts to read from or write to a memory address that is invalid, protected, unmapped, or otherwise non-dereferenceable under the process's current memory permissions. In Windows, this is reported as an SEH exception code `0xC0000005`.

33.2.1 Immediate Diagnostics Upon Crash

Start your debugger (WinDbg) and stop at the crash. First commands:

```
!analyze -v
.ecxr
r
```

- `!analyze -v` attempts a high-level cause classification (access violation, integer divide, stack pivot, heap corruption, etc.).
- `.ecxr` loads the exception context—registers and state at the instruction that faulted.
- `r` prints register contents at the crash time.

A typical access violation context might reveal:

```
ExceptionAddress: 00007ff7abcd1234
AccessViolationException
  InvalidReadAddress: 0000000000000000
RIP: 00007ff7abcd1234
rax=0000000000000000 rcx=0000000000000000 ...
```

This tells you:

- `RIP` is the faulting instruction address.
- The violation was an invalid read at address `0x0` (null dereference).
- Registers at fault can show which pointer was used.

33.2.2 Disassemble Around the Fault

Now inspect the exact instruction that caused the fault:

```
u @rip-16 L64
```

You are looking for instructions like:

```
mov rax, qword ptr [rcx]
```

If `rcx` is `0x0`, this matches the fault address. Other common forms:

- `mov rdx, [rax+rcx*8]` — index computation fault.
- `stosd / movsxd / lea` — illegal arithmetic or misalignment.
- `call [reg+offset]` — indirect call through invalid pointer.

Match the effective address of the memory operand:

```
EffectiveAddress = BaseReg + IndexReg*Scale + Displacement
```

Compute it using register inspection to correlate with the reported invalid access.

33.2.3 Common Root Causes for Access Violations

- **Null or small pointer dereference**: pointer was never initialized or set to null as an error sentinel but not checked.
- **Use-after-free**: pointer points to memory already freed (often high, nonzero invalid addresses).
- **Buffer overflow/underflow**: pointer walks past allocation bounds.
- **Calling convention violation**: stack misalignment or missing shadow space causes data to be interpreted as pointers.
- **Unwind/corruption**: stack or heap corruption caused an earlier write to trash pointers or saved return addresses.
- **Invalid arithmetic or cast traps**: mixing signed/unsigned or 32-bit truncation of a 64-bit pointer yielding invalid memory access.

33.2.4 Memory Inspection at Fault Site

Once you have the register that holds the pointer, inspect memory around the computed address:

```
db <pointerAddr> L32
dq <pointerAddr> L8
```

If the pointer is near a known region (heap, stack, static), use:

```
!address <pointerAddr>
!heap -p -a <pointerAddr>
```

These commands tell you whether the pointer was:

- in a reserved/uncommitted region,

- inside a freed heap block,
- inside a guard page,
- or near a stack region.

33.2.5 Correlation with Source Code

If you have symbols, you can locate the source file and line corresponding to the faulting instruction:

```
ln @rip
~* e nt!ntdll!DbgBreakPoint
!dlls
```

WinDbg uses PDB information to map instruction pointers to source files. Your goal is to identify the exact source line and the high-level pointer variable that became invalid.

33.3 Stack Trace Reliability

Stack traces show the chain of calls that led to the fault. Reliability ranges from high (when unwind metadata is present) to low (when stack is corrupted or unwind metadata is missing).

33.3.1 Unwind Metadata and ABI Compliance

On Windows x64, unwind metadata is generated by the compiler to allow stack walking without frame pointers. If a function is hand-written in assembly without proper unwind metadata, debuggers may not walk past it correctly.
A reliable stack trace requires:

- valid unwind info for each frame,

- stack memory that has not been corrupted,
- return addresses stored on the stack following calling convention conventions.

33.3.2 Getting a Stack Trace in WinDbg

Use:

```
kp
kv
k
kb
```

- `kp` prints parameter registers and return addresses.
- `kv` shows verbose information, including frame numbers.
- `k` is the basic call stack.
- `kb` shows parameters in stacked order.

Example:

```
# ChildEBP RetAddr
00 000000000012FEE0 00007ff7abcd1a90 my_program!crash_null_deref+0x10
01 000000000012FEF0 00007ff7abcd0c37 my_program!main+0x27
02 00007ffeed8901d2 ntdll!RtlUserThreadStart+0x32
```

Interpretation:

- Frame 0 is the crash site.
- Frame 1 is the caller (in this case, `main`).
- Frame 2 is system thread startup.

33.3.3 When Stack Trace Is Not Reliable

A backtrace may be unreliable when:

- Stack memory is corrupted (return addresses have been overwritten).
- Hand-written assembly functions do not have unwind metadata (no frame pointer, no unwind descriptors).
- Gradient optimizations (inlining, tail calls) transformed or eliminated frames.
- Stack pivoting or exception dispatch changed the expected return address layout.

Symptoms of an unreliable stack trace:

- repeating return addresses,
- return addresses that do not map into known modules,
- abrupt termination of walking before reaching the expected entry point.

In such cases, you must fall back on:

- raw stack inspection (`dps @rsp LXX`),
- manual return address resolution (`ln <address>`),
- and disassembly of possible call sites.

33.3.4 Validating Stack Trace Frames

For each frame in the trace:

- Use `ln <returnAddress>` to confirm it maps to a valid function plus offset.

- Confirm that the return address is within the executable section of the expected module (check with `lmv`).
- Inspect the assembly around the address immediately before the return to confirm the calling convention formed a proper `call` instruction target.

For example:

```
ln 00007ff7abcd1a90
u 00007ff7abcd1a70 L20
```

Disassembly around that return address should show a `call` to the next frame's function.

33.3.5 Handling Missing Unwind Info (Assembly Frames)

If you encounter an assembly function without unwind metadata, the debugger cannot trustfully walk past it. A practical approach:

- Identify the assembly function name via symbols or disassembly.
- Use `dps @rsp` and `ln` to manually resolve likely return addresses on the stack.
- Confirm manually whether those return addresses are plausible.
- Construct a manual backtrace by repeating this process outward from the fault frame.

33.4 Crash Analysis Case Study: Rust Null Dereference

Consider the Rust code:

```rust
#[no_mangle]
pub extern "C" fn crash_null_deref(p: *const u64) -> u64 {
    unsafe { *p }
```

```
}

fn main() {
    let p: *const u64 = core::ptr::null();
    let _ = crash_null_deref(p);
}
```

Assume a release build with PDBs. You launch WinDbg and run until crash.

33.4.1 Step 1: Exception Context

```
!analyze -v
.ecxr
r
```

Registers likely show:

- `rcx == 0` (first argument register holds null pointer),
- `RIP` at the dereference instruction.

33.4.2 Step 2: Disassemble Faulting Instruction

```
u @rip-16 L32
```

Expect:

```
mov rax, qword ptr [rcx]  /* this is the fault: rcx is 0x0 */
```

33.4.3 Step 3: Stack Backtrace

```
kp
```

Expect:

```
crash_null_deref+0x10
main+0x27
RtlUserThreadStart+0x32
```

Verify each return address with `ln`.

33.4.4 Step 4: Correlate With Source

Once you have:

- the faulting instruction,
- the register values,
- a plausible call chain,

you now map it directly to the source line that dereferenced `p`. This confirms the root cause: null pointer dereference.

33.5 Summary: A Repeatable Crash Workflow

Professional crash analysis follows this disciplined path:

33.5.1 Access Violation Analysis

- Stop at the exception context with reliable registers and exception info.
- Disassemble around `RIP` to find the exact memory operation.
- Compute the effective address and correlate with register state.
- Inspect memory permissions and region information for the accessed address.
- Correlate with source when symbols are available.

33.5.2 Stack Trace Reliability

- Attempt a full backtrace using unwind metadata (`kp, kv`).
- Validate return addresses map to known code regions.
- Confirm each frame's caller sequence is ABI-compliant.
- When unwind info is missing, fall back on manual stack inspection (`dps @rsp`) and manual return address resolution (`ln <addr>`).

This workflow isolates:

- whether the violation resulted from a bad pointer, undefined behavior, or calling convention violation,
- whether the stack trace is reliable,
- and how to map machine state back to the original source semantics.

Applied consistently, this methodology turns crashes into precise explanations rather than guesses or incidental observations, fulfilling the academy standard for low-level engineering diagnostics.

Part VII

PRACTICAL CROSS-ARCHITECTURE EXAMPLES

Hello World — Linux x86-64

This chapter builds a precise, repeatable baseline for writing and running the smallest useful program on **Linux x86-64**: printing `"Hello, world"` and exiting. We will do it in two ways:

- **Direct system calls** (no C runtime, no libc): you provide the entry point and call the kernel yourself.
- **Via libc** (normal user-space): you implement `main` and let the runtime handle process startup.

All assembly is **GAS (GNU assembler) with Intel syntax**.

34.1 Why this example matters

A "Hello World" in assembly is not about printing text. It is about learning a concrete set of foundations:

- how an ELF64 process begins (`_start` vs. `main`),
- how Linux x86-64 syscalls are invoked,
- which registers carry arguments and results,
- how sections (`.text`, `.rodata`) and symbols work,
- how to assemble, link, and inspect the resulting binary.

34.2 Minimal prerequisites

Assumptions

- Target: Linux x86-64 (ELF64).
- Assembler: `as` (GAS) or `gcc` driving GAS.
- Linker: `ld` (direct) or `gcc` (recommended for convenience).

Notes on Intel syntax in GAS

- Use `.intel_syntax noprefix` to write Intel-style operands without `%` register prefixes.
- GAS comments:
  - `/* ... */` works well inside Intel syntax.
  - # is also a GAS comment, but you requested `/* ... */`.

34.3 Linux x86-64 system call ABI (the essentials)

On Linux x86-64, a syscall is performed using the `syscall` instruction. Registers are assigned as follows:

Purpose	Register
Syscall number	`RAX`
Argument 1	`RDI`
Argument 2	`RSI`
Argument 3	`RDX`
Argument 4	`R10`
Argument 5	`R8`
Argument 6	`R9`
Return value	`RAX`

Important: The `syscall` instruction clobbers `RCX` and `R11` (the CPU and kernel use them internally). Treat them as destroyed across syscalls.

We will use two syscalls

- `write(fd, buf, len)`: syscall number `1`
- `exit(status)`: syscall number `60`

34.4 Hello World (no libc): pure syscalls

Source file: `hello_syscall.S`

```
.intel_syntax noprefix

/* ------------------------------------------------------------------
   hello_syscall.S  (Linux x86-64, ELF64)
   Entry point: _start
```

```
   Prints "Hello, world\n" using write(1, msg, len) then exits with status 0
   ------------------------------------------------------------------------ */

.section .rodata
msg:
    .ascii "Hello, world\n"
len = . - msg

.section .text
.global _start
_start:
    /* write(1, msg, len) */
    mov     rax, 1          /* SYS_write */
    mov     rdi, 1          /* fd = STDOUT */
    lea     rsi, [rip + msg]/* buf = &msg (RIP-relative for PIE-friendly code) */
    mov     rdx, len        /* count = len */
    syscall

    /* exit(0) */
    mov     rax, 60         /* SYS_exit */
    xor     rdi, rdi        /* status = 0 */
    syscall
```

Build and run

You can assemble and link in two reliable ways.

Option A (recommended): use gcc as a driver

```
gcc -nostdlib -no-pie hello_syscall.S -o hello_syscall
```

```
./hello_syscall
```

Option B: call `as` and `ld` directly

```
as --64 hello_syscall.S -o hello_syscall.o
ld -o hello_syscall hello_syscall.o
./hello_syscall
```

Why `-nostdlib`? Because we do not want the C runtime startup objects or libc; we provide `_start` ourselves.

34.5 Hello World (with libc): normal `main`

In the "normal" model, you implement `main`, and the runtime provides `_start`, initializes the process, and then calls your `main`.

Source file: `hello_printf.S`

```
.intel_syntax noprefix

/* ------------------------------------------------------------------------
   hello_printf.S (Linux x86-64, ELF64)
   Entry point: main (provided by us), _start (provided by runtime)
   Calls libc printf.
   ------------------------------------------------------------------------ */

.section .rodata
fmt:
    .asciz "Hello, world (libc)!\n"
```

```
.section .text
.global main
.extern printf
main:
    /* System V ABI note:
       - Before calling a function, RSP must be 16-byte aligned.
       - On entry to main, the runtime typically ensures alignment.
       - We still avoid unnecessary stack use to keep it simple.
    */

    lea     rdi, [rip + fmt]    /* 1st arg: const char* fmt */
    xor     eax, eax            /* for variadic calls: number of vector registers
    ↪  used = 0 */
    call    printf

    xor     eax, eax            /* return 0 */
    ret
```

Build and run

```
gcc hello_printf.S -o hello_printf
./hello_printf
```

Why `xor eax, eax` before `printf`? Because `printf` is variadic, and the System V ABI requires `AL` to indicate how many vector registers are used for passing floating-point arguments. Here we pass none, so `AL = 0`.

34.6 A precise comparison: syscall vs libc

Topic	Syscall-only	libc-based
Startup symbol	`_start`	`main` (runtime provides `_start`)
Dependencies	none (kernel only)	libc, loader, startup objects
Portability (within Linux)	stable syscall ABI	stable API/ABI via libc
Binary size	minimal	larger
Control	maximum	high, but runtime-managed

34.7 Inspection and verification (engineering habits)

Verify the entry point

```
readelf -h hello_syscall | grep 'Entry point'
```

List symbols

```
nm -n hello_syscall | head
```

Disassemble

```
objdump -d -Mintel hello_syscall
```

Note: `-Mintel` makes objdump print Intel-style disassembly, which matches our source style.

34.8 Cross-architecture takeaways (kept small, but comparable)

The following is the minimal mental template you should carry across architectures:

Template

- **Define the ABI boundary:** how does user code call the OS (syscall instruction, syscall number, arg registers)?
- **Define the program boundary:** what is the entry point (`_start`, `main`) and who sets it up?
- **Define the binary format:** ELF64 vs PE/COFF affects linking, relocation, and startup.

Minimal syscall triple (conceptually consistent)

- stdout handle / fd
- pointer to bytes
- byte count

This same triple appears everywhere; only the calling convention changes.

34.9 Practical pitfalls (what breaks beginners)

Pitfall 1: wrong syscall convention

Using `int 0x80` on x86-64 is not the normal ABI. Use `syscall`.

Pitfall 2: wrong register for argument 4

On Linux x86-64 syscalls, argument 4 is `R10` (not `RCX`).

Pitfall 3: forgetting `RCX` and `R11` clobber

Treat them as destroyed across syscalls.

Pitfall 4: alignment rules when calling libc

When calling normal functions under the System V ABI, keep the stack 16-byte aligned at the call boundary.

34.10 Windows-focused Rust examples (x86-64) — pure Rust, no C runtime required

You requested that Rust examples be for **Windows OS**. The following are Windows x86-64 examples showing the closest conceptual match to the Linux examples:

- "write to stdout"
- "exit process"

On Windows, the native user/kernel boundary is not a stable "syscall numbers" interface for applications. The correct, supported practice is to call Win32 APIs (which are stable), such as: `GetStdHandle`, `WriteFile`, `ExitProcess`.

Rust: Hello World using Win32 (no external crates)

```
#![cfg(windows)]
```

```
use core::ffi::c_void;

type HANDLE = *mut c_void;
type DWORD = u32;
type BOOL = i32;

const STD_OUTPUT_HANDLE: DWORD = 0xFFFF_FFF5; /* (DWORD)-11 */

#[link(name = "kernel32")]
extern "system" {
    fn GetStdHandle(nStdHandle: DWORD) -> HANDLE;
    fn WriteFile(
        hFile: HANDLE,
        lpBuffer: *const c_void,
        nNumberOfBytesToWrite: DWORD,
        lpNumberOfBytesWritten: *mut DWORD,
        lpOverlapped: *mut c_void,
    ) -> BOOL;
    fn ExitProcess(uExitCode: u32) -> !;
}

fn main() {
    let msg = b"Hello, world (Windows x86-64, Win32 API)\r\n";

    unsafe {
        let h = GetStdHandle(STD_OUTPUT_HANDLE);
        let mut written: DWORD = 0;
```

```
        let _ok = WriteFile(
            h,
            msg.as_ptr() as *const c_void,
            msg.len() as DWORD,
            &mut written as *mut DWORD,
            core::ptr::null_mut(),
        );

        ExitProcess(0);
    }
}
```

Build (MSVC toolchain)

```
rustc -O hello_win.rs
hello_win.exe
```

This produces a normal Windows executable that prints to the console using stable Win32 APIs.

Rust: Minimal inline assembly (Windows x86-64) for arithmetic sanity-check

This example is intentionally small: it demonstrates that you can still do low-level register work when needed, without pretending that Windows provides a stable "syscall ABI" for applications.

```
#![cfg(windows)]

use core::arch::asm;

fn add_u64(a: u64, b: u64) -> u64 {
    let out: u64;
```

```
    unsafe {
        asm!(
            "lea {out}, [{a} + {b}]",
            a = in(reg) a,
            b = in(reg) b,
            out = out(reg) out,
            options(nomem, nostack, preserves_flags),
        );
    }
    out
}

fn main() {
    let x = add_u64(40, 2);
    assert_eq!(x, 42);
}
```

Conceptual mapping to Linux: on Linux we used assembly to call the kernel; on Windows we use stable APIs for OS interaction, and assembly remains useful for tight low-level computation and verification.

34.11 Chapter summary

- A syscall-only Linux x86-64 program is defined by `_start`, `syscall`, and correct register conventions.
- A libc-based program is defined by `main` and the System V ABI function-calling rules.
- For cross-architecture practice, keep examples comparable: “write”, “exit”, and a small register-level computation.

- On Windows, prefer stable Win32 APIs for I/O and process termination; reserve assembly for computation and ABI-adjacent work.

Hello World — Windows x64

This chapter builds the Windows x64 baseline that matches the Linux x86-64 chapter as closely as Windows allows:

- write a message to standard output,
- terminate the process,
- do it with a minimal, precise ABI-correct assembly example,
- provide a comparable Rust example for Windows (no external crates).

All assembly is **GAS (GNU assembler) with Intel syntax**. Because Windows does not provide a stable user-facing "syscall ABI" like Linux, the correct and supported boundary is the **Win32 API** (for console output: `GetStdHandle` + `WriteFile`, for termination: `ExitProcess`).

35.1 Toolchain and file format realities on Windows x64

On Windows x64, executables are **PE/COFF**. You will typically use one of these approaches:

- **MinGW-w64 GCC/Clang + GNU binutils (GAS/ld)**: easiest way to use GAS syntax and link against import libraries such as `kernel32`.
- **MSVC toolchain**: uses MASM (`ml64`) for assembly; GAS is not the native assembler there.

This chapter assumes **MinGW-w64** so we can write **GAS Intel syntax** exactly as requested.

35.2 Windows x64 calling convention essentials

Windows x64 uses a single standard calling convention for user-mode code. The rules you must obey when calling Win32 APIs (and most functions) are:

Register arguments (first four)

Argument	Register
1	`RCX`
2	`RDX`
3	`R8`
4	`R9`

Shadow space (home space)

The caller must reserve **32 bytes** of stack space before a call. This is called *shadow space* (or *home space*). Even when a function has fewer than 4 arguments, the caller still reserves it.

Stack alignment

The stack pointer must be **16-byte aligned at the call boundary**. In practice, this means you usually allocate shadow space plus any extra bytes needed to keep alignment correct.

Non-volatile registers

Registers such as `RBX`, `RBP`, `RSI`, `RDI`, and `R12`–`R15` are non-volatile (callee-saved). If your function modifies them, it must restore them before returning. In our minimal example, we avoid touching them.

35.3 Win32 API mapping for "Hello World"

We will use three Win32 APIs from `kernel32.dll`:

- `GetStdHandle(DWORD nStdHandle)` returns a handle for standard streams.
- `WriteFile(HANDLE hFile, const void* buf, DWORD len, DWORD* written, void* overlapped)` writes bytes to a file/console handle.
- `ExitProcess(UINT exitCode)` terminates the process immediately.

For standard output, Windows uses the constant `STD_OUTPUT_HANDLE = -11`.

35.4 Hello World in pure assembly (no C runtime)

This is the closest Windows equivalent to the Linux "syscall-only" program:

- we define our own entry point,
- we call the OS boundary using Win32 APIs,
- we do not rely on `main` or the C runtime.

Source file: `hello_win64_nocrt.S`

```
.intel_syntax noprefix

/* ---------------------------------------------------------------------------
   hello_win64_nocrt.S (Windows x64, PE/COFF)
   Entry point: Start  (we will set the PE entry point at link time)
   Prints "Hello, world\r\n" using:
     GetStdHandle(STD_OUTPUT_HANDLE)
     WriteFile(handle, msg, len, &written, NULL)
   Then exits using ExitProcess(0)

   Windows x64 ABI requirements we obey:
     - RCX, RDX, R8, R9 carry first 4 args
     - 32-byte shadow space required for each call
     - RSP 16-byte aligned at call boundary
   --------------------------------------------------------------------------- */

.section .rdata,"dr"
msg:
    .ascii "Hello, world (Windows x64, Win32 API)\r\n"
msg_len !=! . - msg

.section .bss,"bw"
    .lcomm written, 4

.section .text,"xr"
.global Start
.extern GetStdHandle
```

```
.extern WriteFile
.extern ExitProcess

/* Constant: (DWORD)-11 */
.equ STD_OUTPUT_HANDLE, -11

Start:
    /* Reserve shadow space and keep alignment.
       We reserve 40 bytes total:
         - 32 bytes shadow space required by ABI
         - 8 bytes extra to maintain 16-byte alignment in common entry conditions
       This is a practical pattern for tiny programs. */
    sub     rsp, 40

    /* h = GetStdHandle(STD_OUTPUT_HANDLE) */
    mov     ecx, STD_OUTPUT_HANDLE
    call    GetStdHandle
    /* handle returned in RAX */

    /* WriteFile(h, msg, msg_len, &written, NULL) */
    mov     rcx, rax                /* 1st arg: HANDLE hFile */
    lea     rdx, [rip !+1 msg]        /* 2nd arg: buffer pointer */
    mov     r8d, msg_len            /* 3rd arg: number of bytes */
    lea     r9, [rip !+1 written]     /* 4th arg: &written */
    /* 5th arg (lpOverlapped) goes on stack, above shadow space.
       Layout (top of stack after sub rsp,40):
         [rsp +  0..31] shadow space (for callee use)
         [rsp + 32..39] extra/padding
       We place the 5th argument at [rsp + 32]. */
```

```
    xor     eax, eax
    mov     [rsp !+! 32], rax         /* lpOverlapped = NULL */
    call    WriteFile

    /* ExitProcess(0) */
    xor     ecx, ecx
    call    ExitProcess
```

Build (MinGW-w64)

Two reliable build methods are shown. Use whichever matches your environment.

Method A: compile and link with `x86_64-w64-mingw32-gcc`

```
x86_64-w64-mingw32-gcc -c -O2 hello_win64_nocrt.S -o hello_win64_nocrt.o
x86_64-w64-mingw32-gcc -nostdlib -Wl,-e,Start -Wl,--subsystem,console \
  hello_win64_nocrt.o -lkernel32 -o hello_win64_nocrt.exe
hello_win64_nocrt.exe
```

Method B: link with `ld` explicitly

```
x86_64-w64-mingw32-as --64 hello_win64_nocrt.S -o hello_win64_nocrt.o
x86_64-w64-mingw32-ld -e Start --subsystem console \
  -o hello_win64_nocrt.exe hello_win64_nocrt.o -lkernel32
hello_win64_nocrt.exe
```

Engineering note: Using gcc as the linker driver (Method A) is often more robust because it supplies the correct default libraries and paths for the target. We still pass `-nostdlib` here to avoid CRT startup objects.

35.5 Hello World with a normal `main` (CRT-managed startup)

This matches the Linux "libc-based" variant conceptually: we implement `main`, and the runtime initializes and then calls it.

Source file: `hello_win64_main.S`

```
.intel_syntax noprefix

/* ------------------------------------------------------------------------
   hello_win64_main.S (Windows x64, PE/COFF)
   Entry point is provided by the runtime; we provide main.
   We still call Win32 APIs directly.

   Windows x64 ABI:
     - must reserve 32 bytes shadow space for each call
     - must keep 16-byte alignment at call boundary
   ------------------------------------------------------------------------ */

.section .rdata,"dr"
msg:
    .ascii "Hello, world (Windows x64, from main)\r\n"
msg_len !=! . - msg

.section .bss,"bw"
    .lcomm written, 4

.section .text,"xr"
.global main
```

```
.extern GetStdHandle
.extern WriteFile
.extern ExitProcess
.equ STD_OUTPUT_HANDLE, -11

main:
    /* Prologue: reserve shadow space + alignment padding */
    sub     rsp, 40

    /* h = GetStdHandle(STD_OUTPUT_HANDLE) */
    mov     ecx, STD_OUTPUT_HANDLE
    call    GetStdHandle

    /* WriteFile(h, msg, msg_len, &written, NULL) */
    mov     rcx, rax
    lea     rdx, [rip !+! msg]
    mov     r8d, msg_len
    lea     r9, [rip !+! written]
    xor     eax, eax
    mov     [rsp !+! 32], rax
    call    WriteFile

    /* Restore stack and return 0 to runtime */
    add     rsp, 40
    xor     eax, eax
    ret
```

Build

```
x86_64-w64-mingw32-gcc -O2 hello_win64_main.S -o hello_win64_main.exe
hello_win64_main.exe
```

35.6 Precise comparison to Linux x86-64

The Linux chapter used `syscall` with a stable syscall ABI and numbers. Windows does not document a stable application syscall interface; applications should use Win32 APIs. The comparable boundary is:

Concern	Linux x86-64	Windows x64
Write to stdout	`write(1, buf, len)` syscall	`WriteFile(GetStdHandle(...), buf, len, ...)`
Exit process	`exit(code)` syscall	`ExitProcess(code)`
ABI focus	syscall register ABI	Windows x64 function ABI (RCX,RDX,R8,R9 + shadow space)
Binary format	ELF64	PE/COFF

35.7 Practical pitfalls (what breaks real code)

Pitfall 1: forgetting shadow space

If you do not reserve 32 bytes of shadow space before calling Win32 APIs, your program may crash unpredictably. Some callees spill register arguments into that home space.

Pitfall 2: stack misalignment

If the stack is not 16-byte aligned at the call boundary, some functions (especially those using SIMD internally) may fault or behave incorrectly.

Pitfall 3: wrong register order

Windows x64 uses `RCX, RDX, R8, R9`. This differs from Linux System V (`RDI, RSI, RDX, RCX, R8, R9`).

Pitfall 4: confusing console output APIs

`WriteFile` works on console handles as well as file handles, but error handling differs if output is redirected. Keep the example minimal, then add robust checks later.

35.8 Extensive Rust examples for Windows (no external crates)

You requested that all Rust examples target Windows OS. The following examples show:

- the same Win32 APIs as the assembly version,
- careful types that match Windows expectations,
- a small reusable wrapper.

Rust: minimal Win32 Hello World (direct FFI)

```
#![cfg(windows)]

use core::ffi::c_void;
```

```
type HANDLE = *mut c_void;
type DWORD = u32;
type BOOL = i32;

const STD_OUTPUT_HANDLE: DWORD = 0xFFFF_FFF5; /* (DWORD)-11 */

#[link(name = "kernel32")]
extern "system" {
    fn GetStdHandle(nStdHandle: DWORD) -> HANDLE;

    fn WriteFile(
        hFile: HANDLE,
        lpBuffer: *const c_void,
        nNumberOfBytesToWrite: DWORD,
        lpNumberOfBytesWritten: *mut DWORD,
        lpOverlapped: *mut c_void,
    ) -> BOOL;

    fn ExitProcess(uExitCode: u32) -> !;
}

fn main() {
    let msg = b"Hello, world (Rust on Windows x64, Win32 API)\r\n";

    unsafe {
        let h = GetStdHandle(STD_OUTPUT_HANDLE);

        let mut written: DWORD = 0;
        let _ok = WriteFile(
```

```
            h,
            msg.as_ptr() as *const c_void,
            msg.len() as DWORD,
            &mut written as *mut DWORD,
            core::ptr::null_mut(),
        );

        ExitProcess(0);
    }
}
```

Rust: a small safe wrapper for writing bytes to stdout

This wrapper keeps unsafe code contained and provides a reusable function suitable for later chapters.

```
#![cfg(windows)]

use core::ffi::c_void;

type HANDLE = *mut c_void;
type DWORD = u32;
type BOOL = i32;

const STD_OUTPUT_HANDLE: DWORD = 0xFFFF_FFF5; /* (DWORD)-11 */

#[link(name = "kernel32")]
extern "system" {
    fn GetStdHandle(nStdHandle: DWORD) -> HANDLE;
```

```
    fn WriteFile(
        hFile: HANDLE,
        lpBuffer: *const c_void,
        nNumberOfBytesToWrite: DWORD,
        lpNumberOfBytesWritten: *mut DWORD,
        lpOverlapped: *mut c_void,
    ) -> BOOL;
}

fn write_stdout(buf: &[u8]) -> Result<usize, ()> {
    unsafe {
        let h = GetStdHandle(STD_OUTPUT_HANDLE);
        if h.is_null() {
            return Err(());
        }

        let mut written: DWORD = 0;
        let ok = WriteFile(
            h,
            buf.as_ptr() as *const c_void,
            buf.len() as DWORD,
            &mut written as *mut DWORD,
            core::ptr::null_mut(),
        );

        if ok == 0 {
            Err(())
        } else {
            Ok(written as usize)
```

```
        }
    }
}

fn main() {
    let _ = write_stdout(b"Hello from a small Rust wrapper\r\n");
}
```

Rust: minimal inline assembly sanity-check (Windows x64)

This example is intentionally small and architecture-focused: it demonstrates register-level work while keeping OS interaction separate.

```
#![cfg(windows)]

use core::arch::asm;

fn add_u64(a: u64, b: u64) -> u64 {
    let out: u64;
    unsafe {
        asm!(
            "lea {out}, [{a} + {b}]",
            a = in(reg) a,
            b = in(reg) b,
            out = out(reg) out,
            options(nomem, nostack, preserves_flags),
        );
    }
    out
}
```

```
fn main() {
    let v = add_u64(41, 1);
    assert_eq!(v, 42);
}
```

35.9 Build instructions for Rust examples (Windows x64)

Using `rustc` directly

```
rustc -O hello_win.rs
hello_win.exe
```

Using Cargo

```
cargo new hello_win
cd hello_win
/* replace src/main.rs with one of the examples */
cargo run --release
```

35.10 Chapter summary

- Windows x64 “Hello World” is an ABI exercise: register arguments, shadow space, and alignment.
- The supported OS boundary is Win32 APIs, not a stable syscall interface for applications.
- With GAS Intel syntax under MinGW-w64, you can produce small PE/COFF console programs without the C runtime.

- Rust on Windows can call the same Win32 APIs directly using `extern "system"` and minimal type definitions, with no external crates.

Hello World — AArch64 Linux

This chapter provides a precise, comparable "Hello World" baseline for **Linux on AArch64 (ARM64)**. It mirrors the intent of the x86-64 Linux chapter:

- a minimal program that prints a message and exits,
- a variant that uses the C runtime and libc,
- a clear ABI-focused explanation of argument passing and system call invocation,
- build commands and inspection habits you can reuse across architectures.

Important note about syntax: "Intel syntax" is specific to x86/x86-64. AArch64 assembly uses a different instruction set and syntax. In this chapter we use **GAS for AArch64** with standard AArch64 assembly syntax. All comments are written using `/* ... */` as requested.

36.1 Toolchain assumptions

You can build AArch64 Linux assembly in two common scenarios:

Scenario A: Native AArch64 Linux

- You are on an ARM64 machine (for example: Linux on ARM server, ARM laptop, single-board computer).

- Tools: `as`, `ld`, `gcc`.

Scenario B: Cross-compiling from x86-64 Linux

- You use a cross toolchain, typically `aarch64-linux-gnu-gcc`, `aarch64-linux-gnu-as`, `aarch64-linux-gnu-ld`.
- You run binaries on real ARM64 hardware or via an emulator such as `qemu-aarch64` (optional).

36.2 AArch64 Linux syscall ABI (the essentials)

On Linux AArch64, syscalls are invoked with `svc #0`:

- Syscall number is placed in `x8`.
- Arguments are placed in `x0`–`x5`.
- Return value is in `x0`.

We will use two syscalls

- `write(fd, buf, len)`: syscall number `64`
- `exit(code)`: syscall number `93`

This differs from Linux x86-64 where the syscall number is in `RAX` and `syscall` is used.

36.3 Hello World (no libc): pure syscalls

This version provides its own entry point `_start` and calls the kernel directly.

Source file: hello_aarch64_syscall.S

```
/* ------------------------------------------------------------------
   hello_aarch64_syscall.S (Linux AArch64, ELF64)
   Entry point: _start
   Prints "Hello, world\n" using write(1, msg, len) then exits(0)
   Syscall ABI:
     x8 = syscall number
     x0..x5 = args
     svc #0
   ------------------------------------------------------------------ */

    .section .rodata
msg:
    .ascii  "Hello, world (Linux AArch64)\n"
msg_end:
    .equ    msg_len, msg_end - msg

    .section .text
    .global _start
_start:
    /* write(1, msg, msg_len) */
    mov     x0, #1              /* fd = STDOUT */
    adr     x1, msg             /* buf = &msg (PC-relative within range) */
    mov     x2, #msg_len        /* len */
    mov     x8, #64             /* __NR_write on AArch64 */
    svc     #0

    /* exit(0) */
```

```
    mov     x0, #0              /* status = 0 */
    mov     x8, #93             /* __NR_exit on AArch64 */
    svc     #0
```

Build and run

Native AArch64 Linux

```
as hello_aarch64_syscall.S -o hello_aarch64_syscall.o
ld hello_aarch64_syscall.o -o hello_aarch64_syscall
./hello_aarch64_syscall
```

Cross-compile from x86-64 Linux

```
aarch64-linux-gnu-as hello_aarch64_syscall.S -o hello_aarch64_syscall.o
aarch64-linux-gnu-ld hello_aarch64_syscall.o -o hello_aarch64_syscall
```

If you run on an AArch64 machine, execute it normally. If you run on x86-64, you need an emulator or a real AArch64 target.

Why `adr` here?

`adr x1, msg` computes the PC-relative address of `msg` when it is within range. For tiny programs, this is the simplest approach. In large position-independent programs, you will typically see sequences that use `adrp` + `add` to form full addresses reliably across wider ranges.

36.4 Hello World (with libc): normal `main`

This mirrors the “libc-based” variant on x86-64: the runtime provides `_start` and calls `main`. We call `puts` (or `printf`) from libc.

Source file: `hello_aarch64_libc.S`

```
/* ------------------------------------------------------------------
   hello_aarch64_libc.S (Linux AArch64, ELF64)
   Entry point is managed by the runtime; we provide main.
   Calls libc puts(const char*).
   ABI:
     x0 = first argument
     return value in w0/x0
   ------------------------------------------------------------------ */

    .section .rodata
str:
    .asciz "Hello, world (Linux AArch64, libc)"

    .section .text
    .global main
    .extern puts

main:
    /* puts(str) */
    adr     x0, str             /* x0 = pointer to string */
    bl      puts

    /* return 0 */
    mov     w0, #0
    ret
```

Build and run

Native AArch64 Linux

```
gcc hello_aarch64_libc.S -o hello_aarch64_libc
./hello_aarch64_libc
```

Cross-compile from x86-64 Linux

```
aarch64-linux-gnu-gcc hello_aarch64_libc.S -o hello_aarch64_libc
```

36.5 A precise comparison: Linux x86-64 vs Linux AArch64

Topic	Linux x86-64	Linux AArch64
Syscall instruction	`syscall`	`svc #0`
Syscall number reg	`RAX`	`x8`
Arg registers	`RDI,RSI,RDX,R10,R8,R9`	`x0–x5`
Return register	`RAX`	`x0`
`write` syscall #	1	64
`exit` syscall #	60	93

The **concept** is identical across architectures:

- Place syscall number in the designated register.
- Place arguments in the designated registers.
- Execute the privileged transition instruction.
- Read result from the designated return register.

36.6 Practical pitfalls (common failure modes)

Pitfall 1: using the wrong syscall numbers

Syscall numbers are architecture-specific. Linux AArch64 uses different numbers than Linux x86-64.

Pitfall 2: mixing up function ABI with syscall ABI

Normal function calls on AArch64 use `x0`–`x7` for arguments and `bl` for branching with link. Syscalls use `x8` for the syscall number and `svc #0`.

Pitfall 3: address formation in larger programs

For very small examples, `adr` is convenient. In large PIE programs, you often need `adrp+add` to reliably compute full addresses across large binary layouts.

36.7 Cross-architecture takeaway

When you want "small but comparable" examples across CPU architectures, keep the **behavior constant** and vary only the ABI surface:

- **Same behavior:** write bytes to stdout, then exit.
- **Different ABI:** registers, syscall instruction, syscall numbers.
- **Same engineering method:** minimal code, explicit inputs/outputs, and simple build steps.

36.8 Rust examples (Windows OS only, as requested)

You requested that **all Rust code in this booklet be for Windows OS**. Therefore, even though this chapter targets Linux AArch64 in assembly, the Rust examples below are Windows x64 equivalents that demonstrate the same conceptual action: **write to stdout and exit** using stable Win32 APIs.

Rust: Hello World using Win32 (no external crates)

```rust
#![cfg(windows)]

use core::ffi::c_void;

type HANDLE = *mut c_void;
type DWORD = u32;
type BOOL = i32;

const STD_OUTPUT_HANDLE: DWORD = 0xFFFF_FFF5; /* (DWORD)-11 */

#[link(name = "kernel32")]
extern "system" {
    fn GetStdHandle(nStdHandle: DWORD) -> HANDLE;
    fn WriteFile(
        hFile: HANDLE,
        lpBuffer: *const c_void,
        nNumberOfBytesToWrite: DWORD,
        lpNumberOfBytesWritten: *mut DWORD,
        lpOverlapped: *mut c_void,
    ) -> BOOL;
```

```
    fn ExitProcess(uExitCode: u32) -> !;
}

fn main() {
    let msg = b"Hello, world (Rust on Windows, Win32 API)\r\n";
    unsafe {
        let h = GetStdHandle(STD_OUTPUT_HANDLE);
        let mut written: DWORD = 0;
        let _ok = WriteFile(
            h,
            msg.as_ptr() as *const c_void,
            msg.len() as DWORD,
            &mut written as *mut DWORD,
            core::ptr::null_mut(),
        );
        ExitProcess(0);
    }
}
```

Rust: small reusable wrapper for stdout

```
#![cfg(windows)]

use core::ffi::c_void;

type HANDLE = *mut c_void;
type DWORD = u32;
type BOOL = i32;
```

```
const STD_OUTPUT_HANDLE: DWORD = 0xFFFF_FFF5; /* (DWORD)-11 */

#[link(name = "kernel32")]
extern "system" {
    fn GetStdHandle(nStdHandle: DWORD) -> HANDLE;
    fn WriteFile(
        hFile: HANDLE,
        lpBuffer: *const c_void,
        nNumberOfBytesToWrite: DWORD,
        lpNumberOfBytesWritten: *mut DWORD,
        lpOverlapped: *mut c_void,
    ) -> BOOL;
}

fn write_stdout(buf: &[u8]) -> Result<usize, ()> {
    unsafe {
        let h = GetStdHandle(STD_OUTPUT_HANDLE);
        if h.is_null() {
            return Err(());
        }
        let mut written: DWORD = 0;
        let ok = WriteFile(
            h,
            buf.as_ptr() as *const c_void,
            buf.len() as DWORD,
            &mut written as *mut DWORD,
            core::ptr::null_mut(),
        );
        if ok == 0 { Err(()) } else { Ok(written as usize) }
```

```
    }
}

fn main() {
    let _ = write_stdout(b"Hello from a reusable Rust wrapper\r\n");
}
```

36.9 Chapter summary

- Linux AArch64 syscalls use `x8` for syscall number, `x0`–`x5` for arguments, and `svc #0`.
- The minimal syscall-only program uses `_start` and calls `write` and `exit` directly.
- The libc-based variant provides `main` and calls a libc function such as `puts`.
- For cross-architecture learning, keep examples behavior-identical and vary only the ABI boundary.
- Rust examples are provided for Windows only (Win32 API), per your booklet-wide constraint.

Hello World — RISC-V Linux

This chapter provides a precise, comparable "Hello World" baseline for **Linux on RISC-V**. It is designed to mirror the same learning outcomes as the x86-64 Linux and AArch64 Linux chapters:

- a minimal program that prints a message and exits using **direct syscalls**,
- a variant that uses the **C runtime and libc**,
- a compact but rigorous explanation of the **Linux RISC-V syscall ABI**,
- build commands for native and cross-toolchains,
- small cross-architecture comparisons that keep behavior constant.

Important note about syntax: "Intel syntax" is specific to x86/x86-64. RISC-V assembly is a different ISA with its own syntax. In this chapter we use **GAS for RISC-V** with standard RISC-V assembly syntax. All comments use `/* ... */` as requested.

37.1 Toolchain assumptions

You can build RISC-V Linux assembly in two common scenarios:

Scenario A: Native RISC-V Linux

- You are running Linux on a RISC-V machine.
- Tools: `as`, `ld`, `gcc`.

Scenario B: Cross-compiling from x86-64 Linux

- You use a cross toolchain such as `riscv64-linux-gnu-gcc`, `riscv64-linux-gnu-as`, `riscv64-linux-gnu-ld`.
- You run binaries on real RISC-V hardware or via an emulator (optional).

37.2 Linux RISC-V syscall ABI (the essentials)

On Linux RISC-V, syscalls are performed using the **environment call** instruction:

- `ecall` traps to the kernel.
- Syscall number is placed in `a7`.
- Arguments are placed in `a0`–`a5`.
- Return value is in `a0`.

We will use two syscalls

- `write(fd, buf, len)`: syscall number `64` (common on rv64 Linux)
- `exit(status)`: syscall number `93`

Engineering note: syscall numbers are defined by the Linux kernel ABI for the architecture. In practice, for RV64 Linux, `write=64` and `exit=93` are the standard values used by user space.

37.3 Hello World (no libc): pure syscalls

This version provides its own `_start` entry point and calls the Linux kernel directly.

Source file: `hello_riscv64_syscall.S`

```
/* ------------------------------------------------------------------
   hello_riscv64_syscall.S (Linux RISC-V 64-bit, ELF64)
   Entry point: _start
   Prints "Hello, world\n" using write(1, msg, len) then exits(0)

   Syscall ABI (Linux RISC-V):
     a7 = syscall number
     a0..a5 = args
     ecall
     return in a0
   ------------------------------------------------------------------ */

    .section .rodata
msg:
    .ascii  "Hello, world (Linux RISC-V)\n"
msg_end:
    .equ    msg_len, msg_end - msg

    .section .text
    .globl  _start
_start:
    /* write(1, msg, msg_len) */
    li      a0, 1           /* fd = STDOUT */
```

```
    la      a1, msg         /* buf = &msg */
    li      a2, msg_len     /* len */
    li      a7, 64          /* __NR_write (rv64 Linux) */
    ecall

    /* exit(0) */
    li      a0, 0           /* status = 0 */
    li      a7, 93          /* __NR_exit (rv64 Linux) */
    ecall
```

Build and run

Native RISC-V Linux

```
as hello_riscv64_syscall.S -o hello_riscv64_syscall.o
ld hello_riscv64_syscall.o -o hello_riscv64_syscall
./hello_riscv64_syscall
```

Cross-compile from x86-64 Linux

```
riscv64-linux-gnu-as hello_riscv64_syscall.S -o hello_riscv64_syscall.o
riscv64-linux-gnu-ld hello_riscv64_syscall.o -o hello_riscv64_syscall
```

To run a cross-compiled binary, execute it on a RISC-V Linux target (hardware or emulator).

37.4 Hello World (with libc): normal `main`

In this variant, the runtime provides `_start` and calls `main`. We call libc `puts`.

Source file: `hello_riscv64_libc.S`

```
/* ------------------------------------------------------------------
   hello_riscv64_libc.S (Linux RISC-V 64-bit, ELF64)
   Entry point is managed by the runtime; we provide main.
   Calls libc puts(const char*).

   Function ABI (RISC-V):
     a0..a7 = argument/return registers (a0 is also return)
     call via jal/jalr (assembler uses 'call' pseudo)
   ------------------------------------------------------------------ */

    .section .rodata
str:
    .asciz "Hello, world (Linux RISC-V, libc)"

    .section .text
    .globl  main
    .extern puts

main:
    /* puts(str) */
    la      a0, str
    call    puts

    /* return 0 */
    li      a0, 0
    ret
```

Build and run

Native RISC-V Linux

```
gcc hello_riscv64_libc.S -o hello_riscv64_libc
./hello_riscv64_libc
```

Cross-compile from x86-64 Linux

```
riscv64-linux-gnu-gcc hello_riscv64_libc.S -o hello_riscv64_libc
```

37.5 A precise comparison: Linux x86-64 vs AArch64 vs RISC-V

Topic	x86-64	AArch64	RISC-V
Syscall instruction	`syscall`	`svc #0`	`ecall`
Syscall # register	`RAX`	`x8`	`a7`
Arg registers	`RDI,RSI,RDX,R10,R8,`	`x0–x5`	`a0–a5`
Return register	`RAX`	`x0`	`a0`

The cross-architecture constant is:

- the kernel boundary needs a syscall number,
- arguments must be placed where the ABI expects,
- one instruction triggers the privilege transition,
- a return value indicates success or an error convention.

37.6 Practical pitfalls (real failure modes)

Pitfall 1: confusing the RISC-V Linux syscall ABI with SBI calls

RISC-V also has the Supervisor Binary Interface (SBI) used by firmware/hypervisors. Linux user-space syscalls are **not** SBI calls; they are Linux syscalls via `ecall` with `a7` as syscall number.

Pitfall 2: mixing RV32 and RV64 assumptions

This chapter assumes RV64 (64-bit). While the mechanism is the same, toolchains, ABI details, and some constants can differ in 32-bit environments.

Pitfall 3: assuming syscall numbers match other architectures

Syscall numbers are **architecture-specific**. Always treat them as part of the architecture ABI.

Pitfall 4: address loading in position-independent code

`la` is a pseudo-instruction. The assembler expands it into sequences appropriate for the selected code model. For tiny examples it is ideal; for large PIE/shared objects, understand how it expands.

37.7 Cross-architecture takeaway

For “small but comparable” exercises, keep the behavior identical:

- **Write** a fixed buffer to stdout.
- **Exit** with a status code.

- Use the minimum ABI surface (syscall ABI or stable OS API boundary).

Then vary only the machine-dependent parts:

- syscall instruction and syscall-number register,
- argument registers,
- syscall numbers for the chosen kernel ABI.

37.8 Rust examples (Windows OS only, as requested)

You requested that **all Rust code in this booklet be for Windows OS**. Therefore, even though this chapter targets Linux RISC-V for assembly, the Rust examples below are the Windows equivalents that implement the same behavior: **write to stdout and exit**, using stable Win32 APIs.

Rust: Hello World using Win32 (no external crates)

```
#![cfg(windows)]

use core::ffi::c_void;

type HANDLE = *mut c_void;
type DWORD = u32;
type BOOL = i32;

const STD_OUTPUT_HANDLE: DWORD = 0xFFFF_FFF5; /* (DWORD)-11 */

#[link(name = "kernel32")]
```

```
extern "system" {
    fn GetStdHandle(nStdHandle: DWORD) -> HANDLE;
    fn WriteFile(
        hFile: HANDLE,
        lpBuffer: *const c_void,
        nNumberOfBytesToWrite: DWORD,
        lpNumberOfBytesWritten: *mut DWORD,
        lpOverlapped: *mut c_void,
    ) -> BOOL;
    fn ExitProcess(uExitCode: u32) -> !;
}

fn main() {
    let msg = b"Hello, world (Rust on Windows, Win32 API)\r\n";
    unsafe {
        let h = GetStdHandle(STD_OUTPUT_HANDLE);
        let mut written: DWORD = 0;
        let _ok = WriteFile(
            h,
            msg.as_ptr() as *const c_void,
            msg.len() as DWORD,
            &mut written as *mut DWORD,
            core::ptr::null_mut(),
        );
        ExitProcess(0);
    }
}
```

Rust: reusable stdout writer

```rust
#![cfg(windows)]

use core::ffi::c_void;

type HANDLE = *mut c_void;
type DWORD = u32;
type BOOL = i32;

const STD_OUTPUT_HANDLE: DWORD = 0xFFFF_FFF5; /* (DWORD)-11 */

#[link(name = "kernel32")]
extern "system" {
    fn GetStdHandle(nStdHandle: DWORD) -> HANDLE;
    fn WriteFile(
        hFile: HANDLE,
        lpBuffer: *const c_void,
        nNumberOfBytesToWrite: DWORD,
        lpNumberOfBytesWritten: *mut DWORD,
        lpOverlapped: *mut c_void,
    ) -> BOOL;
}

fn write_stdout(buf: &[u8]) -> Result<usize, ()> {
    unsafe {
        let h = GetStdHandle(STD_OUTPUT_HANDLE);
        if h.is_null() {
            return Err(());
```

```
        }
        let mut written: DWORD = 0;
        let ok = WriteFile(
            h,
            buf.as_ptr() as *const c_void,
            buf.len() as DWORD,
            &mut written as *mut DWORD,
            core::ptr::null_mut(),
        );
        if ok == 0 { Err(()) } else { Ok(written as usize) }
    }
}

fn main() {
    let _ = write_stdout(b"Hello from Rust wrapper on Windows\r\n");
}
```

37.9 Chapter summary

- Linux RISC-V syscalls use `a7` for syscall number, `a0`–`a5` for arguments, and `ecall`.
- The minimal syscall-only program provides `_start` and performs `write` then `exit`.
- The libc-based variant provides `main` and calls a libc function such as `puts`.
- For cross-architecture practice, keep behavior identical and vary only the ABI boundary (instruction, registers, syscall numbers).
- Rust examples are provided for Windows only (Win32 API), per your booklet-wide constraint.

Function Call Comparisons — Same Function on 3 ISAs

This chapter shows how the same function is implemented in three instruction set architectures (ISAs):

- x86-64 (System V ABI),
- AArch64 (ARM64 Linux ABI),
- RISC-V 64 (Linux ABI).

We will implement a simple function in C and then present its ABI-compliant assembly equivalents that:

- accept integer, pointer, and floating-point arguments,
- return an integer result,
- handle arguments in registers per each ABI,
- preserve required registers and conform to calling conventions.

In addition, we provide:

- exact build steps,

- disassembly inspection commands,
- a small Rust program for Windows that calls a similar function using the Microsoft x64 calling convention (for completeness).

38.1 Specification of the test function

Here is the C function we wish to compile and then provide assembly equivalents:

```c
int combine(int a, int b, int *outp, float f) {
    /* Store sum of a and b in *outp and return (int)(f * 2.0f). */
    *outp = a + b;
    return (int)(f * 2.0f);
}
```

This function:

- receives two 32-bit integers,
- a pointer to an integer where it stores the sum,
- a 32-bit floating-point value,
- returns a 32-bit integer result.

The implementation mixes integer and floating-point argument passing and return value handling, making it ideal for ABI comparison.

38.2 x86-64 System V ABI implementation

The System V AMD64 ABI (used on Linux, macOS, and many Unix-like systems) assigns registers as follows:

Integer args	`RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9`
Floating args	`XMM0`, `XMM1`, `XMM2`, `XMM3`, `XMM4`, `XMM5`, `XMM6`, `XMM7`
Integer return	`RAX`
Floating return	`XMM0`

Caller-saved registers must be preserved by the caller if needed; the callee may clobber them.

Assembly source: `combine_x86_64.S`

```
.intel_syntax noprefix

/* combine_x86_64.S
   System V ABI
   int combine(int a, int b, int *outp, float f)
   a -> edi
   b -> esi
   outp -> edx
   f -> xmm0
   returns integer result in eax
*/

    .text
    .globl combine
combine:
    /* Save volatile registers if needed (none here) */

    /* Compute *outp = a + b */
    mov     eax, edi          /* eax = a */
    add     eax, esi          /* eax = a + b */
```

```
    mov     DWORD PTR [rdx], eax

    /* Compute (int)(f * 2.0f) */
    movss   xmm1, DWORD PTR .LC_TWO /* load constant 2.0f */
    mulss   xmm0, xmm1        /* xmm0 = f * 2.0f */
    cvttss2si eax, xmm0       /* convert float to int with truncation */

    ret

    .section .rodata
.LC_TWO:
    .float  2.0
```

Build and disassemble

```
gcc -O2 -c combine_x86_64.S -o combine_x86_64.o
gcc -O2 combine_x86_64.o -o combine_x86_64
objdump -d -Mintel combine_x86_64.o
```

38.3 AArch64 (ARM64 Linux ABI) implementation

The AArch64 ABI (Linux and many Unix-like) assigns arguments as follows:

Integer args	`x0`, `x1`, `x2`, `x3`, `x4`, `x5`
Floating args	`v0`, `v1`, `v2`, `v3`, ...
Integer return	`x0`
Floating return	`v0`

The first integer args go in `x0`–`x3`, then remaining in `x4`, . . . and the first floating arg goes in `v0`.

Assembly source: `combine_aarch64.S`

```
/* combine_aarch64.S
   AArch64 Linux ABI
   int combine(int a, int b, int *outp, float f)
   a -> x0
   b -> x1
   outp -> x2
   f -> v0.s[0]
   returns int in x0
*/

    .text
    .globl combine
combine:
    /* Compute *outp = a + b */
    add     w3, w0, w1          /* w3 = a + b */
    str     w3, [x2]            /* *outp = w3 */

    /* Compute (int)(f * 2.0f) */
    fmov    s1, #2.0            /* s1 = 2.0f */
    fmul    s0, s0, s1          /* s0 = f * 2.0f */
    fcvtzs w0, s0               /* convert float to int (zero-trap) */

    ret
```

Build and disassemble

```
aarch64-linux-gnu-gcc -O2 -c combine_aarch64.S -o combine_aarch64.o
aarch64-linux-gnu-gcc -O2 combine_aarch64.o -o combine_aarch64
```

```
riscv64-linux-gnu-objdump -d -Abin combine_aarch64.o
```

38.4 RISC-V 64 Linux ABI implementation

RISC-V 64 Linux ABI argument registers:

Integer args	`a0–a7`
Floating args	`fa0–fa7`
Integer return	`a0`
Floating return	`fa0`

For our function:

- `a0` = `int a`,
- `a1` = `int b`,
- `a2` = `int *outp`,
- `fa0` = `float f`.

Assembly source: `combine_riscv64.S`

```
/* combine_riscv64.S
   RISC-V 64 Linux ABI
   int combine(int a, int b, int *outp, float f)
   a -> a0
   b -> a1
   outp -> a2
   f -> fa0
   returns int in a0
```

```
*/

    .text
    .globl combine
combine:
    /* Compute *outp = a + b */
    add     a3, a0, a1          /* a3 = a + b */
    sw      a3, 0(a2)           /* *outp = a3 */

    /* Compute (int)(f * 2.0f) */
    li      t0, 0x40000000      /* bit pattern for 2.0f */
    fmv_h      fa1, t0          /* fa1 = 2.0f (as float) */
    fmul    fa0, fa0, fa1       /* fa0 = f * 2.0f */
    fcvt.w.s a0, fa0, rtz       /* convert float to int (round toward zero) */

    ret
```

Build and disassemble

```
riscv64-linux-gnu-gcc -O2 -c combine_riscv64.S -o combine_riscv64.o
riscv64-linux-gnu-gcc -O2 combine_riscv64.o -o combine_riscv64
riscv64-linux-gnu-objdump -d combine_riscv64.o
```

38.5 Comparison tables: passing ints, pointers, floats

The following table shows how each ABI passes the arguments:

Argument	x86-64 SysV	AArch64 Linux	RISC-V 64 Linux
int a	`RDI / w0`	`x0`	`a0`
int b	`RSI / w1`	`x1`	`a1`
int *outp	`RDX / w2`	`x2`	`a2`
float f	`XMM0 / s0`	`v0.s[0]`	`fa0`

And how the return value is conveyed:

Return type	x86-64 SysV	AArch64 Linux	RISC-V 64 Linux
int	`EAX / w0 / a0`	`x0`	`a0`

38.6 Calling from Rust on Windows (MS x64 ABI)

Although the above assembly focuses on Linux ABIs, for Windows on x86-64 the Microsoft ABI differs slightly:

- first four integer or pointer args in `RCX, RDX, R8, R9`,
- floating args in `XMM0–XMM3`,
- caller reserves 32 bytes of shadow space,
- return values in `RAX/XMM0` per type.

Here is a Rust example that declares the same function signature and calls it.

```
#![cfg(windows)]

#[no_mangle]
pub extern "C" fn combine_rust(
```

```
    a: i32,
    b: i32,
    outp: *mut i32,
    f: f32,
) -> i32 {
    unsafe {
        if !outp.is_null() {
            *outp = a + b;
        }
        (f * 2.0) as i32
    }
}

fn main() {
    let mut out_val: i32 = 0;
    let result = combine_rust(10, 32, &mut out_val as *mut i32, 3.14);
    println!("sum = {}, result = {}", out_val, result);
}
```

38.7 Practical pitfalls

Pitfall 1: mixing up float argument registers

On x86-64 SysV floating args go in `XMM0+`, on AArch64 they go in `v0`, on RISC-V they go in `fa0`. Be sure to assign them properly.

Pitfall 2: pointer size mismatch

All three ABIs use 64-bit pointers in 64-bit mode, but if mixing with 32-bit code or structs, ensure pointer sizes match.

Pitfall 3: wrong return conventions

Each ABI conveys integer and floating return values in different registers; mixing them will corrupt the caller.

38.8 Summary

This chapter demonstrated:

- three ISA-specific ABI-compliant implementations of the same function,
- how integer, pointer, and floating-point args are passed,
- how return values are conveyed,
- how calling conventions influence register allocation,
- how to call a similar function from Rust on Windows using the MS x64 ABI.

Minimal Atomic Increment Example

This chapter presents the smallest *practical* atomic increment across three ISAs:

- x86/x86-64 using `LOCK` (single-instruction atomic read-modify-write),
- AArch64 using `LDAXR/STLXR` (exclusive load/store loop),
- RISC-V using `AMO` (atomic memory operation, optionally with `.aq/.rl` ordering).

The goal is not a large lab. The goal is a precise, comparable example that highlights:

- atomicity mechanism,
- minimal ordering implications,
- a stable function shape you can reuse later for counters, reference counts, tickets, and locks.

39.1 What exactly are we implementing?

We implement the following conceptual operation on a 32-bit integer located in memory:

```
uint32_t atomic_fetch_add_u32(uint32_t* p, uint32_t delta);
/* Atomically:
     old = *p;
```

```
     *p  = old + delta;
   Return old.
*/
```

Returning the *previous* value (`fetch_add`) is a standard primitive. From it you can easily get the post-increment value: `old + delta`.

39.2 Ordering model used in this chapter

Atomicity and ordering are distinct concepts:

- **Atomicity:** no intermediate state is visible; increments do not get lost.
- **Ordering:** whether surrounding loads/stores can be reordered across the atomic operation.

For comparability, we present a **strong default** equivalent to *acquire*+*release* around the update:

- x86 `LOCK`-prefixed RMW instructions are strongly ordered (sufficiently strong for most lock-free counters).
- AArch64 will use `LDAXR` (acquire) and `STLXR` (release).
- RISC-V will use `amoadd.w.aqrl` to request acquire+release semantics on the AMO.

This is not the only valid choice, but it is a good educational baseline: the code demonstrates the explicit places where weaker/stronger ordering can be selected per architecture.

39.3 x86: `LOCK` atomic increment

On x86, a `LOCK`-prefixed read-modify-write instruction performs an atomic update visible to all cores. A canonical `fetch_add` implementation uses `XADD`:

- `LOCK XADD [mem], reg` atomically exchanges `reg` with `[mem]`, then adds the original `reg` into memory.
- After the instruction, `reg` contains the old memory value.

x86-64 System V ABI (Linux) version

Function prototype:

```
uint32_t atomic_fetch_add_u32(uint32_t* p, uint32_t delta);
```

Register mapping (x86-64 System V):

- `p` in `RDI`
- `delta` in `ESI`
- return value in `EAX`

Assembly: `atomic_x86_64_sysv.S`

```
.intel_syntax noprefix

/* atomic_x86_64_sysv.S
   uint32_t atomic_fetch_add_u32(uint32_t* p, uint32_t delta)
   p     -> rdi
   delta -> esi
   return old -> eax
*/

.text
.globl atomic_fetch_add_u32
```

```
atomic_fetch_add_u32:
    /* Place delta in EAX because XADD returns old value in the reg operand */
    mov     eax, esi

    /* Atomic: old = *p; *p += delta; eax = old */
    lock xadd DWORD PTR [rdi], eax

    ret
```

Why this is minimal: one instruction provides atomicity and a strong ordering effect typical of `LOCK`-prefixed RMWs.

39.4 AArch64: `LDAXR/STLXR` exclusive loop

AArch64 does not generally provide a single instruction that is both universally available and directly identical to x86 `LOCK XADD` for all sizes and orderings (some extensions add LSE atomics, but the portable baseline is the exclusive loop).
The classic pattern:

- `LDAXR` loads a value and marks the address as *exclusive* (acquire ordering).
- We compute the new value in registers.
- `STLXR` attempts to store the new value if the exclusive monitor is still valid (release ordering).
- If the store fails, we retry (another core modified the location).

AArch64 Linux ABI mapping

- p in `x0`

- `delta` in `w1`
- return old in `w0`

Assembly: `atomic_aarch64.S`

```
/* atomic_aarch64.S
   uint32_t atomic_fetch_add_u32(uint32_t* p, uint32_t delta)
   p     -> x0
   delta -> w1
   return old -> w0
*/

.text
.global atomic_fetch_add_u32
atomic_fetch_add_u32:
    /* w0 will hold old; w2 will hold new; w3 will hold status */
1:
    ldaxr   w0, [x0]          /* old = *p (acquire) */
    add     w2, w0, w1         /* new = old + delta */
    stlxr   w3, w2, [x0]       /* try store new (release), w3=0 on success */
    cbnz    w3, 1b             /* retry if store failed */
    ret
```

Key idea: The loop is the price you pay for a general atomic RMW without assuming additional architectural extensions. The ordering is explicit: acquire on load, release on store.

39.5 RISC-V: `AMO` atomic increment

RISC-V provides **AMO** (Atomic Memory Operations) for performing atomic RMW updates. The instruction `amoadd.w` atomically adds a 32-bit value to memory and returns the original

value.

Ordering on RISC-V is explicit. AMOs can carry ordering qualifiers:

- `.aq` requests acquire ordering,
- `.rl` requests release ordering,
- `.aqrl` requests both.

RISC-V 64 Linux ABI mapping

- `p` in `a0`
- `delta` in `a1` (we use `w` operation for 32-bit)
- return old in `a0`

Assembly: `atomic_riscv64.S`

```
/* atomic_riscv64.S
   uint32_t atomic_fetch_add_u32(uint32_t* p, uint32_t delta)
   p     -> a0
   delta -> a1
   return old -> a0
*/

.text
.globl atomic_fetch_add_u32
atomic_fetch_add_u32:
    /* amoadd.w.aqrl rd, rs2, (rs1)
       Atomically:
         t = *(uint32_t*)p
```

```
      *(uint32_t*)p = t + delta
    Return old in rd.
*/
amoadd.w.aqrl a0, a1, (a0)
ret
```

Why this is minimal: one instruction performs the atomic RMW and returns the previous value, with explicit acquire+release ordering when requested.

39.6 A compact cross-ISA comparison

Aspect	x86	AArch64	RISC-V
Atomic mechanism	LOCKed RMW	Exclusive monitor loop	AMO RMW
Typical primitive	`lock xadd`	`ldaxr/stlxr`	`amoadd.w`
Retry loop needed	No	Yes	No
Ordering control	implicit strong	explicit via acquire/release ops	explicit via `.aq/.rl`
Return old value	Yes	Yes	Yes

39.7 Build notes (minimal, practical)

Linux x86-64 (SysV) with GCC

```
gcc -O2 -c atomic_x86_64_sysv.S -o atomic_x86_64_sysv.o
```

Linux AArch64 (native or cross)

```
aarch64-linux-gnu-gcc -O2 -c atomic_aarch64.S -o atomic_aarch64.o
```

Linux RISC-V 64 (native or cross)

```
riscv64-linux-gnu-gcc -O2 -c atomic_riscv64.S -o atomic_riscv64.o
```

39.8 Common pitfalls (what breaks correctness)

Pitfall 1: assuming all atomic increments imply the same ordering

Atomicity alone prevents lost updates, but does not automatically provide the ordering you might need for building locks or publishing data. On ARM and RISC-V, you must choose ordering explicitly.

Pitfall 2: forgetting the retry loop on AArch64

With exclusives, the store can fail under contention. If you do not loop, increments will be lost.

Pitfall 3: mixing 32-bit and 64-bit operations

If the shared object is 32-bit, use 32-bit atomic instructions consistently across all code paths. Mixing sizes can break alignment assumptions or create partial updates depending on the platform.

Pitfall 4: using unaligned addresses

Atomic operations generally require naturally aligned operands for the chosen width. Misalignment can trap or degrade correctness/performance.

39.9 Windows-only Rust examples (as requested)

You requested that all Rust code in this booklet target Windows OS. Below are Windows-focused Rust examples that implement the same semantic operation (`fetch_add`) using:

- the standard library atomics (portable and correct),
- an optional x86-64 inline assembly version using `lock xadd` for educational parity.

Rust (Windows): portable atomic increment using `AtomicU32`

```rust
#![cfg(windows)]

use std::sync::atomic::{AtomicU32, Ordering};

fn atomic_fetch_add_u32(p: &AtomicU32, delta: u32) -> u32 {
    /* Strong default ordering for demonstration */
    p.fetch_add(delta, Ordering::AcqRel)
}

fn main() {
    let x = AtomicU32::new(100);

    let old = atomic_fetch_add_u32(&x, 5);
    let new = x.load(Ordering::Acquire);

    println!("old = {}, new = {}", old, new);
}
```

Rust (Windows x86-64): educational inline asm `lock xadd`

This version is only meaningful on x86-64 targets and is included to mirror the ISA-level primitive from the x86 section. It uses a raw pointer to a `u32`. The pointer must be valid and naturally aligned.

```
#![cfg(all(windows, target_arch = "x86_64"))]

use core::arch::asm;

unsafe fn atomic_fetch_add_u32_lock_xadd(p: *mut u32, delta: u32) -> u32 {
    let mut reg = delta;
    /* After lock xadd [mem], reg:
       - memory is incremented by reg
       - reg receives old memory value
    */
    asm!(
        "lock xadd dword ptr [{ptr}], {reg:e}",
        ptr = in(reg) p,
        reg = inout(reg) reg,
        options(nostack, preserves_flags),
    );
    reg
}

fn main() {
    let mut x: u32 = 100;
    let p: *mut u32 = &mut x;

    unsafe {
```

```
        let old = atomic_fetch_add_u32_lock_xadd(p, 7);
        println!("old = {}, new = {}", old, x);
    }
}
```

For production Rust on Windows, prefer `AtomicU32::fetch_add` unless you are implementing a specialized runtime or experimenting with ABI-level primitives.

39.10 Chapter summary

- x86 uses `LOCK`-prefixed RMW instructions (e.g., `lock xadd`) to perform atomic increments without loops.
- AArch64 commonly uses an exclusive loop; `LDAXR/STLXR` expresses acquire+release directly.
- RISC-V uses `AMO`; ordering is explicit via `.aq/.rl` qualifiers.
- The same high-level primitive (`fetch_add`) becomes three different low-level mechanisms, but the semantic contract remains identical.
- Rust on Windows should prefer `AtomicU32`, with inline asm used only for controlled educational parity.

Part VIII

ENGINEER REFERENCE

ABI Quick Tables

This chapter is a compact engineer reference: quick tables you can use while reading, writing, or debugging assembly and low-level code across the major 64-bit platforms covered in this volume. Scope (64-bit):

- x86-64 System V ABI (Linux and most Unix-like systems),
- x86-64 Microsoft ABI (Windows),
- AArch64 ABI (Linux),
- RISC-V 64 ABI (Linux).

All tables focus on what matters most in practice:

- argument passing registers,
- return registers,
- stack alignment rules,
- caller-saved and callee-saved register sets,
- syscall ABI quick maps (where applicable),
- minimal notes for writing correct mixed-language boundaries.

40.1 Legend and conventions

- **GPR**: General-purpose registers.
- **FP/SIMD**: Floating-point / vector registers.
- “volatile” means caller-saved (callee may clobber).
- “non-volatile” means callee-saved (callee must preserve).
- “stack aligned at call boundary” means alignment requirement *when executing a* `call/bl/jalr`.

40.2 x86-64 System V ABI (Linux): quick tables

Argument registers

Integer / pointer args (in order)	`RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9`
Floating args (in order)	`XMM0`, `XMM1`, `XMM2`, `XMM3`, `XMM4`, `XMM5`, `XMM6`, `XMM7`

Return registers

Integer / pointer return	`RAX`
Floating return	`XMM0`
128-bit integer (when used)	`RDX:RAX` (high:low)

Stack alignment and red zone

- Stack alignment: **16-byte aligned** at call boundary.

- Red zone: **128 bytes below RSP** may be used by leaf functions without adjusting RSP (Linux SysV feature).

Callee-saved (non-volatile) vs caller-saved (volatile)

Callee-saved (must preserve)	`RBX`, `RBP`, `R12`, `R13`, `R14`, `R15`
Caller-saved (may clobber)	`RAX`, `RCX`, `RDX`, `RSI`, `RDI`, `R8`, `R9`, `R10`, `R11`

Syscall ABI (Linux x86-64)

Syscall instruction	`syscall`
Syscall number	`RAX`
Args 1–6	`RDI`, `RSI`, `RDX`, `R10`, `R8`, `R9`
Return	`RAX`
Clobbered	`RCX`, `R11`

40.3 x86-64 Microsoft ABI (Windows): quick tables

Argument registers

Integer / pointer args (first 4)	`RCX`, `RDX`, `R8`, `R9`
Floating args (first 4)	`XMM0`, `XMM1`, `XMM2`, `XMM3`
Additional args	passed on stack (right-to-left)

Return registers

Integer / pointer return	`RAX`
Floating return	`XMM0`

Stack alignment and shadow space

- Stack alignment: **16-byte aligned** at call boundary.
- Shadow space: caller must reserve **32 bytes** of home space for the callee before each call.
- No red zone on Windows x64.

Non-volatile vs volatile registers (Windows x64)

Non-volatile (callee-saved)	`RBX`, `RBP`, `RDI`, `RSI`, `R12`, `R13`, `R14`, `R15`
Volatile (caller-saved)	`RAX`, `RCX`, `RDX`, `R8`, `R9`, `R10`, `R11`

FP/SIMD registers:

- `XMM0–XMM5` are volatile.
- `XMM6–XMM15` are non-volatile (must be preserved by callee) in the Windows x64 ABI.

40.4 AArch64 ABI (Linux): quick tables

Argument registers

Integer / pointer args	`x0–x7`
Floating args	`v0–v7` (scalar float in `s0..s7` lanes)

Return registers

Integer / pointer return	`x0`
Floating return	`v0`

Stack alignment

- Stack pointer must be **16-byte aligned** at public interfaces and call boundaries.

Callee-saved vs caller-saved (AArch64)

Callee-saved (must preserve)	`x19`–`x28`, `x29` (FP), `x30` (LR preserved by convention when needed)
Caller-saved (may clobber)	`x0`–`x18` (except platform-reserved roles), `x30` used as LR across calls

FP/SIMD registers:

- `v0`–`v7` are argument/return registers (caller-saved).
- `v8`–`v15` are typically callee-saved in common ABI conventions.

Syscall ABI (Linux AArch64)

Syscall instruction	`svc #0`
Syscall number	`x8`
Args 1–6	`x0`–`x5`
Return	`x0`

40.5 RISC-V 64 ABI (Linux): quick tables

Argument registers

Integer / pointer args	`a0–a7`
Floating args	`fa0–fa7`

Return registers

Integer / pointer return	`a0` (and `a1` for 128-bit returns)
Floating return	`fa0`

Stack alignment

- Stack pointer alignment: **16 bytes** at call boundaries in common 64-bit Linux ABIs.

Callee-saved vs caller-saved (RISC-V)

Callee-saved (must preserve)	`s0–s11` (with `s0` often used as frame pointer)
Caller-saved (may clobber)	`a0–a7`, `t0–t6`, `ra`

Syscall ABI (Linux RISC-V)

Syscall instruction	`ecall`
Syscall number	`a7`
Args 1–6	`a0–a5`
Return	`a0`

40.6 Quick reference: argument passing examples (by type)

Example signature

```
int demo(int a, long b, void* p, float x, double y);
```

Where the first arguments go (typical case):

Arg	x86-64 SysV	AArch64	RISC-V 64
a (int)	`EDI`	`w0`	`a0`
b (long)	`RSI`	`x1`	`a1`
p (void*)	`RDX`	`x2`	`a2`
x (float)	`XMM0`	`s0`	`fa0`
y (double)	`XMM1`	`d1`	`fa1`

Windows x64 differs in integer/pointer order:

Windows x64	`a` in `ECX`, `b` in `RDX`, `p` in `R8`, `x` in `XMM3` or `XMM?` depends on position, `y` in `XMM?`

Practical rule for Windows x64: the first four arguments (regardless of type class) map to the four slots (RCX/RDX/R8/R9 and XMM0–XMM3 for floating). The exact mapping depends on the argument positions and whether each position is floating or integer/pointer.

40.7 Engineer checklist: writing correct ABI boundaries

- Confirm which ABI you are targeting (SysV vs Windows x64 vs AArch64 vs RISC-V).
- Confirm which register class each argument uses (integer/pointer vs FP/SIMD).
- Confirm stack alignment at call boundary (almost always 16 bytes for modern 64-bit ABIs).
- On Windows x64: always allocate 32-byte shadow space before calls.
- Preserve callee-saved registers if you modify them.
- For syscalls: use the syscall ABI registers, not the function-call ABI registers.

40.8 Windows-only Rust reference (as requested)

You requested that all Rust code in this booklet be for Windows OS. The following is a compact reference showing:

- how Rust `extern "C"` matches the platform ABI,
- how to declare Win32 APIs with `extern "system"`,
- how to safely contain `unsafe` for OS calls.

Rust: `extern "C"` ABI and a mixed-type call

```
#![cfg(windows)]

#[no_mangle]
```

```
pub extern "C" fn demo(a: i32, b: i64, p: *mut core::ffi::c_void, x: f32, y: f64) ->
↪  i32 {
    let _ = (a, b, p, x, y);
    0
}

fn main() {
    let mut v: i32 = 0;
    let p = &mut v as *mut _ as *mut core::ffi::c_void;
    let r = demo(1, 2, p, 3.5, 7.25);
    println!("r = {}", r);
}
```

Rust: Win32 console output with `extern "system"` (no crates)

```
#![cfg(windows)]

use core::ffi::c_void;

type HANDLE = *mut c_void;
type DWORD = u32;
type BOOL = i32;

const STD_OUTPUT_HANDLE: DWORD = 0xFFFF_FFF5; /* (DWORD)-11 */

#[link(name = "kernel32")]
extern "system" {
    fn GetStdHandle(nStdHandle: DWORD) -> HANDLE;

    fn WriteFile(
```

```
        hFile: HANDLE,
        lpBuffer: *const c_void,
        nNumberOfBytesToWrite: DWORD,
        lpNumberOfBytesWritten: *mut DWORD,
        lpOverlapped: *mut c_void,
    ) -> BOOL;
}

fn main() {
    let msg = b"ABI quick tables: Rust Win32 output\r\n";
    unsafe {
        let h = GetStdHandle(STD_OUTPUT_HANDLE);
        let mut written: DWORD = 0;
        let _ = WriteFile(
            h,
            msg.as_ptr() as *const c_void,
            msg.len() as DWORD,
            &mut written as *mut DWORD,
            core::ptr::null_mut(),
        );
    }
}
```

40.9 Chapter summary

- This chapter collected the most-used ABI facts into quick tables for engineers.
- System V and Windows x64 differ primarily in integer argument registers, shadow space, and SIMD preservation rules.

- AArch64 and RISC-V provide clean register-based ABIs with explicit syscall conventions.
- Keep this chapter nearby when debugging call boundaries, mixed-language linking, and syscall glue.

Register Quick Reference

This chapter is a dense engineer reference for the most-used registers and register classes across the ISAs in this volume:

- x86-64 (general-purpose + SIMD),
- AArch64 (general-purpose + SIMD/FP),
- RISC-V 64 (general-purpose + floating-point).

It is designed for rapid lookup while writing assembly, reading compiler output, debugging ABIs, or interpreting crash dumps. The focus is on what an engineer most frequently needs:

- register names and sizes,
- subregister relationships (where they exist),
- architectural roles (SP, FP, LR, PC, flags),
- typical calling-convention roles and common conventions,
- practical examples that show correct usage patterns.

All comments in assembly examples use `/* ... */`.

41.1 x86-64 register set (GPR)

Naming and sizes

x86-64 general-purpose registers are 64-bit, with legacy subregisters:

- 64-bit: `RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8–R15`
- 32-bit: `EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, R8D–R15D`
- 16-bit: `AX, BX, CX, DX, SI, DI, BP, SP, R8W–R15W`
- 8-bit: `AL, BL, CL, DL` plus extended forms `SIL, DIL, BPL, SPL, R8B–R15B`

Important practical rule: writing a 32-bit subregister (e.g., `EAX`) *zeros* the upper 32 bits of the corresponding 64-bit register (`RAX`). This is a key performance and correctness detail.

Special roles

- `RSP`: stack pointer.
- `RBP`: optional frame pointer (depends on compiler settings).
- `RIP`: instruction pointer (not directly writable; used for RIP-relative addressing).
- `RFLAGS`: status flags register (condition codes).

Flags (common subset)

- `ZF` (zero flag): result was zero.
- `SF` (sign flag): sign bit of result.
- `CF` (carry flag): unsigned carry/borrow.

- `OF` (overflow flag): signed overflow.

RIP-relative addressing (most common pattern)

```
.intel_syntax noprefix

/* Load address of a global symbol using RIP-relative LEA */
lea rax, [rip !+! some_global]  /* rax = &some_global */
```

41.2 x86-64 SIMD/FP registers (XMM/YMM/ZMM)

Modern x86 uses vector registers for floating-point and SIMD:

- `XMM0–XMM15`: 128-bit (SSE) registers.
- `YMM0–YMM15`: 256-bit (AVX) registers (overlap with XMM).
- `ZMM0–ZMM31`: 512-bit (AVX-512) registers (when available).

Scalar floating-point values:

- `float` often uses `XMMn` low 32-bit lane.
- `double` uses `XMMn` low 64-bit lane.

Scalar float multiply example

```
.intel_syntax noprefix

/* Multiply scalar floats in XMM registers */
mulss xmm0, xmm1  /* xmm0 = xmm0 * xmm1 (low 32-bit lane) */
```

41.3 AArch64 register set (GPR)

Naming and sizes

AArch64 general-purpose registers:

- `x0–x30`: 64-bit registers.
- `w0–w30`: 32-bit low halves of `x0–x30`.
- `sp`: stack pointer (separate name, not `x31` in all contexts).
- `xzr/wzr`: zero register (reads as zero, writes are discarded).

Special roles

- `pc`: program counter (not a general register).
- `sp`: stack pointer.
- `x29`: frame pointer (FP) by convention when used.
- `x30`: link register (LR), holds return address after `bl`.

Condition flags

AArch64 uses condition flags in `PSTATE`:

- `N, Z, C, V` correspond to negative, zero, carry, overflow.

PC-relative addressing: `adr` and `adrp`

Two common forms:

- `adr`: compute address within a limited range.
- `adrp+add`: compute page-based address for larger ranges.

```
/* Small example: address of a nearby label */
adr x0, msg          /* x0 = &msg (within range) */

/* Typical large/PIE form: page + offset */
adrp x0, msg
add  x0, x0, :lo12:msg
```

41.4 AArch64 SIMD/FP registers (V registers)

AArch64 provides 32 vector registers:

- `v0-v31`: 128-bit each.

They can be viewed with lane types:

- `s0-s31`: 32-bit scalar float lanes (low lane of `vN`).
- `d0-d31`: 64-bit scalar double lanes.

Scalar float arithmetic example

```
/* fmul s0, s0, s1  => s0 = s0 * s1 */
fmul s0, s0, s1
```

41.5 RISC-V 64 register set (GPR)

Naming and ABI roles

RISC-V registers have numeric names `x0–x31` and conventional ABI aliases. Engineers typically read assembly in ABI names:

Numeric	ABI name	Typical role
`x0`	`zero`	constant zero (reads 0, writes ignored)
`x1`	`ra`	return address
`x2`	`sp`	stack pointer
`x3`	`gp`	global pointer
`x4`	`tp`	thread pointer
`x5–x7`	`t0–t2`	temporaries (caller-saved)
`x8`	`s0/fp`	saved reg or frame pointer
`x9`	`s1`	saved reg
`x10–x17`	`a0–a7`	args/return (caller-saved)
`x18–x27`	`s2–s11`	saved regs (callee-saved)
`x28–x31`	`t3–t6`	temporaries (caller-saved)

Key practical rules

- `zero` is always 0. Use it to clear registers efficiently.
- `ra` must be saved by a function that calls other functions (non-leaf), otherwise it will be overwritten.
- `sp` alignment is typically 16 bytes at call boundaries in 64-bit Linux ABIs.

Function call skeleton (leaf vs non-leaf)

```
/* Leaf function: does not call others; may not need to save ra */
.globl leaf
leaf:
    addi a0, a0, 1
    ret

/* Non-leaf function: calls others; must preserve ra (and any used s-regs) */
.globl nonleaf
nonleaf:
    addi sp, sp, -16
    sd   ra, 8(sp)        /* save return address */
    sd   s0, 0(sp)        /* save s0/fp if used */
    call leaf
    ld   ra, 8(sp)
    ld   s0, 0(sp)
    addi sp, sp, 16
    ret
```

41.6 RISC-V floating-point registers (FPR)

When the floating-point extension is present and the ABI uses FP registers:

- `f0-f31` are floating-point registers.
- ABI aliases commonly used: `fa0-fa7` for FP args/return, `ft0-ft11` temporaries, `fs0-fs11` saved.

Scalar float add example

```
/* fa0 = fa0 + fa1 */
fadd.s fa0, fa0, fa1
```

41.7 Quick cross-architecture mapping tables

Core special registers

Role	x86-64	AArch64	RISC-V
Stack pointer	`RSP`	`SP`	`sp`
Frame pointer	`RBP` (optional)	`x29` (FP by convention)	`s0/fp`
Return address	stack via `call/ret`	`x30` (LR)	`ra`
Zero register	none	`xzr/wzr`	`zero`
Flags	`RFLAGS`	`PSTATE NZCV`	none (explicit compare/branch)

Argument and return registers (first slots)

Kind	x86-64 SysV	AArch64	RISC-V 64
Int arg #1	RDI	x0	a0
Int arg #2	RSI	x1	a1
Ptr arg #1	RDI	x0	a0
Return int/ptr	RAX	x0	a0
Float arg #1	XMM0	s0/v0	fa0
Return float	XMM0	s0/v0	fa0

41.8 Windows-only Rust examples (as requested)

You requested that all Rust code in this booklet be for Windows OS. Below are minimal Windows-focused Rust snippets that help interpret register-level realities while staying in correct, supported user-mode APIs.

Rust: reading a stable Windows ABI boundary

Use `extern "system"` for Win32 APIs to match the Windows x64 calling convention.

```
#![cfg(windows)]

use core::ffi::c_void;

type HANDLE = *mut c_void;
type DWORD = u32;

const STD_OUTPUT_HANDLE: DWORD = 0xFFFF_FFF5; /* (DWORD)-11 */

#[link(name = "kernel32")]
```

```
extern "system" {
    fn GetStdHandle(nStdHandle: DWORD) -> HANDLE;
}

fn main() {
    unsafe {
        let h = GetStdHandle(STD_OUTPUT_HANDLE);
        let _ = h;
    }
}
```

Rust: minimal inline assembly example (Windows x86-64)

This is a safe, local computation example that demonstrates register constraints without touching OS syscalls.

```
#![cfg(all(windows, target_arch = "x86_64"))]

use core::arch::asm;

fn add3(a: u64, b: u64, c: u64) -> u64 {
    let out: u64;
    unsafe {
        asm!(
            "lea {out}, [{a} + {b}]",
            "add {out}, {c}",
            a = in(reg) a,
            b = in(reg) b,
            c = in(reg) c,
            out = out(reg) out,
```

```
            options(nomem, nostack, preserves_flags),
        );
    }
    out
}

fn main() {
    let x = add3(10, 20, 12);
    assert_eq!(x, 42);
}
```

41.9 Chapter summary

- x86-64 has layered subregister naming and a unified SIMD register file (XMM/YMM/ZMM).
- AArch64 has clean `xN`/`wN` pairing, a dedicated zero register, and `vN` FP/SIMD registers.
- RISC-V uses numeric registers with ABI aliases; `zero`, `ra`, and `sp` are central to reading code quickly.
- For engineers, the fastest path is to memorize the special-register roles and the argument/return register slots for your target ABI.

Memory Ordering Quick Reference

This chapter is a compact engineer reference for **memory ordering** in modern multiprocessors, with ISA-level primitives you will actually see in compiler output and hand-written assembly:

- x86/x86-64: `LOCK`ed instructions, fences `LFENCE/SFENCE/MFENCE`
- AArch64: acquire/release loads and stores (`LDAR/STLR`), exclusives (`LDAXR/STLXR`), full barriers (`DMB`)
- RISC-V: AMOs with `.aq/.rl`, and `FENCE` for explicit ordering under RVWMO

The goal is not to re-teach the entire memory model. The goal is to provide:

- a precise mental model for **what can reorder** (compiler and CPU),
- minimal, comparable patterns (message passing, atomic increment, spin lock edges),
- quick tables that map **intent** to **instructions**.

42.1 Two independent reorderers: compiler and CPU

When debugging or writing concurrency code, always separate two layers:

Compiler reordering

- The compiler may reorder loads/stores as long as single-thread semantics appear unchanged.
- Atomics and fences restrict compiler motion and force specific instruction choices.

CPU reordering

- Even if the compiler emits instructions in a given order, the CPU and memory system may make other cores observe them in a different order.
- Barriers (and certain atomic instructions) restrict what can become visible out of order.

Therefore, a correct synchronization plan must typically address **both**.

42.2 A minimal vocabulary: atomicity vs ordering

Atomic operations provide two different properties:

Atomicity

A read-modify-write update (like increment) does not lose updates under contention.

Ordering

Which other loads/stores can be observed before/after an operation by other threads.
For ordering, engineers most frequently use these intent levels:

- **Relaxed:** atomicity only, minimal ordering.
- **Acquire:** subsequent reads/writes stay after the acquire-load.

- **Release:** prior reads/writes stay before the release-store.
- **AcqRel:** both acquire and release effects (often used for RMWs).
- **SeqCst:** a single global total order for participating operations (strongest, often slowest on weakly-ordered ISAs).

42.3 x86/x86-64 quick reference (TSO-style behavior)

x86 is relatively strongly ordered compared to ARM and RISC-V. In practice:

- Loads are not reordered with other loads.
- Stores are not reordered with other stores.
- The main allowed hardware reordering is **Store → Load** (a later load can be observed before an earlier store becomes visible to other cores).
- `LOCK`-prefixed read-modify-write instructions are atomic and act as strong ordering points for surrounding memory operations in typical cached WB memory.

Primitives

- **Atomic RMW:** `lock xadd`, `lock add`, `lock cmpxchg`, `lock inc` (avoid `inc` for flags differences in some patterns, prefer `add` or `xadd`).
- **Full fence:** `mfence`
- **Load fence:** `lfence` (commonly used for ordering loads and also as a speculation barrier in some contexts)
- **Store fence:** `sfence` (often relevant for streaming/non-temporal stores)

Minimal message-passing pattern (release/acquire intent)

Intent:

- Producer writes data, then publishes a flag.
- Consumer reads the flag with acquire, then reads data.

On x86, the CPU usually does not need explicit fences for this pattern if you use proper atomic operations for the flag, because ordering is already strong enough; the compiler barrier aspect is the bigger concern.

```
struct Msg { int data; };
_Atomic int flag = 0;
struct Msg m;

void producer(void) {
    m.data = 42;
    __atomic_store_n(&flag, 1, __ATOMIC_RELEASE);
}

int consumer(void) {
    if (__atomic_load_n(&flag, __ATOMIC_ACQUIRE)) {
        return m.data;
    }
    return -1;
}
```

Minimal atomic increment: x86-64 (`LOCK XADD`)

```
.intel_syntax noprefix
```

```
/* uint32_t fetch_add_u32(uint32_t* p, uint32_t delta)
   SysV: p->rdi, delta->esi, return old->eax */
.text
.globl fetch_add_u32
fetch_add_u32:
    mov eax, esi
    lock xadd DWORD PTR [rdi], eax
    ret
```

When to actually use `mfence/sfence/lfence`

Typical engineer guidance:

- Use `LOCK`-RMW operations for atomic counters/locks instead of manual fences.
- Use `mfence` when you must prevent Store→Load reordering without using a locked operation (rare in normal lock-free code, more common with special memory types or non-temporal patterns).
- Use `sfence` with non-temporal (streaming) stores when you need global visibility completion.
- Use `lfence` to prevent certain load reorderings and for controlled serialization of load operations.

42.4 AArch64 quick reference (weakly ordered, explicit tools)

AArch64 is more relaxed: many reorderings are allowed unless you use ordering primitives. Practical takeaway:

- You often need **acquire/release** operations (or barriers) to implement correct lock-free message passing.
- The ISA provides **one-way ordering** via acquire-load / release-store variants and **full barriers** via `DMB`.

Acquire/Release loads and stores

- `LDAR` (load-acquire): ensures subsequent memory operations do not move before the load.
- `STLR` (store-release): ensures prior memory operations do not move after the store.

Exclusive pair for RMW and locks

- `LDAXR` (load-acquire exclusive)
- `STLXR` (store-release exclusive)

These form the standard portable atomic RMW loop (unless you assume LSE atomics).

Full barrier

`DMB ish` is a common full system-scope barrier used for inter-core synchronization on normal memory.

Minimal message passing (AArch64)

Producer: store data then release-store the flag. Consumer: acquire-load the flag then load data.

```
/* AArch64 conceptual pattern */
    /* producer */
    str  w0, [xData]          /* data = 42 */
```

```
    stlr w1, [xFlag]          /* flag = 1 (release) */

    /* consumer */
    ldar w2, [xFlag]          /* r = flag (acquire) */
    cbz  w2, not_ready
    ldr  w3, [xData]          /* read published data */
```

Minimal atomic increment: `LDAXR/STLXR` loop

This example returns the old value (fetch-add) for a 32-bit counter.

```
/* uint32_t fetch_add_u32(uint32_t* p, uint32_t delta)
   AArch64: p->x0, delta->w1, return old->w0 */
.text
.global fetch_add_u32
fetch_add_u32:
1:
    ldaxr w0, [x0]            /* old = *p (acquire) */
    add   w2, w0, w1          /* new = old + delta */
    stlxr w3, w2, [x0]        /* attempt store (release), w3=0 success */
    cbnz  w3, 1b              /* retry on contention */
    ret
```

Barrier selection cheat-sheet (AArch64)

- **Message passing:** `STLR` on publish, `LDAR` on consume.
- **Atomic RMW with ordering:** `LDAXR/STLXR` loop (AcqRel shape).
- **Stronger global ordering (rare, heavy):** `DMB ish` around critical edges.

42.5 RISC-V quick reference (RVWMO, explicit ordering)

RISC-V is designed with an explicit, flexible memory model (RVWMO). In practice:

- Use **AMOs** for atomic RMW updates when the A extension is available.
- Add `.aq`, `.rl`, or `.aqrl` to AMOs (and LR/SC) to request acquire/release ordering.
- Use **FENCE** when you need ordering not provided by the atomic instruction form you chose.

The two key tools

AMO ordering bits

- `.aq`: acquire ordering
- `.rl`: release ordering
- `.aqrl`: both

FENCE

`fence pred, succ` orders certain classes of memory operations:

- Common engineer pattern: acquire fence `fence r, rw`
- Common engineer pattern: release fence `fence rw, w`
- Full-ish ordering often expressed as `fence rw, rw` when needed

Minimal atomic increment: `amoadd.w.aqrl`

```
/* uint32_t fetch_add_u32(uint32_t* p, uint32_t delta)
   RV64: p->a0, delta->a1, return old->a0 */
.text
.globl fetch_add_u32
fetch_add_u32:
    amoadd.w.aqrl a0, a1, (a0)    /* old in a0, memory += delta */
    ret
```

Message passing (RISC-V) with release store and acquire load

Two common shapes:

Shape A: AMO on the flag (acquire/release via `.aq/.rl`)

```
/* Producer: data then publish flag with release */
sw   t0, 0(xData)               /* data = 42 */
amoswap.w.rl x0, t1, (xFlag)    /* flag = 1 with release (discard old) */

/* Consumer: acquire flag then read data */
amoswap.w.aq t2, x0, (xFlag)    /* read flag with acquire (swap with 0 is one trick)
↪  */
beqz t2, not_ready
lw   t3, 0(xData)
```

Shape B: plain store/load plus fences

```
/* Producer */
sw    t0, 0(xData)              /* data = 42 */
fence rw, w                     /* release fence */
```

```
sw    t1, 0(xFlag)              /* flag = 1 */

/* Consumer */
lw    t2, 0(xFlag)              /* read flag */
beqz  t2, not_ready
fence r, rw                     /* acquire fence */
lw    t3, 0(xData)              /* read published data */
```

For minimal code and performance, prefer AMO/LRSC with `.aq/.rl` when appropriate. Use fences when you need ordering outside what the atomic itself provides.

42.6 Quick tables: intent to instruction choices

Acquire-load and release-store

Intent	x86/x86-64	AArch64	RISC-V
Release store	typically plain store + compiler barrier via atomic store	`STLR`	store + `fence rw,w` or AMO with `.rl`
Acquire load	typically plain load + compiler barrier via atomic load	`LDAR`	load + `fence r,rw` or AMO with `.aq`

Atomic increment (fetch-add) with strong ordering intent

Primitive	x86/x86-64	AArch64	RISC-V
fetch_add(u32)	`lock xadd [mem], reg`	`ldaxr/stlxr` loop	`amoadd.w.aqrl`

42.7 Common engineer pitfalls

Pitfall 1: assuming "atomic" implies ordering strong enough for publication

Atomicity prevents torn/lost updates, but **publication** requires ordering. Use release on publish and acquire on consume.

Pitfall 2: using AArch64 exclusive loop without choosing ordering

`LDXR/STXR` without acquire/release may be insufficient for lock edges. Prefer `LDAXR/STLXR` for an AcqRel-shaped RMW baseline.

Pitfall 3: using RISC-V AMO without `.aq/.rl` when you needed it

RISC-V makes ordering explicit. If you intended acquire/release, encode it or add fences.

Pitfall 4: mixing compiler barriers with CPU barriers incorrectly

A CPU fence does not automatically prevent the compiler from reordering independent operations around it unless the compiler treats it as a barrier. Use language-level atomics or inline-asm marked appropriately.

42.8 Windows-only Rust reference (as requested)

You requested that all Rust code in this booklet target Windows OS. The examples below provide a clean memory-ordering reference using Rust's standard atomics. They are not ISA-specific assembly, but they match the same intent levels used throughout this chapter.

Rust: message passing (Release/Acquire)

```rust
#![cfg(windows)]

use std::sync::atomic::{AtomicI32, AtomicBool, Ordering};
use std::sync::Arc;
use std::thread;

struct Shared {
    data: AtomicI32,
    ready: AtomicBool,
}

fn main() {
    let s = Arc::new(Shared {
        data: AtomicI32::new(0),
        ready: AtomicBool::new(false),
    });

    let p = s.clone();
    let prod = thread::spawn(move || {
        p.data.store(42, Ordering::Relaxed);
        p.ready.store(true, Ordering::Release);
```

```
    });

    let c = s.clone();
    let cons = thread::spawn(move || {
        while !c.ready.load(Ordering::Acquire) {}
        let v = c.data.load(Ordering::Relaxed);
        println!("v = {}", v);
    });

    let _ = prod.join();
    let _ = cons.join();
}
```

Rust: atomic increment (Relaxed vs AcqRel vs SeqCst)

```
#![cfg(windows)]

use std::sync::atomic::{AtomicU32, Ordering};

fn main() {
    let x = AtomicU32::new(0);

    let old_relaxed = x.fetch_add(1, Ordering::Relaxed);
    let old_acqrel  = x.fetch_add(1, Ordering::AcqRel);
    let old_seqcst  = x.fetch_add(1, Ordering::SeqCst);

    let now = x.load(Ordering::Acquire);
    println!("old_relaxed={}, old_acqrel={}, old_seqcst={}, now={}",
             old_relaxed, old_acqrel, old_seqcst, now);
```

```
}
```

Rust: a minimal “compiler barrier” style fence

Rust exposes `fence` and `compiler_fence`. `compiler_fence` constrains the compiler but may not emit CPU fences on some platforms.

```rust
#![cfg(windows)]

use std::sync::atomic::{compiler_fence, fence, Ordering};

fn main() {
    compiler_fence(Ordering::SeqCst);
    fence(Ordering::SeqCst);
}
```

42.9 Chapter summary

- Always separate compiler reordering from CPU reordering; correct concurrency requires controlling both.
- x86 is relatively strongly ordered; `LOCK`ed RMW operations are the common workhorse for atomic updates and strong ordering points.
- AArch64 requires explicit ordering more often; `LDAR/STLR` and `LDAXR/STLXR` are the core primitives, with `DMB` for stronger barriers.
- RISC-V makes ordering explicit via `.aq/.rl` on AMOs/LRSC and via `FENCE`.
- Rust on Windows provides a clean, portable way to express Relaxed/Acquire/Release/AcqRel/SeqCst intent using standard atomics.

Debugger Command Cheat Sheet

This chapter is a compact engineer reference: the debugger commands you reach for every day when working with assembly, ABIs, syscalls, crashes, and mixed-language binaries.
Scope:

- Linux: `gdb` (primary), minimal `lldb` notes
- Windows: WinDbg (`windbg`/`cdb`) and Visual Studio debugger essentials
- Cross-architecture: how to display registers, stack, call frames, and disassembly for x86-64, AArch64, and RISC-V
- Rust on Windows: how to produce usable symbols and step through code

This is a cheat sheet. Commands are grouped by task and designed to be copy-paste friendly.

43.1 Symbols and builds that debuggers can use

Minimal debug build flags (C/C++)

```
/* Linux GCC/Clang */
gcc -g -O0 -fno-omit-frame-pointer main.c -o app
```

```
/* Windows MSVC (Developer Command Prompt) */
cl /Zi /Od /EHsc main.cpp
```

Minimal debug build flags (Rust on Windows)

Rust produces debuggable PDBs on Windows in debug builds.

```
/* Debug build (default) */
cargo build

/* Better stepping experience for optimization-sensitive cases */
cargo build -Z build-std=std,panic_abort -Z
↪  build-std-features=panic_immediate_abort
```

If you want debug symbols in a release-like build:

```
/* Cargo.toml */
[profile.release]
debug = true
```

43.2 GDB essentials (Linux)

Start, args, and environment

```
gdb ./app
(gdb) run
(gdb) run arg1 arg2
(gdb) set args arg1 arg2
(gdb) show args
(gdb) set env VAR value
(gdb) show env VAR
```

Breakpoints and control

```
(gdb) break main
(gdb) break file.c:123
(gdb) break *0x401000
(gdb) info breakpoints
(gdb) delete 1
(gdb) disable 1
(gdb) enable 1
(gdb) continue
(gdb) next
(gdb) step
(gdb) stepi
(gdb) nexti
(gdb) finish
```

Disassembly

For x86-64 Intel-style output:

```
(gdb) set disassembly-flavor intel
(gdb) disassemble
(gdb) disassemble /m main
(gdb) x/20i $rip
```

Registers

```
(gdb) info registers
(gdb) info registers rax rbx rcx rdx
(gdb) print/x $rax
(gdb) set $rax = 0x1234
```

Stack and call frames

```
(gdb) backtrace
(gdb) bt full
(gdb) frame 2
(gdb) info frame
(gdb) info locals
(gdb) info args
```

Memory examination

```
(gdb) x/16gx $rsp
(gdb) x/32xb 0x7fffffffe000
(gdb) x/s $rdi
(gdb) x/4wx &some_global
```

Format legend:

- `x/NFMT ADDR`
- `N` count, `F` format (`x`,`d`,`u`,`o`,`t`,`f`,`s`,`i`), `M` size (`b`,`h`,`w`,`g`)

Watchpoints

```
(gdb) watch var
(gdb) rwatch var
(gdb) awatch var
(gdb) info watchpoints
```

Threads

```
(gdb) info threads
```

```
(gdb) thread 2
(gdb) thread apply all bt
```

Core dumps

```
ulimit -c unlimited
./app
gdb ./app core
(gdb) bt
(gdb) info registers
```

Attach to a running process

```
(gdb) attach 12345
(gdb) detach
```

Syscall-focused debugging (Linux)

Useful when stepping syscall stubs or verifying ABI register usage:

```
(gdb) catch syscall write
(gdb) catch syscall exit
(gdb) info registers rax rdi rsi rdx r10 r8 r9
```

43.3 LLDB essentials (Linux/macOS style, minimal)

```
lldb ./app
(lldb) run
(lldb) breakpoint set --name main
(lldb) breakpoint set --file file.c --line 123
```

```
(lldb) register read
(lldb) disassemble --frame
(lldb) thread backtrace
(lldb) memory read --format x --size 8 --count 16 $sp
```

43.4 WinDbg / CDB essentials (Windows)

WinDbg is the most assembly-friendly debugger on Windows. The following commands work in WinDbg and cdb.

Starting and attaching

```
windbg.exe app.exe
cdb.exe app.exe

/* attach */
windbg.exe -p 1234
cdb.exe -p 1234
```

Execution control

```
g       /* go */
p       /* step over */
t       /* step into */
gu      /* go up (finish) */
gc      /* go from exception */
```

Breakpoints

```
bp main
```

```
bp 00007ff6`12341000
bl              /* list breakpoints */
bd 0            /* disable breakpoint 0 */
be 0            /* enable breakpoint 0 */
bc 0            /* clear breakpoint 0 */
```

Registers

```
r               /* show registers */
r rip
r rax=1234
r rcx rdx r8 r9
```

Windows x64 calling convention reminder for debugging:

- args: `RCX, RDX, R8, R9`
- stack has 32-byte shadow space reserved by caller

Stack and backtrace

```
k               /* stack trace */
kp              /* stack trace with parameters */
kv              /* verbose trace */
.frame 1        /* select frame */
```

Disassembly

```
u rip
u 00007ff6`12341000 L40
uf main         /* disassemble full function */
```

Memory inspection

```
dq rsp L20      /* display quadwords */
dd rsp L40      /* display dwords */
db rsp L80      /* display bytes */
da rcx          /* display ASCII string at rcx */
du rcx          /* display Unicode string at rcx */
```

Symbols

If symbols are not loading correctly:

```
.symfix
.reload
x module!*pattern*
lm              /* list modules */
```

Exceptions and first-chance breaks

```
sxe av          /* break on access violation */
sxe ibp         /* break on int3/breakpoint exceptions */
sxd av          /* disable breaking on access violation */
```

Viewing function arguments (Windows x64)

At function entry, first four args typically appear in:

- `RCX, RDX, R8, R9`
- additional args on stack above shadow space

```
r rcx rdx r8 r9
dq rsp L10
```

43.5 Visual Studio debugger essentials (Windows)

When stepping assembly or mixed Rust/C/C++ in Visual Studio:

- Use the Disassembly window to see machine instructions.
- Use Registers window to view GPR and XMM registers.
- Use Memory window with an address expression, e.g. `&var` or `0x...`

Practical toggles:

- "Just My Code": disable to step into runtime/foreign code.
- "Enable address-level debugging": improves disassembly stepping.

43.6 Cross-architecture register display patterns

GDB: x86-64

```
(gdb) info registers
(gdb) info registers rax rbx rcx rdx rsi rdi rbp rsp rip
(gdb) x/16i $rip
(gdb) x/16gx $rsp
```

GDB: AArch64

```
(gdb) info registers
(gdb) info registers x0 x1 x2 x3 x4 x5 x8 sp pc
(gdb) x/16i $pc
(gdb) x/16gx $sp
```

GDB: RISC-V

```
(gdb) info registers
(gdb) info registers a0 a1 a2 a3 a4 a5 a7 sp ra pc
(gdb) x/16i $pc
(gdb) x/16gx $sp
```

43.7 Minimal Rust debugging on Windows: practical workflow

This section is Windows-only, per your constraint for Rust code and data.

Rust example program (calls Win32 output)

```
#![cfg(windows)]

use core::ffi::c_void;

type HANDLE = *mut c_void;
type DWORD = u32;
type BOOL = i32;

const STD_OUTPUT_HANDLE: DWORD = 0xFFFF_FFF5; /* (DWORD)-11 */

#[link(name = "kernel32")]
extern "system" {
    fn GetStdHandle(nStdHandle: DWORD) -> HANDLE;

    fn WriteFile(
        hFile: HANDLE,
```

```
        lpBuffer: *const c_void,
        nNumberOfBytesToWrite: DWORD,
        lpNumberOfBytesWritten: *mut DWORD,
        lpOverlapped: *mut c_void,
    ) -> BOOL;
}

fn main() {
    let msg = b"Rust debug example: Win32 WriteFile\r\n";
    unsafe {
        let h = GetStdHandle(STD_OUTPUT_HANDLE);
        let mut written: DWORD = 0;
        let _ = WriteFile(
            h,
            msg.as_ptr() as *const c_void,
            msg.len() as DWORD,
            &mut written as *mut DWORD,
            core::ptr::null_mut(),
        );
    }
}
```

Build with symbols (Cargo)

```
cargo build
```

WinDbg quick session

```
windbg.exe target\debug\yourapp.exe
bp yourapp!main
```

```
g
u rip
r rcx rdx r8 r9
k
```

Rust name mangling can make symbol names less direct. For deterministic breakpoints in mixed-language debugging, prefer `#[no_mangle]` on exported functions in integration layers.

Rust: creating a stable exported symbol to break on

```rust
#![cfg(windows)]

#[no_mangle]
pub extern "C" fn stable_entry(x: i32) -> i32 {
    x + 1
}

fn main() {
    let y = stable_entry(41);
    println!("{}", y);
}
```

43.8 Engineer checklists

When debugging ABI and calling convention bugs

- Break at function entry.
- Inspect argument registers immediately.
- Inspect the stack around SP/RSP to confirm alignment and any shadow space (Windows).

- Single-step the prologue and confirm saved registers match the ABI.
- Confirm return value register before `ret`.

When debugging crashes in assembly

- Dump registers and SP.
- Disassemble around the instruction pointer.
- Inspect memory at the faulting address (if available).
- Verify pointer arguments and alignment constraints.
- Walk the stack trace; if the trace is corrupted, verify your prologue/epilogue and unwind correctness.

43.9 Chapter summary

- GDB is the primary Linux debugger for assembly and ABI work; master breakpoints, disassembly, registers, and memory examination.
- WinDbg provides the most direct view into Windows x64 execution, arguments, stack layout, and exceptions.
- Cross-architecture debugging becomes routine when you standardize a workflow: break, read args, inspect SP, disassemble, step, verify return.
- For Rust on Windows, build with symbols and consider `#[no_mangle]` exported helpers for stable breakpoints in integration layers.

Professional Assembly Style Guide

This chapter defines a professional, maintainable assembly style that scales from short leaf routines to multi-file, ABI-stable, production-grade low-level components.
Goals:

- correctness first (ABI, alignment, clobbers, ordering),
- readability under pressure (debugging, incident response, reverse engineering),
- portability discipline (clear ISA/ABI separation),
- tool-friendly structure (objdump, debuggers, linkers, LTO boundaries),
- security hygiene (no accidental undefined behavior at the boundary).

The rules below are written to be directly applicable to **GAS Intel syntax** on x86-64 and to remain conceptually consistent across AArch64 and RISC-V chapters (even though their syntax differs).

44.1 Core principles

1) One routine, one contract

Every routine must declare its contract clearly:

- ABI and platform (SysV vs Windows x64),
- argument registers and types,
- return value register and meaning,
- clobbered registers and preserved registers,
- memory side effects (reads/writes, atomicity, ordering).

Write the contract as a block comment at the top of the function.

2) Use named constants, not magic numbers

Prefer:

- `.equ` for constants,
- symbolic labels for data and jump targets,
- computed lengths (label arithmetic) instead of hardcoding sizes.

3) Make alignment visible and intentional

Always document stack alignment requirements:

- SysV x86-64: 16-byte aligned at call boundary, red zone allowed.
- Windows x64: 16-byte aligned at call boundary, 32-byte shadow space, no red zone.

4) Prefer leaf functions when possible

Leaf functions reduce risk:

- no need to preserve return address beyond `ret` semantics,
- less stack manipulation,
- fewer clobbers and simpler verification.

5) Keep the hot path linear

Most performance-critical assembly should:

- minimize unpredictable branches,
- avoid excessive register pressure,
- keep memory access patterns explicit.

44.2 File layout conventions (GAS Intel syntax)

Recommended skeleton

```
.intel_syntax noprefix

/* ------------------------------------------------------------------
   File: foo.S
   ISA:  x86-64
   Syntax: GAS Intel
   ABI: SysV or Windows x64 (state explicitly)
   ------------------------------------------------------------------ */
```

```
    /* Constants */
    .equ SOME_CONST, 123

    /* Read-only data */
    .section .rodata
msg:
    .ascii "example"
msg_end:
    .equ msg_len, msg_end - msg

    /* Writable data (if needed) */
    .section .data
align_var:
    .quad 0

    /* Uninitialized storage */
    .section .bss
    .lcomm scratch, 64

    /* Code */
    .section .text
```

Rules:

- Put `.intel_syntax noprefix` once at top.
- Use `.section .rodata` for constants/strings.
- Use `.bss` for large scratch buffers; avoid implicit stack usage when you need stable addresses.

- Keep `.text` code grouped by exported API.

44.3 Function header template (required in professional code)

Use a consistent header. Example for SysV x86-64:

```
/* ---------------------------------------------------------------------
   uint32_t fetch_add_u32(uint32_t* p, uint32_t delta)

   ABI: x86-64 System V
   Args:
     p     -> RDI
     delta -> ESI
   Return:
     old value -> EAX

   Clobbers:
     RAX, flags

   Memory:
     atomic RMW on *p
     ordering: strong (LOCKed instruction)

   Notes:
     - p must be 4-byte aligned for best performance
     - delta is treated as 32-bit
   --------------------------------------------------------------------- */
```

For Windows x64, explicitly include shadow space requirements if your routine calls other functions.

44.4 Naming conventions

1) Exported symbols

- Use lowercase with underscores: `fetch_add_u32`.
- Keep exported symbols stable (ABI contract).
- If you must version: `fetch_add_u32_v1`.

2) Local labels

Use:

- descriptive local labels for clarity: `retry`, `done`, `slowpath`
- or numeric local labels only for tiny loops: `1: ... jnz 1b`

3) Constant labels

Use `.LC_NAME` convention for local constants.

44.5 Indentation and formatting

Recommended column layout

- 4 spaces indentation before instructions.
- Align operands for readability.
- Comments on the right, but do not exceed a readable line width.

Example

```
    mov     eax, esi          /* eax = delta */
    lock xadd DWORD PTR [rdi], eax  /* atomic fetch-add */
    ret
```

44.6 Comments: rules and quality

Assembly must be readable without constantly consulting ABI PDFs. Use comments to explain:

- why the code is correct, not what the instruction does,
- ABI decisions (why a register is used),
- alignment and ordering constraints,
- invariants before and after a block.

Avoid noisy comments like “move rax to rbx” unless the move is conceptually important.

44.7 Register discipline

1) Never guess ABI-preserved registers

If you modify a callee-saved register, save and restore it or avoid using it.

2) Use scratch registers consistently

Define a convention inside your file:

- `rax` = return / accumulator

- `r10,r11` = scratch temporaries
- do not touch `rbx,r12–r15` unless needed

3) Be explicit about clobbers

Your function header must list clobbered registers and flags.

44.8 Stack discipline

SysV x86-64

- Keep 16-byte alignment at call boundaries.
- Red zone is allowed for leaf functions, but avoid it in code that may run under unusual environments or instrumentation.

Windows x64

- Always reserve 32 bytes shadow space before calling other functions.
- Keep stack 16-byte aligned at call boundaries.
- No red zone. Never write below `RSP`.

Prologue/epilogue patterns

Minimal SysV leaf (no stack):

```
.globl add2
add2:
    lea     eax, [edi !+! esi]
```

```
    ret
```

Windows x64 function that calls an API (shadow space):

```
.intel_syntax noprefix
.globl call_api
call_api:
    sub     rsp, 40           /* 32 shadow + 8 align */
    /* ... set RCX/RDX/R8/R9 ... */
    call    SomeApi
    add     rsp, 40
    ret
```

44.9 Data movement rules

1) Prefer RIP-relative addressing (x86-64)

```
    lea     rax, [rip !+! msg]  /* position-independent */
```

2) Prefer `movzx`/`movsx` over implicit extension assumptions

Make sign/zero extension explicit when moving from smaller widths.

44.10 Control flow rules

1) Structure loops clearly

Use one entry, one exit when possible.

2) Use forward branches for error handling

Keep the hot path straight-line. Move checks to the front and branch to a cold label.

```
    test    rdi, rdi
    jz      .bad_ptr          /* cold */
    /* hot path continues */
```

44.11 Atomic and concurrency style

When writing atomics:

- Document the intended ordering (Relaxed/Acquire/Release/AcqRel/SeqCst).
- On x86, prefer `lock xadd` or `lock cmpxchg` as primitives.
- On ARM, use `ldaxr/stlxr` for AcqRel-shaped loops.
- On RISC-V, use `amo*.aq/.rl` or fences appropriately.

Example: x86-64 atomic fetch-add

```
.intel_syntax noprefix
.text
.globl fetch_add_u32
fetch_add_u32:
    mov     eax, esi
    lock xadd DWORD PTR [rdi], eax
    ret
```

44.12 Error handling and return conventions

Professional assembly should define:

- how errors are reported (return codes, errno-style, out-params),
- whether negative returns are used,
- which registers are left untouched on failure vs success.

If you mirror C conventions, keep it explicit and consistent.

44.13 Interoperability rules (C/C++/Rust)

C/C++ boundary

- Use `extern` declarations on the C side.
- Keep signatures simple for stable ABI: fixed-width integers and pointers.
- Avoid passing structs by value until you master ABI lowering details.

Rust on Windows boundary (booklet constraint)

You requested all Rust content be Windows-only. Use:

- `extern "C"` for a C ABI boundary,
- `extern "system"` for Win32 APIs,
- `#[no_mangle]` for stable exported names.

Rust: style for stable exported low-level functions

```
#![cfg(windows)]

use core::ffi::c_void;

#[no_mangle]
pub extern "C" fn add_i32(a: i32, b: i32) -> i32 {
    a.wrapping_add(b)
}

#[no_mangle]
pub extern "C" fn store_sum(outp: *mut i32, a: i32, b: i32) -> i32 {
    unsafe {
        if outp.is_null() {
            return -1;
        }
        *outp = a.wrapping_add(b);
    }
    0
}

#[link(name = "kernel32")]
extern "system" {
    fn GetStdHandle(nStdHandle: u32) -> *mut c_void;
}

fn main() {
    let mut x = 0;
```

```
    let _ = store_sum(&mut x as *mut i32, 10, 32);
    let y = add_i32(20, 22);
    unsafe { let _h = GetStdHandle(0xFFFF_FFF5); }
    println!("x={}, y={}", x, y);
}
```

44.14 Documentation and verification checklist

Pre-commit checklist for assembly files

- Function headers include ABI, arg regs, return regs, clobbers, ordering notes.
- Stack alignment is correct at every call site.
- Windows x64 shadow space is present for every call.
- Callee-saved registers are preserved if used.
- No writes below RSP on Windows.
- All constants are named; no unexplained magic numbers.
- Disassembly inspection: verify prologue/epilogue and calling convention lowering.

Minimal inspection commands

```
/* Linux */
objdump -d -Mintel your.o
readelf -Ws your.o

/* Windows (MinGW) */
objdump -d -Mintel your.obj
```

```
objdump -p your.exe
```

44.15 Chapter summary

- Professional assembly is a disciplined contract: ABI correctness, explicit clobbers, and visible alignment.
- Readability comes from consistent structure, naming, and comments that explain intent and invariants.
- Interoperability requires stable, simple interfaces and careful calling convention compliance.
- A small set of rules, applied consistently, prevents most assembly defects before they ship.